USB 2.0 设备的设计与开发

边海龙　贾少华　　编著
博嘉科技　　审校

人 民 邮 电 出 版 社

图书在版编目（CIP）数据

USB 2.0 设备的设计与开发/边海龙，贾少华编著.—北京：人民邮电出版社，2004.1
ISBN 7-115-11731-4

Ⅰ.U… Ⅱ.①边… ②贾… Ⅲ.电子计算机—接口—程序设计 Ⅳ.TP334
中国版本图书馆 CIP 数据核字（2003）第 109598 号

内 容 提 要

USB 已经成为计算机上的标准配置接口，是实现外部设备与计算机通信常用的一种方式。本书从两个方面入手，重点介绍了 USB 2.0 协议以及 USB 设备的设计与开发的相关知识。全书共分为 12 章，第 1~8 章介绍了 USB 2.0 协议，内容包括 USB 2.0 规范、USB 集线器、设备检测、控制传输、USB 传输方式、USB 设备机械和电气特性、USB 中数据格式和信号编码等基础知识；第 9~12 章通过具体的实例介绍了如何开发一个符合 USB 2.0 规范的设备，内容包括常见 USB 控制器芯片介绍、USB 设备硬件设计、固体设计、驱动程序设计。全书列举了大量范例程序，并作了详尽解释，可以帮助用户开发自己的 USB 设备。

本书语言通俗易懂，内容丰富详实，突出了以实例为中心的特点，适合具有一定的专业基础知识和 USB 设计开发经验的读者学习和参考，可作为计算机专业高年级本科生和研究生的教材。

USB 2.0 设备的设计与开发

◆ 编　　著　边海龙　贾少华
审　　校　博嘉科技
责任编辑　马　嘉

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
读者热线：　010-67132692
北京汉魂图文设计有限公司制作
北京　　印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092　1/16
印张：16.5
字数：393 千字　　2003 年 9 月第 1 版
印数：1-　000 册　　2003 年 9 月北京第 1 次印刷

ISBN 7-115-11731-4/TP・3652

定价：28.00 元

前　言

随着计算机在各个领域的广泛应用，计算机在软硬件资源方面存在的不足已经成为制约其进一步发展的“瓶颈”。因此，不论是计算机的设计者还是使用者都在努力寻找解决方法。

1998 年 USB 接口出现，USB 是通用串行总线（Universal Serial Bus）的英文简称，它是目前计算机上广泛使用的外部设备接口。设计者们以最大限度地方便用户使用为设计目标，正是由于其低廉的价格、方便灵活的使用方式，USB 一推出便受到了广泛的关注，很多的使用者和开发者都对其进一步的发展表现出了极大的兴趣。

与其他的接口不同，USB 接口规范是完全开放的。因此，它获得了前所未有的生命力。从诞生之初到现在，短短 5 年的时间，由开始的不为人们所熟悉，到现在几乎所有的新型计算机硬件都将其作为新一代的接口；传输速率由开始的 1.1 版本的 12Mbit/s 到现在 2.0 版本的 480Mbit/s，发展速度令人惊讶。

为用户提供方便的前提是开发者必须最大限度地将产品的方方面面考虑周全，那么，怎么样才能够在利用最短时间的前提下，将开发速度提到最高呢？这便是本书所要讲述的问题。

本书主要内容

本书主要介绍 USB 2.0 规范，并列举大量实例以帮助用户开发一个自己的 USB 设备。全书共分 12 章，各章大体内容如下。

第 1 章：讲述 USB 出现的背景、发展情况以及一些 USB 规范的基本概念，使读者对 USB 有一个基本的认识。

第 2 章：介绍 USB 总线结构，以及开发 USB 外设所要做的准备工作。

第 3 章：详细介绍 USB 集线器的体系架构，以及在即插即用中所起的作用。

第 4 章：以 Cypress 公司的 EZ-USB FX2 系列芯片为例讲述 USB 设备的列举过程，使读者了解主机如何识别 USB 设备并与设备进行通信，详细介绍 USB 2.0 规范定义的 9 种描述符。

第 5 章：详细讲述 USB 数据传输的核心——控制传输，其中包括设置阶段、数据阶段和状态阶段 3 个阶段，以及 11 种标准设备请求和供应商特定请求。

第 6 章：从 USB 通信流的角度来讲述 USB 的 4 种数据传输方式：控制传输、批量传输、等时传输和中断传输。

第 7 章：按照高速、全速和低速 3 种不同规范来讲述 USB 的机械和电气特性。

第 8 章：介绍了 USB 规范的数据格式与信号编码。

第 9 章：首先讲述通用 USB 控制芯片的构成组件，然后列举市面上常见的 USB 控制器芯片，并在功能上对其进行比较，以方便读者选取合适的 USB 控制芯片。

第 10 章：主要讲述开发一个 USB 设备的所必须的准备工作，芯片如何选择，以及 USB 外设的硬件电路设计。

第 11 章：主要以 Cypress 公司的 USB 控制芯片 CY7C68013 为例，讲述如何进行 USB 设备的固件设计，同时对所使用的开发语言 C51 进行了简单的介绍。本章列举了大量的固件范例程序，这些程序都是作者在实际开发过程中编写的，因此对读者学习有较大的指导作用。

第 12 章：介绍 USB 设备驱动程序的开发流程，包括 USB 设备驱动程序的基础，编写、

编译和调试设备驱动程序的步骤，详细分析了设备驱动程序的安装文件 INF 文件。

读者对象

本书语言通俗易懂，内容丰富详实，突出了以实例为中心的特点，适合具有一定的专业基础知识和 USB 设计开发经验的读者学习和参考，可作为计算机专业高年级本科生和研究生的教材。

编写分工

本书由边海龙和贾少华担任主要的编写工作，博嘉科技资讯有限公司审校。其中，第 1 章～第 4 章、第 7 章～第 8 章由边海龙编写，第 5 章～第 6 章、第 9 章～第 12 章由贾少华编写。参加本书编写工作的还有金洁、谭细金、袁海英、吴杲、刘治国、李祖裕、庞作田、赵严、蔡勇、杨建军、朱怀宇、葛青、赵继军等，在此一并表示感谢。

相关网站和论坛

笔者在编写本书时参考了以下网站和论坛上的相关资料：

USB 开发者论坛：http://www.usb.org

USB 控制器芯片供应商：

Cypress　http://www.cypress.com

National Semiconductor　http://www.national.com

NetChip　http://www.netchip.com

Philips　http://www.semiconductors.philips.com

Microchip　http://www.microchip.com

USB 工具软件：

Usbready.exe　http://www.acwsoft.de/acw_usb2.html

Keil software　http://www.keil.com

Inside Out Networks　http://www.ionetworks.com

Windows 驱动程序参考：

Microsoft Windows 2000 Driver Development Kit.

By Microsoft Corporation

在此对以上网站和文献作者表示衷心的感谢，祝这些专业网站越办越好，祝文献作者在相关的研究领域取得更好的成绩和长足的进步。

技术支持

如果读者愿意参加“USB 2.0 设计与应用”的学习培训，或是在学习过程中发现问题，或有更好的建议，欢迎与我们联系，我们非常愿意与熟悉 USB 的高手经常保持联系。作者的 E-mail 邮箱如下：

bojiakeji@163.net。

由于编者水平有限，时间仓促，书中难免存在疏漏和错误，恳请读者批评指正。

作 者

目　录

第 1 章　USB 基础知识 1

1.1　快速的发展过程 1
1.2　USB 的优势 3
1.2.1　真正的即插即用 3
1.2.2　速度的提升 4
1.2.3　其他方面 4
1.3　重要的概念 5
1.3.1　“智能化”的接口简析 5
1.3.2　“主机”的概念 6
1.3.3　USB 的端口 8
1.3.4　“Function”的意义 8
1.3.5　“Hub”的意义 9
1.3.6　“Device”的意义 9
1.4　USB 的局限性 9
1.4.1　功能的局限性 9
1.4.2　开发的难度 10
1.5　本章小结 11

第 2 章　如何着手 USB 的开发工作 12

2.1　USB 结构简介 12
2.1.1　USB 的总线结构（物理结构） 12
2.1.2　USB 的逻辑结构 13
2.2　必要的准备工作 14
2.2.1　主机 14
2.2.2　开发过程中应考虑的问题 16
2.2.3　必要设备的准备 18
2.2.4　开发工作流程 18
2.2.5　最后的考虑 19
2.3　关于开发者论坛 20
2.4　本章小结 20

第 3 章　集线器 21

3.1　USB 集线器 21
3.1.1　通常概念意义下的集线器 21

3.1.2 USB 中的集线器（Hub）概念 22
3.2 Hub 的体系结构 22
3.2.1 连接的重续 23
3.2.2 Hub 的错误恢复机制 24
3.3 Hub 的数据帧与微帧的计时器 24
3.3.1 高速模式下微帧的计时范围 24
3.3.2 全速模式下帧的计时范围 25
3.3.3 帧/微帧的计时同步机制 25
3.4 主机在帧结束时的行为 29
3.4.1 全/低速模式下最晚发出的主机数据包 29
3.4.2 全/低速模式下的无效包 29
3.4.3 全/低速下对事务处理完成时间的预测 29
3.5 内部端口 30
3.6 下游端口 31
3.6.1 下游端口状态的描述 33
3.6.2 连接断开的侦测 35
3.6.3 端口指示灯 36
3.7 上游端口 36
3.7.1 全速 37
3.7.2 高速 37
3.7.3 接收器 37
3.7.4 发送器 37
3.8 集线器中继器 37
3.9 集线器控制器 38
3.9.1 端点的组织结构 38
3.9.2 Hub 的消息体系结构和操作 38
3.9.3 端口改变信息的处理 39
3.9.4 集线器和端口的状态改变位图 39
3.9.5 过流报告和恢复 40
3.9.6 对于设备检测的控制 41
3.10 集线器的设置 41
3.11 事务处理转译器（Transaction Translator） 42
3.11.1 综述 42
3.11.2 数据处理时序 44
3.12 本章小结 45

第 4 章 设备检测 46

4.1 概述 46
4.2 FX 2 的设备列举过程 47
4.2.1 FX 2 列举的特点 47

4.2.2 FX 2 的启动模式 48
4.2.3 FX 2 的“默认的 USB 设备” 48
4.2.4 EEPROM 启动导入数据的格式 49
4.2.5 关于 RENUM 位 51
4.2.6 FX 2 对设备请求的回应（RENUM=0） 51
4.2.7 FX 2 中用于固件下载的制造商请求 52
4.2.8 重新列举（ReNumerates） 53
4.2.9 初始化下载的过程 54
4.3 USB 2.0 中对于列举的规定 54
4.3.1 列举过程设备经历的状态 54
4.3.2 总线列举要经历的步骤 56
4.3.3 总线列举过程中要用到的描述符 56
4.4 本章小结 64
第 5 章 控制传输 66
5.1 基本理论 66
5.1.1 设置阶段 67
5.1.2 数据阶段 72
5.1.3 状态阶段 73
5.1.4 错误处理 74
5.1.5 11 种标准请求 75
5.1.6 类特定请求 83
5.1.7 供应商特定请求 84
5.2 实际应用 84
5.2.1 简介 84
5.2.2 控制端点 EP0 84
5.2.3 USB 请求 87
5.3 本章小结 102
第 6 章 数据传输方式 103
6.1 控制传输 104
6.1.1 数据格式 104
6.1.2 包大小的限制 106
6.1.3 总线访问限制 107
6.1.4 控制传输的数据顺序和错误的检测处理 109
6.2 批量传输 109
6.2.1 数据格式 110
6.2.2 数据包大小的限制 111
6.2.3 总线访问限制 111
6.2.4 数据顺序和错误检测 112

6.3 中断传输 112
6.3.1 数据格式 113
6.3.2 包大小限制 113
6.3.3 总线访问限制 113
6.3.4 数据顺序和错误检测 115
6.4 等时传输 115
6.4.1 数据格式 116
6.4.2 数据包大小限制 117
6.4.3 总线访问限制 117
6.4.4 错误检测 118
6.5 本章小结 118
第 7 章 机械特性 119
7.1 综述 119
7.2 内建的连接器的协议 119
7.3 线缆 120
7.4 线缆组件 120
7.4.1 标准的可分离的线缆组合 120
7.4.2 高/全速的束缚型的线缆组合 122
7.4.3 低速的束缚型线缆组合 123
7.4.4 被禁止的线缆组合 123
7.5 USB 连接器的终端数据 124
7.6 线缆的机械构造和材料需求 124
7.7 关于 USB 的电气特性 125
7.8 信号 125
7.8.1 高速信号的概述 125
7.8.2 USB 驱动器特性 127
7.8.3 高速接收器的特性 128
7.9 设备速度的检测 128
7.9.1 全/低速设备速度的检测 128
7.9.2 高速设备的检测 129
7.10 输入特性 129
7.10.1 全/低速的输入特性 129
7.10.2 高速的输入特性 130
7.11 信号的层次 131
7.11.1 全/低速的信号的层次 131
7.11.2 高/全速的信号的层次 132
7.12 连接和断开连接的信号 133
7.12.1 连接建立和断开的检测 133
7.12.2 建立连接的事件时序 134

7.12.3 数据信号 …… 135
7.12.4 重启信号 …… 135
7.12.5 挂起 …… 137
7.12.6 重读 …… 137
7.13 数据信号的速率 …… 137
7.14 电力的分配 …… 138
7.14.1 设备的分类 …… 138
7.14.2 在挂起/重续中电力的控制 …… 138
7.14.3 动态地连入和拔除 …… 138
7.15 本章小结 …… 139

第 8 章 信号编码与传输 …… 140

8.1 字节/位顺序 …… 140
8.2 SYNC 域 …… 140
8.3 数据包域的格式 …… 141
8.3.1 数据包鉴定域 …… 141
8.3.2 地址域 …… 142
8.3.3 帧数量域 …… 143
8.3.4 数据域 …… 143
8.3.5 循环冗余校验 …… 143
8.4 数据包格式 …… 143
8.4.1 标志包 …… 143
8.4.2 分割处理特殊标志包 …… 144
8.4.3 帧开始（Start-Of-Frame）包 …… 147
8.4.4 数据包 …… 148
8.4.5 握手包 …… 148
8.4.6 握手的响应 …… 149
8.5 数据包的处理时序 …… 150
8.5.1 批量传输的处理时序 …… 152
8.5.2 控制传输处理时序 …… 152
8.5.3 中断传输的处理时序 …… 155
8.5.4 同步传输的处理时序 …… 156
8.6 数据触发的同步和重试 …… 156
8.6.1 通过 SETUP 标志进行的初始化 …… 156
8.6.2 成功的数据处理 …… 157
8.6.3 数据的失效或不能被接收 …… 157
8.6.4 失效的握手信号 …… 158
8.6.5 低速的事务处理 …… 158
8.7 错误检测和恢复 …… 159
8.7.1 数据包错误分类 …… 160

8.7.2 总线循环时间 160
8.7.3 EOP 的失效 161
8.7.4 总线错误的恢复 161
8.8 本章小结 162

第 9 章 USB 控制器芯片 163

9.1 USB 控制器芯片的构成 163
9.1.1 USB 端口 163
9.1.2 CPU 165
9.1.3 数据缓冲器 165
9.1.4 程序存储器 165
9.1.5 数据存储器 166
9.1.6 寄存器 166
9.1.7 其他接口 167
9.2 芯片构架 167
9.2.1 专门为 USB 设计的 USB 控制芯片 167
9.2.2 内嵌通用微控制器的芯片 169
9.2.3 需要外接微控制器的芯片 170
9.3 芯片举例 171
9.3.1 Cypress 公司的 M8 CY7C63101A 芯片 171
9.3.2 Cypress 公司的 EZ-USB 芯片 172
9.3.3 National Semiconductor USBN9603 176
9.3.4 Netchip NET2888 180
9.3.5 Philips Semiconductor PDIUSBD12 182
9.4 本章小结 185

第 10 章 USB 设备开发概述 186

10.1 准备工作 186
10.2 开发步骤 187
10.2.1 初步计划 187
10.2.2 硬件计划 187
10.2.3 软件计划 188
10.3 控制器芯片的选择 188
10.4 硬件设计 190
10.5 本章小结 192

第 11 章 固件设计（CY7C68013） 193

11.1 固件的工作 193
11.2 汇编与 C51 的比较 194
11.3 C51 程序设计基础 195

11.3.1 标志符和关键字 195
11.3.2 数据类型 196
11.3.3 中断服务函数和寄存器组定义 197
11.3.4 C51 中的寻址方式 198
11.4 固件程序设计 199
11.4.1 固件程序规划 199
11.4.2 主程序 201
11.4.3 描述符定义 203
11.4.4 响应设备请求 208
11.4.5 中断处理 211
11.4.6 数据传输举例：批量 OUT 传输 214
11.4.7 数据传输举例：批量 IN 传输 215
11.5 本章小结 216

第 12 章 驱动程序设计 217

12.1 设备驱动程序基础 217
12.2 驱动程序的分类 219
12.2.1 VxD 219
12.2.2 KMD 220
12.2.3 WDM 220
12.3 WDM 驱动程序基本结构 220
12.3.1 WDM 驱动程序的层次结构 220
12.3.2 设备对象 222
12.3.3 USB 驱动程序结构 223
12.4 USB 设备驱动程序开发流程 224
12.4.1 准备工作（工具选择） 224
12.4.2 两个重要的概念 IRP 和 URB 226
12.4.3 编写驱动程序 229
12.4.4 编译驱动程序 236
12.4.5 安装驱动程序 239
12.5 INF 文件 241
12.5.1 INF 文件格式要求 241
12.5.2 INF 文件举例 242
12.5.3 INF 文件详解 244
12.5.4 用 geninf 实用程序生成一个 INF 文件 249
12.6 本章小结 250

第 1 章　USB 基础知识

知识点：

- USB 的发展过程
- USB 的协议
- 常见的重要概念
- USB 的局限性

本章导读：

本章从 USB 的发展过程开始，向读者介绍了 USB 接口技术的发展过程、USB 的特点，以及读者在学习 USB 的过程中应该注意的一些概念。

随着计算机微处理器芯片性能的飞速发展，计算机逐渐在各种领域承担起各种各样的复杂任务，伴随着这种广泛应用，随之而来的问题就是计算机本身软硬件资源的严重不足。

软件方面包括操作系统对中断以及 I/O 口的分配，硬件方面则是用户必须面对如何使用有限的主板插槽来合理地接口必要的适配器，而且最大的不便就是在每一次插入或拔除板卡时，都不得不重复执行关闭机器、插入板卡、启动机器、安装驱动、进行调试等一系列繁杂的步骤。在许多场合，敞开的机箱、散乱地摊在工作台上的各种板卡成了专业计算机工作室的标志。

那么，有没有一种简单易行的接口既能够最大限度地节省计算机的软硬件资源，又能方便使用呢？答案是肯定的，那就是日益被用户所熟悉的 USB 接口。

1.1　快速的发展过程

在我们日益被上述所提到的问题所困扰时，开发商们同样也在努力地寻找新的途径来解决问题。

1. 萌芽

新生事物的萌芽可能原于少数人的灵感与努力，但其发展则必然需要很多外界力量的支持，USB 接口技术的发展也不例外。

Philips 与数字设备公司（Digital Equipment Corporation）借鉴 IIC synchronics bus 的优点，联合制定出了 Access.bus 规范。

其思路是让计算机的低速外围设备，如键盘、鼠标等，以简单的线缆插入式的连接方法来进行低速（100bit/s）的工作。同时，它将各种设备进行分类，如键盘、指示设备、文字设

备、显示设备等。

之所以说 Access.bus 是 USB 的基础，是因为其中的机械电气结构以及与主机连接的方式使开发商们眼界为之一开。它与主机的连接只通过 4 条线缆，即电源线、地线、一条数据线（Data Wire）、一条脉冲线（Clock Wire）。同时，它使用了开放收集器（Open-cellector）驱动，这就形成了 USB 的发展雏形。

2. 成长

经过长时期的酝酿，1996 年 1 月，在 Compaq、Intel、Microsoft、NEC 等 4 家公司的联合努力下，USB 1.0 的白皮书问世了，这是 USB 发展史上具有里程碑意义的一页。

随后，经过近两年的完善与修改工作，一个完整的、可行的 USB 1.1 规范于 1998 年 9 月完成。至此，许多开发商已经可以依据 USB 1.1 规范内容来进行相关产品的开发了。

令人欣喜的是，这 4 家公司同意任何人都可免费使用 USB 1.1 的白皮书版本。这与其他组织开发的标准相比无疑是一个明智的创举，也正是基于此，使得 USB 产品的开发在短暂的时间内获得了迅猛的发展。USB 规范的发展历程如表 1-1 所示。

表 1-1　　USB 规范的版本

版本	发表日期	说明
0.7	1994 年 11 月 11 日	覆盖 0.6 版本
0.8	1994 年 12 月 30 日	修改第 3~8、10、11 章，新增附录
0.9	1995 年 4 月 13 日	修改所有章节
0.99	1995 年 8 月 25 日	修改所有章节
1.0FDR	1995 年 11 月 13 日	修改第 1、2、5~11 章
1.0	1996 年 1 月 15 日	修改第 5~11 章
1.1	1996 年 9 月 23 日	修改所有章节
2.0(draft 0.79)	1999 年 10 月 5 日	修改第 5、7~9、11 章，以增加高速协议部分
2.0(draft 0.9)	1999 年 12 月 21 日	修改所有章节
2.0	2000 年 4 月 27 日	高速模式的版本

3. 对抗性的竞争促进了其发展

没有竞争就没有发展，USB 在诞生之初，便面对着许多已趋成熟的计算机接口的挑战，这就要求它必须具有明显的优势，并不断完善，才可能被用户所接受。USB 及其他常用接口如表 1-2 所示。

表 1-2　　USB 及其他常用接口

接口	格式	设备最大数目	最大线缆长度（英尺）	最大传输速率（bit/s）
USB	异步串行	127	16（利用 5 个集成器可连接到 96）	1.5M~2M（1.0~1.1 版） 480M（2.0 版）
RS-232 (EIA/TIA-232)	异步 串行	2	50~100	20k（有的硬件支持 45k）

续表

接口	格式	设备最大数目	最大线缆长度（英尺）	最大传输速率（bit/s）
PS-485	异步	32（有的硬件支持 256 个）	4000	10M
(TEA/EIA-485)	串行			
IrDA	红外异步串行	2	6	115k
Microwire	同步串行	8	10	2M
SPI	同步串行	8	10	2.1M
IIC	同步串行	40	18	3.4M
IEEE-1394 (Fire-wire)	串行	64	15	400M（IEEE-1394.b 可达到 3.2G）
IEEE-488(GPIB)	并行	15	60	8M
Ethernet	串行	1024	1600	10M/100M/1G
MIDI	串行电流循环	2	50	31.5k
并行打印机	并行	2（daisy-chain 支持 8 个）	10~30	8M

值得一提的是 IEEE-1394 接口与 USB 一样，其设计的最初目的便是方便微机用户简捷地连接外围设备到主机上，因此，在目前很多微型计算机上也携带有 IEEE-1394 接口。正是由于 1394 在速度上的优势，才使得制定 USB 标准的 7 家公司（Compaq、Intel、Microsoft、NEC、Hewlett-Packard、Lucent、Philips）又转入高速 USB 版本的开发，以对抗 1394 接口。不过，在 USB 2.0 推出之时，IEEE-1394 也推出了相应的 IEEE-1394.b 标准，将速率提高到了 3.2Gbit/s。因此，在这样的竞争下，USB 接口仍然需要向前不断发展。

应该看到，许多新生事物在其产生之初都有不尽完善之处，USB 也一样，所以，USB 2.0 仍然在不断推出其修订手册，我们接触到的最新的补丁发布于 2002 年 4 月。

1.2　USB 的优势

面对众多的接口，用户当然要问：为什么我应该选择 USB 而不是其他的接口呢？当看完下面讲述的内容之后，相信读者一定会被它所具有的众多优点所打动。

1.2.1　真正的即插即用

1. 自动地检测与设置

当启动计算机后需要再连入一个硬件外围设备时，过去的办法只能是将计算机关闭，然后打开机箱，插入板卡，连接线缆，再启动计算机等一系列的操作步骤。然而现在有了 USB 接口，只需要将配有 USB 接口的外设插入相应的计算机机箱上的接口，剩下的事情便完全由其和主机完成。当然，这需要得到一定的操作系统的支持，在普遍的 Windows 用户中，要注

意的是必须在 Windows 98 及其以上版本中才会支持 USB 接口。其中，Windows NT 4 是不支持的，Windows 2000 中加入了对 USB 接口的支持，最新的 Windows XP 是基于 Windows 2000 的，所以也支持 USB。

需要注意的是在 Windows 98 中，大多数情况下在第一次将一个新的 USB 设备接入主机时，可能会需要相应的驱动程序。而在 Windows 2000 中，这一问题已得到很好的解决，因此，连这一步也可省去。用户可以直接将 USB 设备从主机上拔开，虽然操作系统往往会提示你这一举动会引起系统的不稳定。

2. 通用的接口

USB 在诞生之初，便以尽可能方便用户使用为目的。因此，其接口的通用性必然是其特点之一。同时，由于越来越多的用户对 USB 的认可，使得许多计算机设备制造商都在其产品中加入了对 USB 接口的支持。因此，在市场上可以方便地得到有 USB 接口的键盘、鼠标、光驱、硬盘、摄像头等一系列的外围设备。同时，这些设备的接口机械、电气特性都是一致的，因此，在计算机上可以方便地连入这些设备。

3. 系统资源的节省

正是由于不同的外设可以使用同一个 USB 接口，因此，操作系统不需为每种设备都配置不同的中断和 I/O 口，从而最大限度地节省了计算机系统资源。

同时，也是由于上述特点，用户不必再为不同硬件使用系统的中断和 I/O 口时产生的冲突而犯愁，也不必再去手动地协调其分配了。

4. 简易的电缆

基于 Access.bus 的构想，USB 同样只使用 4 根线缆便完成了其繁重的数据传输。它们分别是电源线（+5V）、地线、两条差分的数据线（D+、D-），这就使得我们所接触的 USB 接头相当小巧。

单独的一条 USB 线可支持 5m 的传输距离，利用集线器，可达 30m 的传输距离。

5. 不需要单独电源

由于 USB 接口中携带了电源线和地线，因此，它可直接从主机的接口或集线器上得到电源的供给。在中等电源供应的条件下，它完全可以满足设备的需求。

1.2.2 速度的提升

在对 USB 1.1 版本的长期实践与改进基础上推出了 USB 2.0，它在数据传输速率上有了一个飞跃，已经达到了 480Mbit/s 的理论速率。

在 USB 体系中，总共有 3 种数据传输速率：低速（Low Speed）1.5Mbit/s、全速（Full Speed）12Mbit/s、高速（High Speed）480Mbit/s。像许多接口一样，USB 接口是向下兼容的，也就是说，最新的高速版本与 USB 1.1 接口在机械电气等方面是兼容的。

1.2.3　其他方面

USB 接口除了在简易性和速率这两大方面具有了突出的优势之外，为了最大限度地发挥其作用，开发接口的厂商在一开始便尽力使其能有更多体贴的、周详的用户服务方案。

1. 价格的优势

与其他接口相比 USB 的接口在线缆和机械方面实现起来简单易行，这就使其在价格上具有较大的优势。

2. 性能的稳定

为了使 USB 接口在工作时具有最大可能的稳定性，USB 协议从多方面考虑了保障措施。

- 接口的初始化。在第一次将 USB 设备连入主机时，协议规定了要经过设备与主机的两次初始化链接，才能完成接口的接入过程。当然，这两次的链接是用户感觉不到的。
- 与主机进行通信的保障。这种保障来自于以下两个方面：

（1）数据传输协议的保证。在数据传输过程中，要使用严格的错误检测机制，一旦发现错误，要能够做到通知发送者，并进行重传。

（2）硬件设计的保证。在 USB 的发送器、接收器、线缆等硬件规范中，都有关于检测错误及减少干扰的规定。

3. 其他

在电源的管理、电气的连接等各个方面，USB 接口都具有相应的功能以保证其工作的可靠与稳定，这些内容将在后续章节中进行一一介绍。

1.3　重要的概念

在学习和研究 USB 接口的过程中，我们常常会被一些 USB 所特有的、与我们通常理解的概念相矛盾的特性所困扰，这也是下面我们所要解决的问题。

1.3.1　“智能化”的接口简析

USB 设备与主机的结构如图 1-1 所示。

由图可以看出，所谓的 USB 设备按严格意义上来分，都应将其划分为两大部分，即接口部分和设备部分。

- 接口部分：是设备与主机连接通信的指挥中心，它负责整个主机与设备的信息流的交换，而且与其他计算机板卡最大的不同是 USB 接口部分本身必须具有“智能”指挥中心——CPU。

这是因为普通的板卡都是直接与主机板相连的，因此，普通板卡的数据线、地址线及电源线等都是由主机板控制的，并无条件地接受主机的指令。USB 设备则有所不同，由于它与

主机惟一的通信便是两条差分的数据线，因此，所有的地址、数据、控制信号全部都是以差分信号的方式进行双向传输的。所以，必须在 USB 设备接口中加入“智能”控制中心，也就是 CPU 芯片，才能完成信息的分类与解释。

经过分类、解释的信号才能真正地成为 USB 设备（例如硬盘、鼠标等）所能接受的信息。

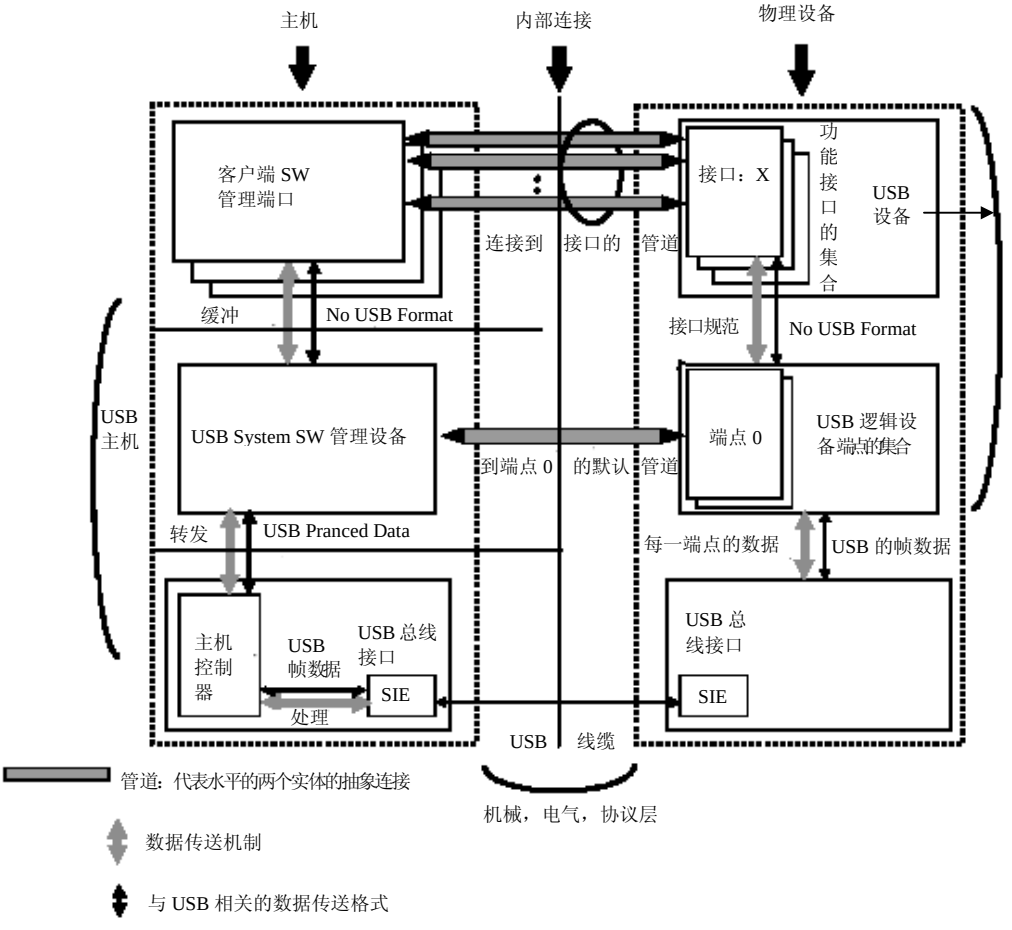


图 1-1　USB 设备与主机结构图

- 设备部分：是真正意义上的特定功能的外设，如硬盘、光驱等。

1.3.2 “主机”的概念

USB 设备与主机的关系是否等同于两台微机间的通信呢？事实上这两者并不能做完全的类比。

在更深入地接触 USB 时，会明显地感觉到，几乎所有相关的协议和设备开发商提供的 USB 控制器资料都在反复强调一个问题：“Host”只能是主机。

这是什么意思呢？简要地说，在处理主机与设备间的通信时，所有的请求、数据与控制信号都能够而且只能被主机响应后才能实现其功能。例如：在数据传输类型之一的中断传输中，接口控制器可以向主机提供中断传输数据请求，但是只有在主机轮询到该设备所在的端

口时，该请求才能被响应。

接下来，将对主机和外围设备所执行的功能做一简单介绍。

1. 主机的职能

正如前面所讲，主机在与外设进行通信时，始终占据着主动的地位，因此，对 USB 接口来说，主机最主要的功能便是对数据总线的管理。

- 检测设备：对于 USB 接口设备来说，主机在带电的前提下，能够对该种设备的连入与拔除做出自动的检测。主机检测设备的过程称为识别。在识别过程中，主机会从设备中读取一系列必须的信息，并为设备分配所需的资源，例如，地址、中断等。
- 管理数据流：连入主机的设备绝不可能只有一个，因此，主机在与每一设备建立了正确的连接之后，便需要协调其与每一设备间的通信。

在众多设备同时请求数据传送时，主机将数据通道按时间分割为帧或微帧，然后按帧或微帧分配给每个设备。

那些有特定带宽要求的设备，在识别过程中便会提出申请。主机将根据具体情况，或是不允许通信，或是将申请的带宽分配给该设备。有些设备还可以降低要求，仅申请带宽的一部分来适应传输。

- 错误检查：数据传送过程中，错误检测是必须的。不同的设备协议有着不同的校验方法。总的来说是在主机向设备传送的数据中加入错误检测位，设备会在收到数据后按照一定的规则进行检测。如果有错误，则返回否认信息，主机收到该信息后，便知道需要重新发送数据。

同样，设备在向主机发送数据时，也要在其中加入相应的检测位，主机在接收数据的同时进行校验。

- 提供电源：对 USB 设备来说，主机还需给 USB 设备的电源线提供+5V 电压及连接地线。

协议规定的最大电流为 500mA，而在某些情况中，一些由电池供电的微机端口和集线器只能支持低功耗的设备，其电流也被限定在了 100mA 以内。因此，有的设备需要自己提供电源，只是在初始通信时使用主机电源。

2. USB 设备的职能

如果说主机是微机与设备通信的控制中心，那么，USB 设备的控制器则可以作为信息的中继站。

它接收从主机来的信息，处理外设的响应，并使两者能够协调工作。因此，USB 设备控制器要既能处理设备与主机的通信，又能够独立地完成相应的任务。

（1）检测直接到芯片的通信

在总线中流动的数据信息总是携带着相应的目的地址，每一个外设都能收到该信息，设备在侦听到地址信息后，便要与自己相应的地址作比较，如一致则接受并执行相应的动作，如不同则丢弃。

在所有的控制器芯片中，这一系列动作都是自动完成的，且响应信息也是自动发出的。

（2）与主机交换数据

在与主机进行了正确的连接后，设备与主机的数据交换便是主要任务。如前所述，设备

可以向主机提出数据请求，同样，主机也可以向设备发送数据通信请求，这一请求可以是定时的（例如，特定程序中的定时数据传送），也可以是不定时的，而请求的类型由设备的驱动程序和应用程序共同决定。

对于主机的请求，设备以一定的方式进行响应，在 USB 接口中，它的响应方式分为 4 种：ACK、NAK、STALL 或是不发出任何反馈信息。

- ACK：响应正常。
- NAK：设备忙，请稍后再发。
- STALL：非法请求、请求失败、终端失败。
- 不响应：表示数据出现错误。

在设备独立完成的任务中，包括了特定设备的特定功能，例如：调制解调器对信号的调制与解调，打印机对信息的预处理等。

（3）USB 中标准请求的响应

在主机对设备进行识别的过程中，会要求设备给予相应的响应。在 USB 中规定的标准请求有 11 种，这 11 种请求是对设备能力各状态的查询及相应的配置请求。

设备只需将所响应的信息置于相应的缓冲区中即可。

（4）错误检测

检测的思路和方法与主机对数据错误的检测类似，在此不再赘述。

（5）管理电源

正如前一节所提到的，低功耗同样是 USB 接口的一大特点，而该特点的实现则主要依赖于 USB 设备控制器对电源的管理方案。

在整个工作过程中，控制器在不断监视着 USB 总线的工作状态，如果在一定的时间间隔内没有总线的活动时，设备便会首先进入低功耗状态。同时，保持对总线状态的监控，如总线恢复活动，则退出低功耗阶段。

而总线活动停止一定时间以上时，设备便进入挂起状态，并限制其从总线中获取电流。因而，在这样的管理体制下，USB 设备的功耗将被降低到了最小的限度。关于电源的管理，将会在后续章节中作详细的阐述。

1.3.3 USB 的端口

通常意义上的计算机端口被定义为用于连接其他电路的可寻址的地址，正如通常所接触到的较低级的语言一样，计算机对端口的访问与对内存的访问采用的是不同的指令。普通的端口之间其地址是相互独立的，连接到该端口上的设备单独地享用自己的数据通道，设备数据的发送与接受在设备间是独立的。在 USB 接口中，为了最大限度地节省主机资源，采取了不同的设计思路。

在用户的计算机上，也许有两个 USB 接口，但是它们并不像通常所看到的那样彼此独立，而是汇总到一个叫作根集线器（Root Hub）的地方，并由一个主机控制器所控制并共享一个数据通道。通常把每个 USB 接口称为一个端口。也许在每一端口上我们都接入了 USB 设备，但在实际工作中，在某一时刻，只有一个设备与主机进行通信。每个设备只能等候主机轮到与自己通信时才能占用数据通道。连接到主机上的设备越多，那么，每

个设备的可用通信时间也就越少。计算机系统中，这样共享数据通道的接口还有 Fire wire 和 SCSI。

1.3.4 “Function”的意义

在 USB 中将为主机提供单个功能的设备定义为一个“Function”，意思是此设备为主机提供了某些附加的功能，大多数的设备具有单一功能，例如硬盘、鼠标等。然而也有的设备集成了几种功能，例如常见的带轨迹球的键盘。

1.3.5 “Hub”的意义

在 USB 中 Hub 被定义为：包含有一个或多个为 USB 设备提供的接口或作为内部连接的设备。集线器“Hub”是 USB 规范中特别定义出来的外围设备，其主要功能为：

- 为 USB 设备提供连接口。
- 中继（Repeat）主机到设备或由设备到主机的信号。
- 控制各下游端口的电源管理。

1.3.6 “Device”的意义

Device 设备是指一个 Function（功能）或一个 Hub（集线器），其中，应注意的是由此衍生出来的 Device 的设备的含义，即该种设备中既包含了集线器又包含了一个或多个功能，或者是具有多个功能的设备，我们统称其为复合设备。

1.4 USB 的局限性

在上一节的内容中，着重介绍了 USB 的优点。那么，USB 真的是完美无缺吗？答案是否定的，任何事物都或多或少地存在着不足之处。

1.4.1 功能的局限性

虽然在其诞生之初，USB 便尽最大可能为使用者的方便着想，但其本身仍然具有不可避免的缺陷。

1. 带宽的限制

虽然经过了近两年的努力，开发商们将 USB 总线上的数据传输速率提高到了 480Mbit/s，但是在处理实时的图形数据时，这个速率仍然显得力不从心。因为这一取值只是一个理论上

的最大值，实际可用的带宽仅为 200Mbit/s~300Mbit/s。

虽然传输速率看起来颇为可观，实际上要将其应用到视频信号的实时传输中时，却显得远远不足。举例来说，在传输画质比较低劣的 640×480×24（bit）×30（Fram/s）的图像信号时，需要的带宽约为 220Mbit/s，已经几乎是 USB 2.0 可用带宽的全部了。

2. 对旧硬件的支持不足

由于完整的第一套 USB 可行的协议 USB 1.0 推出在 1996 年，因此，可以说这是一个新生的事物，以前的软硬件设施在一开始便对其缺乏足够的支持。像过去老式的计算机便没有携带 USB 控制器和根集线器。1996 年之前推出的操作系统 Windows 95 也没有 USB 设备的驱动信息，所以，用户要在这些旧式的计算机和系统中使用 USB，就要做出一些补充性的工作。

硬件方面，可用 USB 的适配卡在计算机中引入 USB 的接口；操作系统方面，最直接的方法便是引入更高级版本的系统来获得 USB 的使用支持。

3. 点对点的通信

正如在前面所强调的，在 USB 系统中，几乎所有的通信都是由主机作为主控者而发起的。从而直接导致了一个关键的问题，即 USB 设备之间不能做直接的数据交换，而必须在主机的介入下才能够完成。

作为 USB 接口的对手并起着补助作用的 IEEE-1394 则在这一点上有明显的优势，它允许设备之间作直接的连接与通信。

4. 距离的限制

USB 协议规定单条 USB 线缆的长度不能超过 5m，同时可以通过集线器的方式将其进行连接，最多可接入 5 个集线器将线缆距离延伸为 30m。如果是一般的 USB 设备应用场合，这一距离已基本能够满足用户的需求，但要将 USB 接口引入特殊的应用场合（例如远程的设备控制）时便有些力不从心。目前可用接口转换的方式将其转为可供长距离使用的接口，例如用 RS-485 等。

1.4.2 开发的难度

接触和开发像 USB 这样的一个新生的事物，对开发者来说并不是一个令人愉快的事情，因为新生事物往往意味着其本身的不完善，可供借鉴的经验的缺乏，以及设备本身的不稳定。

1. 复杂的协议

为了最大限度地方便用户的使用，所付出的代价必然是开发者投入更多的精力。这一点便集中体现在 USB 协议之中。为了尽量保证使用的稳定性，协议在数据传输和寄存器的配套使用上做出了十分细致的规定，开发者不得不对其中数以百计的寄存器做出一一合适的协调与使用。同时，由于其中加入了 CPU 的控制，因此，更要做好 CPU 与设备的连接控制、CPU 对总线的控制以及 CPU 在与主机通信时的连接等一系列的事情。

在开发者好不容易理解了几百页的协议资料后（一般是英文版），还不得不对自己选定的

芯片的资料加以掌握（又将是同样繁杂的几百页的英文资料）。在完成了对设备开发所必须的上述所有步骤后，往往还要在协议的基础上完成主机对设备驱动程序的编写。因此，要真正做好一个 USB 设备的开发，所要做的工作是十分繁重的。过去我们所熟悉的板卡设备的开发工作，是在一些非常简单的协议的基础上完成的。

2. 测试工作的困难

由于在 USB 产品开发过程中，涉及到了多方面的软件程序与硬件电路的联合使用，因此，给产品后期的测试工作带来了困难。通常使用的测试工具只能对其中某一部分的工作做出较好的检测（例如：固件程序的测试，利用厂商提供的开发包及模拟环境便能完成），但是对于系统的联合调试往往需要用昂贵的专用设备才能完成（例如：协议分析器）。

3. 控制器芯片本身的不足

由于 USB 的新生性，其相关的硬件尤其是控制器芯片本身存在着一定的缺陷。因此，随着协议本身的不断完善，厂商们也在不断地对自己的产品做出改进与修订。这便要求开发者必须不断地对 USB 相关制造商的产品信息给予关注并及时对自己所选的硬件做出更新。

1.5　本章小结

通过本章的学习，可以看出 USB 在其诞生之初，便注定具有强大的生命力和发展空间，这是由计算机要越来越智能化并方便人们使用这一基本的要求所决定的。

然而，世上没有绝对完美的事物，USB 既然具有了以前的计算机接口所不具备的便利性和实用性，那么必然决定了其开发过程的复杂和繁琐，而且在现有的条件下，并不是所有的场合都适合使用 USB 接口。在下一章中将会给出一些在工程的开发中对 USB 接口的选择建议。

第 2 章　如何着手 USB 的开发工作

知识点：

- USB 结构
- 开发工程的准备工作
- USB 论坛

本章导读：

本章将首先重点介绍 USB 的体系结构，然后结合作者本身的开发经验，给出开发 USB 设备应该注意的一些问题，简要地介绍 USB 开发者论坛。

在阅读完第 1 章的内容后，相信读者对 USB 的初步印象是：一个小巧的接口，简便易用而且价格低廉。那么，USB 是否对您的工作有益？在开发过程中应该注意些什么呢？本章所讲述的内容将对 USB 开发人员有所帮助。其中 USB 结构部分的详细内容，我们会在后续章节中一一介绍。

2.1　USB 结构简介

在上一章中介绍了 USB 接口是一个在高速模式下可达 480Mbit/s 的数据传送速度的串行接口，同时，主机掌握着整个数据总线的控制权。接下来，将介绍 USB 的两种基本结构的相关概念。

2.1.1　USB 的总线结构（物理结构）

在 USB 中采用的层叠星型拓扑（Tiered Star Topology）结构，如图 2-1 所示。

在 USB 协议中，通过级连的方式，最多可在主机上连入 127 个设备。

集线器是每一个星型拓扑结构的中心，而其终点便是相应的 USB 设备（Device）。正如前面讲到的设备概念一样，它可以是一个功能、一个集线器，或是一个复合设备。通常所见到的集线器为 2、4 或 7 个端口。所以，每一层或每一层中的设备数不定。

物理上的层叠星型结构并不影响主机在实际工作中的效率。主机在工作时并不考虑某一设备是通过几级的层叠进入的。同样，其中的设备在经主机端发送信息时，也并不需要考虑通过几级集线器。在逻辑上，集线器对主机和设备来说是透明的。

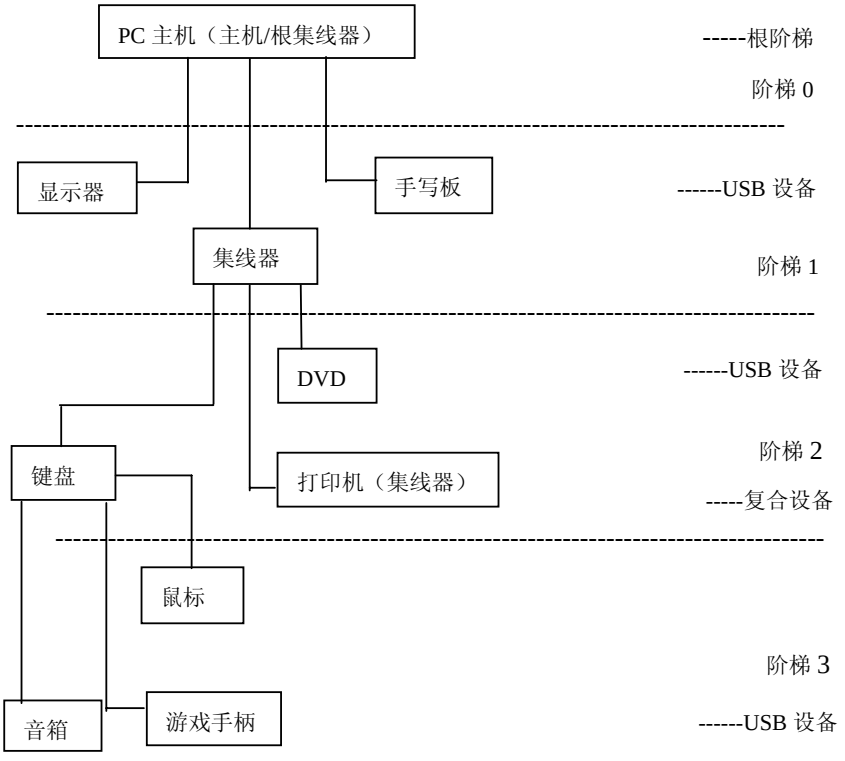


图 2-1　USB 总线拓扑结构

要强调的是，不管在一个根集线器上连入多少设备，所有的这些设备都在共享主机的数据通道。因此，在其中接入的设备越多，每一设备所能得到的带宽也就越少。

2.1.2　USB 的逻辑结构

正如上节所述，物理上的连接并不代表逻辑上的结构，在 USB 中主机与设备之间总是以一一对应的方式进行逻辑连接的。例如：图 2-1 中所示的物理结构可以由图 2-2 所示的主机与设备的逻辑连接表示。

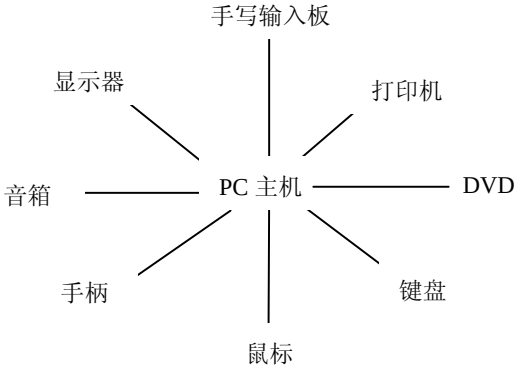


图 2-2　USB 设备与主机的逻辑连接

USB 中的逻辑结构是主机与设备交换数据的基础。同时，所有的这些设备都在共享同一条数据通道（在连入同一根集线器的情况下），可以将这条数据通道理解为一条管道。在高速模式下，这条管道的带宽为 480Mbit/s。而在这条管道中，最多可容纳 127 条小管道。每一条小管道的终点便是一个 USB 设备。

在每一条小管道中流动的数据都带有一个称为令牌（Token）的信息包，它给出了数据的流向。如果令牌信息为 IN，则数据流向为由设备到主机；如果令牌信息为 OUT，则数据流向为由主机到设备。

每一个令牌中有 7 位数据用来寻址设备，最多可寻址 128 个设备。但由于地址 00H 总是默认分配给刚刚连入的设备暂时使用。因此，实际工作中的寻址设备数为 127 个。

在令牌包中还包含了 4 位的端点（End Point）地址，因此，在每一条小管道内又可分为 16 条微管道，也就是可以寻址 16 个输入/输出的端点，该端点就是数据流中的最基本的信息单元，每一个端点中可以携带不同的信息，例如：数据、音频、视频及控制信号等。

2.2 必要的准备工作

2.2.1 主机

着手更进一步的工作之前，首先应确定所使用的主机是否支持 USB 接口，此处所指的主机就是通用的 PC 机。

1. 主机控制器的支持

由于所有的 USB 的通信都必须由主机来进行启动，因此，支持 USB 接口的主机控制器是必不可少的。USB 的主机控制器负责启动数据总路线上的数据交易（Transaction），根集线器（Root Hub）负责为设备提供端口。

Microsoft 公司与 Intel 公司在 PC 99 指导方针中推荐所有的制造商提供两个可用的 USB 端口，后来，这两家公司又共同在 PC 2001 System Design Guide 中更进一步强调并限制了制造商必须对 USB 端口提供支持。

在 USB 开发者论坛中，同样提供了检测 PC 是否支持 USB 的测试工具。地址为：

Http://www.acwsoft.de/acw-usb2.html

在该论坛中可以对计算机中系统软件、硬件及驱动程序对 USB 的支持与否进行全面的测试。如果用户的计算机已经支持了 USB 接口，那么，在系统的【控制面板】/【系统】/【设备管理器】中找到 USB 设备的信息。单击“”标志的通用串行总线控制器，会发现系统已将所有连入主机的 USB 设备一一列出在其后打开的目录中了，如图 2-3 所示。

当然，最基本的两个设备如下：

- Standard Uinversal PCI to USB Universal Host Controller。
- USB Root Hub。

也就是 USB 主机控制器和 USB 根集线器。同时，细心的读者还会发现，在图 2-3 中包含有不止一个 USB 的主机控制器和根集线器，这是因为在该计算机中，又加入了一个单独的

USB 转接卡的缘故。

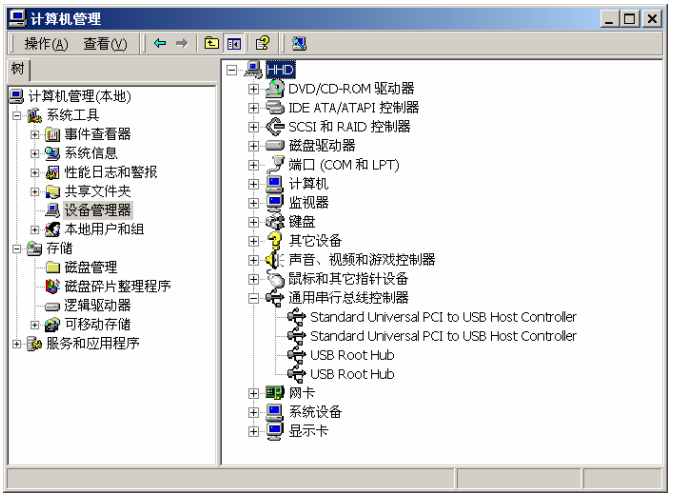


图 2-3　设备管理器中对 USB 设备的列举

USB 主机控制器按其驱动程序接口的不同又可以分为：

- 开放式主机控制器（Open Host Controller，OHC）。
- 通用式主机控制器（Universal Host Controller，UHC）。

其中，Microsoft、Compaq 和 Notional Semiconductor 等厂商联合制定了开放式主机控制器的标准，而 Intel 公司则自己研发了通用式主机控制器。

它们两者虽然在体系结构和名称上有所不同，但对于使用者来说，提供的是相同的功能。当然，如果用户的计算机主板没有带 USB 接口的主机控制器，那么，也可以在相应 PCI 插槽中插入由 PCI 到 USB 转接的适配器（如图 2-3 中所示）。

2. 操作系统的支持

具备了必要的硬件设备，下一步便是寻求合适的软件系统。在我们所熟悉的操作系统中，最早提供对 USB 支持的是 Microsoft 系列中的 Windows 95 的 OEM Service Release 2 版本，它同其后的 OSR 2.1 与 OSR 2.5 一样，服务的对象是计算机制造商和零售商。

在 1997 年 7 月问世的 Windows 98 则将 USB 的支持包含在了操作系统之中，也就使其服务的对象转向了广大的消费者。随后的 Windows 98 SE（Second Edition）更进一步完善了对 USB 接口的支持。

在 Microsoft 系列中的另一体系 Windows NT 中，Windows NT 4 并没有提供对 USB 的支持，随后推出的以 Windows NT 4 为基础的 Windows 2000 弥补了这一不足，加入了对 USB 的支持，而且与 Windows 98 相比，运行于 Windows 2000 上的 USB 更为简便和稳定，这是因为 Windows 2000 中本身便已携带了大多数的通用 USB 接口设备的驱动程序。这就使得用户省去了第一次接入 USB 设备时需加载相应的驱动这一步骤。

最新的家用型 Windows XP 是基于 Windows 2000 的构架发展起来的，因此，同样支持 USB 的工作。

要强调一点，这里所说的支持与不支持是对于操作系统内部有无对 USB 接口的底层设备驱动信息来说的。因此，即便是在不支持 USB 接口的系统中，我们仍然可以在系统中载入第

三方的软件驱动，使相应的 USB 控制器作为一个外设工作于系统之中（例如：Windows NT 4）。当然，在系统中创建一个完整的第三方的软件驱动是一个复杂和高难的工作，这些内容会在相应的章节中进行介绍。

在确定了使用的主机中有了 USB 主机控制器和根集线器，同时系统也具备了对 USB 接口的支持特性后，下一步便应考虑一些具体的问题了。

2.2.2 开发过程中应考虑的问题

在决定了开发过程要使用 USB 接口之后，下一步就应该明确考虑将可能遇到的问题。

1. 选择合适的 USB 设备控制器芯片

这是一个首要而慎重的步骤，随着 USB 的日益发展，它已经有了 3 个阶段性的跨越：低速模式、全速模式和高速模式。确定工程适合使用哪种模式的控制器直接决定了随后工作的方向和复杂度。

同时，随着 USB 的日益推广，越来越多的厂商开始提供 USB 设备控制芯片。例如：CYPRESS、Philips、Net chips 等许多厂商都提供相应的 USB 设备控制器芯片。那么该从哪几方面来确定芯片的选择呢？

（1）所需的速度

正如前面所说，不同的设备应配备不同的 USB 芯片，在一些需要高速的、大批量处理数据的设备中应考虑 USB 2.0 的高速版本的芯片；而在一些普通的、只需处理一些简单的、非实时数据的设备中，则只需使用全速或低速的芯片。在这种情况下，虽然高速芯片同样能够适应工作但那只会增加开发工作的复杂性。

常见的外围设备对控制芯片的应用情况如表 2-1 所示。

表 2-1　　USB 芯片使用范围的比较

控制芯片	产品举例	备注
低速（10K~100kbit/s）	鼠标、键盘、杆、POS 机	开发周期短，所需应用程序代码简单
全速（50K ~10Mbit/s）	显示器、扫描仪、音箱、ISDN、电话、话筒	具有一定带宽保证，所选芯片较多，应用程序相对较为简单
高速（25M~500Mbit/s）	影像、实时视频、海量储存	芯片开发周期长，有较高的带宽保证

从上表可以看出，不同版本的 USB 芯片有不同的应用场合。

（2）是否有集成于内部的 CPU 芯片

一个 USB 控制器必须有相应的 CPU 处理器才能够完成 USB 非通信的及主机的响应工作。一个内置的 CPU 可以简化一定的数据处理的工作，而外置的 CPU 的设计思路要更明晰一些。因此，出于个人的习惯及工作的需要，确定使用芯片是很必要的。

（3）连接端口的数量

不同的 USB 控制器芯片提供了不同的 I/O 口、控制信号线及 IIC 高等外部的接口线，确定所选芯片的信息端口能否满足工作的需要是另一个重要的问题。

（4）合适的数据

通常的控制器芯片内都集成有一定的内存来作为随机数据存储器，一般情况下 RAM 大小在 128~1024Byte 之间。应确定所选芯片的 RAM 大小能否满足设备数据的储存与读取。

（5）程序存储器

要使 USB 设备正常工作，还需要特定的用于初始化的设备和与主机进行通信的程序。

在 USB 中这样的程序称为“固件”（有关“固件”的相关知识将会在第 11 章中着重介绍）。不同的芯片固件的存储位置不同，有的需要存入特定的程序内存中，有的利用芯片本身提供的程序储存器，这便要求用户充分分析应用场合程序的复杂度，来确定所需程序存储器的大小。

（6）其他

当然，除了上述的要点外，像驱动示例的支持、模拟环境的提供等都是作为芯片选择应有的考虑。

2. 确定创建固件所需的语言环境

前面已提及，固件的编写是控制器芯片正常工作的前提。通常在选择了一定的 USB 控制器芯片后，芯片本身便会要求开发者使用一定的编程语言来进行固件的编写。例如：CYPRESS 公司的芯片，由于其内部集成的是与 8051 兼容的 CPU 芯片，因此，其固件要用相应的汇编语言来编写其固件。

当然，对于内部没有集成 CPU 芯片的控制器，在选择开发环境时便有了更大的灵活性。可以使用 C 或其他的高级语言，不过，所必须的是要有能够将其翻译为机器语言的编译器。

3. 硬件电路设计时的考虑

这里，硬件电路不仅仅是指 USB 控制器本身的外围电路，而且应该考虑到用户要用 USB 连接的外设。

首先，由于 USB 控制器本身内建电压为 5V，因此，首先应考虑相关外设的电压匹配与驱动能力的问题。

其次，USB 向上游连接入主机的接口及相关电路已成为标准化的规定，因此，要考虑从 USB 控制器向下接入外设时的数据信号，控制信号的连接，往往大多数的 USB 控制器芯片都提供了足够强的功能以满足外设的需求，灵活并合理地利用所提供的软硬件功能，往往会使用户的工作收到事半功倍的效果。

最后，完成硬件电路的设计与绘制。制作硬件电路板的计算机辅助工具已有很多，如 Protel、AutoCAD 等都可以完成相关的工件。不过考虑到 USB 接口引入的本身便是为了简便地使用各种设备，因此，应以产品更加小巧和简便易用为目标。

4. 主机中设备驱动的缩写

要使所开发的产品能够在计算机上顺利地工作，计算机必须能够“识别”用户的产品，并将其引导入系统中，对其进行协调和控制，这个“识别”的过程便是设备驱动程序所要完成的任务。对大多数的标准设备来说（例如鼠标、键盘），计算机系统本身已经有了内建的对其支持的驱动程序，而用户自己所开发的产品往往是不能直接在计算机上使用的，这便需要自己编写该设备的驱动。

设备驱动的编写思路是一致的，实现方法可以有多种，就 Microsoft 公司本身来说，它的

系统驱动可以以 Win DDK 的方式来开发，也可以利用其他的开发软件来进行。但由于前一种方法对计算机专业知识的要求太高，而且实现起来相当困难。因此，大多数的用户会选择后一种方法，编译的平台可以是标准 C、VC++等。

5. 应用程序的实现

在顺利地将设备连入主机之后，USB 开发才真正地迈出了第一步。因为要在计算机上实现对所开发的产品进行应用与控制，所以，必要的应用程序是设备满足用户需求的基础，而这一部分的程序便更具有灵活性了。

6. 相关的辅助软硬件设备

在确定了大的设计模块之后，还要认真分析一下实现完整的一个产品的开发还需要哪些东西。

就产品硬件方面来说，必要的辅助电路（例如：开头的控制、数据的读写与存储、产品外形的设计等）都是应该考虑到的问题。

就 USB 接口的开发过程来说，除了厂商一般所能提供的开发板、模拟运行平台等，还可能需要用到必要的监控程序、协议分析器等设备。

2.2.3 必要设备的准备

综上所述，用户在开发过程开始之前所具备的条件分为元件和工具两个部分。

元件包括以下几个方面：

- 合适的 USB 设备控制器芯片。
- 相应的支持 USB 设备的计算机。
- 相关的外设所需的硬件。

工具包括以下几个方面：

- 开发 USB 控制器固件所需的软件平台。
- 开发 Windows 驱动程序所需的系统。
- 开发应用程序所需的编译工具。
- 硬件电路设计的软件平台。
- 硬件电路组装的硬件工具。
- 必要的开发板，模拟器调试工具。
- 必要的硬件接口测试器，协议分析器。
- 系统联调时所需的应用环境。

2.2.4 开发工作流程

在完成了上述的工作之后，终于可以进入实质性的开发阶段了。但是，在进行产品的实质性开发之前应事先明确开发的条理性，一般开发过程的工作流程如图 2-4 所示。

整个 USB 的开发过程可以分为以下步骤。

（1）硬件的设计

主要考虑控制器与主机的连接，与外设的匹配，以及它们之间的协调工作。

（2）USB 控制芯片固件的实现

要注意到具体的应用环境控制器芯片中各种寄存器、控制器的不同的应用与实现。

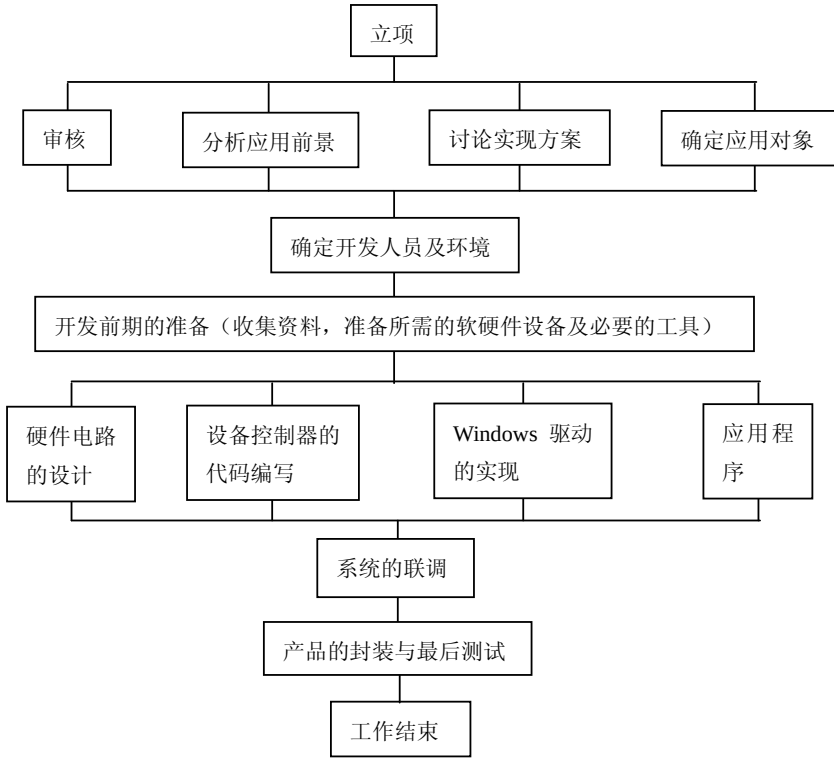


图 2-4　一般开发过程的工作流程

（3）Windows 驱动的编写

Windows 驱动的编写包括检测设备时系统要用到的引导文件、设备信息文件等。

（4）应用程序的实现

根据不同的工程编写不同的客户应用文件，这几部分的工作可以同时进行，因为它们之间并无明显的时间前后的顺序关系，但是必要的交叉与协作是工作进行的前提。

最后是这几部分的联合测试与协调工作问题，在这一过程中可能使用户不得不重新返回到前几步的工作中对产品各方面的开发工作进行重新的审视与设计。

当完成了上述的工作后，开发工作便基本结束了，剩下的只是产品的外形及完善工作。

2.2.5　最后的考虑

在拿出最后的成品之前，作为一个有经验的开发者，总会在下面一系列问题上做出预先的考虑。

（1）产品的升级

随着技术的进步，USB 的版本会不断地推陈出新，产品在问世之后，对于新的 USB 的

芯片及更高的用户要求如何快速地适应，将是今后工作的重点。这就要求用户在开发时要更全面地考虑到产品的升级与修订。

（2）产品的多用途性

如果所有的开发工作只是针对非常狭窄的工作应用场合，那么，其效益必定不会很好。因此，在开发一个产品过程中，应尽量使自己的软硬件设计具有较强的可移植性。

（3）产品的稳定性与健壮性

实验室的实验环境毕竟与实际工作中的工作环境有着或多或少的差别，如果所开发的产品只能在特定的实验环境中才能良好地工作，那么，售后的服务将会是一个很大的问题。当然，关于这些方面的问题在一些相关的更专业的书籍中会有更具体更系统的阐述。在此不再详述。

在对 USB 开发有了全面的了解之后，接下来便要进到对 USB 的学习当中了。在此之前首先简单地介绍一下 USB 开发者论坛。

2.3 关于开发者论坛

USB 开发者论坛（www.usb.org）是由开发 USB 的厂商所成立的非盈利性的公司 USB 实施者论坛（USB Implementers Forum，InC，USB-IF）建立的网站。在这个网站上可以找到一系列的 USB 的协议、工具程序及测试程序等相关信息。由于创办这个网站的目的便是为了更方便地对 USB 产品的开发给予足够的支持帮助，因此，其上的所有工具软件和协议都可能免费获得。用户还可以在论坛上发表自己的见解，提出所遇到的问题以供大家讨论并获得帮助。

但是对于 USB 产品的制造商来说，要在市场上出售带有 USB 接口的设备，则必须在网站上进行登记，以取得一个厂商的 ID 使产品合法化。

2.4 本章小结

这一章在介绍了 USB 的体系结构的基础上，对所要从事的 USB 开发工程作了简要的介绍。总的来说，对 USB 开发工程的考虑主要应该从软件和硬件两个方面着手。硬件的考虑包括：芯片、外围电路以及相应的接口电路。软件的方面主要有：设备的驱动、固件的编写。

正如在前一章中所提到的，由于 USB 本身体系结构的复杂性，使得开发工作必然不是一帆风顺的，需要以许多的相关知识作为铺垫。在以后的章节中将会以简明扼要、重点突出的方式来介绍 USB 的相关知识，希望能对读者有所裨益。

第 3 章　集线器

知识点：

- USB 的集线器
- USB 体系中的帧和微帧的概念
- USB 的端口

本章导读：

USB 设备最大的特点：即插即用的实现是由协议、硬件和软件等方面来保证的。本章将介绍实现即插即用功能的第一部分：集线器。

在刚开始接触 USB 时，其最吸引人的莫过于其便捷的使用方式，而在其中“即插即用”这一特性扮演了决定性的角色。在接下来的各章节中，将力图对 USB 的这一特性的实现做出简明的分析。

3.1　USB 集线器

集线器对于每个计算机工程人员来说并不陌生，USB 中集线器的概念与普通的集线器的概念既有相似之处，但也有其不同的地方。

3.1.1　通常概念意义下的集线器

集线器（Hub）是指单一总线共享式设备，它需提供多个网络接口，负责将网络中多个计算机连在一起。所谓共享是指集线器的所有端口共用一条数据总线，即共享带宽。其带宽由它的端口平均分配，如总带宽为 10Mbit/s 的集线器连接 4 台工作站同时上网时，每台工作站平均带宽仅为 10/4=2.5Mbit/s。

集线器为共享方式，即同一网段的计算机共享所有的带宽，传输通过碰撞检测进行，同一网段中计算机越多，传输碰撞也越多，传输速率就会越慢；现在的集线器普遍采用了自适应（Auto-Sense 或 Auto-Negotiation）技术，可以有自动适应 100Mbit/s 和 10Mbit/s 这两种目前所用最多的网络速度。集线器按以下顺序适应工作速率：100M 全双工、100M 半双工、10M 全双工、10M 半双工。

一般的集线器上都有 Collision 灯。由于以太网中普遍采用了 CSMA/CD 协议，传输过程中可能发生冲突，此时，Collision 会不断闪烁。Collision 灯闪烁的频繁程度，表明了网络负载的繁重程度。

3.1.2　USB 中的集线器（Hub）概念

USB 中的 Hub 提供了 USB 设备和主机之间的电气接口，其主要由 3 个部分构成：中继器（Hub Repeater）、控制器（Hub Controller）、事务转发处理器（Transaction Translator）。

- Hub Repeater：用来响应主机与设备的连接的建立与断开。同时，支持总线数据错误的检测与恢复、总线连接与断开的检测等功能。
- Hub Controller：主要负责主机与集线器间的通信事务。提供 Hub 的特殊状态与控制命令以便主机来对 Hub 进行配置，同时负责监视和控制其下游端口的活动。
- Transaction Translator：对高速通信过程中的事务进行分割，并在其下游端口有全/低速设备接入时，将事务转换为下游设备可以接受的全/低速型事务。

Hub 具有许多用来给使用者提供方便，并掩盖 USB 通信的复杂过程的特性。Hub 的主要功能如下：

- 管理主机与设备的连接。
- 电源管理。
- 设备接入/断开检测。
- 总线错误的检测与恢复。
- 对高速、全速、低速设备的支持。

3.2　Hub 的体系结构

Hub 及其上下游端口的结构示意图如图 3-1 所示。由图可知，Hub 必须包括 3 大部分：Hub Repeater、Hub Controller、Transaction Translator。Hub 在其上游端口接入高速环境中时，必须工作于高速模式；在其上游接入全/低速环境中时，必须工作于全/低速模式；在上下游端口接入同种设备时，Hub Repeater 负责连接的建立与管理，它应能支持 3 种速度模式。Hub Controller 提供必要的状态和控制信号来使主机能够访问 Hub。

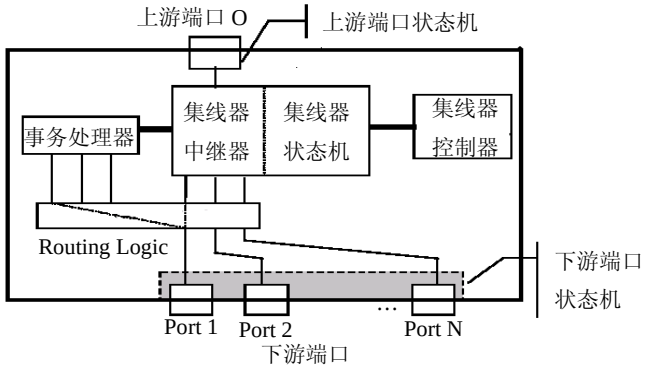


图 3-1　集线器的内部结构

当 Hub 工作于高速模式下，但其下游接入全/低速设备时，Transaction Translator 对高速

事务进行分制，将其转换为全/低速事务。下游端口接入设备的速度决定了 Hub 中路由逻辑部分是否要接入 Transaction 或 Hub Repeater 部分。

当一个 Hub 的上游端口接入一个工作于全/低速的电气环境中时，Hub 的高速功能是被禁止的，这意味着 Hub 只能工作于全/低速模式。同时，事务转发处理器和高速中继器不会参与工作。在这种情况下，Hub Repeater 必须作为一个全/低速的中继器来工作，而且路由逻辑部分会将端口接入 Hub Repeater。

当 Hub 的上游端口连入一个高速模式的环境中时，全/低速的中继器不会工作。在这种情况下，当下游端口也接入高速设备时，路由逻辑将端口连入 Hub Repeater，且 Hub Repeater 工作于高速模式。当下游端口接入的是全/低速设备时，路由逻辑则将端口接入 Transaction Translator。

1. Hub 的连接

依据 Hub 是否要传送交易包，或转发信号，或是处于空闲状态，Hub 会表现出不同的连接行为。

2. 数据包信号的连接

Hub Repeater 通常会包含一个向上游的端口和多个向下游的端口。通常上游端口接入主机，下游端口接入设备。Hub 处理上下游数据时信号的流向如图 3-2 所示。Hub 也可能处于空闲状态，该状态用作在接收数据包间的等待时间。

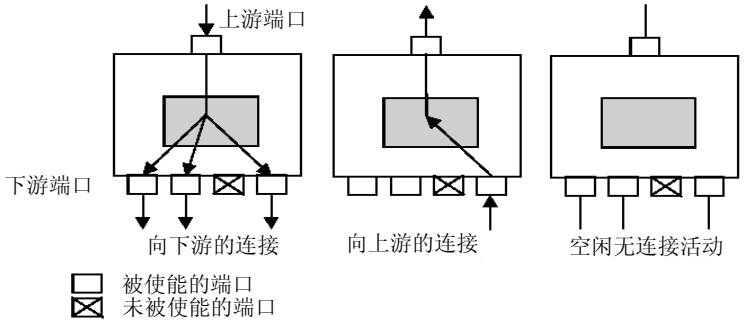


图 3-2　HUB 的信号连接

如图 3-2 所示，在某下游端口被接通后，并且 Hub 检测到了有从下游向上游流动的数据时，它会建立一个流向上游的数据通道。只有 Hub 本身能够“看到”该通道，而其他下游端口是不可见的。

在数据向下游流动的模式中，Hub 采用广播的方式。在数据包被检测到后，Hub 为每一条接通的下游端口都建立一条数据通道，使其能够收到数据，对于那些未接通的端口，Hub 则不会做类似的传递工作。

3.2.1　连接的重续

依据向上游或向下游的不同的重续信号，Hub 会做出不同的连接行为。重续信号的示意图如图 3-3 所示。

对于从上游端口来的重续信号，挂起的 Hub 将其转发给所有被连通的下游端口。对于从下游的被暂时挂起或连通的端口来的重续信号，Hub 不仅将其传递给上游端口，同时还将该信号反馈给所有连通的下游端口，包括发出重续信号的端口本身。

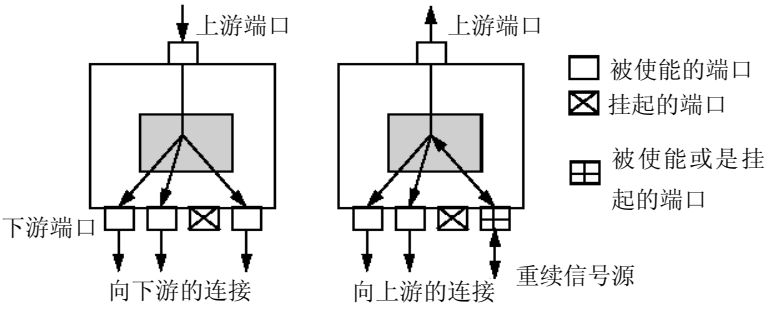


图 3-3　重新连接

3.2.2　Hub 的错误恢复机制

Hub 是主机和设备之间建立连接的重要部件，能够检测到并阻止那些连接中的错误。例如：可能的死锁等。对于 Hub 的正常工作来说这是至关重要的，只有在工作于中继器模式时，Hub 才需要检测连接的错误。

Hub 也必须能够对那些发往控制器的数据包的丢失和损坏做出检测和恢复。事实上，Hub Controller 应该属于 USB 的设备，因此，它必须遵守与其他 USB 一样的时序规则。

3.3　Hub 的数据帧与微帧的计时器

每一个 Hub 都有一个帧（微帧）的计时器，它的时钟从 Hub 的本地时钟信号获得，并且通过由主机发出的 SOF（Start of Frame）信号与主机帧/微帧取得同步，帧/微帧的计时器提供时钟参考用来检测下游失效的设备，同时预防被上游 Hub 错误地中断工作。Hub 的帧或微帧的计时器必须跟踪主机的帧/微帧时间，而且要能够在连续丢失两个 SOF 信号的情况下仍能与主机保持同步。

帧/微帧的计时器必须在主机时钟最大偏差或时序紊乱的情况下仍然能够锁定数据帧/微帧的时间。在高速/全速模式下工作的 Hub 仍然对其有特殊的要求。

3.3.1　高速模式下微帧的计时范围

在高速模式下，微帧的计时范围为 59904~60096 个高速数据位传送的时间，每个数据微帧规定的名称时间为 60000 个数据“位”时间。而在实际中，为了与主机的准确性、Hub 的准确性、中继器的抖动等进行容错，在该数值的基础上加入了+1~96 位的调整值，具体描述如表 3-1 所示。

表 3-1　　高速模式微帧计时误差的来源

误差的来源	变更值（ppm）	每一微帧间的误差值（位）	备注
主机的准确率	+1~500	+1~30	

续表

误差的来源	变更值（ppm）	每一微帧间的误差值（位）	备注
Hub 的准确率	+1~500	+1~30	
主机的抖动		+1~2	
Hub 的连环抖动		+1~20	这是在假设上游已接入 4 个 Hub，每个 Hub 有 0.25 位的抖动情况下
量化误差		+1~14	对于需要循环的数据位，总的误差应乘以 16

3.3.2　全速模式下帧的计时范围

全速模式下帧的计时范围应为 11958~12042 全速数据位时间。全速帧计时的定义值为 12000 个数据位时间。同样，为了容错，在此数值的基础上加入了+1~42 个数据位的纠错时间，如表 3-2 所示。

表 3-2　　全速帧的误差来源

误差来源	变更值（ppm）	帧间变更值（位）	备注
主机的准确率	+1~500	+1~6	
Hub 的准确率	+1~500	+1~36	其中，+1~6 位由 Hub 的准确度决定，1500（ppm） +1-30 位来自于其上级 Hub 的准确率（2500ppm）

3.3.3　帧/微帧的计时同步机制

一个 Hub 的帧计时器由 Hub 本身的时钟源锁定，而由从主机来的 SOF 信号进行同步。在复位或重续之后，Hub 的帧计时器不需要被同步。但在接收到了两个连续的 SOF 包之后，Hub 的帧/微帧的计时器便必须被同步化。

下面将给出一个建立合适的同步化的计时器的示例。

1. 计时器同步化

Hub 保存着 3 个计时值：帧/微帧计时器（递减计数器）、当前帧/微帧（递增计数器）和下一帧/微帧（寄存器）。在完成了一个复位和重续之后，应该有一标识位被置位来示意帧/微帧计时器没有被同步。

当第一个 SOF 标识被检测到后，当前帧/微帧计时器被复位，并且按 Hub 的位时间（Bit time）来计数。在下一个 SOF 包到来时，如果当前帧计时器没有翻转，则当前帧/微帧计时器

的值将被装入下一帧/微帧寄存器（Next/Micro frame register）和帧/微帧计时器（Frame timer）中。当前帧计时器被复位为零，重新开始计数，前面所提到的标识位应被置位，暗示帧/微帧计时器已被锁定。帧/微帧计时器在高速下计时到 60096 时，或在全速下计时到 12042 时会进行翻转。

在下一个 SOF 包到来时，如果当前帧计时器已经翻转，那么一个 SOF 包将会被错过，帧计时器和下一帧寄存器也不会被装入值。在这种情况下，同样应有相应的标识位给出计时器仍未被锁定的提示。

在任何时候如果帧计时器递减到零，那么下一帧寄存器中的值将会被装入帧计时器。当 SOF 被探测到时，且当前帧计时器仍未翻转，那么当前帧计时器中的值将会被装入帧计时器和下一帧寄存器，当前帧计时器被清零且重新开始计时。

如果当前帧计时器已经翻转，那么“下一帧寄存器”中的值应被装入“帧计时器”。因此，这一过程会使得帧计时器在一个帧/微帧时间内被更新两次：一次是帧/微帧计时器已经递减到零，另一次是 SOF 包到达时。

2. 帧结束（End of frame）的提前

在 Hub 的帧开始（Start of frame）解码时间的基础上它必须在一定程度上将自己的 EOF 时间点提前，这是因为集成器采用了层叠结构，因此在每一级层叠中都会产生延时，从而远离主机的 Hub 总比靠近主机的 Hub 有着较晚的 EOF 时间。

时间提前的范围一般来说由以下部分决定：

用来进行同步的电路决定其 EOF 时间点是以成功解码 SOF 的标识包为基础。这意味着帧/微帧计时器同步的基准时间要比真正的帧开始时间晚一些，高速模式下晚至少 40 个位时间，全速模式下晚至少 16 个位时间。同时，每一个反应过程（如写寄存器等）也要花费一定的时间，反应时间在高速模式下为 192 个位时间，全速模式下为 4 个位时间。由帧定时器控制的反应动作被分别在 EOF1、EOF2、EOF 中进行定义。这些定义都假设 Hub 的帧计时器在 EOF 到达时都已递减为零，而在实际中实现以上功能的电路都要在 SOF 之后再计数 40~192 位时间（高速）或 16~20 位时间（全速）才能使帧计时器递减为零。这些计时和位的偏移都必须在其后的计时过程中加以修正。将数据处理过程产生的延时修正后就能确保在 SOF 之间只有传输延时的存在。

举例来说，在 USB 层叠结构中，在 USB 中任何两个地方的 EOF 的传递延时都不应超过 216 个位时间（其中，144 个是在每个中继器上假设为 36 个位的最大延时而且穿过 4 个中继器，而 72 个位时间是假设每条线缆要延时 30ns，不经过 5 条线缆），如图 3-4 所示。由于帧计时器递减到零所需数值的不确定性，使得离主机较近的 Hub 可能的延时要比距主机较远的 Hub 产生的延时长。

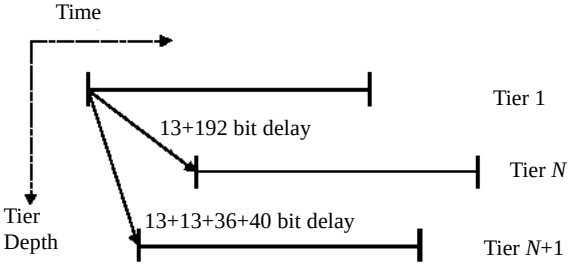


图 3-4　在没有 EOF 提前机制的情况下，由于传输的延时引起高速模式下 EOF 的偏移

经过了修正的 SOF 时间如图 3-5 所示，远离主机的 Hub 总是要比靠近主机的 Hub 的 SOF 时间晚。

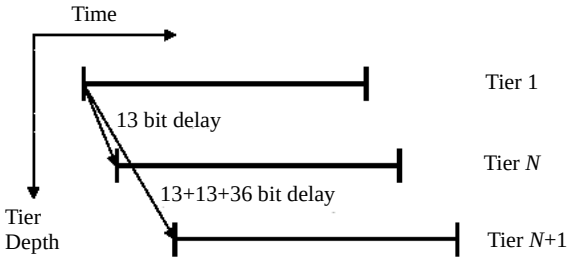


图 3-5　引入了 EOF 提前机制后，高速模式下正确的 EOF 时序

3. 关于 EOF1 和 EOF2 时间点

EOF1 和 EOF2 是从 Hub 的帧计时时间中分离出来进行定义的时间点，EOF1 和 EOF2 的时间点如表 3-3 所示。

表 3-3　　Hub 或 Host 的 EOF1 和 EOF2 时间点

名称	高速下比 EOF 提前的时间（位）	全速下比 EOF 提前的时间（位）	备注
EOF1	560	32	帧/微帧结束点 1
EOF2	64	10	帧/微帧结束点 2

这些时间点的定义是为了在从主机发出的 EOF 包丢失的情况下，仍不影响设备的正常工作而定义的。需注意的是，这些时间点的定义只有在帧/微帧的计时器已经通过 SOF 进行了同步的情况下才有意义。

在 HUB 和主机的帧计时时间已经同步后，它们必须在主观上明确 EOF 点本身具有的偏离。

高速模式下 EOF2 的时间点如图 3-6 所示，高速模式下 EOF1 的时间点如图 3-7 所示。

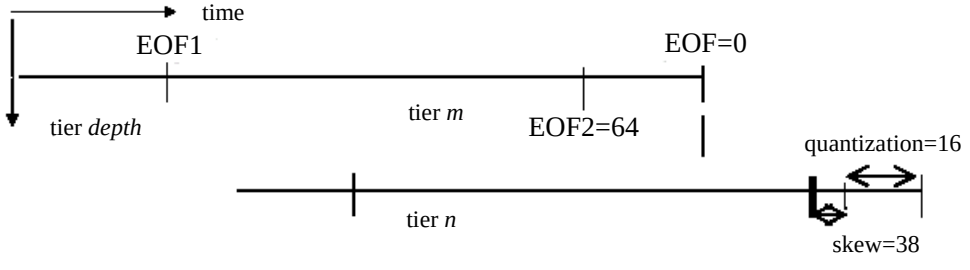


图 3-6　高速模式下 EOF2 的时间点

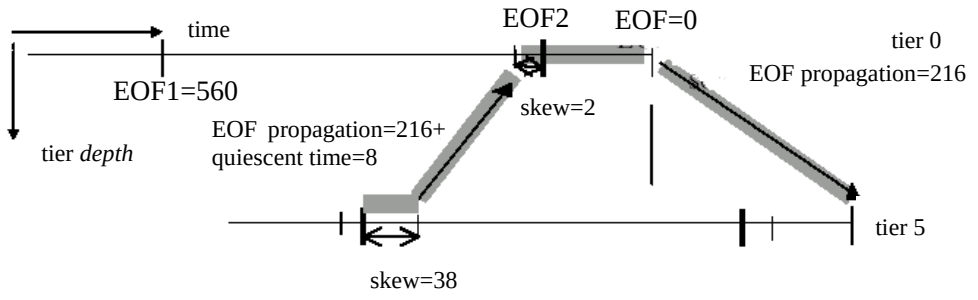


图 3-7　高速模式下 EOF1 的时间点

在 EOF2 时间点，任何向上游申请连接的端口请求都被视为无效。在全/低速模式下工作的中继器为了避免被禁止工作，它们往往需要在上游 Hub 到达其 EOF2 点时，送出一个包结束（EOF）信号到上游 Hub。全/低速模式下 EOF 的时间点如图 3-8 所示。

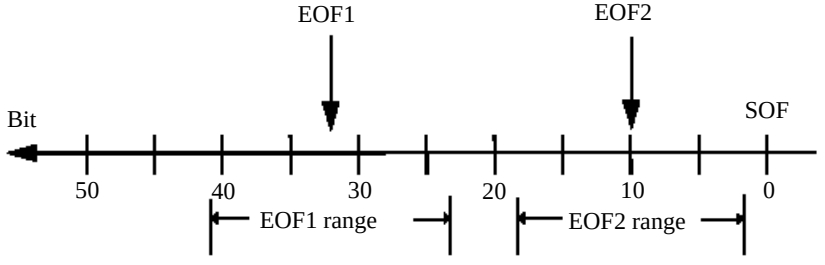


图 3-8　全速模式下 EOF 的时间点

作为全/低速中继器工作的 Hub 在 EOF1 时间内仍未同上游建立连接，便允许发出 EOP 信号；全速模式的中继器如果与下游的任何一个端口在 EOF1 点前建立了连接，则必须发出 EOP 信号。

高速模式的中继器必须在 EOF1 时间点中断与上游的连接活动。高速模式的中继器在总线返回空闲状态之后必须中止连接行为而不能像在全/低速模式中那样以 EOP 来解决问题。主机控制器的设计者必须有能力正确处理这样的信号包。

（1）高速模式下的 EOF1 和 EOF2 时间点

虽然 Hub 已经与 SOF 取得了同步，但时间的偏差仍然有可能在主机和 Hub 或 Hub 之间进行积累，这个时间的偏差是由于在不同的主机和 Hub 上产生不同的帧定时时间而导致的。这个偏差最大可达到 38 个“位”时间。其中，包括±2 个“位”时间的主机抖动，0~36 位的中继器的抖动。这些偏差都将影响 EOF1 和 EOF2 点的确定。

注意：高速模式下微帧的定时器一旦被同步后，微帧的间距就被认为不会被主机所改变，这是 Hub 偏移时间估算的前提。

EOF 的偏移范围可能是-2~+38 位，因此，所有的 EOF 信号之间的累积差异就可能达到 256 位（216 位的传输延时+40 位的 EOF 偏移）。

注意：对于高速的 EOF2 的确定来说，除了要考虑 38 位时间的偏移之外，还必须加上 16 位时间的量化误差，也就是说 EOF2 最少应该位于 EOF 之前的 54 位时间。EOF2 被置为 EOF3 之前的 64 个位时间（Bit times）是为了允许在位时钟（Bit clock）被 16 分频

之后的时钟环境下对错误做出检测。所有在 EOF1 前的数据包的结尾都必须在主机或 Hub 的 EOF2 时间之前到达。也就是说，所有动作必须在 EOF 之前的 544 个位时间前结束。

（2）全速的 EOF1 和 EOF2 定时点

全速模式下的 EOF2 被定义在包开始标志（SOP）的第一位到来之前的一个位时间处。一个 EOP 在全速下的允许时间为 4 个位时间。

同高速模式一样，虽然 Hub 已经经过 SOF 取得同步，时间的偏移仍可能在主机和 Hub 间或 Hub 与 Hub 之间进行积累。这一时间偏移代表了不同的 Hub 和 Host 之前取用了不同的帧计时时间，这一偏差最大可能达到±9 个位时间。其中包括±1 个位时间的每一帧量化误差和±1 个位时间的每一帧游移误差。量化误差的概念读者可在相关书籍中查找。帧的游移误差发生在主机的帧计时器被 USB 的系统软件进行调整的时候，此时，Hub 在前一帧中的采样值与主机中实际执行值不同。由于 SOF 包可能被丢失，所以，这样的偏移可能在多个帧间进行积累。USB 规范规定最多允许两个 SOF 包的丢失，因此，累积后的数值如下：

量化误差：±3 位时间

偏移误差：±1±2±3=±6 位时间

总共加起来为±9 个位时间（经过三个帧时间），所有的这些偏差就将影响到 EOF1 和 EOF2 点的确定。

在帧结束前的一个位时间内，Hub 必须到达其 EOF2 点。为了确保这一点，必须在其中加入 9 个位时间的警戒时间，所以，当本地帧计时器递减到 10 时，EOF2 便会出现。Hub 必须在其到达 EOF2 点之前完成其 EOP 过程。

3.4 主机在帧结束时的行为

在帧计时的 EOF2 时间处，主机不应再向设备发出请求响应的命令，而且由于在到达 EOF1 时间点后，Hub 将停止任何传向上游的数据包。因此，主机应充分考虑需要设备响应的事务处理时间，如在帧计时到达 EOF1 前不能完成的话，则不应发起该事务。

由事务处理所要花费的时间可以通过对不同部分产生的延时的总体估算而得到。

3.4.1 全/低速模式下最晚发出的主机数据包

在 EOF1 时间中，如果在 Hub 中没有流向下游的数据，那么，Hub 可以向其上游端口发出一个 EOP 信息。为了避免潜在的危险，如果主机在 Hub 到达 EOF1 之前仍未能建立起传送数据包所需的必要连接，那么，这样的数据传送是被禁止的。这意味着主机必须在帧开始后的 42 个位时间之前开始数据的传送。

3.4.2 全/低速模式下的无效包

如果在设备送出数据包之时，位于其上游通路中的某一 Hub 已经到达了其 EOF1 点，那

么，这个 Hub 将会发出全速 EOP 命令。这样的话，任何被 Hub 接收了的数据包都应被丢弃。

主机的进程会丢弃第 41 个位时间处接收到的数据包。但是，对于一个事务来说，如果在第 41 个位时间已经收到了数据包开始的标志，主机便会尽最大可能利用总线来接收数据。

3.4.3　全/低速下对事务处理完成时间的预测

由于对帧计时的严格要求，因此有必要建立一套完整的机制来对所处理的事务进行预测。

一个设备可能送出两种数据包：数据信息和握手信息。握手信息的数据包通常为标准的 16 位大小（同步字节加上 PID 字节）。从主机发出结束请求包，到接收到相应的握手信号的头一位之间，允许的时间间隔为 17 个位时间。如果从开始试图建立握手连接算起，到有信号从设备返回的时间间隔为 35 个位时间。因此，如果主机送出的数据包能够得到设备的握手信号，而且如果主机在第 76 位时间之前完成发送该数据包并开始 EOP 过程的话，那么，主机便应能够在第 41 位时间之前预测到设备是否能够完成握手连接并开始 EOP。

如果主机已经在第 76 个位时间时送出了需要设备响应的数据包，那么，它应采取一系列非常手段来确保设备不会试图发出响应。那就是在 EOP 之前发出 7 位的连续信号“1”，这样做的目的是使该设备端收到非法的接收信号来中止连接。

那么，主机究竟是如何计算所需的数据传送时间的呢？一般可用下式来计算，对于一个全速的数据包，从 IN 标志开始送出，到数据包的结束所需时间：

token_length + (packet_length + header + CRC)×716 + 18

token_length：34 位时间

packet_length：数据包中包括的数据位数。

Header：8 位 sync + 8 位 PID

CRC：16 位

×716：为了满足数据包中最坏情况下的填充位。

+18：最大允许的延时误差。

对于低速模式，计算方式一致，但要在结果上乘以 8 来转换为相应的全速的位时间，并且要在其上再加上 20 个全速的位时间作为前缀，那么，在得到该值后，主机便应考虑在 EOF1 之前是否还有足够多的位时间用以数据传输。

对于计算公式，可能会有不同的方式，但总的来说，都应在规定的时间之前完成对所需握手信号的传送及数据传送时间的计算，否则的话，数据的传送便不应发起。

3.5　内部端口

Hub 的内部端口是建立 Hub Controller 和 Hub Repeater 之间联系相关部分，它的功能包括：将串行数据传递给 Hub Controller 或从其中传出。另外，它还负责发出特定的重续信号。

内部端口的状态机如图 3-9 所示。内部端口发出的信号和事件如表 3-4 所示。

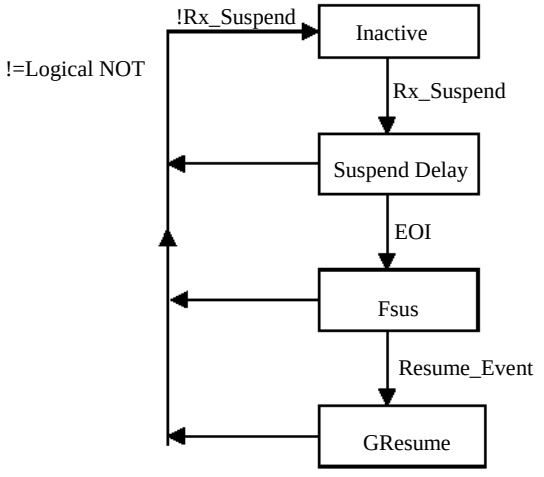


图 3-9　内部端口状态机

表 3-4　　内部端口信号/事件的定义

信号/事件的名称	来源	描述
EOI	内部	定时间隔的结束
Rx_Suspend	内部	接收器处于挂起状态
Resume_Event	Hub Controller	Hub Controller 中存在重续的环境

下面对内部端口的状态机的各部件功能作一简要介绍。

1. 停止状态（Inactive）

在任何时候只要接收器不处于挂起状态（Suspend State），端口便会进入该状态。

2. 挂起延时（Suspend Delay）

当接收机从非挂起状态转入挂起状态后，端口便会进入挂起延时的状态。通常该状态被规定为 2ms。

3. 全部挂起（Full suspend——Fsus）

在挂起延时结束后进入该状态。

4. 重续的产生（Generate Resume——GResume）

当 Hub Controller 内存处在可进行重续的环境时，端口便进入该状态。只要在任意端口的 C_PORT-SUSPEND 标志位被置位后，或 Hub 作为一个唤醒源可以起作用后，重续的环境便会存在。同时，在端口变换域或集线器变换域的任意数据位被置位后，同样会产生重续的环境。在这一状态，内部端口将产生类似于 SOP_FD 的信号，并将其发送至 Hub Repeater。

3.6　下游端口

集线器的下游端口用来提供给下游设备的电气连接环境。下游端口所要经历的各个状态如图 3-10 所示。各状态和信号的定义如表 3-5 所示。

表 3-5　　下游端口的信号/事件定义

信号/事件名	来源	描述
Power-source-off	具体应用	给端口提供电源的设备已不可用或被移除
Over-current	Hub Controller	在 Hub 或 Port 上过流
EOI	内部	定时的序列结束
SEO	内部	SEO 信号被接收到
Disconnect_Detect	内部	端口的连接被断开
LS	Hub Controller	低速设备连入端口
SOF	Hub Controller	接收到 SOF 包
True RWU	内部	K 状态持续一个 TDDIS
PK	内部	K 状态持续一个 TDDIS

续表

信号/事件名	来源	描述
PS	内部	SEO 将持续一个 TDDIS
K	内部	端口收到 K 状态
Rx_Resume	接收器	上游接收器处于重续状态
Rx_Suspend	接收器	上游接收器处于挂起状态
Rptr_Exit_WFEOPFU	中继器	中继器脱离 WFEOPFU 状态
Rptr_Enter_WFEOPFU	中继器	中继器进入 WFEOPEU 状态
Port_Error	内部	发现错误
Set Test	控制器	对 SetPortFeature(Test_SEO_NAK), Setportfeature(Test_J), SetPortFeature(Test_K), SetPortFeature(Test_PRBS), SetPortFeature(Test_Force_Enable)进行逻辑的“或”运算
Configuration	控制器	控制器的设置值为“0”

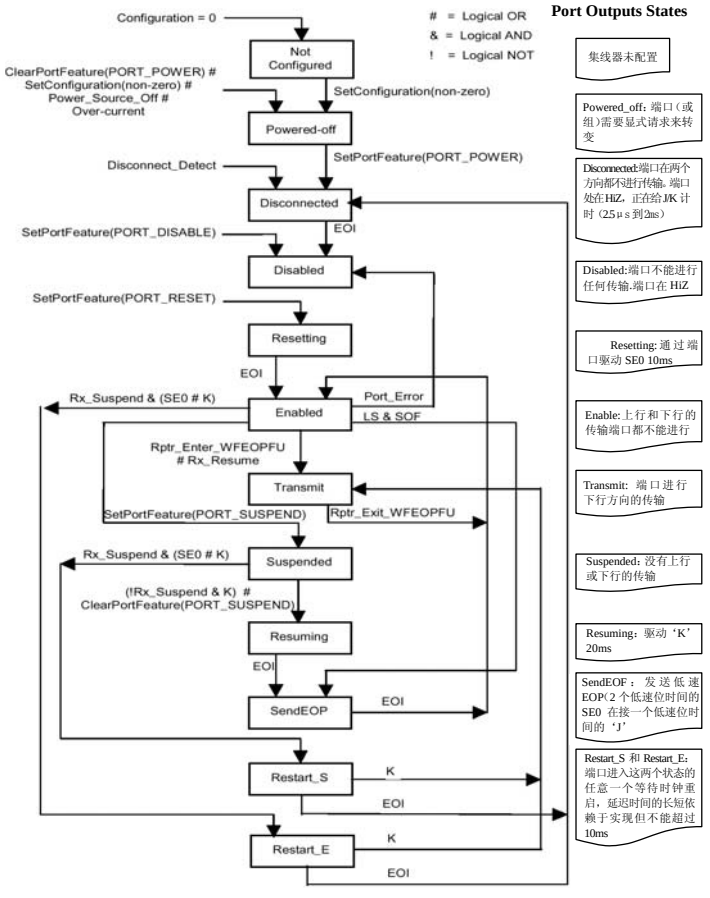


图 3-10　下行集线器端口状态变迁图

每个端口需要一个未连接计时器。该计时器用来连续监视端口的单端接收器，并检测未连接的事务。

3.6.1　下游端口状态的描述

1. 未配置（Not Configured）

当 Hub 设置为“零”时，端口将进入并保持在该状态。端口处于该状态时不会发出其他有效信号。

2. 断电（Powered-off）

这一状态对所有 Hub 都有效。下列任何一个事件发生后，端口便会进入该状态。

- 除了未配置状态，只要 Hub 从其他状态接收到来自于某一端口的 ClearPortFeature（PORT_POWER）。
- 当 Hub 接收到 SetConfiguration()请求时，且配置值为非零。
- 除了未配置状态，只要在任何状态，端口掉电或过流。

如果 Hub 是自己提供电源，那么，一旦该电源掉电，Hub 必须进入 Powered-off 状态。如果 Hub 被设为从总线供电，在当前电源失电时，Hub 可以切换到外部电源。然而，当外部电源也失电时，Hub 则不得不进入 Powered-off 状态。

3. 连接的断开（Disconnected）

在下列任何一种情况下，端口进入该状态：

- 在 Powered-off 状态，Hub 接收到了一个 SetPortfeature(PORT_POWER)请求。
- 除了 Not Configured 和 Powered-off 状态，在其他任何状态端口的断开连接定时器计时完毕。
- 在重启过程中的末尾的 Restart_E 和 Restart_S 状态。

在断开连接状态，该端口的发送和接收都是被禁止的，只有侦测连接的活动可能发生。这是一个定时状态，在这一状态下，定时器只要发现端口的信号处于 SE0 或 SE1 状态时便会重置。该状态下，如果检测到了另一个信号状态，计时便会开始，并持续 2.5μs~2ms，除非由于时钟停止 Hub 被挂起。

处于挂起状态的 Hub 如果具备了远端唤醒功能，已断开的连接的端口在转移到其他状态的过程中（除了 SE0、SE1），Hub 都会启动它的时钟，并开始对该事件计时。在状态转换的 12ms 内，Hub 的计时就应该开始。如果远端唤醒功能被禁止，那么，在该端口的任何状态转变都会被忽略，直到 Hub 被重启。

4. 禁止状态（Disabled）

在下列任何一种情况下，端口都将转入禁止状态。

- 在 Disconnected state，当计时器计时完毕，暗示端口有连接。
- 在其他状态（除了 Powered_off，Disconnected，Not configured）接收到了 ClearPort-Feature（PORT_ENABLE）请求。
- 在 Enable 状态，端口发现了错误的环境。

在该状态，端口不会向上游或下游传递信号，如果端口处于高速模式，那么在进入该状态后，它会转入全速模式。只有在进入该状态 4ms 之后，端口才应试图执行通常的断开连接的检测。

5. 重置（Resetting）

在端口接收到 SetPortFeature（PORT_RESET）时（Power_off、Disconnected 状态下除外），端口便会进入重置状态。

6. 使能（Enabled）

在下列任何一种情况下，端口转入使能状态：

- Resetting 状态结束。
- 当 Hub Repeater 脱离 JWFEOPFU 状态时，处于 Transmit 或 TransmitR 状态。
- 在挂起状态下，如果上游中继器也处于挂起状态并且在端口检测到“K”状态时。
- 进入 Start_E 状态后，没有检测到持续 900μs 以上的“K”或“SE0”状态。

在该状态下，端口的差分接收器对于 Hub Repeater 是可用的。因此，合适的信号传输可以建立起向上游的连接。

7. 传送（Transmit）

在 Hub Repeater 进入 WFEOPFU 状态时，端口从 Enable 状态进入该状态。在该状态下，端口将传送从上游端口接收到的数据。

低速下，端口在收到 PRE_PID 信号后进入该状态。高速下，该状态用以检测端口的连接是否断开。

8. R 传送状态（TransmitR）

在下列情况下，端口进入 R 传送状态：

- 上游的接收器处于重启状态时，端口从 Enabled 状态进入该状态。
- 如果端口检测到了 PK 信号，则从 Restart_S 或 Restart_E 状态进入。

9. 挂起（Suspended）

端口进入挂起状态的条件如下：

- 在 Enable 状态下接收到了 SetPortFeature（PORT_SUSPEND）请求。
- 在 Restart_S 状态下，未见到持续 900μs 以上的 K 或 SE0 信号。

端口处于挂起状态时，差分信号的传送是被禁止的。高速端口将其高速的活动转为全速。进入该状态 4ms 后，端口才能试图断开连接。

10. 重启（Resuming）

下列情况下，端口进入重启状态：

- 在接收器未处于悬挂状态时，端口检测到“K”信号持续 2.5μs 以上。
- ClearPortFeature（PORT_SUSPEND）请求被收到。

该状态是一个定义为 20ms 的定时状态，该状态下，Hub 在端口上发出“K”信号。

11. 发送 EOR（Send EOR）

端口进入 Resuming 状态后计时完毕，或在 Enabled 状态下当 SOF 信号被接收到而且有低速设备被连入端口。该状态是一个定时为 3 个低速位时间的状态。

- 高速下：端口将在该状态下将总线驱动到空闲状态持续 3 个位时间，然后进入 Enabled 状态。
- 全/低速下：端口持续两个低速位时间的 SE0 信号和一个位时间的空闲状态，然后转入 Enabled 状态。

12. Restart_S

在 Repeater 处于挂起状态时，端口接收到 SE0 或“K”信号后，从挂起状态进入该状态。

该状态下，端口一直会监测总线的活动。在进入该状态 900μs 以后，将根据不同的信号端口转入不同状态。如未收到任何信号，端口便转入挂起状态。

13. Restart_E

在接收器处于挂起状态时，如果在端口接收到了“SE0”或“K”信号，则端口从 Enabled 状态转入该状态。

同样，该状态下端口仍要监测总线活动，900μs 之后将根据不同信号进入不同状态，否

则返回 Enabled 状态。

14. 测试（Testing）

在任何状态下，端口收到 SetTest()请求后进入该状态。

在该状态下，端口执行主机命令。

3.6.2 连接断开的侦测

1. 高速模式下断开连接的信号

工作在高速模式下的下游端口的收发器是通过对 D+和 D-信号线之间电平幅度的差值再乘以 2 后的结果进行比较，来判断端口的连接是否已断开。两条信号线在设备移除后，便会产生不同的信号幅度。

下游端口连接断开时，下游端口的收发器发出的信号得到的响应将是在原信号基础上加入一个额外的相位，基于此情况，专用的检测连接断开的探测器便会输出一个高电平。当信号幅差≥625mV 时，便会激活“断接探测器”，在其幅差小于 625mV 时，则不会激活该探测器。

2. 全/低速的连接断开的检测

全/低速设备连入端口时，每一个端口都会激活一个等同的用来检测连接是否断开的定时器。这一定时器用来不断地监测端口的单端点接口机，从而对断开和连接行为做出检测。

这个定时器在以下状态会重置：

- 端口的 D+和 D-线不在 SE0 或 SE1 状态。
- 端口未处于 Enabled、Suspended、Disabled、Restart_E、Restart_S 状态。

同时，该定时器在端口进入 Suspended 和 Disabled 状态时被置为计时 4ms。

通常，定时器计时时间为一个 TDDIS（一个 TDDIS 时间通常为 2.0μs~2.5μs）。当定时器计时完毕，它便会发出 Disconnect-Detect 信号到端口的状态机中。

3.6.3 端口指示灯

Hub 的每一个下游端口都有一个状态指示灯，该指示灯由 Hub 类描述符 wHubCharacteristics 第七位来定义。

该指示灯有两种颜色：绿色和琥珀色。由 Hub 硬件和软件所共同控制，用来为使用者提供有用的指示信息。指示灯在端口处于不同状态时的指示如表 3-6 所示。

表 3-6　　端口指示灯的状态图

电	下游端口状态

源切换	掉电	Disconnected Disabled Not Configured Resetting Testing	Enabled Transmit TransmitR	Suspended Resuming SendEOR Restart_E Restart_S
有	关闭或琥珀色（取决于电流状态）	关闭	绿	关闭
无	关闭	关闭或琥珀色（取决于电流环境）	绿	关闭

对于用户来说，不妨以表 3-7 所示的内容来理解指示灯所给出的提示。

表 3-7　　端口指示灯的定义

颜色	定义
关闭	无操作
琥珀色	错误环境
绿色	工作状态
闪烁的绿色	软件的问题
闪烁的琥珀色	硬件的问题
交替两种颜色的闪烁	保留状态

3.7　上游端口

上游端口具有 4 个组件：发送机、发送状态机、接收机和接收状态机。发送机和发送状态机组成了发送器，接收机和接收状态机组成了接收器。两者都可以工作于高速或全速模式下，这取决于当前 Hub 的设置。

3.7.1　全速

发送机和接收机都有差分和单端点的组件，差分的发送机和接收机可以向总线发送或从总线接收“J”和“K”信号，单端点的组件用来发送/接收 SE1、SE0、挂起和重续信号。如果有必要对两者的信号加以区分的话，则“DJ”和“DK”用于指示差分信号，而 SF、SK、SE0、SE1 用于指示单端点信号。

3.7.2　高速

发送机和接收机中都有差分部件，它们的信号被称为“HJ”和“HK”。

为了使电源的消耗降到最低，一般设定差分的发送机和接收机在挂起状态会被关闭，而单端点部件则会一直开启。

3.7.3　接收器

接收状态机用来对上游端口送出的长期性的信号事件做出监测响应，例如，总线的重置、复位和挂起。

在不同操作速度的模式下，会有不同的状态信号来表示接收器的工作状态。

3.7.4　发送器

发送器的状态机是用来在 Hub Repeater 连入上游方向时监视上游端口的，监视的目的是为了发现并阻止上游的错误信号。另外，还负责阻止那些下游端口发出的可能引起 Hub 停止工作或断开连接的信号和事件。

3.8　集线器中继器

集线器中继器（Hub Repeater）执行以下功能：

- 在数据包的边缘建立或断开连接。
- 确保 Hub 正确地进入和退出挂起状态，包括对远端唤醒的合适的控制。

1. 高速数据包的连接

高速中继器必须对双向的数据包进行重新定频（Reclock）。所谓重新定频是指中继器对从接收数据流收到的数据进行解压，然后再用自己的本地时钟进行打包发送。这样做是为了将各方面所引起的抖动保持在一个接收机可允许的范围之内。

2. 集线器中继器状态机（Hub Repeater State Machine）

集线器中继器状态机给出了集线器中继器的正常工作所需的状态流程。

3.9　集线器控制器

集线器控制器（Hub Controller）的逻辑结构如图 3-11 所示。下面将逐一介绍集线器控制器的功能。

3.9.1　端点的组织结构

除了默认的控制管道之外，集线器类中还定义了一个附加的端点：状态改变端点。

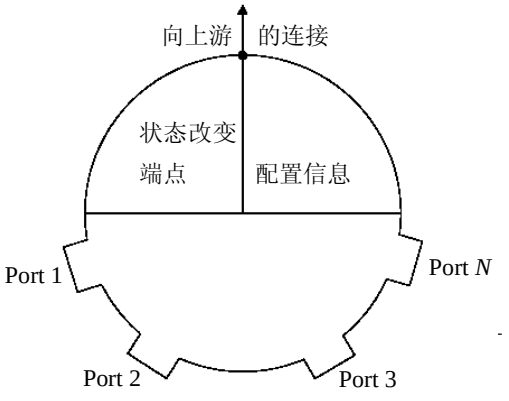


图 3-11　集线器控制器的内部结构

主机通过该端点来接收端口和集线器的状态改变通知。状态改变端点是一个中断端点。如果没有集线器或端口的状态改变位被置位，则 Hub 会依据相应的该情况返回 NAK 信息；如果有某一状态位被置位，则 Hub 便会以相应的数据做出响应来指示是哪个部分的状态位被置位。

3.9.2 Hub 的消息体系结构和操作

设备状态、状态的改变和控制信息与设备状态的关联如图 3-12 所示。

Hub 的描述符、Hub/Port 的状态和控制信号都是从默认的控制管道中得到的。Hub 的描述符不能在任意时候被要求读取。只有在 Hub 检测到端口的变化或是 Hub 改变了本身的状态时，状态改变端点便将特定规格的数据传给主机从而起到通知的作用。Hub 或 Port 的状态改变位可以通过硬件或软件置位。当被置位后，这些位便保持该状态直到被 USB 的系统软件通过 ClearPortFeature()请求或是 Hub 的重启而清除。只要有一个状态改变位被置位，Hub 便会持续不断地发出状态改变的报告，直到所有的状态改变位都由系统清除为止。USB 系统软件使用中断管道来配合状态改变端点，用以检测 Hub 或 Port 的状态改变。

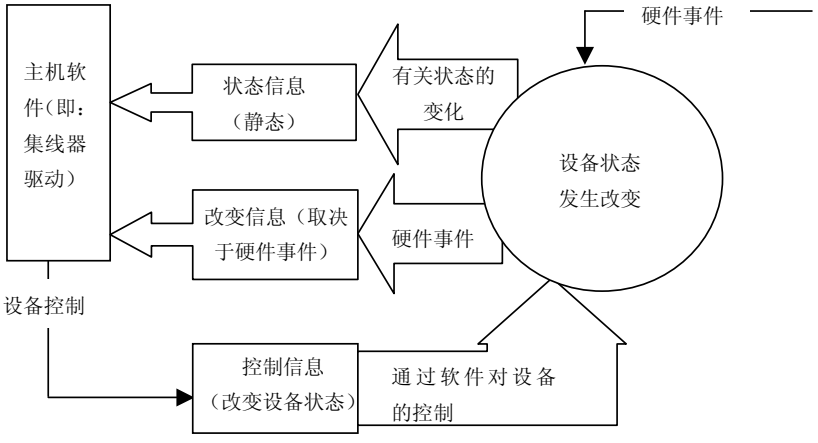


图 3-12　与设备有关的状态、状态改变和控制信息

3.9.3　端口改变信息的处理

Hub 通过基于每一个端口的命令来报告端口的状态。USB 系统软件通过清除相应的位来响应状态改变的通知。Hub 对于端口改变信息的处理过程如图 3-13 所示。

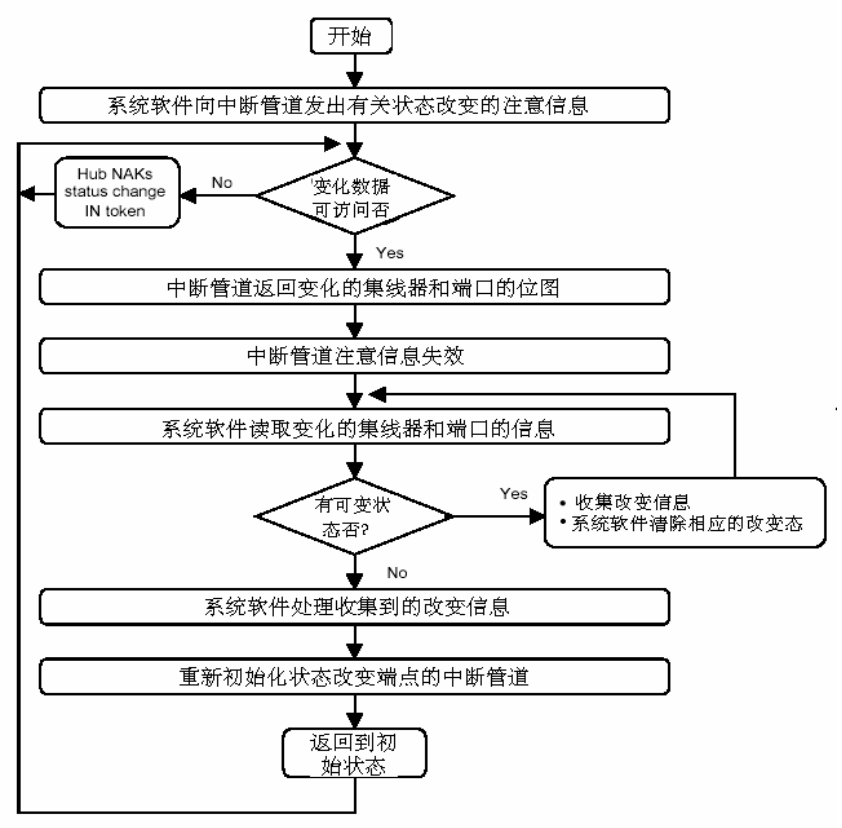


图 3-13　端口状态的控制机制

3.9.4　集线器和端口的状态改变位图

Hub/Port 状态改变的位图如图 3-14 所示，位图的改变是通过一个字节的值来反映的。位图不仅给出了 Hub/Port 是否有过状态改变，而且通过状态改变端点来反映状态值。Hub 通过逐字节增加的值来做出报告，例如：一个 Hub 有 6 个端口，那么，它会返回一个字节的值，字节中的每位代表一个端口，对于那些无效的端口位，则以 0 来代替。USB 的系统软件会注意到在一个 Hub 中端口的数量（这会在 Hub 描述符中得到），然后对相应的 Hub 和 Port 状态位图做出解码。只要有一个状态改变位不为零，Hub 和端口的状态改变位图就会生效，如图 3-15 所示。

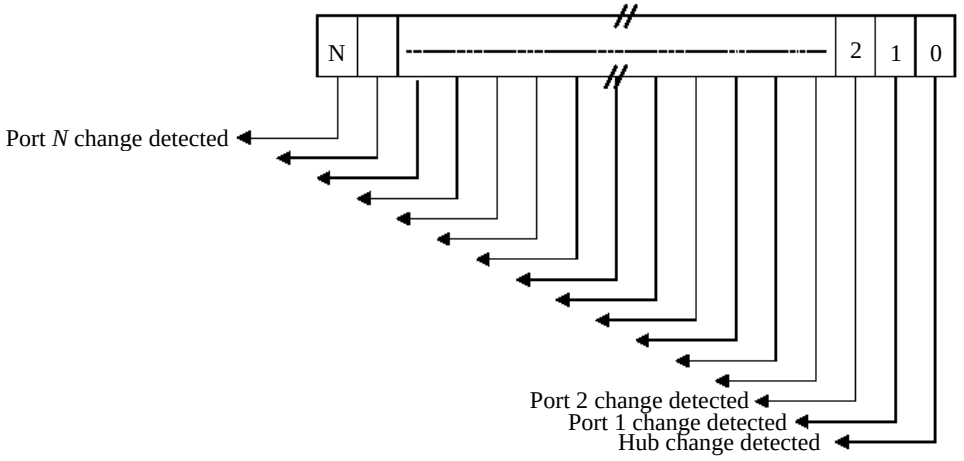


图 3-14　集线器和端口状态的改变位图

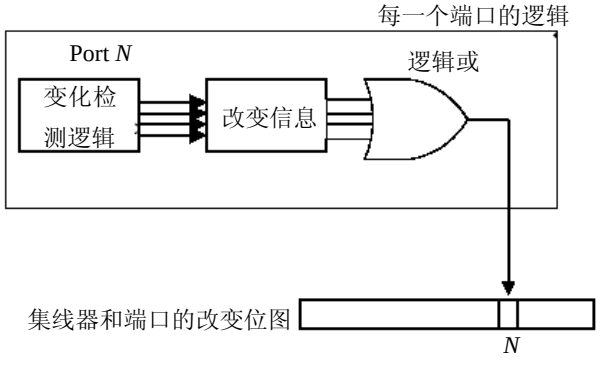


图 3-15　集线器和端口改变位的采样示例

3.9.5　过流报告和恢复

USB 设备必须要满足一定的安全运行要求。对于 Hub 来说，如果是 Hub 自我提供电源的话，那么它便会限制其下游端口的电流量。如果有过流的现象发生，那么便会引起一个或多个端口的工作不正常。这一变化往往要报告给 USB 主机以便采取相应的纠正措施。

Hub 可能被设计为要对 Port 或 Hub 的过流现象做出反应。Hub 描述符域中的 wHubCharateristics 用来对 Hub 的该种能力做出描述。

当 Hub 处于过流状态时，它必须将相应的端口置于断电（Powered-off）状态。如果 Hub 有对于每一端口的电源切换和电流限制的功能，那么一个过流的端口可能会引起其他的端口电源供应的不足。在这种情况下，出差错的端口将被置于断电状态，该端口的 C_PORT_OVER_CURRENT 位被置位，而 PORT_OVER_CURRENT 没有被置位。

如果 Hub 具有基于单端口过流检测的功能，那么任何一个端口的过流，都会导致所有端口进入断电状态。在这样的情况下，上述的两个标志位都不会被置位。

Hub 对于过流事件应采取的动作包括：

- 主机从 Hub 得到过流事件的通知。

- 主机对合适的 Hub 和 Port 的改变信息做出解压。
- 主机等候过流状态位被清零。
- 主机对所需的端口做出循环供电，也就是说，逐一发出 SetPortFeature（PORT_POWER）请求到每一端口。
- 主机对所有端口进行重排序。

3.9.6　对于设备检测的控制

Hub 的设备类命令被用来掌握其下游端口的状态。当一个设备被连入后，该事件被 Hub 检测到并通过状态改变端点发出中断报告。主机收到该报告后向该端口发出 SetPortFeature（PORT_RESET）请求，在总线的一部分重置后，由 Hub 的端口硬件做出速度的检测。

在主机随后发出 Get_Starts（PORT）请求后，如果端口连入高速设备，则 Hub 以“Not PORT_LOW_SPEED and PORT_HIGH_SPEED”信息响应。

如果连入低速设备，则以“PORT_LOW_SPEED”信息响应。

如果连入全速设备，则以“Not PORT_LOW_SPEED and Not PORT_HIGH_SPEED” 信息响应。

当设备从端口撤离时，端口通过状态改变端点做出状态改变的报告，然后该端口会被重新连入高速中继器。端口接着会等待下一个设备的连入。

3.10　集线器的设置

集线器通过标准的设备配置命令来对其进行设置，USB 的系统软件通过检查 Hub 描述符中的信息来决定 Hub 的特征，通过对 Hub 特征的检验，使 USB 系统软件能知道 Hub 的端口是否建立了合适的拓扑关系。

通过设备的状态和配置信息，还可以决定 Hub 是通过总线供电还是自身供电。Hub 供电方式的总结如表 3-8 所示。

自我供电的 Hub 自己带有一个本地电源，但同时它也可以配备一个从上游端口来的电源供应单元。这便使得当本地电源不可用时，接口仍然能够通过上游电源工作。当本地电源被移除后，Hub 会将所有端口置于断电状态后，进入等待设置状态。此时，所有端口的状态信息都为“零”，并忽略所有的 SetPortFeature()请求。Hub 通过状态改变端点来通知 USB 系统软件所发生的事件。

MaxPower 域是用来描述 Hub 可以从总线得到多少电力供应的描述符。对于总线供电的 Hub，该值必须排除外部下游端口的电力需求。外部设备会报告它们自己的电力需求。

表 3-8　　Hub 供电操作模式总结

配置描述符		集线器设备状态（自我供电）	注释
Max Power	bmAttributes（Self_ Powered）		
0	0	N/A	非法设置信息

0	1	0	设备自我供电，但并无可用电源
0	1	1	Hub 自我供电且电力供应正常

续表

配置描述符		集线器设备状态（自我供电）	注释
Max Power	bmAttributes（Self_ Powered）		
>0	0	N/A	Hub 通过总线供电，下游端口可能未被供电。Hub 设备状态位无意义
>0	1	0	Hub 可工作于两种供电模式，当前只能工作于总线供电模式
>0	1	1	Hub 可工作于两种供电方式，当前只能工作于自我供电模式

复合设备可能对 Hub 本身和固定连入 Hub 的设备一起供电。所有这些都要通过 Hub 的设置描述符来得到相应的信息。每一个固定部件都应报告其自身的电力需求。

3.11　事务处理转译器（Transaction Translator）

集线器工作于高速模式而下游端口接入全/低速设备时具有特殊的响应规则。在这种情况下，集线器必须能将高速的信号环境与全/低速的信号环境隔离开来。这一功能便是由 Hub 的 Transaction Translator（TT）来完成的。

3.11.1　综述

Transaction Translator 的结构示意图如图 3-16 所示。

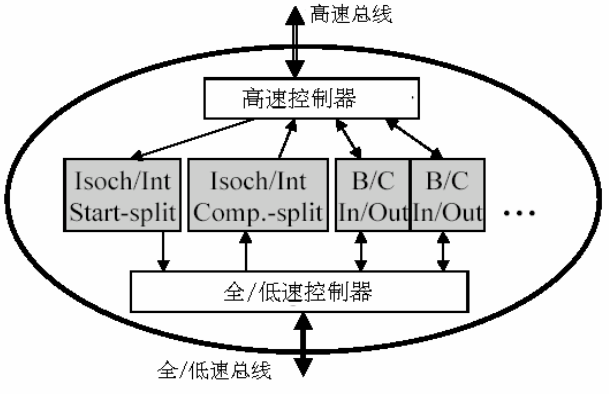


图 3-16　事务转译处理器结构

TT 从其上游端口的高速总线接收到高速的分割事务，然后向下游端口的低速设备发送相应的低速模式的事务。在高速总线上的 TT 运行于高速模式，而在下游全/低速端口看来，TT 则扮演着主机控制器的角色。因此，在 TT 内就有了高速和全/低速的两个不同的控制器。

TT 拥有自己的缓冲区，用于保存数据处理过程中的事务并对其进行状态上的跟踪，同时，缓冲区提供了高速和全/低速控制器之间的连接。对于不同的数据传送方式，TT 有着不同的

状态跟踪方式。

- Start-split buffer：高速控制器将开始分割的事务置入本地的缓冲区，等候全/低速的控制器来调用。事务将会提供必要的信息给全/低速的控制器，使它能将相应的事务发送给下游端口的全/低速设备。

全/低速的控制器能够将事务处理的结果进行缓冲（Comp-split），以便能够对高速的数据总线做出数据分割完毕的响应。

- TT 有两个状态跟踪的缓冲：（如图 3-17 所示）周期的（用于同步/中断模式的全/低速事务处理）和非周期的（用于批量/控制模式的全/低速事务处理）。

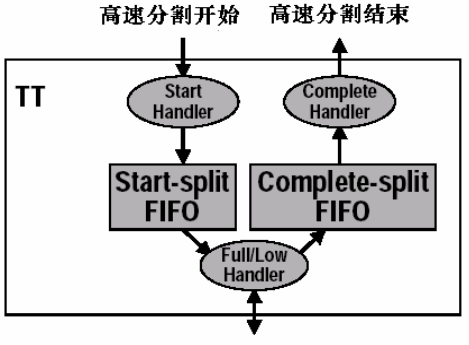


图 3-17　对于周期分割事务处理的 TT 微帧管道

1. 主机控制器和 TT 的事务分割处理

主机控制器通过 TT 使用事务分割协议来开始全/低速的事务处理，然后决定事务处理完毕后的标志状态。这样就能使主机控制器在将全/低速的事务分割事务提交后，无需等待处理的完成就能够转入其他高速事务的处理。

事务的分割处理只能应用于主机和 Hub 之间。事务分割由两部分组成：分割的开始和分割的结束。其他设备并不能应用事务分割处理机制。

当主机控制器在高速模式下开始了事务的分割处理后，这一事务将会定位到特定设备的 TT 地址，TT 接收并将其缓存。高速控制器对接收到的事务与主机控制器进行合适的握手，但并不是所有的事务分割的开始都有握手过程。开始分割的事务将暂时保存在 TT 的事务处理缓冲区中。

全/低速的控制器对周期性分割事务进行处理的前提是全/低速的控制器对接收下一个分割处理事务做好了准备。全/低速的控制器从下游总线接收结果信息，并将其存入当地的缓冲，用于将来传给主机控制器。

在此后的某时刻，主机控制器将会发出一个高速数据分割完毕的信号来找回相应的状态/数据/结果信息。高速主机控制器依次根据本地事务处理缓冲中的信息来检查这一高速事务处理是否完成。

2. 多重事务处理器

对于事务分发转译处理的组织，Hub 有两种选择，一种是 Hub 用一个 TT 来对所有连入下游端口的全/低速设备进行处理；另一种是对每一个下游端口都使用一个 TT。在 Hub 的类描述符中，Hub 必须报告其组织结构。

3.11.2　数据处理时序

作为高速控制器，它要开始分割事务，接收全/低速数据信息和数据，将其存入缓冲区，并且等候下游总线对所有这一切的响应。主机对于周期的 TT 缓冲和非周期的数据处理缓冲采用不同的管理方式。

1. TT 同步/中断/周期性的事务处理的缓冲

周期性的事务处理有着严格的时序要求以满足全/低速的总线需求。也就是说，事务处理必须首先经过高速总线，穿过 TT，经过全/低速总线；然后再返回 TT，并适时进入高速总线。

高速总线上的数据处理以开始事务分割为起始点，然后被高速控制器接收，接下来存入分割事务缓冲区。全/低速的控制器在对相应的分割事务进行处理时，读取相应的分割事务，而事务处理的结果将保存在完成分割缓冲区中。TT 对主机发出分割完成的请求做出响应，然后从完成分割缓冲区中解压出相应的响应信息。

所有这一切才组成了一个完整的周期类的事务处理，而这一过程被称为周期性的事务处理流程。

TT 在习惯上利用其周期类事务处理缓冲器来完成其处理流程，其中包括了独立的开始分割和完成分割数据的缓冲。主机负责填充开始分割缓冲数据，并取出相应的完成分割缓冲中的数据。主机软件控制主机控制器，使其在合适的时候发起高速事务的分割。主机要选择合适的时间对两个缓冲中的数据进行收发，以使缓冲的占有率降到最低。

USB 对于周期性事务处理和总线同步传输的能力有着严格的定义，这便使得主机能够准确地预测什么时候应将周期类的事务的数据送入总线。

2. TT 批量/控制（非周期）事务处理的缓冲策略

非周期的事务处理没有时序的要求，TT 支持最大的全/低速的数据吞吐量。

主机和 TT 使用简单流通控制机制来管理批量/控制的非周期类的事务处理的缓冲。主机发送开始分割事务，如果有可用的缓冲空间，TT 接受该事务。全/低速的控制器使用缓冲中的信息来保存结果。在对主机发出的结束分割请求响应过程中，缓冲被清空。

图 3-18 所示为一个具有两个批量/控制缓冲的 TT。

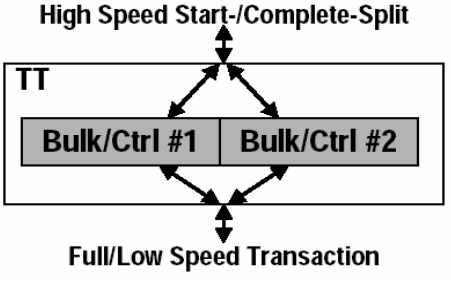


图 3-18　TT 中非周期的缓冲策略

3. 全/低速的事务处理控制器的时序

全/低速的控制器使用单一的、预定优先级的状态机机制来对挂于下游总线的事务进行处理。只要全/低速控制器完成了一个下游总线上的事务处理，它便会试图开始从缓冲区中取出下一个排队中的周期性事务来进行分割处理，如果没有周期性的事务可处理，控制器便会试图开始下一个非周期类的事务分割处理。

如果队列中有等候的事务，并且在全/低速的 1ms 的帧中有充足的时间来完成处理，那么

全/低速控制器便会开始发起一个批量/控制传输事务的处理。全/低速控制器对事务进行分割的算法的伪代码如下：

```
While (…) loop
    While (not end of microframe) loop
    …… Process next start-split transaction
    If available periodic start-split transaction
    Then
        Process next full-/low-speed periodic transaction
  Else if (available bulk/control transaction) and
        (fits in full-/low-speed /ms frame) then
  Process one transaction
  End if
End loop
  Advance-pipeline();
End loop
```

正如在本章中前面所述的那样，TT 从下游总线上得到的 1ms 的 SOF 定时是来自于高速的 1ms 的帧，这意味着主机和 TT 有同样的 1ms 的帧时间用于所有的 TT。由于在帧和微帧的起始之间建立了严格的联系机制，因此，就没必要在周期性的 TT 事务处理中加入另外的时间限制信息。

3.12 本章小结

集线器是 USB 设备能够实现即插即用的硬件基础，它的概念和工作原理既与我们通常所说的集线器有相似之处，又有其不同的方面。相信读者在学习完本章内容之后对两者的异同有了一个比较清晰的认识。在第 4 章中将讲述 USB 设备实现即插即用的另一个基础：主机对设备的检测过程的相关知识。

第 4 章　设备检测

知识点：

- 设备的检测
- 固件
- USB 总线上的各个状态
- USB 设备描述符

本章导读：

本章将介绍 USB 实现其即插即用功能的第二个部分：设备连入主机的检测过程，并结合 CYPRESS 公司的芯片，详细介绍如何实现这一过程。

为了能够使 USB 实现真正意义上的即插即用，规范的制定者们从多方面考虑到了设计的完整性与严密性。总的说来，USB 体系从以下 3 个方面对这一特性做出了支持：

- 硬件方面——USB 的集线器的支持。
- 主机对 USB 设备的检测策略。
- 设备与主机建立初始连接的基础——控制传输。

在本章中，将结合实例对主机对 USB 设备的检测策略作详细的介绍。

4.1　概述

在 USB 的体系中，不论是其本身的规范还是各种厂家所提供的芯片资料，关于主机对 USB 的设备检测都称为 Enumeration（and ReNumeration），汉语直译为列举（与重新列举）。

在第 1、2 章关于 USB 的概述中已经知道，在 USB 通信系统中，每一个 USB 的根集线器上不论层叠多少级，最大能够接入的设备仅为 127 个。同时，在 USB 的令牌包中包含了 7 个用来寻址的位，最大可以寻址 128 个设备。

由于所有的这些设备都要共享主机为 USB 所提供的同一端口的带宽。因此，必须有一个合适的机制来管理设备向主机提出的请求与数据传送。

在 USB 中，将地址为 00000000B 的端点 0 设为默认地址，所有初始连入 USB 接口中的设备都要先使用地址 0。在这一地址上，初始连入的设备要完成与主机间的多次握手过程，使主机能够顺利对其进行所需的配置，然后，主机会依次从端点 1 开始给其分配一个一一对应的端口。这样，主机在轮询 USB 接口时，便会依次检测这 128 个端点有无请求信号，这便是最初步的 Enumeration 的含义。

同时大多数的 USB 控制芯片都提供了一个控制位，通常称之谓“RENUM”（Renumerated），这一控制位给出了 USB 设备是处于初始的、未经设置的状态，还是处于已经经过主机的设置，转为正常的，已由其本身的固件所控制的状态的提示。

1. 设备检测进程中要完成的任务

总的来说，在设备的列举过程中，应该完成以下的基本任务：

- 给请求的设备分配一个地址。
- 从设备中读取所需的描述符。
- 加载相应的设备驱动程序。
- 根据设备所反映的数据来决定设备的最终配置。

此外，应该明确的是，只有在集线器的配合下，主机才能完成对设备的列举过程。集线器大体上应该由 3 大功能模块所组成：集线器中继器（Hub Repeater）、集线器控制器（Hub Controller）和事务处理转译器（Transaction Translator）。而在集线器控制器中，又可分为 3 大逻辑功能模块：状态改变端点、配置信息端点（端点 0）及各个端口。

这三部分正是与主机对设备进行列举关系最为密切的部分。

当一个设备被连入集线器后，该事件将被集线器检测到，然后它通过状态改变端点，以中断信息流的方式向主机提出报告。

然后，主机便会通过配置信息端口（端口 0）发送一系列请求，以便了解设备的特征描述，根据设备返回的信息，主机给予合适的配置。

当设备从端口撤离时，端口通过状态改变端口做出状态改变报告，然后端口会被重新连入高速中继器（高速模式下），等待下一设备的连入。

因此，可以看出，在集线器、主机、设备的协调配合下，才完成了设备的即插即用的功能。

2. 设备检测过程中的现象

一般来讲，列举过程中，设备与主机要完成两次握手和一次重置才能实现设备的真正连入。但是，在用户看来，也许只是 USB 设备灯闪烁了几下，显示器屏幕上出现了“正在检测 USB 设备”这样的字样等一系列现象，也就是说，整个列举过程对用户来讲是透明的。

下面我们将以赛普拉斯（CYPRESS）公司的设备为例，详细讲述关于主机对设备的列举过程。

4.2　FX 2 的设备列举过程

FX 2 是 CYPRESS 公司 USB 控制器家族中的一支，它是 EZ-USB 系列的延伸，专门用于 USB 2.0 类设备的控制器。

关于 FX 2 的系列芯片，将会在本书第 6 章中给予详细介绍。

4.2.1 FX 2 列举的特点

FX 2 的设置过程是软设置，所谓软设置是指：其程序代码和数据都存储在内部的 RAM 中，这些代码可以通过 USB 接口从主机下载，基于 FX 2 的外设可以没有 ROM、EPROM 或 FLASH 存储器。

为了支持这种软件设置，FX 2 有能力对一个内部没有固件的设备进行排序。在这一过程中，主机首先将 USB 设备置为一个“默认的 USB 设备”的状态。在该状态下，它会利用自己的接口和端点，并且能够完成从主机下载固件的过程。

实际上有两个独立的“默认 USB 设备”存在，一个用于全速下的排序，另一个用于高速下的排序。在设备连入后，FX 2 会自动执行速度检测协议，并选择合适的“默认的 USB 设备”。

一旦“默认的 USB 设备”开始排序，它便会从主机中将固件和设备描述符表下载到 FX 2 的片内的 RAM 中，然后，FX 2 开始执行下载的程序，它在这一过程中会通过电气的方式来使设备与主机完成物理上的断开以及再次的连接，也就是说，这是一个断开连接和重新连入的过程，对于用户来说是感觉不到的。而且 FX 2 会将设备进行再次的排序，在这一过程中，设备便会以下载的固件和描述符的特征来工作。这一特有的排序过程被称为“重排序”。

FX 2 的寄存器中，有一个专用的寄存器包含了一个“RENUM”位，当 RENUM=0 时，“默认的 USB 设备”在工作；而当 RENUM=1 时，则是设备的固件在工作。

4.2.2 FX 2 的启动模式

当 FX 2 上电之后，它的启动行为要由以下几个特征来决定：

- 如果没有片外的存储器连入 FX 2，那么，排序便首先以“默认 USB 设备”状态开始，列举过程中的描述符由硬件内部逻辑所提供（即 RENUM=0）。
- 如果由 EEPROM 连入 FX 2 的 SCL 和 SDA 引脚来提供当前的描述符，FX 2 同样以“默认设备”开始，但利用 EEPROM 中的描述符来取代“默认设备”中的值。EEPROM 必须在其地址空间的第一个字节中包含“0xC0”值以表示当前这种工作模式。因此，这种启动方式被称为“C0 Load”。同样，PENUM=0 时在 EEPROM 中应包含 16 个字节的地址空间用于“C0 Load”。
- 另一种情况，EEPROM 中含有固件并连入到了 FX 2 的 SCL 和 SDA 引脚，固件会自动地从 EEPROM 中下载到 FX 2 的内置 RAM 中，在 FX 2 复位上电后，便开始执行这一程序代码。在这种情况下，描述符被压缩到固件中，同时，RENUM=1，暗示是由固件来管理着设备请求。那么，在 EEPROM 中的第一个字节中存储的值此时则应为“0xC2”。所以，这种模式又称为“C2 Load”。
- 如果有 FLASH、EPROM 或是其他的存储器连入到了 FX 2 的地址数据总线上（只有 128 脚的 FX 2 可用），而且没有 EEPROM 连入。同时，如果 FX 1 的 EA 引脚被连入高电位，则表示 FX 2 开始从片外存储器的地址 0*0000 开始执行代码，也就是说，固件（包括压缩在其中的描述符）都存放在外部存储器，这时 RENUM=1。

4.2.3　FX 2 的“默认的 USB 设备”

在上一节中，我们提到了 FX 2 中的“默认设备”，这是 FX 2 的一个特有设置。为了使 USB 设备连接顺利过渡，它包含了一套简单的 USB 配置所需的软硬件，其中包括：一个接口，以及可变的 0、1、2、3 等 4 种设置方式。不同的设置方式会影响到 FX 的各端点缓冲的不同利用方式。

4.2.4　EEPROM 启动导入数据的格式

在本小节中将讨论 3 种 EEPROM 的启动方案，以及用来支持这 3 种启动方案的数据格式。3 种方案分别是：

- 无 EEPROM 或是无效的 EEPROM。
- “C0”EEPROM（只是装载 VID/PID/DID 值）。
- “C2”EEPROM（将整个固件装入片内的 RAM）。

1. 无 EEPROM 或是无效的 EEPROM

这是最简单的一种情况，正如前面几节所述：当在 FX 2 上没有 EEPROM 出现在其　串行兼容总线上，或是有 EEPROM 但其第一个字节既不是“C0”也不是“C2”时，便属于这种情况。

在这样的条件下，描述符的数据是由 FX 2 内部固化的数据表所提供的，FX 2 首先将其作为“默认设备”进行列举，其 ID 值如表 4-1 所示。

表 4-1　　FX 2 设备描述符（无 EEPROM）

Vendor ID	0×04B4（CYPRESS Semiconductor）
Product ID	0×8613（EZ-USB FX 2）
Device Release	0×XXYY（depends on revision）

USB 的主机会在列举过程中询问 FX 2 的默认的 USB 设备。读取其设备描述符，使用表 4-1 中的 ID 值来决定哪一个设备驱动应被装载入操作系统。

这也是 USB 即插即用能够实现的第一个重要步骤。

2. 有 EEPROM 且其首字节为 0xC0

如果在系统复位上电之后，FX 2 检测到有 EEPROM 连入到了其 IIC 兼容总线上，且其首地址为 0xC0，那么，FX 2 会自动将其中的 VID、PID、DID 写入自己的内部存储器。FX 2 会将这些 EEPROM 字节作为对主机的 Get_Descriptor_Device()请求响应的一部分，即用这些字节来代替“默认设备”中的 VID、PID、DID。主机操作系统会搜寻与这些字节所匹配的驱动。表 4-2 列出了 EEPROM 地址序号及对应的地址。

表 4-2　　“C0 Load”格式

EEPROM 地址	注释

0	0xC0
1	Vendor ID(VID) L
2	Vendor ID(VID) H
3	Product ID(PID) L
4	Product ID(PID) H
5	Device ID(DID) L
6	Device ID(DID) H
7	Configuration Byte

在第一次排序过后，主机驱动下载 FX 2 的固件，USB 描述符的数据被载入 FX 2 的 RAM 中，FX 2 中的 CPU 开始工作，然后，FX 2 将对当前设备进行重排序。此时，主机可能会装载一个新的驱动，这要依赖于刚刚装载的 VID/PID/DID。EEPROM 中第 8 个字节的含义为：

- I^2C 兼容总线的速度，默认为 100kHz。
- 断开连接的极性。

所谓断开连接的极性是指系统复位之后 FX 2 是否连入 USB 设备。

3. 有串行的 EEPROM 且首字节为 0xC2

同样，在系统启动后，如果 FX 2 检测到有 EEPROM 连入 I^2C 总线，且首地址为“0”，FX 2 将会把 EEPROM 中的数据装入 RAM，而且它同样会置 RENUM=1，使得设备请求由固件来掌握而不是以“默认设备”状态来出现，如表 4-3 所示。

表 4-3　　“C2 Load”格式

EEPROM 地址	注释
0	0xC2
1	Vendor ID(VID) L
2	Vendor ID(VID) H
3	Product ID(PID) L
4	Product ID(PID) H
5	Device ID(DID) L
6	Device ID(DID) H
7	Configure Byte
8	Length H
9	Length L
10	Start Address H
11	Start Address L
12	Data Block
---	----------
---	Length H
---	Length L
---	Start Address H
---	Start Address L

---	Data Block
---	----------
---	0x80
---	0x01
---	0xE6
---	0x00
Last	00000000

首字节为“C2 Load”，它指导 FX 2 将 EEPROM 中的数据拷入 RAM，FX 2 读取其后的 6 个字节。同样与上一节一样第 8 个字节为配置字节。

紧随其后的是数据记录，每一笔数据记录都由以下 3 部分组成：

- 10bit 的长度域（0~1023），给出了随后数据块中的字节数。
- 13bit 的开始地址（0~0x1FFF）。
- 数据块本身。

在数据记录的最后，总是以一个单独的全“0”字节组成，它将被载入位于地址 0xE600 的 CPU LS 寄存器中。

4. 关于 I^2C 兼容总线

FX 2 的 I^2C 兼容总线控制器有两个用途：

首先，它负责管理着串行的 EEPROM 接口，而这一接口又会在 USB 系统启动时自动检测和决定列举的方式；其次，一旦 CPU 开始工作，固件便可以将 I^2C 总线上的 EEPROM 作为普通目的存储器来访问。这便使得更广泛的一系列的标准 I^2C 外设可以应用于 USB 系统。

在没有地址冲突的情况下，其他 I^2C 设备可以被连入 SCC 和 SDA 引脚，而作为普通设备使用。

4.2.5　关于 RENUM 位

在以上各节中，都提到了一个称为“RENUM”的数据位，它的状态决定了通过端点“0”送来的请求是由“默认设备”还是由 FX 2 的固件来管理。

在系统启动之初 RENUM=0，暗示“默认设备”将会自动管理 USB 设备请求。但当固件下载完毕，FX 2 中的 CPU 开始运行后 RENUM=1，即随后的设备请求都会由固件和描述符表来回应。如果 FX 2 使用片外的程序存储器（位于地址 0x0000）而且没有启动目的 EEPROM 连入，则 FX 2 会自动将 RENUM 置为“1”，所以，设备请求会由固件和描述符表来回应。

在“C2 Load”模式中，EA 引脚为低时，FX 2 同样置“RENUM=1”。这样，固件会以内部 RAM 中下载得到的代码来运行。当然，所有设备请求也是由固件来回应的。

4.2.6 FX2 对设备请求的回应（RENUM=0）

在 RENUM=0 时，“默认 USB 设备”如何响应从端点“0”送来的设备请求如表 4-4 所示。

表 4-4　　设备请求的响应

bRequest	名称	FX2 的响应
0x00	Get_Status(Device)	返回两个 0 字节
0x00	Get_Status(Endpoint)	给已知 EP 提供响应的停止位
0x00	Get_Status(Interface)	返回两个 0 字节
0x01	Clear_Feature(Device)	无
0x01	Clear_Feature(Endpoint)	对已知的 EP 清除停止位
0x02	(reserved)	无
0x03	Set_Feature(Device)	无

续表

bRequest	名称	FX2 的响应
0x03	Set_Feature(Endpoint)	对已知的 EP 将停止位置位
0x04	(reserved)	无
0x05	Set_Address()	更新 FNADD 寄存器
0x06	Get_Descriptor()	提供内部列表
0x07	Set_Descriptor()	无
0x08	Get_Configuration()	返回内部值
0x09	Set_Configuration()	设置内部值
0x0A	Get_Interface()	返回内部值（0-3）
0x0B	Set_Interface()	设置内部值（0-3）
0x0C	Sync_Frame()	无
Vendor Request		
0xA0	下载固件	对 RAM 进行上传/下载
0xA1-0xAF	保留	由 CYPRESS 保留
All other		无

USB 主机通过发送 Set_Address()、Get_Descriptor()和 Set_Configuration()请求开始列举过程。列举过程之后，默认的 USB 设备将会响应从主机来的下列设备请求：

- 置位/清除一个端点的停止位（Set/Clear_Feature(End point)）。
- 读取一个端点的停止状态（Get_status(Endpoint)）。
- 设置/读取一个 8 位的配置数字（Set/Get_Configuration()）。
- 设置/读取一个 2 位的接口可变设置（Set/Get_Interface()）。
- 下载或上传 FX 2 的 RAM。

4.2.7　FX 2 中用于固件下载的制造商请求

先于“重新列举”，主机会将数据下载到 FX 2 的 RAM 中。主机可以访问两部分 FX 2 的片内 RAM 空间：

- 程序/数据 RAM，位于 0x0000~0x1FFF。
- 数据 RAM，位于 0xE000~0xE1FF。

这两部分由 CPU 处于重启或正常运行的状态来决定是进行上传还是下载的工作，当然，它们也可以通过 EEPROM 来进行装载。而对比来看，片外的 RAM 却不能够通过“Firmware Load”制造商请求来上传或下载。

在 USB 的规范中，对“制造商请求”（Vendor_Specific Request）也做出了相应的规定。FX 2 遵从这些规定来在主机和 FX 2 的 RAM 间进行数据传递。FX 2 会对固件的上传和下载请求做出不同的响应。

表 4-5、表 4-6 分别揭示了固件的上传和下载的字节定义。

表 4-5　　固件的下载

字节	域	值	含义	FX 2 的响应
0	bmRequest	0x40	制造商请求，输出“固件载入”	无要求
1	bRequest	0xA0		
2	wValue L	Addr L	起始地址	
3	wValue H	Addr H		
4	wIndex L	0x00	字节数	
5	wIndex H	0x00		
6	wLength L	Len L		
7	wLength H	Len H		

表 4-6　　固件的上传

字节	域	值	含义	FX 2 的响应
0	bmRequest	0xC0	制造商请求，输入“固件载入”	无要求
1	bRequest	0xA0		
2	wValue L	Addr L	起始地址	
3	wValue H	Addr H		
4	wIndex L	0x00	字节数	
5	wIndex H	0x00		
6	wLength L	Len L		
7	wLength H	Len H		

注意：所有这些请求总是由 FX 2 来掌握的，而不考虑 RENUM 位的状态。

4.2.8 重新列举（ReNumerates）

FX 2 中 USB CS（USB 控制和状态）寄存器的两个控制位，DISCON 和 RENUM 控制着“重排序”的过程，如表 4-7 所示。

表 4-7 USB CS 寄存器

b7	b6	b5	b4	b3	b2	b1	b0
HSM	0	0	0	DISCON	NOSYNSOF	RENUM	SIGRSUME
R/W	R	R	R	R/W	R/W	R/W	R/W
0	0	0	0	0	1	0	0

为了断开 USB 连接，固件威风将 DISCON 置为 1；为了“重连接”，固件威风将 DISCON 清零。在重新连接之前，固件将会把 RENUM 位置位或清除。当 RENUM=1 时，表示由固件来回应从端点“0”来的请求；当 RENUM=0 时，则是由“默认的设备”来完成这一功能。

4.2.9　初始化下载的过程

下面以一个例子来说明一个完整的设备列举过程，这里假定的环境为 FX 2 控制芯片，固件由 EEPROM 下载到 FX 2 的内置 RAM 中。具体的步骤为：

（1）USB 设备从总线上得到电力供给。

（2）复位电路将 USB 芯片暂时保持在重启状态，直到 PLL（Phase Lock Loop）锁定在 24MHz 的晶振上。

（3）FX 2 检测到有 EEPROM 连入到了 I^2C 总线上。FX 2 从 EEPROM 中读取 VID 和 PID 值，并用其代替自身内部的 VID 和 PID 值。

（4）主机检测到设备已连入，并向设备请求其 ID 值，设备用代表用户所连入设备的 ID 值来下载固件设备的 ID 值进行回应。

（5）主机接收到这些 ID 值，并开始装载其驱动程序。

（6）在驱动程序的引导下，FX 2 开始执行从外部将固件下载到内部 RAM 的步骤。

（7）驱动程序使 FX 2 中的 8051 CPU 芯片重新启动，用户的固件程序开始运作。

（8）固件使 USB 在电气上与总线断开连接。

（9）主机发现了初始设备的脱离，于是便将驱动程序从其内存中清除。

（10）固件使 USB 设备重新连入总线。

（11）主机检测到了设备的连入，开始询问其 ID 值。

（12）固件以代表用户已经配置好的设备的 ID 值进行回应。

（13）主机收到 ID，开始装载典型驱动或是用户自己所做的设备驱动。

（14）到这一步，主机和 FX 2 的固件便连接完毕，开始执行相应的具体应用程序。

4.3　USB 2.0 中对于列举的规定

通过上一节的学习，读者已经在 FX 2 的列举过程中明白了列举所要经历的一系列步骤。作为 USB 的规范本身对 USB 的设备列举做出了相应的规定。下面介绍 USB 设备列举的通用准则。

4.3.1　列举过程设备经历的状态

在 FX 2 的示例中，看到了设备连入 USB 集成器后，主机与设备之间电气的物理的连接步骤。在 USB 的规范中，将这些步骤设备所经历的过程分为 6 个状态。

1. 连接状态

一个设备可以在任何时候插入或拔出总线，USB 规范并未定义设备未连接时的状态。如果设备接入总路线，并从集线器抽取电流，则可以说设备处于连接状态，但此时的供电状态可能还没有完全稳定下来。

2. 加电状态

USB 设备可能自身供电，也可能从 USB 的集线器上得到电力的供给。虽然自身供电的设备在连入总线前就有可能已经有电力供给了，但此时并不能认为它已处于加电状态，只有在设备连入总线，而且 VBUS 也与设备接通后，才能认为设备处于了加电状态。

设备要通过设备描述符来报告其电源的供给情况，当前的电源是作为配置描述符的一部分报告给主机的。设备可以在任意时刻改变其电源的来源。如果配置允许两种供电方式的话，则应该将两种能够从 VBUS 得到的最大电力供给作为描述符的一部分报告给主机。

如果设备在最终是自身供电且供电电流大于 100mA，则当设备切换到了总线供电方式后，设备将会返回到地址状态。

对于集线器来说，其端口必须始终处于供电状态，以便于检测设备的连入与拔除。总线供电的集线器在得到正确的配置之前是不会向其下游端口供电的。在经过配置之后，集线器将向下游提供额定的电流。在检测到一个设备的连入后，主机会使相应的端口处于等待工作的状态，这可能会引起设备的重新启动。

3. 默认状态

在设备加电之后，它在接收到总线上的重启信号之前不应该响应任何的事务处理。在接收到重启信号之后，设备接下来便会接受默认的地址设备。

重启过程完成之后，USB 设备以正常的速度运行，对于全/低速的选定取决于设备终端的电阻。如果设备是高速设备的话，则作为重启过程的一部分，应该决定是否应用高速模式（关于这部分内容，将在 4.3.3 节作详细的介绍）。

在设备重启成功之后，设备必须成功地响应设备和配置描述符。

4. 地址状态

在上电和重新启动之初，所有的 USB 设备都要使用默认的地址。在重新启动完成之后，每一个 USB 设备都会被分配一个独立的地址。在设备处于挂起状态时，仍将保持其现有的地址。

在设备被分配到一独立的地址或是使用默认的地址时，USB 设备会通过其默认的管线来响应请求。

5. 设置状态

在 USB 设备正常运作之前，设备必须被配置。从设备的观点来看，配置过程必须包括一个非零的设置值来响应 SetConfiguration()的请求。

在配置一个设备或更改设置的过程中，都会引起相应的端点状态和配置值被设为默认值。

6. 挂起状态

为了节省电力，USB 设备在处于总线没有活动状态一段特定的时间之后，便会进入挂起状态。处于挂起时，设备会保持其任何的内部状态，包括其地址和配置信息。

连入总线的 USB 设备必须在加电之后，准备好在任何时候进入挂起状态，不管它是对于非默认的地址状态还是配置状态。另外，USB 设备在其对应的端口被禁止后也应进入挂起状态，这可以与选择性的挂起作类比。

一旦总线恢复活动，设备便会脱离挂起状态。USB 设备可能要求主机脱离挂起或选择性

挂起的状态，这可能要用到进程唤醒的信号。设备进程唤醒的功能是可选的，当设备处于重启状态时，远端唤醒信号必须被屏蔽。

4.3.2 总线列举要经历的步骤

当一个 USB 的设备被连入或拔出时，主机便用上述的列举过程来对设备必要的状态改变进行鉴定和管理。当一个 USB 设备被连入一个已上电的端口时，将会经历以下几个步骤：

（1）设备连入的集线器通过其状态改变管道来通知主机所发生的事件。此时，USB 的设备处于上电状态，相应的端口还处于屏蔽状态。

（2）通过询问集线器，主机决定改变的正确状态。

（3）在主机获知有新的设备连入端口后，主机最少需要等待 100ms 来确保插入过程完成，以及设备供电的稳定。然后，主机发出一个端口使能信号和相应的端口重启命令。

（4）集线器执行相应端口所需的重启过程。当重启信号被发现时，端口必须处于使能状态。USB 设备现在处于默认状态，并能从 VBUS 线上得到不多于 100mA 的电流供给。设备所有的寄存器和状态都将被重启，并以默认的地址做回应。

（5）主机为 USB 设备分配一个惟一的地址，使设备转入地址状态。

（6）在 USB 设备收到惟一地址之前，仍然可以通过默认地址访问其默认的控制管道。主机读取设备描述符来决定读 USB 设备的默认管道所能使用的最大有效数据载荷。

（7）主机读取配置信息，在这一过程中，设备可能需要向主机提交多条配置信息。这一过程可能会花费若干毫秒来完成。

（8）基于配置信息和 USB 设备的使用方式，主机给 USB 设备分配一个配置值。然后，设备转入配置状态。此时，设备可以从 VBUS 线上得到其描述符中所需的电流大小。

到这一步，从设备的角度来看，设备完成了配置过程，等候具体的应用。

当设备被移除后，集线器会重新给主机发出一条通知信息。设备的拔除会使相应的端口处于被禁止的状态。根据接收到的设备撤除的信息，主机将会更新内建的设备拓扑信息。

4.3.3 总线列举过程中要用到的描述符

在列举过程中，USB 设备使用各种描述符来向主机报告它的属性。描述符是具有一定格式的数据结构，它包括了设备所有的属性信息。在主机使用控制传输请求描述符时，设备就会将这些信息以标准描述符格式发送给主机，从而使主机控制器获取诸如设备及供应商 ID，设备通信能力等重要信息。所有的 USB 外设都必须响应对标准 USB 描述符的请求，并且要按照一定的规范进行。如果在一个返回的描述符中长度字段的值小于 USB 规范所定义的值，那么，主机将会认为该描述符无效，并会丢掉所接收到的描述符。如果这个长度字段的值大于 USB 规范所定义的值，那么，冗余的字节将会被主机忽略。但是下一个描述符的位置是由返回的字节长度来定位的，而不是由所期望的长度来确定的。

1. 描述符的种类

标准 USB 描述符包括设备描述符、配置描述符、接口描述符、端点描述符和字符串描述符。它们分别从不同的层次来描述设备的属性，每个 USB 设备有且仅有一个描述设备整体信息，并且指定这个设备支持的设备配置号的设备描述符。而一个设备有可能支持一个或多种

不同的配置，每种配置都支持各自的接口，而每个接口描述符包含零个或多个端点描述符，这些端点描述符包括了与一个端点通信需要的信息。没有端点的接口只能使用控制传输来通信。

最后一种描述符是字符串描述符，它可以保存诸如设备供应商或设备的名字的文本信息。一个设备不一定要包括字符串描述符，但是一定要有对字符串描述符的引用字段。如果一个设备不支持字符串描述符，那么，该引用字段必须复位为零以指示没有字符串描述符。

除了这 5 种标准描述符外，设备还可以有“类”或“供应商”特定描述符。这些描述符为一个设备描述属于自己的、更详细的信息提供了一个结构化的方法。设备可以使用下面的两种方法来返回类或供应商特定描述符。

- 如果这两种描述表的格式与标准格式相同（以长度字节打头，紧跟着类型字节）则它们可由 Get_Descriptor（Configuration）请求在配置信息里与标准描述符一同返回。在这种情况下，类或供应商特定描述表一般跟在被修改的或被扩展的标准描述符之后。
- 如果这两种描述符独立于配置信息或是使用的非标准的格式，则可以使用指定了类或供应商特定的描述符及索引的 Get_Descriptor()请求，从设备获取这两种描述符。类或供应商特定的规范会指出正确获取这两种描述符的途径。

以上介绍的这些是 USB 2.0 规范和 USB 1.1 规范共有的描述符。在 USB 2.0 规范中还扩展定义了两种速度相关描述符。他们分别是 the（other_speed）device_qualifier 描述符和 the other_speed configuration 描述符。前面已经提过对于支持高速的设备来说，操作速度的选择是在复位时正确地管理收发器来确定的，而且复位过程结束后，设备只能运行在一个单一的速度之上，更重要的是在设备正常操作时是不允许进行速度切换的。但是，一个支持高速的设备可以有与速度相关的配置。这就是说，设备可以具有一些只在设备运行于高速或是全速时才有效的配置。一个高速设备必须支持报告这些与速度相关的描述符。

每个描述符都包含一个表明其描述符类型的值。USB 2.0 规范定义的描述符类型值如表 4-8 所示。

表 4-8　　描述符类型值

类型	值（十六进制）
DEVICE	01
CONFIGURATION	02
STRING	03
INTERFACE	04
ENDPOINT	05
DEVICE_QULIFIER	06
OTHER_SPEED_CONFIGURATION	07
INTERFACE_POWER	08

每个描述符由一系列字段组成。这些字段名字的大部分使用前缀来表明该字段的内容属性：b 表示该字段为一个字节的长度；w 则表示两个字节的长度；bm 表示位映射，即字节中的每一位都对应着不同的含义；bcd 表示 BCD 码，即二进制编码的十进制；i 表示索引或是指针；id 表示标志符。

2. 设备描述符

设备描述符描述关于一个 USB 设备的总体信息，它是列举过程中主机从设备读取的第一个描述符，一个设备仅有一个设备描述符。在全速和高速下有不同设备配置信息的高速设备必须具有 device_qualifier 描述符。设备描述符包含 14 个字段，表 4-9 按它们出现在描述符中的顺序列出了这些字段。

表 4-9　　设备描述符

偏移量	域	大小	值	描述
0	bLength	1	数字	描述符的字节长度
1	bDescriptorType	1	常量	设备描述符类型
2	bcdUSB	2	BCD 码	USB 规范发行版本号（BCD 码）
4	bDeviceClass	1	类	设备类代码
5	bDeviceSubClass	1	子类	子类码
6	bDevicePortocol	1	协议	协议码
7	bMaxPacketSize0	1	数字	端点 0 的最大包大小
8	IdVendor	2	ID	厂商标志（由 USB 标准付值）
10	IdProduct	2	ID	产品标志（由厂商付值）
12	BcdDevice	2	BCD 码	设备发行号（BCD 码）
14	iManufacturer	1	索引	描述厂商信息的字符串的索引
15	iProduct	1	索引	描述产品信息的字符串的索引
16	iSerialNumber	1	索引	描述设备序列号信息的字符串的索引
17	bNumConfigurations	1	数字	可能的配置号码

- bLength：该描述符的字节长度。
- bDescriptorType：描述符类型字段，在此处应该为设备描述符类型值（01H）。
- bcdUSB：设备和它的描述符所遵循的 USB 规范发行版本号，采用 BCD（二进制编码的十进制）码格式。假定版本为 JJ.M.N，则这个字段的值是 0xJJMN。其中，JJ 表示主版本号，M 表示次版本号，N 则表示再次版本号。例如，规范版本为 2.1.3，在此表示为 0x0213，2.0 版本表示为 0x0200。
- bDeviceClass：这个字段是类代码。它是由 USB－IF 来统一分配的。如果这个字段的值是 0，则一个配置中的每一个接口都定义各自的类信息，各个接口可以相互独立工作。

从 1~0xFF 之间的值是为 USB 定义的类保留的。例如，HID 类、集线器类、音频设备类等。如果是 0xFF，说明类是由供应商指定的。

- bDeviceSubClass：该字段是子类代码，也是由 USB－IF 统一分配的。对于属于 USB 定义的类中的一个设备来说，这个字段可以指定这个类中的一个子类。因此，这个值要由 bDeviceClass 的值来决定。如果 bDeviceClass 的值为 0，则这个子类也必须为 0；如果设备类的值介于 0x01~0xFF 之间，那么，子类必须是一个 USB 规范定义的代码。
- bDeviceProtocol：如果设备支持基于设备基础上而不是接口基础上的类特定协议，那么，这个字段就指定一个规范定义的设备类协议代码。如果这个值为 0，说明设备不支持

基于设备的类特定协议，但是它可以在接口基础上使用类特定协议。如果这个值为 0xFF，则设备使用一个基于设备的供应商特定协议。

- bMaxPacketSize0：终端 0 的最大包的大小。主机在随后的请求中使用这个信息，低速设备允许的值为 8；全速设备允许的值为 8、16、32 或 64；而高速设备所允许的值必须是 64，表明控制端点只能使用 64 个字节的最大包。
- IdVendor：USB 使用者论坛的会员和那些支付管理费的用户收到一个惟一的供应商 ID。每一个商业产品的设备描述符都必须有一个供应商 ID。例如，菲利普的 idVender 是 0x0471。CYPRESS 公司的 idVender 为 0x0512。主机中设备的 INF 文件也包含了这个值，操作系统也需要这个值来决定为该设备载入哪个驱动程序。
- IdProduct：这个字段的值是产品 ID，它是由设备生产商来指定的，用于识别这个设备。设备描述符和主机中的 INF 文件都包含了这个值，操作系统也需要这个值来决定为该设备载入哪个驱动程序。
- BcdDevice：BCD 格式的设备发行版本号。是由设备制造商指定的，是可选择的；也可以用来决定为这个设备载入哪个驱动。
- iManufacturer：这个字段是一个指向描述设备制造商的字符串的索引，是可选择的。如果没有字符串，则为零。
- iProduct：如果设备包含描述产品的字符串描述符，则该字段是这个描述符的索引，也是可选择的。
- iSerialNumber：这是一个指向包含设备序列号的字符串的索引，也是可选择的，不用时为零。如果用户在总线上有不止一个相同的设备并且主机需要跟踪固定的设备时，序列号是很有用的。它们也使得主机能确认这个设备是与先前使用的哪个是完全相同的外设，还是一个只具有相同的供应商和产品 ID 的外设。如果一个设备有序列号，则用户把这个设备插入到主机的一个不同的端口中时，操作系统就不需要重新载入驱动程序了。
- bNumConfigurations：该字段的值指明了设备当前运作速度下的配置号。如果有针对其他特定速度的设备配置，这个字段仅仅反映了其中一个速度的配置。

3. 设备限定符描述符

设备限定符描述符属于速度相关描述符。它描述了一个高速设备在进行速度切换时所需改变的信息。例如，一个设备当前运行于全速状态，那么，当主机请求时，它会返回说明设备如何运行在高速状态的信息，反之亦然。主机也是通过 Get_Descriptor()请求来访问该描述符的。如果是一个全速的设备（但设备描述符的协议版本号是 0x0200）收到对该描述符的请求，则它必须以请求错误来响应主机。设备限定符描述符的字段如表 4-10 所示。

表 4-10　　设备限定符描述符

偏移	字段	大小	值	描述
0	blength	1	数字	描述符大小
1	bDescriptorType	1	常量	描述符类型（0x06）
2	BcdUSB	2	BCD	BCD 码表示的 USB 规范版本号(例如 V2.0 表示为 0x0200）
4	bDeviceClass	1	类	类代码
5	bDeviceSubClass	1	子类	子类代码
6	bDeviceProtocol	1	协议	协议代码

7	bMaxPacketSize0	1	数字	在其他速度下端点 0 的最大包大小
8	bNumConfiguration	1	数字	其他速度配置号
9	bReserved	1	0	保留字段，必须为 0

4. 配置描述符

主机在接收了设备描述符之后，就可进一步得到设备的配置、接口和终端描述符了。

配置描述符描述了关于一个特定的设备配置信息，每个设备至少有一个描述设备的特征和能力的配置。通常一个就够了，但是一个多用途或多模式的设备可以支持多个配置，每个配置都有一个配置描述符。主机用 Set_Configuration()请求来选择一个配置，而用 Get_Configutation()请求来返回一个配置。但主机请求一个配置描述符时，与该配置相关的所有的接口和端点描述符将被一同返回。

一旦配置过后，如果一个特定的接口有可选择的设置，那么，可以在配置过后进行选择。这个描述符有 8 个字段，如表 4-11 所示。

表 4-11　　配置描述符

偏移	字段	大小	值	描述
0	bLength	1	数字	描述符大小
1	bDescriptorType	1	常量	描述符类型（0x02）
2	wTotalLength	2	数字	这个配置返回数据的总长度
4	bNumInterfaces	1	数字	这个配置支持的接口号
5	bConfigurationValue	1	数字	SetConfiguration( )请求的参数，用以选择这个配置
6	iConfiguration	1	索引	这个配置的字符串描述符索引
7	bmAttributes	1	位映射	自供电/总线供电和远程唤醒属性的设置
8	bMaxPower	1	毫安	USB 设备能消耗的最大总线功率

- blength：该描述符的字节长度。
- bDescriptorType：描述符类型字段，在此处应该配置描述符类型 0x02。
- wTotalLength：对该配置请求所返回的所有数据字节总数。包括这个配置的所有接口、端点和类及供应商特定的描述符。
- bNumInterfaces：为该配置支持的接口号。
- bConfigurationValue：作为 Set_Configuration() 请求中的参数，用来识别和标志这个配置。如果在请求中该值为零，将会导致设备进入未配置状态。
- iConfiguration：描述这个配置的字符串描述符的指针，是可选择的。
- bmAttributes：这个字段是位映射的值，它描述了这个配置的属性。第 5 位设置为 1，表明这个设备配置支持远程唤醒功能；第 6 位设置为 1，表明设备是自供电。但是如果一个设备配置支持两种供电模式，那么这个配置仍然会设置第 6 位为 1，并在 MaxPower 字段里报告它所需要的总线电能的值。在实际运行时，主机会通过 GetStatus（DEVICE）来获知实际的供电模式。第 0 到第 4 位保留未用，应为 0；第 7 位也保留未用，但是由于历史的原因，该位应设置为 1。
- bMaxPower：指定设备在该配置中正常运作所需的最大总线电流。这个值是以 2mA 为单位的。例如，设备需要 100mA，则 bMaxPower 的值为 50。设备可以从总线获得的最大电流是 500mA。

5. 速度（Other_Speed Configuration）配置描述符

其他描述符描述了一个高速设备运行在其他可能的速度模式下的一个配置，它的结构等同于一个配置描述符。主机同样使用 Get_Descriptor()请求来获取该描述符。此描述符的字段如表 4-12 所示。

表 4-12　　其他速度配置描述符

偏移	字段	大小	值	描述
0	bLength	1	数字	描述符大小
1	bDescriptorType	1	常量	描述符类型（0x07）
2	wTotalLength	2	数字	这个配置返回数据的总长度
4	bNumInterfaces	1	数字	这个配置支持的接口号
5	bConfigurationValue	1	数字	Set_Configuration( )请求的参数，用以选择这个配置
6	iConfiguration	1	索引	这个配置的字符串描述符索引
7	bmAttributes	1	位映射	自供电/总线供电和远程唤醒属性的设置
8	bMaxPower	1	毫安	USB 设备能消耗的最大总线功率

6. 接口描述符

接口描述符描述了一个配置中的特定接口。一个配置可以提供一个或多个接口，每一个接口都包括零个或多个描述该配置中一组端点属性的端点描述符。如果一个配置可以支持不止一个接口，那么当主机发出 Get_Descriptor()请求时，一个特定接口所包含的端点描述符会跟在该接口描述符后被返回。而接口描述符总是作为配置描述符的一部分被返回的，主机不能直接用 Set_Description()和 Get_Descriptor()请求来访问接口描述符。一个接口描述符有 9 个字段，这些字段在描述符中出现的顺序如表 4-13 所示。

表 4-13　　接口描述符

偏移	字段	大小	值	描述
0	BLength	1	数字	描述符大小
1	bDescriptorType	1	常量	描述符类型（0x04）
2	bInterfaceNumber	1	数字	接口号数
3	bAlternateSetting	1	数字	用于选择一个替换设置的值
4	bNumEndpoints	1	数字	这个接口所支持的端点号
5	bInterfaceClass	1	类	类代码
6	bInterfaceSubClass	1	子类	子类代码
7	bInterfaceProtocol	1	协议	协议代码
8	IInterface	1	索引	接口的字符串描述符索引

- bLength：为该描述符的字节长度。
- bDescriptorType：描述符类型字段，在此处应该配置描述符类型 0x04。
- bInterfaceNumber：接口号。该字段指定了指向这个配置所支持的接口队列的索引的

值。作为 Get_Interface()和 Set_Interface()请求的参数。

- bAlternateSetting：一个接口可以包括可替换的设置，这样就允许即使在设备被配置好之后，端点和它们的特性仍然可以改变。一个接口缺省的设置总是设置 0。可选的接口设置使得部分的设备配置能在其他接口进行操作的情况下改变。如果一个配置对于它的一个或多个接口有替换的设置，每一设置包括一个独立接口描述表和相关结点。如果一个设备配置支持单个接口，并且此接口有两个可选设置，配置描述符返回以后会紧跟着返回皆为 0 的 bInterfaceNumber 与 bAlternateSetting 域，即第一个设置的接口描述符和相关的端点描述符，随后是另一个设置接口描述符与端点描述符。第二个接口描述符的 bInterfaceNumber 域也应为 0，但 bAlternateSetting 域应为 1。
- bNumEndpoints：除了终端 0 外接口支持的端点号。对于一个只支持端点 0 的接口，传送完毕接口描述符后将不会再传送端点描述符。这时，bNumEndpoints 应设置为零。
- bInterfaceClass：与设备描述符的 bDeviceClass 类似，但是它用于接口定义的类。0x01 到 0xFF 之间的值是为 USB 定义的类而保留的。0xFF 表示销售商定义的类，零是保留值。
- bInterfaceSubClass：与设备描述符的 bDeviceSubClass 类似，但它是用于接口定义的类。对属于 USB 定义的类中一个类的接口来说，这个字段可以指定这个类中的一个子类。如果 bInterfaceClass 是 0，在 bInterfaceSubClass 也必须为 0。如果 bInterfaceClass 介于 0x01~0xFF 之间，则 bInterfaceSubClass 必须是 USB 规范定义的一个类代码。0xFF 表示销售商定义的类。
- bInterfaceProtocol：与设备描述符的 bDeviceProtocol 类似，用于那些由接口定义的类设备，可以指定一个由选定的 bInterfaceClass 或 bInterfaceSubClass 定义的协议。如果 bInterfaceClass 是介于 0x01~0xFF 的值，则 bInterfaceProtocol 必须是一个由 USB 规范定义的代码。

7. 端点描述符

一个接口所使用的每一个端点都有它自己的描述符。这个描述符包含了主机用来确定每个端点带宽要求的信息。端点 0 是不需要描述符的，端点描述符也不能直接由 Get_Descriptor()和 Set_Descriptor()请求来访问，而只能作为配置信息的一部分由 Get_Descriptor(Configuration)请求来获得。端点描述符的 6 个字段如表 4-14 所示。

表 4-14　　端点描述符

偏移	字段	大小	值	描述
0	bLength	1	数字	描述符大小
1	bDescriptorType	1	常量	描述符类型（0x05）
2	bEndpointAddress	1	端点	端点地址
3	bmAttributes	1	位映射	端点属性
4	wMaxPacketSize	2	数字	支持的最大包的大小
5	bInterval	1	数字	轮询的时间间隔

- bLength：该描述符的字节长度。
- bDescriptorType：描述符类型字段，在此处应该配置描述符类型 0x05。
- bEndpointAddress：该描述符所描述的端点的地址，它包括端点号和传输方向。从第 0 位到第 3 位是端点号，低速设备最多支持 3 个端点，而全速设备可以支持 16 个。第 4 到第

6 位保留，被设置为 0。第 7 位是方向位，0 表示 OUT，1 表示 IN。

❑ bmAttributes：该字段的 0 和 1 位描述了端点的传输类型。其中，00 表示控制传输，01 表示等时传输，10 表示批量传输，11 表示中断传输。如果该端点不是等时传输，那么该字段的 2 位到 5 位都应该为保留位并被设置为 0。如果是等时传输端点，则 2 位和 3 位表示同步化类型。其中，00 表示没有同步化，01 表示异步，10 表示适应，11 表示同步。4 位和 5 位表示端点的用法类型。其中，00 表示该端点用于正常的数据传输；01 表示端点用来传递明确的反馈信息；10 表示该端点作为数据端点，但同时起着暗含的反馈作用；11 为保留值，其余均为保留位。如果一个端点被作为明确反馈端点，那么传输类型必须设置为等时传输（位 5.4=01B）。

一个反馈端点，不管是明确的还是隐含的，都必须和一个或多个等时数据端点联系在一起，以便为它们提供反馈服务。这个联系是基于端点号数的匹配。一个反馈端点的方向总是和它提供服务的数据端点的方向相反。如果多个数据端点要由同一个反馈端点来提供服务，那么这些数据端点必须具有按升序排列但不一定连续的端点号数。其中，第一个数据端点必须和反馈端点有相同的端点号（不同的方向）。

❑ wMaxPacketSize：当前配置下，该端点能够发送和接收的最大包的大小。对于等时端点，这个值用于为每个微帧的有效数据负荷预留总线时间，而通道可能在实际运行时不需要所预留的那么多带宽。实际带宽可由设备通过一种正常的，但非 USB 定义的机制汇报给主机。所有的端点都使用位 0 到 10 来表示这个最大值。对于高速等时和中断端点，位 11 和位 12 用来指定每一微帧中附加的事务的数量。其中，00 表示没有附加事务（每一微帧一个事务），01 表示附加一个事务，10 表示附加两个事务，11 为保留值。如果这两位是 00，那么该端点的最大包的大小可以是任何协议所允许的值。但是，如果这两位不是 00，则 wMaxPacketSize 字段的 10 位到 0 位将受到限制，如表 4-15 所示。

表 4-15　　wMaxPacketSize 的格式

wMaxPacketSize（位 12~11）	wMaxPacketSize（位 10~0）
00	1-1024
01	513-1024
10	683-1024
11	保留

❑ bInterval：轮询端点数据传输的间隔时间。根据设备运行速度，高速设备以 125us 为单位，低速和全速设备以 1ms 为单位。等时和中断端点的 bInterval-1 值是作为 2 的指数来使用的。例如，bInterval 的值为 4，意味着以 8 为周期。对于全速和高速等时端点，这个值只能在 1~16 之间变化；对于低速和全速中断端点，这个值可以在 1~255 之间变化；对于高速中断端点，这个值也只能在 1~16 之间变化；对于高速批量和控制 OUT 端点，bInterval 的值用于指定端点可插入最大数量的 NAK 标志的比率，0 表示该端点永远不会 NAK。其他的值表明每 bInterval 数量个微帧最多 1 个 NAK，并且这个值在 0~255 之间变化。

8. 字符串描述符

字符串描述符是可选择的。正如前面提到的，如果一个设备不支持字符串描述符，那么，在设备描述符、配置描述符、接口描述符、端点描述符中所有关于字符串描述符的索引都必

须设置为 0。字符串描述符使用国际统一的字符编码标准 UNICODE 编码格式，USB 设备中的字符串可以支持多种语言。当主机请求一个字符串描述符时，请求器通过由 USB-IF 定义的一个 16 位的语言 ID（LANDID）来指定所需的语言。字符串描述符 0 的字段如表 4-16 所示。

表 4-16　　字符串描述符 0

偏移	字段	大小	值	描述
0	bLength	1	数字	描述符大小
1	bDescriptorType	1	常量	描述符类型（0x03）
2	wLANGID[0]	2	数字	LANGID 代码 0
..	……	..	…	……
N	wLANGID[N]	2	数字	LANGID 代码 N

- bLength：该描述符的字节长度。
- bDescriptorType：描述符类型字段，在此处应该配置描述符类型 0x05。
- wLANGID[0..N]：这是 LANGID 代码队列。每个 LANGID 是一个双字节的语言 ID。英国英语的子代码是 0x0009，美国英语的子代码是 0x0004。LANGID 代码队列的大小可以由该字符串描述符的第一个字节的值减去 2 而得到，并且这个代码队列是不能以零作为结束的。
- bString：这个字段包含了一个 UNICODE 字符串（如表 4-17 所示），采用双字节进行编码。这个字符串描述符也不能以 0 作为结尾。

表 4-17　　UNICODE 字符串描述符

偏移	字段	大小	值	描述
0	bLength	1	数字	描述符大小
1	bDescriptorType	1	常量	描述符类型（0x03）
2	bString	N	数字	UNICODE 编码字符串

- bLength：该描述符的字节长度。
- bDescriptorType：描述符类型字段，在此处应该配置描述符类型 0x05。
- wLANGID[0..N]：这是 LANGID 代码队列。每个 LANGID 是一个双字节的语言 ID。英语的代码是 0x0009，美国英语的子代码是 0x0004。LANGID 代码队列的大小可以由该字符串描述符的第一个字节的值减去 2 而得到，并且这个代码队列是不能以零作为结束的。
- bString：这个字段包含了一个 UNICODE 字符串，采用双字节进行编码。这个字符串描述符也不能以 0 作为结尾。

4.4　本章小结

在本章中介绍了 USB 设备实现即插即用的第二部分：主机对设备的检测过程。值得一提

的是，设备检测严格说来并不是一个独立的功能，它是在集线器规范，主机数据传送协议和请求数据格式的规定等各个方面的保证下才能够完成的。然而，设备的检测又有其自身的独立性。因为这正是 USB 设备与其他计算机设备的最大不同之处，正是其设备监测与加载过程的不同，才造就了 USB 设备的即插即用特性。

希望读者通过学习本章内容，能够对检测过程有所理解，在第 5 章中将对 USB 即插即用特性基础的第 3 部分——控制传输做详细讲解。

第 5 章　控制传输

知识点：

- 控制传输
- 设置阶段
- 数据阶段
- 状态阶段
- 设备请求

本章导读：

在本章中将详细介绍控制传输的相关知识，包括设置阶段、数据阶段和状态阶段，以及 11 种标准设备请求。

5.1　基本理论

在 4 种传输类型：控制传输（Control Transfer）、批量传输（Bulk Transfer）、中断传输（Interrupt Transfer）和等时传输（Isochronous Transfer）中，控制传输是最复杂的，同时也是规范所定义的惟一带有函数的传输类型。尽管在第 4 章中已经简单介绍了控制传输，但是因为控制传输的特殊性，它是 USB 设备正常工作的前提，设备接入总线后的列举过程都是要使用控制传输的。因此，本章将会继续深入地介绍有关控制传输的相关知识，包括如何在固件中使用标准和供应商特定的请求。

主机使用控制传输来与设备交换配置信息，同时，控制传输也可以用于一般的数据传输。每一个控制传输都有一个规范定义的格式，包括一个设置阶段（Setup Stage），一个可选择的数据阶段（Data Stage），以及一个状态阶段（Status Stage）。每一个阶段都是由一个或多个事务组成的，而事务又包括一个令牌段（Token Phase），一个数据段（Data Phase），以及一个握手段（Handshake Phase）。

在低速和全速传输环境下，传送给低速设备的所有下行数据包都要求一个前导包，它包括一个 SYNC（同步字段），紧跟着一个 PRE PID（包标志符）。高速传输还会使用 PING 协议，USB 为分割事务（Split Transaction）定义了一个特殊的标志包。分割事务标志包用于支持主机控制器和一个运行于高速但挂接着低速和全速设备的集线器之间进行分割事务传输。

5.1.1　设置阶段

设置阶段包括一个设置事务（Setup Stage），这个事务起到两个作用：第一就是用来识别该传输是控制传输；第二就是传输请求和其他必要信息。设备必须接受并响应每一个设置事务，如果设备正在处理其他的控制传输，它必须放弃那个传输，转而响应新的设置事务。接下来，将详细介绍在设置阶段中各个包的信息。

1. 标志包

（1）格式

图 5-1 显示了标志包的字段格式，所有的包都是以 SYNC 字段开始的。一个标志包不仅包括一个 PID（Packet Identifier）字段，用于指示包的类型是 IN、OUT 还是 SETUP（表 5-1 列出了规范所定义的所有的 PID），还包括一个 ADDR（地址）和 ENDP（端点）字段，这两个字段是用来寻址功能端点的。PING 特殊标志包和一般标志包有相同的字段格式。对于 OUT 和 SETUP 事务来说，地址和端点字段惟一指定了一个接收后续数据包的端点；对于 IN 事务来说，这两个字段惟一指定了一个可以发送数据包的端点；而对于 PING 事务来说，这两个字段惟一指定了一个以握手包进行响应的端点。

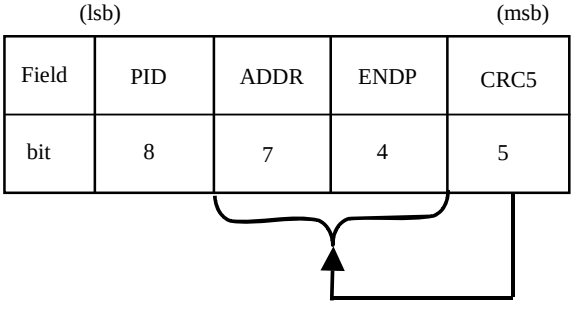


图 5-1　标志包字段格式

表 5-1　　PID 类型

PID 类型	PID 名称	PID[3:0]	描述
标记（Token）	OUT	0001B	在主机到设备功能的事务中，有地址+端口号
	IN	1001B	在设备功能到主机的事务中，有地址+端口号
	SOF	0101B	帧开始标记和帧数
	SETUP	1101B	在主机到设备功能建立一个控制管道的事务中，有地址+端口号
数据（DATA）	DATA0	0011B	偶数数据包 PID
	DATA1	1011B	奇数数据包 PID
	DATA2	0111B	在一个微帧中高速高带宽等时事务的数据包 PID
	MDATA	1111B	高速分割事务和高带宽等时事务的数据包 PID
握手（Handshake）	ACK	0010B	接收器收到无错的数据包
	NAK	1010B	接收设备不能接收数据，或发送设备不能发送数据
	STALL	1110B	端口挂起，或一个控制管道请求不被支持

	NYET	0110B	接受者还没有响应

续表

PID 类型	PID 名称	PID[3:0]	描述
特殊（Special）	PRE	1100B	（标志）主机发送的前导标志，指示低速设备的下行总线通信
	ERR	1100B	（握手）分割事务错误联络（重新使用 PRE 值）
	SPLIT	1000B	（标志）高速分割事务标志
	PING	0100B	（标志）批量/控制端点高速流探测
	保留	0000B	保留 PID

（2）功能

标志包的功能是用来指定数据接收器，并标志该事务为设置事务。

发送方：主机。

PID（Packet Identifier，包标志符，如图 5-2 所示）：SETUP。

其他内容：设备和端点地址。

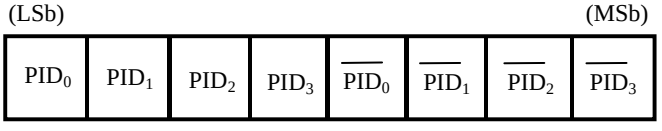


图 5-2　PID 字段格式

在所有的 USB 包中，PID 字段都是紧随于 SYNC 字段之后。PID 字段的低 4 位指明了包的类型，并可以由此推断出包的格式和错误检测方式。PID 的高 4 位是检测位，用于确保 PID 的正确译码和其后数据的正确解释。表 5-1 列出了所有的 PID 类型，在 PID 中 ADDR 的字段格式如图 5-3 所示。

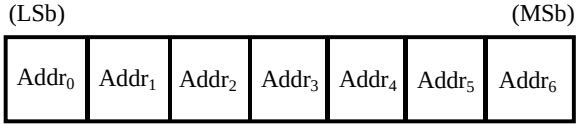


图 5-3　ADDR 字段格式

ADDR 字段通过地址来指定一个设备功能，它既可以是一个数据包的源，也可以是目的地，这由标志包的 PID 来决定。ADDR 字段可以用于 IN、SETUP 和 OUT 标志包，以及 PING 和 SPLIT 特殊标志包，一个 ADDR 值只能指定一个功能。在复位和上电时，功能的地址被设置为缺省值 0，并且必须在列举过程中由主机指定。从图 5-3 中可以看出，ADDR 字段一共可以指定 128 个地址，但是，地址 0 只能保留为缺省地址，不能分配作其他使用。PID 中 ENDP 的字段格式如图 5-4 所示。

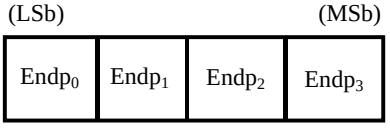


图 5-4　ENDP 字段格式

图 5-4 所示为附加的 4 位端点字段（ENDP），允许功能更加灵活地寻址，它使得传输可以指向不同的端点。除了端点 0 以外，其余的端点数都是与功能相关的。这个字段也可以用于 IN、SETUP 和 OUT 标志包，以及 PING 特殊标志包。所有的功能设备在端点 0 都必须支持一个控制管道。低速设备的一个功能最多支持 3 个管道，全速和高速设备的的一个功能最多可以支持 16 个 IN 和 OUT 端点。

2. 数据包

（1）格式

一个数据包包括一个 PID、一个包含零个或多个字节数据的数据字段和一个 CRC 字段。有 4 种类型的数据包，分别由不同的 PID 来识别：DATA0、DATA1、DATA2 和 MDATA。其中，前两个数据包 PID（DATA0 和 DATA1）被定义用来支持数据交换同步化。低速设备所支持的最大数据字段为 8Byte，全速设备所支持的最大数据字段为 1023Byte，而高速设备所支持的最大数据字段为 1024Byte。设置阶段的数据包永远都是 8 个 Byte（只包括数据字段），如图 5-5 所示。

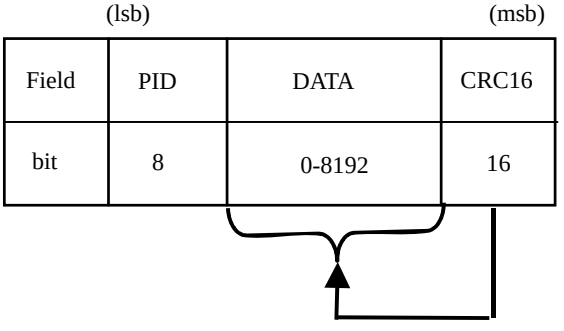


图 5-5　数据包的格式

（2）功能

数据包的功能是用于传输请求和相关信息的，如图 5-6 所示。

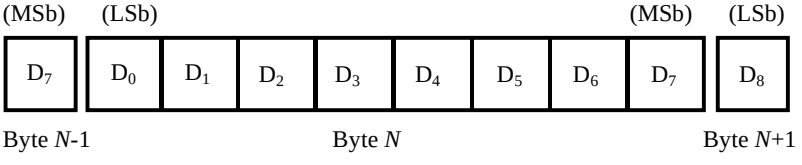


图 5-6　数据字段的格式

发送者：主机。

PID：DATA0。

其他内容：8 个数据字节分成 5 个字段：bmRequestType、bRequest、wValue、wIndex 和 wLength，如表 5-2 所示。

❑ bmRequestType：这个位映射的字段描述了请求的各种特性，包括控制传输的第二个阶段的数据传输方向、请求的类型以及接受者等。这个字段的位 7 指明数据阶段的数据传输方向，0 表示数据从主机传输到设备，1 表示数据从设备传输到主机。字段的位 6 和位 5 表示请求的类型，可以是 11 种标准请求（00），也可以是一个特定 USB 类定义的请求（01），或

者是一个供应商特定的请求（10）。字段的位 4 到位 0 表示请求的接收者，一个请求可以指向设备（00000），一个特定的接口（00001），端点（00010）或设备中其他的元素（00011）。

❑ bRequest：用来指定请求的字节。这个字段的值会受到 bmRequestType 字段的影响。当 bmRequestType 为 00 时，bRequest 的值是 USB 规范定义的 11 种标准请求中的一种；当 bmRequestType 为 01 时，bRequest 的值是一个类特定的请求；当 bmRequestType 为 10 时，bRequest 的值是一个供应商特定的请求。这 11 种标准请求如表 5-3 所示。

表 5-2　　设置阶段数据包数据字段的格式内容

偏移量	字段	大小	数值	描述
0	bmRequestTye	1	位映射	请求特征: D7: 数据传输方向 0=主机至设备 1=设备至主机 D6...5: 类型 0=标准(Standard) 1=类(Class) 2=厂商(Vendor) 3=保留(Reserved) D4..0: 接收者 0=设备(Device) 1=接口(Interface) 2=端点(Endpoint) 3=其他 4..31=保留
1	bRequest	1	数值	具体请求（参见表 5-3）
2	wValue	2	数值	根据不同的请求含义改变
4	wIndex	2	索引或偏移	根据不同的请求含义改变，典型用于传送索引或偏移
6	wLength	2	计数	如果有数据阶段，此字段为数据字节数目

表 5-3　　11 种标准设备请求

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	特征选择符 见(表 5-8)	0 接口 端点	0	无
10000000B	GET_CONFIGURATION	0	0	1	配置值
10000000B	GET_DESCRIPTOR	描述符类型和索引	0 或语言 ID	描述符长度	描述符
10000001B	GET_INTERFACE	0	接口号	1	替换接口
10000000B 10000001B 10000010B	GET_STATUS	0	0 接口号 端点号	2	设备、接口或端点状态

00000000B	SET_ADDRESS	设备地址	0	0	无

续表

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_CONFIGURATION	配置数值	0	0	无
00000000B	SET_DESCRIPTOR	描述符类型和索引	0 或语言 ID	描述符长度	描述符
00000000B 00000001B 00000010B	SET_FEATURE	特征选择符	0 接口号 端点号	0	无
00000001B	SET_INTERFACE	替换设置	接口号	0	无
100000010B	SYNCH_FRAME	0	端点号	2	帧数目

❑ wValue：是两个字节的字段，主机可以用来向设备传递信息。不同的请求可以传送不同含义的信息。例如，在一个 Set_Address()请求中，wValue 的值就是设备的地址。

❑ wIndex：是两个字段的字段，主机可以用它来向设备传递信息。这个字段的值也会随不同的请求而有所变化，一般用于传递索引或偏移量，例如，指定一个接口或端点。当传输一个端点的索引值时，位 0~3 表示该端点的号码，如图 5-7 所示。位 7 如果是 0 表示一个控制或 OUT 端点，位 7 如果是 1 表示一个 IN 端点。当传输的是接口的索引时，位 0~7 是接口号码，所有未使用的位都是 0，如图 5-8 所示。

D7	D6	D5	D4	D3	D2	D1	D0
Direction	Reserved(Reset to zero)			Endpoint Number			
D15	D14	D13	D12	D11	D10	D9	D8
Reserved(Reset to zero)							

图 5-7　指定一个端点时的 wIndex 的格式

❑ wLength：是两个字段，用于指定在控制传输第二阶段传输的数据的长度。如果这个字段是 0，则表示没有数据传输阶段。对于一个输入的请求，设备不会返回比由这个字段所指定值更多的数据字节。对于一个输出的请求，wLength 字段将指定主机发送确切的数据字节。

D7	D6	D5	D4	D3	D2	D1	D0
Interface Number							
D15	D14	D13	D12	D11	D10	D9	D8
Reserved(Reset to zero)							

图 5-8　指定一个接口时的 wIndex 的格式

3. 握手包

（1）格式

如图 5-9 所示，握手包只包含一个 PID 字段。握手包用来报告数据事务的状态，另外可以表示数据成功接收，命令的接收或拒绝，流控制（Flow Control）和停止（Halt）条件。只有支持流控制的事务类型才能返回握手信号。握手包总是在事务的握手段（Phase）中被返回，也可在数据段代替数据被返回。握手包由包字段后 1 个字节的 EOP 确定界限。如果包被解读为合法的握手信号，但没有以 1 个字节后面的 EOP 为终止，则它被认为是无效的，并且被接收器忽略。

(lsb) msb)

Field	PID
bit	8

图 5-9 握手包的格式

一共有 4 种类型的握手包和一个特殊握手包：ACK、NAK、STALL、NYET 和 ERR，它们分别应用于不同的情况下。

（2）功能

握手包的功能是用于传输设备的确认信号。

发送者：设备。

PID：ACK。

其他内容：无。握手包只包含 PID。

5.1.2 数据阶段

如果控制传输存在数据阶段的话，此数据阶段会包括一个或多个 IN 或 OUT 事务。端点描述符会指定每个事务所能携带的数据字节数目。

当数据阶段使用的是 IN 事务时，数据从设备传向主机，例如：主机发出 Get_Descriptor() 请求，则设备向主机传递一个指定的描述符。当数据阶段使用的是 OUT 事务时，数据由主机传向设备，例如：主机发送一个 Set_Report()请求，主机传送一个报告给 HID 类设备。如果设置阶段 wLength 字段的值是 0，则说明该传输没有数据阶段，例如，在 Set_Configuration() 请求中，主机在设置阶段数据包的 wValue 字段内，给设备传送了一个配置号，所以就不再需要数据阶段了。

当所要传输的数据无法装入一个数据包时，数据阶段就需要多个事务。所需的事务总数等于设置事务的 wLength 字段值除以端点描述符中的 wMaxPacketSize 字段的值，并且进位后取整，例如，在 Get_Descriptor()请求中，wLength 字段的值是 18，端点描述符中 wMaxPacketSize 字段的值是 8，则要完成传输需要 3 个数据事务。数据阶段的所有事务都必须是同一个方向的。

数据阶段的每一个 IN 或 OUT 事务都包含标志包、数据包和握手包。控制读写的顺序如图 5-10 所示。

1. 标志包

标志包的功能是指定接收器，并指定该事务是 IN 或者 OUT 事务。

发送者：主机。

PID：如果数据是从设备向主机传送，则 PID 为 IN；如果数据是从主机向设备传送，则 PID 为 OUT。

其他内容：设备和端点地址。

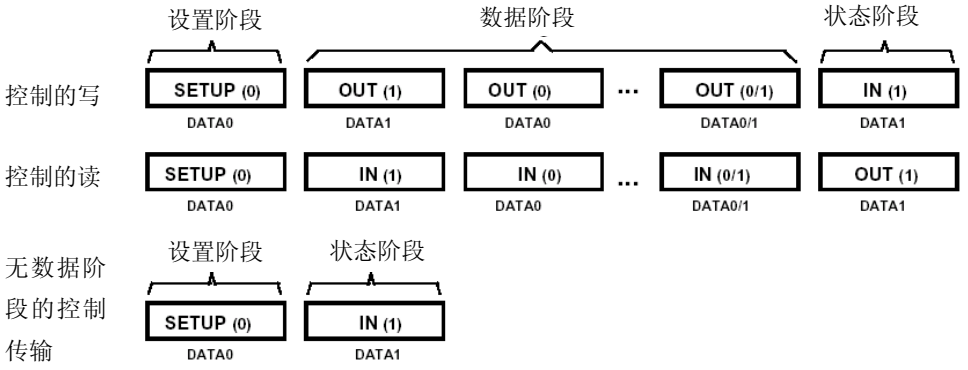


图 5-10　控制读写的顺序

2. 数据包

数据包的功能是传输所有在设置事务的数据包的 wLength 字段指定的数据。

发送者：如果标志包的 PID 是 IN，则发送者是设备；如果标志包的 PID 是 OUT，则发送者是主机。

PID：第一个数据包的 PID 是 DATA1，随后的数据包的 PID 是交替的 DATA0/DATA1。

其他内容：数据。

3. 握手包

握手包的功能是数据包接收者传回状态信息。

发送者：也就是数据阶段中数据包的接收者。如果标志包的 PID 是 IN，则发送者是主机；如果标志包的 PID 是 OUT，则发送者是设备。

PID：任何设备都可以传回 ACK（表示收到有效数据）、NAK（端点正忙）或 STALL（收到不支持的请求或是端点挂起）。收到多个数据包的高速设备可以传回 NYET（表示接受目前事务的数据，但是端点还没有准备好接受其他的数据包），但是主机只能在 IN 事务的握手包中传回 ACK。

其他内容：无，握手包只包含 PID。

补充：当接收者在收到标志包或数据包时检测到了错误，那么它就不会传回握手包。

5.1.3　状态阶段

状态阶段是控制传输的最后一个阶段，是设备用来报告前面设置和数据阶段传输是否成功的阶段，它的功能和事务的交换包类似，而且有时信息就是在状态阶段的握手包中传递的。区别是，状态阶段报告的是整个传输的成功与否，而不是单个事务是否成功。

在有些情况下，例如，在列举过程中，主机会在设备传输完所有数据之前就开始进入状态阶段，当设备检测到这一点时，它就会停止继续发送数据，转而完成状态阶段。

下面详细介绍状态阶段的各个包。

1. 标志包

标志包的功能是用于指定接收器，并指定状态阶段中数据包的流向。

发送者：主机。

PID：与前一事务的数据包有相反的方向。如果整个传输有数据阶段，且数据阶段的标志包 PID 是 OUT，则状态阶段的 PID 是 IN；如果数据阶段的标志包 PID 是 IN，则状态阶段的 PID 是 OUT；如果没有数据阶段，则状态阶段的 PID 也是 IN。

其他内容：设备与端点地址。

2. 数据包

数据包的功能是向数据阶段中数据的接收者指示整个传输的状态。

发送者：如果状态阶段的标志包 PID 是 IN，则发送者是设备；如果状态阶段的标志包 PID 是 OUT，则发送者是主机。

PID：DATA1。

其他内容：主机可能发送零长度的数据包，只包含 PID 和错误校验位。设备也可能传送一个零长度的数据包 ACK（成功）、NAK（忙）或 STALL（端点挂起）。

补充：对大部分的请求而言，零长度的数据包表明请求已经被执行了，但 Set_Address 是个例外，它需要等到状态阶段完成后才能生效。

3. 握手包

握手包的功能是数据阶段中数据的发送者用以指明传输的状态。

发送者：即状态阶段中数据包的接收者，如果标志包的 PID 是 IN，则发送者是主机；如果标志包 PID 是 OUT，则发送者是设备。

PID：设备可能传回 ACK（成功）、NAK（端点正忙）或 STALL（不支持此请求或端点挂起）。主机只能返回 ACK，状态阶段的响应如表 5-4 所示。

表 5-4　　状态阶段的响应

状态响应	控制写传输（在数据相位发送）（Data Phase）	控制读传输（在握手相位发送）（Handshake Phase）
功能完成	零长度的数据包	ACK 握手
功能有错	STALL 握手	STALL 握手
功能正忙	NAK 握手	NAK 握手

其他内容：无，握手包只包含 PID。

补充：状态阶段的握手包是整个传输中最后的事务。当接收者在收到标志包或数据包时检测到了错误，它就不会传回握手包。

5.1.4 错误处理

并不是所有的控制请求都会得到设备正确的响应，请求可能因为设备固件不支持或者端

点暂停等原因而不能被执行。设备固件不支持某个请求，可能是因为固件不能识别请求代码，或者设备能够识别请求代码，但是在设置阶段中其他配置信息与设备所期待的信息不符。这时，都会发生请求错误，设备会发送一个 STALL 握手包来通知主机。设备必须对设置事务响应以 ACK，所有 STALL 握手包必须在下一个数据阶段或状态阶段中传输。

如果主机无法取得预期的信息，或在接收数据时检测到了错误，或端点暂停（Halt），主机就会放弃该传输，然后，主机发送一个新的设置（Setup）标志包，重新建立通信。如果设备在完成上一个控制传输之前收到了一个信息的设置事务的标志包，那么，设备必须放弃上一个传输，转而开始新的控制传输。如果传输使用的是缺省控制管道，而一个新的标志包没有使设备恢复状态，则主机会要求集线器重新配置设备所连接的端口。

主机也可能在完成所有的数据阶段的事务之前，启动一个状态阶段，从而提前结束传输。在这种情况下，设备必须放弃剩下的数据，对状态阶段作出必要的响应。

5.1.5　11 种标准请求

USB 规范所定义的 11 种标准请求如表 5-5 所示。所有的设备都必须对这些请求作出响应，即使是当主机还没有给设备分配一个地址或者还没有配置设备，设备也应该作出响应，尽管得到的响应可能是 STALL。标准请求的数值从 0x00 到 0x0C，但其中有些数值没有使用，作为保留值。大部分的请求都是成对的，即有 Set 的请求就会有相应的 Get 或 Clear 请求。但是，Set_Address()、Synch_Frame()和 Get_Status()请求是例外。USB 2.0 所支持的 8 种描述符类型如表 5-6 所示。

表 5-5　　标准请求代码

请求（bRequest）	数值（Value）
GET_STATUS	0
CLEAR_FEATURE	1
为将来保留使用	2
SET_FEATURE	3
为将来保留使用	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

表 5-6　　描述符类型

描述符类型	数值（Value）

DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5

续表

描述符类型	数值（Value）
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER	8

（1）Clear_Feature

Clear_feature()请求的结构如表 5-7 所示。

表 5-7　　Clear_feature()请求

bmRequestType	bRequest	WValue	Windex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	特征选择符（表 5-8）	0 接口 端点	0	无

这个请求用来清除或禁止一个具体的特性。

wValue 字段中的特征选择符的值必须根据接收者来设定。接收者如果是设备则要用设备特征选择符，是接口就必须用接口特征选择符，是端点的就要用端点的特征选择符。特征选择符与接收者的对应关系如表 5-8 所示。

表 5-8　　标准特征选择符

特征选择符	接收者	数值
DEVICE_REMOTE_WAKEUP	设备	1
ENDPOINT_HALT	端点	0
TEST_MODE	设备	2

一个 Set_Feature()请求如果选择了一个不能被清除的特征，或者一个根本不存在的特征，或者特征所属于的接口或端点不存在，都会引起设备响应请求错误。

如果 Set_Feature()请求的 wLength 字段值为 0，则表示设备的行为没有被指定。

3 种状态下的设备行为如下：

- 缺省状态（Default State）：当设备处于缺省状态，收到这个请求时，设备的行为没有被指定。
- 地址状态（Address state）：在设备处于地址状态时这个请求是有效的，但如果该请求所指向的是接口或是非零端点，则会引起请求错误。
- 配置状态：当设备处于配置状态下，该请求是有效的。

注意：特征 TEST_MODE 是不能被这个请求所清除的。

（2）Get_Configuration

Get_Configuration()请求的结构如表 5-9 所示。

表 5-9　　Get_Configuration()请求

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_CONFIGURATION	0	0	1	配置值

Get_Configuration()请求返回当前设备的配置值。

如果返回的值是 0，则表明设备没有被配置。如果这个请求的 wValue、wIndex 和 wLength 这 3 个字段值与表 5-9 中的值不同，则表示设备的行为没有被定义。

3 种状态下的设备行为如下。

- 缺省状态（Default State）：当设备处于缺省状态，收到这个请求时，设备的行为没有被指定。
- 地址状态（Address State）：必须返回 0 值。
- 配置状态（Configuration State）：当前配置的非零 bConfigurationValue 值必须被返回。

（3）Get_descriptor

Get_descriptor()请求的结构如表 5-10 所示。

表 5-10　　Get_descriptor()请求

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_DESCRIPTOR	描述符类型和索引	0 或语言 ID	描述符长度	描述符

Get_descriptor()请求返回指定的描述符。

wValue 字段的高字节用于指定描述表类型，低字节表示描述符的索引。当一个设备包含有几个同类型的描述符时，描述符索引用于指定一个特定的描述符（低字节用于作描述符索引的情况只能使用于配置和字符串描述符）。对于其他标准描述符，描述符索引值只能被设置为 0。

wIndex 字段用于指定字符串描述符的语言类型（Language ID），如果是其他描述符的话，该字段就设为 0。wLength 指定要返回多少字节，如果描述符长度大于 wLength 字段所指定的值，则只有 wLength 字段所指定的值的描述符的数据被返回；如果描述符的长度比 wLength 字段值小，则主机将会发送一个短包来标志传输的结束。一个短包被定义成一个长度小于最大有效载荷的长度或一个空（NULL）包。

对一个设备的标准请求包括 3 种类型的描述符：设备（device_qualifier）、配置（other_speed_configuration）及字符串。一个高速设备支持 device_qualifier 描述符，用于返回关于设备非当前运行速度的信息。一个对配置描述符的请求会一次返回配置描述符、所有的接口描述符和所有接口的端点描述符。第一个接口描述符紧跟着配置描述符，第一个接口的端点描述符随后。如果有其他的接口与端点，它们的描述符跟在第一个接口与端点描述符之后。与类有关的描述符和厂商定义的描述符跟在标准描述表之后。

所有的设备必须提供一个设备描述符和至少一个配置描述符，如果设备不支持所请求的

描述符，则返回请求错误。

3 种状态下的设备行为如下：

- 缺省状态（Default State）：当设备处于缺省状态时，此请求有效。
- 地址状态（Address State）：当设备处于地址状态时，此请求有效。
- 配置状态（Configuration State）：当设备处于配置状态时，此请求有效。

（4）Get_Interface

Get_Interface()请求的结构如表 5-11 所示。

表 5-11　　Get_Interface()请求

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000001B	GET_INTERFACE	0	接口号	1	替换接口

Get_Interface()请求返回该请求所指定接口的被选中的可选设置。

有些 USB 设备的接口配置有互斥的设置。这个请求使得主机来决定当前被选择的可替换设置。如果 wValue 和 wLength 两个字段的值没有按照表 5-11 中进行设置，那么设备的行为没有定义。如果所指定的接口不存在，那么设备以请求错误来响应。

3 种状态下的设备行为如下。

- 缺省状态（Default State）：当设备处于缺省状态，收到这个请求时，设备的行为没有被指定。
- 地址状态（Address State）：设备返回请求错误。
- 配置状态（Configuration State）：此请求有效。

（5）Get_status

Get_status()请求的结构如表 5-12 所示。

表 5-12　　Get_status()请求

BmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B 10000001B 10000010B	GET_STATUS	0	0 接口号 端点号	2	设备、接口 或端点状态

Get_status()请求返回所指定接收者的状态。期望的接收者是由 bmRequestType 字段的 Recipient 位指定的。返回的数据就是指定的接收者的当前状态。

如果 wValue 和 wLength 字段的值与表 5-12 中所指定的不符，或者在 Get_status()请求中 wIndex 的值不为 0，设备的行为没有被指定。如果所指定的接口或端点不存在，那么，设备响应请求错误。

一个对设备的 Get_Status()请求返回的信息如图 5-11 所示。

D7	D6	D5	D4	D3	D2	D1	D0
Reserved(Reset to zero)						Remote Wakeup	Self Powered
D15	D14	D13	D12	D11	D10	D9	D8
Reserved(Reset to zero)							

图 5-11　设备 Get_Status 请求传回的信息

其中，Self Powered 字段指示设备当前是否是自我供电。如果 D0 为 0，则表示设备是总线供电的；如果 D0 被设成 1，则表示设备是自我供电的。此字段的值不应当被 SetFeature()或 Clear_Feature()请求所改变。

Remote Wakeup 字段表明此设备当前是否支持远程唤醒，对于有支持远程唤醒能力的设备，上电时该字段的缺省值是禁止的（disabled）。如果 D1 被设置为 0，远程唤醒能力则被禁止；如果设置为 1，设备则具有远程唤醒的功能，此字段可被 Set_Feature()和 Clear _Feature()请求使用 DEVICE-REMOTE-WAKEUP 特征选择符所修改。设备复位时此字段被设成 0。

一个对接口的 Get_Status()请求返回的信息如图 5-12 所示。

D7	D6	D5	D4	D3	D2	D1	D0
Reserved(Reset to zero)							
D15	D14	D13	D12	D11	D10	D9	D8
Reserved(Reset to zero)							

图 5-12　接口 Get_Status()请求返回的信息

一个对端点的 Get_Status()请求返回如图 5-13 所示的信息。

D7	D6	D5	D4	D3	D2	D1	D0
Reserved(Reset to zero)							Halt
D15	D14	D13	D12	D11	D10	D9	D8
Reserved(Reset to zero)							

图 5-13　对端点的 Get_Status()请求所返回的信息

Halt 特性应当在所有的中断及批量端点中实现。如果端点当前被停止（Halted），那么，这个 Halt 特性就被设为 1，否则为 0。Halt 特性可以由 Set Feature(ENDPOINT-HALT)请求来设置，一旦被 Set_Feature()请求设置，端点表现出的停止行为就会像这个字段由硬件条件设置的一样。如果引起暂停（Halt）的条件去除了，用 Clear_Feature (ENDPOINT-HALT)请求清除 Halt 特性会导致端点再也不会返回 STALL 信号了。对于使用数据交替（Date toggle）的端点，不管一个端点的 Halt 特性是否已被设置，一个 Clear_Feature（ENDPOINT-HALT）总会导致（date toggle 数据交替）被重新初始化成 DATA0。Halt 特性在收到 Set_Configuration()

或 Set_Interface()请求后总会被复位成 0。

Halt 特性不要求也不建议在缺省控制通道实现，然而，设备可设置缺省控制通道的 Halt 特性来反映一个功能出错的状态。如果这个特性被设置，设备则对除 Get_Status()、Set_Feature()、Clear_Feature()之外的请求返回 STALL 信号，设备可以不对类有关的及厂商定制的请求返回 STALL 信号。

3 种状态下的设备行为如下。

- 缺省状态：当设备处于缺省状态，收到这个请求时，设备的行为没有被指定。
- 地址状态：如果所指定的是接口或是一个非 0 端点，设备将返回请求出错。
- 配置状态：如果所指接口或端点不存在，返回请求错误。

（6）Set_Address

Set_Address()请求的结构如表 5-13 所示。

表 5-13　　Set_Address()请求

BmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_ADDRESS	设备地址	0	0	无

Set_Address()请求为以后访问设备而设置地址。wValue 指定了所要设置成的设备地址值。

正如前面所讲述的那样，请求实际上可能会引发最多 3 个阶段。在第一阶段，设置（Setup）包被送至设备，在第二个可选的阶段，数据在设备与主机之间传送，在最后一个阶段，状态信息在主机与设备之间传送。数据与状态传送的方向要看是主机发数据给设备还是设备发数据给主机。状态的传送方向总是与数据传送方向相反的，如果没有数据传输阶段，则状态由设备传向主机。

Setup 包传送以后的两个阶段的地址保持与 Setup 包传送阶段的一致。USB 设备只有在 Status 阶段过后才能改变设备地址。

注意：在这方面 Set_Address()请求不同于其他请求。其他请求总是在状态传送阶段之前完成指定操作的。

如果所设定的设备地址大于 127、wIndex 或 wLength 非零，则设备的行为没有定义。

设备对设置值是 0 的 SetAddress()请求的响应无定义。

3 种状态下的设备行为如下。

- 缺省状态：如果地址值非 0，则设备将进入地址状态，否则地址仍留在缺省态（此非出错状态）。
- 地址状态：如果指定的地址值为 0，那么设备就进入缺省状态，否则仍留在地址状态但使用新指定的地址。
- 配置状态：在此状态下设备对此请求的响应无定义。

（7）Set_Configuration

Set_Configuration()请求的结构如表 5-14 所示。

表 5-14　　Set_Configuration()请求

bmRequestType	Brequest	WValue	wIndex	wLength	Data

00000000B	SET_CONFIGURATION	配置数值	0	0	无

Set_Configuration()请求用于设置设备的配置。

wValue 字段的低字节指定想要的配置值，这个配置值必须为 0 或与配置描述符中的一个配置相匹配。如果配置值为 0，则设备被置于地址状态，wValue 字段的高字节保留。

如果 wIndex、wLength 或 wValue 的高字节非 0，则设备对这个请求的响应无定义。

3 种状态下的设备行为如下。

- 缺省状态：当设备处于缺省状态，收到这个请求时，设备的行为没有被指定。
- 地址状态：如果所指定的配置值为 0，设备停留在地址状态。如果所指定的配置与描述表中的一个值相匹配，那么这个配置就被选中，并且设备转到配置状态。否则返回请求错误。
- 配置状态：如果配置值为 0，设备进入地址状态。如果配置值与配置描述符中的一个配置相匹配，则设备仍留在配置状态，但用新配置值，否则返回请求错误。

（8）Set_Descriptor

Set_Descriptor()请求的结构如表 5-15 所示。

表 5-15　　Set_Descriptor()请求

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_DESC-RIPTOR	描述表类型和索引	0 或语言 ID	描述符长度	描述符

Set_Descriptor()请求的 wValue 字段的高字节指定了描述符类型，低字段指定了描述符索引。当一个设备包含有几个同类型的描述符时，描述符索引用于指定一个特定的描述符（描述符索引只能用于配置和字符串描述符）。例如，一个设备描述符可以包含几个配置描述符。对于其他标准描述符，描述符索引值只能被设置为 0。

wIndex 字段指定了字符串描述符的语言 ID，如果是其他描述符，该字段应该设为 0。

wlength 字段指定了从主机传向设备的数据字节数目。如果设备不支持该请求，则设备返回一个请求错误。

3 种状态下的设备行为如下。

- 缺省状态：当设备处于缺省状态，收到这个请求时，设备的行为没有被指定。
- 地址状态：如果设备支持这个请求，那么当设备处于地址状态时，该请求有效。
- 配置状态：如果设备支持这个请求，那么当设备处于配置状态时，该请求有效。

（9）Set_feature

Set_Feature()请求的结构如表 5-16 所示。

表 5-16　　Set_Feature()请求

bmRequestType	Brequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	SET_FEATURE	特征选择符	0 接口号 端点号	0	无

Set_Feature()请求用于指定一个特定的特征。

wValue 字段中的特征选择符的值必须根据接收者来设定适当的值。接收者是设备则必须用设备特征选择符，是接口则必须用接口特征选择符，是端点则必须用端点的特征选择符（参照表 5-8 特征选择符与接收者的对应关系）。

特征 TEST_MODE 只能用于设备接收者，并且，wIndex 字段的低字节必须设为 0。设置这个特性将会导致设备上行接入端口进入测试模式。如果请求包含一个无效的测试选择符，则设备就会响应请求错误。向测试模式的转变必须在请求状态阶段完成后的 3ms 之内完成。高速设备的缺省、地址和配置状态都必须支持 TEST_MODE 特征。如果 Set_Feature()请求指定的是一个不存在的接口或端点，则设备返回一个请求错误。测试模式选择符如表 5-17 所示。

表 5-17　　测试模式选择符

数值	描述
0x00	保留
0x01	Test_J
0x02	Test_K
0x03	Test_SE0_NAK
0x04	Test_Packet
0x05	Test_Force_Enable
0x06-0x3F	保留给标准测试选择符
0x3F-0xBF	保留
0xC0-0xFF	保留给标准测试选择符

如果 wLength 字段的值不是 0，那么设备的行为没有指定。如果所指定的接口或是端点不存在，那么设备会响应请求错误。

3 种状态下的设备行为如下。

- 缺省状态：当设备处于缺省状态时，它必须能够接收一个 Set_Feature(TEST_ MODE，TEST_SELECTOR)请求。如果收到其他的 Set_Feature 请求，则设备的行为没有制定。
- 地址状态：如果指定了一个接口或者是非 0 端点，设备会响应请求错误。
- 配置状态：当设备处于配置状态时，该请求有效。

（10）Set_Interface

Set_Interface()请求的结构如表 5-18 所示。

表 5-18　　Set_Interface()请求

bmRequestType	bRequest	wValue	WIndex	wLength	Data
00000001B	SET_INTERFACE	替换设置	接口号	0	无

Set_Interface()请求将使主机为指定的接口选择一个设置。

如果在 USB 设备接口配置中包含有互斥设置，则此请求将让主机选择所要的设置。如果一个设备的特定接口只支持缺省设置，在状态阶段设备将返回 STALL；如果所指定的接口或可替换设置不存在，设备将返回请求错误；如果 wLength 不为 0，设备行为无定义。

3 种状态下的设备行为如下。

- 缺省状态：当设备处于缺省状态，收到这个请求时，设备的行为没有被指定。
- 地址状态：设备返回请求错误。
- 配置状态：当设备处于配置状态，该请求有效。

（11）Synch_Frame

Synch_Frame()请求的结构如表 5-19 所示。

表 5-19　　Synch_Frame()请求

bmRequestType	BRequest	wValue	wIndex	wLength	Data
100000010B	SYNCH_FRAME	0	端点号	2	帧数目

Synch_Frame()请求用于设置或报告一个端点的同步帧。

如果一个端点支持等时传输，那么端点将会根据某一特定的模式来以变长方式传送每一帧。主机与端点必须在重复模式出现在哪一帧上的问题上达成一致。模式开始帧的序号由设备返回给主机。

如果一个高速设备支持 Sync_Frame()请求，那么它必须在内部将自己同步到 0 号微帧，并且要有经典帧的时间概念。因为只有帧数（而不是微帧）才能用于同步和被设备端点所报告。

这个值仅用于隐式模式的等时数据传输。如果 wValue 字段不等于 0，或 wLength 字段不等于 2，则表示设备的行为没有定义；如果所指的端节点不支持此请求，设备将返回一个请求错误。

3 种状态下的设备行为如下。

- 缺省状态：当设备处于缺省状态，收到这个请求时，设备的行为没有被指定。
- 地址状态：设备返回请求错误。
- 配置状态：当设备处于配置状态，该请求有效。

5.1.6　类特定请求

除了 11 种标准的设备请求外，USB 还支持类和供应商特定的请求。

当一组设备或功能有类似的属性，或者是提供或请求类似的服务时，用户就可以将它们归为一类，并在类规范中定义只属于它们的属性和服务。USB 实施者论坛上发布了这些类规范，它们是由这个领域的专家成员组开发的。

使用类可以简化开发流程，因为 Windows 和其他的操作系统提供了很多通用类的驱动程序，如果用户所开发的设备正好属于其中的一类，那么用户就不必自己编写驱动程序，用户只需编写设备固件就可以与系统提供的驱动进行通信。如果用户想给自己的设备附加新的功能，那么，可以开发一种 filter driver 驱动程序，这样将大大降低开发难度。

类规范可以定义类中设备的标准描述符、类特定描述符、接口、端点以及类特定控制请求的项目。例如，集线器类的设备描述符包含一个 0x09 的 bDeviceClass 数值，用于表示此设备属于集线器类。集线器也可以有集线器类特定描述符。同时，集线器也支持类特定请求。例如，Get_Port_Status()请求，可以得到集线器连接端口的状态信息。

除了集线器类以外，USB 实施者论坛上还定义了很多其他设备类规范。但是，这并不是说有了类规范，操作系统就一定会包含它的驱动程序。接下来，将介绍部分设备类，以及操作系统的支持情况。

- 音频设备（Audio Device）：传输音频和语音以及相关控制的设备。Windows 98 及以后的版本中都包含一个音频类的驱动程序。Windows 2000 还包含一个 MIDI 驱动程序。
- 通信设备（Communication Device）：电话、调制解调器以及其他电讯传输设备。Windows 98 及以后版本中都包括一个体制解调驱动程序。
- 人机接口设备（Human Interface Device，HID）：键盘、鼠标、游戏杆或者其他以中等速度，使用中断或控制传输与主机进行大量数据传输的设备。Windows 98 及以后的版本包括一个 HID 1.0 驱动程序。Windows 98 SE 之后包含一个 HID 1.1 驱动程序，HID 1.1 支持控制输出传输。
- 海量存储设备（Mass Storage）：CD-ROM、录音机和移动存储设备等。Windows 2000 包含一个海量存储驱动程序（usbstor.sys）。
- 打印机（Printer）：打印机接口。Windows 2000 包含一个打印机驱动程序（usbprint.sys），此驱动也可用在 Windows 98 中。

5.1.7 供应商特定请求

在控制传输中，一个供应商也可以为它生产的某个设备定义特别的请求，从而为设备提供特殊服务。为了在控制传输中使用供应商这种特定请求，需要进行以下环节。

- 在设置阶段和可选的数据阶段，定义所需的供应商特定字段。将设置阶段的数据包的位 6 和位 5 设置成 10，表示一个供应商特定的请求。
- 要在设备固件中添加程序以检测请求号，并知道如何进行响应。如果用户有标准请求的程序代码，那么可以把它用作供应商特定请求的模型。
- Windows 包含的通用驱动程序并不能使主机应用程序通过它传送供应商特定的请求，所以，供应商必须自行开发驱动程序，才能传送自定义的请求。

5.2 实际应用

下面，将以 CYPESS 公司的 FX 2 为例，详细讲述 USB 设备如何与主机之间通过端点 0 来完成控制传输的过程。

5.2.1 简介

从前面几章的介绍可以知道，所有的控制传输都是通过端点 0 来实现的。在 USB 系统中端点 0 具有特殊重要的作用，因此，它具有一个自己的名字“控制端点”，每一个 USB 设备都需要它。USB 主机使用特殊的 SETUP 标志来开始设备控制的信号传输，只有控制端点才

能够接收到这些信息。

USB 主机通过端点 0 送出一系列的标准设备请求。FX 2 为端点 0 的管理提供了一系列强有力的硬件支持。

端点 0 是 FX 2 支持的惟一一个控制端点。端点 0 是双向的，具有一个 64 字节的缓冲：EP0BUF。该缓冲是由固件来控制的，在控制传输阶段作为批量缓冲来使用。此外，它还有一个 8 字节的缓冲，称为 SETUPDAT，这一个缓冲只有端点 0 才有，它会保存在控制传输的 SETUP 阶段到达的数据。这会减轻 FX 2 在跟踪控制传输的 3 个阶段时的负担（SETUP、DATA、STATUS）。FX 2 同样会对不同的阶段产生不同的中断请求。对于 USB 主机来说端点 0 总是可用的。

5.2.2　控制端点 EP0

控制端点接收特殊的 SETUP 数据包，在该数据包中包含了一个 8 位的数据结构，它提供控制事务处理中主机所需要的信息。控制传输包含了一个最终的 STATUS 阶段，它由标准的 PID 构成（IN/OUT、DATA 1、ACK/NAK），如图 5-14 所示。

有的控制事务处理在其 8 位的 SETUP 数据包中包含了所需的所有数据，而其他的一些控制传输则需要更多的输出或输入数据，以至于 8 个字节的空间远远不能满足。那么，这些事务处理就是用类似批量传输的方式来进行数据传输的。在图 5-14 中 DATA 阶段看起来更像一个批量传输，正如在批量传输中一样，端点 0 的字节计数器的值必须被载入每一个数据传输阶段的 ACK 域中。

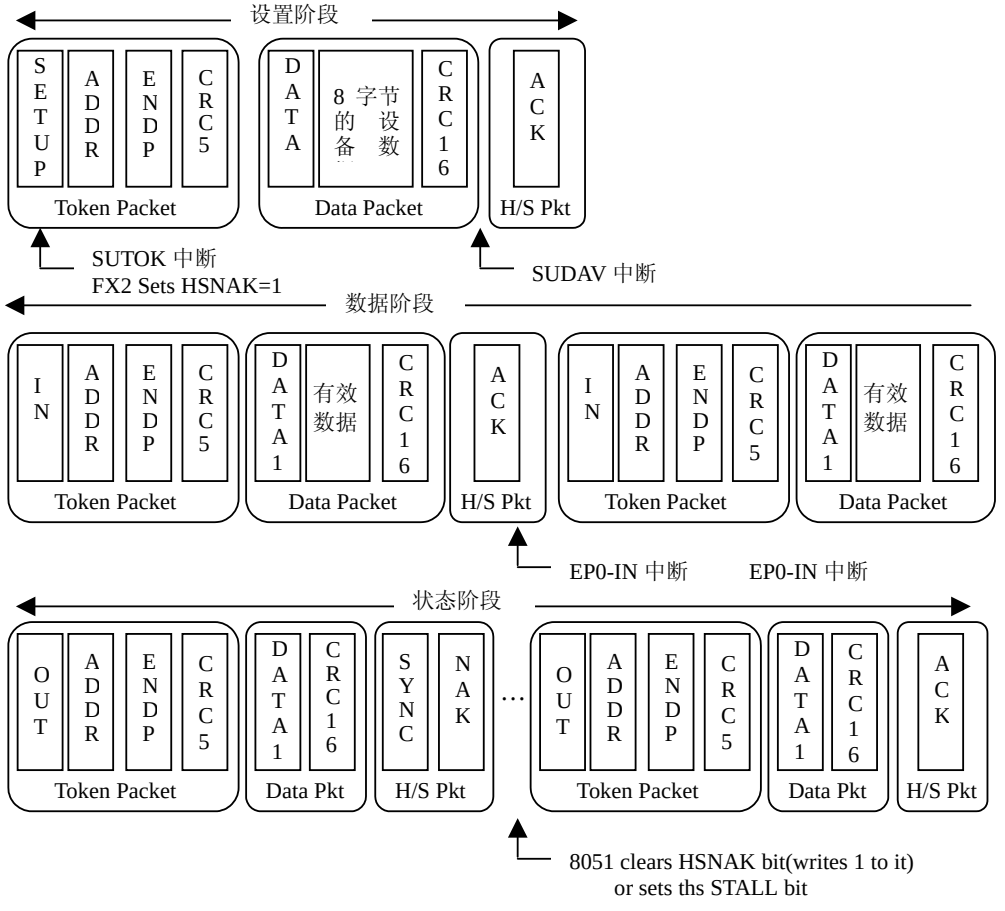


图 5-14　USB 的控制传输

状态（STATUS）阶段包含了一个空的数据包，数据包的传输方向与数据阶段相反，如果没有数据阶段则该数据包的方向为 IN。根据这一数据包的状态，设备以 ACK 或 NAK 来回应控制传输请求。HSNAK 位会在整个控制传输过程中保持下去，直到设备有时间来回应请求。举例来说，如果主机发出了一个 Set_Interface()请求，而此时固件又在执行一项非常重要的工作，例如调节内部工作模式或是初始化一个端点。在这一过程中，主机发出握手信号（在 STATUS 阶段），而 FX 2 以 NAK 来回应，意思是“忙，请等候”。那么，在固件完成了当前的工作之后，它会清除 HSNAK 位（通过在其中写入“1”），这将会引导 FX 2 在 STATUS 阶段作出回应，来结束 STATUS 阶段。这样的握手就保证了主机在未完成对一个设备的配置之前不会使用使用该设备配置。

为了在 DATA 或 STATUS 阶段对一个端点执行暂停（在 SETUP 阶段永远不会暂停）固件必须将 STALL 和 HSNAK 位置为“1”。

在一些控制传输过程中不需要数据阶段。也就是说，在处理 SETUP 数据的过程中就应该检查 SETUP 数据中的长度域（SETUPDAT 中的 8 字节缓冲），然后使端点 0 做好接受数据的准备（通过下载 EP0BCH:L 中的数据）。

在 SETUP 数据包到达时会产生两个中断，如图 5-15 所示。

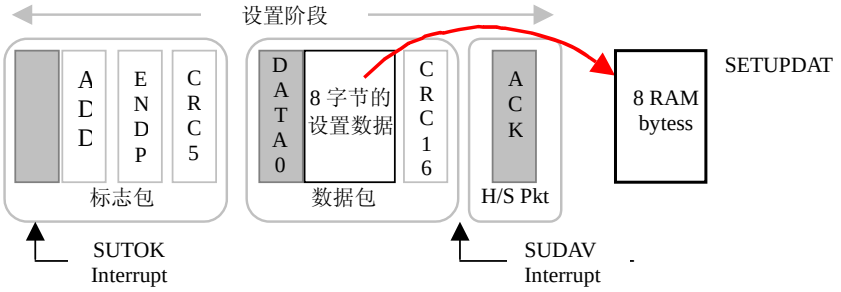


图 5-15　两个与通过端点 0 的控制传输有关的中断

在检测到标志控制传输开始的 SETUP 标志到达时，FX 2 发出 SUTOK 中断请求（Setup Token），这一中断通常被用来调试程序。在 SETUP 阶段的数据被无误地接受并且传送到 SETUPDAT 缓冲中后，FX 2 会发出 SUDAV 中断（Setup Data Available）。如果在到达的数据中有错误，FX 2 会自动关注任何重试的信号，这两个中断位都必须由固件来清除。

固件通过直接检查 SETUPDAT 中的 8 个数据字节或是将该数据字节传送到缓冲中等候下一步的处理这两种方法来响应 SUDAV 中断。响应 SETUP 的数据必须具有较高的优先权，这是因为 USB 规范规定控制传输必须被接受，而且不能以 NAK 来回应。然而，有可能在控制传输到达时，固件仍然在处理上一个控制传输。在这种情况下，前一个控制传输应该被忽略，而转向当前的传输服务。SUDOK 中断给出了进一步的警告：新的控制传输即将在 8 个字节的 SETUPDAT 中写入新的数据。

如果固件使端点 0 处于暂停状态（通过使 STALL 和 HSNAK 位为 1），则在新的 SETUP 标志到达时，FX 2 会将暂停位清零。

就像所有的 FX 2 的中断一样，SUTOK 和 SUDAV 位可以直接被测试，也可以被固件清除（通过在其中写入 1）。与通过端点 0 的控制传输有关的寄存器如图 5-16 所示。

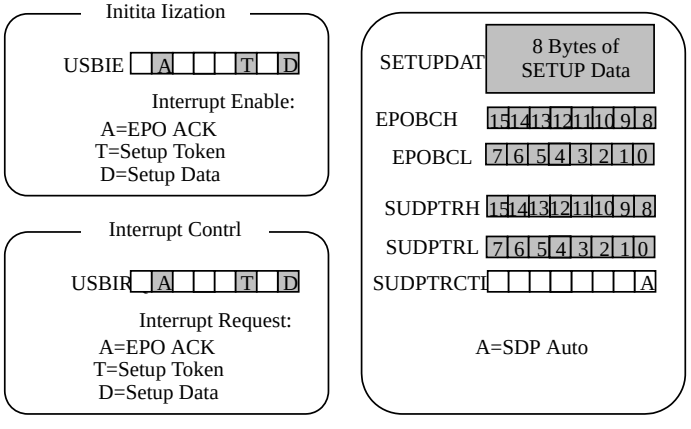


图 5-16　与控制传输有关的寄存器

这些寄存器与通过端点 0 通信的批量传输也有关联。

在 USBIE 寄存器中有两个位控制着 SETUP Token（SETOK）和 SETUP Data Available 这两种中断是否能够发出，而实际的中断请求位则位于 USBIRQ 寄存器中。

FX 2 将 8 个 SETUP 字节送入位于 SETUPDAT 的 RAM 的 8 个字节中。一个 16 位的指针：SUDPTRH:L，帮助硬件控制 CONTROL IN 的传输。

5.2.3 USB 请求

在 USB 规范中定义了一系列标准的设备请求。在 FX 2 的系统中，当固件控制了端点 0 时（RENUM=1），FX 2 只对其中的一个请求负责响应（Set Address），而其他的请求则是由固件来支持。固件对包含在 SETUP 数据包中，并且存储在 SETUPDAT 中的 8 个字节进行解码，根据所得的值来作出相应的动作。表 5-20 定义了这 8 个字节。

表 5-20　　对在 USB SETUP 数据包中的 8 个字节

字节序号	域	含义
0	bmRequestType	请求类型，方向，存储位置
1	bRequest	实际的请求（如表 5-21）
2	wValueL	16 位的值，根据 bRequest 的值作出反应
3	wValueH	
4	wIndexL	16 位的域，根据 bRequest 的值作出反应
5	wIndexH	
6	wLengthL	如果有数据阶段，标志数据的字节数
7	wlengthH	

表 5-20 的第一列给出了在 SETUPDAT 寄存器中的偏移量；第二列给出了请求中的不同字节，在这里“bm”表示位图（Bit Map），“b”表示字节，“w”表示“字”。

表 5-21 给出了由 bRequest 定义的不同值，以及固件应该如何响应每一个请求。

表 5-21　　固件掌握 USB 的设备请求（RENUM =1）

bRequest	名称	FX2 的动作	固件的响应
0x00	Get_Status()	SUDAV 中断	支持 RemWU、SelfPwr 和停止位
0x01	Cleat_Feature()	SUDAV 中断	清除 RemWU、SelfPwr 和停止位
0x02	（reserved）	无	暂停 EP0
0x03	Set_Feature()	SUDAV 中断	置 RemWU、RemWU、SelfPwr 和停止位
0x04	（reserved）	无	暂停 EP0
0x05	Set_Address()	更新 FNADDR 寄存器	无
0x06	Get_Descriptor()	SUDAV 中断	支持通过端点 0 的表格输入
0x07	Set_Descriptor()	SUDAV 中断	由应用程序决定
0x08	Get_Configuration()	SUDAV 中断	送出当前的配置信息
0x09	Set_Congfiguration()	SUDAV 中断	改变当前的配置信息
0x0A	Get_Interface()	SUDAV 中断	支持从 RAM 来的改变信息

续表

bRequest	名称	FX2 的动作	固件的响应
0x0B	Set-Interface()	SUDAV 中断	改变可变的设置信息
0x0C	Sync_Frame()	SUDAV 中断	支持一个从端点 0 输入的帧序号

Vendor Requests		
0A0	上传/下载 RAM	—
0A1	SUDAV 中断	由 Cypress Semiconductor 保留
除了 0A0	SUDAV 中断	取决于应用

在重新列举的过程中（RENUM=1），FX 2 将所有的 USB 请求都通过 SUDAV 中断传递给固件，并提出了一个请求 Set_Address()。

FX 2 要提供一个供应商特别请求：“Firmware Load”，0xA0（作为 bRequest 的值，在请求的第 0 个字节 bmRequestType 为“x10xxxxx”时，即暗示一个供应商特别请求时，0xA0 的值是有效的）。固件下载的请求是永远有效的，所以，即使在重列举之后下载的特征仍然有可能被用到，同时，用户不应该在其他地方使用这些特征。

为了与今后的版本保持兼容性，供应商请求 0xA0-0xAF 被 Cypress 公司保留。

1. 获取状态

USB 规范定义了 3 种状态请求，用于接口的第 4 种请求在规范中被作为保留。这 4 种状态分别如下。

- ❑ 远端唤醒（设备请求）。
- ❑ 自我供电（设备请求）。
- ❑ 暂停（端点请求）。
- ❑ 接口请求（保留）。

FX 2 会自动发出 SUDAV 中断来告诉固件对 SETUP 数据包进行解码，并提供相应的状态信息。正如图 5-17 所解释的，固件根据拷入 RAM 中 SETUPDAT 的 8 个字节进行解码的结果，对 SUDAV 中断作出响应。固件通过两个方面来获得 Get_Status()请求（bRequest），即下载两个字节到 EP0BUF 缓冲中，以及用值 0x0002 装载字节计数器 EP0BCH:L。然后，FX 2 通过发送这两个字节来响应 IN 标志。最后，固件清除 HSNAK 位，这将会引导 FX 2 对传输的状态阶段作出响应。

用于 Get_Status()请求的 8 个 SETUP 字节如表 5-22 所示。

表 5-22　　获得设备状态（远端唤醒和自我供电）

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x80	输入，设备	在 EP0BUF 中装载两个字节。 字节 0：位 0=Self-Powered； 位 1＝Remote Wakeup。 字节 1：0
1	bRequest	0x00	“Get_Status()”	
2	wValueL	0x00		
3	wValueH	0x00		
4	wIndexL	0x00		
5	wIndexH	0x00		

续表

字节序号	域	值	含义	固件的响应
6	wLengthL	0x02	两个字节的请求	
7	wLengthH	0x00		

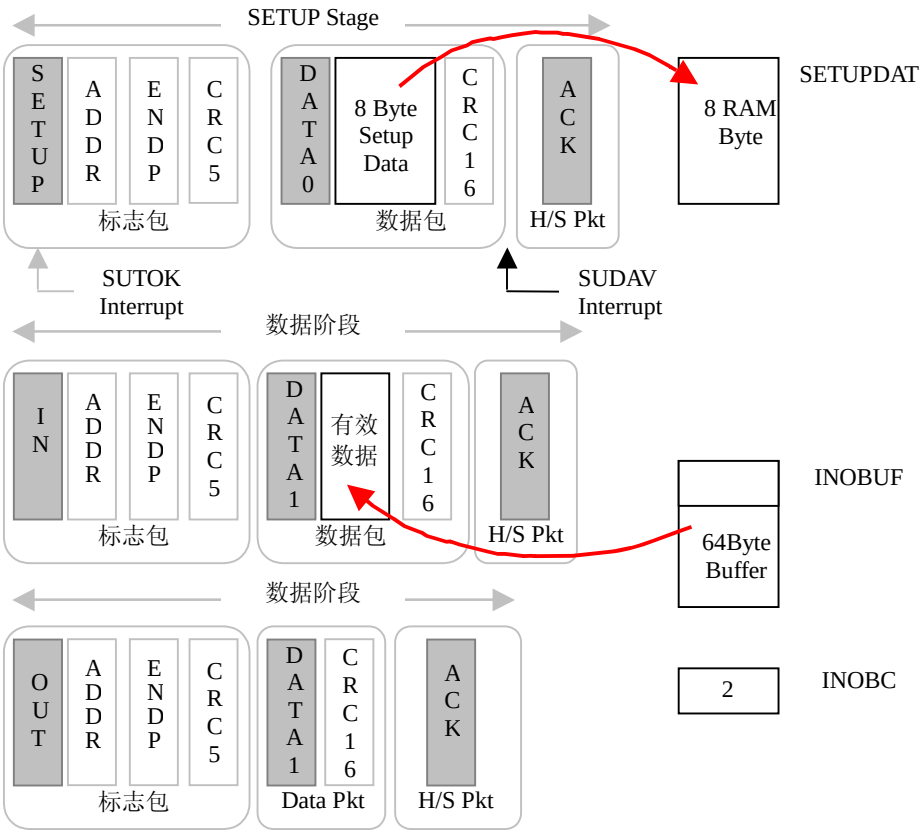


图 5-17　对于 Get_Status()请求的数据流动

获得设备状态请求需要查询两个状态位，即“Remote Wakeup”和“Self-Powered”。远端唤醒位表示设备有无远端唤醒的能力，自我供电位表示设备能否自我供电。

固件通过在 EP0BUF 中写入两个字节的数来给出这两位的信息，然后装载一个值为 0002 的字节计数器到 EP0BCH:L 中。

每一个端点都在其 EPxCS 寄存器中有一个暂停位（STALL）。如果该位被置位，则以 STALL 来回应任何访问该端点的请求。Get_Status(Endpoint)请求会根据该请求中第 4.5 个字节来决定究竟是哪个端点来作出 STALL 回应。要注意在端点序号 EP 字节中的第 7 位给出了数据的方向（0=OUT，1=IN），如表 5-23 所示。

端点 0 是一个控制端点，被 USB 规范定义为双向端点。但是，它只有一个暂停位。

USB 的暂停的握手信号用于暗示是否有非预期的事件发生。例如，如果主机请求一个无效的可变设置，或是传送数据到一个不存在的端点，那么设备将以暂停（STALL）回应。

暂停被用于所有的端点类型，但要除了同步传输（ISOCHRONOUS），因为这种方式并不需要握手信号。每一个 FX 2 的批量传输端点都有其自己的暂停位。固件通过将端点的 EPCS 寄存器中的 STALL 位置为“1”来使该端点处于暂停状态。主机通过使用 Set_Feature(Stall)或是 Clear_Feature(Stall)请求来使端点进入或是退出暂停状态。

表 5-23　　获得端点的状态（Status-Endpoint）（暂停位）

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x82	输入，端点	将两个字节装入 EP0BUF。 字节 0：位 0＝对于端点 *n* 的暂停位 字节 1：0
1	bRequest	0x00	“Get_Status()”	
2	wValueL	0x00		
3	wValueH	0x00		
4	wIndexL	EP	000－008：OUT0-OUT8	
5	wIndexH	0x00	080-088：IN0-IN8	
6	wLengthL	0x02	两个字节的请求	
7	wLengthH	0x00		

设备可能也需要自己决定是否要进入暂停状态。例如，在服务于端点 0 的设备请求的过程中，当一个未定义或是不支持的请求被发出时，固件就应该使端点 0 处于暂停状态。

一旦固件暂停了一个端点，在主机发出 Clear_Feature(Stall)请求之前不应该使端点退出暂停状态。惟一一个例外是端点 0，它只对当前的事务处理报告一个暂停事件，之后它便会自动退出暂停。这样做是为了阻止端点 0 这个默认的控制端点对设备请求作出响应，如表 5-24 所示。

表 5-24　　获得接口状态

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x81	输入，端点	将两个字节装入 EP0BUF。 字节 0：0 字节 1：0
1	bRequest	0x00	“Get_Status()”	
2	wValueL	0x00		
3	wValueH	0x00		
4	wIndexL	EP		
5	wIndexH	0x00		
6	wLengthL	0x02	两个字节的请求	
7	wLengthH	0x00		

Get_Status(Interface)的请求的响应比较简单：固件通过 EP0BUF 返回两个 0 字节，清除 HSNAK 位。

2. 设置特征

设置特征被用来使设备获得远端唤醒功能，或是使一个端点处于暂停状态。在这一过程中，并不需要数据阶段，如表 5-25 所示。

在 USB 规范中定义的惟一的设备设置请求便是设置远端唤醒，这与主机发出的 Get_Status(Device)设备请求中处于设备远端唤醒功能的设置处于同一位置。设备利用该位来反应其是否具有远端唤醒的功能；同时，主机使用同一位来表示对设备的远端唤醒功能的支持与否，如表 5-26 所示。

表 5-25　　设置设备的特征

字节序号	域	值	含义	固件的响应

0	bmRequestType	0x00	输出，设备	设置远端唤醒位
1	BRequest	0x03	“Set_Feature()”	
2	WvalueL	0x01	特征选择：	
3	WvalueH	0x00	远端唤醒	
4	WindexL	0x00		
5	WindexH	0x00		
6	WlengthL	0x00		
7	WlengthH	0x00		

表 5-26　　设置端点特征（STALL）

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x02	输出，端点	对于所示的端点设置暂停位
1	bRequest	0x03	“Set_Feature()”	
2	wValueL	0x00	特征选择：	
3	wValueH	0x00	暂停	
4	wIndexL	EP	0x00-0x08：OUT0-OUT8	
5	wIndexH	0x00	0x80-0x88：IN0-IN8	
6	wLengthL	0x00		
7	wLengthH	0x00		

在 USB 规范中定义的惟一的设置端点的特征请求便是暂停一个端点。对于所要设置的端点（由请求的第四个字节决定），固件通过设置 EPCS 寄存器中的 STALL 位来回应该请求。固件有两种响应方式，即将自己的端点置于暂停状态和对主机的请求作出积极的回应。端点的暂停可以通过主机的 Clear_Feature(Stall)请求清除。

固件对于 Set_Feature(Stall)请求的响应分为以下几步：

（1）对于所要设置的端点的 EPCS 寄存器中的 STALL 位进行设置。

（2）重新设置数据锁。

（3）将暂停的端点重新恢复在默认的状态下，准备在主机清除了暂停状态后进行数据的发送或是接收。例如对于端点 1 的输入状态，固件应该清除在 EP1CS 中的 BUSY 位；对于端点 1 的输出状态，固件应该在 EP1 字节计数器中装入相应的数值。

（4）清除 EP0CS 中的 HSNAK 位，结束 Set_Feature(Stall)的控制传输。

对于主机发出的设置接口请求，同样要执行前 3 步。

关于数据锁，FX 2 自动在 USB 的数据传输过程中根据端点的不同加入相应的数据锁，这样是为了保持数据的完整性。对于一些非常有限的环境，固件应该巧妙地使用这些数据位：Set_Feature(Stall)、Set_Configuration()、Set_Interface()。

3. 清除特征

清除特征被用来禁止远端唤醒或是使端点退出暂停状态，如表 5-27 和表 5-28 所示。

表 5-27　　清除设备特征（清除远端唤醒位）

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x00	输出，设备	清除远端唤醒位
1	bRequest	0x01	“Clear_Feature()”	
2	wValueL	0x01	特征选择：远端唤醒	
3	wValueH	0x00		
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	0x00		
7	wLengthH	0x00		

表 5-28　　清除端点特征（清除暂停）

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x02	输出，端点	对于所要求端点清除暂停位
1	bRequest	0x01	“Clear_Feature()”	
2	wValueL	0x00	特征选择：暂停	
3	wValueH	0x00		
4	wIndexL	EP	0x00-0x08：OUT0-OUT8	
5	wIndexH	0x00	0x80-0x88：IN0-IN8	
6	wLengthL	0x00		
7	wLengthH	0x00		

如果 USB 设备支持远端唤醒（在设备列举过程中其描述符的列表中会有报告），Clear_Feature(Remote)唤醒请求将会禁止该功能。

Clear_Feature(Stall)使端点退出暂停环境。固件通过清除所要求的端点的 EPCS 寄存器的 STALL 位来作出响应。

4. 获得描述符

在列举的过程中，主机通过获得描述符（Get_Descripter()）请求来获取设备的能力和需求信息。通过描述符表，设备送回必要的信息，例如，需要装载什么样的设备驱动，需要使用多少个端点，自身的特殊配置，可能使用到的可变设置等等。

FX 2 提供了一个特殊的设置数据指针，来使固件对主机发出的 Get_Descripter()请求的回应得到简化。固件将可能用到的描述符的起始地址装入其指针中，清除 HSNAK 位，然后固件传送整个描述符。

图 5-18 解释了对于设置数据指针的使用情况。指针通过两个寄存器来实现即 SUDPTRL 和 SUDPTRH。大多数获取描述符的请求所包括的数据不止一个数据包。描述符的数据包含了 91 个字节，如图 5-18 所示。

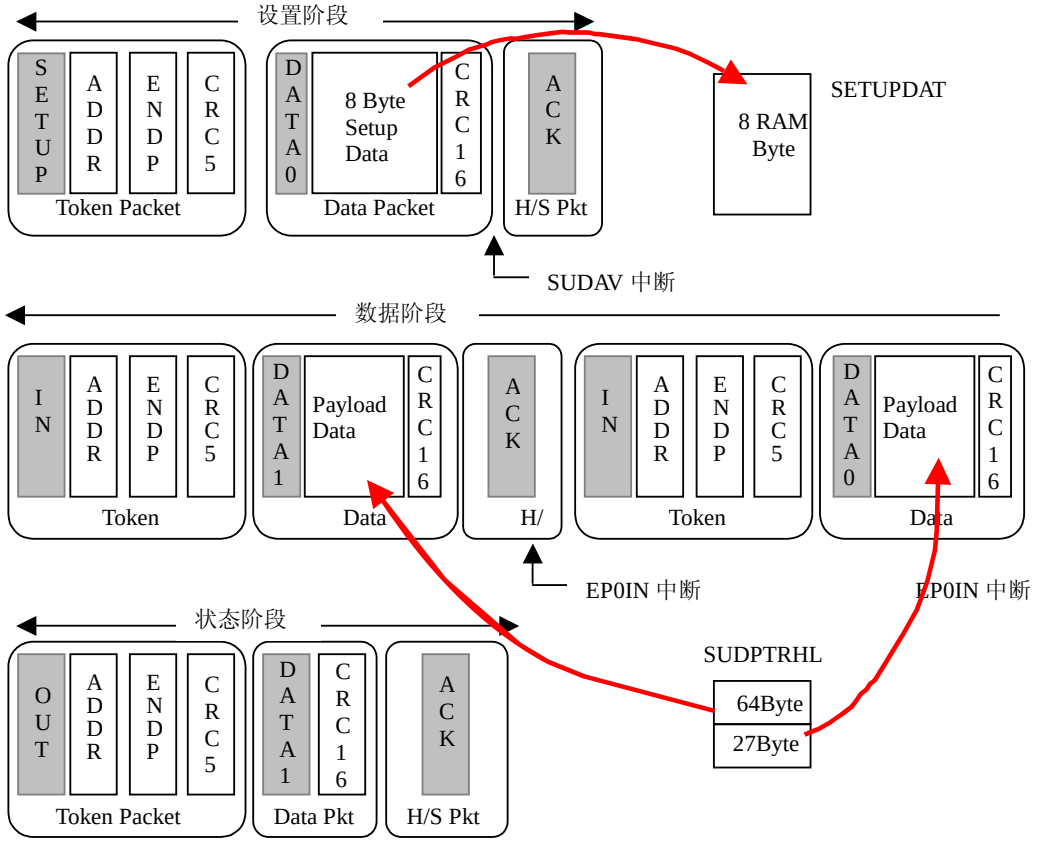


图 5-18　使用设置数据指针（SUDPTR）回应 Get_Descriptor 请求

控制传输的数据处理通过正常的方式开始。具体的过程如下：

❑ FX 2 自动传输 SETUP 包中的 8 字节到 SETUPDAT 的 RAM 中，然后发出 SUDAV 中断。

❑ 固件解码 Get Descriptor 请求，通过清除 HSNAK 位来响应。

❑ 在 SUDPTRL:H 寄存器中装入请求的描述符表地址。对 SUDPTRL 寄存器的装载会引起 FX 2 自动通过两个输入传输来响应，它们分别为 64 和 27 个字节，并且使用 SUDPTR 作为基地址。

❑ 用 ACK 信号来响应 STATUS 阶段。

通常的端点 0 的中断 SUDAV 和 EP0IN 在这个自动传输的过程中是始终保持有效的，所以，固件在通常情况下会屏蔽这些中断，因为正常的传输并不需要这些中断的干预。

描述符总共包括 3 种类型：设备、配置和字符串。

（1）获得设备的描述符

下面通过表 5-29 来看看主机如何获得设备的描述符。

正如图 5-16 所解释的，固件用描述符表的起始地址来装载两个字节的 SUDPTR。当 SUDPTRL 被装入数据时，FX 2 会自动执行以下操作：

❑ 从 SETUP 数据包中读取需要传输的字节数，这一信息位于包的第 6 和第 7 字节。

❑ 读取请求的描述符的长度域来确定实际的描述符的长度。

❑ 使用设置数据指针通过 EP0BUF 送出请求的字节数和描述符中实际的字节数两者中

的较小者。

表 5-29　　获得设备的描述符

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x80	输入，设备	通过对 SUDPTRH:L 的读取来设置 RAM 中的描述符表
1	bRequest	0x06	“Get_Descriptor()”	
2	wValueL	0x00		
3	wValueH	0x01	描述符类型：设备	
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	LenL		
7	wLengthH	LenH		

作为数据表的索引，它组成了控制传输 3 个部分中的第 2 个部分。必要时 FX 2 将会把数据包分割为几个部分进行传送。

- 进行错误检测，如果必要进行重传。
- 对控制传输的第 3 个阶段进行响应来结束整个操作。

任何 Get Descriptor 请求都可使用设置数据指针，该指针也可用于供应商的特殊请求。如果字节 6 和 7 传输字节数时，设置数据指针会像在获得描述符请求中一样自动工作。如果在这两个字节中并没有包含数据的长度，则字节数必须被明确地传输出来。

对于固件来说，有可能通过直接用不同的数据包装载 EP0BUF 来开始“人工的”控制传输，同时对有效的设置阶段进行跟踪。这是一个很好的 USB 练习例程，但并不是必须的，因为内建在 FX 2 中的硬件系统已经提供了对控制传输的支持。

对于少于 64Byte 的数据传输阶段，数据传入 EP0BUF 后将字节数装入 EP0BCH:L 寄存器的方法等同于使用设置数据指针的方法。然而，这样做却会浪费带宽，因为这需要将字节传入 EP0BUF，使用设置数据指针则不用。

（2）获得设备限制描述符

主机获得设备限制描述符的情况如表 5-30 所示。

表 5-30　　获得设备限制描述符

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x80	输入，设备	通过对 SUDPTRH:L 的设置来开始设置 RAM 中合适的设备限制描述符表
1	bRequest	0x06	“Get Descriptor”（确认？）	
2	wValueL	0xx00		
3	wValueH	0xx06	描述符类型：设备限制	
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	LenL		
7	wLengthH	LenH		

设备限制描述符只适用于设备工作在高速模式下的情况，这个描述报告设备工作于其他模式时情况会发生改变。例如，当前的设备工作于高速模式，设备限制描述符将会报告它如何在全速时进行工作。

设备限制描述符的使用类似于设备描述符，固件将合适的描述符地址装入 SUDPTRH:L，然后 FX 2 重新启动。

（3）获得配置描述符

获得配置描述符的使用情况如表 5-31 所示。

表 5-31　　获得配置描述符

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x80	输入，设备	通过对 SUDPTRH:L 的设置来开始设置 RAM 中的配置描述符表
1	BRequest	0x06	“Get_Descriptor()”	
2	WValueL	CFG	配置数字	
3	WValueH	0x02	描述符类型：配置	
4	WIndexL	0x00		
5	WIndexH	0x00		
6	WLengthL	LenL		
7	WLengthH	LenH		

（4）获得字符串描述符

获得字符串描述符的使用与设备描述符类似。如表 5-32 所示，固件读取设置数据的第 2 个字节来决定哪个配置或是字符串要被用到，然后将合适的描述符地址装入 SUDPTRH:L，FX 2 进行重启。

表 5-32　　获得字符串描述符

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x80	输入，设备	通过对 SUDPTRH:L 的设置来开始设置 RAM 中的字符串描述符表
1	bRequest	0x06	“Get_Descriptor()”	
2	wValueL	STR	字符串数目	
3	wValueH	0x03	描述符类型：字符串	
4	wIndexL	0x00	Language ID L	
5	wIndexH	0x00	Language ID H	
6	wLengthL	LenL		
7	wLengthH	LenH		

（5）获得其他速度配置描述符

所谓的获得其他速度配置描述符（Get_Descriptor(Other Speed Configuration)）只用在那些具有高速功能的设备中，它描述了设备如果工作于其他速度时的配置情况。例如，如果设备当前工作于高速模式，其他速度描述符将会返回在全速下的配置信息。

其使用方式的描述如表 5-33 所示。

表 5-33　　获得其他速度配置描述符

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x80	输入，设备	通过对 SUDPTRH:L 的设置来开始设置RAM中的其他速度配置描述符表
1	bRequest	0x06	“Get_Descriptor()”	
2	wValueL	CFG	其他速度配置序号	
3	wValueH	0x07	描述符类型：其他速度配置	
4	wIndexL	0x00	Language ID L	
5	wIndexH	0x00	Language ID H	
6	wLengthL	LenL		
7	wLengthH	LenH		

其他速度配置描述符的工作情况与配置描述符类似，固件将合适的描述符地址装入 SUDPTRH:L，然后 FX 2 重新启动。

5. 设置描述符

对设备本身、设备配置以及设备字符串的设置情况如表 5-34、表 5-35、表 5-36 所示。

表 5-34　　设置设备描述符

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x00	输出，设备	通过 EP0BUF 读取设备描述符数据
1	bRequest	0x07	“Set_Descriptor()”	
2	wValueL	0x00		
3	wValueH	0x01	描述符类型：设备	
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	LenL		
7	wLengthH	LenH		

表 5-35　　设置配置描述符

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x00	输出，设备	通过 EP0BUF 读取配置描述符数据
1	bRequest	0x07	“Set_Descriptor()”	
2	wValueL	0x00		
3	wValueH	0x02	描述符类型：配置	
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	LenL		
7	wLengthH	LenH		

表 5-36　　设置字符串描述符

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x00	输入，设备	通过 EP0BUF 读取字符串描述符数据
1	bRequest	0x07	“Get_Descriptor()”	
2	wValueL	0x00	字符串序号	
3	wValueH	0x03	描述符类型：字符串	
4	wIndexL	0x00	Language ID L	
5	wIndexH	0x00	Language ID H	
6	wLengthL	LenL		
7	wLengthH	LenH		

固件通过清除 HSNAK 位响应设置字符串描述符，就可直接从 EP0BUF 中读取描述符数据。FX 2 保持对从主机传送到 EP0BUF 的字节数的跟踪，然后将得到的数据与描述符中的 6、7 字节进行比较。如果传输的数目相同，FX 2 自动响应 STATUS 阶段，即控制传输的最后一个阶段。

在控制传输的数据阶段，固件控制着数据的流动。固件每一次处理完一个输出数据包后，将会在字节计数器中写入值以重新设置端点。

（1）关于配置、接口和可变设置

一个 USB 设备可以有一个或者多个配置状态。在任何时候只能有一个配置处于激活状态。

一个配置可以有一个或者多个接口，在所处的配置被激活时所有的接口也都处于激活状态。多重接口允许不同的主机方面的设备驱动与不同的 USB 设备的部分相关联。

一个接口可以有一个或是多个可变的设置，每一个设置拥有一个或是多个端点组成的集合，如图 5-19 所示。

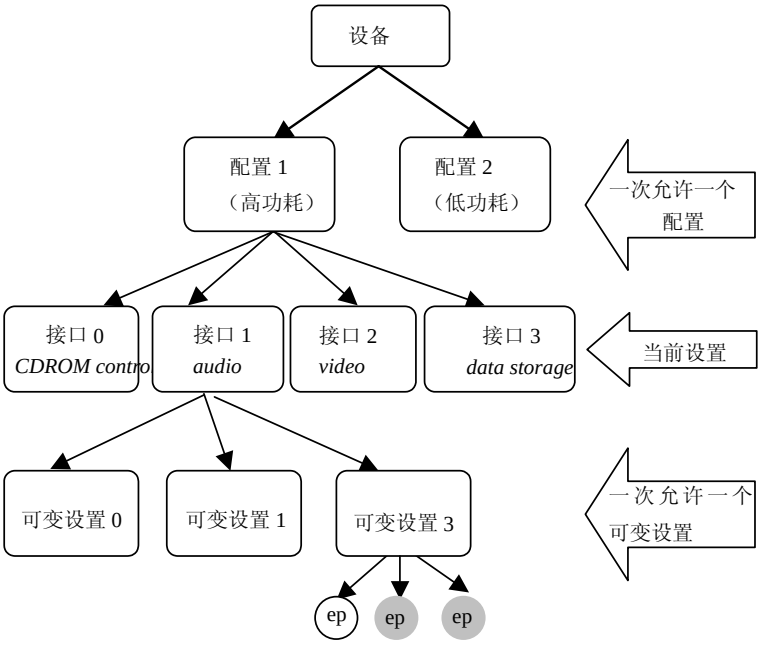


图 5-19　配置、接口和可变设置的关系

这一结构是一软件的模式，FX 2 在这些设置改变时并不采取任何动作。然而，在主机改变了配置或是接口设置发生了变化后，固件必须重新初始化端点。

至于与固件有关的配置，只是一个反映当前设置的不同的字节。

主机发出 Set_Configuration()请求来选择一个设置，然后通过 Get_Configuration()请求来决定一个设置。

（2）设定配置信息

主机设定配置信息时用到的描述符如表 5-37 所示。

表 5-37　　决定设置信息

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x00	输出，设备	在固件中读写 CFG 来改变设置
1	BRequest	0x09	“Set_Configuration()”	
2	WValueL	CFG	设置序号	
3	wValueH	0x00		
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	0x00		
7	wLengthH	0x00		

当主机发出设定配置（Set_Configuration()）请求时，固件保存设置序号（位于第 2 个字节：CFG），然后通过执行必要的内部操作来对该设置进行支持，最后清除 HSNAK 位来结束控制传输的设定配置阶段。

经过配置的设定后，主机发出 Set_Interface()命令来设置配置中包含的不同接口。

6. 获得配置信息

主机在获得配置信息时所用到的描述符如表 5-38 所示。

表 5-38　　获得配置信息

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x80	输入，设备	在重新配置后通过 EP0 发送 CFG
1	bRequest	0x08	“Get_Configuration()”	
2	wValueL	0x00		
3	wValueH	0x00		
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	1	LenL	
7	wLengthH	0	LenH	

当主机发出获得配置（Get_Configuration()）请求时，固件返回当前的配置序号。固件将配置序号装入 EP0BUF，装载一个字节的计数值到 EP0BCH:L，最后清除 HSNAK 位来结束控制传输。

7. 设置接口

设置接口 Set_Interface()这个令人迷惑的 USB 命令所表达的意思实际上是要处理一个特殊的有多种设定状态的接口，通过表 5-39 所示的内容来改变其设定状态。

描述符的情况如表 5-39 所示。

USB 设备可以有多重的协作接口。举例来说，一个音频设备可能有 0、1、2 多重设置用以支持 8kHz、22kHz、44kHz 的采样率。接口的面板可能有 0、1 设置用于区别中文和英文用户。Set/Get Interface 请求就是要在这些不同的设置中选择合适的配置。

表 5-39　　设置接口（实际上是选择接口可变状态的一种）

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x00	输出，设备	对于选定的接口读取或是存储由第 2 个字节确定的设置，在固件中改变接口设置
1	bRequest	0x0B	“Set_Interface()”	
2	wValueL	AS	可变的设置序号	
3	wValueH	0x00		
4	wIndexL	IF	接口序号	
5	wIndexH	0x00		
6	wLengthL	0x00		
7	wLengthH	0x00		

固件通过以下步骤来响应设置接口（Set_Interface()）请求：

（1）执行请求所需的内部操作（例如确定采样率）。

（2）重新设置接口中的每一个端点的数据锁。

（3）使端点恢复到其默认状态，准备发送或是接收数据。例如，对于端点 1 的输入，固件应该清除位于 EP1CS 寄存器中的 BUSY 位；对于端点 1 的输出，固件应该将合适的值装入 EP1 的字节计数器中。

（4）清除 HSNAK 位来结束控制传输的设置接口阶段。

8. 获得接口状态

主机在获得接口状态时用到的描述符如表 5-40 所示。

当主机发出获取接口状态（Get_Interface()）请求时，固件将会简单地返回响应的接口的设置状态，然后清除 HSNAK 位。

表 5-40　　获得接口（实际上，是确定可变接口状态中的一种）

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x81	输入，设备	通过 EP0 对相应的接口送出设置信息
1	bRequest	0x0A	“Get_Interface()”	
2	wValueL	0x00		
3	wValueH	0x00		
4	wIndexL	IF	接口序号	
5	wIndexH	0x00		

续表

字节序号	域	值	含义	固件的响应
6	wLengthL	1	LenL	
7	wLengthH	0	LenH	

9. 设置地址

当一个 USB 设备被第一次连入时，会通过地址 0 来进行响应，直到主机为其分配一个惟一的地址为止，这时，主机要用到设置地址 Set_Address()请求。FX 2 将这个设备的地址拷贝到 FNADDR（Function Address）寄存器中，然后通过这个地址来响应所有的请求。直到设备从主机拔离这一地址、主机发出 USB Reset 请求或者切断电源时才会影响到这一地址。

FNADDR 寄存器是一只读寄存器。只要 FX 2 开始重新列举，它便会自动将 FADDR 置为 0，允许设备作为新设备连入。

FX 2 的程序并不需要设备的地址，因为 FX 2 只对主机签署的地址作出响应。只有在调试、诊断时设备地址才可以读取。

10. Sync 帧

Sync 帧的结构如表 5-41 所示。

表 5-41　　Sync 帧

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x82	输入，端点	通过 EP0 送出帧序号来与所选端点取得同步
1	bRequest	0x0C	“Sync_Frame()”	
2	wValueL	0x00		
3	wValueH	0x00		
4	wIndexL	EP	端点序号	
5	wIndexH	0x00		
6	wLengthL	2	LenL	
7	wLengthH	0	LenH	

Sync 帧请求用于建立一个实时的标志，这样，主机和 USB 设备就能在通过同步端点发送多重帧时取得同步的效果。

假设一个同步传输由 300 个数据包组成，要通过端点 8 进行从主机到设备的发送。主机和设备都包含序列计数器，它重复地从 1 计数到 5，来保持对一个传输中的数据包的跟踪。为了取得同步的开始，主机和设备都要同时将其计数器置为 0。

为了实现同步，主机通过 EP=EP8OUT（0x80）来发出 Sync 帧。在其后的某一时间，固件用 EP0BUF 装载两个字节的计数器作为响应。例如，在当前帧数加上 20，表示将当前帧之后的第 20 个帧作为同步帧。在这一过程中，双方都要将其序列计数器置为 0。在 USBFRAMEH 和 USBFRAMEL 寄存器中可以访问到当前帧计数值。

多重同步端点可以通过这种方式进行同步，固件会给每一个端点分配一个独立的内部序列计数器。

关于 USB 的帧的说明：在全速模式下，USB 主机每隔一个毫秒发送一个 SOF 数据包（Start Of Frame），每一个 SOF 数据包包含 11 位帧序号。固件通过 SOF 中断和向导，在 SOF 时间服务于所有的同步传输。如果 FX 2 检测到一个 SOF 包的丢失或是非法，它可以使用一个内部的计数器来自动产生 SOF 中断。

在高速模式下，每一个帧被分为 125 个微帧。但是，帧计数器仍然是每隔一个帧计数一次，主机每隔一个微帧发出一个 SOF。主机和设备总是在第零帧取得同步，该帧是通过设备对 Sync 帧请求的响应来确定的，对于微帧没有再另行规定同步机制。

11. 装载固件

USB 的“端点 0 协议”提供了一个将供应商特别请求与标准请求混合的机制。位 6、5 为 00 时用于标准请求，为 10 时用于供应商特别请求。

供应商特别请求包括两种，分别为下载固件和上传固件，如表 5-42 和表 5-43 所示。

表 5-42　　下载固件

字节序号	域	值	含义	固件的响应
0	bmRequestType	0x40	供应商请求，输出	无需求
1	bRequest	0xA0	“Firmware_Load()”	
2	wValueL	AddrL	起始地址	
3	wValueH	AddrH		
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	LenL	字节数	
7	wLengthH	LenH		

表 5-43　　上传固件

字节序号	域	值	含义	固件的响应
0	bmRequestType	0xC0	供应商请求，输入	无需求
1	bRequest	0xA0	“Firmware_Load()”	
2	wValueL	AddrL	起始地址	
3	wValueH	AddrH		
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	LenL	字节数	
7	wLengthH	LenH		

FX 2 响应两个端点 0 的供应商特别请求，RAM 的上传、下载，这两个请求无论是 RENUM=1 还是 RENUM=0 时都有效。

由于 SETUP 数据包中的第一个字节的第一位表明了传输的方向，因此，在上传、下载请求中只对一个字节（bRequest）有要求。这些命令对于使用 FX 2 的 USB 设备来说都是可用的。

主机装载程序，首先要在 CPUCS 寄存器中写入 0x01，以使 FX 2 的 CPU 重新启动，然

后将程序代码装入 FX 2 的内部 RAM，接着在 CPUCS 寄存器中写入一个“0”，使得 CPU 脱离复位状态。

5.3　本章小结

这 1 章对 USB 设备实现即插即用的第 3 个基础——控制传输作了比较详细的介绍。

控制传输的目的是要使主机给设备进行合适的配置，从而使它们之间能够按照设计者预定的目的来进行工作。在这一期间，要用到一系列的描述符，这初看起来有些繁杂，但这一部分内容又是设计者不得不掌握的，所以希望读者能够认真阅读。

下面一章将对 USB 设备的信号传输作一个简要介绍。

第 6 章　数据传输方式

知识点：

- 控制传输
- 批量传输
- 等时传输
- 中断传输

本章导读：

本章从 USB 通信流的角度来讲述 USB 的 4 种数据传输方式：控制传输、批量传输、等时传输和中断传输。

USB 通过通道在主机缓冲区与设备端点间传送数据。在消息通道中传送的数据具有 USB 定义的格式，但在 USB 定义的消息数据有效载荷区中也可以传输具有设备指定格式的数据。USB 要求在任何通道上传送的数据均被打包，最终，在一个总线传输事务中的有效数据载荷的格式化和解释工作由客户端软件和使用通道的设备功能来完成。USB 提供了不同的数据传输类型，使之尽可能满足客户软件和设备功能的服务要求。一个 IRP 需要由一个或多个总线处理事务来完成。

每种传输类型在以下的几个通信流特征上会有所不同：

- USB 规定的数据格式。
- 信息流的方向。
- 包大小的限制。
- 总线访问的限制。
- 延时的限制。
- 数据顺序要求。
- 出错处理。

USB 设备的设计者可以决定设备上每个端点的能力。一旦为这个端点建立了一个通道，这个通道的绝大多数传输特性也就固定下来了，一直到这个通道被取消为止。也有部分传输特性可以改变，对这样的特性，将会在介绍每个传送类型时作出相应的说明。

USB 定义了 4 种传输类型：

- 控制传输：突发式、非周期性的，由主机软件发起的请求或响应的通信，通常用于命令事务和状态事务。
- 等时传输：在主机与设备之间周期性的、连续的通信，一般用于传输与时间相关的信息。这种类型保留了将时间概念包含于数据中的能力。但这并不意味着，传送这样数据的时间总是很重要的，即传送并不一定很紧急。

- 中断传输：低频率、固定延迟的通信。
- 批量传输：非周期性的、大量的突发性传送。典型地用于传送那些可以利用任何带宽的数据，而且当没有可用带宽时，可以延时传输。

每种传输类型都有不同的用途，以及前面所提及的不同的通信流特性。本章将从 USB 通信流的角度来介绍 USB 的 4 种数据传输方式：控制传输、批量传输、等时传输和中断传输。

6.1 控制传输

控制传输用于支持在客户软件和设备功能之间的关于配置、命令、状态类型的通信流。在设备列举过程中，主机通过控制传输向设备功能发出标准 USB 请求来获取设备和配置等描述符，从而了解设备信息，建立起客户软件和设备功能之间的正常通信。

每一个 USB 设备都必须实现缺省控制通道，并将它实现成一个消息通道。这个通道由 USB 系统软件使用，USB 设备的确认信息、状态信息以及控制信息由该通道传输。如果需要，一个设备功能可以为端点实现额外的控制通道，但这并不是必须的。

6.1.1 数据格式

控制传输最少有两个事务阶段：设置阶段和状态阶段。控制传输可以有选择性地在设置阶段和状态阶段之间插入数据阶段。在设置阶段，主机通过发送请求信息来启动一个设置事务，用于向一个设备功能的控制端口传输信息。设置事务在格式上类似于 OUT 事务，但是使用的 PID 是 SETUP 而不是 OUT。设置事务的格式如图 6-1 所示。

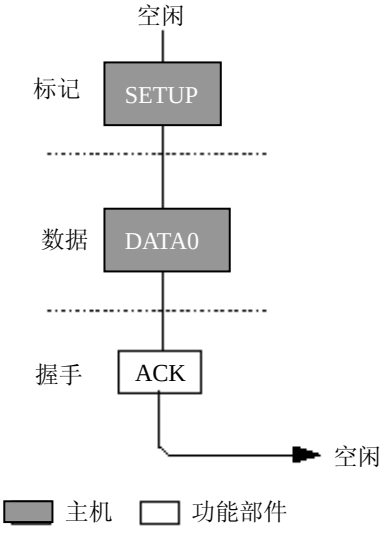


图 6-1 控制传输的设置事务

设置事务的数据字段包含了请求信息。USB 规范定义了 11 种标准请求（第 5 章 5.1.5 小节已经详细介绍了这 11 种标准请求），尽管设备不需要 0 支持所有的请求，但是，一个特定的 USB 类设备必须能够响应为这个类定义的请求，而且任何设备都可以支持供应商和设备特定的请求。设置阶段的数据字段总是使用 DATA0 PID。设备功能收到一个设置包必须接受设置数据并用 ACK 应答，如果数据被损坏，则丢弃数据且不返回握手信号。

如果有控制传输的数据阶段，则由一个或多个 IN 或 OUT 事务构成，并且遵守与批量传输相同的协议规则。所有数据阶段中的事务都必须具有相同的方向（即全部 IN 或者全部 OUT），在数据阶段中要发送的数据量和其方向在建立阶段中被指定。如果数据的数量超过了先前确定的数据包大小，则数据先按照所支持的最大数据包在多个事务中被传送（IN 或 OUT），剩下的数据小于最大数据包大小时，所有的数据在最后一个事务中被一次传送。

控制传输的状态阶段是最后一个事务，状态阶段的事务也遵循与批量事务相同的协议次序。高速设备的状态阶段也包括 PING 协议。一个状态阶段是以相对前面的阶段的数据流方向的变化来划分定界的，并且总是使用 DATA1 PID。例如，如果数据阶段是由 OUT 事务构成的，那么，状态阶段是单一的 IN 事务。如果控制传输没有数据阶段，那么它由设置阶段和包含 IN 事务的状态阶段构成。

低速和全速控制写传输的各个阶段如图 6-2、图 6-3 和图 6-4 所示。低速和全速控制读传输的各个阶段如图 6-5、图 6-6 和图 6-7 所示。

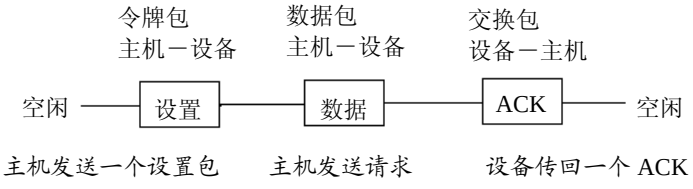


图 6-2　设置阶段的控制写传输

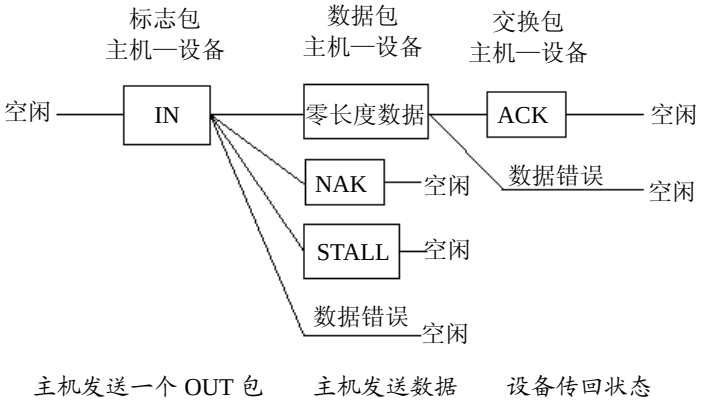


图 6-3　数据阶段的控制写传输可以有 0 个或多个数据事务

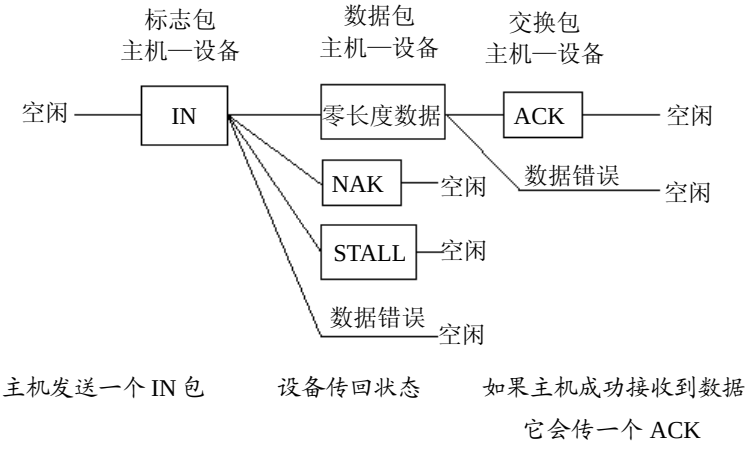


图 6-4　状态阶段的控制写传输

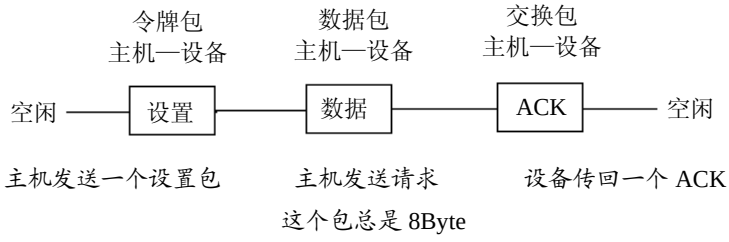


图 6-5　设置阶段的控制读传输

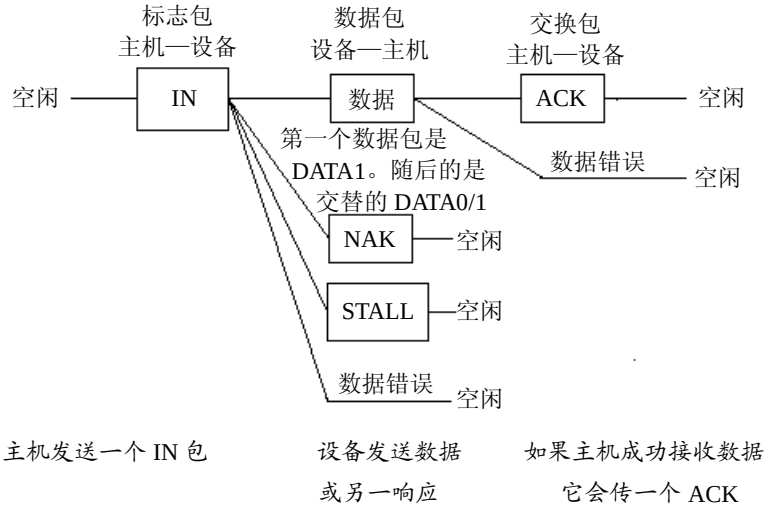


图 6-6　数据阶段的控制读传输可以有 1 个或多个数据事务

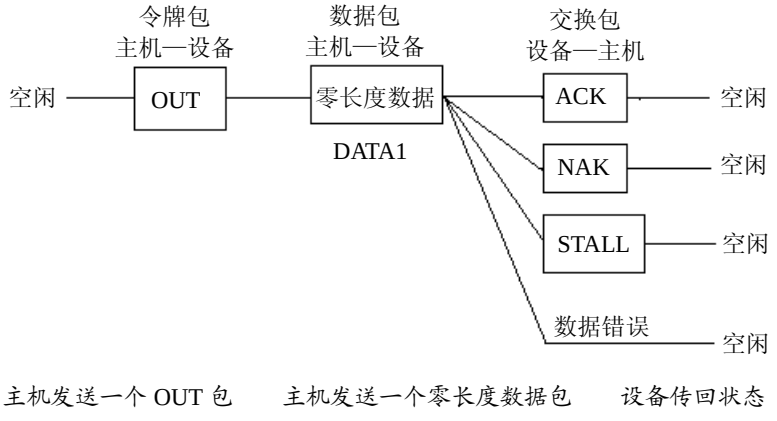


图 6-7　状态阶段的控制读传输

6.1.2　包大小的限制

一个控制传输端点需要被指定能够从总线上发送或接收的最大数据有效载荷的大小，也

就是一次传输最大包的大小。在全速设备中，允许数据包的最大值为 8、16、32 或 64 个字节。在高速设备，允许的最大值为 64 个字节；而低速设备只能支持 8 个字节的最大包。这个最大值应用于紧随一个设置包之后的数据包的数据字段，也就是说这个值只包括了包的数据字段，而不包括协议要求的其他信息，如 PID 和 CRC 校验位。一个端点在自己的配置信息中报告自己允许的最大数据有效载荷。USB 并不要求数据有效载荷必须达到最大长度，当数据长度不够时，不必填充到最大长度。

为了确认缺省控制通道的最大包的大小，USB 系统软件读取设备描述符。主机至少会读取设备描述符的前 8 个字节以确保已经读到这个缺省通道中的 wMaxPacketSize 字段（设备描述符的第 7 个字节）。对于其他控制端点，USB 系统软件将会在配置之后获取他们的最大数据有效载荷的大小，从而确保端点所收发的数据不会超长。

端点所传送的数据有效载荷区长度必须小于或等于端点的 wMaxPacketSize 值，当一个控制传输所包含的数据大于一个最大包的长度时，该数据就会被分为多个具有最大数据载荷的包来传送，直到剩下的数据小于或等于最大长度为止，这时，所有剩下的数据都会在最后一个传输事务中被一次性传送。

6.1.3　总线访问限制

高速、全速和低速 USB 设备都可以使用控制传输。一个设备端点无法为控制通道指定所需的总线访问频率，而由主机控制器平衡所有控制通道和等待的特定 IRP 请求的总线访问需求，从而保证客户软件和设备功能之间数据的最大努力交付（“Best Effort”Delivery）。USB 要求为控制传输保留部分（微）帧时，将包括以下几点：

- 对于全速和低速端点，如果控制传输只消耗不到 10%的帧时，或对于高速端点消耗少于 20%的微帧时，那么剩下的时间可以用于支持批量传输。
- 一个需要重传的控制传输可以在当前（微）帧或以后的（微）帧中重传，而不一定在相同的帧中。
- 如果保留给控制传输的时间不够用，但恰好有一些额外的等时和中断传输的（微）帧未用时，则主机控制器利用这些时间进行额外的控制传送。
- 如果有过多的控制传输在等待可用的（微）帧时，那么先对它们进行排序，然后再传送。
- 如果各个控制传输申请的是不同的端点，主机控制器将根据公平访问原则决定它们的访问顺序。公平访问原则的具体内容决定于主机控制器的实现。
- 一个频繁重传的控制传输事务也不能不公平地占用总线访问时间。

这些要求使得主机和 USB 设备之间的控制传输可以以最大努力交付为原则，在总线上有规律地进行传输。

USB 系统软件会根据自己的判断来改变一个特定端点的控制传输速率，一个端点和它相应的客户软件可占用的总线时间会因为其他设备的插入/退出系统，或这些设备端点请求控制传输而改变。

总线频率和（微）帧时序限定了任何 USB 系统的（微）帧内能够成功完成控制传输的最大数量。对于全速和低速总线，每一帧中成功完成控制传输的数量应小于 29 个全速 8 个字节

数据的有效载荷，或小于 4 个低速 8 个字节数据的有效载荷。对于高速总线，控制传输的数量被限制为每个微帧传输 32 个高速的 64 个字节数据有效载荷。不同大小的低速包和（微）帧中可能的最大包数的信息如表 6-1 所示。

表 6-1　　低速控制传输限制

	协议包头（共 63 个字节，包括 15 个 SYNC 字节，15 个 PID 字节，6 个端点+CRC 字节，6 个 CRC 字节，8 个设置数据字节，1 个 13 字节的数据包间延迟（EOP 等））					
	数据有效载荷	最大带宽 B /Frame	每个传输的帧的带宽	最大传输数	剩余字节	有用数据 B/Frame
	1	3000	26%	3	40	3
	2	6000	27%	3	37	6
	4	12000	28%	3	31	12
	8	24000	30%	3	19	24
最大		187500				187

不同大小的全速控制传输和一帧中可能的最大传输数量的信息如表 6-2 所示。这个表是假定控制传输只有一个数据阶段，并且这个数据阶段中含有零长度的状态字段。

表 6-2　　全速控制传输的限制

	协议包头（共 45 个字节，包括 9 个 SYNC 字节，9 个 PID 字节，6 个端点+CRC 字节，6 个 CRC 字节，8 个设置数据字节，1 个 7 字节的数据包间延迟（EOP 等））					
	数据有效载荷	最大带宽 B /Frame	每个传输的帧的带宽	最大传输数	剩余字节	有用数据 B /Frame
	1	32000	3%	32	23	32
	2	62000	3%	31	43	62
	4	120000	3%	30	30	120
	8	224000	4%	28	16	224
	16	384000	4%	24	36	384
	32	608000	5%	19	37	608
	64	832000	7%	13	83	832
最大		1500000				1500

不同大小的高速控制传输和一微帧中可能的最大传输数量的信息如表 6-3 所示。这个表也是假定控制传输只有一个数据阶段，并且这个数据阶段有零长度的状态字段。

表 6-3　　高速控制传输限制

	协议表头（共 173 个字节，以 480Mbit/s 以及 8bit 信息包间间隔为基础，包括 88bit 的最小总线转换，32 bit SYNC，8 bit EOP，9×4 个 SYNC 字节，9 个 PID 字节，6 个 EP/ADDR+CRC，6 个 CRC16，8 个设置数据，9×(1+11)个字节的数据包间延迟（EOP 等））					
	数据有效载荷	最大带宽 B /Frame	每个传输的帧的带宽	最大传输数	剩余字节	有用数据 B /Frame
	1	344000	2%	43	18	43

	2	672000	2%	42	150	84
	4	1344000	2%	42	66	168

续表

	数据有效载荷	最大带宽 B /Frame	每个传输的帧的带宽	最大传输数	剩余字节	有用数据 B /Frame
	8	2624000	2%	41	79	328
	16	4992000	3%	39	129	624
	32	9216000	3%	36	120	1152
	64	15872000	3%	31	153	1984
最大		60000000				7500

6.1.4　控制传输的数据顺序和错误的检测处理

要进行控制传输，先要由主机向设备发一个总线设置（SETUP）信息。它描述了控制访问的类型，设备将执行此控制访问。在设置阶段之后，是零个或多个控制数据信息的传送，这是进行访问的具体信息。最后，由状态事务的传送来结束这次控制传输，并允许端点将这次控制传输的状态回送给客户软件。状态事务完成后，该端点可以进行下一次控制传输，每次控制传输何时在总线上进行由主机控制器的具体实现决定。在控制传输的数据和状态阶段，设备可能处于“忙”状态。此时，端点可以设法指明自己正忙，主机将试着在稍后的时间重传一次。

如果在上一个控制传输结束之前，端点又收到一个总线设置事务，那么，设备将结束现在未完成的传送，转而处理新的控制传输。正常情况下，设置事务是不会在上一个控制传输完成之前发出的。如果上一个控制传输因错误而被中止后，主机可发送下一个控制传输的总线设置事务。在端点看来，这是在上一个控制传输结束之前发出的。

一旦主机遇到一个引起中止的条件或检测到一个错误，端点可以通过接收下一个 SETUP 包的 PID 来恢复，也就是说，主机将会重试三次。如果端点还是收不到 SETUP 的 PID 时，主机最终会要求设备复位来清除中止条件或错误条件。

在控制传输中，USB 提供了强大的错误检测和恢复重传机制。发送器和接收器在控制传输中可以保持同步，这样就可以以最小的代价恢复传输。接收器可以识别一个数据重传包或状态信息重传包，因为包中带有数据重传的指示。一个发送器可以通过对方给它发送的握手信息来确知它发的数据重传包和状态信息包已被成功接收，除了 SETUP 包以外，协议可以将一个重新传送的包与原来的包区分开来。SETUP 包可以因为出错而重传，但无法说明此包是重传的，还是原来的。

6.2　批量传输

批量传输可以在不确定的时间内，传送相对大量的数据而不会使总线阻塞，因为它会让

其他传输类型首先执行，等到有可以利用的总线带宽时再进行传输，也就是说批量传输只发生在有可用总线带宽时。对于一个有大量数据和空闲带宽的 USB 设备，批量传输的速度相当快。然而在只有很少可用带宽的情况下，批量传输却会等待较长的一段时间。批量传输可用于从主机向打印机发送数据、从扫描仪向主机发送数据以及磁盘的读写等。

6.2.1 数据格式

一个批量传输包含有一个或多个 IN 或 OUT 事务。批量管道是流式管道，因此，对于一个给定的批量管道，它的传输总是单方向的。所有的事务要么全是 IN 事务，要么全是 OUT 事务。如果有设备要求双向的批量通信流，那么必须有两个批量管道，分别进行不同方向的批量传输。这两个方向的事务如图 6-8 和图 6-9 所示。

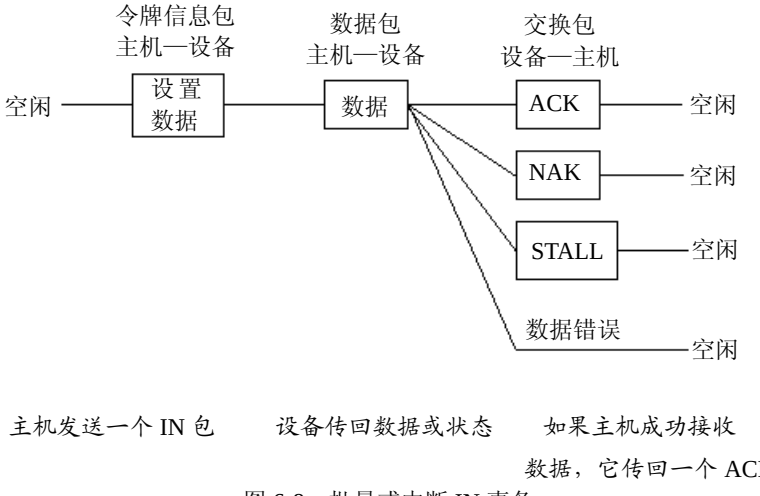


图 6-8　批量或中断 IN 事务

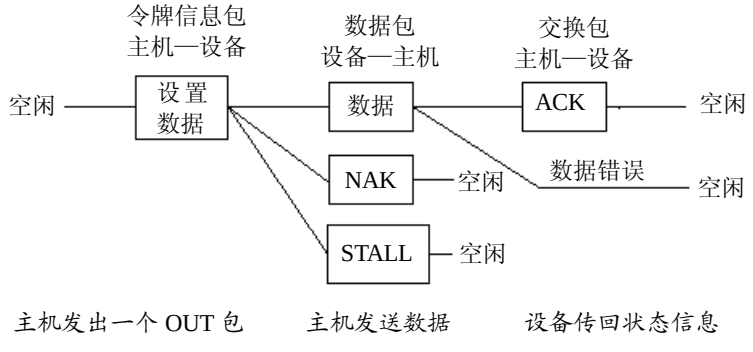


图 6-9　批量或中断 OUT 事务

当所要求的数据量已经传输完，或者当数据包的大小小于所规定的最大值时，批量传输就算结束了。

在高速设备中，主机会使用 PING 协议来节约总线带宽。如果高速批量传输传送多于一个的数据包，并且设备在收到其中一个之后传回 NYET，主机就会使用 PING 来查看是否可以开始下一个数据传输事务。

6.2.2 数据包大小的限制

一个全速的批量传输端点，允许的最大数据有效载荷可以是 8、16、32 和 64 个字节。高速的批量传输端点，最大的数据有效载荷必须是 512 个字节。低速设备不支持批量传输。在配置过程中，SUB 系统软件读取每一个批量传输端点的最大数据包的大小，并且确保所传输的所有数据载荷都不大于这个值。一个传输的数据量可以小于、等于或大于最大数据包的大小。如果一个批量传输包括的数据量无法放在一个数据包中，那么主机会使用多个事务来进行传输，并且除了最后一个事务外，所有事务的数据包都必须按所支持的最大数据包的大小来传输。

6.2.3 总线访问限制

主机控制器并不像对控制传输那样为批量传输保留任何的总线带宽。控制传输比批量传输有更高的优先级。在低速或全速模式时，系统会为控制传输保留 10%的总线带宽；在高速时，会保留 20%的带宽。然而，批量传输只有在有空闲总线带宽时才能进行传输。当总线忙时，批量传输可能会等待很长的时间。当总线闲置时，批量传输可以利用任何形式的大部分带宽，而且批量传输的包头很小，所以是最快的传输方式。如果多个端点同时请求批量传输，主机控制器将会根据公平访问原则，安排它们的顺序。

在闲置的全速总线上，每帧可以传输 19 个 64 字节的批量传输，也就是每帧 1216 个字节，数据传输速率是 1.216Mbit/s。这样还会剩下 18%的总线带宽，可以用于其他的传输类型。在全速时一个批量传输数据包的协议包头是 13 个字节，而在高速时是 55 个字节。

在闲置的高速总线上，每帧可以传输 13 个 512 字节的批量传输，也就是每帧 6656 个字节，数据传输速率是 53.248Mbit/s。这样会有 2%的总线带宽，可以用于其他的传输类型。全速和高速批量传输的总线访问限制如表 6-4 和表 6-5 所示。

表 6-4　　全速批量事务限制

	协议表头（共 13 个字节，包括 3 个 SYNC 字节，3 个 PID 字节，2 个端点+CRC 字节，1 个 3 字节的中间包延时（EOP 等））					
	数据有效载荷	最大带宽 B/Frame	每个传输的帧的带宽	最大传输数	剩余字节	有用数据 B/Frame
	1	107000	1%	107	2	107
	2	200000	1%	100	0	200
	4	352000	1%	88	4	352
	8	568000	1%	71	9	568
	16	816000	2%	51	21	816
	32	1056000	3%	33	15	1056
	64	1216000	5%	19	37	1216
最大		1500000				1500

表 6-5　　高速批量事务限制

	协议表头（共 55 个字节，包括 3×4 个 SYNC 字节，3 个 PID 字节，2 个 EP/ADDR+CRC，2 个 CRC16 字节，1 个 3×(1+11)字节的数据包间延时（EOP 等））					
	数据有效载荷	最大带宽 B/Frame	每个传输的帧的带宽	最大传输数	剩余字节	有用数据 B/Frame
	1	1064000	1%	133	52	133
	2	2096000	1%	131	33	262
	4	4064000	1%	127	7	508
	8	7616000	1%	119	3	952
	16	13440000	1%	105	45	1680
	32	22016000	1%	86	18	2752
	64	32256000	2%	63	3	4032
	128	40960000	2%	40	180	5120
	256	49152000	4%	24	36	6144
	512	53248000	8%	13	129	6656
最大		60000000				7500

6.2.4　数据顺序和错误检测

当端点由控制传输正确配置后，批量事务最初都是使用 DATA0，主机发起的第一个批量传输也同样使用 DATA0。

批量传输也可以检测错误，如果一个设备没有返回预期的交换包，主机控制器会重试多次。如果收到一个 NAK 交换信息，则主机会无限制地重试。批量传输也会使用数据交换位来保证收发之间的同步，从而确保可靠的传输。

6.3　中断传输

中断传输适用于那些请求传输的频率不高，但是必须在指定时间内完成传输的数据。一般的应用包括 USB 的键盘、鼠标、游戏杆和集线器的状态报告。中断传输需要快速地向主机报告当前的状态，这是由设备的属性和使用的场合所决定的。因为用户不希望在按键或移动鼠标时，在屏幕上看到有明显的动作延迟。低速设备只支持控制传输和中断传输，因此，在低速设备中，有可能将中断传输用于一般的数据传输，而不是键盘或鼠标等标准设备。Windows 98 就嵌入了 HID（Human Interface Device，人机接口设备）类的驱动程序，应用程序可以与符合 HID 规范的设备通过中断传输进行通信。

中断传输的名字可能会给人以误解，认为设备的动作会主动触发一个中断。其实，中断

传输也和传输类型一样，只能发生在主机轮询设备的时候，并不是由设备触发的硬件中断。

低速、全速和高速设备都支持中断传输。设备并不一定都需要支持中断传输，但是一些设备类可能需要中断传输，例如，HID 类的设备。

6.3.1　数据格式

一个中断传输包含有一个或多个 IN 事务，或者一个或多个 OUT 事务。中断传输的结构和批量传输相同。中断传输也是单向的：所有的事务都必须是 IN 事务，或者都是 OUT 事务。如果设备要求双向地中断传输，那么必须有两个中断管道，分别进行不同方向的中断传输。

当所有的数据都传输完，或者当数据包的长度小于所规定的最大值时，批量传输就算结束了。

在高速数据总线上与低速或全速设备进行中断传输时，主机使用分割事务（SPLIT）来传输所有的事务。与高速批量传输不同的是，高速中断传输在传输多于一个事务时不使用 PING 协议。

6.3.2　包大小限制

全速设备所允许的最大数据有效载荷的大小是 1~64 个字节。高速设备所允许的最大数据有效载荷的大小是 1~1024 个字节，而低速设备被限制在 8 个字节以下。一个高速高带宽端点可以特别指定在一个微帧中传输 2 个，还是 3 个事务。在设备配置期间，系统软件获取一个中断通道的最大数据有效载荷的大小，这个值将一直有效，直到设备被重新配置。系统软件使用这个值来确定在分配的时间内是否有充足的总线时间来完成这个最大的数据包的传输。如果有充足的总线时间，那么这个中断管道就被建立；如果没有，系统将不会建立这个中断管道。然而，实际数据包的大小还要由数据发送器来决定。如果数据无法在一个事务里传输完毕，主机控制器就会把这个传输分成多个事务。最后一个事务的数据包的大小可能小于或等于这个值。

6.3.3　总线访问限制

一个中断传输的端点可以指定一个它所想要的总线访问周期，也就是一个事务与另一个事务之间的时间间隔。中断传输没有稳定的传输速率，只是保证事务之间的间隔时间不会超过这个最大值。设备的端点描述符规定了这个间隔值（bInterval 字段）。如果是全速设备，最大间隔可以在 1~255ms 之间；如果是低速设备，这个值将被限制在 10~255ms 之间；如果是高速设备，最大间隔可以在 125μs~4s 之间（通过 2EXP（bInterval-1）运算而得，单位是 125μs，bInterval 在 1~16 之间）。

USB 系统软件所提供的周期可能比端点所需要的要短，所以，中断传输无法保证一个稳定的传输速率，但这个周期值最短只能是 125μs 或 1ms。

高速的中断传输可以在每个微帧时间内传输 3072 个字节的数据包，也就是 192Mbit/s，需要高于每个微帧 1024 字节的中断端点，被称为高带宽端点。一个高带宽端点在一个微帧中使用多个传输事务。而且高带宽的端点必须指定事务间隔为 1×125μs（bInterval 的值为 1）。

全速的中断传输可以在每一帧内传输 64 个字节，也就是 512kbit/s。低速的中断传输可以

每 10ms 内传输 8 个字节，也就是 4.8kbit/s。

主机可以在指定的最大间隔内的任何时间开始一个中断事务。以全速时最大间隔 10ms 为例，5 个传输可以用掉 50ms，也可以只用 5ms。然而，OHCI 主机控制器使用 2 的指数值来作为最大间隔（最大是 32ms）。所以，一个最大间隔在 8~15ms 之间的全速设备，主机会以每 8ms 的间隔开始一个传输事务。如果是 32~255ms 的最大间隔，主机会以 32ms 的间隔开始一个传输事务。

一个空闲总线理论上可以在每个帧中传输 19 个 64 字节的全速中断传输，在低速时，是 6 个 8 字节的中断传输。实际上，主机不可能把 19 个传输安排在一个帧内，因此，实际的最大速率要小于这个值。

低速、全速和高速设备都支持中断传输。高速端点最多可以分配微帧中 80%的时间用于周期性的传输（中断传输和等时传输），而低速和全速端点最多可以使用一帧中 90%的时间。低速、全速和高速端点中断传输的速度限制如表 6-6、表 6-7 和表 6-8 所示。

表 6-6　　低速中断传输限制

	协议包头（共 19 个字节，包括 5 个 SYNC 字节，5 个 PID 字节，2 个端点+CRC 字节，2 个 CRC 字节，1 个 5 字节的数据包间延迟（EOP 等））					
	数据有效载荷	最大带宽 B/s	每个传输的帧的带宽	最大传输数	剩余字节	有用数据 B/Frame
	1	9000	11%	9	7	9
	2	16000	11%	8	19	16
	4	32000	12%	8	3	32
	8	48000	14%	6	25	48
最大		187500				187

表 6-7　　全速中断传输限制

	协议表头（共 13 个字节，包括 3 个 SYNC 字节，3 个 PID 字节，2 个端点+CRC 字节和 1 个 3 字节的中间包延时（EOP 等））					
	数据有效载荷	最大带宽 B/Frame	每个传输的帧的带宽	最大传输数	剩余字节	有用数据 B/Frame
	1	107000	1%	107	2	107
	2	200000	1%	100	0	200
	4	352000	1%	88	4	352
	8	568000	1%	71	9	568
	16	816000	2%	51	21	816
	32	1056000	3%	33	15	1056
	64	1216000	5%	19	37	1216
最大		1500000				1500

表 6-8　　高速中断传输的限制

	协议表头（以 480Mbit/s 及 8 bit 信息包间间隔为基础，包括 88 bit 的最小总线转换，32 bit SYNC，8 bit EOP，3×4 bit SYNC 字节，3 个 PID 字节，2 个 EP/ADDR+CRC，2 个 CRC16，一个 3×(1+11) 字节的数据包间延迟（EOP 等））

	数据有效载荷	最大带宽 B/Frame	每个传输的帧的带宽	最大传输数	剩余字节	有用数据 B/Frame
	1	1064000	1%	133	52	133

续表

	数据有效载荷	最大带宽 B/Frame	每个传输的帧的带宽	最大传输数	剩余字节	有用数据 B/Frame
	2	2096000	1%	131	33	262
	4	4064000	1%	127	7	508
	8	7616000	1%	119	3	952
	16	13440000	1%	105	45	1680
	32	22016000	1%	86	18	2752
	64	32256000	2%	63	3	4032
	128	40960000	2%	40	180	5120
	256	49152000	4%	24	36	6144
	512	53248000	8%	13	129	6656
	1024	49152000	14%	6	1026	6144
	2048	49152000	28%	3	1191	6144
	3072	49152000	42%	2	1246	6144
最大		60000000				7500

6.3.4　数据顺序和错误检测

如果设备没有返回一个预期的联络信息包，主机控制器会再试两次。主机如果收到的是 NAK，那它也会重试。中断传输也使用数据交替位，来确保传输的准确性。当成功进行一次传输，数据交替位就改变一次。

6.4　等时传输

等时传输适用于以固定速率或在固定时间内的传输。对于由于错误而导致的传输失败，主机不会进行重新传输。因此，可以容忍偶尔的错误。在全速情况下，等时传输每个帧传输的数据要比中断传输的多。等时传输的应用包括实时的语音和声音码流。这些数据尽管要使用固定的速率传输，但是并不一定非要等时传输。例如，主机也可以使用批量传输，传送一个音乐文件设备。设备接受到该文件之后，再以一个固定的速率播出。

等时传输是保证大量数据可以迅速通过繁忙总线的一个很好的方式。数据不一定需要以固定的速率传输。与批量传输不同，一旦等时传输开始之后，主机会确保在预期的时间内完成。因此，等时传输是可以预测的。

只有全速和高速设备才支持等时传输。设备并不一定需要支持等时传输，但是设备类会需要它。

6.4.1　数据格式

等时传输以一个固定的速率传输数据，即每帧或每微帧传输固定字节的数据。所有其他的传输类型都不能保证每帧或每微帧传送指定数量的字节数据。

一个全速的等时传输，在每帧内只能有一个 IN 或 OUT 事务。高速的等时传输最多可以支持每微帧内 3 个 IN 或 OUT 事务，最少可以在 32768 个微帧内包含一个等时传输事务。等时传输也是单方向传输的。所有的事务要么全是 IN 事务，要么全是 OUT 事务。如果有设备要求双向的等时通信流，那么必须有两个等时管道，分别进行不同方向的等时传输。这两个方向的事务如图 6-10 和图 6-11 所示。

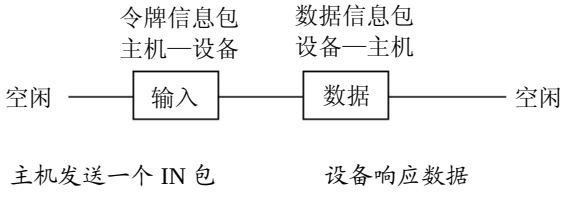


图 6-10　全速的等时传输事务

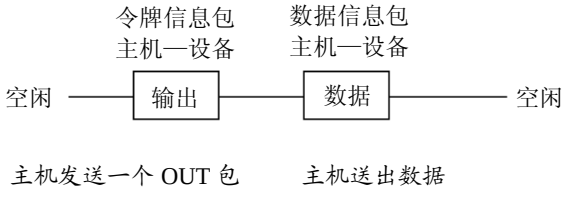


图 6-11　全速的等时传输事务

在配置等时传输的管道之前，主机控制器会将要求的缓冲区大小与总线上目前可用的、未保留的带宽作比较，从而决定所请求的带宽是否可以得到。一个每帧 1023 个字节的全速中断传输使用 USB 总线带宽的 69%。如果有两个全速的设备都要求建立每帧 1023 个字节的中断传输，则主机将会拒绝配置请求第二个管道的设备，因为数据无法在可用的时间里传输完成。

虽然等时传输可以在每帧中传输固定字节数目的数据，但并不是每帧传输一个恒定的数据位。每一个事务都有一个头部，并且与其他设备共享总线。所有数据实际上是以 12Mbit/s 或 480Mbit/s 的速率在一个帧或微帧内的任何时候传输的。

等时传输可以与其他的数据来源、接受者或 USB 的帧开端信号同步。例如，一个麦克风的输入可以和一个扩音器的输出同步。USB 规范描述了几种与内部和外部时钟同步的方法。USB 2.0 协议里等时端点的描述符可以指定一个同步类型以及一个有用的数值，用来指明端点是包含数据或反馈（Feed Back）信息来维持同步的。

如果在高速总线上与全速的设备做等时传输，主机会使用分割事务来处理传输中的所有事务。等时传输事务使用的是开始分割事务，而不是完成分割事务，因为它没有可以回报给主机的状态信息。等时传输不使用 PING 协议。

6.4.2 数据包大小限制

对于全速等时端点来说，最大数据有效载荷的大小是 1023 个字节。高速等时端点的最大数据有效载荷可以达到 1024 个字节。如果传输的数据太多而无法放入一个数据包内，则主机控制器会使用多个事务来进行传输。

每帧的数据量不必一样。例如，一个每秒 44100 样本的数据，可以使用 9 个帧，每帧包含 44 个字节，余下的每帧包含 45 个字节。

6.4.3 总线访问限制

只有全速和高速端点支持等时传输。全速的周期性传输（包括等时传输和中断传输）可以使用不超过一个帧时的 90%的时间，高速的等时传输和中断传输可以使用不超过一个微帧时的 80%的时间。

一个全速的等时传输最多每帧中包含 1023 个字节，即 1.023MB/s。这样就为其他类型的传输留下了 31%的总线带宽。对于具有一个数据包的传输，每个传输数据包的协议头传输 9 个字节。

一个高速的等时端点最多每微帧传输 3072 个字节，也就是 192Mbit/s。在每个微帧内要求传输超过 1024 个字节的高速等时端点，被称为高带宽端点。一个高带宽端点可以每个微帧使用多个传输，并且必须指定周期为 1×125μs（bInterval 的值为 1）。全速和高速时等时传输的总线访问限制如表 6-9 和表 6-10 所示。

表 6-9　　全速等时传输限制

	协议表头（共 9 个字节，包括 2 个 SYNC 字节，2 个 PID 字节，2 个端点+CRC 字节，2 个 CRC 字节，1 个字节的数据包间延迟（EOP 等））					
	数据有效载荷	最大带宽 B/Frame	每个传输的帧的带宽	最大传输数	剩余字节	有用数据 B/Frame
	1	150000	1%	150	0	150
	2	272000	1%	136	4	272
	4	460000	1%	115	5	460
	8	704000	1%	88	4	704
	16	960000	2%	60	0	960
	32	1152000	3%	36	24	1152
	64	1280000	5%	20	40	1280
	128	1280000	9%	10	130	1280
	256	1280000	18%	5	175	1280
	512	1024000	35%	2	458	1024
	1023	1023000	69%	1	468	1023
最大		1500000				1500

表 6-10　　高速等时传输限制

	协议表头（以 480Mbit/s 及 8 bit 信息包间间隔为基础，包括 88 bit 的最小总线转换 32 bit SYNC，8 bit EOP，2×4 个 SYNC 字节，2 个 PID 字节，2 个 EP/ADDR+CRC，2 个 CRC16，一个 2×(1+11) 字节的数据包间延迟（EOP 等））					
	数据有效载荷	最大带宽 B/Frame	每个传输的帧的带宽	最大传输数	剩余字节	有用数据字节/Frame
	1	1536000	1%	192	12	192
	2	2992000	1%	187	20	374
	4	5696000	1%	178	24	712
	8	10432000	1%	163	2	1304
	16	17664000	1%	138	48	2208
	32	27392000	1%	107	10	3424
	64	37376000	1%	73	54	4672
	128	46080000	2%	45	30	5760
	256	51200000	4%	25	150	6400
	512	53248000	7%	13	350	6656
	1024	57344000	14%	7	66	7168
	2048	79152000	28%	3	1242	6144
	3072	49152000	41%	2	1280	6144
最大		60000000				7500

6.4.4　错误检测

等时传输不支持重传以响应总线出现的错误，这正是为了保证在指定时间内传输大量数据所作出的必须的牺牲。USB 不允许对一个等时管道的发送器返回握手信息。因为对于等时传输来说，传输的时间性要比正确性更加重要。等时传输使用在可以接受偶尔发生小错误的传输上，例如语音或者音乐数据传输上的短暂错误。不过，在正常情况下，一个 USB 传输最多只能有一次噪声引起的很偶然的错误。因为在等时传输中，接收器不能在忙或者一检测到错误时就要求重新发送数据。如果接收器怀疑数据有错，它可以要求发送器重传整个传输，不过这种做法的效率很低。

6.5　本章小结

本章从 USB 数据流的角度对 USB 的 4 种传输类型进行了深入的介绍，包括各种传输类型数据包的格式、大小和传输速率等。在第 7章中将介绍 USB的机械特性。

第7章　机械特性

知识点：

- USB 的连接器
- USB 的线缆
- USB 的电气特性
- USB 信号的层次

本章导读：

USB 设备的生命力是多方面造就的，除了前几章所介绍的即插即用特性之外，USB 接口本身的小巧与灵活，造价的低廉等，都是 USB 接口的吸引人之处。本章主要从 USB 的机械电气特性方面来讲述它是如何方便用户的使用。

7.1　综述

USB 的物理拓扑框架包含了集线器向下游集线器或设备提供的连接方式。USB 可以工作在 3 种模式下：在高速模式（480Mbit/s）和全速模式下都需要两根电源线和两根双绞线，且都需要进行屏蔽；在而低速模式下推荐（但不要求）使用双绞线。

连接头应被设计为插拔式的。在插头上面应有一个突起的 USB 标志，使得使用者更容易定位。

7.2　内建的连接器的协议

为了最大限度地减少使用者的问题，USB 使用了一个“键控的连接器”协议。它将接头分为系列“A”和“B”连接器，从而更方便地供用户使用。“A”系列是提供设备向上游的接头或是集线器向下游的端口的，所有 USB 设备必须配备本章所讲到的“A”系列连接器；“B”系列连接器允许设备供应商提供一种标准的可分离的线缆，这会方便用户使用。

USB 系统中可能用到的各种插头和接口的示意图如图 7-1 所示。

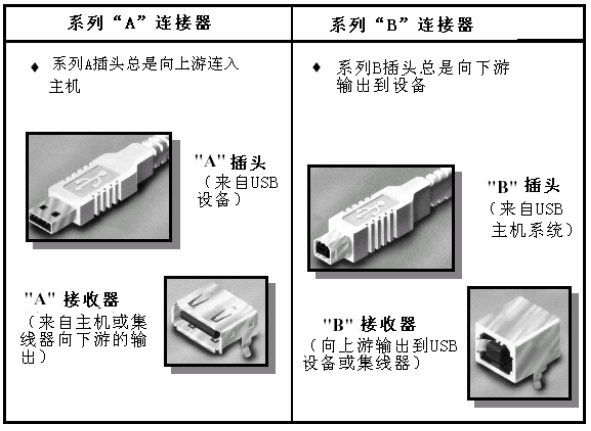


图 7-1　USB 的各个系列的插头和接口

下列规范列出了怎样使用系列插头和接口：

- 系列“A”接口与系列“A”插头。系列“A”接口在电气上提供了从主机系统或是集线器来的连接。
- 系列“A”插头与系列“A”接口。系列“A”插头总是向上连入主机系统。
- 系列“B”接口与系列“B”插头。系列“B”接口在电气上提供了进入集线器或设备的连接。
- 系列“B”插头和系列“B”接口。系列“B”插头总是向上连入 USB 集线器或设备。

7.3　线缆

USB 的连接头由 4 条线缆组成：两条电源线和两条信号线。高/全速的线缆包括一对双绞信号线、一条 VBUS、一条 GND，及对它们的全屏蔽。高/全速的线缆也可以用于全/低速的设备。同时，USB 规范推荐制造商使用 3 种表皮颜色：白色、灰色或黑色。

7.4　线缆组件

USB 规范关于线缆的规定包含 3 个方面的内容：标准的可分离的线缆、高/全速的束缚型电缆和低速的束缚型线缆。

一根标准的可分离的线缆是高/全速线缆中的一种，它的两头分别由系列插头“A”和系列插头“B”构成。一根高/全速的束缚型的线缆的一头是系列“A”插头，另一头则是高/全速的外围设备。低速的束缚型的线缆一头为系列“A”插头，另一头是低速的外设。任何其他的线缆组合都不是 USB 规范所推荐的。

7.4.1　标准的可分离的线缆组合

高/全速的设备可以利用“B”插头，这便允许设备使用标准的可分离的线缆。于是，免除了在设备上还要连入固定的线缆的麻烦，同时最大化地方便了使用者。

使用系列“B”接口的设备必须考虑在最大可能的情况下减小线缆的长度。标准的可分离的线缆集合只能用于高/全速下，如用于低速设备，则其线缆可能超过允许使用的长度。

标准的可分离线缆的示意图如图 7-2 所示。

标准的可分离的线缆组合必须满足以下电气需求：

- 线缆的一头为铸模的系列“A”插头，另一头为铸模的系列“B”插头。
- 线缆必须应用于高速或全速模式。
- 线缆的阻抗必须与高/全速驱动器阻抗匹配。
- 在两个信号导线间产生的传输延时的差别要尽可能地减小。
- 最大允许的线缆长度取决于信号线的衰减和传输的延时。
- GND 线提供了一个上游端口和下游端口间的参考的电平。最大线缆长度很大程度上取决于经过 GND 线时电压的下降。最小的可接受的线缆规格由所连接的设备的最高电压计算得到。
- VBUS 线提供了设备的电源。对于标准的可分离的线缆，其要求与 GND 类似。

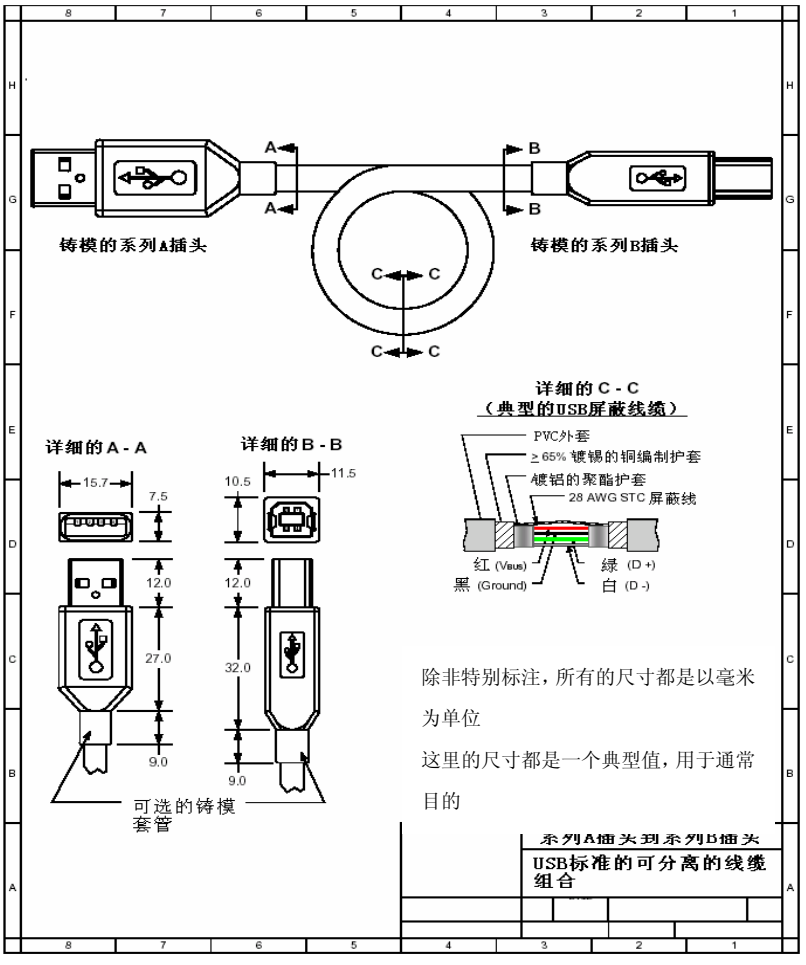


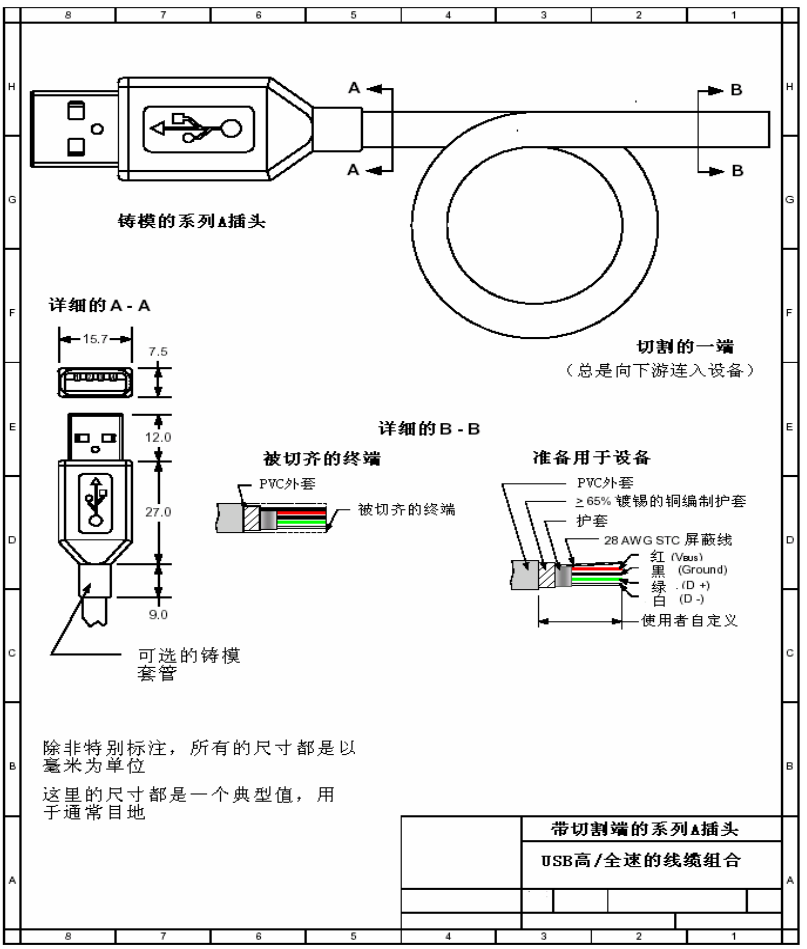
图 7-2　标准的可分离的 USB 线缆

7.4.2　高/全速的束缚型的线缆组合

在一些特殊的应用场合，常常会见到束缚型线缆组合的应用。高/全速的束缚型线缆可以应用在高/全/低速情况下，但对于低速设备来说，线缆应满足其长度的规定。高/全速束缚型线缆组合如图 7-3 所示。

高/全速束缚型线缆的组合必须满足以下的电气需求：

- 线缆的一头必须为系列“A”插头，另一端为相应的制造商提供的产品。如果制造商规定的内部连接必须为热插拔，那么它必须满足与 USB 的系列“B”连接器同样的需求。
- 线缆同样可用于全/低速。
- 线缆的阻抗必须满足高/全速驱动器的需求。
- 线缆最大允许的长度取决于信号线产生的衰减及其阻抗。
- 在两条信号线之间产生的传输延时差别要尽可能被减小。
- GND 线提供了上下游端口间的通用参考面。最大线缆长度同时也取决于电压通过 GND 线时产生的压降。
- VBUS 线提供了到设备的电源。



这里的尺寸都是一个典型值，用于通常目的

图 7-3　USB 高/全速束缚型线缆组合

7.4.3　低速的束缚型线缆组合

在制造商特别需求的情况下，通常会考虑束缚型的线缆，低速的线缆只能用于低速的设备。低速束缚型线缆组合如图 7-4 所示。

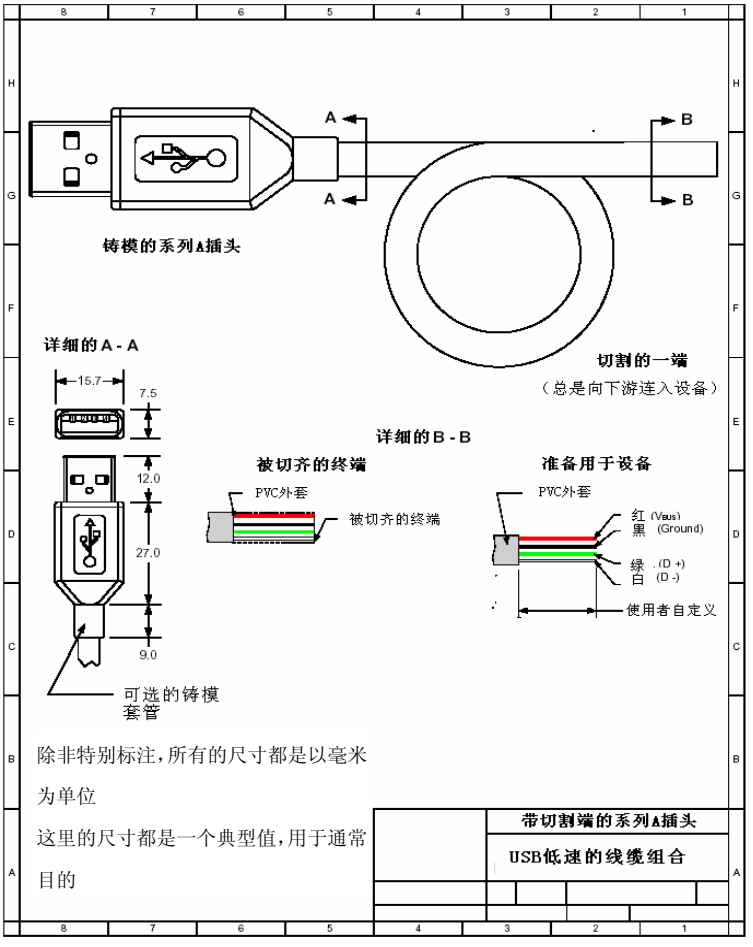


图 7-4　USB 低速束缚型线缆组合

线缆的电气规格应满足的条件与高/全速时大体相同，但要特别注意以下几点：

❑　通常情况下，低速的线缆必须在一定的电容范围内工作，它包括了所有 D－和 D+线上的电容，而不仅仅是线缆本身的电容。因此，线缆的总电容必须介于规定的最大值和最小值之间。如果所设计的产品没能达到所需的最小电容值，那么应该在设备上加入附加的电容。

❑　低速线缆的最大长度由低速信号的上升和下降时间决定，这便使得低速的线缆要比全高速的短很多。

7.4.4 被禁止的线缆组合

USB 设计之初的目的是为了方便用户的使用，它希望只要设备被连入，设备便能工作。规范规定的可能限制 USB 使用的情况只有以下几种：

- 掉电。
- 带宽不足。
- 超出了最大连接数。

同时，规定了非法的线缆组合使用的情况，这些组合在特定场合也许可以工作，但并不能想当然地将其应用于所有场合。要注意以下情况的使用：

- 特殊的线缆组合，例如，线缆的两头同为“A”系列或“B”系列。这样可以方便地将几条线缆连在一起，但要注意决不能超出最大允许长度（30m）。
- 线缆组合违反 USB 拓扑规则的情况，例如，线缆的两头同为“A”系列的插头或同为“B”系列的接口。这可能会使两个下游端口直接相连。
- 标准的可分离的线缆用于低速设备。低速设备不允许使用标准的可分离的线缆，这是因为通常情况下标准的可分离的线缆长度总会超过低速设备允许的最大长度。

7.5 USB 连接器的终端数据

标准的终端设计的连接器的有关数据如表 7-1 所示。

表 7-1 USB 连接头的设计规定

序号	信号名称	典型的线缆设计
1	VBUS	Red
2	D－	White
3	D+	Green
4	GND	Black
Shell	保护层	Drain Wire

7.6 线缆的机械构造和材料需求

高/全速线缆与低速线缆之间的区别在于数据线的排列和屏蔽等方面不同。低速线缆中的两条数据线推荐使用双绞线，但并不作硬性要求；同时推荐使用镀锡的铜金属网做为外层保护，但并不作硬性要求。典型的高/全速线缆的结构如图 7-5 所示。

全/高速线缆由一条电源线，一条地线，一对差分的数据线组成。各条线的机械电气要求分别如下：

❑　电源线：由一对截面面积为 20~80 个 AWG 单位（AWG 是美国线缆标准，用来衡量一条线缆的横截面面积）的非双绞的线缆构成。

❑　数据线：由一对截面面积为 28 个 AWG 单位的双绞线构成。

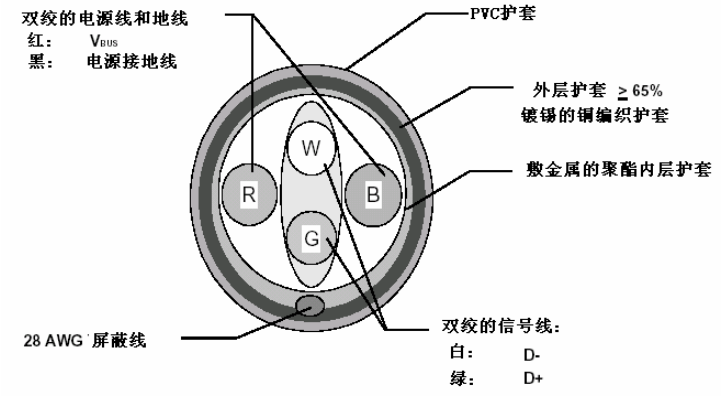


图 7-5　高/全速线缆的结构

7.7　关于 USB 的电气特性

USB 2.0 的规范要求 HUB 对高速模式的支持，而 USB 2.0 的设备却不一定要求支持高速的模式。同时，高速的面向上游的接收器的线缆不能用于传输低速的信号模式传输。而 USB 2.0 的下游的接收器必须支持高速、全速、低速模式。

为了确保在高速数据模式下的可靠操作，规范要求线缆的使用必须能够满足当前所有的规范。

7.8　信号

7.8.1　高速信号的概述

USB 的高速信号要经过屏蔽的双绞线来传输，且该线缆往往要与所有 USB 规范都保持一致。高速的接收电路的示意图如图 7-6 所示。

可以看出，高速电路的基础 50%以上来源于 USB 1.1 的接收器，但其中加入了高速操作所必须的新的元素。高速的操作支持 480Mbit/s 的传输速率。为了可靠地实现这一目标，在每一条线缆的终端，都用电阻连入了地线。该电阻的标准值为差分线缆阻抗的 1/2 或是 45Ω。

对于一个在高速模式下的连接操作，当两条线缆的终端接收器都接地，并且设备任何一个接收器中有信号驱动进入 D+或 D－线时，便会出现一个空闲状态。这个信号状态是通过使用全/低速的驱动器来发出一个单独的“0”信号来表示的。同时，要实现该状态，就要使驱

动器的输出阻抗和 Rs 电阻的总和尽量与标称值接近。推荐的作法是使驱动器的固有阻抗尽可能地小，而使 Rs 的值尽可能接近 45Ω。这通常会达到最为理想的效果。

为了在高速模式下传送数据，接收器通常会用到一个内部的电流源，它由其电压源的正端引出，然后将这一电流通过一高速的电流转换接头引入两条数据线中的一条。通过这种方法，接收器在线缆上发出高速的 J、K 状态。

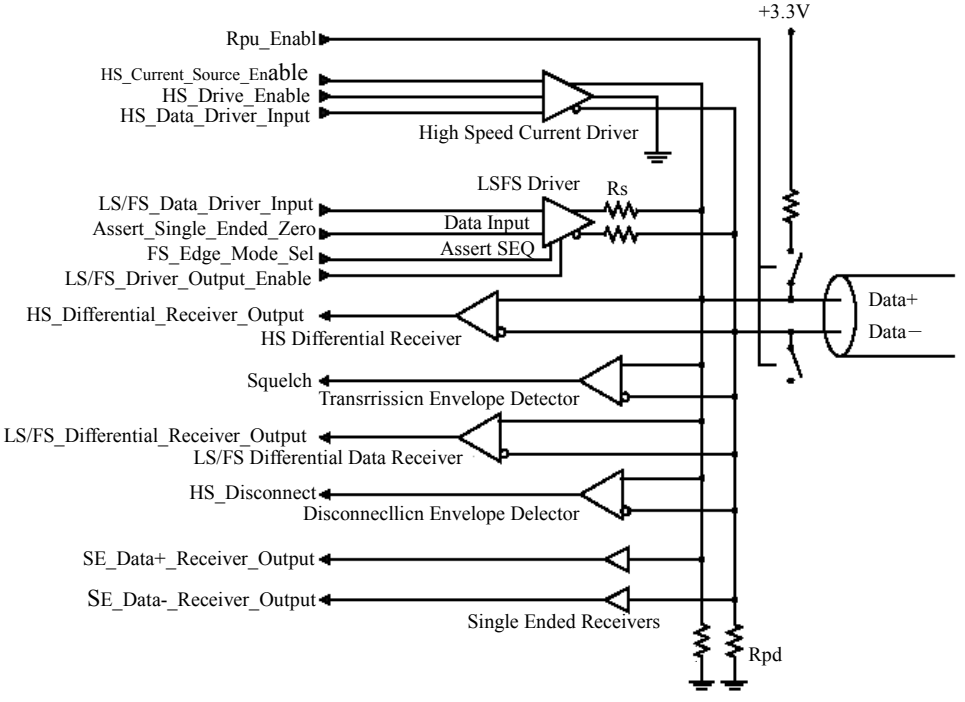


图 7-6　高速的接收线路

这种将电流动态地切换入 D+或 D－线，并随之以 NRZI 方式进行数据解码的方法，同样可用在全速/低速中，同时也会用在传送的数据包的位填充中。为了发出 J 状态，电流被导入 D+线；为了发出 K 状态，电流被导入 D－线。在高速模式下要对 SYNC 域和 EOP 中的分隔符作适当地修改。电流源的大小以及终端电阻的值必须被控制在规定的范围内，这两者合起来决定了实际的电压驱动水准。从 D+或 D－线到地的电阻 DC 大小为 45Ω±10%。在这两条线上得到的电压输出值的不同必须被限定在±400mV±10%。

在信号线上传输的差分电压通常用于以下 3 种目的：

- 从（用）接收线缆终端的差分接收器来接收差分信号。
- 在接收线终端的差分封装的探测器，用来决定连接何时应处于压制状态。接收器使用压制探测来提示连接器上的信号是否有效。
- 在面向下游的集线器接收器中，差分的探测器用以监视连接器上的信号是否处于高速状态。在这样的情况下，接收器需要在特殊的时间点处检测是否是高速的状态，而这一时间点往往是 3 个 SOF 数据包的开始。

同样，它也用于检测连接的断开，在缺少远程终端设备的情况下，差分的电压通常会产生倍增。

USB 2.0 要求面向下游的接收器必须能够工作于低速、全速、高速模式。向上游的高速接收器不允许工作于低速模式，但却要求可以工作于全速模式。因此，在用于高速设备的 D－线上不能再接入 1.5kΩ 的下拉电阻。

表 7-2 描述了将图 7-6 作为范例时，高速的接收器所需的功能元素。

表 7-2　　接收器中的功能元素描述

元素	描述
全/低速的驱动器	全/低速的驱动器用于全/低速的数据传输，它需要满足所有的在 USB1.1 规范中规定的要求，除了一个例外情况：高速的接收器中每一个输出阻抗，包括 R_s 必须介于 45Ω±10%之间。高速模式的操作必须使驱动器连接 D+和 D－线到地线，这样根据上述章节中对输出阻抗的要求，便能够很好地满足信号线到地线的连接阻抗的要求
全/低速的差分接收器	全低速的差分接收器用于接收全/低速的数据传输
单端点的接收器	单端点的接收器用于全/低速的信号接收
高速的电流驱动器	高速的电流驱动器用于高速的数据传送，电流从其电源的正极性引出，然后切换入两条数据线，从而产生 J、K 状态。标称的电流值为 17.78mA，当这一电流从信号线经过上述的电阻流入地线时，标称的高电平为(V_HSOH)+400mV，标称的对于两条信号线上的 J 状态电平为 400mV，K 状态为-400mV 当接收器中没有事务处理时，应当关闭该电流源
高速的差分数据接收器	高速的差分数据接收器用于接收高速的数据。对于设计者来说，既可以将两种数据接收器合并在一起，也可以使其单独运作
传送封装探测器	这一探测器用于检测当接收器的输入信号的峰值低于抑制电压时，输入的信号是否有效
断开连接封装探测器	这一探测器用在下游端口，用于检测信号线上的断开连接状态
上拉电阻	这一电阻只用于上游端口的接收器，用于示意端口的传输速度。高速的设备在连入的最初是作为全速设备使用的，但在初始化完成之后，数据的传输必须以高速来进行。一旦其运作于高速，那么 1.5kΩ 的电阻必须在电气上与信号线断开连接。图 7-6 中一条被称为 RPU_Enable 的控制线就用于实现这一目的
下拉电阻	这一电阻只用于面向下游的端口

7.8.2　USB 驱动器特性

USB 使用差分输出驱动器将 USB 的数据信号送入 USB 的线缆。对于全速和低速的操作，驱动器静态输出电压范围为：下界 *VOL*=0.3V，通过一只 1.5kΩ 的电阻将 3.6V 的电压引入得到；上界 *VOM*=2.8V，通过同一只 1.5kΩ 的电阻连入地线得到。

输出信号介于两者之间的抖动值必须尽可能地被降到最低。与幅射噪声和交叉噪声相关的由驱动器控制的固转速率也应该得到很好地控制。驱动器的输出还必须支持 3 种速度模式

下的双工双向的数据传输。

注意：短路现象的承受能力。USB 的收发器需要有能力承受其 D+、D－、VBUS、GND 间的短路状态，而且时限不低于 24 小时。因此，推荐设计的收发器必须能承受这样的不确定的短路状态。设备在这种短路状态下，必须能够保证收发状态都不被损坏。并且在最大电压 *VBUS*=3.5V 时也必须能满足这样的要求。

7.8.3　高速接收器的特性

如图 7-6 所示，高速模式下的收发器利用高速的差分数据接收器和传送探测器来“收听”到来的串行数据流。同时，面向下游的收发器利用“断开连接探测器”来监视信号线上的差分信号幅值的变化。

当在高速模式下接收数据时，高速的接收器应该有能力接收混杂于－50mV~500mV 之间的共模干扰电压之中的信号（标准的高速信号的共模干扰为 200mV）。低频率的 J、K 信号通常出现在重启的握手过程中，也应该能够在有－50mV~+600mV 共模干扰的情况下被接收到。

接收到的数据受限于传送探测器的输出。接收器对于低于抑制电压 VHSSQ（≤100nV）的信号可以不予恢复。探测器必须在差分电压的幅度变化小于 100mV 时给出抑制信号的提示；在变化大于 150mV 时，便脱离抑制。

抑制的探测必须用一个差分的幅度变化探测器来完成。对于高速数据包的 SYNC 方案的定义，要求与高速的集线器中继器一起，保证接收器能够收到最少 12 位的 SYNC（即 KJ KJ KJ KJ KJ KJ），其后为数据部分。这便意味着抑制响应时间、DLL 锁定时间以及 SYNC 探测的完成，都必须在 12 个时钟内同期完成。这样是为了保证数据有效载荷的第一位能够被正确接收。在面向下游端口的情况下，高速收发器必须包括一个差分探测器，用来给出何时数据信号超出了高速脱离连接的电平值的提示。如果差分信号电压大于 625mV，给出提示；如果小于 525mV，则不予提示。

在适当的时间中采样完成后，探测器便能判断连接是否断开。

7.9　设备速度的检测

7.9.1　全/低速设备速度的检测

全速和低速设备是根据位于其下游端点的尽头处的上拉电阻的位置来确定的。

- 全速的设备是由位于 D+线上的上拉电阻来作为终结点的，如图 7-7 所示。

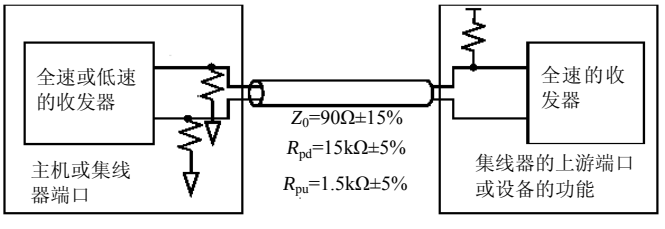


图 7-7　全速设备的线缆和电阻的连接

- 低速的设备是由位于 D－线上的上拉电阻来作为终结点的，如图 7-8 所示。

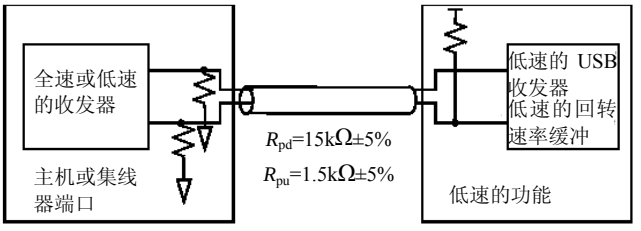


图 7-8　低速的设备和线缆的连接

- 主机或集线器中下游的端口的下拉电阻是连入地线的 15kΩ±5%的电阻。

注意：终结端的电阻不应包括位于主机和集线器中的 15kΩ±5%的下拉电阻。下拉电阻的电压源是从 VBUS 线上分出来的，因此，一但 VBUS 线被移除，下拉电阻便不能对数据线进行正常的供电。

7.9.2　高速设备的检测

高速模式的重启和探测机制与全/低速下的类似，当重启完成之后，设备必须工作于合适的模式，而且在端口状态寄存器中速度表示位也应正确地反映这种模式。软件上，只需要初始化中断，并且读取反映重启完成的状态寄存器即可。

高速设备在初始化时是作为全速应用的。这便意味着对于面向上游的端口，RPU（1.5kΩ±5%）必须被连入 D+与 3.3V 电源线之间，而且要被一只可以被 SW 控制的开关所控制。

在初始化完成后，高速设备利用底层协议来完成高速的连接，并使反应下游端口的工作状态的寄存器作出恰当的反应。

7.10　输入特性

7.10.1　全/低速的输入特性

收发器终端的输入阻抗允许值为 7300k Ω（Z_{INP}）。

在每一引脚测量可以测得端口的输入电容，下游的端口可以使用不同的输入电容。下游端口 D+、D－线上最大允许的输入电容为 150pF。它由以下几部分构成：

- 每一个收发器和连接器的集总电容：75pF。
- 接口和收发器间的每一根导线的 75pF 电容。

面向上游端口的最大电容在可分离式的电缆情况下为 100pF，这由以下几部分构成：

- 每一个收发器和连接器的集总电容：75pF。
- 接口和收发器间每一根导线的电容：25pF，D+和 D－线上电容的差异必须小于 10%。

全速束缚型电缆的设备本身可能具有在 D+、D－线上的 75pF 的集总电容。

低速的设备要求使用束缚型的电缆，低速设备的电容应该包括线缆上的电容。最大的单端或是差分输入电容为 450pF（C_{LINVA}）。对于采用束缚型线缆的设备，单端输入电容必须能够将 D+和 D－线的电压在 2.5μs 内控制在 0V~VIH 之间。D+或 D－线上的电容包括单端的设备输入电容和大小为 150pF 的主机/集线器输入电容。

在实际的实现方案中可能会用边缘上的电容来更好地控制收发器的边缘速率。因此，附加电容（C_{EDGE}）和收发器跟踪连接的电容的总和不能超过 75pF。用于边缘速率控制的电容必须被连接在收发器的引脚和 R_S 之间，如图 7-9 所示。

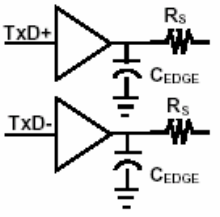


图 7-9　全/低速中可选的边缘速率控制电阻的设置

7.10.2　高速的输入特性

简单的高速接收模式下的等效电路如图 7-10 所示。

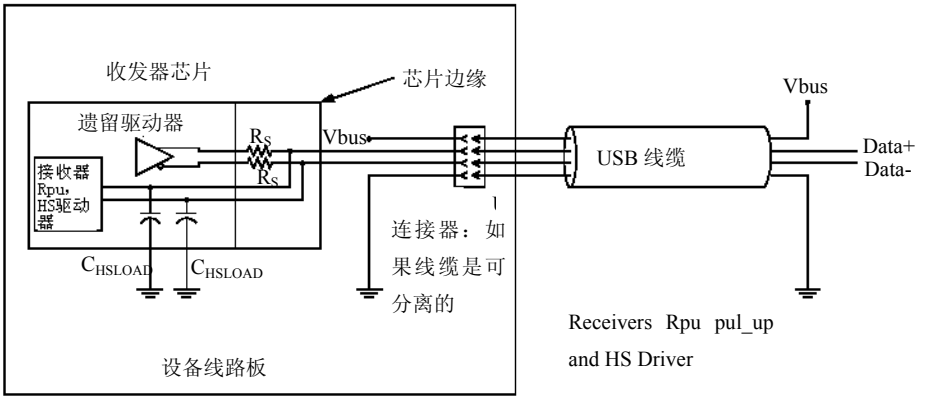


图 7-10　高速模式下的接收电路

收发器工作于高速模式下时必须能够满足下列规范：

- DC 输出电压和电阻规定。
- DTR 加载规范。

另外，工作在此信号模式下的收发器应该满足以下的电容需求：

- DC 输出电压和电阻规范：高速模式下的收发器必须在它的每一只数据引脚处都连入一个标称值为 45kΩ 的电阻到地线。
- TDR 加载规范：高速模式下的空闲状态的 AC 加载规范被称为差分的 TDR（Time Domain Reflectometer）。关于 TDR 在此不作详述，有兴趣的读者可以参阅有关专业书籍。
- 收发器部件的集总电容向导。当将一个收发器芯片作为一个独立的部件时，可以在不同 USB 连接器或线缆的情况下，在芯片的边缘进行测量。测量仪器的寄生电容的影响可以

通过测量结果中叠加部分的影响而得到。如果终端设备是片外的，那么应该在测量过程中加入离散的 R_S 电阻，测量应该在电阻的“连接端”进行。在测量过程中，收发器应该处于 Test_SE0_NAK 模式。

- 允许的每一条线到地的电容：C_{HSLOAD}≤10pF。
- 允许的到地的匹配电容：≤1.0pF。

7.11 信号的层次

7.11.1 全/低速的信号的层次

表 7-3 总结了 USB 的信号层次，表中的第二列是要求信号源发出的驱动信号要求。第三列为所要求的正确接收数据的总路线状态。

表 7-3 全/低速的信号层次－－

总线状态	信号层次		
	信号源（bit time 结束处）	最终的目标连接器	
		要求的范围	可接受的范围
差分信号“1”	D+ > VOH (min)且 D－ < VOL (max)	(D+) - (D－) > 200 mV 且 D+ > VIH (min)	(D+) - (D－) >200 mV
差分信号“0”	D－ > VOH (min) 且 D+ < VOL (max)	(D－) - (D+) > 200 mV 且 D－ <VIH (min)	(D－) - (D+) > 200 mV
单端点“0”SE0	D+ and D－ < VOL (max)	D+and D－<VIL(max)	D+ and D－ < VIH (min)
单端点“1”SE1	D+ and D－ > VOSE1(min)	D+ and D－ > VIL (max)	
数据 J 状态			
全速	差分信号“1”	差分信号“1”	
低速	差分信号“0”	差分信号“0”	
数据 K 状态			
全速	差分信号“0”	差分信号“0”	
低速	差分信号“1”	差分信号“1”	
空闲状态 全速 低速	NA	D－ > VIHZ (min) 且 D+ < VIL (max) D+ > VIHZ (min) 且 D－ < VIL (max)	D－ > VIHZ (min) and D+ < VIH (min) D+ > VIHZ (min) and D－ < VIH (min)
重续状态	数据状态 K	数据状态 K	
数据包的开始（SOP）	数据线从空闲状态切换到状态 K		
数据包的结束（EOP）（4）	持续两个“位”（1）时间的 SE0，然后是一个“位”（3）时间的 J 状态	持续大于一个“位”（2）时间的 SE0，然后是一个“位”时间的 J 状态	持续大于一个“位”（2）时间的 SE0，然后是一个“位”时间的 J 状态

断开连接（位于下游端口处）	NA	大于 2.5μs 的 SE0 时间	
建立连接（位于下游端口处）	NA	大于 2ms 的空闲状态	大于 2.5μs 的空闲时间
重启	D+and D－ <VOL (max) for >=10ms	D+ and D－ < VIL (max) for >=10 ms	D+ and D－ <VIL (max) for >=2.5μs

注 1：用“位”时间定义的 EOP 与传输速度有关。

2：用“位”时间定义的 EOP 与接收 EOP 设备的类型有关。

3：跟随在 EOP 之后的 J 状态的位时间与缓冲器的边缘速率有关。从低速缓冲器到达的 J 状态必须是一个低速的“位”时间，从全速缓冲器到达的 J 状态必须是一个全速的“位”时间。

J、K 是两个用来建立系统中差分信号间的联系的两个逻辑标准。

注意：在接收器中，空闲（Idle）和重续（Resume）状态在逻辑功能上与 J、K 状态相似。

正如表 7-3 所示的用于全速的 J、K 状态是从那些用于低速的 J、K 状态转换而来的。这些数据空闲、重读信号的判断是由那些被连入端口的设备类型所决定的。如果接入端口的是一个全速设备，那么 USB 便使用全速信号的协议，然而在数据线上的数据仍然以低速运行。

低速的信号协议只是用于低速的设备和低速设备连入的端口。

7.11.2　高/全速的信号的层次

高速的信号电压规范如表 7-4 所示。测量时应该利用精确的 45Ω 的电阻接地作为参考面，然后对最靠近收发器的连接器进行测量。

表 7-4　　高速的信号层次

总线状态	在源连接器处所需的信号层次	在目标连接器处所需的信号层次
高速的差分信号“1”	直流层： VHSOH (min) <=D+<=VHSOH (max) VHSOL (min) <=D－ <=VHSOL (max)	交流差分层： 在目标连接器处的信号必须能够被恢复
高速的差分信号“0”	直流层： VHSOH (min) <=D－ <=VHSOH (max) VHSOL (min) <=D+ <=VHSOL (max)	交流差分层： 在目标连接器处的信号必须能够被恢复
高速的 J 状态	高速的差分信号“1”	高速的差分信号“1”
高速的 K 状态	高速的差分信号“0”	高速的差分信号“0”
高频的 J 状态（差分电压，只用于设备和端口都处于高速的情况下）	直流层： VCHIRPJ (min) <= (D+ - D－)<= VCHIRPJ (max)	交流差分层： 目标连接处的差分信号必须大于等于 300mV
高频的 K 状态（差分电压，只用于设备和端口都处于高速的情况）	直流层： VCHIRPK (min) <= (D+ - D－)<= VCHIRPK (max)	交流差分层： 目标连接处的差分信号必须小于等于-300mV
高速的抑制状态	NA	VHSSQ-接收器在差分信号的幅度

		小于 100mV 时给出处于抑制状态的提示，当信号幅度大于 150mV 时脱离抑制状态
高速的空闲状态	NA	直流层： VHSOI min <=(D+, D-)<=VHSOI (max) 交流差分层： 差分信号的幅值小于 100mV

续表

总线状态	在源连接器处所需的信号层次	在目标连接器处所需的信号层次
高速数据包的开始（HSSOP）	数据线从高速的空闲状态切换到高速的 J 或 K 状态	
高速数据包的结束（HSEOP）	数据线从高速的 J 或 K 状态切换到高速的空闲状态	
高速的断开连接状态（面向下游的端口）	NA	VHSDSC-下游端口在差分信号的幅度小于等于 525mV 时不会给出断开连接的提示，当差分信号的幅度大于等于 625mV 时，给出设备断开连接的提示

7.12 连接和断开连接的信号

7.12.1 连接建立和断开的检测

在全/低速的情况下，当没有设备接入集线器或主机的端口时，下拉电阻会使主机或集线器的收发器的端口上的 D+和 D－线都下降到极限电压以下，这便会在下游端口产生一个 SE0 状态。如果主机和集线器不由数据线驱动，同时在下游端口的 SE0 状态持续 T_{DDIS} 时间以上，则会给出断开连接状态的提示。

当两条数据线中的任一条在电压 V_{IH} 以上保持 T_{DCNN} 时间时，便会使集线器探测到有连接的发生。

在发出 SE0 状态使设备处于重启之前，集线器必须通过对总线状态的采样来判断出接入设备的速度。

所有在表 7-3 中给出的信号层次都必须在这一总路线时段被设置完毕，前提是设备的速度已经确定。连接的建立和断开检测过程如图 7-11、图 7-12 和图 7-13 所示。

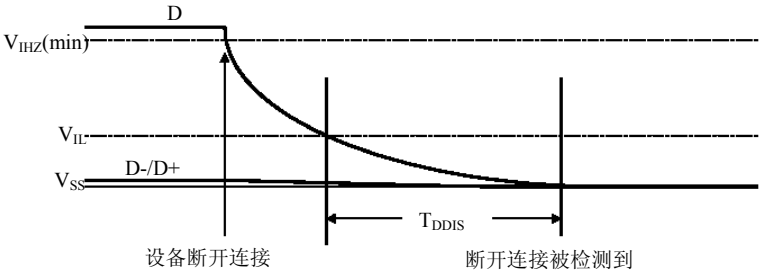


图 7-11　全/低速断开连接的检测

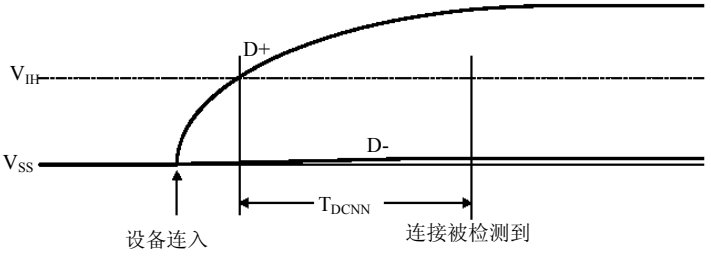


图 7-12　高/全速设备连接的检测

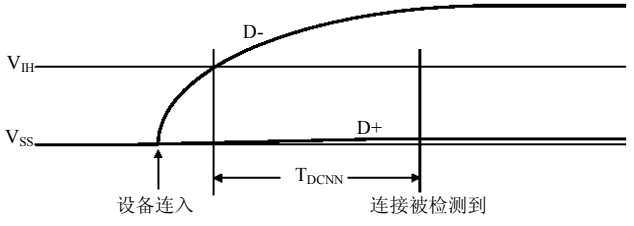


图 7-13　低速设备连接的检测

7.12.2　建立连接的事件时序

因为 USB 的设备被设计为可热插拔的而且可以实现电源的自由切换。因此，使用者有必要了解在电源切换或是设备连入之初以及设备内部电源到达稳定状态之间的时序。

图 7-14 给出了从设备一开始接入主机到其稳定工作之间的事件的时序。其间总共有 6 段延时过程以及一系列的规范定义的事件。

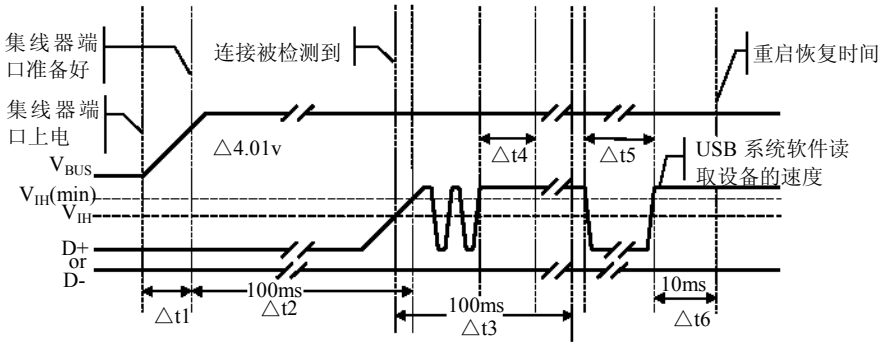


图 7-14　USB 设备上电和连接事件的时序

❑ △t1：这是用于电源切换的时间。这段延时是集线器端口功能的一种。集线器会在其描述符中报告这一时间，并可通过向集线器发送请求来得到该值（在第三章已经有过描述）。如果设备连入的集线器不进行电源切换或已切换完毕，则△t1=0。

❑ △t2（TSIGATT）：它代表着设备的内部电压已经稳定到了 D+或 D－线，已经达到了 VIH 的幅度。△t2 必须小于 100ms。

❑ △t3（TATTDB）：这是 USB 系统软件提供的小于 100ms 的反弹时间。它确保了软件开始对设备重启之前，机械的、电气的连接都能达到一个稳定的状态。它在 USB 系统软件检测到连接事件的发生时开始。

❑ △t4（T2SUSP）：在任何时候，如果设备查觉没有总线的活动时，它必须进入挂起状态。

❑ △t5（TDRST）：这是集线器使一个设备开始重启的时间。

❑ △t6（TRSTRCY）：USB 系统软件用于保证恢复重启的最小时间 10ms，在重启恢复时间未被定义时，设备响应所有的发给默认设备地址的总线事务。

面向下游端口的收发器工作在高速模式时，是通过对 D+和 D－线上信号幅度的变化的检测来感知设备的接入与断开的。当下游端口的收发器从开路状态返回设备脱离的信息后，“断开连接探测器”输出将转为高电平。当差分信号的幅度大于 625mV 时，便要激活探测器，在小于等于 625mV 时，则不会激活探测器。

为了确保这一附加事件有效被探测到，在高速 SOF 的末端的 EOP 被延长到 40 个“位”时间，这一长度能确保下游端口连接器的电压的上升。这是因为允许的最大回环延迟时间为 30 个“位”时间。

当下游端口在高速模式下发现超过 32 个“位”时间的长度没有事务处理的发生后，“断开连接探测器”在下一个收发器输出事务的 8 个“位”时间内的输出必须被采样一次。

7.12.3　数据信号

1. 低速的信号

数据包的开始（SOP）是通过初始端口对 D+、D－线从空闲状态到相反的逻辑状态（K 状态）的驱动发出的。这一状态的切换代表着 SYNC 域的第一位 SE0 状态被用于表示数据包的结束（EOP）。通过对 D+、D－线驱动产生两个“位”时间的 SE0 状态，随后加入一个位

时间的 J 状态，这样便构成了 EOP 从 SE0 态到 J 态的转化，标志着接收器数据的结束。然后，总线终端电阻将总线置于空闲状态。

2. 高速信号

高速信号的空闲状态被定义为两条数据线的电平等于地电平。

高速模式的 SOP 是通过将 D+、D－线从空闲态转入 K 态实现的。同样，K 也是 SYNC 域的第一个标志。高速数据包的结束，是通过在 EOP 之前将最后一个标志转为其相反的状态形成的。这一相反的标志是 EOP 方案（NRZI 01111111）的第一个标志。随着 EOP 的完成，驱动器又再向 D+、D－线中注入电流，信号线转入高速的空闲状态。

7.12.4　重启信号

1. 信号详述

集线器通过向下游端口发出一个扩展的 SE0 来给出重启的信号。在重启过后，设备处于默认状态。重启信号由系统软件在任何主机控制器或集线器的端口上产生，信号至少应该持续 10ms。在重启过后，集线器的端口转入“使能”状态。

对于根集线器的端口来说，重启信号至少应该持续 50ms 以上。这并不是要求在 50ms 内连续地发出 Reset 信号。相反地，每个 Reset 信号间至少应该有 3ms 的间距，而每个 SE0 信号持续 10ms。

当一个工作于全/低速模式的设备在其上游端口看到持续 2.5ms 以上的 SE0 信号时，它便将其作为一个重启信号。

在一个端口处于“使能”状态后，集线器将会持续将重启信号转送到下一个端口。而连入重启端口的设备应该在这一过程前认识到总线的这一活动，从而避免进入“挂起”状态。

在重启信号过后的重启恢复时间的 10ms 后，集线器必须能够接收所有集线器请求，同时，设备也能够接收 Set_Address()请求。

高速的设备必须能够在上电、默认、设置地址、配置或是挂起状态接受重启，同时，位重启信号是与全/低速下的重启一致的。这便意味着集线器可以对下游任意一个设备进行重启，而设备也必须能够接受集线器的重启信号。

2. 高速集线器和设备的重启协议

高速集线器和设备的重启过程应该经历以下几个阶段：

（1）集线器对设备进行检测，保证设备非低速（低速的设备不允许工作于高速模式，如果集线器发现连入的是低速设备，那么它将不会在重启过程中遵守高速重启协议）。

（2）集线器发出 SE0 信号。

（3）设备检测到 SE0 信号。

- 如果设备是从挂起状态重启的，那么设备将在 SE0 持续不少于 2.5ms 的情况下开始检测握手信号。SE0 同时会使晶振时钟重新开始工作。时钟必须能够被及时用于检测高速集线器的高频信号。
- 如果设备是从非挂起的全速状态重启的，那么在检测到 SEO 信号持续了大于 2.5ms

小于 3ms 后，将会开始检测高速的握手信号。

❑ 如果设备是从非挂起的高速状态重启的，那么设备必须在进入全速状态之前等候 3.0ms~3.125ms。进入全速状态是通过移除高速终端并且重新连入上拉电阻来实现的。

（4）在转入全速状态后的 100ms~875ms 之后，设备采样总线，并检测 SEO 信号。如果检测到了 SEO 信号，设备开始高速的握手信号的探测。

提示：从以下步骤开始便是高速检测的握手信号。

（5）高速设备保持 D+上拉电阻的连接，保持高速终端的屏蔽，并且将高速的信号电流驱动进入 D－线。这样将会在总线上产生一个高频的 K 信号，这一信号的持续时间≥1.0ms，≤7.0ms。

（6）集线器要检测信号是否持续 2.5ms 以上，如果未检测到，则持续发出 SEO 直到重启的完成。

（7）总线脱离高频 K 状态 100ms 之内，集线器必须开始送出变频的高频 J、K 信号。J、K 状态之间不能有空闲态。在重启结束之前，这一信号应持续 100ms~500ms。每一个独立的 J、K 信号持续 40ms~60ms。

（8）在集线器高频序列结束后，它仍将发出 SEO 直到重启的结束。在重启结束后，集线器转入高速使能状态。

（9）设备完成高频序列后，它将寻找集线器的高频序列。设备最少应该看到“K-J-K-J-K-J”，从这样一个 K、J 序列中能够判断出是否是一个集线器的高频序列。

（10）如果设备检测到这样的序列，在其后的 500ms 之内，设备需要将 D+上拉电阻断开，从而使高速终端进入高速的默认状态。

（11）如果设备在发出其自己的高频序列后的 1.0~2.5ms 内未检测到集线器的高频序列，那么设备需要转入全速的默认状态，等候重启的结束。

7.12.5 挂起

所有的设备都支持挂起状态，设备可以从任意上电的状态进入挂起状态。设备将在其上游端口出现了连续的空闲态并持续 3.0ms 以上后进入挂起状态。任何上游端口的总线的活动都将使设备脱离挂起。

1. 全局挂起

全局挂起应用在总线所有的地方都没有通信发生的情况下。主机通过停止其所有的传输活动来发出全局挂起信号。每一个下游设备在总线持续一定的空闲态后便进入挂起状态。

2. 选择性的挂起

总线上的区段可以通过对其相应的端口发出 Set_Port_Feature（PORT_SUSPEND）命令使端口进入挂起状态。挂起的端口将会阻止流向相应总线段的活动，同样，在相应区段的设备在总线进入空闲态一段时间后设备将进入挂起。

下游端口工作于高速模式时，在接收到 Set_Port_Feature（PORT_SUSPEND）信号后，立即转入全速空闲态。在持续 4.0ms 以上后，允许进行全速的断开连接测试。

7.12.6　重读

处于挂起状态的设备，在其上游端口接收到非空闲态的信号时，其工作便会重读。在重读信号的发生和传送过程中，集线器扮演着极为重要的作用。

主机可以在任何时候发出重读信号，且持续 20ms 以上。依据下游端口的不同挂起模式，重读信号的结束方法也有所不同。如果下游端口处于低/全速的挂起状态，重读信号必须由一个标准的低速的 EOP 来结束。如果下游端口在挂起前处于高速模式，重读信号由传送高速的空闲态来结束。

在总线重读后，主机必须在空闲态开始的 3ms 内发出数据传送信息，以避免系统再次进入挂起状态。

7.13　数据信号的速率

高速数据信号传送的速率为 480Mbit/s，要求的精确度为±500ppm。

全速的信号传送速率为 12 Mbit/s，精确度为±250ppm。

低速的信号传送速率为 1.5Mbit/s，精确度为±125ppm。

上述速率偏差来自于以下几个方面：

- 初始的频率精确度。
- 加入晶振的电压。
- 温度。
- 老化。

7.14　电力的分配

7.14.1　设备的分类

各种设备对于电力的产生和消耗的种类划分需要引入一个被称为负载单元的概念。一个负载单元被定义为 100mA。设备的负载单元应该以其最大的值来决定。一个设备可能处于只消耗一个负载单元的低电力状态，也可能处于消耗 5 个单元的高电力状态。通常情况下，设备是处于其低电力状态。其由低向高的转化是由软件来控制的，因此，在设备进入高电力消耗时，软件就有必要保证有足够的电力供其使用。

根据不同的电力供应和消耗，USB 将设备分为以下几类：

- 根集线器。

- 总线供电集线器。
- 自身供电集线器。
- 低能耗的总线供电功能。
- 高能耗的总线供电功能。
- 自我供电功能。

7.14.2 在挂起/重续中电力的控制

挂起电流状态是负载单元允许的功能，这时所有的 USB 设备被初始化为低能耗的状态。低能耗或高能耗的设备在工作于低能耗时都被限制于只能获取 500mA 的挂起电流。

处于挂起状态时，设备可以获取不超过平均电流的能量。同时，电流的峰值不能超过 500mA，设备允许获得的最大电流为 100mA（或 500mA）。当设备被唤醒时，它们必须被限制在 VBUS 线上的可用电流范围之内，而引起的集线器上 VBUS 的最大电压降为 330mV。设备必须有足够的上拉电容或有完善的上电排序机制，这样才能够使设备被集体唤醒时在端口出现的电流不超过端口允许的最大电流。

7.14.3 动态地连入和拔除

1. 峰电流的限制

当一个功能或集线器被连入网络时，在设备上应该拥有一个 VBUS 和地线间的板上旁路电容。设备的控制器会在设备连入时给其输出的旁路电容和设备同时供电。因此，如果没有合适的度量方法来限制这一电流，那么将会有大量的电流涌入设备。这样有可能使集线器上的 VBUS 线下降到其最小的可操作的水平。同时，在一个设备从其常态转入高能耗状态时，也可能引发峰涌电流。集线器的 VBUS 线上允许的最大压降为 330mV，或者是连入的功能的标称值的 10%。

2. 动态地拔除

当一个设备从其网络上被动态地拔除时，线缆的自感将会在设备的开路端点处感应出大量的回扫电压。虽然这一电压并不具有破坏性，但是也应该在集线器端口采取合适的电路措施尽可能地抑制耦合入的噪声。这一噪声的频率与线缆的长度成反比，例如一英尺的线缆最大引起的噪声频率为 60Hz，因此需要将一些低容、低自感系数的电容连入每一个集线器端口的连接器。回扫电压和由其引起的噪声将会被线路电容限制在一个适当的范围内。同样，在设备的端点处也应该连入一些小电容，用以确保在线缆的开路端处的感应回扫电压不致于引起设备端点处电压的反转。推荐使用的最小连入 VBUS 线的电容为 1.0μF。

7.15　本章小结

在本章中介绍了 USB 设备应该遵循的机械和电气规范，这一部分的内容非常繁杂，本章选取的内容都是读者在学习 USB 开发时必须掌握的。读者如果想了解更多的内容，可以参照其他的相关书籍。

第 8 章　信号编码与传输

知识点：

- USB 各种数据包的结构
- 数据错误的检测与恢复
- 数据传送的时序

本章导读：

本章介绍了 USB 的信号编码及其在数据线上的传送方式，其中的重点应该是数据的格式、错误的检测与恢复以及数据的传送时序。

在前 3 章中已经介绍了 USB 设备实现即插即用的基础：集线器、主机对设备的检测方式以及控制传输。有了这些知识之后，读者就可以开始下一步的学习了。

很多初次接触 USB 的读者都会提出这样的问题，即 USB 设备为什么能够实现如此之高的数据传送速率呢？要回答这个问题需要综合许多的知识，读者可以建立这样一个概念：实现高速数据的传送需要多方面的配合，包括信号线对数据传送的支持、收发双方的数据同步机制、错误检测恢复等。这里就 USB 协议中关于数据传送的规范作一介绍，这也是 USB 实现高速数据传送的核心部分。

8.1　字节/位顺序

送入总线的数据位首先是最不重要的位（Least-Significant Bit，LSB），紧跟着是次重要位，依次类推直到最重要的位（Most-Significant Bit，MSB）。

多重字节域，包括标准的描述符、请求和响应都要根据 Little-endian order，即 LSB 到 MSB 的方法进行转译和传送。

8.2　SYNC 域

所有的数据包都要以一个同步域（SYNC）开始，这样的域是以保持最大的边缘传送密度序列的方式进行编码的。它使用输入电路，依据本地时钟对到达的数据进行排队。从源收发器来的 SYNC 域在全/低速下定义为 8 个字节，而在高速下定义为 32 个字节。SYNC 的作用是建立一种同步的机制，因此，在随后的数据包中它并不会出现。SYNC 域的最后两位是

SYNC 结束的标志，同时它意味着随后的 PID 的开始。

8.3　数据包域的格式

这一节将要讲述关于标志、数据和握手数据包的模式。每一个数据包位的定义都是以未解码的数据格式进行说明的。为了讲述清楚，我们不考虑 NRZI 编码和位填充的部分。所有的数据包都有明显的开始和结束的界定符。数据包的开始（SOP）界定符是 SYNC 域的一部分，而关于结束界定符（EOP）在第 4 章中已作出阐述。

8.3.1　数据包鉴定域

在每一个 USB 数据包中紧随 SYNC 域的便是数据包鉴定域（PID）。PID 由 4 个包类型位和 4 个校验域构成，如图 8-1 所示。

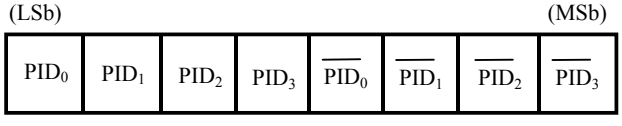


图 8-1　数据包鉴定域的构成

PID 示意了包的类型，同时，可以推断出数据包的格式、错误检测的类型。PID 中的 4 个检测位确保了对 PID 的可靠的解码。数据包校验域是数据包类型域的一部分。如果 4 个 PID 检测位没有完成对期望的检测的填充，那么便说明在 PID 中有错误发生。

主机和所有的功能都要对收到的 PID 域进行完整的解码。任何收到的错误的检测域或是没有定义的值，都被等效为错误，因此，包的剩余部分将会被接收者所抛弃。

如果一个功能收到了它不支持的数据处理功能或传输方式的要求，功能对其不作响应。举例来说：只用于输入的端点必须忽略输出标志。PID 类型、编码和描述符如表 8-1 所示。

表 8-1　　PID 类型、编码和描述符

PID 类型	PID 标志	PID<3:0>*	描述
标志	OUT	0001B	主机到功能的传送中，表示地址加上端点的序号
	IN	1001B	功能到主机的传送中，表示地址加上端点的序号
	SOF	0101B	帧开始标志和帧序号
	SETUP	1101B	对于控制管道的 SETUP 事务，表示地址加端点的序号
数据	DATA0	0011B	奇数的数据包 PID
	DATA1	1011B	偶数的数据包 PID
	DATA2	0111B	高速高带宽的同步传送的一个微帧中的数据包 PID
	MDATA	1111B	高带宽的用于高速数据分割的同步传送事务中数据包的 PID
握手信号	ACK	0010B	接收器接收无错误的数据包

	NAK	1010B	接收器不能够接收数据或是发送器不能够发送数据

续表

PID 类型	PID 标志	PID<3:0>*	描述
握手信号	STALL	1110B	端点被暂停或是不被支持的控制管道请求
	NYET	0110B	仍未收到从接收器来的响应
特殊	PRE	1100B	主机发出的前导标志，使低速的设备能够开始下游总线事务处理
	ERR	1100B	用于握手的分割传送事务出错
	SPLIT	1000B	高速的分割传送事务标识
	PING	0100B	用于批量/控制传送的高速流量控制探测
	保留	0000B	保留的 PID

注意：PID 的位是按 MSb 的顺序来显示的，当送往 USB 时，最右边的位（bit 0）被最先送出。

PID 被分为 4 个码段区域：标志、数据、握手和特殊。它是用转送的前两个位（PID<0、1>）来表示是属于哪一组的。

8.3.2 地址域

功能端点是使用两个域来进行定位的：功能地址域和端点域。一个功能需要对两者都进行全部的解码。而且两者之中的任何一个都不允许有混淆现象发生，任何一个域出错都会使整个标志被忽略，而对未初始化的端点进行访问也会引起标志被忽略。

1. 功能地址域

功能地址域（ADDR）通过其地址来规定了某功能是数据流的源还是目的。如图 8-2 所示，功能地址域最多可以定位 128 个地址。通过定义，每一个 ADDR 的值都定义了一个单一的功能。在设备重启或上电之初，功能的地址被默认为地址“0”，但在随后的过程中必须由主机在列举的进程中进行重新的定位。

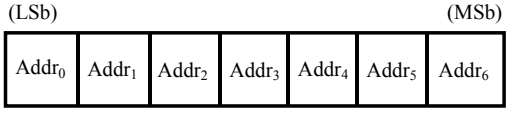


图 8-2 功能地址域

功能地址“0”总是用于默认地址，而且不能用于其他用途。

2. 端点域

一个附加的四位的端点域（ENDP）（如图 8-3 所示），使得那些不止拥有一个端点的功能的定位更加灵活。除了端点地址零，端点地址的数量都是由功能确

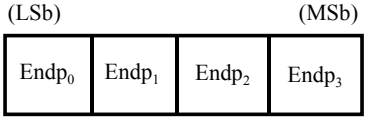


图 8-3 端点域

定的。这一端点域被定义用于 IN、OUT、SETUP 和 PING 的标志。所有的功能都必须支持一个位于端点“0”的控制管道（默认控制管道）。低速的设备的每个功能最多支持 3 个管道：一个控制管道（位于端点 0）和两个附加管道（或是两个控制管道，或是一个控制管道和一个中断端点，或是两个中断端点）。全速和高速功能最大可支持 16 个输入和输出端点。

8.3.3　帧数量域

帧数量域是由主机按每一个帧来进行增加的 11 位的域。帧序号域在达到其计数的最大值 7FFH 后便进行翻转，而且在每一个帧（微帧）的开始时在 SOF 中送出。

8.3.4　数据域

数据域可以是 0~1024 个字节，而且必须是一个取整数字节。多重字节的格式如图 8-4 所示。数据字节是从 LSB 开始进行传送的，数据包的大小根据不同的传输类型会有所不同。

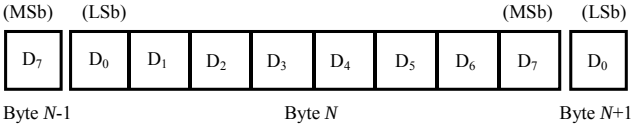


图 8-4　端点数据域

8.3.5　循环冗余校验

循环冗余校验（CRCS）是用来保护所有的在标志和数据包中的非 PID 标志域的。在这一章中这些非标志域的部分都被认为应该是受保护的，而 PID 标志域则不必接受 CRC 校验。所有的 CRC 是在执行位填充之前由收发器在其各自独立的域中完成的。同样，在接收器端，在填充位被去掉之前，CRC 被解码。标志和数据包的 CRC 能够对所有的一位或两位的错误进行恢复。而如果 CRC 恢复失败，那么说明有多个错误在数据传送的过程中发生，这便会使接收器忽略这一个域甚至丢弃整个数据包。

在发生器和检验器中都是用同一方案对 CRC 进行发生和检验的。当校验域的最后一位校验完成后，发生器的 CRC 值便进行翻转，并且按 MSb 的顺序送到接受器。当 CRC 中的最后一位被接收器收到后，在没有错误的情况下，计数器中剩余的值应该与预先设定的值相同。

8.4　数据包格式

这里要讨论标志、数据和握手包的格式，下面会以数据包中各个域移入总线的顺序来讨论包中的各个域。

8.4.1 标志包

标志包中各个域的格式如图 8-5 所示。一个标志包由 PID、ADDR 和 ENDP 域构成，用于确定数据包的类型：IN、OUT 或者 SETUP。PING 数据包同样含有类似的标识域。在 OUT 和 SETUP 类的事务处理中，地址和端点域用来确定要接收相应数据的端点；对于 IN 事务，则要根据数据包确定相应的发送端点。对于 PING 事务，这两个域则用来确定要用哪个端点发送响应的握手信号。

只有主机才能发出标志包。IN PID 定义了一个从功能到主机的数据事务处理，OUT 和 SETUP PID 则定义了一个从主机到功能的事务处理，而 PING PID 定义了从功能到主机的握手处理。

由图 8-5 可知，标志包中有 5 个 CRCS 位用于填充地址和端点域。CRC 不能覆盖 PID 部分，因为 PID 有其自身的校验域。

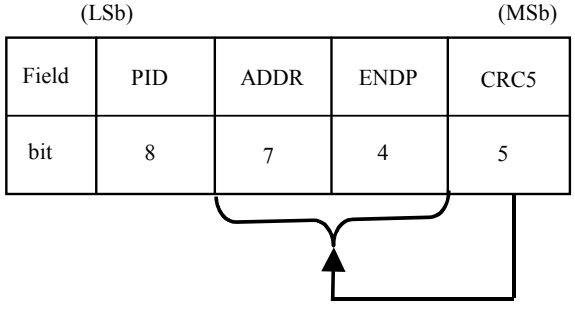


图 8-5　标志域的格式

8.4.2 分割处理特殊标志包

USB 为事务的分割定义了一个特殊的标志：SPLIT。与其他的 3 个字节的标志包不同，这是一个 4 字节标志包。它通过附加的特殊事务处理信息提供了特殊事务处理类型。

分割事务处理标志用于在高速模式工作的集线器中的主机控制器和全/低速设备之间通信时的事务的分割处理。

利用 SPLIT 特殊标志定义了两个分割过程：事务分割的开始（SSPLIT）和事务分割的结束（CSPLIT）。

1. 分割处理

高速的分割处理只用于主机和集线器间，而且必须是集线器下游端口连入了全/低速设备的情况。高速的分割处理用于初始化全/低速的事务处理。它同样使主机能够从集线器得到全/低速事务处理完毕的状态标志。这一步骤使主机控制器在开始全/低速事务处理之后，能够不用等待全/低速事务处理的结束转而处理另外一个高速的事务。

高速的事务分割处理包括两部分：Start-split 和 Complete-split。图 8-6 所示为一个普通的开始分割事务（Start-split）的包的组成。

首先在标志阶段有两个数据包被送出：SPLIT 包和全/低速标志包。标志包后是一个可选的数据包或握手包。因此，开始分割事务处理部分就可能由 2、3、4 个包构成，如图 8-6 所示。

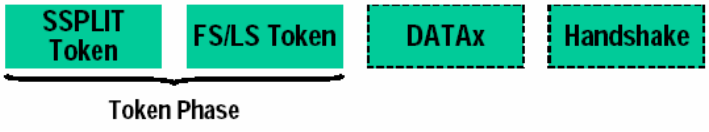


图 8-6　开始分割事务中的数据包

结束分割（Complete-split）事务包的构成如图 8-7 所示。同样，它会有一个标志阶段，由 SPLIT 标志和全/低速标志构成，其后是数据或握手包。结束分割阶段可由 2 个或 3 个包构成。

图 8-7　结束分割事务包的构成

事务分割的结果由结束分割事务负责返回。下面是一个以输入中断传输为例的事务分割处理过程，如图 8-8 所示。

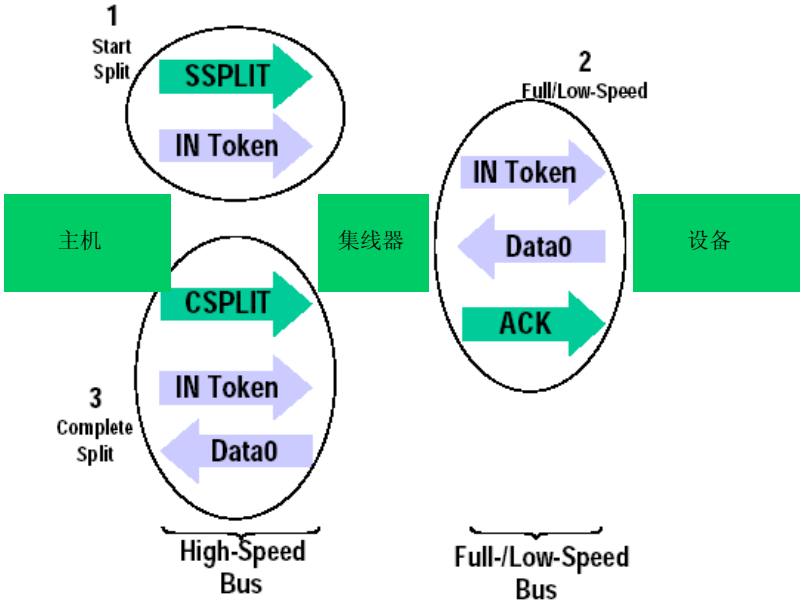


图 8-8　中断输入事务和高速分割之间的关系

主机向集线器发出一个开始分割信号，然后转入其他的高速事务处理（图 8-8 所示的第一部分）。开始分割的信号引起集线器经过一段时间后发出全/低速的输入（IN）标志（图 8-8 所示的第二部分）。设备通过数据包响应 IN 标志，然后集线器用握手信号响应设备。最后，主机在一段时间过后发出结束分割（图 8-8 的第三部分）请求，来取回设备提供的数据。

注意：在这一例子中，集线器在分割完成之前向设备的端点提供了全/低速的握手信号（例中为 ACK）。然而，分割完成后设备并不会向集线器提供高速的握手信号。

下面来仔细探讨一下“开始”和“结束”分割标志包的构成。

2. 开始分割事务标志

开始分割标志的构成如图 8-9 所示。

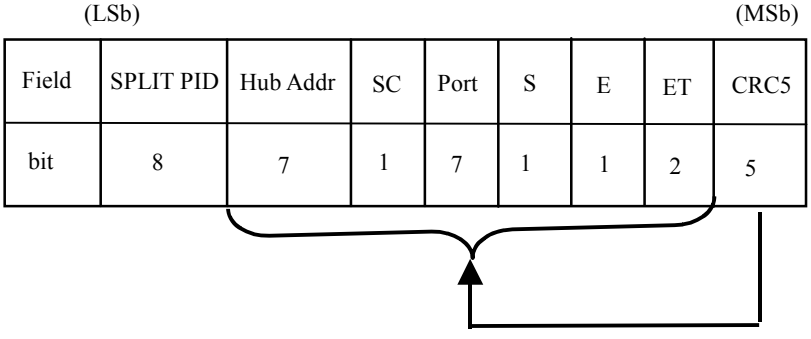


图 8-9　开始分割标志（SSPLIT）

“Hub Addr”——集线器地址域包含了集线器支持的全/低速设备地址。SPLIT 标志中 SC（Start-split）域为 0 时，暗示了这是一个开始分割的事务处理。

“Port”——端口域定义了全低速事务处理中目标集线器的端口数量。由图 8-10 可知，总共可以定义 128 个端口。对于单重或多重 TT 的集线器，主机必须正确地设定端口域，一个单一 TT 的集线器可能忽略端口域。

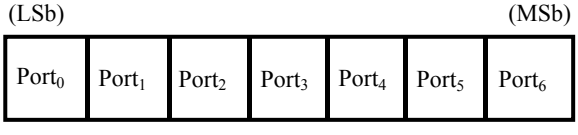


图 8-10　端口域

“S”——速度（Speed）位定义了中断或控制传输的设备速度。

- 0：全速。
- 1：低速。

对于批量的 IN/OUT 和同步的 IN 事务的分割，S 位必须被置“0”；对于批量或是控制的 IN/OUT，中断的 IN/OUT 和同步的 IN 的分割事务，E 位必须被置为“0”；对于全速的同步输出的开始分割，S 位和 E 位确定了高速的数据载荷怎样对一个全速的数据包做出响应，如表 8-2 所示。

表 8-2　　同步输出载荷的连续解码

S	E	高速到全速的数据转化
0	0	高速数据是全速数据载荷的中部分

0	1	高速数据是全速数据载荷的末尾部分
1	0	高速数据为全速数据载荷的开始部分
1	1	高速数据是全部的全速数据载荷

同步输出的开始分割事务利用这样的译码方式来使得集线器对于各种错误可以作出检测。举例来说，一个全速的数据的有效载荷可能需要被分割为 3 个部分：数据首部、数据中间和数据尾部。那么，如果在接收器一端通过译码发现任何一部分的分割出错或数据被丢失，那么它会将整个数据包丢弃或是发出错误的警告。

“ET”——端点类型（Endpoint Type）域规定了全/低速传输的端点类型，如表 8-3 所示。

表 8-3　　分割处理标志中的端点类型

ET 值（Msb：Lsb）	端点类型
0　0	控制
0　1	同步
1　0	批量
1　1	中断

这个域的值便告诉了集线器应该利用哪一个状态机来处理全/低速的事务范动作。全/低速的设备地址和端点序号在 SPLIT 标志包的正常标志包中会给出说明。

3. 完成分割处理标志

完成分割标志的示意图如图 8-11 所示，将 SPLIT 中的 SC 位置为“1”，便暗示了这是一个完成分割标志（CSPLIT）。保留位 U 应被置为“0”，其中，其他位的定义与开始分割标志相似。

(LSb)　　(MSb)

Field	SPLIT PID	Hub Addr	SC	Port	S	U	ET	CRC5
bit	8	7	1	7	1	1	2	5

图 8-11　完成分割处理标志

8.4.3　帧开始（Start-Of-Frame）包

SOF 包是由主机在全速下以 1ms±0.0005ms 或在高速下以 125μs±0.0625μs 的速率发出的。SOF 包中包含 8 位的 PID，11 位的帧计数位，如图 8-12 所示。

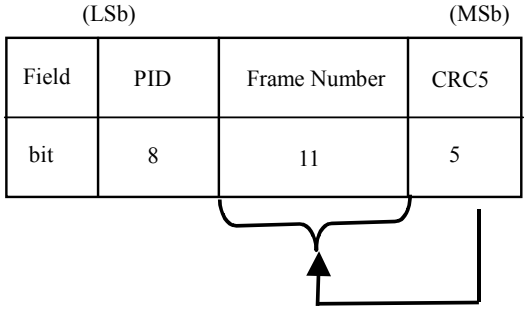


图 8-12　SOF 包

所有的高速和全速的功能，包括集线器都要接收 SOF 包，但对于 SOF 标志，所有的功能都不会给出反馈信号，也就是说，SOF 对于任一功能所作出的规定不一定会得到保证。

SOF 中包含了两个信息：一是帧到达时间，这是通过功能对 SOF PID 的探测得到的；另一部分是帧数，那些对时间敏感的功能设备不必理会这一部分（例如全速集线器），它们只需对 SOF PID 进行解码即可。

关于帧和微帧，USB 定义了包含抖动误差的由 SOF 开始的以 1ms 为间隔的帧，同样定义了包含抖动误差的，由 SOF 开始的，以 125μs 为间隔的微帧。它们之间的对比关系如图 8-13 所示。

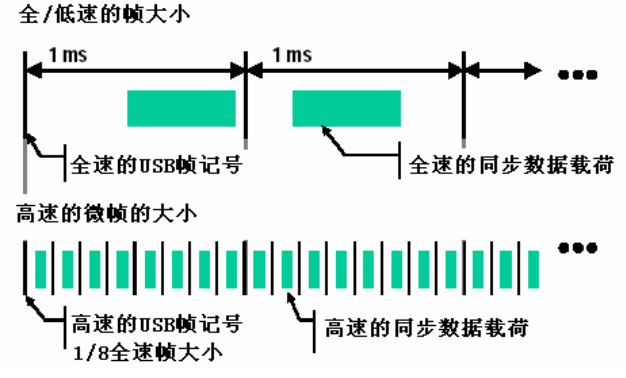


图 8-13　帧和微帧的关系

但 SOF 数据包总是以 1ms 的间隔发出。也就是说，高速的设备在每隔 8 个微帧才能“看到”一次 SOF。

8.4.4　数据包

数据包由 PID、数据域和 CRC 构成，如图 8-14 所示。

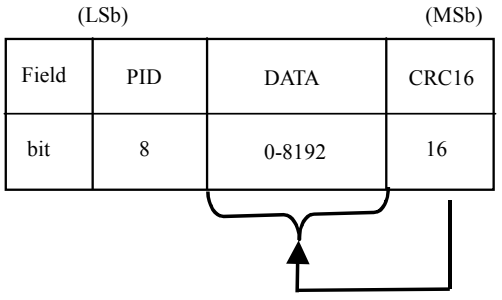


图 8-14　数据包的格式

总共有 4 种类型的数据包 PID：DATA0、DATA1、DATA2 和 MDATA。两种数据包 PID（DATA0、DATA1）用于定义支持数据帧的同步触发。对于高速高带宽的同步端点来说，这 4 种 PID 都可用于数据包的排序。3 种 PID（MDATA、DATA0、DATA1）被用于事务分割。

数据必须以整数的字节送出。数据的 CRC 只是对数据域部分的校验，而不包括 PID。PID 有其自身的校验方法。

低速设备最大有效数据载荷为 8 个字节，全速为 1023 个字节，高速为 1024 个字节。

8.4.5　握手包

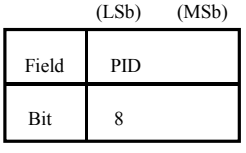


图 8-15　握手包

握手包只包含了一个 PID，如图 8-15 所示。

它被用来报告数据处理的状态，而且通过返回的数值确定数据是否被成功接收，命令的接收与拒绝，流量的控制以及环境暂停等。只有支持流量控制类型的事务处理才能够返回握手包。握手包总是在数据处理的握手阶段返回，同时也有可能在事务处理的数据阶段代替数据包而返回。

握手包是在它的数据域结束后，以 EOP 标志将数据包分隔出来的。如果握手包在被解码后，主机（或设备）未能在它的后面收到 EOP 信息，那么，该握手包的信息则被认为是无效的。

下面具体看看每一种握手信息的含义。

- ACK：暗示数据在位填充和 CRC 中没有错误，而且数据的 PID 被正确地接收。它应用于希望有握手信息包返回的数据传输中。ACK 可以在 IN 事务中由主机输出，在 OUT、SETUP 和 PING 事务中由设备发出。
- NAK：暗示不能接收从主机来的事务（OUT）或是设备没有数据要发送（IN）。NAK 在 IN 事务的数据阶段和 OUT、PING 事务的握手阶段被发出。主机永远不发送 NAK，它意味着设备对数据输入、输出暂时不能响应，但在一定时间间隔后可能会接收该事务的请求。
- STALL：是设备功能对 IN 标志、OUT 的数据阶段或者 PING 处理的响应。它暗示了功能不能发送或接收数据或者控制管道的请求不被支持。发送完 STALL 后，设备的状态是不确定的。主机在任何时候都不会返回 STALL。
- NYET：这是一个只有在高速的两种情况下才有的握手信号。

其一，被高速端点作为 PING 事务中的一部分发出 NYET。

其二，集线器对于事物分割处理的响应，它表示分割未完成或是不能进行分割。

❑ ERR：这也是一个高速模式下才有的握手信号。它是高速集线器对低/全速总线错误的报告。他只能被高速的集线器作为事物分割处理协议的一部分送出。

8.4.6 握手的响应

执行发送和接收的功能必须按照表 8-4~表 8-6 的内容返回握手信号，但并不是所有的握手信号都可以随意发送，这要依据不同的传输类型或是发送者的不同来决定。值得注意的是，如果在向功能传送标识的过程中发生了错误，功能不会对任何数据包作出响应，直到下一个标志数据包到达为止。

1. 对于 IN 事务功能的响应

表 8-4 给出了可能的对于 IN 标志的响应。如果是由于数据传送的暂停或流量限制的原因，或是功能根本不能发送数据，那么设备将会发出 STALL 或是 NAK 信号。如果接收标志失效，那么功能不会作出任何反应。

表 8-4　　功能对于 IN 事务的响应

接受标志是否失效	功能端点是否暂停	功能能否接受事务	采取的动作
是	无关	无关	不作任何响应
否	是	无关	STALL
否	否	否	NAK
否	否	是	发送数据

2. 主机对于 IN 事务的响应

表 8-5 给出了主机对于 IN 事务的响应。主机只能发送一种标志：ACK。如果主机不能接收数据或是收到无效的数据，那么主机将不作任何反应。只有在正确无误地收到数据后，主机才发出 ACK。

表 8-5　　主机对于 IN 事务的响应

数据包是否失效	主机能否接收数据	主机返回的信号
是	-	丢弃数据，不响应
否	否	丢弃数据，不响应
否	是	ACK

3. 功能对于输出事务的响应

在标志被成功解码的情况下，根据接收到的数据包的不同，功能会返回 3 种握手信号，如表 8-6 所示。

表 8-6　　功能对于 OUT 事务的响应

数据包是否失效	接收器停止位的特征	是否有顺序匹配位	功能能否接收数据	握手信号
是	-	-	-	无
否	被置位	-	-	STALL
否	未被置位	无	-	ACK
否	未被置位	有	是	ACK
否	未被置位	有	否	NAK

4. 功能对于 SETUP 事务的响应

SETUP 是一种主机到功能的特殊的事务处理，它用于初始化端点的同步位。如果有 SETUP 标志，功能必须无条件地接收数据。对非控制端点，则忽略收到的 SETUP 标志，不作任何响应。

8.5　数据包的处理时序

依据不同的端点类型，数据包的处理过程各不相同。在前面章节中已经知道共有 4 种端点类型，即批量、控制、中断和同步。

根据不同的处理类型，主机控制器和设备都分别需要不同的状态机，主机控制器和设备的状态机的关系如图 8-16 所示。它们两者可能会根据上游的数据包来作出响应。主机控制器的状态机可以从总线上接收数据包，然后提供结果给主机控制器。向总线送出什么样的数据包的细节取决于端点类型和状态机的状态。

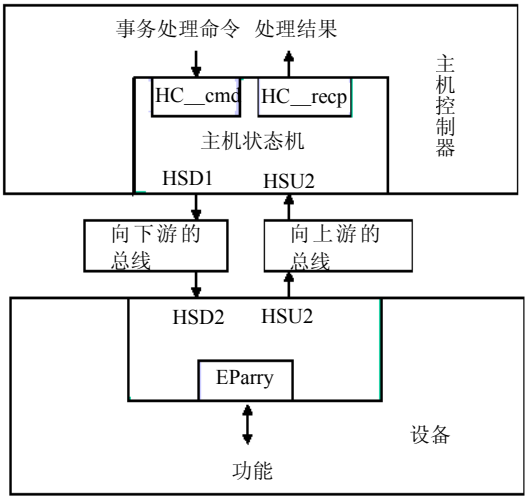


图 8-16　主机和设备状态机

状态机用等级框图的方式显示，结果如图 8-17 所示。图中给出了主机控制器的顶层状态机，即事务分割的状态面。

主机控制器发送命令来决定一个端点的下一个处理状态，主机控制器对几个端点同时进行跟踪。同时，主机控制器的状态机时序决定了主机控制器下一步应该采取的动作，同理，设备状态机时序决定了设备应该采取的动作。

主机控制器对于非分割事务类型的整体状态机的层次如图 8-18 所示。设备状态面的层次如图 8-19 所示。

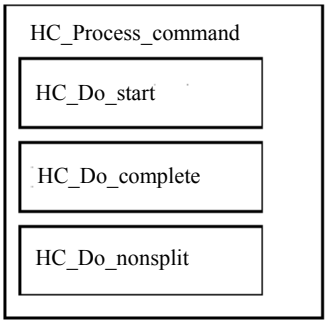


图 8-17　主机控制器的顶层状态机

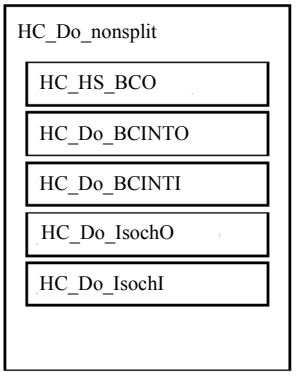


图 8-18　主机控制器对于非分割事务类型的整体状态机的层次

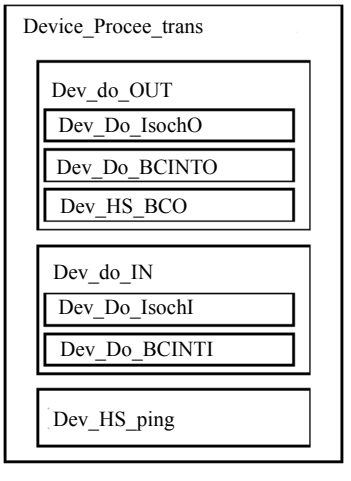


图 8-19　设备状态面的层次

可以看出状态机中首先出现的是关于设备端点类型的命令。

8.5.1　批量传输的处理时序

批量传输的特征便是有能力对主机和功能之间发生的数据的分发的错误进行检测，如图 8-20 所示，批量传输包括标志数据和握手包的 3 个处理阶段。

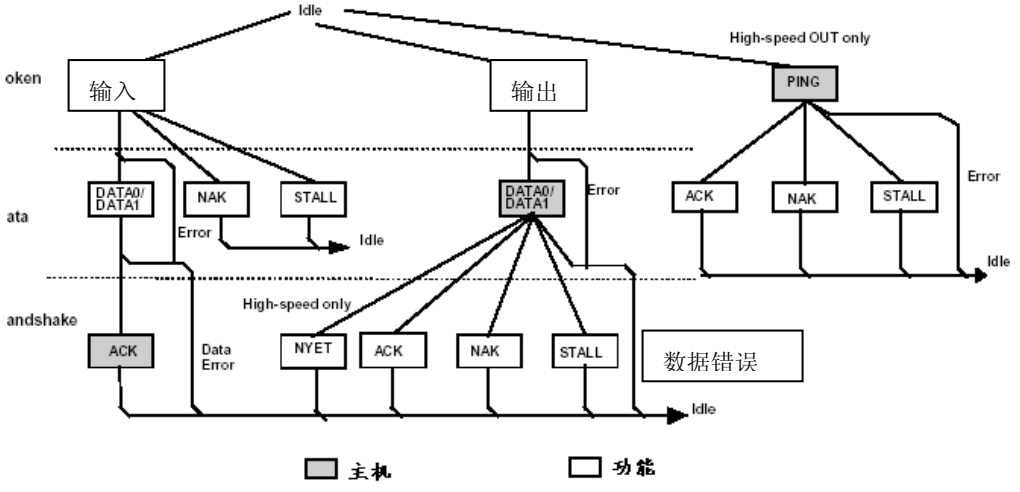


图 8-20　批量传输的格式

基于确定的流量控制和暂停环境，数据阶段可能被一个握手结果所代替，而这一结果是基于其他两个没有数据传输的阶段的处理来决定的。用于批量读/写的时序位和数据 PID 如图 8-21 所示。数据包通过使用数据时序触发和 DATA0/DATA1 的 PID 来实现同步化。

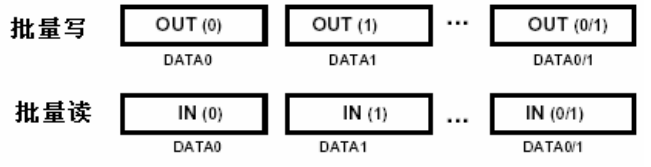


图 8-21　批量的读/写

当一个批量端点经过了配置之后，其数据触发时序将会以 DATA0 进行初始化。

主机总是通过一个配置事件来对总线上的第一笔传输事务进行 DATA0 PID 的初始化。第二笔事务则使用 DATA1 PID，并且以此类推对剩余的数据接收来进行锁定，而接收器则是依据有效数据包的接收来锁定的。

8.5.2　控制传输处理时序

控制传输包含两个阶段：设置和状态。同时，它还允许有一个可选的数据阶段，位于前两个阶段之间。在设置阶段，设置处理被用于向功能的控制端点传送信息。设置阶段类似于

输出传输，但使用的 PID 不是 OUT PID 而是 SETUP。设置事务处理的格式如图 8-22 所示。

对于设置阶段的数据包，总是以 DATA0 PID 作为标识的。对于接受设置的功能同样必须要接受 SETUP 的数据，并以 ACK 进行响应。如果数据被毁坏，那么将丢弃数据并不作任何响应。

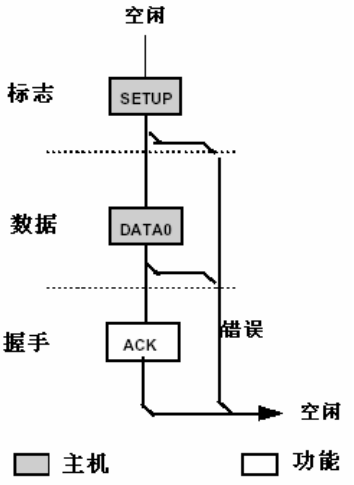


图 8-22　控制传输的设置阶段

如果有数据阶段的话，控制传输必然要包括一个或多个输入或输出的处理事务，并且同样遵循批量传输的协议。所有的数据包必须是同一方向的（即全为输出或全为输入），而数据包的数量和方向是由设置阶段所规定的。如果数据包的大小超出了最大包限制，那么它将会被以多重数据包的形式发出。

控制传输的状态阶段是最后一部分时序处理阶段。状态阶段的事务处理遵循与批量处理同样的时序协议。对于高速的状态阶段同样包含 PING 协议。状态阶段通过对数据流方向的改变来作出提示，并使用 DATA1 PID。例如，对于一个由输出组成的数据阶段，状态阶段是一个单独的输入处理。如控制时序没有数据阶段，那么它便由设置阶段和输入处理的状态阶段所构成。

处理的顺序、数据时序位的值以及数据 PID 类型如图 8-23 所示。

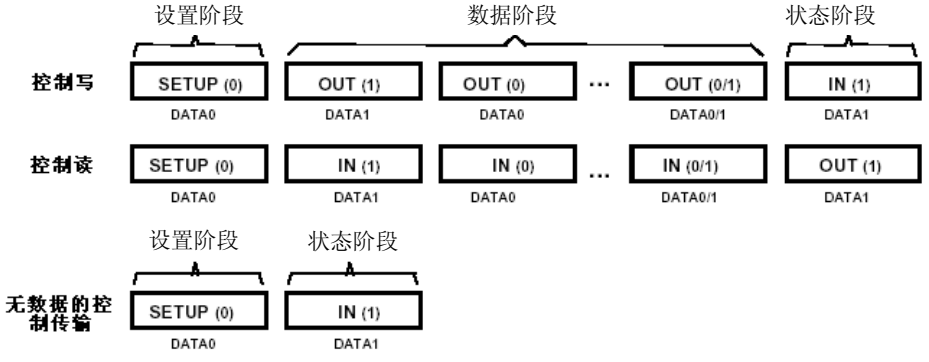


图 8-23　控制读/写时序

控制传输的数据或状态阶段由端点送出一个 STALL 握手信号时，在 SETUP PID 被收到之前，所有对端点的访问都将以 STALL 回应。对于默认的端点来说，如果设备对于 SETUP 处理返回的是 ACK 握手信号，那么主机便等候端点从 STALL 环境中脱离，并转入正常工作状态。

1. 报告状态结果

在状态阶段将会报告给主机前一个设置和数据阶段的传输的结果。可能有 3 种结果被返回：

- 命令时序成功地完成。
- 命令执行失败。

- 功能仍在执行命令。

状态报告的方向总是从功能到主机的。设备对于每种响应的需求如表 8-7 所示。控制的“写”传输在状态阶段的数据部分返回状态信息。控制的读传输在状态阶段的握手部分返回状态信息，但是要在前一数据阶段主机发送完一个“零”长度的数据包之后才能进行。

表 8-7　　状态阶段的响应

状 态 响 应	控制读传输（在数据阶段送出）	控制写传输（在握手阶段送出）
功能完成	零长度的数据包	ACK
功能出错	STALL	STALL
功能忙	NAK	NAK

对于控制的读传输，主机必须送出一个 OUT 或 PING 标志（高速模式）到用于初始化状态阶段的控制管道。对于主机在这一段发出的数据包，功能设备都应将数据作为有效的状态请求来回应。管道对于这一数据包的握手响应暗示了当前的状态。

- NAK：功能仍在处理命令，主机应当继续状态阶段。
- ACK：功能已经完成了命令，准备接受一个新命令。
- STALL：功能出错，阻止了它执行命令。

对于控制的写传输，主机送出了一个 IN 标志到最初的状态阶段的控制管道。功能以握手信号或“零”长度的数据包来示意当前状态。

- NAK：功能仍在执行命令，主机应该持续状态阶段，零长度数据包，命令执行正常完毕。
- STALL：功能不能执行该命令。

功能希望主机在状态阶段用 ACK 来响应数据包。如果功能不能接收 ACK，那么它将保持在命令的状态阶段，而且只要主机发出 IN 标志，将会持续返回零长度的数据包。如果在数据阶段送出的数据比在设置阶段要求返回的数据要多，那么包将以 STALL 作为返回信号。如果控制管道在数据阶段返回 STALL 信号，那么该控制传输将不会有状态阶段。

2. 不同长度的数据阶段

一个控制管道可能拥有不同长度的数据阶段，在这一阶段，主机请求的数据量可能多于规定的数据结构中所能包含的数据量。当所有的数据结构都被返回主机后，功能应该通过返回比 MaxPacketSize 小的数据包来暗示传输已经完成。如果数据正好是 MaxPacketSize 的整数倍，那么，功能将会返回一个零长度的数据包来暗示数据阶段的结束。

3. 关于最后一次数据处理的错误检测

如果对输入数据处理的 ACK 握手信号失效，功能和主机会在处理事务是否成功这一点上产生暂时的不一致；如果紧随其后的也是一个输入事务，那么数据触发的重试机制将会对错误进行检测和恢复；如果失效发生在最后一次输入的数据阶段，数据触发就起不到作用，必须引入另外的检测机制。

主机如果成功地接收到输入数据，它会以 ACK 来响应，然后它将发出 OUT 标志进入传输的状态阶段。如果功能在数据阶段末尾没有接收到 ACK，功能将会对状态阶段的开始进行

译码来查证主机是否成功地接收到了数据。

对于控制的“写”操作来说情况则不同。一旦用于 OUT 操作的 ACK 信号失效，那么主机便会进行数据的重发，而不进入下一个阶段。

4. 由控制管道返回的延迟（STALL）握手信号

在控制传输下由于功能的问题控制管道具有独一无二的返回 STALL 握手信号的能力。如果设备不能够完成一个命令，那么它将在控制传输的数据或是状态阶段返回 STALL 信息。

与功能性的延迟不同，协议上的延迟并不意味着设备出错。协议性暂停的环境将会一直持续到下一个 SETUP 事务的到来，并且功能在这一期间会一直以 STALL 回应 IN/OUT 请求。通常情况下，协议的延迟暗示着请求或其参量不能被设备所理解，因此可以看到，在 USB 的协议中提供了一个专用的扩展的 USB 请求。

控制管道可能同样支持功能性的延迟，但这样的方式并不被推荐。这是一种退化的情况，因为一个控制管道上功能上的延迟（暂停）暗示着设备已失去了与主机进行通信的能力。如果控制管道支持功能性的延迟，那么它必须拥有暂停的特征，这一特征位将会由主机置位或清除。

8.5.3　中断传输的处理时序

中断处理可能由输入或输入传输构成。依据收到的 IN 标志，功能可能返回数据 NAK 或 STALL 信号。如果端点没有新的中断信息需要返回（即没有中断等候处理），功能将会在数据阶段返回 STALL 信号。如果有中断等候处理，功能将中断信息作为一个数据包返回。主机根据接收到的数据包发出不同的响应信号。如果数据被无错误地收到，那么主机以 ACK 响应；如果数据包出错，则不发出任何握手信号，中断处理的格式如图 8-24 所示。

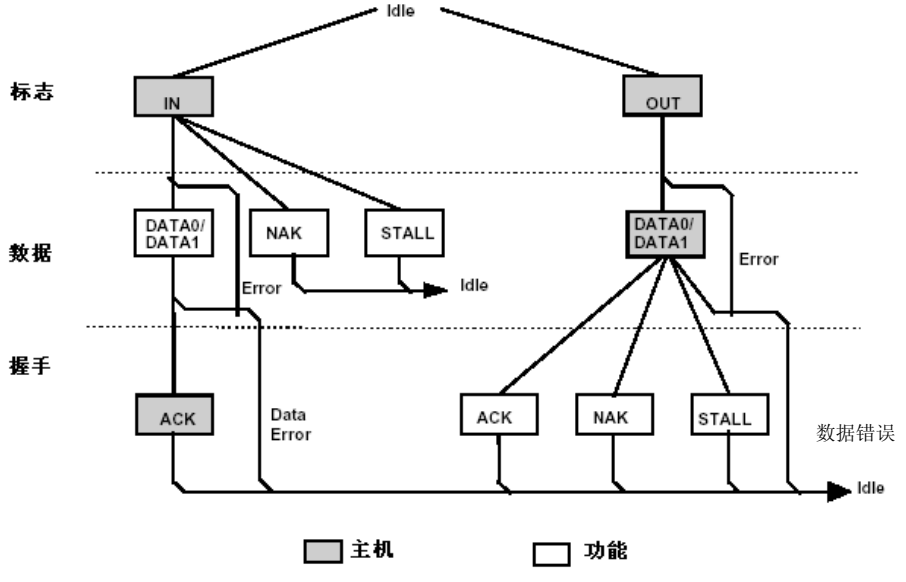


图 8-24　中断传输的处理格式

当在一个端点上将中断传输机制用于数据中断情况时，必须遵守数据触发的协议。这一协议使得功能知道数据已经被主机收到而且事件环境应当被清除。这种以“保证”的方式对事件的处理，允许功能在主机接收到信息之前发送中断信息，而不是每次设备功能被询问到时才必须发出中断信息，持续到 USB 的系统软件将中断环境清除为止。在使用数据触发的模

式时，中断端点没有配置事件发生时，便被初始分为 DATA0 PID 的模式。这一点类似于图 8-21 所示的批量事务处理的情况。

8.5.4 同步传输的处理时序

同步处理拥有标志和数据阶段，但没有握手阶段，如图 8-25 所示。在主机发出 IN 或 OUT 标志后，端点（输入时）或主机（输出时）便在紧随其后的数据阶段进行数据的传送。同步处理机制并不支持握手阶段或是重试的能力。

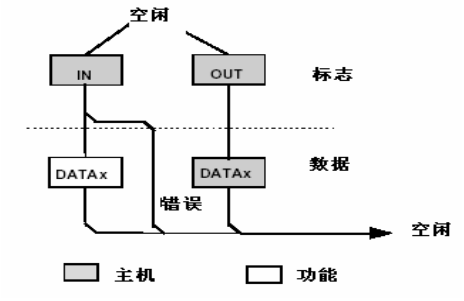


图 8-25 同步处理的数据格式

全速的同步处理机制也不支持数据触发时序，高速的同步处理在每一微帧中只有一种单独的同步处理支持数据 PID 的时序处理。

8.6 数据触发的同步和重试

USB 提供了数据时序在数据的发送者和接收者经过多重事务处理之后还能保持同步的一种机制。这种机制保证了对数据处理握手阶段，发送者和接收者都能够正确译码。同步是通过使用 DATA0 和 DATA1 PID 以及独立的数据触发时序位来得到的。只有在接收器有能力接收数据而且接收到的是带有正确的数据 PID 的无错误的数据的情况下，接收器的时序位才能被触发。而对于发送者来说，只有在发送者接收到了一个有效的握手信号之后，其数据时序位才能被触发。数据的发送者和接收者必须在数据处理开始之时保证其时序位取得同步。同步机制随着处理类型的不同而不同。高速、高带宽的同步和中断端点支持相似的功能，但在数据同步方面不同，它称为数据 PID 时序。这样的机制代替了数据触发的同步。

8.6.1 通过 SETUP 标志进行的初始化

控制传输使用 SETUP 标志来初始化主机和功能的时序位。图 8-26 所示为主机发送一 SETUP 数据包到某一功能，然后紧接着是一个输出的事务处理。

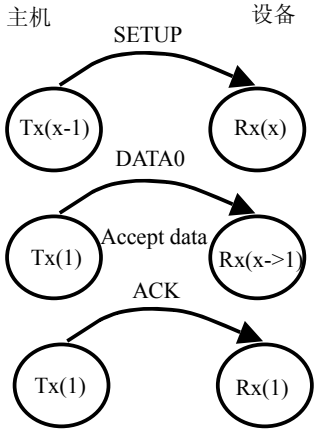


图 8-26　SETUP 的初始化

在圆环中的数字代表着发送器和接收器的时序位。功能必须接受数据，然后返回 ACK 信号。当功能接受处理时，它必须将其时序位置位，这样便能使主机和功能的时序位在 SETUP 事务处理的末尾都为“1”。

8.6.2　成功的数据处理

两种成功的数据处理的情况如图 8-27 所示。对于数据的发送者来说，这意味着它一旦收到 ACK 信号便要触发其时序位；而接收器只有在其收到了有效的数据包，而且数据包的 PID 能够与当前的时序位相匹配时它才会触发其时序位。

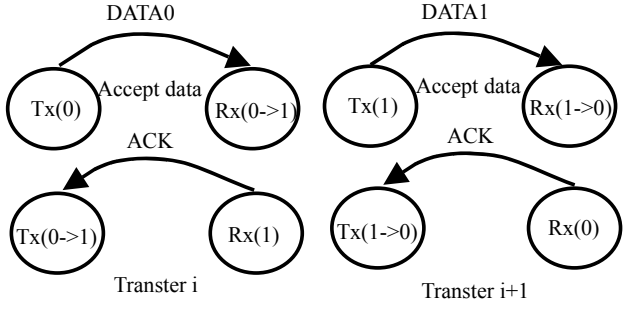


图 8-27　成功的连续的事务处理

在每一个时序处理的过程中，接收器将其自身的时序位与发送器的时序位相比较（对数据包 PID 进行解码，看其结果为 DATA0 还是 DATA1）。如果数据不能被接收，接收器必须发送 NAK 信号，而且发送器和接收器的时序位都保持不变。如果数据能被接收而且接收器的时序位与 PID 的时序位相匹配，那么数据便会被接收，而且时序位也会被触发。

8.6.3 数据的失效或不能被接收

如果数据不能被接收或是收到的数据包已失效，那么接收器将会发出一个 NAK、STALL 或是超时信号，具体取决于环境的设定，而且接收器将不会触发其时序位。

事务处理被以 NAK 信号回应，而且进行重试的情况如图 8-28 所示。任何非 ACK 的握手信号或是超时信号都会产生相似的重试的行为。发送器在未接收到 ACK 握手信号之前，是不会触发其时序位的。作为一个结果，失效的数据将使得发送器和接收器的时序位保持在同步且未触发状态。事务的处理将会被重试，如果成功，就将引起接收和发送时序位都被触发。

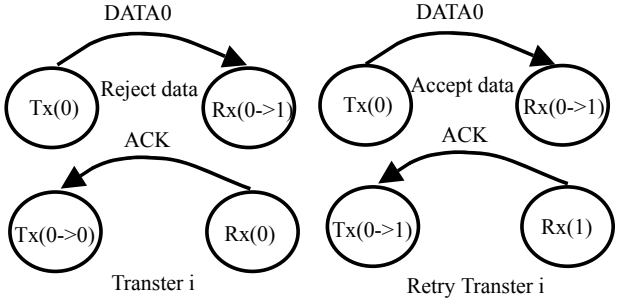


图 8-28　事务处理被 NAK 后的重试

8.6.4　失效的握手信号

发送器是数据传送成功与否的最终的判断者。如图 8-29 所示，一个 ACK 握手信号的丢失或损坏，都会导致发送器和接收器暂时的不同步。

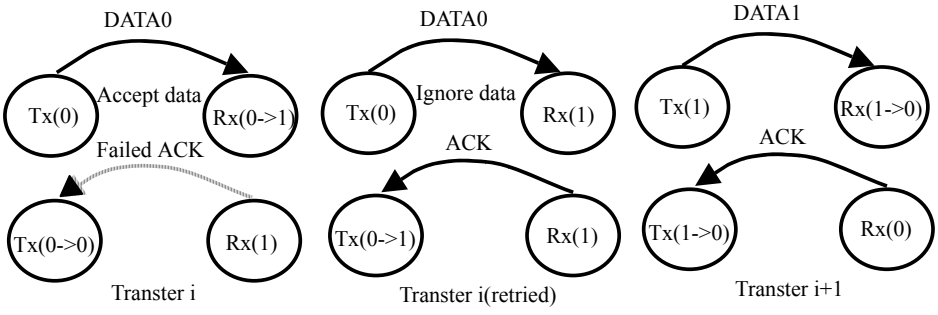


图 8-29　无效的 ACK 后的重试

由图可知，发送器发出了一个有效的数据包，而且被接收器顺利收到，然而 ACK 信号被损坏了。在这个处事过程中，发送器和接收器之间出现了暂时的不匹配，这是根据其各自的时序位所决定的。接收器收到了完整的数据，并触发它的时序位，但发送器并不知道它对数据的发送是否成功。在下一个处理过程中，发送器将会再一次地将上次的数据发出，并且使用前一个标识 DATA0 PID。接收器的时序位和再次收到的数据 PID 不同，因此，接收器便知道这是上次的数据。相应地，它将接受这次到来的数据，但不对其时序位进行触发。然后它发出 ACK，这次如能被发送器接收，那么发送器便认为重试失效，并触发自己的时序位。这样，在下一次的传送中，二者的时序位便又能保持在同步状态。

数据的发送者必须保证任何重试的数据包在属性上（同样的长度和内容）与上一次的相同。如果由于某方面的原因不能实现这一过程，那么它必须以在全/低速的情况下通过一个违规的位填充来忽略这次传送，而在高速下必须以适当的 CRC 计数来产生一个错误信号，而这一错误信号是可被接收器检测的，这样便保证了一部分数据包不被译码。

注意：发送器不应当试图使接收器端收到一个通常定义好的失效的 CRC 情况，这是因为失效的 CRC 加上损坏的数据包会被接收器作为一个好的数据包来处理。

8.6.5　低速的事务处理

从前面章节已经知道，USB 系统支持 3 种速度：高速、全速和低速，而集线器起到了将高速信号与全/低速信号隔离的作用。在全/低速的环境下，在低速的设备连入正在处理全速信号的下游端口时，集线器将屏蔽所有流向这些端口的事务。

低速的事务处理如图 8-30 所示，其中包括主机发出标志、握手及接收数据一系列过程。

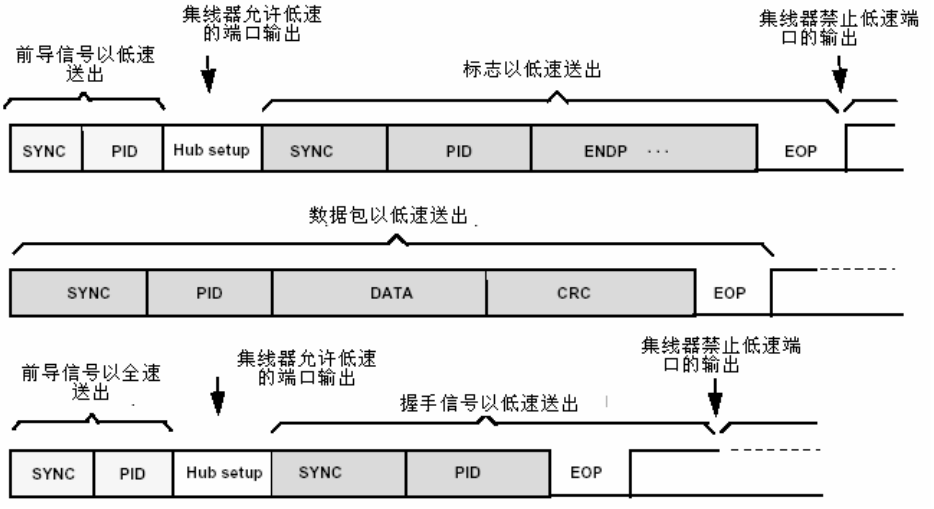


图 8-30　低速的事务处理

所有在全/低速环境下发向下游的低速数据包都需要有一前导过程，但前导从不用在高速的信号环境。前导由 SYNC 和 PREPID 组成，并都以全速送出。集线器必须能够“理解”PRE PID，而其他的 USB 设备可以忽略 PRE PID，并将其作为非定义的数据。在前导 PID 的末尾，主机将总线驱动至最少一个“位”时间的空闲状态。在这一过程之后集线器要进入设置阶段，这一阶段通常要持续 4 个全速的“位”时间。在这一过程中，集线器将使其全速和低速的端口进入其期望的空闲状态。集线器必须准备好在其设置阶段结束之前进行低速信号的重发。总的来说，低速的连接规则如下：

- 低速设备在连接过程中被鉴定，相应的集线器端口被定义为低速。
- 所有的低速的下游数据包都必须有一个前导（以全速送出），这将会开启相应端口的输出缓冲。
- 在接收到 EOP 后，低速的集线器端口缓冲关闭，直到下一个低速前导 PID 被收到才重新开启。
- 上游的连接不应该被下游端口状态影响。

低速的信号以低速的 SYNC 开始，紧随其后的是数据包。在数据包的最后以 End-Of-Packet（EOP）结束，此时，集线器将断开相应的连接，并将相应的端口屏蔽。但要注意的是集线器并不会关闭向上游的通道，也就是说，低速端口仍然能够向上游发出必要的信号。

低速和全速的事务处理与高速协议有共同点。然而，低速的信号有下列的局限性：

- 数据最大有效载荷为 8 个字节。
- 只有中断和控制传输被支持。
- 低速设备并不接收 SOF 包。

8.7　错误检测和恢复

USB 支持在物理信号层出错的情况下仍能可靠地通信的机制。这包括了能够可靠地检测

几种可能的主要的错误，并对其进行恢复的功能。举例来说，在控制传输中，需要确保数据信号高度的可靠性，而对于同步传输来说，由于其高带宽和延时的需求，便不需要重试并且必须能够对未纠正的错误保持高容错性。

8.7.1　数据包错误分类

USB 系统中使用了 3 种错误检测机制：填充位的违例、PID 校验位和 CRCS。

除了 SOF 标志，任何接收到的数据包的损坏都将引起接收器对其忽略，并且抛弃随后来的任何数据或信息。

表 8-8 列出了错误检测机制、所应用的数据类型和接收者所作的合适的反应。

表 8-8　　包错误类型

域	错误	动作
PID	PID 检测，位填充	忽略包
地址	位填充，CRC 地址校验	忽略标志
帧序号	位填充，CRC 帧序号校验	忽略帧序号域
数据	位填充，CRC 数据校验	丢弃数据

8.7.2　总线循环时间

不管是主机还是设备都不会对到达的错误包给出提示，而对于正确握手信号的丢失则会认为有错误发生。作为错误报告的结果，主机和 USB 的功能必须确定从发送器正确发出数据到接收到相应的响应数据包为止的时间。这一时间是指总线循环时间，设备和主机需要总线循环时间来测量错误发生的时间。

在全/低速的事务处理中，计时器在 EOP 由状态从 SE0 向“J”转换时开始计时，直到 SOP 从空闲态转换到“K”状态后停止计时。对于高速的事务处理，计时器在数据线转入抑制电压时开始计时，而在数据线离开抑制电压时停止计时。

设备总线的循环时间被定义为：最坏情况下的循环延时，再加上设备响应的延时。如果在最坏情况下，即计数值都计时完毕后仍未收到响应，那么发送器便认为数据包的传送失败了。

计时溢出这一策略被作为一种事务处理出错的情况用于多种传输方式。如果主机希望使用这一策略的话，那么它必须要等候整个的循环时间结束才能发送下一标志。

如图 8-31 所示，设备将循环计时器用于数据和标志之间，而主机将计时器用于数据和握手信号或是标志和数据阶段之间。

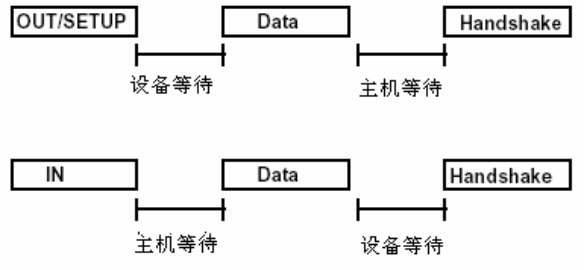


图 8-31　总线循环时序的使用

8.7.3　EOP 的失效

必须用一种合适的方法来管理 EOP，使得在 EOP 失效时能够保证下一次数据的传送必须在当前的数据处理完成之后。EOP 的失效是根据数据包中是否包含一个损坏的 CRC 来进行判断的。

主机和设备对于失效的 EOP 的处理各不相同。当设备收到一个损坏的数据包时，它不会发出任何响应并等候主机发出的下一个标志。这一机制便保证了设备不会在主机可能发送数据时试用上传握手信号。如果有失效的 EOP 出现，主机数据包将会终止，这样设备便能等到下一个标志的到来。

如果主机收到一个失效的 EOP 数据包，那么主机会忽略该包，并且同样不发送握手信号。对于设备来说，因为要等候从主机来的握手信号，因此便会使用上节所提到的计时溢出。如果主机收到了一个损坏的全/低速的数据包，那么它便假定有无效的 EOP 发生，接下来它要等候 16 个“位”时间来看看有没有相应的流向上游的数据处理。如果没有总线的变迁，而且总线保持在空闲状态，那么主机可能会发送下一个标志。

否则的话，主机将等候其余的全/低速的数据包的发送完毕。16 个“位”时间保证了两个环境：

- 确保设备已经发送完毕其数据包，这是因为 16 个“位”时间远远大于最坏情况下 6 个“位”时间的位填充间隔。
- 发送设备的循环计时可以计时完毕。

要注意的是计时溢出的时间间隔是与处理速度紧密相关的。在全速的事务处理中，主机等候全速的“位”时间，在低速的事务处理中，主机等候低速的“位”时间。

如果主机收到了一个高速的损坏的数据包，那么它将忽略所有数据，直到数据线转为抑制电压，然后发出相应的标志信息。在高速的事务处理中，主机并不需要等候附加的时间（超出正常的内部处理间隔时间）。

8.7.4　总线错误的恢复

USB 必须能够检测到并且恢复那些可能使得总线无限制地等候在全/低速的 EOP 状态，或是在一个微帧结束后，总线处于其他状态，而不是空闲态的情况。

- 全/低速的活动失效（LOA）的特征是，SOP 之后总线无动作，并且在帧之后无 EOP。
- 全/低速的错误活动的特征是 SOP 之后总线保持在帧结束之后的状态。
- 高速的失效错误的特征是数据线在微帧结束之后处于非抑制电压。

LOA 和错误的存在又可能使总线被死锁或出现下一帧被延时的危险。因此，必须禁止所有的该类情况的发生。作为 USB 控制连接的责任部分，集线器要能能够负责 LOP 和错误情况的检测和恢复。所有的 USB 设备的数据传送的失败，都会导致距该设备最近的集线器将该设备所在的端口屏蔽，从而阻止错误的漫延。

8.8 本章小结

在本章中介绍了 USB 的数据信号在其数据通道上的传输情况，可以看出设计者们为了更方便用户的使用，从各个方面想尽了办法来使得数据的传送既快速又可靠。因此才使得 USB 的数据传送协议比其他接口相关的规定显得更为复杂。

那么读者在掌握之一部分的内容时，就不必像前几章那样要求做到了然于胸了，更应该把它看作一个参考手册。这样的话，便不至于在学习本章时，产生茫然的感觉。

第 9 章　USB 控制器芯片

知识点：

- 串行接口引擎（Serial Interface Engine，SIE）
- USB 控制器芯片
- 微控制器内核

本章导读：

要开发一个 USB 外设，其首要工作就是选择一款合适的 USB 控制器芯片。现在市面上的 USB 控制器芯片琳琅满目，种类繁多。不同类型的芯片，在功能上有着很大的差别。有些 USB 控制器芯片功能强大，可以完成绝大部分的 USB 协议工作，从而大大减轻了用户的开发难度；有些 USB 控制器芯片功能相对简单，用户需要编写更多的固件程序来完成所需的功能。因此，用户需要根据所开发 USB 设备的功能要求，来选择一款合适的 USB 控制器芯片。本章将首先介绍一款通用 USB 控制器芯片所包含的各种组件，接下来将会详细介绍市面上常见的几种 USB 控制器芯片，并对它们的功能做一些必要的对比，以方便读者选择一个合适的 USB 控制器芯片。

9.1　USB 控制器芯片的构成

USB 协议是很复杂的，这就要求 USB 控制芯片有很高的智能以隐藏其内部的复杂性。呈献给使用者的应该是强大的功能和简单的使用，便于用户的开发。USB 控制器芯片不仅需要能够检测并且对端口的事件作出正确的响应，还需要能够存储需要发送的数据和接收到的数据。对数据的编码和解码，以及错误的检测都是靠硬件来完成的。

在设备进行通信时不同的 USB 控制芯片所做的工作不完全相同，换句话说就是需要固件程序参与的程度不同。有些控制器芯片只需要程序来访问一系列的寄存器，用来存储和读取 USB 数据；有的控制器芯片则需要程序做更多的事情，包括管理和传送描述符，设置数据的交替以及保证正确的握手包被发送等。

有的控制器芯片内置有微控制器内核，而有些则需要连接外部的微控制器来处理非 USB 事务的工作。所有的 USB 控制器都有一个或多个 USB 端口，以及传输数据用的缓存区、寄存器和其他的 I/O 接口。如果是内嵌微控制器内核的芯片，还应该包含有内置的程序和数据存储器，或者提供一个外部的通用存储器接口。

9.1.1 USB 端口

所有的 USB 控制器都应该具有 USB 端口和扩展电路，来支持与主机的通信。在控制器内的数据收发器（Transceiver）直接连接了 USB 的数据线 D+和 D-，用来接收和发送总线上的数据。数据收发器只是简单地提供数据通路，并进行信号的缓冲。数据的编码与解码以及其他的协议处理工作都是由其后的电路——串行接口引擎 SIE（Serial Interface Engine）进行的。SIE 由于集成了可以独立于微控制器而自动处理 USB 总线活动的硬件，所以简化了微控制器和 USB 之间的接口。SIE 通常用于处理事务数据的传输和接收工作。有的芯片的 SIE 实现了全部的 USB 底层协议，完全由硬件实现而不需要固件的参与。一个 USB 批量传输事务中 SIE 所做的工作如图 9-1 所示。

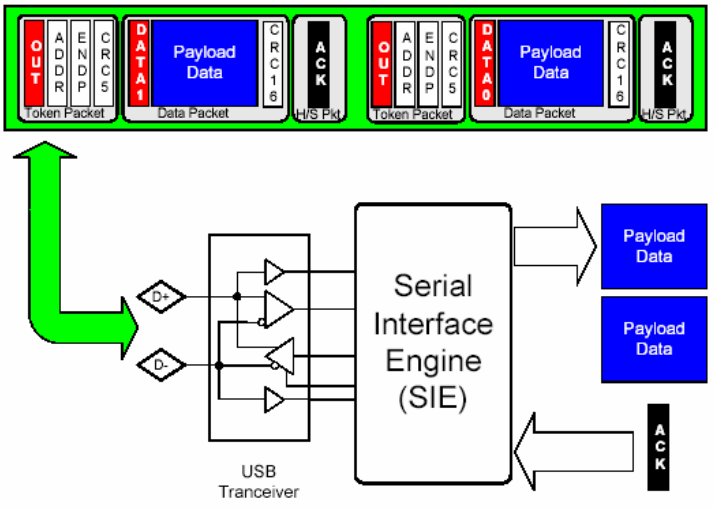


图 9-1　一个批量传输事务中 SIE 的工作

SIE 解码包的 PID，使用传输的循环冗余校验（Cyclic Redundancy Check，CRC）位执行数据的错误校验，并且将有效数据发送给 USB 设备。如果 SIE 发现数据有错，它会自动指示“不响应”来代替提供握手包中的 PID，这将会指导主机在稍候的时间里重传数据。控制传输属于异步传输，这意味着要有一个流量控制机制：SIE 会给主机发送一个 NAK 握手包以指示忙，当外设成功传输了数据，它会命令 SIE 发送一个 ACK 握手包。为了向主机发送数据，SIE 从 USB 设备接受数据字节和控制信号，格式化之后在总线上发送。因为 USB 使用的是 NRZI（Non Return to Zero Invert）编码格式，所以，SIE 还需要执行位填充的工作，并且这个工作是对用户透明的。

SIE 模块的功能一般包括：

- ❑ 翻译接收到的经过编码的数据以及格式化将要在总线上传送的数据。
- ❑ CRC 校验和产生，当传输时发生错误，可以向微控制器产生标志。
- ❑ 地址校验，忽略发送给设备的带有错误地址的事务。
- ❑ 发送适当的 ACK/NAK/STALL 握手包。
- ❑ 标准类型的鉴别，当收到有效的标准包时设置适当的标准位。

- 将收到的有效数据放入适当的端点缓冲区。
- 发送并且更新数据交替位。
- 位填充和去除填充位。

固件只需要处理余下的 USB 传输事务和其他的控制工作，这大大减轻了用户编写固件的工作量。

9.1.2 CPU

USB 控制器的 CPU 通过执行存储在芯片里的固件程序来控制芯片的运行。每一个 CPU 都支持一组通用或专用的指令集（Instruction Set），包括传送数据的指令、执行数学和逻辑运算的指令等。它可以是在通用功能的微控制器（如：8051）基础上扩展一些特殊功能的 CPU，也可以是专门为 USB 通信而设计的专用 CPU。例如，Cypress 公司的 EZ-USB 就嵌入了一个增强的 8051 微控制器内核。

没有通用功能 CPU 的 USB 控制器也可以支持与 USB 相关的通信命令集，或是通过使用一系列的寄存器来存储通信需要的 USB 数据和配置信息。如果要执行其他的功能，用户就必须使用外接的微控制器，例如：Philips 公司的 PDIUSBD12 芯片。

9.1.3 数据缓冲器

所有的 USB 控制器芯片都应该提供数据的接收和发送缓冲区（Buffer），用来提供 CPU 和外部逻辑之间的缓冲。有的芯片使用寄存器来作为缓冲区，例如：Netchip 公司的 NET2888；有的芯片则用存储器来作为缓冲区，例如：Cypress 公司的 EZ-USB。

数据缓冲器经常使用 FIFO（First In First Out，先进先出）结构。在缓冲区内会有一个指针，当固件读取或写入缓冲区时，这个指针会自动增加以指向下一个要访问的数据。有些芯片，例如：Cypress 公司的 CY7C63000，USB 缓冲区位于数据存储器中，固件会自动选择要访问的数据在存储器中的位置，这类芯片没有使用自动加一的指针来确定地址。USB 传输缓冲区内数据按照从低字节到高字节的顺序依次输出，而 USB 接收缓冲区的数据也将会按照接收到的数据从低字节到高字节的地址进行存储。这种缓冲器虽然已经不是传统的 FIFO 缓冲器，但是仍然沿用这一叫法。

有些芯片为了得到更高的传输速率，为其中有些端点使用了双缓冲区，从而增加数据吞吐量，满足实时数据的传输。双缓冲区是指在每个传输方向上为端点设计有两个缓冲区。如果是发送缓冲区，当其中一个缓冲区的数据正在传输时，用户可以同时将接下来要传送的数据写入另一缓冲区。这样，当第一个缓冲区的数据传输完成之后，第二个缓冲区的数据也已经准备好了，可以立即开始传输，节省了等待的时间。同样，如果是接收缓冲区的话，在用户处理好第一个接收缓冲区的数据之前，第二个接收缓冲区仍然可以接收数据。并且，这两个缓冲区之间会自动切换，不需要用户进行管理。

9.1.4 程序存储器

程序存储器是用于存储 CPU 要执行的程序代码的器件，这些代码可以帮助控制器完成 USB 通信任务，以及其他的辅助工作。程序存储器可以位于 CPU 内部或者是一个单独的芯片中。

程序存储器有很多种类型：ROM、EPROM、EEPROM、Flash ERPROM 和 RAM。这些程序存储器除了 RAM 以外，其他的类型都是非易失型的，即掉电以后保留的程序不会丢失。存储器的容量大小从几千字节（Kilobyte）到一兆（Megabyte）字节以上。通常所提到的固件程序（Firmware），就是放在程序存储器中的程序，之所以称其为固件，是因为它的数据在掉电后不会丢失。不像加载在 RAM 中的程序，每次上电都要重新写入。

ROM（Read-Only Memory）是厂家掩膜编程的器件，必须在出厂之前就烧制好，并且不能再次修改。

EPROM（Erasable Programmed ROM）是用户可编程的器件，但是编程和擦除都比较麻烦。编程需要专门的烧写器，而如果用户想要重新编程的话，先要用擦除器进行擦除，也就是用紫外线照射芯片，通常需要数十分钟的时间，之后才能重新烧制新的程序。而且这种器件如果放置时间过长，或是长时间暴露在太阳之下，就会因为紫外线的照射而丢失内部的程序。

OTP PROM（One-Time Programmable）顾名思义是一种只能编程一次的器件。这种器件比较便宜，经常用来作为 EPROM 的替代品。它的编程方式类似于 EPROM，但是不能擦除和重新写入。因此，用户在开发固件的过程中，为了便于修改可以使用 EPROM。一旦成型后就可以使用 OTP 器件进行大批量的生产。

Flash EPROM 是最近开发的可擦除存储器件。它不需要紫外线照射的石英窗口，也不需要特殊的编程电压，而且可以反复进行擦除和重写。

EEPROM（Electrically Erasable PROM）是电可擦除的 PROM 器件。它也不需要石英窗口以及特殊的编程电压。EEPROM 的存取时间要比 Flash PROM 长。EEPROM 可以提供与 EPROM 以及 Flash PROM 一样的并行接口，也可以使用同步串行接口。例如：Microwire、I2C 和 SPI。串行的 EEPROM 适合存储小容量的数据。例如，Cypress 的 EZ-USB 控制芯片就提供了一个 I2C 接口，用户可以将供应商 ID 和产品 ID 存储在串行 EEPROM 之中，并且连接至 I2C 接口。设备列举时通过 EEPROM 来提供这两个 ID 号。

RAM（Random-Access Memory）是一种易失型的存储器，掉电之后所有的数据都会丢失。但是它可以任意地擦除和重新写入数据，可以实现在线编程。Cypress 的 EZ-USB 就是使用 RAM 来存储程序的，而且使用了特殊的驱动程序，使得上电之后程序自动从主机加载到设备的存储器中。

9.1.5 数据存储器

数据存储器是用于在程序执行期间暂时存储数据。例如，从 USB 端口接收到的数据，要发送到 USB 端口的数据，以及程序执行期间所有要保存的数据都存放在数据存储器中。数据

存储器经常使用的是 RAM，有些芯片也使用非易失性器件来保存不经常变动的数据，例如设备的配置信息等。

9.1.6　寄存器

寄存器位于芯片数据存储区中，作为临时存放数据之用。但是，CPU 使用不同于其他存储器的指令来访问寄存器。而且寄存器都有预先定义的功能，不需要程序的控制，所以存取的速度比其他存储器快。

USB 控制器芯片所提供的寄存器可以分为状态寄存器和控制寄存器。状态寄存器用于保存芯片当前的一些状态，可以供用户查询；而控制寄存器则是用来产生控制信号的。例如用户可以设置某个控制寄存器特定的一位，启动或是挂一个特定的端点。

9.1.7 其他接口

每一个 USB 芯片除了有 USB 端口外，还会有其他的接口和外围设备，用于进行数据通信。一般 USB 控制芯片都会有通用的并行数据 I/O 口，用于将从 USB 端口收到的数据传输给其他的电路，或是将外部的数据传输给 USB 控制器芯片，然后通过 USB 端口传送给主机。另外，有的芯片还会提供串行的接口，例如 RS-232 的异步串口，或者是 Microwire、I2C 和 SPI 的同步串行接口。比如，Cypress 的 EZ-USB 就提供了 I2C 接口。

有些芯片还会提供特殊用途的接口，例如 Philips 公司的 USA1321 芯片就包含有一个数字模拟转换器（DAC，Digital-to-Analog Converter），可以使用在扬声器等音频设备上。芯片可以将 USB 数据转换成为模拟信号，取样频率可以达到 55kHz。

9.2 芯片构架

从芯片大的构架来分，市面上所有的 USB 控制器芯片可以分为不需要外接微控制器的芯片和需要外接微控制器的芯片。而不需要外接微控制器的芯片又可以分为专门为 USB 设计的芯片和嵌入通用微控制器内核的芯片。这几类芯片都有各自不同的特点，因此，用户可以根据设计的需要进行选择。

9.2.1 专门为 USB 设计的 USB 控制芯片

这些芯片是厂商为开发 USB 应用设备而特别设计的。内部用的是专用的 CPU（比如 Cypress 公司的 CY7C63101A 内部就是 8 位 RISC 微控制器，采用的是哈佛架构），这些芯片都有一套特定的指令集。但是不要期望这些指令集会像通用微控制器的指令集一样丰富，所以它们所作的工作是有限的。如果想要实现其他的一些功能，就必须想其他的办法。但是这些指令集都是为 USB 应用优化的指令集，因此，对于完成 USB 通信工作非常适用。Cypress 公司就有几款这样的芯片，比如 USB M8 系列和 enCoRe USB 系列的芯片。如表 9-1 所示为 CY7C63101A（M8）、CY7C63723（enCoRe）和 CY7C64113 的各项参数对比，同时它们的逻辑框图分别如图 9-2、图 9-3 和图 9-4 所示。

表 9-1　　Cypress 公司专门为 USB 设计的 USB 控制器芯片

芯　　片	CY7C63101A（M8）	CY7C63723（enCoRe）	CY7C64113
支持的速度	低速 1.5Mbit/s	低速 1.5Mbit/s	全速 12Mbit/s
端点数目	2（一个控制端点一个数据端点）	3	5
缓冲区字节大小	8Byte	8Byte	8 或 32Byte
RAM 容量（Byte）	128Byte	256Byte	256Byte
程序存储器类型	EPROM	EPROM	EPROM

程序存储器的大小（Byte）	4KB	8KB	8KB
通用 I/O 管脚	16	16	32

续表

芯　　片	CY7C63101A（M8）	CY7C63723（enCoRe）	CY7C64113
其他 I/O 接口	无	SPI	I2C、DAC 功能
电源电压	5V	4.0～5.5V	4.0～5.5V
管脚数目	24	18	48

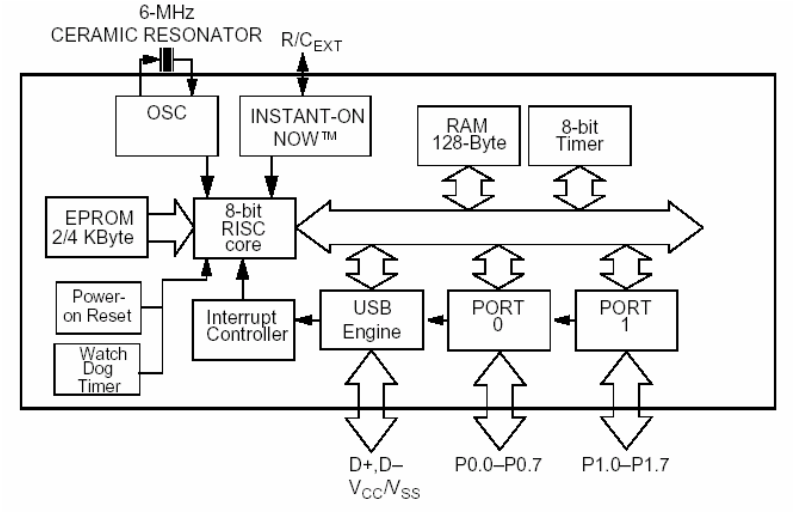


图 9-2　CY7C63101A（M8）逻辑框图

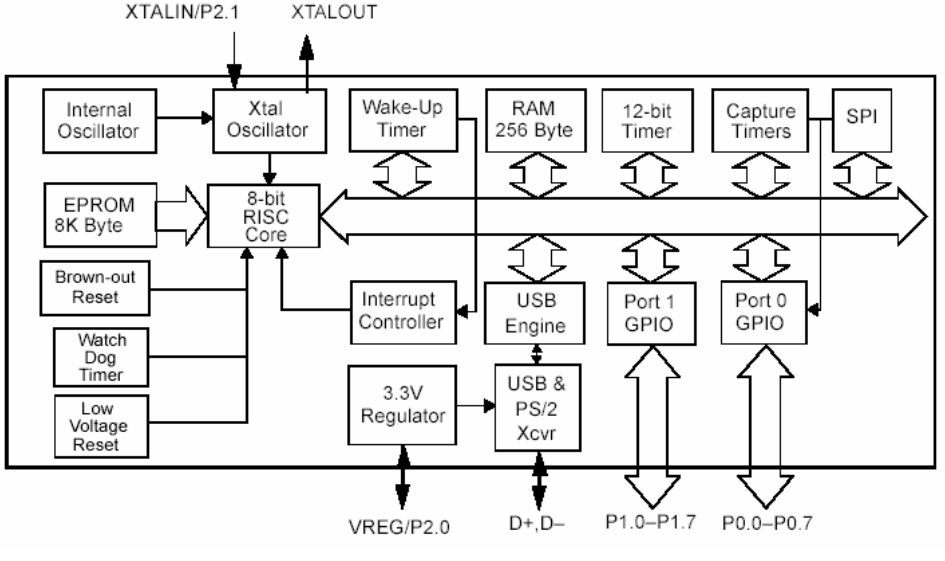


图 9-3　CY7C63723（enCoRe）的逻辑框图

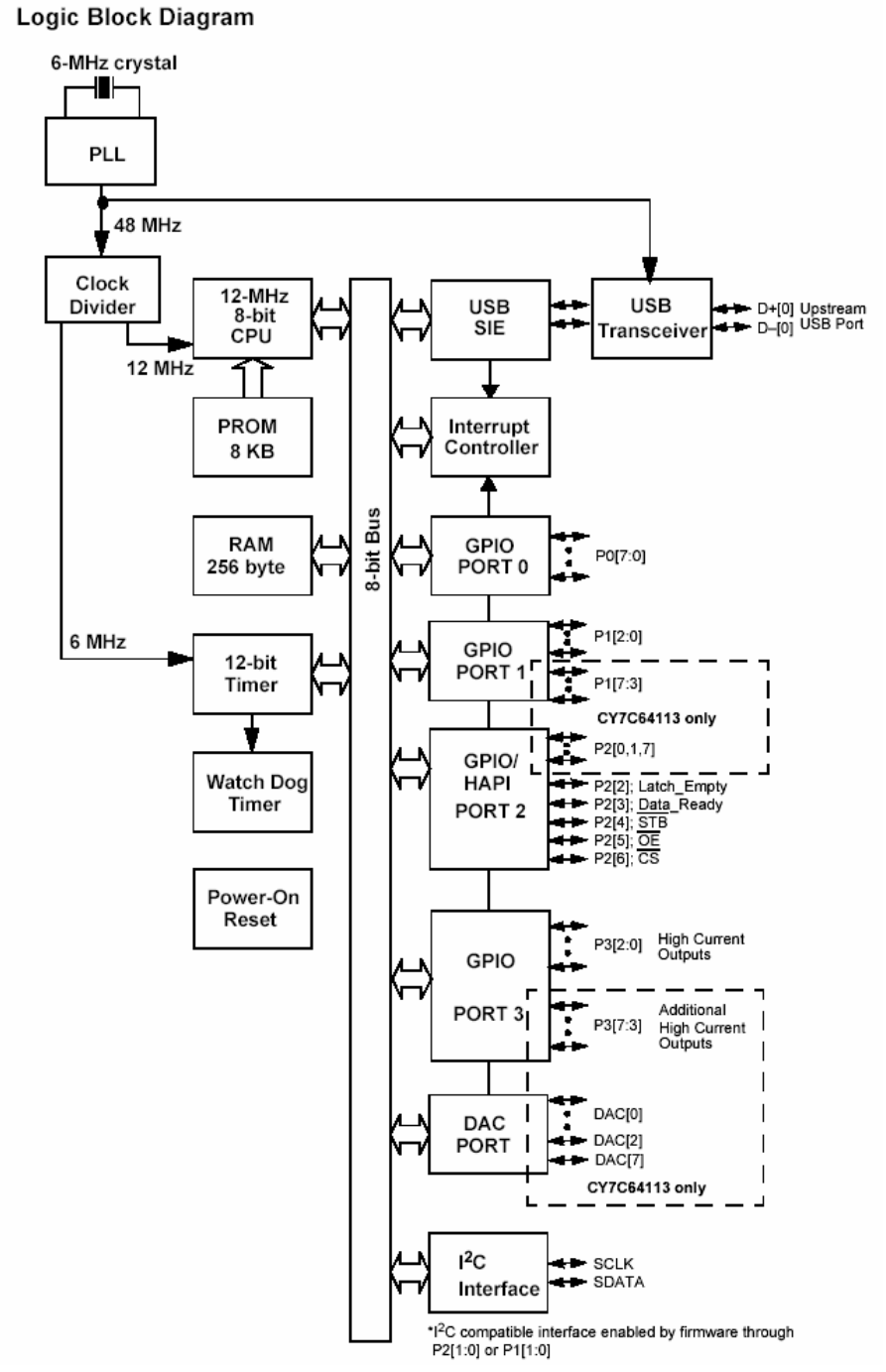


图 9-4　CY7C64113 的逻辑框图

9.2.2　内嵌通用微控制器的芯片

内嵌通用微控制器的 USB 控制芯片可以说是在通用的微控制器的基础上扩展了 USB 功能。这些控制器芯片的优点是开发者已经熟悉了这些通用微控制器的结构和指令集，所以开发起来就相对比较容易。即使用户不熟悉这些芯片的结构，但是介绍这些微控制器的书籍以及关于它们的范例程序、开发工具等都可以作为设计时的参考。有几家公司推出了基于 8051 的 USB 控制芯片，例如 Cypress 公司的 EZ-USB 芯片内部就嵌入了增强型的 8051 内核。8051 系列微控制器在我国使用很广泛，因此，这类 USB 控制芯片使用起来就比较方便。

另外，其他的公司也生产了与它们各自的通用微控制器兼容的 USB 控制芯片，如表 9-2 所示。

表 9-2　　基于通用微控制器的 USB 控制芯片

公司	兼容微控制器	产品型号
AMD	80C186	AM186
Atmel	Atmel AVR	AT76C711
Cypress	Enhanced 8051	EZ-USB 系列
SIEMENS	80C51/80C52	C541U
Microchip	PIC	16C7x5
Motorola	68HC08	68HC08JB3

9.2.3　需要外接微控制器的芯片

这些 USB 控制芯片只处理 USB 相关的通信工作，而且必须由外部微控制器的控制才能正常工作。如果选择了这种设计方案，那么必须再选择一个微控制器芯片，这样就增加了设备的体积。但是也有一个优点就是用户可以选择任何一种自己熟悉的微控制器，而且这种芯片也比较便宜。既然需要外接微控制器，这些芯片就必须提供一个串行或并行的数据总线微控来与控制器进行连接。另外还需要一个中断引脚，当接口芯片收到或是发送完 USB 总线数据时，这个中断引脚会向微控制器发出中断信号。

Philips 公司的 PDIUSBD12 提供了 8 位并行数据总线，2Mbit/s 的读写速率，如果是在 DMA 模式下会更快。Netchip 公司的 NET2888 芯片使用的也是并行数据总线，有 8 条数据线和 5 条地址线。读写速率可以达到 10MB/s，也可以运行于 DMA 模式下。National Semiconductor 公司的 USBN9603 芯片有较为灵活的选择。该芯片有一个 8 位并行总线，可以有两种寻址选择模式：复用和非复用（Intel 兼容模式）。这两种模式可以通过芯片提供的引脚 MODE0 进行选择。另外，芯片还包括一个 Microwire 同步串行数据。Microwire 只需要 4 条线就可以与任何微控制器连接，而串行和并行的连接方式之间又可以由引脚 MODE1 来进行选择。部分此类芯片的比较如表 9-3 所示。

北京海德电子公司开发的 USB 控制芯片的逻辑框图如图 9-5 所示，其他芯片的逻辑框图将在下一节芯片介绍中给出。

表 9-3　　需外接微控制器的芯片

芯　　片	USBN9603	NET2888	PDIUSBD12	HW9911
制造商	National Semiconductor	NetChip	Philips	海德电子
传输速率	全速	全速	全速	全速
端点数目	一个控制端点+6 个其他数据传输端点	一个控制端点+5 个其他数据传输端点	一个控制端点+4 个其他数据传输端点	一个控制端点+4 个其他数据传输端点

续表

芯　　片	USBN9603	NET2888	PDIUSBD12	HW9911
双缓冲区	无	无	有	无
微控制器接口	复用或非复用并行总线	非复用并行总线	复用或非复用并行总线	复用或非复用并行总线　和 I2C 串行总线
电源电压	3.3V 或 5V	3.3	4.0～5V/3.3V	5V
管脚数	28	48	28	28

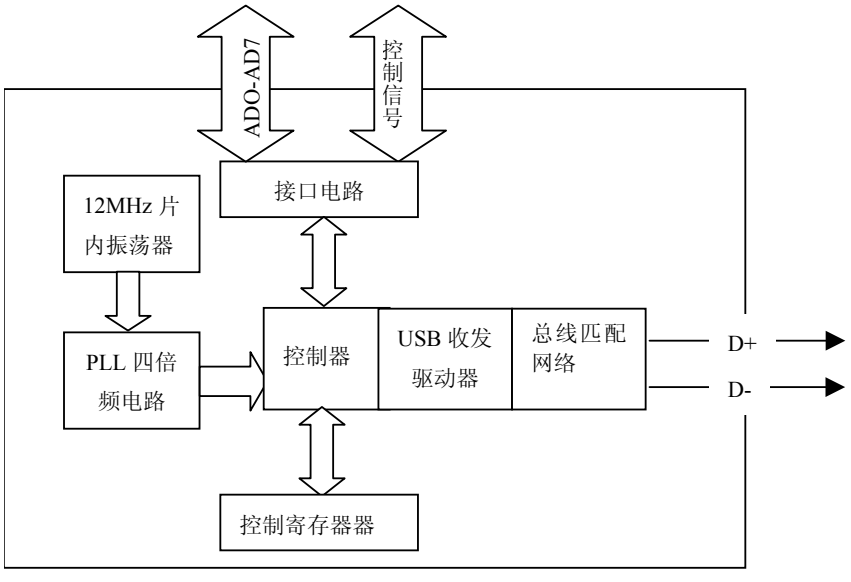


图 9-5　HW9911 内部逻辑框图

9.3　芯片举例

前面两个小节分别介绍了构成 USB 控制芯片的组件和 USB 控制芯片的架构分类，在这一小节中将会具体地介绍几款 USB 控制芯片。

9.3.1　Cypress 公司的 M8 CY7C63101A 芯片

CY7C63101A 芯片属于 Cypress 公司生产的 M8 系列产品，同系列的还包括有 CY7C63001A、CY7C63413/513/613 等产品。该系列芯片为低价位和低速 USB 外设提供了很好的解决方案，特别适用于人机接口类的计算机外设，如：鼠标、键盘、游戏杆等低速外设。CY7C630/1xxA 系列芯片都具有相同的结构，只是内部存储器大小、管脚数以及封装形式有所不同而已。图 9-2 所示为该系列芯片的逻辑框图，图 9-6 是芯片的管脚封装图。

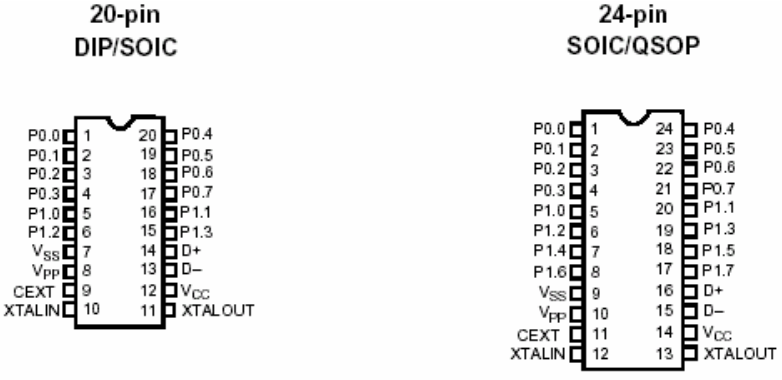


图 9-6　CY7C63101A 芯片管脚图

CY7C630/1xxA 系列芯片内部是 8 位 RISC OTP 微控制器，由于是专门为 USB 设计的内核，所以与通用的微控制器结构不同，它采用的是哈佛（Harvard）结构。该系列芯片属于精简指令系统的微控制器，指令执行效率很高。一共有 35 条专门为 USB 应用优化的指令，包括基本的数据移动、算术和逻辑运算以及跳转指令。指令集数量有限，减轻了用户的学习压力，但同时用户也不能期望有完成复杂任务的指令，比如乘法或是除法指令，用户只能通过加法、减法以及移位指令综合起来完成这些工作。

CY7C630/1xxA 系列芯片使用 EPROM 作为用户的程序存储器，使用 SRAM 作为数据存储器。其中，CY7C63000A 和 CY7C63100A 都是 2KB 的程序存储空间。需要更多程序空间的用户可以选用 CY7C63001A 和 CY7C63101A，这两款芯片都提供了 4KB 的 EPROM。程序空间被分为两个功能组：中断向量和程序代码。另外，Cypress 公司还为用户设计了一个安全熔断位，当安全熔断被编程以后，EPROM 的编程器只能从 EPROM 程序空间读出 0xFF，这样就确保了用户代码的安全。除了程序存储器外，CY7C630/1xxA 系列芯片还包括了 128Byte 的数据 RAM，其中高 16 个 Byte 用作端点 0 和端点 1 的数据 FIFO。

CY7C630/1xxA 系列芯片还包括有 16 位的 GPI/O（General Purpose I/O），使用户可以方便地扩展外围设备。另外，还有一个看门狗定时器（Watch Dog Timer），提供 WDR（Watch Dog Reset）功能，保证用户程序正确执行。

CY7C630/1xxA 系列芯片符合 USB 1.1 规范，但是只支持低速 1.5Mbit/s。因此，不能使用批量传输和等时传输。而且这些芯片只支持 1 个设备地址和两个端点（一个控制端点和一个数据端点），数据端点可以被配置为中断传输类型。每个端点有一个 8Byte 的缓冲区。

9.3.2 Cypress 公司的 EZ-USB 芯片

Cypress 半导体公司的 EZ-USB 系列包括了 6 个分类：基本的 EZ-USB（AN21xx）、EZ-USB FX（CY7C646xx）、EZ-USB FX2（CY7C68013）、EZ-USB SX2（CY7C68001）、EZ-USB TX2（CY7C68000）、EZ-USB AT2（CY7C68300）。这几种类型中，FX（Faster Xcelerator）新增了较快的 I/O，以及一个可编程接口，来支持配置和自动联络。其他的 FX2、SX2、TX2 和 AT2 几种类型都是为 USB 2.0 设计的解决方案，支持高速数据传输（480Mbit/s），而且各自有不同的特点和应用范围。FX2 具有可编程的 MCU，为高速 USB 外设设计的高集成单片解决方案。SX2 具有智能的 SIE core，可以自动处理所有底层的 USB 协议工作，缩短了 USB 的学习曲线。TX2 只是一个高速 UTMI 兼容的收发器和串行/解传器，并提供了一个可以运行于 30MHz 的 16 位并行接口或是运行于 60MHz 的 8 位并行接口。AT2 提供了一个从一个 USB 端口到一个 ATA 或者 ATAPI 的大容量存储设备之间的功能桥接。其中的 3 个主要的系列的比较如表 9-4 所示。AN21xx (EZ-USB)、CY7C646xx (EZ-USB FX)和 CY7C68013 (EZ-USB FX2) 的逻辑方框图如图 9-7、图 9-8 和图 9-9 所示。

表 9-4　　EZ-USB 系列芯片

芯片	AN21xx(EZ-USB)	CY7C646xx(EZ-USB FX)	CY7C68013(EZ-USB FX2)
速度	全速	全速	高速
端点数目	13，16，31	31	11
内核微控制器	增强型（Enhanced）8051	增强型（Enhanced）8051	增强型（Enhanced）8051
数据存储器大小（RAM）	256+4/8KB 复合数据与程序内存	256+4/8KB 复合数据与程序内存	256+4/8KB 复合数据与程序内存
内部程序存储器类型	RAM，串行 EEPROM 或外部并行存储器	RAM，串行 EEPROM 或外部并行存储器	RAM，串行 EEPROM 或外部并行存储器
内部程序存储器空间	4/8KB 复合数据与程序内存	4/8KB 复合数据与程序内存	4/8KB 复合数据与程序内存
外部程序存储器扩展空间	64KB	64KB	一个或两个 64KB
通用 I/O 管脚	16～24	16～40	16～40
其他 I/O	2UARTs、I2C	2UARTs、I2C	2UARTs、I2C
供电电压	3～3.6	3～3.6	3～3.6
管脚数目	44，48，80	52，80，128	56，100，128

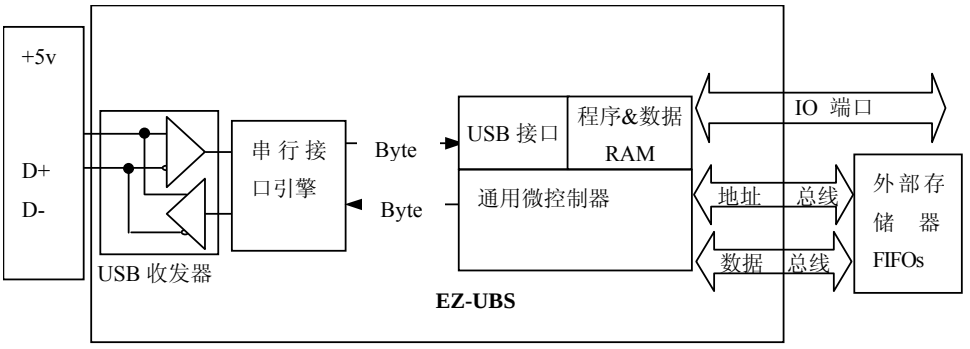


图 9-7　AN2131Q（80 脚）简化逻辑方框图

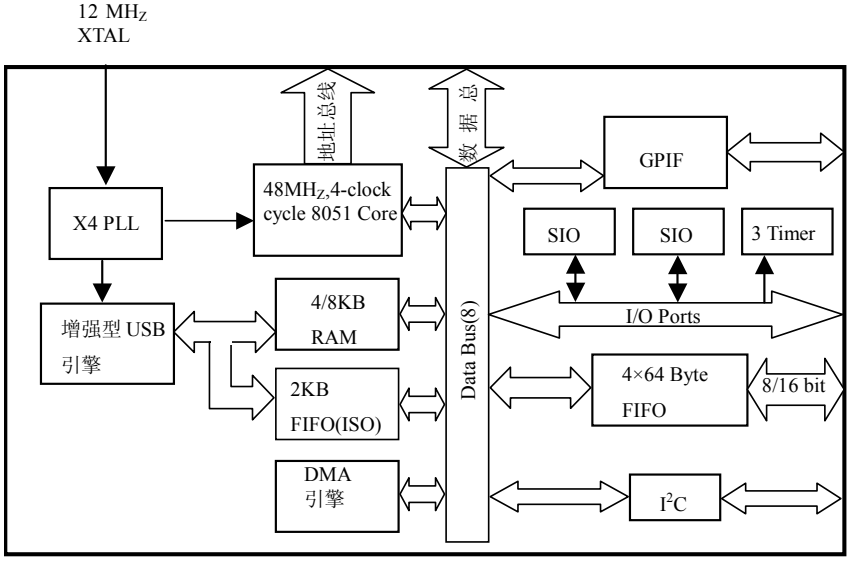


图 9-8　CY7C64613（128 脚）逻辑方框图

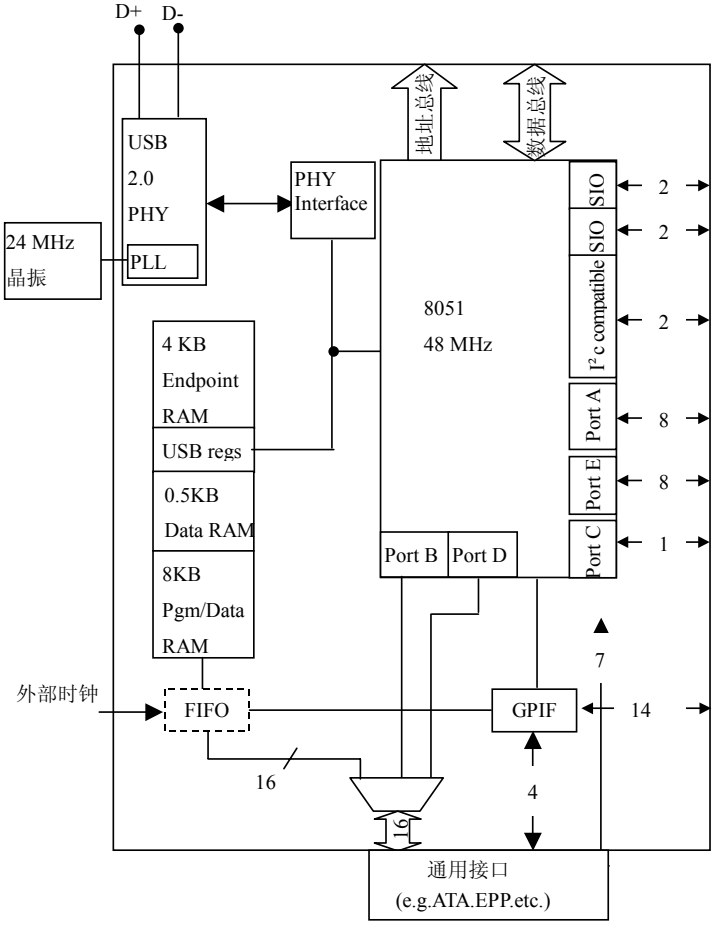


图 9-9　CY7C68013 逻辑方框图

Cypress 公司的 EZ-USB 系列芯片有以下 3 个主要的特点：

- EZ-USB 系列提供一个软的解决方案（RAM-based），允许无限制地配置与升级。
- 满足最大限制的 USB 吞吐量，使用 EZ-USB 系列芯片进行的设计将不会受到端点数量、缓冲区大小和传输速率的限制。
- EZ-USB 系列芯片的 USB 核将会处理大部分的 USB 事务，简化了固件代码，加速了 USB 的学习曲线。

EZ-USB 内嵌增强型 8051 微控制器内核，并使用标准 8051 指令集，但是指令的执行速度却比标准 8051 快。因为该内核中的一个总线周期由 4 个时钟周期组成，而标准 8051 的一个总线周期是由 12 个时钟周期组成的，而且，它的时钟频率为 24MHz 或者 48MHz，而标准的 8051 的时钟频率为 6MHz 或者 12MHz。

除了速度的改进之外，增强型 8051 内核还包括以下结构：

- 双数据指针。
- 两个 UART（Universal Asynchronous Receiver/Transmitter）。
- 三个定时器（增加了 16 位的 Timer2）。
- 提供了一个非复用的 16 位地址总线作为高速存储器接口。

- 增加了 8 个中断（INT2-INT6、WAKEUP、T2 和 UART1）。
- 可变的 MOVX 指令时间以满足低速和高速的 RAM 外设。
- 3.3V 供电。

EZ-USB 系列芯片使用的是智能增强型的 SIE（Serial Interface Engine）。它具有执行所有 USB 列举过程的功能，并且能够通过预设的端点和可选的设置来创建一个缺省的 USB 设备，这就大大减轻了固件的工作。而且增强型的 SIE 对于该系列芯片提供的“软”操作特性也起到关键的作用，因为它提供了在 8051 运行之前下载固件的机制。

EZ-USB 提供了丰富的端点和缓冲。例如，CY7C64613 就提供了一个双向的控制端点，14 个批量/中断端点，每一个都支持一个 64Byte 的最大包，还有 16 个等时端点，共享 2KB 的缓冲空间。

EZ-USB 系列最大的一个特点就在于固件的存储和更新上，EZ-USB 芯片可以将固件存储在主机上，而不必存入芯片之中，这也就是我们前面提到的软特性。但是固件程序没有存储在设备之中，设备是如何工作以响应主机的列举过程的呢？其实答案在前面刚刚讲述过，那就是因为该芯片内部具有智能增强型的 SIE，它可以不需要固件的参与以及微控制器的控制就能响应主机的列举请求，并将设备配置成为缺省设备。而且 SIE 还可以响应供应商特定请求 AnchorLoad，这个请求就是用来下载固件的。其次就是主机方面要有相应的驱动程序的支持。Cypress 提供了 ezloader.sys 驱动程序来完成开机时固件的下载工作。所有这些工作都是在芯片的 8051 内核处于复位状态时进行的。

在 USB 设备接入主机后，USB 核将会根据外部 I2C 总线接口上的一个串行 EEPROM 中的内容来决定如何列举。这样就会有 3 种不同的模式：

- 最简单的模式是 USB 内核没有检测到 EEPROM 的存在，或者是 EEPROM 存在但第一个字节的内容不是 0xB4 或 0xB6（EZ-USB FX）。这种情况下，USB 核会使用内部存储的描述符数据进行列举，这些描述符数据包括有 Cypress 公司的 VID、PID 和 DID，如图 9-10 所示。这些 ID 字节将会导致主机操作系统载入 Cypress 公司的通用驱动程序。USB 内核也将建立缺省 USB 设备。这时，需要用户通过 Cypress 提供的主机应用程序进行固件代码的下载。这种模式主要用于固件的开发和调试过程。
- 第二种情况是存在串行 EEPROM，并且第一个字节是 0xB4。USB 核将会像没有串行 EEPROM 那样以相同的内部存储的描述符数据进行列举，但是有一点不同：PID/VID/DID 这 6 个字节的数据是来自 EEPROM，而不是 USB 核。这些由用户提供的 VID/PID/DID 将会导致主机操作系统载入一个与这些 ID 号匹配的驱动程序。
- 最后一种情况是，如果 USB 核检测到了 EEPROM 连接在 I2C 总线上，而且第一个字节是 0xB6。那么，USB 核就会把 EEPROM 中的内容载入内部 RAM 中，同时设置 RENUM 位以指示由 8051 而不是 USB 核来响应主机的设备请求。因此，所有的描述符数据（包括 PID/VID/DID）都是由 8051 固件程序提供的，从 EEPROM 载入的最后一个字节会释放 8051 的复位信号，允许 EZ-USB 芯片按照 RAM 中的固件程序配置为用户想要的设备。

供应商 ID	0 × 0547 （ Cypress Semiconductor/ Anchor Chips）
产品 ID	0×2235（EZ-USB FX）
设备发行版本	0×XXYY（根据版本而定）

图 9-10　没有串行 EEPROM 时的 EZ-USB FX 设备特性

9.3.3 National Semiconductor USBN9603

National Semiconductor 的 USBN9603 是一个集成的 USB 节电控制器，它的突出特点是增强性能的 DMA，支持很多自动的数据处理工作。兼容 USB 规范 1.0 和 1.1，是 USB9602 的升级版本。该芯片属于需要外接微控制器的 USB 控制芯片，它可以使用 8 位多功能的并行数据总线、Microwire 串行接口或是由固件控制的 4 个 I/O 引脚，来连接到任何的外部微控制器。USBN9603 两种不同封装的引脚图如图 9-11 所示，USBN9603 的逻辑框图如图 9-12 所示。

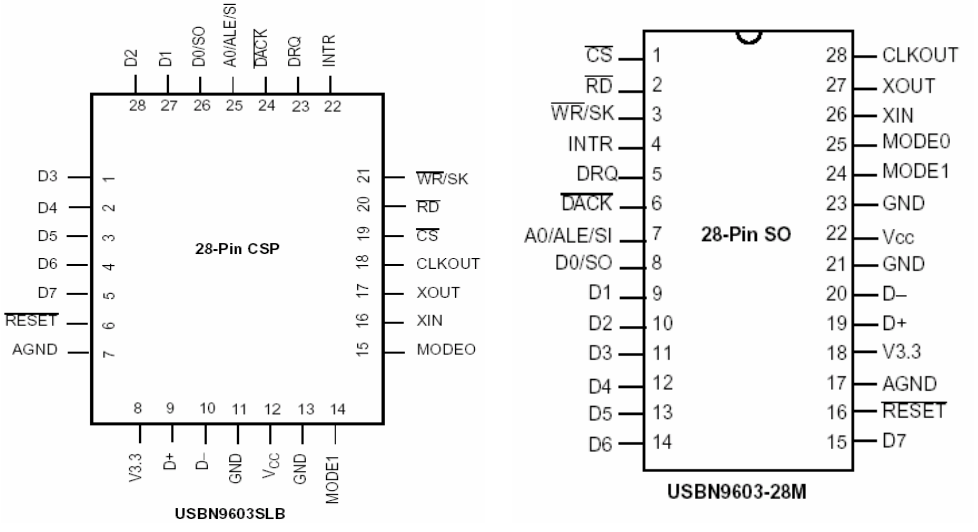


图 9-11　National Semiconductor USBN9603 引脚图

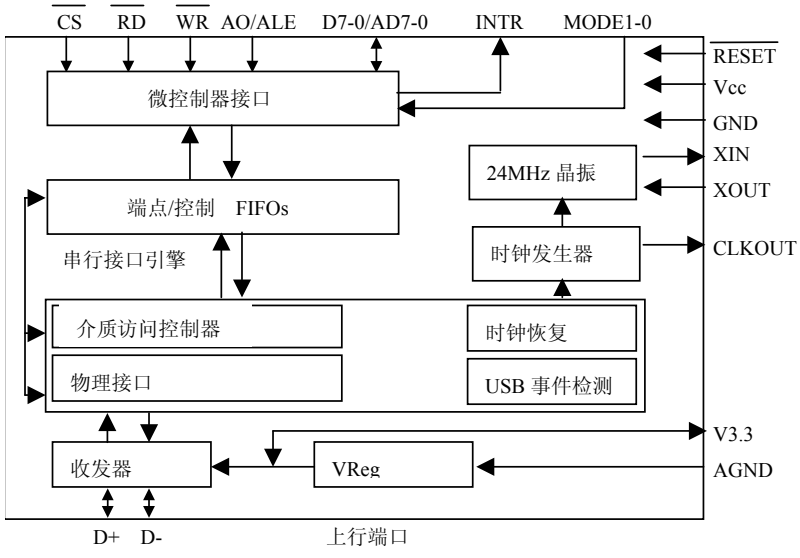


图 9-12　USBN9603 的逻辑框图

USBN9603 芯片集成了一个带 3.3V 稳压器的收发器和一个串行接口引擎（SIE）用来处理 USB 传输事务。还有 USB 端点缓冲、一个多功能的 8 位并行接口、一个时钟发生器和一个 Microwire/Plus 接口。微控制器可以通过外部的总线在地址 0x00 到 0x3F 直接访问端点缓冲区和状态以及控制寄存器。

USBN9603 提供了 3 种与外部微控制器的接口方式：

- 非多路复用并行方式（non-multiplexed parallel）。
- 多路复用方式（multiplexed parallel）。
- Microwire 同步串行方式（Microwire synchronous serial）。

多路复用模式使用控制引脚/CS、/RD、/WR，地址锁存使能信号 ALE 和双向的数据总线 AD7~AD0，如图 9-13 所示。这种模式的选择需要将 MODE1 接至 GND，而 MODE0 接至 VCC。多路复用模式是在一个总线周期内读取或写入一个字节数据，当 ALE 为高电平时，地址被锁存到 ADDR 寄存器中，当随后的/RD 或/WR 信号有效时，数据被输出或输入。这种接口模式下，所有的寄存器都可以被直接访问。这种接口模式的基本时序如图 9-14 所示。

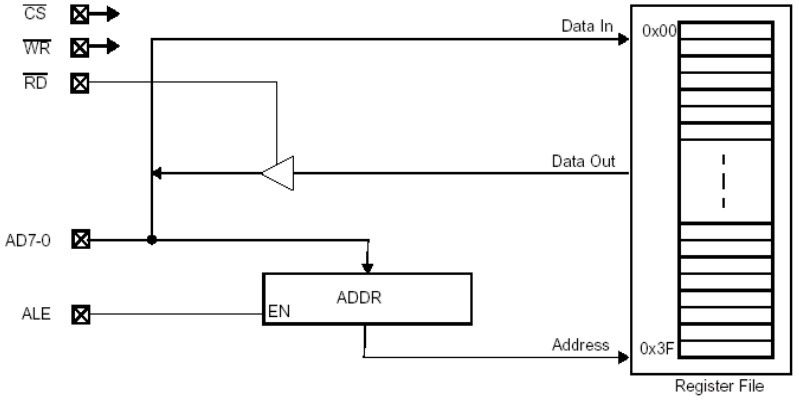


图 9-13　多路复用模式方框图

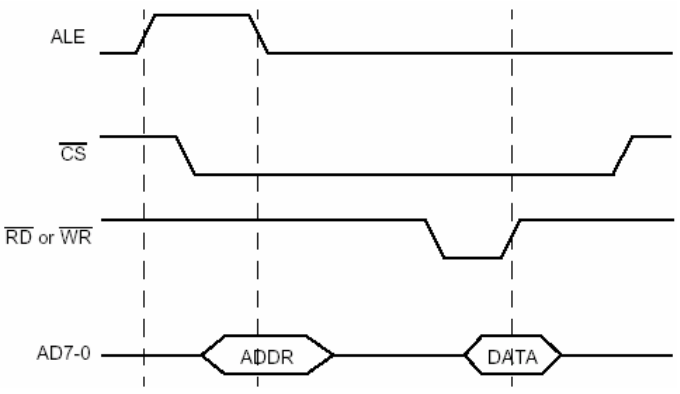


图 9-14　复用模式的基本读/写时序图

非多路复用方式使用控制引脚/CS、/RD、/WR，地址引脚 A0 和双向数据总线 D0~D7，这种模式需要将 MODE1 和 MODE0 引脚接至 GND，如图 9-15 所示。对于非多路复用传输方式，USBN9603 仍然用 D0~D7 传送数据和地址信号，但是地址和数据在不同的总线周期内传输。第一个总线周期发送地址，第二个总线周期传送数据。外部的微控制器可以直接访问 DATA_IN、DATA_OUT 和 ADDR 寄存器，读写数据又可以按两种不同的模式进行：标准访问模式和突发式模式。

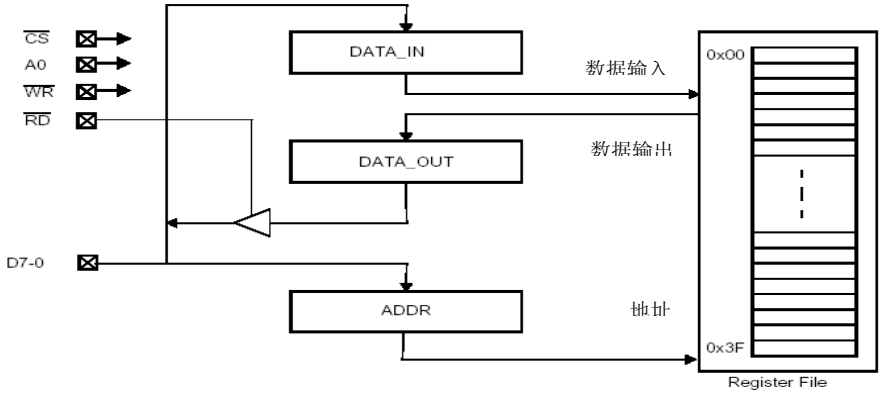


图 9-15　非复用模式方框图

❑　标准访问模式：对于非多路复用模式，标准访问顺序是先将地址写入 ADDR 寄存器，然后再从 DATA_OUT 寄存器中读取数据，或是向 DATA_IN 寄存器中写入数据。在写入 ADDR 寄存器之后，DATA_OUT 寄存器中的内容将会更新。芯片通过引脚 A0 来选择是访问寄存器 ADDR 还是 DATA_OUT/DATA_IN。

❑　突发模式：在突发模式中，ADDR 被写入一个期望的片内寄存器地址，然后就可以连续地从 DATA_OUT 寄存器中读取数据，或是向 DATA_IN 寄存器中写入数据，而不需要再写入新的地址。这样就节省了总线周期，可以达到更高的访问速度。DATA_OUT 的内容在每一次读或写操作之后都会更新。

非多路复用模式的时序图如图 9-16 所示。

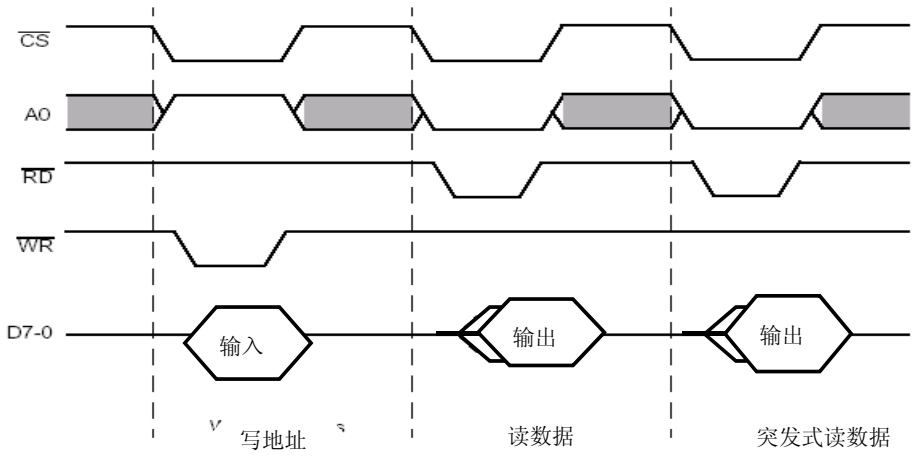


图 9-16　非多路复用模式的时序图

除了 8 位多功能并行数据总线外，USBN9603 还提供了一个同步串行总线 Microwire，用于连接外部微控制器。这种模式的选择需要将 MODE1 引脚拉高，而将 MODE0 引脚拉低。Microwire 模式使用 4 根线：片选线/CS（Chip Select）、串行时钟 SK（Serial clock）、串行数据输入线 SI（Serial data In）和串行数据输出线 SO（Serial data Out）。如图 9-17 所示为 Microwire/plus 接口由片选信号/CS 的下降沿使能，上升沿复位。在时钟信号 SK 的上升沿之后，数据在 SI 上移位输入；而在时钟信号 SK 的下降沿之后，数据在 SO 上移位输出。在第 8 个 SK 时钟的下降沿之后，数据被移入或是移出移位寄存器。数据的传输顺序按从高位到低位的顺序。

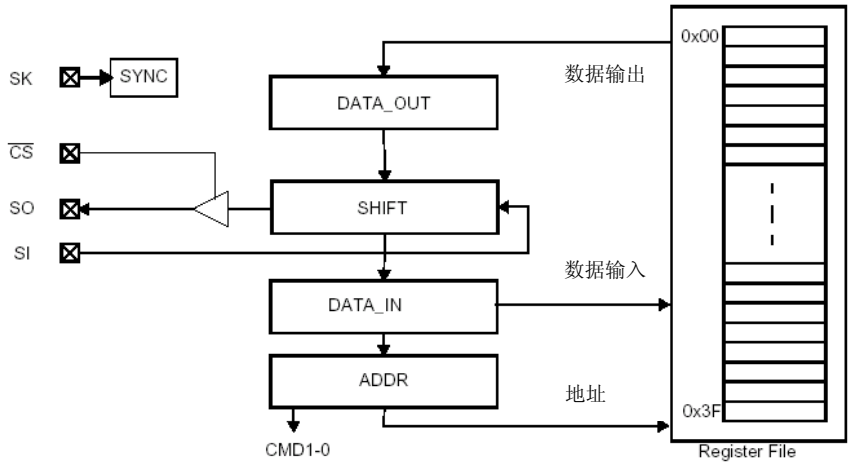


图 9-17　Microwire 接口方框图

USBN9603 芯片支持 7 个端点：一个双向的控制端点 0（带 8 字节的缓冲区）、3 个发送端点和 3 个接收端点（每个端点 64 字节的缓冲区）。一个端点也可以计划或是接收超过端点缓冲区大小的数据，只要固件按照数据到达的顺序从缓存区内及时读取，以防止缓冲区溢出；或是按照顺序写数据到缓冲区内，以防缓冲区在所有数据发送完之前变空。

9.3.4 Netchip NET2888

Netchip 公司的 NET2888 芯片不包括通用的微控制器和存储器，它只有一个 USB 控制器和一个通用的数据总线，通过数据总线可以连接到任何外部微控制器。

NET2888 除了 USB 端点的缓冲区之外，没有带有任何的程序和数据存储器。它的外部总线由 5 位地址线 A0~A4 和 8 位数据线 D0~D7 构成，最多可以访问 32 个地址。NET2888 的引脚图如图 9-18 所示。

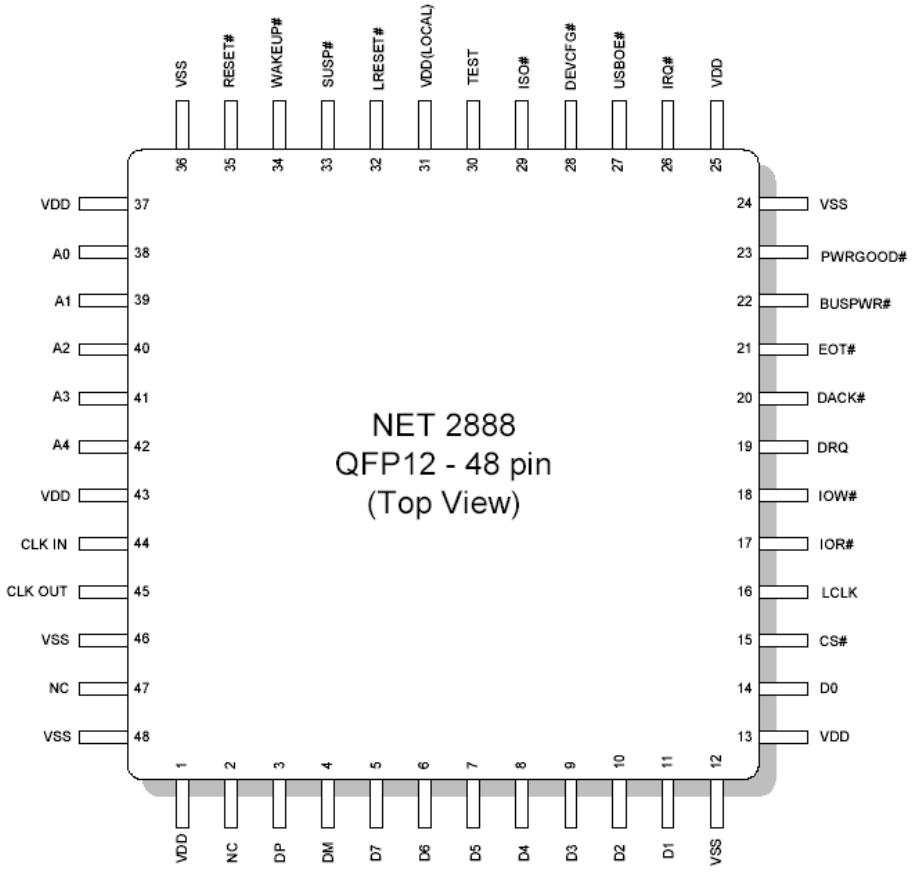


图 9-18　NET2888 引脚图

NET2888 USB 接口控制器允许在一个通用的本地总线和 USB 之间进行批量或者等时数据传输，NET2888 支持主机与一个智能外设的连接。例如，数字照相机或者扫描仪。该芯片由 3 个主要的组成部分：USB 总线接口、双 64 字节的 FIFOs 和一个局部的总线接口。其中，USB 接口负责如下的工作：

- 主机与设备的通信。
- 批量和等时端点访问端点 FIFOs。
- 中断端点访问局部总线到 USB 的 Mailbox 寄存器。
- 批量端点访问 USB 到局部总线的 Mailbox 寄存器。

而局部总线接口负责下面工作：

- FIFO 控制。
- 局部微控制器接口。
- 局部 DMA 控制器接口。
- 中断。

局部总线上的设备可以使用片选信号/CS 来使能对芯片 NET2888 内部寄存器的访问。如果信号/DACK（DMA Acknowledge）有效，那么片选信号/CS 被忽略。另外，芯片还提供了/IOR 和/IOW 信号线。当 I/O 读选通信号/IOR 连同片选/CS 和地址 A[0:4]一同有效时，局部总线上的设备将会读取内部寄存器或者 FIFO。而当 I/O 写选通信号/IOW 连同片选/CS 和地址 A[0:4]一同有效时，局部总线上的设备将会写入内部寄存器或 FIFO。

NET2888 芯片还支持直接存储器存取（Direct Memory Access），用以满足快速的数据传输。在 DMA 传输中，NET2888 芯片控制着局部总线。当请求一个 DMA 传输时，局部存储器和芯片 FIFO 之间的数据块传输就会自动进行，而不需要外部微控制器的参与。

NET2888 芯片内部保留了一块存储区，来存储要传输的数据。一个 DMA 地址计数器存储着这个数据块的内存地址；另一个 DMA 字节计数器保存了剩余数据的字节数目。在一个主机到设备的传输中，当设备接收到 USB 数据时，它就将数据复制到保留的内存中。而在一个设备到主机的传输中，只要传输缓冲区为空，设备就将数据复制到传输缓冲区中进行发送。NET2888 芯片的逻辑方框图如图 9-19 所示。

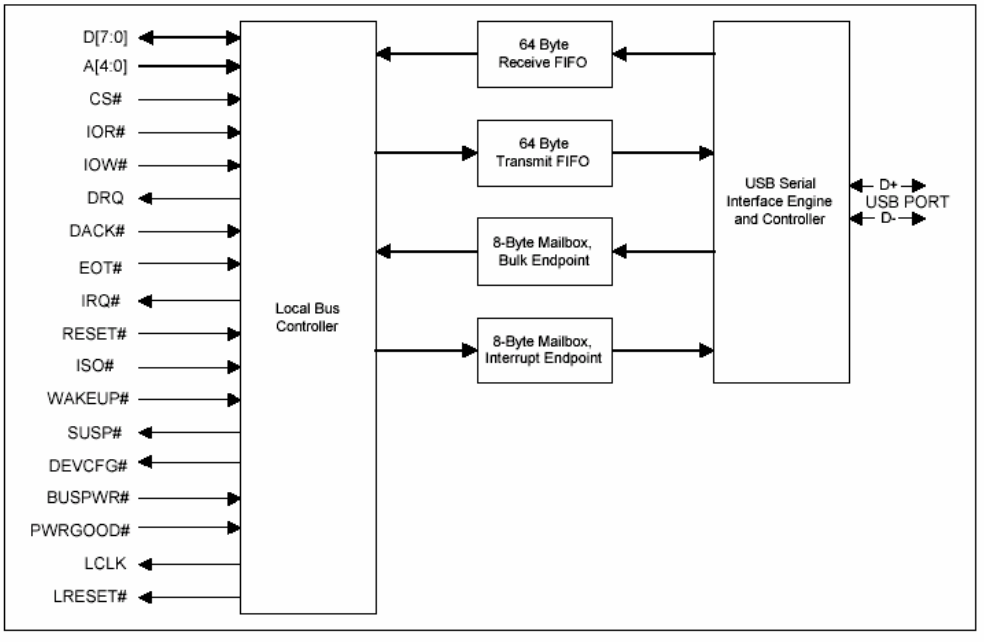


图 9-19　NET2888 芯片的逻辑方框图

NET2888 芯片有 5 个端点，支持全部的 4 种传输类型：端点 0 为双向的控制传输，端点 1 支持 OUT 方向的批量传输，端点 2 支持 IN 方向的中断传输，端点 3 支持 OUT 方向的批量或者等时传输，端点 4 支持 IN 方向的批量或者等时传输。

端点 1 和端点 2 分别有 8 个字节的邮箱寄存器（Mailbox Register），也就是数据缓冲区，来传送与接收 USB 数据。每一个邮箱寄存器的数据使用了局部总线的一个单一地址。第二个

地址包含一个索引值，来指示邮箱寄存器内的数据是要读取的还是要写入的。

端点 3 和端点 4 分别有 64 个字节的缓冲区，用来传送与接收 USB 数据。每一个缓冲区使用一个单一的地址，以及一个计数寄存器来指示缓冲区内的字节数目。

外部微控制器负责写入配置信息到 NET2888 寄存器中，但是端点的配置工作是由硬件完成的，所以固件要做的工作比其他芯片少。

9.3.5 Philips Semiconductor PDIUSBD12

PDIUSBD12 是一款性价比很高的 USB 器件，它通常在微控制器系统中实现与微控制器进行通信的高速通用并行接口，它还支持本地的 DMA 传输。

这种实现 USB 接口的标准组件使得设计者可以在各种不同类型微控制器中选择出最合适的微控制器，这种灵活性减小了开发的时间、风险以及费用（通过使用已有的结构和减少固件上的投资），从而用最快捷的方法实现最经济的 USB 外设的解决方案。PDIUSBD12 完全符合 USB1.1 版的规范，它还符合大多数器件的分类规格：成像类、海量存储器件、通信器件、打印设备以及人机接口设备。同样地，PDIUSBD12 理想地适用于许多外设，例如打印机、扫描仪、外部的存储设备（Zip 驱动器）和数码相机等。它使得当前使用 SCSI 的系统可以立即降低成本。PDIUSBD12 的引脚图如图 9-20 所示。

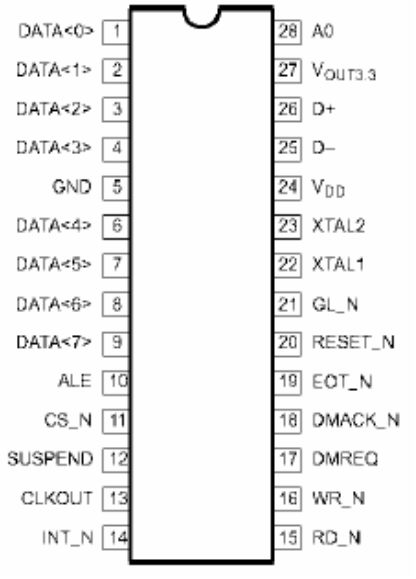


图 9-20　PDIUSBD12 的引脚图

PDIUSBD12 所具有的低挂起功耗连同 LazyClock 输出可以满足使用 ACPI、OnNOW 和 USB 电源管理的要求。低的操作功耗可以应用于使用总线供电的外设。此外，它还集成了许多特性，包括 SoftConnet^{TM}、GoodLink^{TM}、可编程时钟输出、低频晶振和终止寄存器集合。所有这些特性都显著为系统节约了成本，同时使 USB 功能在外设上的应用变得容易。

SoftConnet^{TM} 技术是 Philips 公司的一项新技术。USB 控制芯片与主机的连接是通过 1.5KΩ 的上拉电阻将 D+线（用于全速 USB 器件）置为高电平实现的。1.5KΩ 上拉电阻集成在 PDIUSBD12 片内，默认状态下不与 VCC 相连。连接的建立通过外部系统微控制器发送命

令来实现。这就允许系统微控制器在决定与 USB 建立连接之前完成初始化时序。USB 总线连接可以重新初始化而不需要拔出电缆。

PDIUSBD12 在连接可以建立之前会检测 USB VBUS 是否可用。VBUS 可通过 EOT_N 管脚进行检测。

注意：内部电阻的误差为 25%，大于 USB 规格的 5%。但用于连接的 VSE 电压规格仍然有足够的余量。

GoodLink™ 技术可提供良好的 USB 连接指示。在枚举中，接在引脚 GL_N 上的 LED 指示根据通信的状况间歇闪烁。当 PDIUSBD12 成功地枚举和配置后，LED 指示将一直点亮。随后主机与 PDIUSBD12 之间成功的数据传输和应答将关闭 LED。处于挂起状态时，LED 也将会关闭。

该特性为 USB 器件、集线器和 USB 通信状态提供了用户友好的指示。作为一个诊断工具，它对隔离故障的设备是很有用的。该特性降低了现场支持和热线的成本。PDIUSBD12 的方框图如图 9-21 所示，其中包含有 GoodLink™ 技术的简化示意图。

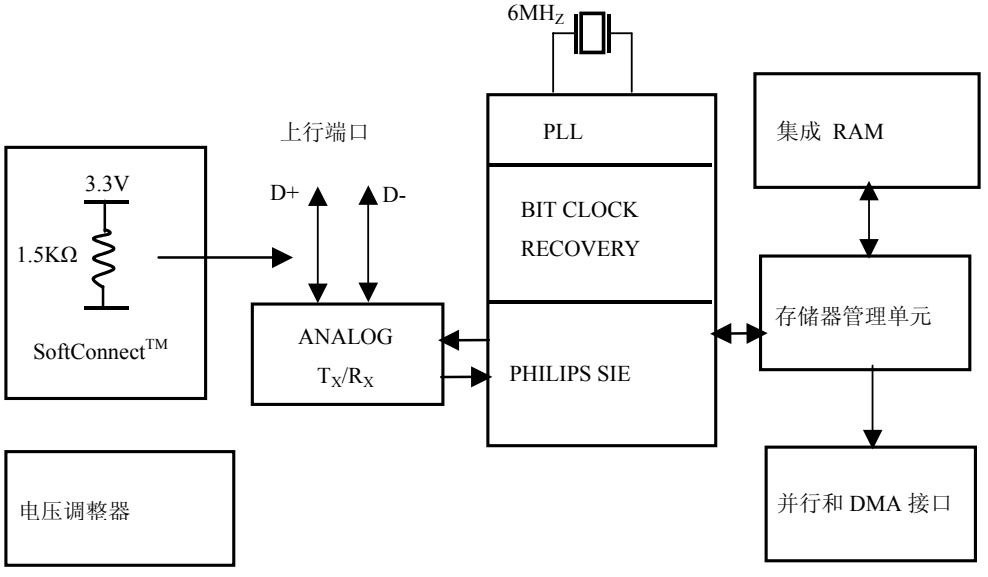


图 9-21　PDIUSBD12 的方框图

PDIUSBD12 将一个普通的并行接口定义成易用、快速而且可以与主流的微控制器直接连接的接口。对一个微控制器而言，PDIUSBD12 看起来就像一个带 8 位数据总线和一个地址位占用两个位置的存储器件。PDIUSBD12 支持多路复用和非复用的地址和数据总线，支持主端点与本地共享 RAM 之间直接存取的 DMA 传输；支持单周期和突发模式的 DMA 传输。这些概念在前面讲述 USBN9603 芯片时已经详细介绍过了。

PDIUSBD12 的文档里介绍了一种芯片与通用微控制器典型的连接方式，如图 9-22 所示。这种连接也可以作为不带微控制器的一类 USB 控制器芯片的连接参考。在该例中 ALE 接为低电平，表示一个非复用模式的地址和数据总线配置，PDIUSBD12 的 A0 脚与 80C51 的任意

一个 I/O 口相连，该端口控制 PDIUSBD12 的命令和数据状态。80C51 的多位地址和数据总线可直接与 PDIUSBD12 的数据总线相连。80C51 的频率输入可由 PDIUSBD12 的 CLKOUT 提供。

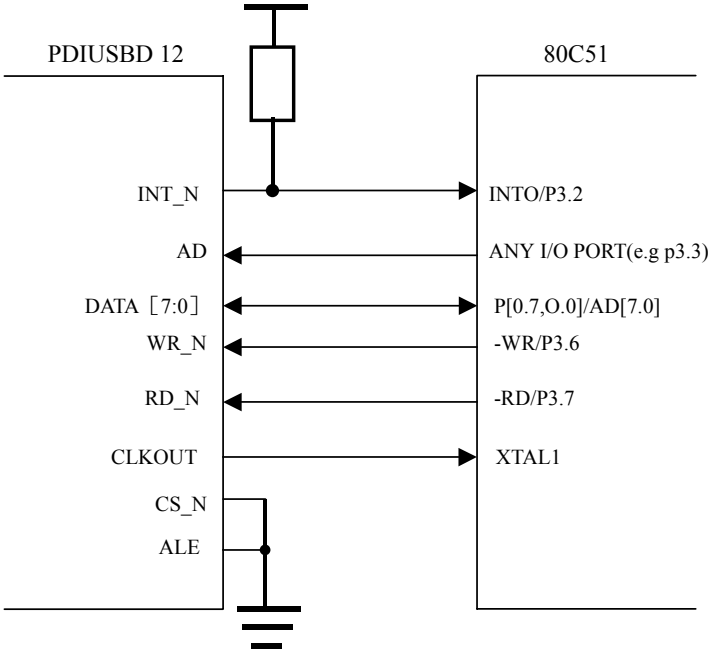


图 9-22　PDIUSBD12 与微控制器非复用模式的连接

PDIUSBD12 支持全速 1.5Mbit/s 的速率，它包括有一个双向的控制端点和 4 个数据端点。PDIUSBD12 的端点适用于不同类型的设备，例如图像打印机、海量存储器和通信设备。数据端点可通过 Set Mode 命令配置为 4 种不同的模式，分别为：

- 模式 0（Non-ISO 模式）非同步传输。
- 模式 1（ISO-OUT 模式）同步输出传输。
- 模式 2（ISO-IN 模式）同步输入传输。
- 模式 3（ISO-IO 模式）同步输入输出传输。

在不同的模式下，端点有不同的配置。这 4 种不同传输模式下各个端点的配置情况如表 9-5、表 9-6、表 9-7 及表 9-8 所示。

表 9-5　　模式 0（非同步模式）

端　点　数	端点索引	传输类型	端点类型	方　向	最大信息包规格（字节）
0	0	控制输出	默认	输出	16
	1	控制输入		输入	16
1	2	普通输出	普通	输出	16
	3	普通输入	普通	输入	16
2	4	普通输出	普通	输出	64
	5	普通输入	普通	输入	64

表 9-6　　模式 1（同步输出模式）

端　点　数	端点索引	传输类型	端点类型	方　向	最大信息包规格（字节）
0	0	控制输出	默认	输出	16
	1	控制输入		输入	16
1	2	普通输出	普通	输出	16
	3	普通输入	普通	输入	16
2	4	同步输出	同步	输出	128

表 9-7　　模式 2（同步输入模式）

端　点　数	端点索引	传输类型	端点类型	方　向	最大信息包规格（字节）
0	0	控制输出	默认	输出	16
	1	控制输入		输入	16
1	2	普通输出	普通	输出	16
	3	普通输入	普通	输入	16
2	4	同步输入	同步	输入	128

表 9-8　　模式 3（同步输入/输出模式）

端　点　数	端点索引	传输类型	端点类型	方　向	最大信息包规格（字节）
0	0	控制输出	默认	输出	16
	1	控制输入		输入	16
1	2	普通输出	普通	输出	16
	3	普通输入	普通	输入	16
2	4	同步输出	同步	输出	64
	5	同步输入	同步	输入	64

主端点（端点 2）在有些方面是比较特别的，它是进行吞吐大数据块的主要端点。同样地，它执行主机的特性以减轻传输大数据量的任务：

- 双缓冲，允许 USB 访问与本地 CPU 访问之间的并行操作以增加数据的吞吐量。缓冲区切换是自动处理的，这导致了透明的缓冲区操作。
- 支持 DMA 直接存储器访问操作，可以与对其他端点的正常 I/O 操作交叉进行。
- DMA 操作中的自动指针处理，在跨过缓冲区边界时不需要本地 CPU 的干预。
- 可配置为等时传输或非等时的批量和中断传输。

9.4　本章小结

在本章中首先介绍了一个通用 USB 控制器所包括的基本组件，然后就市面上常见的几款 USB 控制芯片的功能特点做了相应的介绍，并对它们进行了相应的对比。通过对本章的学习，读者可以根据自己的需求选择一个合适的 USB 控制芯片。

第 10 章　USB 设备开发概述

知识点：

- USB 设备开发步骤
- 控制芯片选择
- 硬件设计

本章导读：

通过前面几章节的学习，读者应该对 USB 协议以及 USB 设备如何工作都已经有了大体的了解。特别是第 4 章中的设备列举过程和第 5 章的控制传输给读者说明了 USB 设备如何开始与主机的通信工作。但是究竟如何开发一个 USB 设备呢？从本章开始将向读者介绍开发一个 USB 设备的具体步骤。

要开发一个 USB 设备并非一件容易的事情。一般可以分为 4 大部分的工作：硬件接口的设计、设备固件的设计、主机应用程序的设计和设备驱动程序的设计。每一部分的工作都需要有比较丰富的知识才能够完成。当然，用户并非一定要做全部的事情，这取决于用户所设计的 USB 设备的类型，以及用户的准备工作。比如用户所设计的 USB 外设属于 Windows 支持的一种类设备，那么就可以省去驱动程序的设计环节。除非用户想为自己的 USB 设备添加此通用类设备所没有的属性或功能，那么用户就必须为自己的设备开发一个过滤驱动程序（Filter Driver），但是这也要比从头开始写一个 USB 驱动容易得多。

10.1　准备工作

开发一个 USB 设备有很多要准备的工作，以下是所需的开发工具和一些必要的部件。

- 必须有一个支持 USB 的主机。
- USB 接口控制芯片及外围硬件。
- USB 的主机应用程序。该应用程序的开发与其他接口或总线应用程序的开发类似。可以使用 VC 或 VB 的开发环境进行开发。
- 设备固件程序。所谓固件程序就是固化在设备 USB 控制器内部程序存储器中或外部扩展的程序存储器之中的程序。开发此固件程序需要一个汇编或 C 语言编译器，用来将源程序编译成为微控制器可以执行的机器代码。C 语言比汇编语言开发起来要方便很多（在下一章中将会具体介绍）。
- 设备驱动程序。要编写设备驱动程序需要 DDK（Driver Development Kit）和 VC

编译环境。如果设备所要求的功能不是很复杂，用户也可以使用 DriverStudio 等开发工具，这些驱动开发工具可以根据用户的需要自动生成一个驱动程序框架，从而减轻用户的开发难度。

另外，用户还可以借助监控程序（Monitor）、协议分析器或其他的调试工具来帮助固件的开发。还可用 USBView、USBCheck 等工具帮助检查通信的错误。

10.2　开发步骤

做任何项目之前都需要有详细的计划，开发一个 USB 设备自然也不例外。一个好的开发计划可以使工作人员的工作有条不紊、事半功倍。开发计划包括有初步计划、硬件设计具体计划和软件设计具体计划。

10.2.1　初步计划

初步计划也可以说是一些准备工作，虽然很细小，但是却决定了以后的工作。初步计划包括以下几个方面：

（1）要判断清楚项目是否适合使用 USB 总线接口

这不是一个小问题，尽管 USB 有诸多的优点，但是它并非处处都满足用户的需要，而且 USB 开发难度也比较大。例如，如果用户的设备与主机之间传输的数据量不大，而且要求的速率也不高，那么串口就完全可以解决问题了。

（2）要决定 USB 设备的一些属性

比如：用户要求传输的数据量的大小、数据传输的速率，通过这些可以决定使用哪个版本的 USB 协议规范，用哪一种传输类型。除此之外，还要判断设备的耗电量，从而决定设备是总线供电方式，还是设备自供电方式。如果对设备的耗电量估计不足，而选择了总线供电方式，那么设备可能会因为无法从总线得到足够的能量而不能正常工作。

（3）用户还可以决定是否购买某个厂家的 USB 开发包，以节约更多的时间

这是因为一般 USB 设备的开发包内都带有驱动程序，还有固件的范例程序，以及主机的应用程序。尽管这些范例程序和应用程序不一定完全满足用户所开发的设备功能的需要，但是也是一个很好的借鉴。并且有的固件程序已经为用户编制了一个程序框架，在开发自己的设备功能时，用户只需要在框架中添加所需的程序代码即可。对于那些刚刚接触 USB 的新手，可以将开发板与主机相连，利用开发包所提供的应用程序来了解 USB 设备究竟是如何工作的，这样可以增强对 USB 设备的感性认识。

10.2.2　硬件计划

硬件的准备工作最重要的就是选择 USB 控制芯片。可以根据设备的功能要求和属性来选

择满足要求的 USB 控制芯片。芯片选择好之后，可以按照芯片手册所提供的典型接口电路来进行具体的设计。芯片的选择将会在下一小节中详细介绍。

10.2.3　软件计划

USB 设备软件部分的设计是个难点，它包括了 3 个方面：设备的固件、主机应用程序和驱动程序。

在 USB 设备与主机正常通信之前，主机必须检测并配置设备，所以用户必须编写固件程序代码，响应主机的控制请求命令，并能够提供描述符，使主机了解设备的功能，从而配置设备，开始与设备的正常通信。另外，固件还要完成正常的数据传输功能。

主机的应用程序可以提供给用户一个人机交互界面，能将用户的控制命令通过对驱动程序的调用传递给设备固件，并且可以通过界面了解设备的状态。如果用户设计的是鼠标、键盘等标准设备，那么可以利用一些通用的应用程序来访问设备。

驱动程序介于主机应用程序与设备固件之间。主机应用程序不能直接访问硬件，必须通过设备驱动程序将其请求转换为硬件识别的格式。如果用户所设计的 USB 设备属于 Windows 支持的类型，那么操作系统就嵌入了该设备的通用驱动程序。否则，用户只能自行开发驱动程序了。

10.3　控制器芯片的选择

现在生产 USB 控制芯片的厂商很多，市面上的 USB 控制器芯片也是琳琅满目，种类繁多。尽管这给了 USB 设备开发者很多的选择性，但同时太多的种类也使得用户无所适从，不知道该怎样选择适合自己需求的控制芯片。在上一章中介绍了几种常见的 USB 控制芯片的结构和功能，大家可以进行对比和参照。

选择一个 USB 控制芯片时，用户当然会考虑该芯片所包含的功能、价格以及开发的难易程度等因素。一个 USB 控制器开发的难易度由很多因素决定。其中，就芯片本身来说，带有微控制器内核的 USB 控制芯片要比需要外接微控制器的 USB 控制芯片的开发容易一些，但是价格也要贵一点。另外，还有如下几点需要考虑。

1. 芯片的文档

每一家芯片制造商都会提供芯片完整的说明文档，来解释芯片所提供的功能、具体的用法、管脚的说明，还有芯片内部具体的构架。如果是带有微控制器内核的芯片，文档还会提供芯片所支持的指令集。芯片的指令集定义了微控制器使用的每条汇编语句的语法以及使用方法。如果用户想用汇编语言来编写固件程序，那么这些描述指令集语法的文档会给用户提供很大的帮助。但是，如果用户更喜欢使用高级语言，例如 C 语言，那么这些指令集将没有什么作用。

总之，芯片文档对开发 USB 设备是至关重要的环节，有的文档还提供了典型电路的连接，以及某些功能的源代码。因此，一定要选择有详细芯片说明文档的 USB 控制器。

2. 固件范例

对于初次开发 USB 设备固件，特别是那些从未写过任何固件程序的用户来说，要完全地从零开始编写一个 USB 设备固件程序确实比较困难。所以，最好的办法就是能够找到一个功能相似的范例代码。通过阅读范例代码，了解如何开始编写固件程序。不同的芯片供应商和开发工具厂商所提供范例代码的数量和质量都是不同的。有时候，也可以在互联网上找到所需的范例代码。

3. 驱动程序

驱动程序是主机应用程序访问设备的桥梁。如果用户的设备属于 Windows 所支持的类，那么用户就可以不必自己编写驱动程序了。例如：HID 类的设备，主机的应用程序就可以使用标准的 API 函数与 Windows 所提供的 HID 驱动程序进行通信。有些芯片供应商也会提供应用程序的范例。比如：美国的国家半导体公司的 9603 芯片就提供了 HID 类的应用程序的范例。

有些芯片供应商也提供通用的驱动程序，用于主机与设备进行数据通信。此外它们还提供了一些特殊的驱动程序，用于设备完成一些特殊的功能。例如：Cypress 公司的 EZ-USB FX2 系列芯片。该芯片供应商提供了一个 EZ-LOADER 的特殊驱动程序，使得该芯片有一个独特的特性，那就是当设备接入主机之后，主机会自动将设备所需的固件加载到设备的程序存储器。这个特性对于固件程序的更新是非常方便的。

4. 调试工具

大多芯片的供应商都会提供调试工具和开发包以及开发板，用来帮助用户使用他们的芯片。开发板就是一个可以实现一些简单功能的 USB 设备，配合开发包中的应用程序和设备驱动程序使用。将开发板与主机相连后，开发板就可以与主机进行数据通信了。开发板的使用可以增加用户对 USB 的感性认识，并且缩短 USB 设备的学习和开发曲线。

开发包中包含的调试工具可以让用户监控或者控制程序的执行情况，以及查看执行结果。调试工具的功能包括：程序的单步执行、设置断点以及观察芯片的内部寄存器、内存的内容和变量的结果。通过调试程序，用户可以看到芯片内部的工作情况，增强对芯片，以及对芯片与主机之间的通信过程的理解。

5. 项目评估

以上几点都是单从芯片功能及所提供的帮助的角度来考虑的。除此之外，还应该从项目的功能需求情况来选择芯片。以下就是需要考虑的一些因素。

（1）所需的数据传输速率

设备的数据传输速率由几个方面的因素所决定：设备是低速、全速还是高速，使用的是何种传输类型，以及总线是否繁忙。设备的开发者无法决定总线的繁忙程度，因为它不知道总线带宽的占用情况。

如果用户的设备只需要低速的中断和控制传输，则可以选择相对比较便宜的低速芯片，同时也可以在电路板的设计上和电缆的选择上节省开发费用，但是低速设备只能支持每个事务传输 8 个字节。尽管 USB 规范定义低速设备的传输速率可以达到 1.5Mbit/s，但是这只是理论速率。

（2）所需端点的数目和类型

每一个端点都只能配置为支持一种传输类型和传输方向。如果设备只需要控制传输，那么就可以用默认端点了。但是如果想使用其他的中断传输、批量传输或等时传输的任何一种，则用户都需要另外配置这些端点。

（3）设备固件更新

很多程序存储器都使用的是 EPROM、OTP PROM 或是其他不容易擦除和重新写入的器件。这样要更新固件程序，则需更换新的芯片或是擦除重写烧入新的程序，特别是在程序调试的时候很不方便。Cypress 公司的 EZ-USB FX2 系列芯片支持一种简便的更新方式：设备在每次接入主机时，主机自动将程序代码加载到设备的程序存储器中，然后重新列举设备，这样，在调试固件程序时就很方便了。用户修改好程序之后，只需要重新插入设备就可以了。当用户的固件程序定性之后，还可以存储到串行 EEPROM 中，在设备接入主机时，从 EEPROM 中加载到设备。如果用户使用的是这种方式，在 USB Implementers Forum 网站上的 Device Class Specification for Device Firmware Upgrade 中规范描述了从主机加载固件程序的这种机制。

（4）电缆的选择

低速的电缆可以更细更有柔韧性，而且没有其他严格的要求，这也正是鼠标之所以是低速设备的原因。USB 2.0 规范建议使用屏蔽双绞线电缆。低速电缆只能在 3m 之内使用，高速电缆可以达到 5m 的距离。

（5）其他硬件特性和能力

其他硬件特性和能力包括是否需要一些通用的或是专用的 I/O 接口，程序和数据存储器的大小以及是否需要定时器等内容。例如，Cypress 公司的 EZ-USB FX2 芯片就提供了 GPIF（General Programmable Interface），它是一个 8 位或 16 位的并行接口，可供用户连接不同的外围设备，降低系统开销。

10.4　硬件设计

做了这么多准备工作以后，现在就可以开始具体的硬件设计了。这里以 Cypress 公司的 CY7C68013 为例来介绍 USB 接口电路的设计。其实，对于像 CY7C68013 这样带有通用微控制器内核的 USB 控制芯片来说，USB 接口电路的设计相对比较简单，只需要连接好 USB 接口的 4 根线就基本可以了。不像需要外接微控制器的 USB 接口芯片，还要考虑接口芯片和微控制器之间数据总线、地址线和控制信号的连接。芯片 CY7C68013 的一个标准的 USB 接口电路图如图 10-1 所示。

从图中可以看出这个接口电路很简单：左边的 USB-B 就是一个标准的 USB 接头，一般设备上用的是 B 型头。D+和 D-两根信号线通过 22（正负 5%）Ω 的电阻连接到芯片的 DPLUS 和 DMINUS 引脚，这两个电阻是用作阻抗匹配的。所有的 USB 控制器都需要这个匹配电阻，但是不同的芯片，所需的阻值不同，例如：Philips 公司的 PDIUSBD12 的匹配电阻是 18Ω。另外，CY7C68013 芯片内部还集成了一个 1.5KΩ 的上拉电阻，将 DPLUS 线通过开关上拉到 VCC，用来模拟 USB 设备的断连，这个功能主要用在该芯片独有的重新列举过程中。此电路中设备用的是外接电源，所以 VCC 就没有连接。如果想用总线供电，则需要将 VCC 接到一

个 DC-DC 变换芯片上，因为 CY7C68013 芯片是+3.3V 供电，而总线电压是+5V，所以还要经过一个电压变换，可以选用美信（Maxim）公司的 MAX882 进行电压的变换，该芯片可以将+5V 输入电压变换为+3.3V 输出。该芯片的典型连接如图 10-2 所示。

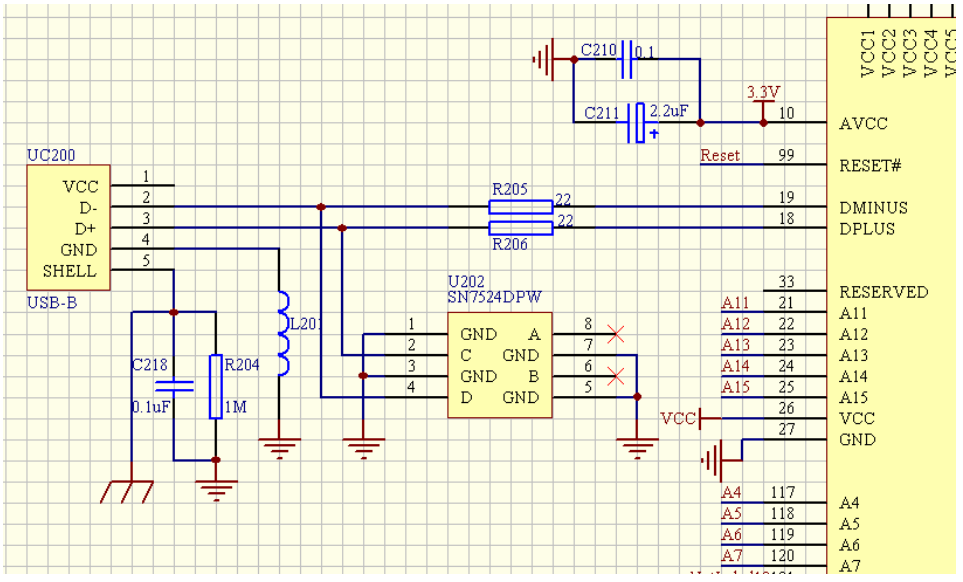


图 10-1　CY7C64613 USB 接口电路

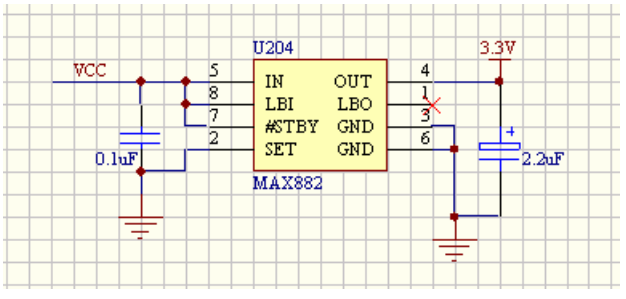


图 10-2　电压变换电路

通常情况下，按照上面介绍的进行连接就行了，但是在图 10-1 中看到还有一个芯片 SN75240。这个芯片是一个噪声抑制器，是专门为 USB 接口电路设计的，用以抑制 USB 数据线上的瞬时电气噪声。这个芯片并不是必须要加的，但是为了数据的可靠性，最好还是加上。另外，还要将 USB 接头的屏蔽层接地，这样可以更好地屏蔽电磁干扰，保证数据的可靠性。

还有一点需要介绍的就是I2C总线的连接。在前面介绍过，可以将设备的 VID 和 PID 存储在 I2C 接口的 EEPROM 中，在设备列举的时候向主机发送，因此，I2C 接口也是会使用到的。EEPROM 与 CY7C68013 的连接图如图 10-3 所示，A0、A1、A2 是 3 条地址线，可以用来标志连接在同一个 I2C 总线上的不同的

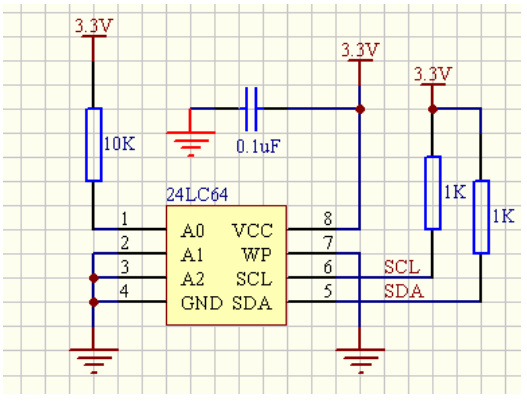


图 10-3　EEPROM 与 CY7C64613 的连接

EEPROM，SCL 是 I2C 的时钟信号，SDA 是 I2C 的数据线，这两根线都需要经过 1KΩ的上拉电阻接 3.3V 电源。

这样，USB 接口电路就算设计完了，剩下的工作就是用户根据需要设计自己的微控制器外围电路。由于 CY7C68013 集成的是 8051 内核，相信接触过微控制器的用户都很熟悉了。这里最大的不同就是，CY7C68013 提供的独立的 16 位地址总线和 8 位数据总线，而标准的 8051 则是地址数据线复用的，但是这并不影响外围电路的设计。另外，还有一个不同就是该芯片的读写信号/WR 和/RD 都需要通过一个 10KΩ的上拉电阻接到 VCC 上。

总的来说，USB 接口电路的设计在整个 USB 设备的开发过程中属于相对比较简单的工作，用户只需要按照芯片资料介绍的进行连接即可。

10.5　本章小结

本章主要讲述开发一个 USB 外设所需的准备工作，芯片如何选择以及 USB 外设的硬件电路设计。

第 11 章　固件设计（CY7C68013）

知识点：

- Keil C51
- 固件程序
- 重新列举 (Renumeration)
- 描述符
- 中断服务程序

本章导读：

前面介绍了很多种 USB 控制器芯片，有带微控制器内核的，还有需要外接微控制器的，但是无论选择那一种芯片，用户都需要编写辅助 USB 控制芯片与主机和外设中其他硬件电路通信的固件代码，因为硬件不可能做所有的工作，必须在固件程序的参与和控制下才能完成预期的设备功能。本章中就以 Cypress 公司的 CY7C68013 为例，介绍如何编写 USB 设备固件程序。另外，本章还简单介绍了 C51 语言的语法和使用，因为本章固件编写举例使用的就是 C51。

11.1　固件的工作

所有基于微控制器及其外围电路的功能设备的正常工作都离不开固件的参与，固件的作用就是辅助硬件，或者说是控制硬件来完成预期的设备功能。没有固件的参与和控制，硬件设备只是芯片的简单的堆砌，无法实现预期的功能。正如同一台没有安装操作系统的计算机（裸机）一样，是无法正常工作的。

USB 设备自然也不例外，尽管有些 USB 的控制芯片已经相当智能化了，但是也不可能包揽所有的事情。因此，用户必须编写固件程序来辅助硬件完成 USB 通信任务。前面介绍 Cypress 公司的 CY7C68013 时，已经详细地讲述了它的硬件所能完成的工作，余下的工作就要由用户编写固件程序来完成了。具体地说，包括以下几点：

- 初始化工作，包括设置一些特殊功能寄存器的初值以实现所需的设备属性或者功能，例如开中断、使能端点、配置端口等。
- 辅助硬件完成设备的重新列举（ReNumeration）过程，包括模拟设备的断开与重新连接，对接收到的设置包进行分析判断，从而对主机的设备请求作出适当的响应，完成主机

对设备的配置任务。

- 对中断的处理。
- 数据的接收与发送。
- 外围电路的控制。

当然，不同的外设实现的功能不同，固件代码的繁简程度也会有不同，但是对于实现 USB 通信这一部分功能，固件代码所作的工作是基本相同的。在后面的小节中以具体的实例来讲解如何编写固件程序，以实现 USB 通信的功能。

11.2 汇编与 C51 的比较

CY7C68013 内部集成了增强型的 8051 内核，它是与标准 8051 指令集二进制兼容的。因此，熟悉 8051 的用户就可以选择标准的 8051 编译器或者汇编器进行汇编程序的开发了。如果用户不熟悉 8051 架构，或是更喜欢使用高级语言，那么，也可以选择使用 C51 来编写固件代码。这两种语言各有优缺点，适用的环境也不同。

汇编语言程序最大的特点就是代码的执行效率很高，通常要高于高级语言程序。这里所谓的“效率”就是指程序的目标代码的长短和程序的运行速度。所以，在节省内存空间和提供程序运行速度为重要指标的场合（如实时过程控制），常常使用汇编语言来编制程序，而且汇编语言可以直接对硬件进行操作，且简单方便，编程思路很清晰。但是，汇编语言也有很多缺点：汇编语言程序的可读性和可移植性较差，并且采用汇编语言开发周期长，调试和排版都比较困难。

C 语言是一种通用的计算机程序设计语言，既具有一般高级语言的特点，又能直接对硬件进行操作，而且表达和运算能力也比较强。它的可读性和可移植性远胜于汇编语言程序。C51 就是在 ANSI C（标准 C）的基础上扩展了很多适合对微控制器硬件操作的属性，例如，增加了 sfr 和 sbit 关键字（数据类型），很容易实现对微控制器的特殊功能寄存器的操作。很多公司都有自己的 C 语言编译器。德国的 Keil Software 公司开发的 Keil C51 是一种专门为 8051 系列微控制器设计的高效率 C 语言编译器（如图 11-1 所示），它符合 ANSI C 标准，且生成的程序代码执行效率很高，所需的内存空间极小，几乎可以和汇编语言媲美。尽管 C51 具有很多优点，但是同其他任何一种语言一样，也有其自身的缺点：比如不能自动检测数组的边界，各种运算符的优先级别太多，某些运算符具有多种用途等，但是 C51 的优点远远超过了它的缺点，以前只能采用汇编语言来解决的问题现在大都可以改用 C 语言来解决了。而且 C 语言还提供了汇编语言的接口，在一些实时性要求很高的场合，用户也可以使用汇编和 C 语言的混合编程。

本章所举的固件程序范例就是用 C51 编写的，因此，在下一小节中，首先简单介绍一下 C51 的基本语法。不过，用户一定要有 C 语言的基础，再熟悉一些 C51 的特殊的语法，才能看懂固件程序。C51 的具体用法和编译环境的使用有很多的内容，有兴趣的读者可以参阅一些 C51 的专著。

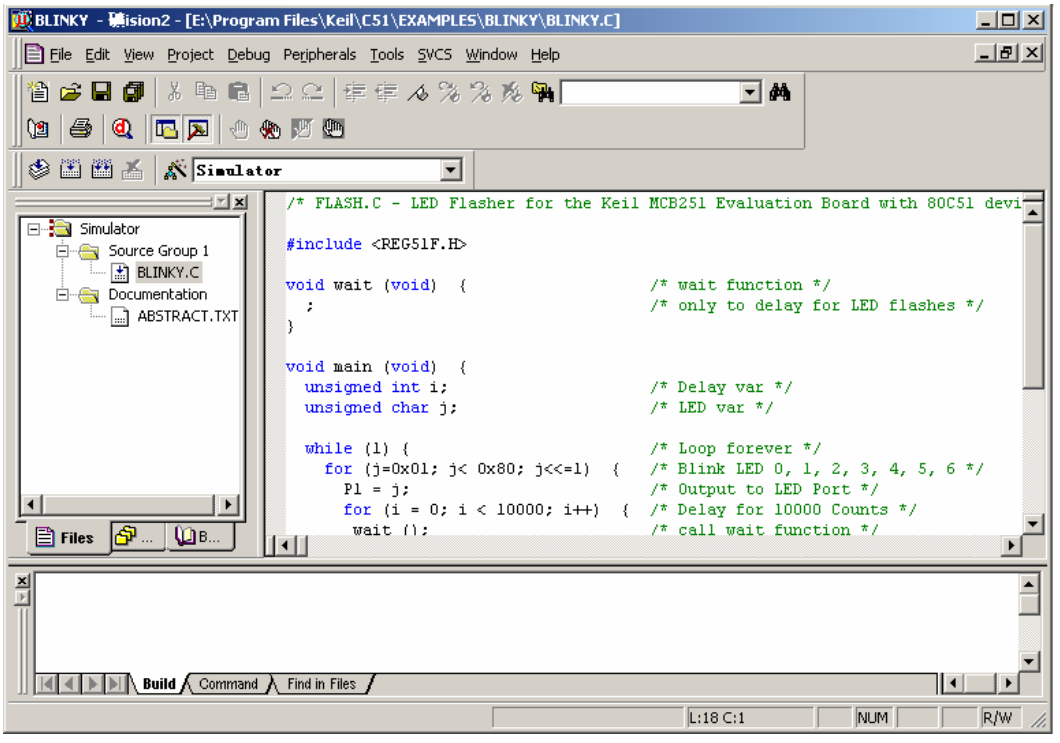


图 11-1　德国 Keil Software 公司的 C51 编译器 Keil Uvision2 开发编译环境

11.3　C51 程序设计基础

C51 是一种专门为 8051 微控制器设计的 C 语言编译器，符合 ANSI C，同时又做了一些特殊扩展以适应 8051 微控制器的特点。

11.3.1　标志符和关键字

任何程序设计语言都有标志符（Identifier）和各自的构成规则。例如：汇编语言中的标志符指的是指令语句中的标号和伪指令语句中的符号名称。C 语言的标准符是用来标志源程序中的函数、变量、常量数据类型等对象名称的。一个标志符由字符串、数字和下划线等组成，而且第一个字符必须是字母或下划线。C 语言是大小写敏感的，同一个字母由于大小写的不同可以代表不同的变量。一般情况下变量等标志符都采用小写，对于一些具有特殊意义的变量或常数才采用大写字母。例如 8051 的特殊功能寄存器 SCON、TCON、P0 等。

关键字是一类具有固定名称和特殊含义的特殊标志符，或者称为保留字。在 C 语言的源程序中通常不允许一般的标准符与关键字重名。ANSI C 标准一共定义了 32 个关键字，C51 在此基础上根据 8051 的特点又扩充了 20 个关键字。利用这些关键字可以实现对 8051 微控制器所有资源的有效操作。例如，关键字“sfr”和“sbit”可以很容易地实现特殊功能寄存器 SFR 的操作。关键字“_at_”可以将变量存入一个指定地址的存储空间。利用关键字“large”、

“compact”、“small”可以决定变量的存储模式，从而根据情况提高程序的运行效率。C51 扩展的关键字如表 11-1 所示。

表 11-1　　C51 扩展关键字

关键字名称	用途	描述
_at_	地址定位	变量在存储器中的绝对地址定位
alien	函数特性声明	声明与 PL/M51 兼容的函数
bdata	存储器类型声明	可位寻址的 8051 内部数据存储空间
bit	位变量声明	声明一个位变量或返回位类型值的函数
code	存储器类型声明	8051 程序存储空间
compact	变量存储模式	指定使用 8051 外部分页寻址的数据存储空间
data	存储器类型声明	直接寻址的 8051 内部数据存储空间
idata	存储器类型声明	间接寻址的 8051 内部数据存储空间
interrupt	中断函数声明	定义一个中断服务函数
large	变量存储模式	指定使用 8051 片外数据存储器
pdata	存储器类型声明	分页寻址的 8051 外部数据存储器
priority	多任务优先级声明	规定 RTX51 或 RTX51 Tiny 的任务优先级
reentrant	函数再入声明	再入函数定义
sbit	位变量声明	声明一个可位寻址的变量
sfr	特殊功能寄存器声明	声明一个 8 位的特殊功能寄存器
sfr16	特殊功能寄存器声明	声明一个 16 位的特殊功能寄存器
small	变量存储模式	指定使用 8051 内部数据存储空间
task	任务声明	定义实时多任务函数
using	寄存器组声明	定义使用哪个 8051 寄存器组
xdata	存储器类型声明	8051 外部数据存储器

11.3.2　数据类型

C 语言的基本数据类型包括有 char、int、short、long、float 和 double。对于 C51 来讲，short 类型和 int 类型相同，double 类型和 float 类型相同。C51 在标准 C 的基础上又扩展了几种数据类型：

1. 位类型 bit

位类型 bit 是 C51 的扩展数据类型，可以定义一个位类型变量，但是不能定义位指针，也不能定义位数组。

2. 特殊功能寄存器 sfr

特殊功能寄存器 sfr 也是 C51 扩展的一种数据类型，利用它可以定义 8051 微控制器内部所有 8 位特殊功能寄存器。sfr 类型数据占一个内存单元，范围在 0~255 之间。例如：sfr P0=0x80。

3. 16 位特殊功能寄存器 sfr16

sfr16 可以定义 8051 微控制器内部 16 位的特殊功能寄存器，sfr16 数据类型占两个内存单元，取值范围在 0~65535 之间。

4. 可位寻址类型 sbit

利用 sbit 关键字可以定义 8051 微控制器内部 RAM 中可位寻址区域，或是特殊功能寄存器中的可位寻址位。它的定义方法有 3 种：

（1）sbit 位变量名=位地址

这种方法将位的绝对地址赋给位变量，位地址必须位于 0x80~0xFF 之间。例如：

sbit OV=0xD2。

（2）sbit 位变量名=特殊功能寄存器^位次序

当可寻址位是特殊功能寄存器中的一位时，可以采用这种方法，“位次序”表示该位在特殊功能寄存器中的次序，取值范围介于 0~7 之间。例如：

sfr SCON = 0x98。

sbit TI = SCON^1。

（3）sbit 位变量名=字节地址^位次序

这种方法以一个常数作为基地址，该常数的取值范围在 0x80~0xFF 之间。“位次序”是一个 0~7 的常数。例如：

sbit TI = 0x98^1。

11.3.3 中断服务函数和寄存器组定义

C51 编译器支持在 C 语言源程序中直接编写 8051 微控制器的中断服务程序。从而减轻了采用汇编语言编写中断服务程序的复杂程度。为了满足这一要求，C51 对函数的定义进行了扩展，增加了一个关键字 interrupt。定义函数时加上这个选项就表示该函数为中断服务函数。定义中断服务函数的一般形式是：

函数返回类型 函数名（形参列表） [interrupt n] [using n]

例如：

```
void int0 ( ) interrupt 0 using 1
{
}
void timer1 ( ) interrupt 3 using 3
{
}
```

关键字 interrupt 后面的 n 是中断号，n 的取值范围是 0~31。C51 编译器在 8*n*+3 处产生中断向量。具体的中断号 n 取决于 8051 系列微控制器的信号，常用的中断源和中断向量如表 11-2 所示。

表 11-2　　常用中断号和中断向量

中断号	中断源	中断向量 8*n*+3
0	外部中断 0	0x0003
1	定时器 0	0x000B
2	外部中断 1	0x0013
3	定时器 1	0x001B
4	串口中断	0x0023

8051 系列微控制器片内 RAM 中有 4 个不同的寄存器组，每个寄存器组包括 8 个工作寄存器（R0~R7）。C51 扩展了一个关键字 using，专门用来选择 8051 微控制器内这 4 个不同的寄存器组。在定义函数时，using 是一个可选项，如果没有该选项，则由编译器自动选择一个寄存器组。

编写 8051 微控制器中断服务程序时应该遵循以下的规则：

- 中断服务程序不能进行参数传递，如果中断服务程序中包含任何参数声明，都将会导致编译错误。
- 中断函数没有返回值，因此即使为一个中断服务函数定义了返回值，也不可能得到正确的结果。建议在定义中断服务函数时使用关键字 void，以说明没有返回值。
- 在任何情况下都不要企图直接调用中断服务程序，否则将会产生编译错误。
- 如果中断服务程序中又调用了其他的函数，则被调用的函数所使用的寄存器组必须与中断服务程序相同。
- 在编译源程序的时候，可以使用控制命令 ＃pragma NOIV 来抑制编译器自动产生中断向量，从而使用户可以编写外部汇编程序来提供中断向量。

11.3.4　C51 中的寻址方式

在 C51 中有两种方法可以实现对存储器地址的直接操作，一种就是利用指针变量，另一种是利用 C51 提供的一组预定义宏，该宏定义文件为“ABSACC.H”，即绝对地址访问。利用它可以十分方便地实现对任意存储空间的直接操作。

指针是 C 语言中一个十分重要的概念，或者说是特点，正因为有了指针的引入，C 语言才能直接对硬件地址进行访问和操作。一个程序的指令、常量和变量等都要存放在存储器的存储单元中，存储器是按照字节来划分存储单元的，地址用于给每个字节存储单元赋予的一个编号或是索引。各个存储单元中所存放的数据就是这个存储单元的内容。指针就是专门用来确定其他数据类型地址的，因此，一个变量的地址就是这个变量的指针。例如，一个整型变量 I 存放在存储单元 0x10 中，那么，该存储单元地址 0x10 就是变量 I 的指针，而指针变量就是专门用来存放另一个变量地址的变量。

指针变量的定义和一般变量的定义类似，其一般形式如下：

数据类型 [存储器类型 1] * [存储器类型 2] 标志符

其中，“标志符”就是所定义的指针变量名。数据类型说明了该指针所指向的变量的类型。“存储器类型 1”和“存储器类型 2”是可选项。如果带有“存储器类型 1”的选项，则指针被定义为基于存储器的指针，该类型指针只占一个或两个字节，而且生产的代码速度较快。

另外一种直接操作存储器地址的方法是绝对地址访问。C51 在文件 ABSACC.H 中定义了几个宏，其函数原型如下：

```
# define CBYTE ((unsigned char volatile code * )0)
# define DBYTE ((unsigned char volatile idata * )0)
# define PBYTE ((unsigned char volatile pdata* )0)
# define XBYTE ((unsigned char volatile xdata * )0)
```

上述这些宏定义用来对 8051 系列微控制器的地址空间进行绝对地址访问，可以用作字节寻址。CBYTE 用于寻址 CODE 区、DBYTE 用于寻址 DATA 区、PBYTE 用于寻址分页的 XDATA 区（采用 MOVX @R0 指令）、XBYTE 也用于寻址 XDATA 区（采用 MOVX @DPTA 指令）。例如，下面的指令可以访问外部存储器地址 0x2000：XBYTE[0x2000]=10。另外，除了上面这 4 个可以进行字节单元绝对地址访问的宏定义，C51 还定义了 4 个可以进行字单元的绝对地址访问的宏定义：

```
# define CWORD ((unsigned char volatile code * )0)
# define DWORD ((unsigned char volatile idata * )0)
# define PWORD ((unsigned char volatile pdata * )0)
# define XWORD ((unsigned char volatile xdata * )0)
```

这些宏定义和前面的类似，只是它们所指向的数据类型是 unsigned int。通过灵活运用不同的数据类型，可以访问 8051 微控制器内所有的地址空间。

11.4　固件程序设计

Cypress 公司为了简化和加速用户使用 EZ-USB FX2 芯片进行 USB 外设的开发过程，特别设计了 CY7C68013 的开发板，并带有一个开发包，内含开发一个 USB 外设所必须的驱动程序、应用程序以及一个完整的固件程序的框架。这个框架可以执行 EZ-USB FX2 芯片的初始化、USB 标准设备请求的处理和 USB 挂起电源管理服务。用户只需要提供一个 USB 描述符表，添加其他端点接收和发送数据的通信代码，以及控制外围电路的程序代码。本章所举的固件程序范例就是基于这个框架而开发的。

11.4.1　固件程序规划

这个程序框架按照结构化的程序设计方法，将整个程序分为几个不同的功能模块，如图 11-2 所示。这样既易于编写调试，也增强了程序的可读性。

这个框架实现了一个简单的互操作的任务执行器。程序一开始执行，固件架构会执行以

下的步骤：

（1）初始化所有的内部状态变量，然后调用用户的初始化程序函数 TD_Init()，进行用户自定义变量的初始化工作。

（2）在完成全局变量的初始化工作之后，程序框架就会将 USB 接口初始化为未配置状态，并且使能中断。然后，程序就以一秒钟为间隔开始重新列举（ReNumerate）设备（也就是软件模拟设备断开和重新连接），直到端点 0 收到设置包为止。

（3）一旦检测端点 0 收到一个设置包，固件框架程序就开始启动执行一个互操作的任务分配器，这个任务分配器按照给定的顺序重复执行下面的任务：

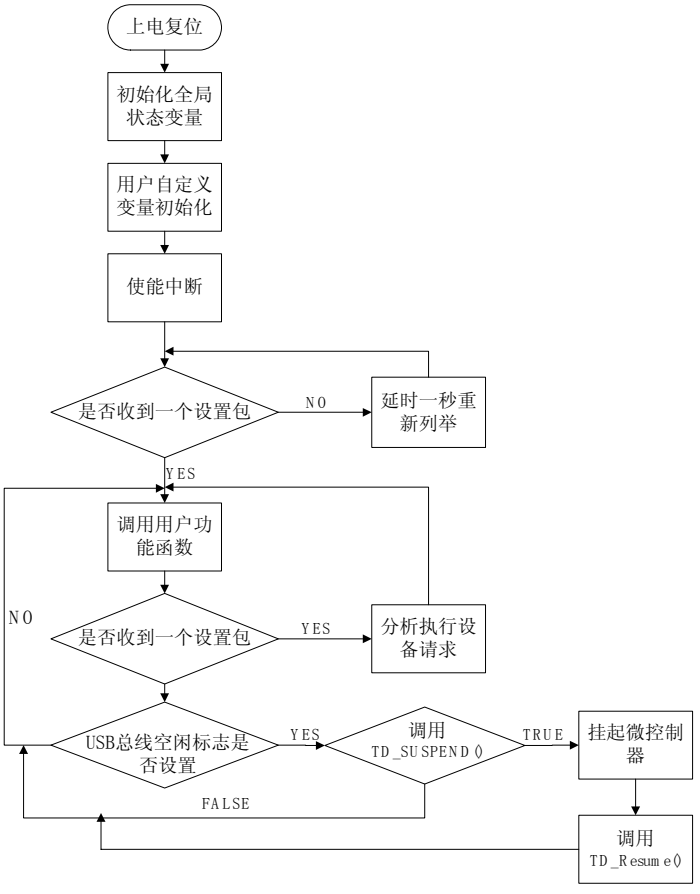


图 11-2　固件程序框架流程图

❑　调用函数 TD_Poll()。TD_Poll()函数应当包括一个执行用户外围功能的状态机，在从这个函数返回主函数之前，高优先级的任务应该首先被执行。然而如果没有成功地从这个函数返回的话，将会阻止固件程序对设备请求和 USB 挂起事件的响应。如果该函数内的任务需要大量的处理器时间，那么这些任务必须被分割到多个 TD_Poll()函数的调用中执行。

❑　判断是否有标准设备请求等待处理。如果有，任务处理器就分析收到的命令请求，并且作出适当的响应。

❑　确定 USB 核是否报告了 USB 挂起事件。如果有，则调用用户函数 TD_Suspend()；如果能够成功返回，任务处理器就测试一个恢复（Resume）事件；如果没有检测到恢复事件，

处理器就被设置到挂起模式；如果检测到一个恢复事件，便会调用用户函数 TD_Resume()。

标准和供应商特定的设备请求也是由程序框架分析和执行的。在缺省情况下，任务处理器将会按照 USB 规范的定义来响应给定的标准请求。此外，固件架构提供了许多钩子（HOOKS）程序，并且使其成为一个完整的分析器。这样就可以允许用户的代码处理或者覆盖掉特定的设备请求，从而为用户提供很大的灵活性，并减轻了用户设计固件程序代码的负担。

当设备收到设置数据时，中断服务程序就设置一个全局标志位为真，然后返回。当主程序检测到这个标志位为真后，就调用相应的处理函数进行分析和处理。这样，将原本应该由中断服务程序处理的复杂事务工作，交给专门的处理函数进行处理，减轻了中断服务程序的工作量。

11.4.2　主程序

主程序中所要做的工作其实就是在上一小节中所提到的初始化、重新列举和响应设备请求，下面就分别进行介绍。

1. *初始化*

主程序中一开始首先进行一些全局变量的初始化工作，这些全局变量是被作为状态标志位使用的，固件程序可以利用这些状态变量了解 USB 设备所处的状态与属性，并控制固件程序的运行。这些全局变量有以下几个。

- Sleep：表示 USB 芯片是否进入挂起状态的标志。当 USB 总线上 3ms 没有检测到传输事务，芯片就进入挂起状态，并产生挂起中断。在中断服务程序中，该标志位被设置为真，主程序检测到该标志位为真后，就会将处理器置为空闲模式（将 8051 的特殊功能寄存器 PCON 的第零位置高）。
- Rwuen：表示是否使能远程唤醒功能的标志。这个变量可以经过主机发送 Set_Feature()和 Clear_Feature()请求，进行设置和清除。
- Selfpwr：表示设备是否为自供电模式。此设置可在设备操作过程中动态地改变。
- GotSUD：标志端点 0 是否收到了设置数据。当端点 0 收到设置数据后就会产生中断。在中断服务程序中，该标志位被设置为真，表示收到了设置数据。当主机检测到该标志位为真时，就会调用专门的设置命令处理函数进行分析和处理。
- Configuration：表示当前所选择的配置号标志。主机可以发送 Set_Configuration()标志请求进行设置，如果该变量为零，则说明设备未配置。

初始化全局变量后，程序调用 TD_Init()函数。用户可以在该函数中添加自己的初始化代码。例如，这里可以根据设备功能的需要配置标志的 I/O 端口。

```
PORTSETUP =0x01;                //使能端口到 SFR（特殊功能寄存器）的映射
    PORTCCFG = 0xC4;            //打开读写信号线，以及 INT0 信号线
    OEC=0x3B;
    PORTACFG = 0x00;            //端口 A 配置为标志 I/O 端口
    OEA=0xFF;                   //输出使能
    PORTBCFG = 0x00;            //端口 B 配置为标志 I/O 端口
```

```
OEB=0x00;                    //端口 B 为三态
OED=0xFF;
OEE=0x00;
```

初始化工作还包括开中断、清除所有等待的 USB 中断请求等，这些工作都是通过设置 EZ-USB 的特殊功能寄存器实现的。

2. 重新列举

当所有的初始化工作做完之后，固件程序将会检测是否收到设置数据（GOTSUD 标志位是否为真），如果没有，程序就会以一秒钟为间隔，软件模拟设备的断开和重新接入，直到收到设置数据为止。

EZ-USB 提供了 USBCS 寄存器，可以进行 USB 相关属性的控制和反映。软件模拟设备的断开和重新连接就是靠设置这个寄存器来实现的。该寄存器中的 b3 位 DISCON 可以设置引脚 DISCON＃的状态，这一位通常被设置为 0，而 DISCON#引脚的状态与该位相反。b2 位 DISCOE（Disconnect Output Enable）可以控制引脚 DISCON 的输出缓冲器。当 DISCOE=0 时，该引脚浮空；当 DISCOE=1 时，该引脚将会被驱动为与位 DISCON 的状态相反的状态。

以下这几行程序就可以实现软件模拟设备的断开和重新接入过程。

```
USBCS |=bmDISCON;          //设置 USBCS 寄存器的 b3 位为高，驱动芯片引脚
                           //ISCON#为低电平
WRITEDELAY()               //两次读写寄存器之间的延时恢复
USBCS & = ~bmDISCOE        //将引脚 DISCON#浮空，也就是与 USB 端口的 D+线断开
WRITEDELAY()
USBCS |=bmRENUM            //将 USBCS 寄存器的 b1 位 RENUM 设置为高。
WRITEDELAY()
EZUSB_Delay(1500)          //设备断联与重新连接之间的延时
USBCS &=~bmDISCON          //驱动芯片引脚 DISCON#为高电平
WRITEDELAY()
USBCS |=bmDISCOE           //释放引脚 DISCON#的三态，重新连接设备
```

上面的范例中，WRITEDELAY( )是自定义的一个延时程序。芯片资料规定在两次读写寄存器之间要有一定的延时，以满足寄存器的读写恢复时间，这个延时时间大约是两个指令周期。

EZUSB_Delay(1500) 也是一个延时程序，因为在断开与重新连接之间如果没有足够的延时，一些集线器或者主机可能不会检测到一个设备的断连事件。程序中的 bmDISCON 和 bmDISCOE 都是定义的位屏蔽变量，用于屏蔽一个寄存器中的某些位。它们都是在头文件中定义的，这样改动起来比较方便，也增加了程序的可读性。

范例中有一条语句：USBCS |=bmRENUM，其作用是将 USBCS 寄存器的 b1 位 RENUM 设置为高，即将 USB 的控制权交给 8051 内核。该位控制着由哪一个实体，即 USB 核还是 8051 微控制器来处理 USB 请求。当 RENUM 被设置为低时，USB 核处理所有的设备请求；当 RENUM 被设置为高时，8051 处理除了 Set_Address()之外的所有设备请求。USB 核会在两种情况下自动设置 RENUM=1：

- 完成一个“B6”引导加载（EEPROM 的第一个字节为 B6）。

- 没有使用 I2C 兼容的 EEPROM，而使用的是外部存储器（EA=1）。

3. 任务处理器

下面，固件程序就进入了任务处理器，任务处理器其实就是一个循环过程。程序首先会轮询用户设备是否有需要执行的任务等待处理。如果有，程序就会按照从高优先级到低优先级的顺序来执行这些任务。接下来程序会检测是否收到设置数据（通过 GOTSUD 标志位的状态）。如果检测到设置包，就调用相关的请求命令处理函数 Setup_Command()进行处理，并作出适当的响应。下面的范例就是任务处理器的部分代码：

```
While(TRUE)                                  //循环，该标志在中断处理例程中设置
  {
if(GotSUD)                                   //判断是否收到设置数据
        {
                SetupCommand()               //执行设置命令
                GotSUD = FALSE               //清除 GOTSUD 标志位
        }
      ……
TD_POLL();
  }
```

固件程序会通过 TD_POLL()函数不停地轮询设备，因此，用户可以将所要完成的外围电路功能代码按照执行的优先级放入该函数中。

> 注意：不能添加耗时过多的程序代码，因为如果耗时过多，则会影响到程序对设备请求的响应。

11.4.3 描述符定义

尽管有了固件程序的框架，但是用户必须为自己的设备定义描述符，从而使主机通过描述符来识别和配置设备。用户可以将所有的描述符定义在一个文件中，包括设备描述符、配置描述符、接口描述符、端点描述符和字符串描述符。而且，这些描述符表要按照一定的顺序进行存放，以便于寻址。一般按照下面的顺序进行列表：

```
设备描述符
    配置 1 描述符
        接口 1 描述符
                端点 1 描述符
                端点 2 描述符
                ……
         接口 2 描述符
                端点 1 描述符
                端点 2 描述符
                ……
        ……
    配置 2 描述符
```

……
字符串描述符 1
字符串描述符 2
……
类描述符 1
类描述符 2
……
空描述符

每一个描述符表数据内容都按照 USB 规范或设备类规范进行定义。所有的双字节数据值要按照 little_endian 的格式进行定义，因此，如果供应商 ID 是 0x0547，那么就要表示为 0x47、0x05。接下来，举例说明如何定义各种描述符表，以下描述符定义的范例都是笔者在实际项目开发过程中所编写的，因此对于读者定义自己的描述符有实际的指导意义。这些描述符定义是用汇编语言编写的，这样有利于进行准确的地址定位。当然也可使用 C51 来进行编写，相关内容稍后也会有简单的讲解。

1. 设备描述符

设备描述符用以描述设备的整体属性，一个设备仅有一个设备描述符。

```
DeviceDscr:     db    deviceDscrEnd-DeviceDscr ; 设备描述符长度 18 个字节
                db    DSCR_DEVICE              ; 描述符类型，DSCR_DEVICE=0x01
                dw    0002H                    ; 符合 USB 规范 2.0
                db    00H                      ; 设备类代码
                db    00H                      ; 设备子类代码
                db    00H                      ; 设备协议代码
                db    64                       ; 端点 0 的最大包大小
                dw    4705H                    ; 厂商 ID（VID）
                dw    1208H                    ; 产品 ID（PID）
                dw    0100H                    ; 设备版本号 0.01
                db    1                        ; 制造商的字符串描述符索引
                db    2                        ; 产品的字符串描述符索引
                db    0                        ; 设备序号的字符串描述符索引
                db    1                        ; 可能的配置数目
deviceDscrEnd:
```

2. Device_Qulifier 描述符

这个描述符是高速设备所独有的描述符。同时，支持全速和高速的设备，必须有一个 device_qulifier 描述符。当设备转换速度的时候，设备描述符中的某些字段可能改变。device_qulifier 描述符存储目前不使用的速度所具有的字段设置。设备描述符和 device_qulifier 描述符中的字段数值，根据不同的速度来做交替。

Device_qulifier 描述符的字段和设备描述符的相同，不过，device_qulifier 描述符描述的是具有目前不在使用中的速度的设备配置。当速度改变时，设备描述符的 VID、PID、设备版本号、制造商等字段的值是不会改变的。所以，device_qulifier 描述符不需要包括这些字

段。

```
DeviceQualDscr:
        db    DeviceQualDscrEND－DeviceQualDscr ；描述符长度
        db    DSCR_DEVQUAL          ；描述符类型，DSCR_DEVQUAL=0x06
        dw    0002H                 ；符合 USB 规范 2.0
        db    00H                   ；设备类代码
        db    00H                   ；设备子类代码
        db    00H                   ；设备协议代码
        db    64                    ；端点 0 的最大包大小
        db    1                     ；可能的配置数目
        db    0                     ；保留待用
DeviceQualDscrEND:
```

3. 配置描述符

配置描述符用以描述一个单一的设备配置。一个设备所支持的所有配置都是相互排斥的。多个配置描述符可以用来方便地实现多操作模式，不同的操作模式从 USB 总线上可以提取不同数量的电流。一个配置包括一个或者多个接口。

```
ConfigDscr:   db   ConfigDscrEnd-ConfigDscr ；配置描述符长度 9 个字节
              db    DSCR_CONFIG           ；描述符类型，DSCR_CONFIG=0x02
              db    39H                   ；描述符总长度，低字节（LSB）
              db     00                   ；描述符总长度，高字节（MSB）
              db     1                    ；接口数目
              db     1                    ；配置值
              db     0                    ；配置的字符串描述符索引
              db     10100000b            ；配置的属性
              db     0                    ；最大电流
ConfigDscrEnd:
```

4. 接口描述符

接口描述符描述一个单一的逻辑设备接口。一个给定的配置所包含的接口可能会同时作用或者相互排斥。接口号数和交替设置值决定接口之间如何共存。有相同接口号数和不同可替换设置值的接口是相互排斥的；有不同接口号的接口之间是共存的，一个接口有零个或者多个端点。

```
IntrfcDscr:   db IntrfcDscrEnd-IntrfcDscr ;   接口描述符长度 9 个字节
              db DSCR_INTRFC                ；描述符类型，DSCR_INTRFC=04H
              db    0                       ；接口号数为 0
              db    0                       ；替换设置号数
              db    2                       ；除了端点 0 外，支持的端点数目为 2
              db    0ffH                    ；接口类代码，制定为厂商定义的类
              db    00H                     ；接口子类代码
```

```
                db    00H                    ; 接口协议代码
                db    0                      ; 此接口的字符串描述符索引
IntrfcDscrEnd:
```

5. 端点描述符

端点描述符描述一个单一的 USB 端点，端点描述符的参数包括端点号数、类型、包大小等。一个给定接口所包含的端点是共存的。由上面的接口描述符可知，该接口支持除了端点 0 以外的两个端点，因此，这里定义了两个端点描述符。

```
EpO1Dscr:       db    EpO1DscrEnd-EpO1Dscr   ; 端点描述符长度 7 个字节
                db    DSCR_ENDPNT            ; 描述符类型，DSCR_ENDPNT=05H
                db    01H                    ; 端点 1，方向为 OUT
                db    ET_BULK                ; 批量传输类型，ET_BULK =02H
                db    40H                    ; 支持的最大数据包为 64 Byte
                db    00H
                db    00H                    ; 最大延时或轮询时距，对于全速的批量
                                             ; 端点可以忽略
EpO1DscrEnd:

EpI1Dscr:       db    EpI1DscrEnd-EpI1Dscr; 端点描述符长度 7 个 Byte
                db    DSCR_ENDPNT            ; 描述符类型，DSCR_ENDPNT=05H
                db    81H                    ; 端点 1，方向为 OUT
                db    ET_BULK                ; 批量传输类型，ET_BULK =02H
                db    40H                    ; 支持的最大数据包为 64 Byte
                db    00H
                db    00H                    ; 最大延时或轮询时距，对于全速的批量
                                             ; 端点可以忽略
EpI1DscrEnd:
```

6. 字符串描述符

字符串描述符包括一个可以被USB设备驱动程序或用户应用程序使用的UNICODE字符串，它们的索引是由列表的顺序隐含表示的。因此，列表的第一个字符串就是字符串 0，第二个字符串就是字符串 1，依此类推。字符串 0 包括设备所支持的语言代码。

字符串描述符 0 定义如下：

```
StringDscr0:    db    StringDscr0End-StringDscr0 ; 字符串描述符长度
                db    DSCR_STRING            ; 描述符类型，DSCR_STRING=03H
                db    09H,04H                ; 设备所支持的语言代码，0x0409
                                             ; 表示的是英语
StringDscr0End:
```

制造商的字符串描述符如下：

```
StringDscr1:    db   StringDscr1End-StringDscr1 ; 制造商的字符串描述符长度
                db   DSCR_STRING               ; 描述符类型
                db   ' C ', 00                 ; 制造商名称，因为这里使用的是
                                               ; Cypress 公司的芯片，因此，这里
                                               ; 的字符串为 Cypress
                db   ' y ',00
                db   ' p ', 00
                db   ' r ',00
                db   ' e ', 00
                db   ' s ',00
                db   ' s ',00
StringDscr1End:
```

产品的字符串描述符如下：

```
StringDscr2:    db    StringDscr2End-StringDscr2; 产品的字符串描述符长度
                db    DSCR_STRING              ; 描述符类型
                db    'U',00                   ; 产品名称，此名称用户可以自行决
                                               ; 定，此处使用的产品名为 USB Device
                db    'S',00
                db    'B',00
                db    ' ',00
                db    'D',00
                db    'e',00
                db    'v',00
                db    'i',00
                db    'c',00
                db    'e',00
StringDscr2End:
```

空描述符，表示描述符表的结束：

```
NullDscr:  dw    0000H
        end
```

以上介绍了用汇编语言来定义描述符，当然也可以使用 C51 来编写描述符表。使用 C51 定义描述符时，要用到结构型变量。首先按照规范定义各个描述符类型，然后才能定义描述符变量。下面，以设备描述符为例，介绍如何用 C51 来定义描述符，其他几个标准描述符定义与此相同。

首先，定义设备描述符结构类型：

```
typedef   struct   _USB_DEVICE_DESCRIPOR {
            unsigned char     bLengt1h;
```

```
    unsigned char     1bDescriptorType;
    unsigned short    bcdUSB;
    unsigned char     bDeviceClass;
    unsigned char     bDeviceSubClass;
    unsigned char     bDeviceProtocol;
    unsigned char     bMaxPacketSize0;
    unsigned shotrt   idVendor;
    unsigned shotrt   idProduct;
    unsigned shotrt   bcdDevice;
    unsigned char     iManufacture;
    unsigned char     iProduct;
    unsigned char     iSerialNumber;
    unsigned char     bNumConfigurations;
} USB_DEVICE_DESCRIPTOR，*PUSB_DEVICE_DESCRIPTOR
```

定义了设备描述符结构类型后，就可以定义描述符变量了：

```
code   USB_DEVICE_DESCRIPTOR   DeviceDescr =
{                                          //此设备描述符位于代码段
   sizeof (USB_DEVICE_DESCRIPTOR) ，       //设备描述符长度
   USB_DEVICE_DESCRIPTOR_TYPE ，           //设备描述符类型 0x01
   SWAP(0x0200) ，                         //USB 规范版本（高低字节交换）
   0 ，                                    //设备类代码
   0 ，                                    //子类代码
   0 ，                                    //协议代码
   0x40，                                  //端点 0 的最大包大小 64 个字节
   SWAP(0x0547) ，                         //厂商 ID
   SWAP(0x0812) ，                         //产品 ID
   SWAP(0x0001) ，                         //设备版本号为 0.01
   1 ，                                    //制造商的字符串描述符索引
   2 ，                                    //产品的字符串描述符索引
   0 ，                                    //设备序号的字符串描述符索引
   1                                       //可能的配置数目为 1
 }
```

其他的配置描述符、接口描述符、端点描述符和字符串描述符的定义也与此类似。这里由于篇幅的限制就不再赘述了。

11.4.4　响应设备请求

EZ-USB FX2 芯片定义了几个特殊寄存器，用以辅助用户固件程序响应设备请求，并向主机传送数据。当设备收到设置包时，USB 核会自动将设置数据放入 SETUPBUF 缓冲区中。SETUPBUF 是一个 8 字节的缓冲区，用于存放刚刚收到的设置包数据。用户只需要从这个缓冲区中读取设置数据，进行分析就可以判断是什么样的请求了，这大大减轻了固件的工作量。

另外，固件程序可以使用 SUDPTR 寄存器向主机返回描述符。SUDPTR 其实是一个指针，它分为高字节 SUDPTRH（Setup Data Pointer High）和低字节 SUDPTRL（Setup Data Pointer），用户可以将它指向主机所请求的描述符的首地址。例如，当 EZ-USB FX2 芯片的端点 0 收到一个“Get_Descriptor()”请求，用户可以将事先定义好的设备描述符的首地址赋予这个寄存器，接下来 USB 核就会处理多个数据包的 IN 传输了。

描述符表可以存放在内部的程序或数据存储器中，也可以放在未用的端点 0~7 的 RAM 中。但是，SUDPTR 只能寻址内部存储空间。如果利用 SUDPTR 寄存器来传送描述符，描述符表的数据就必须位于内部程序或者数据存储空间中。

任何使用 EZ-USB FX2 芯片中 SUDPTR 指针来传送 IN 数据的主机请求，都必须在设置数据包的字节 6（wLengthL）和字节 7（wLengthH）中指明要传输的数据字节数目。在所有的标准设备请求中（例如：“Get_Descriptor”），这两个字节都是按照 USB 规范预先设定在长字节中的。如果有供应商特定的请求想要使用设置数据指针来传输大块的数据，它们必须在字节 6 和 7 中包含这个预定义的长度字段，告诉 USB 核有多少数据需要使用设置数据指针来传输。

定义好了描述符表，当收到设置命令后，用户只需要用一个开关语句来判断请求类型，然后分别进行处理。下面就是相应设备请求的范例代码：

```
switch (SETUPDAT[1])                          //判断请求类型
{
case   SC_GET_DESCRIPTOR                      //0x06 Get_Descriptor()设备请求
switch (SETUPDAT[3])
 {
 case   GD_DEVICE :                           // 0x01，设备描述符
   SUDPTRH = MSB(pDeviceDscr) ;               // pDeviceDscr 是设备描述表的首地
                                              //址，是一个双字节指针，MSB()用
                                              //于取这个指针的高字节
   SUDPTRL = LSB(pDeviceDscr) ;               // LSB()用于取这个指针的低字节
        break;

        case GD_DEVICE_QUALIFIER:             //Device_qulifier 描述符
         SUDPTRH = MSB(pDeviceQualDscr) ;
         SUDPTRL = LSB(pDeviceQualDscr)   ;
        break;

        case   GD_CONFIGURATION :             // 0x02，配置描述符
        SUDPTRH = MSB(pConfigDscr);           // pConfigDscr 是配置描述符表的首地址
```

```
            SUDPTRL = LSB (pConfigDscr);          //此处需要注意的是，如果有不止一个配
                                                  //置描述符表，则应判断请求是哪个配置
                                                  //述符可根据设置数据的 SETUPDAT[2]中
                                                  //的内容来判断请求是第几个配置描述符。
                                                  //如果没有所请求的配置描述符，则按照
                                                  //USB 规范定议将该端点挂起：EP0CS |=
                                                  //bmEPSTALL
          break;

case    GD_STRING :                   //0x03，字符串描述符，
                                      //以下采用另一种手动发送数据的方法，也就是直
                                      //接向端点的缓冲区发送数据，而不使用 SUDPTR
                                      //寄存器。此方法不仅适用于端点 0，而且其他端
                                      //点发送数据也只能如此。
          STRINGDSCR *sdp;            //定义一个字符串结构类型指针
char    len;                          //定义一个字节型长度变量
sdp = dscr_ptr;                       //此处和配置描述符一样有可能有多个字符串描述
                                      //符，所以需判断是哪一个字符串描述符，判断的
                                      //程序比较复杂，此处没给出。dscr_ptr 就是所请求
                                      //的字符串描述符的首地址
 len = sdp->length;                   //取描述符表中的长字段

   if (len > SETUPDAT[6])             //如果描述符表的长度大于设置数据包中的长度字
                                      //段，那么就将长度限制在所请求的长度范围内

        len = SETUPDAT[6];
       while (len)
        {
          for (i=0; i<min(len,64); i++)
          *(IN0BUF+i) = *((BYTE xdata *)sdp+i);
                                      //循环将字符串描述符数据向端点 0 缓冲区发送最
                                      //多 64 个字节
          IN0BC=min（len，64 );       //将装入 IN0BUF 的数据字节数目写入
                                      //IN0BC 寄存器，写入这个寄存器将会通
                                      //过设置端点的 BSY 位而装备（arm）端点
          len=len－min(len，64);      //从长度变量中减去已经发送的字节数目
          while(EP0CS & 0x04)
;                                 //等待端点将该次数据发送完，即判断 EP0CS}寄存
                                  //器的 BUSY 位是否跳变，如果没有发送完，程序
                                      //会不断执行空语句循环
```

```
            IN0BC=0;              //表明所有的数据发送完毕
            EP0CS=0x02;           //清除 EP0CS 寄存器的 HSNAK 位
          break ;

            default :
            EZ_STALL_EP0 () ;     //挂起端点 0
}
break ;

case  SC_SET_CONFIGURATION:    //0x09 Set_Configuration()请求
Configuration = SETUPDAT[2] ;  //Configuration 是用户定义的一个字节型的全
                               //局变量，用于保存当前的配置。
……
```

其他请求也应用类似的处理，只是不同的请求，要求有不同的响应而已，都要按照 USB 规范来处理，在此就不再举例了。

11.4.5　中断处理

EZ_USB FX2 除了包含 8051 的中断外，还新增加了 8 个中断。USB 核提供了 4 个中断组类型：

- Wakeup：在 EZ_USB FX2 芯片检测到 USB 挂起，8051 进入空闲状态后，USB 核会响应引脚 WAKEUP#上的一个外部信号，或者是 USB 总线重新活动都会引起这个中断的产生。该中断可以重起芯片的晶振，恢复 8051 的操作。
- USB 信号：这个中断包括 18 个与端点相关的中断，还有 4 个不属于特定端点的中断（SOF、Suspend、USB Reset、ERRLIMIT），以及 4 个控制传输的中断（HISPEED、EP0ACK SUTOK、SUDAV），这些中断共享一个 USB 中断 INT2 。
- I2C：兼容传输（INT3）。
- 从 FIFO 标志（Slave FIFO）（INT4）。

FX2 提供 28 个与 USB 相关的中断，其中一个中断“Resume”有自己单独的中断，其他 27 个共享一个 USB 中断 INT2。这 27 个共享 USB 中断的 USB 请求如表 11-3 所示。

表 11-3　　USB 中断源

优先级	INT2VEC 值	中断源	注释
1	00	SUDAV	收到 Setup 数据
2	04	SOF	（微）帧起始符
3	08	SUTOK	收到 Setup 令牌包
4	0C	SUSPEND	USB 挂起请求
5	10	USB RESET	总线复位
6	14	HISPEED	进入高速操作
7	18	EP0ACK	FX2 以 ACK 响应控制传输

8	1C	reserved	
9	20	EP0-IN	EP0-IN 准备好装载数据
10	24	EP0-OUT	EP0-OUT 收到数据
11	28	EP1-IN	EP1-IN 准备好装载数据
12	2C	EP1-OUT	EP1-OUT 收到数据
13	30	EP2	IN：缓冲区可用 OUT：缓冲区有数据
14	34	EP4	IN：缓冲区可用 OUT：缓冲区有数据
15	38	EP6	IN：缓冲区可用 OUT：缓冲区有数据
16	3C	EP8	IN：缓冲区可用 OUT：缓冲区有数据

续表

优先级	INT2VEC 值	中断源	注释
17	40	IBN	IN－BULK－NAK
18	44	reserved	
19	48	EP0PING	EP0-OUT 收到 PING 协议并响应以 NAK
20	4C	EP1PING	EP1-OUT 收到 PING 协议并响应以 NAK
21	50	EP2PING	EP2-OUT 收到 PING 协议并响应以 NAK
22	54	EP4PING	EP4-OUT 收到 PING 协议并响应以 NAK
23	58	EP6PING	EP6-OUT 收到 PING 协议并响应以 NAK
24	5C	EP8PING	EP8-OUT 收到 PING 协议并响应以 NAK
25	60	ERRLIMIT	总线错误超过了预先的限制
26	64	reserved	
27	68	reserved	
28	6C	reserved	
29	70	EP2ISOERR	ISO EP2 OUT PID 顺序错误
30	74	EP4ISOERR	ISO EP4 OUT PID 顺序错误
31	78	EP6ISOERR	ISO EP6 OUT PID 顺序错误
32	7C	EP8ISOERR	ISO EP8 OUT PID 顺序错误

EZ-USB FX2 提供了称为自动向量的技术来帮助用户进行中断处理，如图 11-3 所示。

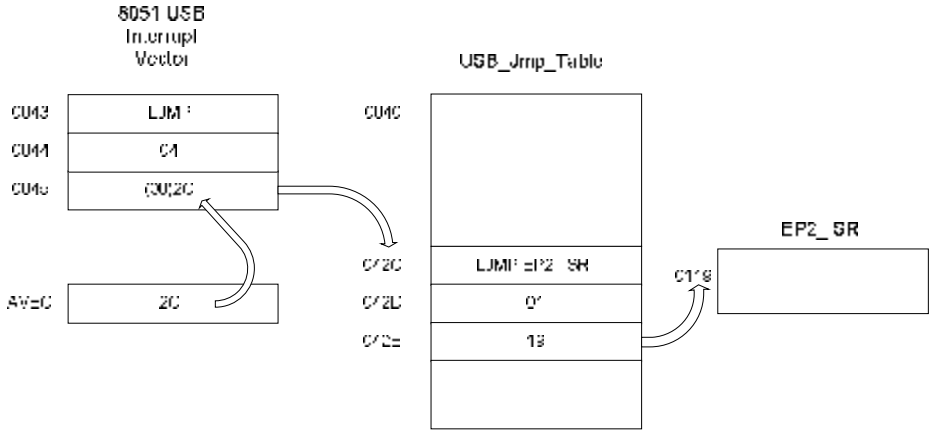


图 11-3　自动向量机制示意图

USB 核划分 USB 中断的优先级，并且在寄存器 AVEC 中建立一个自动向量。如果有两个中断同时发生，USB 核会根据优先级决定在 AVEC 寄存器中指示哪一个中断。如果 8051 使能了自动向量机制，那么寄存器 AVEC 中的字节将会替换 8051 程序存储器中地址 0x45 的内容，这将会导致 USB 中断自动寻址到不同的中断源。因此，用户应建立一个中断跳转表，放置一系列的跳转指令来引导程序跳转到相应的中断服务例程中。下面是一个 C51 和汇编混合编程而建立的中断跳转表：

```
CSEG AT 43H                    //定义代码段起始地址 0x43
USB_AutoVector equ   $+2       //USB_AutoVector 为一个全局变量，等于当前地址+2
ljmp    USB_Jump_Table         //寄存器 AVEC 会替代存储器地址 0x45 的内容
;  ---------------------------------------------------------------------------
;    USB Jump Table
;  ---------------------------------------------------------------------------
?PR?USB_JUMP_TABLE?USBJT
                RSEG        ?PR?USB_JUMP_TABLE?USBJT
                                    //以上两行是 C51 中断的定义格式
USB_Jump_Table :
        ljmp    ISR_Sudav
                                    //跳转到 SUDAV 中断，即收到设置数据的中断
        db      0
                                    //为了满足两个跳转指令之间 4 个字节的间距
        ljmp    ISR_Sof
                                    //跳转到 SOF 中断，即帧起始符中断
        db      0
        ljmp    ISR_Sutok
                                    //跳转到 SUTOK 中断，即收到设置标志包中断
        db      0
        ljmp    ISR_Susp
```

```
                                        //跳转到 SUSP 中断，即挂起中断
        db      0
        ljmp    ISR_Ures
                                        //跳转到 URES 中断，即 USB 复位中断
        db      0
        ljmp    ISR_IBN
                                        //跳转到 IBN 中断，即 IN 批量传输的 NAK 中断
        db      0
           ... ...
           ... ...
```

后面都是一些跳转语句，不再一一列出。这些中断服务例程都是由 C51 编写的，要使用自动向量机制，程序上有以下 4 点要求：

- 在存储器地址 0x43 插入一个跳转指令，这条跳转指令是跳转到一个跳转指令表的。这个跳转指令表中有一系列跳转到各种中断服务例程的跳转指令，确保跳转表起始自一个 0x100 字节的存储器页边界。
- 在跳转表中编制跳转到每个 USB 中断服务例程的跳转指令。这个跳转表有两个重要的要求，即必须起始自一个存储器页边界（地址 0xNN00）且每个跳转指令占 4 个字节。
- 中断服务例程可以放置在存储器中的任何地方。
- 要写初始化代码来使能 USB 中断和自动向量。

有了跳转表的帮助，用户可以将中断服务例程放在任何地方。用 C51 编写中断服务例程的格式前面已经介绍过了。不过这里有一点需要注意，放置中断服务程序的文件需要使用 C51 的预编译控制命令#pragma NOIV，即指示该文件编译时不产生中断向量。所以，关键字 interrupt 后可以是任何一个有效的中断号。

下面是一个收到 Setup 包的中断服务程序 SUDAV()。中断程序内只是简单地设置一个标志位，然后清除 USB 中断，最后清除 SUDAV 中断。这里有一点需要注意：在所有的中断服务例程中，清除中断时，要首先清除 USB 中断 INT2，然后才能清除特定的 USB 中断请求锁存器。这是因为 USB 中断一清除，任何等待的 USB 中断都会向 8051 的 INT2 中断发出脉冲。如果 INT2 中断请求锁存器没有被预先清除，那么等待的中断就会丢失。

```
void ISR_Sudav(void) interrupt 0
{
      GotSUD = TRUE ;           //设置标志位，表示收到设置数据包
      EXIF &=  ~0x10 ;          //清除 USB 中断 INT2
      USBIRQ = bmSUDAV ;        //清除 SUDAV 中断
}
void ISR_Susp(void) interrupt 0
{
 Sleep = TRUE ;                 //设置标志位，表示 USB 进入挂起状态
 EXIF &=  ~0x10 ;               //清除 USB 中断 INT2
 USBIRQ = bmSUSP ;              //清除 SUSP 中断
}
```

其他的中断服务例程也应用类似的处理方式，只是可能还有其他的工作需要完成。用户可以根据芯片资料和 USB 规范自行编写。

11.4.6　数据传输举例：批量 OUT 传输

USB 批量 OUT 数据从主机传向设备。主机通过给 EZ-USB 发送一个 OUT 标志包请求一个 OUT 传输，紧接着发送数据包。如果 USB 核正确地收到数据，它就会响应一个 ACK。如果端点的缓冲还没有准备好接收数据，USB 核就会抛弃主机的 OUT 数据，并且返回一个 NAK 标志包，以指示“没有准备好”。作为回应，主机持续地向端点发送 OUT 标志包和数据直到 USB 核响应 ACK 为止。

每一个 EZ-USB FX2 批量 OUT 端点都有一个字节计数寄存器 OUTnBC，该寄存器有两个功能：8051 读取字节计数寄存器来决定在刚刚结束的 OUT 传输中收到了多少字节的数据；8051 向该字节计数寄存器写入任何值来通知 USB 核数据已经从缓冲区中读取，可以接收下一个 OUT 传输了。

另外，每个批量 OUT 端点还有一个端点控制核状态寄存器 OUTnCS，该寄存器的 bit1 位 OUTnBSY 是标志端点当前是否忙的标志位。当主机数据在 OUT 端点的缓冲区中可以获得时，USB 核设置 BSY=0，8051 可以通过写入字节计数寄存器来设置 BSY=1。当 BSY=1 时，端点的 OUT 缓冲区正在准备接收从主机传送来的新数据，此时，8051 不能访问它。下面，就以端点 1OUT 为例，说明如何编写固件程序来进行批量 OUT 传输：

```
void ISR_Ep1out(void)   interrupt 0      //当端点正确收到 OUT 数据后，USB 核会
                                         //自动产生
  {                                      //中断，在中断表的引导下，转入此中断服务
                                         //程序
    int i = 0;                           //循环变量定义
    for( i=0 ; i<OUT1BC ; i++)           //判断数据是否读完
     {
       ReceiveData[ i ] = OUT1BUF[ i ];     //从端点的缓冲区读取数据
     }
    OUT1BC =0x00;                            //缓冲区中数据读取完，可以进行下一次
                                             //OUT 传输
   EXIF &=  ~ 0x10 ;                         //清除中断
   OUT07IRQ =bmEP1;
   }
```

11.4.7　数据传输举例：批量 IN 传输

USB 批量 IN 数据从设备传向主机。主机通过给 USB 核发送 IN 标志包来请求一个 IN 传输，当 USB 核准备好时，就向主机传送所请求的数据作为相应。8051 将要发送的数据字节数写入 IN 端点的字节计数寄存器，以指示准备好。如果 USB 核收到一个 IN 标志包，但是

该端点还没有准备好，它就会响应 NAK 的握手包。批量 IN 传输和批量 OUT 传输类似，只是要传送的数据大于端点缓冲区字节时会稍微复杂一些。下面，假设有 220 Byte 的数据要从 EP1IN 端点发送给主机，因为要传输的数据字节大于最大包尺寸，因此固件程序要将所有的 220 字节数据分为 4 个传输事务，各次传输的字节数据分别为 64、64、64 和 28 个字节。

```
void ISR_Ep1in(void)   interrupt 0        //在 USB 核收到主机发来的 ACK 响应后，就
                                          //产生中断
 {                                        //在中断向量表的引导下，进入中断服务程序
    int i=0;
    if (DataLength >= 64)                 //判断要发送的数据字节是否大于最大包大
                                          //小，Data-Length 是一个全局变量，表示剩
                                          //余的要发送的字节长度
  {
     for (i=0;  i<64;  i++)
      {
        IN1BUF[i] =SendData[156 - DataLength + i];
                                         //将待发送的数据写入端点缓冲区
      }
    DataLength = DataLength—64;                 //剩余数据的长度
IN1BC =0x40;                                    //本次发送的数据的长度
      }
else if (DataLength > 0)                        //要发送的数据长度小于最大包字节数，即表
                                          //示最后一次传输
   {
      for(i=0;  i< DataLength;  i++)
       {
          IN1BUF[i] =     SendData[156 - DataLength + i];
           }
       IN1BC =DataLength;
       ACQ_LENGTH=0;
  }
    else                                  //剩余数据长度为零，这种情况是所有要发送
                                          //的数据的字节正好是最大数据包的正倍数。
                                          //这时主机并不知道数据已经发送完毕
       {
       IN1BC=0x00;                        //发送的数据长度字节为 0，表示数据已经传
                                          //送完毕
      }
     EXIF &=  ~0x10 ;                     //清除中断
     IN07IRQ =bmEP1;
  }
```

11.5　本章小结

本章主要以 USB 控制芯片 CY7C68013 为例，讲述了如何进行 USB 设备的固件设计，其中包括相应设备请求、中断服务程序、批量 IN/OUT 传输以及描述符的定义。同时对所使用的开发语言 C51 进行了简单的介绍。本章所使用的范例代码都是从作者开发的实际项目中截取的，所以对读者开发自己的设备固件程序有一定的指导意义。

第 12 章　驱动程序设计

知识点：

- 设备驱动程序
- VxD、WDM、KMD
- 注册表
- INF 文件

本章导读：

从某种观点来看，Windows 2000 和 Windows 98 都是由一个操作系统核心和一系列驱动程序组成的，这些驱动程序提供了连接到计算机的硬件的软件接口。用户应用程序以一种规范的方式通过这些驱动程序组件访问硬件设备，而不需要知道硬件设备的具体细节，以及该如何控制硬件。USB 设备自然也不例外。本章将介绍设备驱动程序的基础，并结合具体实例简述 USB 驱动程序的开发流程。

12.1　设备驱动程序基础

设备驱动程序是一个软件组件，在装入后就成为操作系统内核的一部分。这些驱动程序介于硬件与用户应用软件之间，为它们之间的通信提供桥梁。这些硬件设备可以是键盘、鼠标、打印机、音频设备和视频捕获设备等，以及任何 CPU 可以访问的硬件设备。

应用程序可以不必知道它想要或者正在与之通信的硬件设备的属性，包括电气连接、物理地址、信号种类、通信协议的细节等，甚至应用程序可以不知道与之通信的是何种接口。这些工作全部是由各个层次的驱动来完成的，应用程序只需要知道设备的名称，或是设备的功能就可以了。

在 Windows 系统中，程序代码要么运行于用户模式（User Mode），要么运行于内核模式（Kernel Mode），不同的模式允许不同层次的内存存取和系统资源的分配。用户应用程序通常运行在用户模式，而操作系统代码和大部分驱动程序都是运行在内核模式下的。

应用程序可以说是作为操作系统服务器的一个客户在操作系统上运行的。在用户模式下，代码被限制于对系统无害的操作。例如，通过虚拟内存映射的方法，一个应用程序代码不能访问其他应用程序的内存空间（除非与另一个应用程序偕同工作）。因此，系统可以同时执行多个应用程序，而彼此不会互相干扰。应用程序也不能使用直接的硬件 I/O 指令来访问硬件。

内核模式也可以说是一种特权模式，内核模式的代码没有系统资源存取的限制，可以执行任何有效的 CPU 指令，包括 I/O 操作。在一个 Intel 平台上，使用指令集中的 Ring3 执行

用户模式，而使用 Ring0 来执行内核模式。正因为大部分驱动程序运行于内核模式，因此，一个微小的错误也会损坏操作系统的整体性能，甚至导致系统的崩溃。所以，编写驱动程序是一项谨慎小心的工作。

Windows 提供了标准的 API（Application Program Interface）函数，来启动应用程序与设备驱动程序之间的通信。当用户模式程序需要读取设备数据时，它就调用 Win32 API 函数，向设备驱动程序发出特定的 IRPs 请求。设备驱动程序得到请求后，向更底层的驱动程序传递 IRPs 请求，最后到达硬件设备，完成对硬件设备的控制与访问操作。对于应用程序编写者来说，使用这些 API 函数可以像访问一个文件那样访问设备。例如：Win32 API 函数 CreateFile 用于打开和创建到设备的连接，CloseHandle 用于在完成设备文件的使用时关闭文件句柄，ReadFile 和 WriteFile 系列函数用于向设备文件发出读或写请求。DeviceIoControl 用于发出特殊的请求，它可以从设备文件读取数据，也可以向设备文件发送数据。

Microsoft 公司的 Visual Studio MFC 也提供了很多控件，简化了用户应用程序的编写过程。用户可以不必调用各种 API 函数，直接使用控件就可以完成与设备的通信工作。例如，VC++提供的 MSComm 控件可以用来建立应用程序与 RS-232 串口的通信。该控件封装了 API 调用的细节，但是 Visual Studio 并没有提供 USB 通信的控件，因此，用户必须使用标准的 API 函数与设备进行通信。Windows 2000 操作系统中驱动程序和应用程序以及系统内核之间的关系如图 12-1 所示。

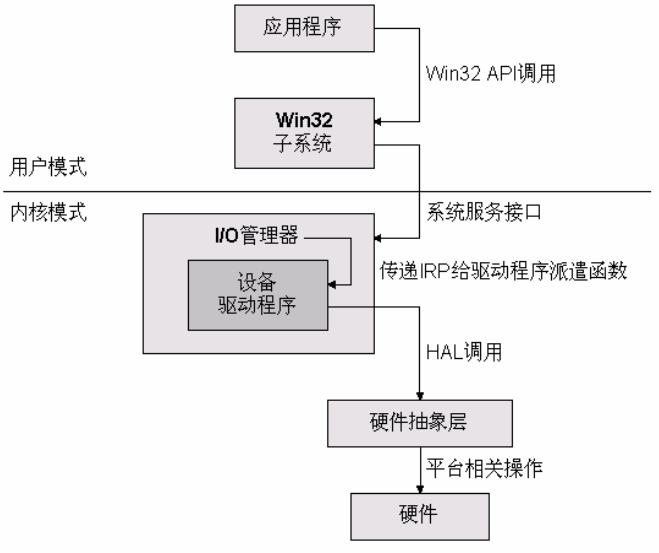


图 12-1　Windows 2000 的系统结构

有些硬件设备具有类似的属性，比如：共享一个总线或完成类似的任务，Microsoft 公司为它们提供了一系列的通用类驱动程序，执行这些相同或相似的任务，设备驱动程序可以使用这些标准类驱动程序以简化其程序的复杂程度。

另外，Microsoft 公司还提供了一系列的系统驱动程序，它们具有许多标准类型服务所需要的所有基本功能。第一种系统驱动程序支持不同类型的总线，如通用串行总线（USB）、IEEE1394（FireWire）和音频端口设备。这些系统驱动程序使一些设备驱动程序的设计变得容易得多，例如，USB 系统驱动程序处理 USB 总线上的所有底层通信。这样，其他驱动程序就可以使用一个定义好的接口与系统驱动程序通信，这使得向 USB 总线发出请求变得相当直接的。

12.2　驱动程序的分类

其实驱动程序早在最初的 DOS、Windows 3.x 系统时就已经存在了，一直到后来的 Windows 9x/NT/2000 等操作系统，驱动程序经历了几次大的演变过程。早期的 DOS 系统只是依靠基本输入输出系统（BIOS）的软件中断向应用程序提供硬件驱动的服务的，并通过 CONFIG.sys 中的 device=xxx 语句的方式进行扩展。到了 Windows 3.x 系统时，出现了标准模式的驱动程序，它们实际上是一些 16 位的动态链接库（DLLs），可以在保护模式中处理 I/O 中断，其中扩展名为.drv 的驱动程序至今还在沿用，例如：在 Windows 98 系统中，Win16 程序通过调用名为 COMM.DRV 的 16 位 DLL 间接地执行串行口 I/O，并且在 Windows 3.0 中引入了虚拟驱动程序 VxD（Virtual Driver）的概念，其中的 x 指的是各种不同的设备，如虚拟键盘驱动程序（vkd）、虚拟鼠标驱动程序（vmd）等。引入 VxD 的原始目的是为了虚化设备。

从 Windows 95 开始，微软就先后提供了 VxD 和 KMD（Kernel Mode Driver）两种不同的 Windows 驱动程序模型，它们分别对应着 16 位和 32 位混合的 Windows 9x 操作系统和纯 32 位的 Windows NT 操作系统。

12.2.1　VxD

虽然在 Windows 3.0 中就出现了 VxD（Virtual x Driver）的初步概念，但是直到 Windows 95 和 Windows 98 中才正式提出了 VxD 驱动程序的模型。在这两个操作系统中，系统的控制实权掌握在虚拟机管理器（VMM）和虚拟设备驱动程序（VxD）手中。VMM 的主要任务就是要在一个实际的硬件基础上创建并维护多个可以共享这个物理硬件的虚拟的计算机，而 VxD 就是帮助 VMM 实现每个虚拟计算机都拥有全部硬件的假象。一个虚拟机就是被软件创建的一个虚假对象。一个虚拟机和在它上面运行的程序进行交互，就像这个程序是在真正的计算机上运行一样。这样，一个程序不知道也不关心自己是否是在虚拟机上运行，只要虚拟机准确得像一个真的计算机一样响应程序，就可以把它当成一个真正的计算机。

VxD 运行在 Intel 系列 CPU 保护模式下的 Ring0 级，拥有对硬件的最高控制权。VxD 是 Windows 9x 特有的，它在 Windows NT/2000 系统下不能运行。Windows 98 的系统结构如图 12-2 所示，从中可以看出 VxD 所起的作用。

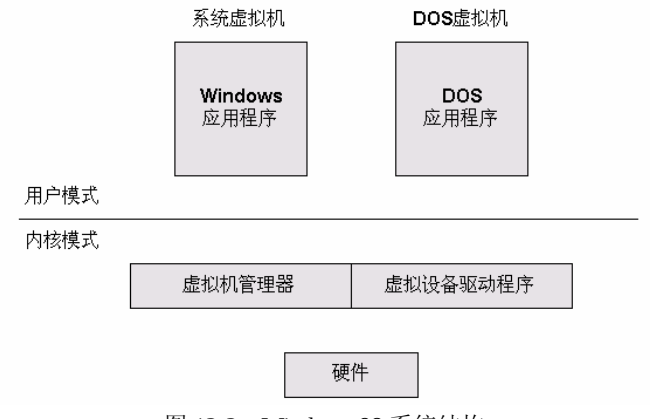


图 12-2　Windows 98 系统结构

12.2.2 KMD

KMD（Kernel Mode Driver）是微软为纯 32 位操作系统 Windows NT 设计的一种管理和维护硬件运作的驱动程序模型。与 Windows 9x 系统中驱动程序类型众多的情况不同，KMD 是 Windows NT 4.0 惟一支持的一种驱动程序类型。KMD 与 VxD 有很大的不同，它采用更灵活的方法，允许在应用程序和硬件之间存在几个驱动程序层次。这个分层允许 NT 更加广泛地定义驱动程序，包括文件系统、逻辑卷管理器和各种网络组件，以及各种物理设备驱动程序。KMD 驱动程序运行于 Windows NT 的内核模式下，类似于 Ring0。但是一个 KMD 的运作环境在不同的时候是根本不同的。驱动程序收到设备请求时的运作环境很可能和设备请求实际操作的运作环境根本不同。在 Windows NT 下，驱动程序的运行受到 Windows NT 很多的限制，一不小心，驱动程序就和 Windows NT 系统同归于尽了。

12.2.3 WDM

WDM（Win32 Driver Model）是在 Windows NT 4.0 驱动程序的基础上发展出来的，旨在改变 Windows 9x 系统多种驱动程序类型混杂的局面。它是 Microsoft 公司力推的驱动程序模式，相信会成为一种主流驱动模式。

早期的 Windows 3x，核心是 VMM，当时的 VMM 已经具备了基本操作系统核心的一些特征。但是 Windows 3x 的驱动程序模式混乱不堪：硬件由 Vxd 驱动，网络和文件系统由实模式驱动程序驱动，多媒体硬件和打印机由 Ring3 DLL 驱动，系统服务绝大部分被转到 V86 模式下由实模式的 DOS 完成。到了 Windows 95，很大一部分系统服务被转到了保护模式下。但是，混乱的驱动模式没有改变。Windows 98 是个大杂烩，包含了所有 Windows 95 的驱动程序模式，同时加上了 WDM。WDM 模式通过提供一种灵活的方式来简化驱动程序的开发，在实现对新硬件支持的基础上减少并降低所必须开发的驱动程序的数量和复杂性。除了通用的平台服务和扩展外，WDM 还实现了一个模块化的、分层次类型的微型驱动程序结构。类型驱动程序实现了支持通用总线、协议或设备类所需的功能性接口。WDM 和 NT 驱动程序有很多相似之处，编写 WDM 和 NT 的驱动程序基本上是相同的工作，驱动程序中的主要不同是如何创建设备。在 WDM 驱动程序中，PnP（即插即用）管理器会告知何时向系统添加了一个设备，或是从系统删除了一个设备。然而，NT 式驱动程序必须在初始化例程中编写代码，自己发现它的设备。

12.3 WDM 驱动程序基本结构

前面介绍了现有的几种驱动程序模型，而本章要讲述的 USB 驱动程序就属于 WDM 驱动程序的模式，所以，这一小节将简单介绍一下 WDM 驱动程序的基本结构。

12.3.1 WDM 驱动程序的层次结构

Windows 驱动程序模型（WDM）重新定义了驱动程序的分层，以适用于即插即用系统。各

个不同层次的驱动程序一起构成了一个设备驱动程序栈，如图 11-3 所示。在 WDM 驱动程序模型中，每个硬件设备至少有两个驱动程序。其中，一个驱动程序为功能（function）驱动程序，即通常硬件设备的驱动程序。它了解使硬件工作的所有细节，负责初始化 I/O 操作，有责任处理 I/O 操作完成时所带来的中断事件，有责任为用户提供一种设备适合的控制方式。另一个驱动程序为总线（bus）驱动程序。它负责管理硬件与计算机的连接。例如，PCI 总线驱动程序检测插入到 PCI 槽上的设备并确定设备的资源使用情况，它还能控制设备所在 PCI 槽的电流开关。

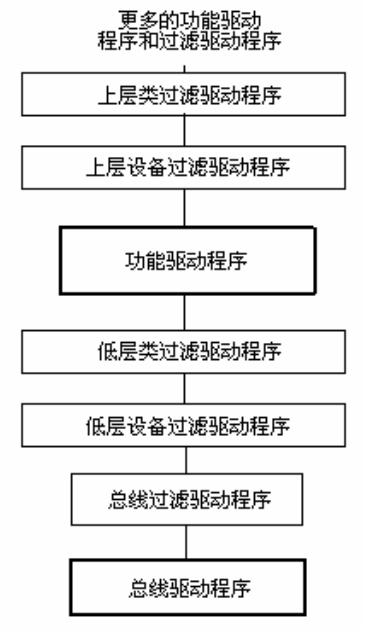


图 12-3　设备驱动程序栈

总线驱动程序要负责枚举总线，并为每个设备创建一个物理设备对象 PDO（Physical Device Object）。当总线驱动程序发现有设备接入或是拔出总线时，它要向上层驱动程序报告事件的发生。有些总线驱动程序只是简单地控制对总线的访问，而其他的总线驱动程序可以处理所有的总线事务。

功能驱动程序知道如何控制设备的主要功能，它位于总线驱动程序的上方。功能驱动程序可以为设备创建一个功能设备对象 FDO（Functional Device Object）。对 USB 设备来说，功能驱动程序必须使用 USB 类驱动程序访问 USB 设备。但是其他情况下，一旦总线驱动程序接管，功能驱动程序可以直接访问硬件。

WDM 功能驱动程序通常由两个分离的执行文件组成。一个文件是类驱动程序，它了解如何处理操作系统使用的 WDM 协议（有些协议相当复杂），以及如何管理整个设备类的基本特征。USB 照相机类驱动程序就是一个例子。另一个文件称为迷你驱动程序（Minidriver），它包含类驱动程序用于管理设备实例的厂商专有特征例程。类驱动程序和迷你驱动程序合在一起才成为一个完整的功能驱动程序。

在功能驱动程序之上以及功能驱动程序和总线驱动程序之间可能存在着各种类型的过滤驱动程序（Filter Driver）。过滤驱动程序也可以创建过滤设备对象 FiDO（Filter Device Object）。位于 FDO 上面的过滤器设备对象称为上层过滤器，位于 FDO 下面（但仍在 PDO 之上）的过

滤器设备对象称为下层过滤器。过滤驱动程序是一个中间层的驱动程序，它可以截获并处理经过它的 I/O 请求。过滤驱动程序用于变更标准设备驱动程序的行为，但它不是重写整个总线驱动程序或者类驱动程序，只是修改一些感兴趣的动作而已。图 12-3 中，有多种过滤驱动程序，过滤驱动程序总是相对于一个特定的驱动程序层而存在的，并帮助其完成一些辅助功能。总线过滤驱动程序作用于连接到特定总线驱动程序的所有设备，类过滤驱动程序在指定类的每个设备栈中安装，而设备过滤驱动程序针对一个特定的设备。低层的过滤驱动程序在功能驱动程序下，而上层的过滤驱动程序在设备栈中功能驱动程序之上。过滤驱动程序只是在设备第一次安装时装入，所以，在设备栈构造之后就不能插入过滤驱动程序了。在某些情况下，例如当 USB 设备插入 USB 总线时，低层过滤器驱动程序可以修改功能驱动程序要执行的总线操作流。

12.3.2 设备对象

除了驱动程序栈外，在前面还提到了 PDO、FDO 等，它们属于设备对象堆栈。设备对象是系统为帮助软件管理硬件而创建的数据结构，一个物理硬件可以有多个这样的数据结构。处于堆栈最底层的设备对象称为物理设备对象（Physical Device Object），或简称为 PDO。在设备对象堆栈的中间某处有一个对象称为功能设备对象（Functional Device Object），或简称 FDO。在 FDO 的上面和下面还会有一些过滤器设备对象（Filter Device Object）。WDM 中驱动程序和设备对象的层次结构和关系如图 12-4 所示。

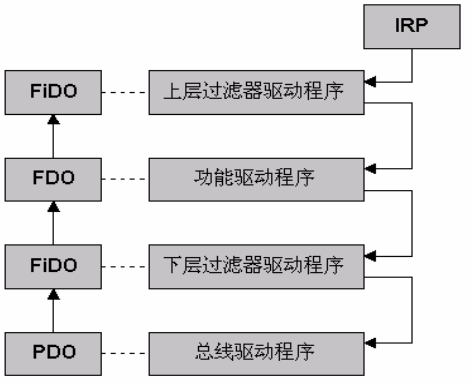


图 12-4　WDM 中设备对象和驱动程序的层次结构

一旦总线驱动程序检查到新硬件存在时，PnP 管理器就创建一个 PDO。创建完 PDO 后，PnP 管理器参照注册表中的信息查找与这个 PDO 相关的过滤器驱动程序和功能驱动程序，系统安装程序负责添加这些注册表项，而驱动程序包中控制硬件安装的 INF 文件负责添加其他表项。这些表项定义了过滤器驱动程序和功能驱动程序在堆栈中的次序。PnP 管理器先装入最底层的过滤器驱动程序并调用其 AddDevice 函数。该函数创建一个 FiDO，这样就在过滤器驱动程序和 FiDO 之间建立了水平连接。然后，AddDevice 把 PDO 连接到 FiDO 上，这就是设备对象之间连线的由来。PnP 管理器继续向上执行，装入并调用每个低层过滤器、功能驱动程序、每个高层过滤器，直到完成整个堆栈。

图 12-4 所示的这种层次结构的驱动程序使得 I/O 请求过程更加明了了，每个影响到设备的操作都使用 I/O 请求包。通常 IRP 先被送到设备堆栈的最上层驱动程序，然后逐渐过滤到

下面的驱动程序，每一层驱动程序都可以决定如何处理 IRP。有时，驱动程序不做任何事，仅仅是向下层传递该 IRP；或者驱动程序直接处理完该 IRP，不再向下传递；或者驱动程序既处理了 IRP，又把 IRP 传递下去。这取决于设备以及 IRP 所携带的内容。

这种分层模式的驱动程序有许多优点。使用分层允许将更高级的协议问题与特定的基础硬件管理分开。这就使得驱动程序可以支持更加广泛的硬件，而不必重新编写大量的代码。另外，如果几种不同种类的外围设备都能连接到相同的控制器上，则分层技术允许将外围设备的管理和控制器的管理分开。只需要为控制器编写一个设备驱动程序，同时为每种外设类型编写一个更高级的类驱动程序，就可以使类驱动程序变得更简单。USB 和 1394 总线的驱动程序正是这样做的。

但是这种分层体系也有不足之处，首先就是分层结构使得整个驱动程序变得很复杂。其次因为每个 IRP 请求每次从一个驱动程序传递到另一个驱动程序时，必须通过 I/O 管理程序，这样就增加了额外的系统开销。另外，安装分层的驱动程序也会比较复杂，因为每个分层都必须提供合适的安装步骤。并且，系统还要按照一定的顺序加载各层驱动程序。

12.3.3　USB 驱动程序结构

USB 驱动程序属于标准 WDM 驱动程序，但与传统 PC 总线（如 PCI 总线）设备的驱动程序相比，USB 设备驱动程序从不直接与硬件对话，总线枚举信息如何传输的细节也与 USB 设备驱动程序不直接相关。USB 设备驱动程序也叫客户驱动程序，它仅靠创建 URB（USB 请求块）并使用 USB 驱动程序接口（USBDI）将 URB 提交到总线驱动程序就可完成硬件操作。USBD.SYS 是接受 URB 的实体，向 USBD 的调用被转化为带有主功能代码为 IRP_MJ_INTERNAL_DEVICE_CONTROL 的 IRP。然后 USBD 再调度总线时间，发出 URB 中指定的操作。USB 驱动程序接口和 USB 驱动程序栈的结构如图 12-5 所示。

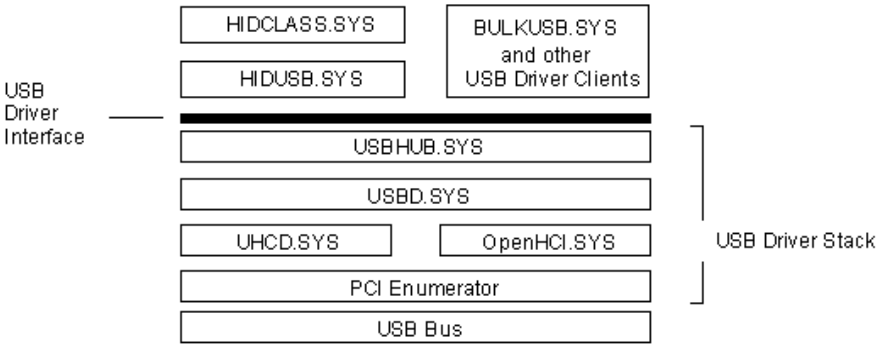


图 12-5　USB 驱动程序接口和 USB 驱动程序栈

- USBD.sys 是 USB 类驱动程序，它使用 UHCD.sys 访问通用主机控制器接口设备，或者使用 OpenHCI.sys 访问开放式主机控制器接口设备。
- HIDCLASS.sys 是人机接口设备（HID，Human Interface Device）的类驱动程序，这个类驱动程序向它的迷你驱动程序（Minidriver）放送 HID 报告，或者从中接收 HID 报告。
- HIDUSB.sys 是 HID 设备驱动程序，它从 USB 总线上发送或者接收 HID 报告。
- UHCD.sys 和 OpenHCI.sys 是 USB 主机控制器驱动程序。其中，OpenHCI.sys 是开

放主机控制器接口（Open Host Controller Interface）的驱动程序，而 UHCD.sys 是通用主机控制器驱动（Universal Host Controller Driver）的驱动程序。

- USBHUB.sys 是 USB 根集线器驱动程序。
- PCI Enumerator 是 PCI 枚举器，当检测到 USB 总线时，它负责装载 USB 堆栈驱动程序组件。
- BULKUSB.sys 是 Microsoft 2000 DDK 中提供的一个可以完成 USB 批量传输的驱动程序范例，这个才是用户真正需要编写的 USB 设备驱动程序或者客户驱动程序。当然，这里也可以是其他的设备驱动程序。USB 客户驱动程序是支持即插即用功能的标准 WDM 驱动程序，但是它绝对不会收到任何硬件资源(如端口或中断),因为 USB 类驱动程序处理所有的低层 I/O。

12.4 USB 设备驱动程序开发流程

前面几小节讲述了驱动程序的一些基础知识，包括驱动程序的作用、类型以及 WDM 驱动程序的分层结构。从本节开始，将介绍一个 USB 驱动程序开发的基本流程。USB 驱动程序属于标准的 WDM 驱动程序。要开发一个优秀的 USB 驱动程序相当复杂，需要具备很多相关的知识，包括操作系统内核的知识。此处介绍开发 WDM 驱动程序的一般流程，包括准备知识、一些必要例程，如何编译调试驱动程序等，从而给读者一个整体的概念，引导读者着手开发驱动程序。

12.4.1 准备工作（工具选择）

在 Windows 操作系统下，根据不同的情况以及用户对驱动程序功能的不同需求，可以选择不同的开发工具。一般来说，开发 WDM 驱动程序可以有两种不同的选择。

一种是 Microsoft 公司的 DDK（Device Developer Kit），包括 Windows 2000 DDK 和 Windows 98 DDK，其中，Windows 2000 DDK 只能用于开发基于 Windows 2000 的驱动程序，而 Windows 98 DDK 只能应用开发基于 Windows 98 的驱动程序。DDK 提供了创建 WDM 驱动程序的一个开发环境，DDK 包括特定驱动程序的头文件、库函数、源代码、各种工具和文档用于开发 Windows 2000 或 Windows 98 的驱动程序。DDK 中也包括了很多 Microsoft 公司提供的范例代码和函数说明，帮助用户开发设备驱动程序。例如：Windows 2000 DDK 中就包含了 USB 批量传输的驱动程序源代码，可以作为用户学习开发 USB 驱动程序很好的范例。DDK 中含有 Build.exe 实用程序，帮助用户编译驱动程序。选择 DDK 开发驱动程序属于从低层开发做起，用户必须自行编写所有的功能代码，对编写者的知识要求比较高。DDK 系列开发包属于免费软件,用户可以在 Microsoft 公司的网站 http://www.microsoft.com/ddk/免费下载。

另外一种就是选择一些驱动程序的辅助开发工具，例如：Driver Studio、Windriver、Vtoolsd 等，这些开发工具可以根据用户的需求，为用户自动生成一个量身定做的驱动程序。用户只需要在生成驱动程序之前，在向导框中按照提示填入自己的需求，并对生成的驱动程序作一些小的改动即可。有的工具生成的设备驱动程序甚至不用作任何的改动就可以使用了。开发者根本不需要了解驱动程序内部和操作系统内核的工作过程，这对于开发驱动程序的初学者

无疑是一种方便快捷的方法。

DriverStudio 是 CompuWare 公司发布的一套 Windows 下的全套驱动开发工具集。DriverStudio 能加速开发、调试、测试、调整和配置用户的 VXD、WDM 和 Windows NT 驱动程序。DriverStudio 把高质量的工具和现代的软件工程实践带到了以前被忽略的驱动开发领域。DriverStudio 包括以下几个主要的工具：

- DriverWorks 提供一个强大的面向对象的构架和简便的向导来开发基于 Windows NT/2000/98 的设备驱动程序。DriverWorks 的运行向导界面如图 12-6 所示。
- Vtoolsd 包括所有使用 C/C++开发面向 Windows 98/95 的 VxD 驱动程序所需要的文档、向导、库和例子代码。
- SoftICE 提供超越传统 Windows SDK/DDK 工具能力的强劲特性来减少调试时间。SoftICE 独特的宽范围系统察看和控制能力使它很容易定位和诊断很多种类的 Windows 软件问题。除此之外，还有 DriverWorkBenth、FieldAgent 等实用工具，功能很强大。SoftICE 调试界面如图 12-7 所示。

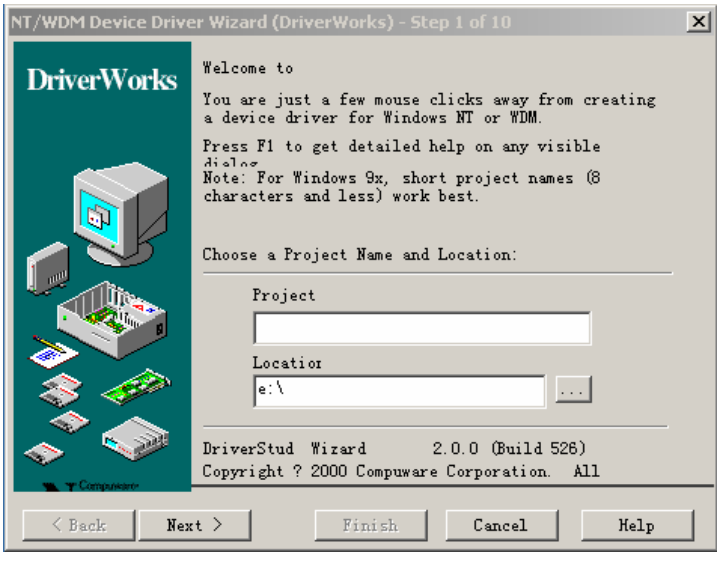


图 12-6　DriverWorks 向导界面

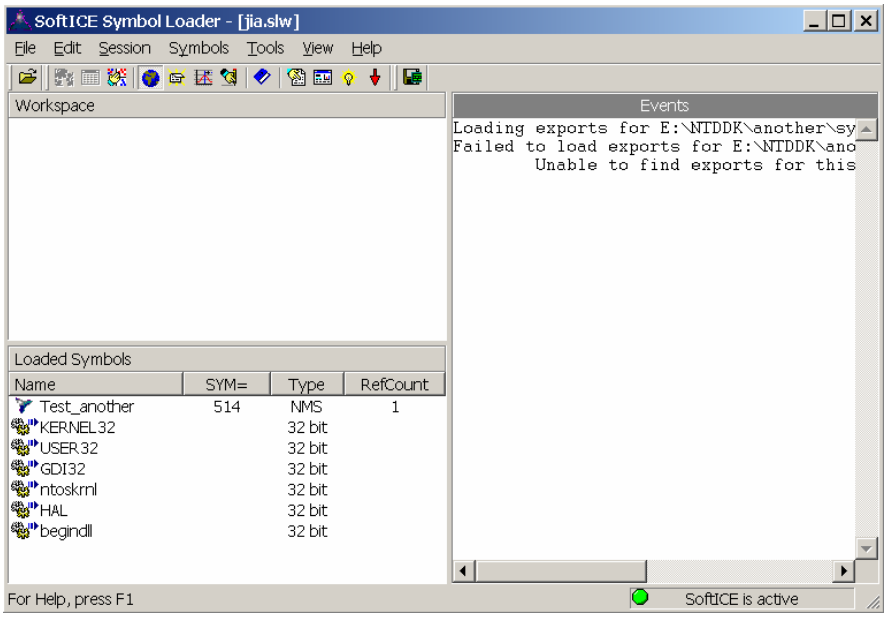


图 12-7　SoftICE 调试界面

尽管这些辅助开发工具使用起来非常方便，减轻了驱动程序开发的工作量，但是这种由辅助工具自动生成的驱动程序大多功能比较简单，所以，如果用户想要开发功能复杂的驱动程序或者想要了解驱动程序内部的工作过程，那么就只有选择 DDK，并从低层做起。而且用 DDK 开发驱动程序也是一种最正宗的方式。本章所介绍的也是用 Windows 2000 DDK 开发驱动程序的一般流程。

注意：尽管也可以使用 Windows 98 DDK 在 Windows 98 下开发 WDM 驱动程序，但是与开发 Windows 2000 WDM 驱动程序有很多的不同之处，例如，Windows 98 和 Windows 2000 的注册表就有所不同。

除了 DDK 以外，还必须有 Microsoft 公司的 VC++ 5.0/6.0 的专业版或者企业版以提供驱动程序的编译环境，VC++还提供了各种有用的工具，例如：rebase 和 guidgen。另外，MSDN（Microsoft Developer's Network）也是一个很好的帮助工具，它包括了大量的文档、范例程序以及开发工具。

12.4.2　两个重要的概念 IRP 和 URB

在开发 USB 设备驱动程序中，有两个重要的概念对于理解 USB 驱动程序非常重要：一个是 I/O 请求包（IRP），另一个就是 USB 请求块 URB。下面就分别对其进行介绍。

1. IRP

IRP 是驱动程序操作的中心，它是一个数据结构，带有一组对它进行操作的 I/O 管理器例程。I/O 管理器接收一个 I/O 请求，然后再把它传递到合适的驱动程序栈中的最高驱动程序之前，分配并初始化一个 IRP。一个 IRP 有一个固定的头部和一个或者多个 IRP 栈单元，每个 I/O 请求有一个主功能代码，并可能有次功能代码。例如：对设备的一个 Windows 32

CreateFile 调用最终转换为“创建”一个 IRP，它是一个主功能代码为 IRP_MJ_CREATE 的 I/O 请求包。全部 Win32 函数和它们对应的 IRP 如表 12-1 所示。

表 12-1　　常见的分发例程

Win32 函数	IRP 主功能代码	驱动程序例程的基名称
CreateFile	IRP_MJ_CREATE	Create
CloseHandle	IRP_MJ_CLOSE	Close
ReadFile	IRP_MJ_READ	Read
WriteFile	IRP_MJ_WRITE	Write
DeviceIoControl	IRP_MJ_DEVICE_CONTROL	DeviceControl

这些只是来自用户态 Win 32 函数的 I/O 请求所产生的 IRP，有几个其他的 IRP 只能由内核产生，响应与内核有关的事件。例如，系统决定关闭时，Power IRP 被发送到所有的驱动程序。一个常见的 IRP 不能从用户态代码产生，内部 IOCTL IRP 只能从内核产生，或者允许驱动程序显露一个不能从 Win 32 使用的接口，这些内部 IOCTL 通常由通用驱动程序使之可用。例如，USB 类驱动程序只接受内部 IOCTL 的命令，它们不支持普通的读和写。

“创建”、“读”、“写”、“关闭”、IOCTL 和内部 IOCTL IRP 的处理程序通常称为分发例程，因为它们常常仅执行 IRP 的一些初始工作。如检查所有的参数是否合法，然后把 IRP 分发到驱动程序中的其他地方处理。IRP 通常需要串行处理，使得驱动程序以一种安全的方式与硬件打交道。

任何内核模式程序在创建一个 IRP 时，同时还创建了一个与之关联的 IO_STACK_LOCATION 结构数组：数组中的每个堆栈单元都对应一个将处理该 IRP 的驱动程序。另外，还有一个堆栈单元供 IRP 的创建者使用，如图 12-8 所示。堆栈单元中包含该 IRP 的类型代码和参数信息以及完成函数的地址。当一个 IRP 由多个驱动程序处理时，使用多个 IRP 栈单元。每个驱动程序从当前 IRP 栈单元得到它的 IRP 参数。如果把一个 IRP 沿驱动程序栈向下传递，必须使用正确的参数设置下一个栈单元。

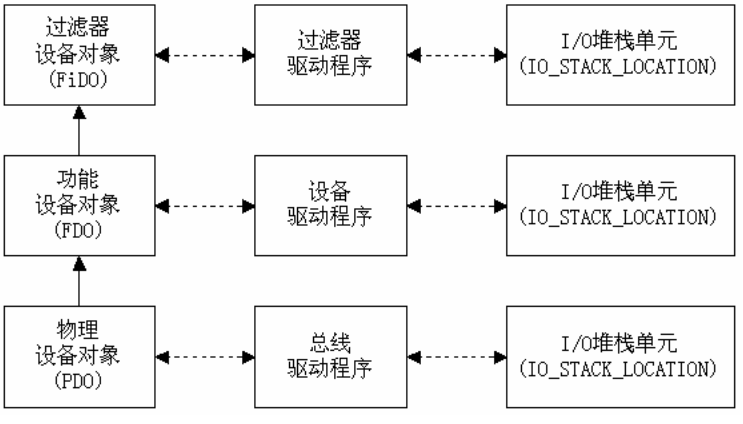


图 12-8　驱动程序和 I/O 堆栈之间的平行关系

前面讲过 IRP 是一个数据结构，下面就是 Windows 2000 DDK 所定义的 IRP 结构声明：

```
typedef struct _IRP {
```

```
.
.
PMDL MdlAddress;
ULONG Flags;
union {
   struct _IRP *MasterIrp;
   .
   .
   PVOID SystemBuffer;
} AssociatedIrp;
.
.
IO_STATUS_BLOCK IoStatus;
KPROCESSOR_MODE RequestorMode;
BOOLEAN PendingReturned;
.
.
BOOLEAN Cancel;
KIRQL CancelIrql;
.
.
PDRIVER_CANCEL CancelRoutine;
PVOID UserBuffer;
union {
   struct {
   .
   .
   union {
      KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
      struct {
         PVOID DriverContext[4];
      };
   };
   .
   .
   PETHREAD Thread;
   .
   .
   LIST_ENTRY ListEntry;
   .
```

```
        .
        } Overlay;
      .
      .
      } Tail;
} IRP, *PIRP;
```

2. URB

URB 是在 USB 设备驱动程序中所用的 USB 请求块。USB 驱动程序高度依赖其总线驱动程序（USBD.SYS），而不直接使用 HAL 函数与硬件通信。USB 驱动程序为了向其硬件设备发送一个请求，首先创建一个 USB 请求块（URB），然后把 URB 提交到总线驱动程序。例如，为了配置一个 USB 设备，驱动程序需要提交几个 URB 来读取各种描述符或发送命令，最后由 USBD.SYS 把请求送到总线上。USB 类驱动程序 USBD.SYS 就是接受 URB 的实体，设备驱动程序把 URB 包装成一个带有 IRP_MJ_INTERNAL_ DEVICE_CONTROL 主功能码的普通 IRP，然后 USBD 再调度总线时间，发出 URB 中指定的操作。

URB 结构是一个联合体，含有 16 个不同的_URB_*结构，每一个功能代码使用其中的一个 URB 结构详细说明它的输入或输出参数。例如，

功能代码：SUB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE

构造例程：UsbBuildGetDescriptorRequest

URB 结构：_URB_CONTROL_DESCRIPTOR_REQUEST

使用这个函数可从设备读取任何的描述符。下面是 Windows 2000 DDK USBDI.h 文件中对 URB 联合的声明：

```
typedef struct _URB {
union {
        struct _URB_HEADER                              UrbHeader;
        struct _URB_SELECT_INTERFACE                    UrbSelectInterface;
        struct_URB_SELECT_CONFIGURATION                 UrbSelectConfiguration;
        struct _URB_PIPE_REQUEST                        UrbPipeRequest;
        struct _URB_FRAME_LENGTH_CONTROL                UrbFrameLengthControl;
        struct_URB_GET_FRAME_LENGTH                     UrbGetFrameLength;
        struct_URB_SET_FRAME_LENGTH                     UrbSetFrameLength;
        struct_URB_GET_CURRENT_FRAME_NUMBER UrbGetCurrentFrame-Number;
        struct_URB_CONTROL_TRANSFER                     UrbControlTransfer;
        struct_URB_BULK_OR_INTERRUPT_TRANSFER UrbBulkOrInterrupt-Transfer;
        struct_URB_ISOCH_TRANSFER                UrbIsochronousTransfer;

                // for standard control transfers on the default pipe
        struct_URB_CONTROL_DESCRIPTOR_REQUEST UrbControlDescriptor-Request;
        struct_URB_CONTROL_GET_STATUS_REQUEST UrbControlGetStatus-Request;
        struct_URB_CONTROL_FEATURE_REQUEST    UrbControlFeature-Request;
```

```
            struct_URB_CONTROL_VENDOR_OR_CLASS_REQUEST UrbControl-VendorClassRequest;
            struct_URB_CONTROL_GET_INTERFACE_REQUEST UrbControlGet-InterfaceRequest;
            struct_URB_CONTROL_GET_CONFIGURATION_REQUEST
UrbCon-trolGetConfigurationRequest;
        };
    } URB, *PURB;
```

12.4.3 编写驱动程序

编写驱动程序与编写应用程序有很大的不同，因为驱动程序是系统受信任的组件。用户必须仔细地编写代码，并遵循一定的规则。在编写前作详细的准备策划工作是必不可少的。编写时用户可以使用由粗到细的方法：首先编写一个不实现任何功能的驱动程序框架，拥有几个必要的分发例程。然后，可以使用 Win 32 程序调用 CreateFile 和 CloseHandle 函数进行简单的测试。接着一步步地添加其他的功能代码和处理其他 IRP 的分发例程。这样，既可以缩短开发周期，又可以确保驱动程序的安全性。

另外，Windows 2000 DDK 中包含有大量的范例代码，采用多种方法使用此代码，可以使驱动程序的开发更加容易。例如，DDK 中提供了 USB 批量传输的驱动程序源代码，这是个学习编写驱动程序的很好的范例，很多驱动程序的初学者都是从这里开始的。

驱动程序使用 C 或者 C++语言进行编写，应该尽量避免使用汇编语言，因为它使得驱动代码难以阅读，而且移植性很差。

可以把一个完整的驱动程序看作是一个容器，它包含许多例程，当操作系统遇到一个 IRP 时，它就调用这个容器中的例程来执行该 IRP 的各种操作。图 12-9 表现了这个概念。有些例程，例如 DriverEntry 和 AddDevice，还有与几种 IRP 对应的派遣函数将出现在每一个这样的容器中，需要对 IRP 排队的驱动程序一般都有一个 StartIo 例程。执行 DMA 传输的驱动程序应有一个 AdapterControl 例程。大部分能生成硬件中断的设备的驱动程序都有一个中断服务例程（ISR）和一个推迟过程调用（DPC）例程。驱动程序一般都有几个支持不同类型 IRP 的派遣函数，其中 3 个派遣函数是必须的。所以，WDM 驱动程序开发者的一个任务就是为这个容器选择所需要的例程。

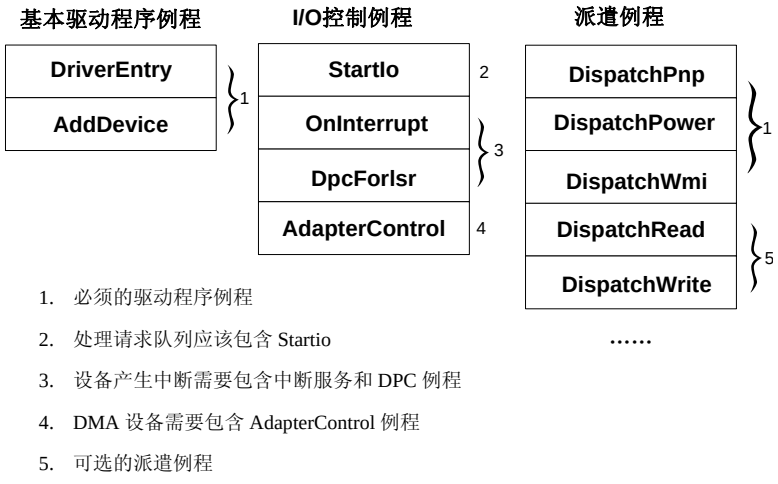


图 12-9　WDM 驱动程序“容器”中的内容

要编写一个完整的驱动程序相当复杂，此处由于篇幅的限制，这里只讲解一些重要的例程，使读者有一个基本的概念，要想彻底地了解驱动程序的编写，还需要阅读相关的专著。

1. DriverEntry()例程

大家都知道所有 C 或者 C++开发的应用程序都是从 main()或者 Winmain()函数开始执行的。同样，每个 Windows 2000 内核模式或者 WDM 驱动程序，不管它的用途是什么，也都需要一个入口点，那就是 DriverEntry()例程。I/O 管理器装入驱动程序时就调用 DriverEntry()，从而开始驱动程序代码的执行。因为 DriverEntry()有一个众所周知的名字，而其他驱动程序没有固定的名字，因此，I/O 管理器需要用其他的方法来定位它们。Windows 2000 WDM 的 DriverEntry() 例程可以完成一些驱动程序的全局初始化工作，它通过声明另一个驱动程序入口点来对其进行定位，初始化驱动程序对象通过把函数指针直接保存到驱动程序对象中完成说明工作。这些函数指针分为两类：

- 在驱动程序中有显式的位置和名字的函数。
- 在驱动程序对象的 MajorFunction 数组中所列出的 IRP 分发例程。

如果初始化成功，DriverEntry 应该返回 STATUS_SUCCESS 状态给 I/O 管理程序；如果由于某种原因 DriverEntry 在初始化过程中失败了，它应该释放为它分配的任何系统资源，并应该返回一个在 NTSTATUS.H 头文件中定义的错误代码，或者返回用户自定义的错误代码，STATUS_SUCCESS 的值为 0。下面就是一个完整的 DriverEntry 例程。

```
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
    NTSTATUS ntStatus = STATUS_SUCCESS;
    PDEVICE_OBJECT deviceObject = NULL;
```

```
    BOOLEAN fRes;
                        //变量定义
    DriverObject->DriverUnload = DriverUnload;
        DriverObject->DriverExtension->AddDevice = AddDevice;
        DriverObject->DriverStartIo = StartIo;
                        //为驱动程序的其他入口点设置函数指针
    DriverObject->MajorFunction[IRP_MJ_CREATE] = BulkUsb_Create;
                        //设置处理用户态 CreateFile()调用 I/O 请求的函数指针
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = BulkUsb_Close;
                        //设置处理用户态 CloseHandle()调用 I/O 请求的函数指针
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]=BulkUsb_Process-IOCTL;
                        //设置处理用户态 DeviceIoControl()调用 I/O 请求的函数指针
        DriverObject->MajorFunction[IRP_MJ_WRITE] = BulkUsb_Write;
        DriverObject->MajorFunction[IRP_MJ_READ] = BulkUsb_Read;
                        //设置处理用户态 WriteFile()和 ReadFile()调用 I/O 请求的函数指针
    DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL]=BulkUsb_Process-SysControlIrp;
        DriverObject->MajorFunction[IRP_MJ_PNP] = BulkUsb_ProcessPnPIrp;
        DriverObject->MajorFunction[IRP_MJ_POWER] = BulkUsb_ProcessPowerIrp;
                        //设置处理系统 PnP 和电源管理 I/O 请求的函数指针


        return ntStatus;
    }
```

DriverEntry 的第一个参数是一个指针，指向一个刚被初始化的驱动程序对象，该对象就代表用户的驱动程序。参数前面的关键字“IN”表明该参数是用作输入的。DDK 头文件并不真正使用这些关键字，WDM 驱动程序的 DriverEntry 例程应完成对这个对象的初始化并返回。非 WDM 驱动程序需要做大量额外的工作，它们必须探测自己的硬件，为硬件创建设备对象（用于代表硬件），配置并初始化硬件使其正常工作。在 WDM 驱动程序中比较复杂的硬件探测和配置工作由 PnP 管理器自动完成。

DriverEntry 的第二个参数是一个 UNICODE_STRING，它包含注册表中设备驱动程序服务键的路径。这个串不是长期存在的（函数返回后可能消失），如果以后想使用该串就必须先把它复制到安全的地方。WDM 驱动程序几乎不要改注册表键名。

WDM 驱动程序的 DriverEntry 例程的主要工作是把各种函数指针输入驱动程序对象。这些指针为操作系统指明了驱动程序容器中各种子例程的位置。上面这个 DriverEntry 例程的前三条语句就为驱动程序的其他入口设置了函数指针。这几个函数是属于上面所讲的第一类函数的。等式后面的函数名可以由用户自行定义，这里使用了见文识意的名字。

下面分别讲述这几个函数指针。

- DriverUnload：指向驱动程序的清除例程。I/O 管理器会在卸载驱动程序前调用该例程。通常，WDM 驱动程序的 DriverEntry 例程一般不分配任何资源，所以 DriverUnload 例程也没有什么清除工作要做。
- DriverExtension->AddDevice：指向驱动程序的 AddDevice 函数。PnP 管理器将为每

个硬件实例调用一次 AddDevice 例程，AddDevice 函数的基本职责是创建一个设备对象并把它连接到以 PDO（物理设备对象）为底的设备堆栈中。

❑ DriverStartIo：如果驱动程序使用标准的 IRP 排队方式，应该设置该成员，使其指向驱动程序的 StartIo 例程。

另外，在 DriverEntry 例程中还有一个 MajorFunction 指针数组，数组中每个指针都指向一个 IRP 派遣函数，用来处理一个特定的 IRP。每个 WDM 驱动程序必须能处理主功能代码为 IRP_MJ_PNP、IRP_MJ_POWER、IRP_MJ_SYSTEM_CONTROL 的 3 种 I/O 请求，应该在这里为这些请求指定派遣函数。除了这 3 个分发例程外，用户还可以添加处理其他 IRP 的分发例程，并设置 MajorFunction 指针数组中对应元素的指针指向这些分发例程。例如，BulkUsb_Write 例程用于处理 Win 32 函数 WriteFile 调用的 I/O 请求，其主功能代码为 IRP_MJ_WRITE。

DriverEntry 例程最后返回状态 STATUS_SUCCESS。

2. AddDevice 例程

另一个重要的例程是 AddDevice 例程。这个函数被调用来创建和初始化功能设备对象（FDO）。对于单一的驱动程序来讲，这个工作可以在 DeviceEntry 例程中完成，但是对于一个即插即用的设备，要等第一个 PnP 事件，才调用 AddDevice 例程完成设备的创建以及其他相关工作。通常，一个驱动程序可以被多个设备利用，PnP 管理器为每个设备实例调用 AddDevice 函数。该函数的原型如下：

```
NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo)
{
}
```

DriverObject 参数指向一个驱动程序对象，即在 DriverEntry 例程中初始化的驱动程序对象。pdo 参数指向设备堆栈底部的物理设备对象。

功能驱动程序的 AddDevice 函数的基本职责是创建一个设备对象并把它连接到以 pdo 为底的设备堆栈中。它采用如下的步骤完成这些工作：

（1）调用 IoCreateDevice 创建设备对象，并建立一个私有的设备扩展对象。

（2）寄存一个或多个设备接口，以便应用程序能知道设备的存在。另外，还可以给出设备名并创建符号连接。

（3）初始化设备扩展和设备对象的 Flag 成员。

（4）调用 IoAttachDeviceToDeviceStack 函数把新设备对象放到堆栈上。

IoCreateDevice 例程用于创建设备对象，其函数原型如下：

```
PDEVICE_OBJECT    fdo;
NTSTATUS status = IoCreateDevice(DriverObject,
                        sizeof (DEVICE_EXTENSION),
                        NULL,
                        FILE_DEVICE_UNKNOWN,
                        FILE_DEVICE_SECURE_OPEN,
                        FALSE,
                        &fdo);
```

第一个参数（DriverObject）就是 AddDevice 的第一个参数。该参数用于在驱动程序和新设备对象之间建立连接，这样，I/O 管理器就可以向设备发送指定的 IRP。

第二个参数是设备扩展结构的大小。正如在本章前面讲到的，I/O 管理器自动分配这个内存，并把设备对象中的 DeviceExtension 指针指向这块内存。

第六个参数（FALSE）指出设备是否是排斥的。通常，对于排斥设备，I/O 管理器仅允许打开该设备的一个句柄。

第七个参数（&fdo）是存放设备对象指针的地址，IoCreateDevice 函数使用该变量保存刚创建设备对象的地址。

如果 IoCreateDevice 由于某种原因失败，则它返回一个错误代码，不改变 fdo 中的值。如果 IoCreateDevice 函数返回成功代码，那么它同时也设置了 fdo 指针。下面是 AddDevice 例程中部分代码：

```
NTSTATUS
Ezusb_PnPAddDevice(
      IN PDRIVER_OBJECT DriverObject,
      IN PDEVICE_OBJECT PhysicalDeviceObject
      )
{
    NTSTATUS                    ntStatus = STATUS_SUCCESS;
    PDEVICE_OBJECT              fdo = NULL;
    PDEVICE_EXTENSION           pdx;
ntStatus = IoCreateDevice (DriverObject,
                                sizeof (DEVICE_EXTENSION),
                                &deviceNameUnicodeString,
                                FILE_DEVICE_UNKNOWN,
                                0,
                                FALSE,
                                DeviceObject);

    if (NT_SUCCESS(ntStatus))
    {

        // Initialize our device extension
        pdx = (PDEVICE_EXTENSION) ((*DeviceObject)->DeviceExtension);

        RtlCopyMemory(pdx->DeviceLinkNameBuffer,
                      deviceLinkBuffer,
                      sizeof(deviceLinkBuffer));
      ... ... ... ...
```

3. 创建并发送一个 URB

上面两个例程是所有 WDM 设备驱动程序都需要的例程。接下来，将介绍在 USB 设备驱动程序中如何创建并提交一个 URB。为了创建一个 URB，首先应该为 URB 分配内存，然后调用初始化例程，把 URB 结构中的各个域填入请求要求的内容。例如，当用户为响应 IRP_START_DEVICE 请求而配置设备时，首要的任务就是读取该设备的设备描述符，UsbBuildGetDescriptorRequest 例程就用于读取 USB 设备描述符：

```
USB_DEVICE_DESCRIPTOR dd;
URB urb;
UsbBuildGetDescriptorRequest(&urb,
                    sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
                    USB_DEVICE_DESCRIPTOR_TYPE,
                    0,
                    0,
                    &dd,
                    NULL,
                    sizeof(dd),
                    NULL);
```

这里，首先声明一个局部变量 urb 来保存 URB 数据结构。URB 在 USBDI.H 中声明，它是一个多子结构的联合。这里，将使用 UrbControlDescriptorRequest 子结构，其类型是 _URB_CONTROL_DESCRIPTOR_REQUEST。UsbBuildGetDescriptorRequest 看上去像一个正常的服务例程，实际上它是一个宏（在 USBDLIB.H 中声明），用于生成“读描述符请求子结构”各个域的初始化语句。在 DDK 头文件中定义了这些创建各种 URB 的宏。

创建完 URB 后，需要创建并发送一个内部 I/O 控制（IOCTL）请求到 USBD 驱动程序，USBD 驱动程序位于驱动程序层次结构的低端。创建内部 IOCTL 请求最简单的方法是调用 IoBuildDeviceIoControlRequest 函数，第一个参数（IOCTL_INTERNAL_USB _SUBMIT_URB）指出 I/O 控制代码，说明这个 IRP 是向 USB 类驱动程序发送 USB 请求块的。第二个参数（pdx->LowerDeviceObject）指定接收请求的设备对象，IoBuildDevic-eIoControlRequest 使用这个指针来决定需要接收多少个堆栈单元。接下来的 4 个参数描述输入输出缓冲区，提交这种 URB 并不需要这些信息，所以，例子中将它们置成 NULL 或 0 值。第七个参数为 TRUE，它指出用户创建的是 IRP_MJ_INTERNAL_DEVICE_CONTROL 请求而不是 IRP_MJ_DEVICE_CONTROL 请求。最后两个参数指出等待 URB 完成的事件和一个接收该操作最终状态的结构 IO_STATUS_BLOCK，然后调用 IoCallDriver 例程把请求发送到下一层驱动程序。USBD 将处理并完成该 URB 请求，然后，I/O 管理器将包含该 URB 请求 IRP 删除并置事件信号。

4. 批量传输

主机应用程序调用 API 函数 DeviceIOControl()通过批量管道进行批量数据传输，应用程序的 IRP 请求被发送到设备驱动程序，设备驱动程序将会调用 UsbBuildInterruptOrB-ulkTransferRequest()函数格式化一个 URB，用来在一个指定的批量管道

上发送和接收数据。然后，程序将 URB 封装成一个 IRP 传向下一层驱动程序 USBD，下面就是这个例程的具体范例：

```
UsbBuildInterruptOrBulkTransferRequest(urb,                    //指向 URB 的指针
                                       (USHORT) urbSize,       //URB 大小
                                        pipeHandle,            //管道句柄
                                        NULL,                  //传输缓冲区
                                        Irp->MdlAddress,
                                        bufferLength,          //传输缓冲区长度
                                       transferFlags,          //传输缓冲区标志
                                       NULL);                  //连接

 ntStatus = Ezusb_CallUSBD(fdo, urb);                          //调用 USBD
NTSTATUS DispatchRead(PDEVICE_OBJECT fdo, PIRP Irp)
{
   return ReadWrite(fdo, Irp, TRUE);
}

NTSTATUS DispatchWrite(PDEVICE_OBJECT fdo, PIRP Irp)
{
   return ReadWrite(fdo, Irp, FALSE);
}

NTSTATUS ReadWrite(PDEVICE_OBJECT fdo, PIRP Irp, BOOLEAN read)
{
   PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
   NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
   if (!NT_SUCCESS(status))
      return CompleteRequest(Irp, status, 0);
   ...
   IoMarkIrpPending(Irp);
   IoSetCompletionRoutine(Irp,(PIO_COMPLETION_ROUTINE) OnReadWriteComplete, ...);
   IoCallDriver(...);
   return STATUS_PENDING;
}
```

当然，要真正实现批量传输，还需要很多其他的辅助例程，在此不再赘述。

5. 结论

编写驱动程序需要涉及大量系统内核的知识，相对难度比较大，这一小节只是介绍了几个重要的例程，并给出了一些范例，要完成 USB 设备驱动程序的编写还需要更多的其他的工作，由于篇幅的限制，不可能都详细地解释，有兴趣的读者可以参看一些 WDM 驱动程序的

专著。

12.4.4　编译驱动程序

设计编写好驱动程序源代码之后，就可以进行编译了。Windows 2000 DDK 提供的 Build 实用程序及相关组件连同 Microsoft Visual C++一起被用来构造所有的驱动程序。Build 实用程序帮助用户维护相关的目标信息，例如源文件名、头文件和输出目录的路径。这个实用程序可自动设置编译器和连接器开关，并调用 nmake 实用程序来构造驱动程序。Build 只是 NMAKE 外面的一个外包装程序，Build 本身其实相当简单，编译的大部分工作实际上由 Build 传递给 NMAKE 来进行。

要构造驱动程序除了有驱动程序的源文件以外，还需要一些必要的辅助文件：一个 SOURCES 文件、一个标准的 makefile 文件、目录结构和可选的 makefile.inc 和 dir 文件。

1. 编译环境

编译环境可以分为自由构造（Free Build 或 Retail Build）和检查构造（Checked Build）。

自由构造（Free Build）是用于构造最终用户版本的环境，系统和驱动程序都是以最优化的方式进行构造的，调试的声明被禁止，调试信息也被从二进制代码中剔除。自由构造环境下构造的驱动程序小巧而且运行速度快，占用的内存少。

检查构造（Checked Build）是用于辅助检测和调试驱动程序的构造环境。检查构造包括额外的错误检测、参数校验和调试信息。一个检查的系统和驱动程序可以隔离和追踪一些导致程序不确定行为、内存泄漏和不正确的设备配置的驱动程序问题。尽管检查构造可以提供额外的保护，但是它比自由构造消耗的内存和磁盘空间更多，系统和驱动程序的性能也会因此而下降。

因此，在开发驱动程序的初期阶段有必要使用检查构造来调试驱动程序。而最后的发行版本应该使用自由构造以提高系统和驱动程序的性能。

2. sources 文件

每一个包含驱动程序源文件的子目录必须包括一个名为 sources 的文件，这个文件使用了一系列 build 实用程序可以识别的宏定义，并列出需要编译和链接的文件。这些宏定义的格式是：宏名=宏值。宏值是一个文本字符串，build 使用程序在内部脚本中用它来替换宏名。

注意：宏和它的等号之间不能有空格。

Sources 文件中可用的宏有很多种，这里列举一些常用的宏。

- TARGETNAME（必须）：用于指定要编译的库的名称，不包括扩展名。
- TARGETPATH（必须）：指定存放构造结果文件（可以是.exe、.dll、.lib 等文件类型）的目录名称。Build 实用程序在此目录下创建平台相关的子目录。Build 实用程序总是在包含 sources 文件的目录下创建一个\obj 的子目录。
- TARGETTYP（必须）：指定被编译的产品的类型，通常是 LIBRARY 或者 DYNLINK（DLLs），但也可以是 PROGRAM（用户模式程序）、HAL（硬件抽象层）、GDI_DRIVER（内

核模式的图形驱动）等。

- TARGETLIBS（必须）：指定用户的产品必须与之链接的导入库组。
- SOURCES（必须）：包含一系列带扩展名的源文件名称。这些源文件构成将要创建的库或者 DLLs。这些文件必须存在于包含 sources 文件的目录中。

下面就是一个实际的 sources 文件的范例：

```
TARGETNAME=BulkUsb
TARGETTYPE=DRIVER
DRIVERTYPE=WDM
TARGETPATH=obj
TARGETLIBS=$(DDK_LIB_PATH)\usbd.lib
USE_MAPSYM=1

SOURCES=                \
        BulkUsb.rc \
        BusbDbg.c \
        BulkUsb.c \
        BulkPnP.c \
        BulkPwr.c \
         IoctlBlk.c \
       OcrwBlk.c
```

3. makefile 文件

每一个包含驱动程序源文件的目录中还必须包括一个名为 makefile 的文件。Build 实用程序为每个在 sources 文件中列出的源文件激活 NMAKE 实用程序，而 NMAKE 实用程序又调用 makefile 来产生命令列表。每一个在驱动程序源代码目录中的 makefile 文件将 NMAKE 文件引导到主宏定义文件 makefile.def，这个文件是由 DDK 提供的。它指定了要传递给编译器和链接器的标志，使用这个文件可以简化创建与平台无关的驱动程序工程的过程。这个文件类似于平台 SDK 提供的 ntwin32.mak 文件，它用来构造 win 32 应用程序。

正如标准 makefile 文件中的注释所述，不要试图编辑这个文件。下面就是一个标准的 makefile 文件：

```
#
# DO NOT EDIT THIS FILE!!!    Edit .\sources. if you want to add a new source
# file to this component.    This file merely indirects to the real make file
# that is shared by all the driver components of the Windows NT DDK
#


!INCLUDE $(NTMAKEENV)\makefile.def
```

如果驱动程序包括不止一个的构造产品，或者如果用户的源文件保存在几个子目录中，那么，用户就需要在每个包含子目录的目录结点中包括一个名为 dirs 的文件。如果只有一个 dirs 文件，build 实用程序就查看其中的 DIRs 伪指令，查找要构造的目录列表。

4. 编译驱动程序步骤

讲述完这些辅助文件的内容后，就可以开始构造驱动程序了。要运行 build 实用程序，可以单击 2000 DDK 程序组中的“Free Build Environment”或“Checked Build Environment”图标（此处假定采用检查构造环境），这样就启动了一个命令提示符窗口，如图 12-10 所示。并且自动调用 DDK 提供的 setenv.bat 批处理文件来设置大多数的环境变量。还有一些环境变量，用户可以在自由构造环境或者检查构造环境的命令窗口中用“set”来设置。

在构建驱动程序之前，必须在驱动程序源文件所在目录下手动创建\lib\i386\checked 或\lib\i386\free 两个目录之一。创建目录后，启动 Windows DDK 进入命令提示符窗口后，使用 CD 命令进入包含驱动程序源代码的目录中，然后键入 build–c 就可以编译链接驱动程序了。在 Windows 2000 操作系统中，编译完后将会在\lib\i386\checked 目录下产生输出文件。如果选择的是自由构造，那么输出的驱动程序文件将会保存在\lib\i386\free 目录下。

注意：由于 build 实用程序的错误，所以必须在驱动程序的源文件所在目录下手动创建\lib\i386\checked 或\lib\i386\free 两个目录之一。

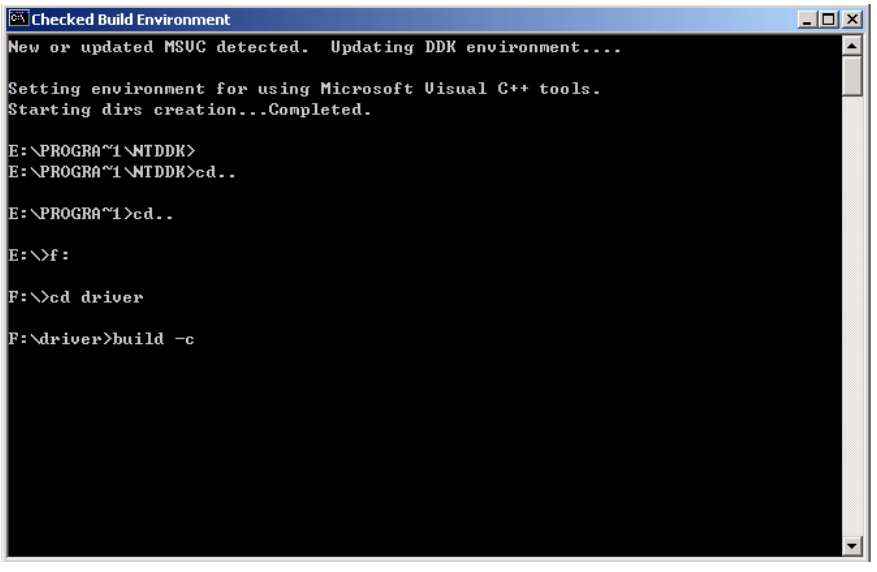
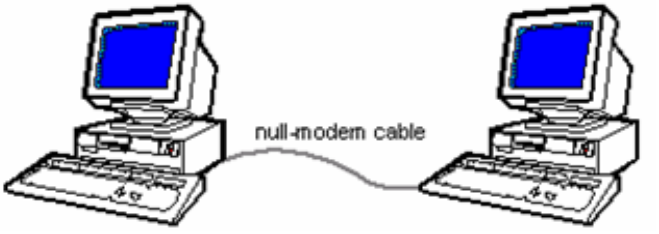


图 12-10　build 实用程序界面

5. 调试驱动

尽管驱动程序已经生成了，但是否可以安全地使用，并能达到预期的功能需求，还需要对其进行测试和调试，有很多工具可以帮助用户测试和调试驱动程序。不幸的是，Visual Studio 调试程序不能用于内核态代码，而 Microsoft WinDbg 内核调试程序必须在两台装有 Windows 2000 系统的计算机之间使用，一台是目标机，一台是主机。目标机用来运行驱动程序或者一个内核模式的应用程序，而主机则用来运行调试器，如图 12-11 所示。



主机（HOST）　　目标机（TARGET）

图 12-11　典型的 Windows 2000 驱动程序调试设置

NuMege Compuware 公司推出的驱动程序开发包 DriverStudio 中包含一个功能强大的调试工具 SoftICE。SoftICE 以其强大的功能和超越传统的 Windows SDK/DDK 工具的特性缩减用户的调试时间，它独有的系统范围的观察和控制能力使得理解和诊断最大限度的 Windows 软件问题变得很容易。SoftICE 具有以下的一些特点：

- 可以通过 Internet 进行远程调试。
- 提供单机、源码级的调试能力。
- 支持 Windows NT/9x/2000，为公司在任何 Windows 平台构造设备驱动程序和系统组件提供强大的、可信赖的调试。
- SMP 支持：SoftICE 现在支持在 Pentium、Pentium Pro 和 PentiumⅢ多处理器系统上进行调试，最多可以支持多达 8 个使用标准 Intel 多处理器方案的 CPUs。
- 支持 Micrososft 的内核调试扩展。

由于篇幅原因，这里不能详细地介绍 SoftICE 的使用方法，SoftICE 有很多的控制命令，有兴趣的读者可以参看一些相关的文档。

12.4.5　安装驱动程序

编写好驱动程序的源代码，并且进行了编译链接之后，就可以安装驱动程序了。下面，就讲述一下如何使用安装 INF 文件来安装 WDM 驱动程序，以及系统如何查找驱动程序。

1. 检测新硬件

当系统发现新的设备时，就调用 Windows 的“添加新设备向导”，如图 12-12 所示。这个向导会扫描所有可用的 INF 文件，试图找到合适的驱动程序。如果是首次安装的设备，系统可能不会找相应的 INF 文件，也就不可能找到合适的驱动程序（也有可能会找到可用的，但不是最合适的驱动程序）。这时，系统会弹出“添加新设备向导”，用户就需要手动为系统指定 INF 文件的路径，安装管理器根据 INF 文件中的表项来查找合适的驱动程序，并进行安装。

图 12-12　添加新硬件向导

如果总线驱动程序检测到新的硬件，或者用户使用控制面板的“添加新硬件”向导安装一个设备，则安装一个 WDM 驱动程序。在这两种情况下，即插即用管理（PnP）程序为该设备和它的驱动程序在注册表的配置表中添加一些条目。系统根据 INF 文件中的指令安装驱动程序。驱动程序可执行文件被复制到正确的位置，通常是在 Windows System32\Driver 目录下创建各种注册表项。

由上面的安装过程可以看出：其中的 INF 文件和注册表在安装过程中起着重要的作用。INF 文件含有安装一个 WDM 设备驱动程序需要的所有必要的信息，包括要复制的文件列表、要创建的注册表项等，Windows 由此来选择安装哪一个驱动程序。INF 文件必须由用户自行编写，并在安装过程中提供给系统。但是，INF 文件的内容都有一定的格式，而且不同设备的 INF 文件内容变化不是很大，因此，用户可以参照其他设备的 INF 文件进行编写。在随后的 12.5 小节中会详细地介绍 INF 文件。

2. 添加注册表项

系统的注册表（Registry）保存着所有已经安装的设备信息，在检测到一个新设备后，设备管理器会将设备的信息复制到注册表相应的键下。注册表以树型结构来管理它的内容。有 3 种注册表键负责设备的安装与配置，它们是硬件（hardware）键、类（class）键和服务（Service）键。设备的硬件键出现在注册表 local machine 分支的 \System\CurrentC-ontrolSet\Enum 子键上。\Enum\USB 子键中就包含了所有以前用过和现在存在的 USB 设备的描述，如图 12-13 所示。用户可以使用 Windows 自带的注册表编辑器 regedit.exe 来观察和修改注册表的内容。但注册表是系统重要的组件，建议不要随便修改。

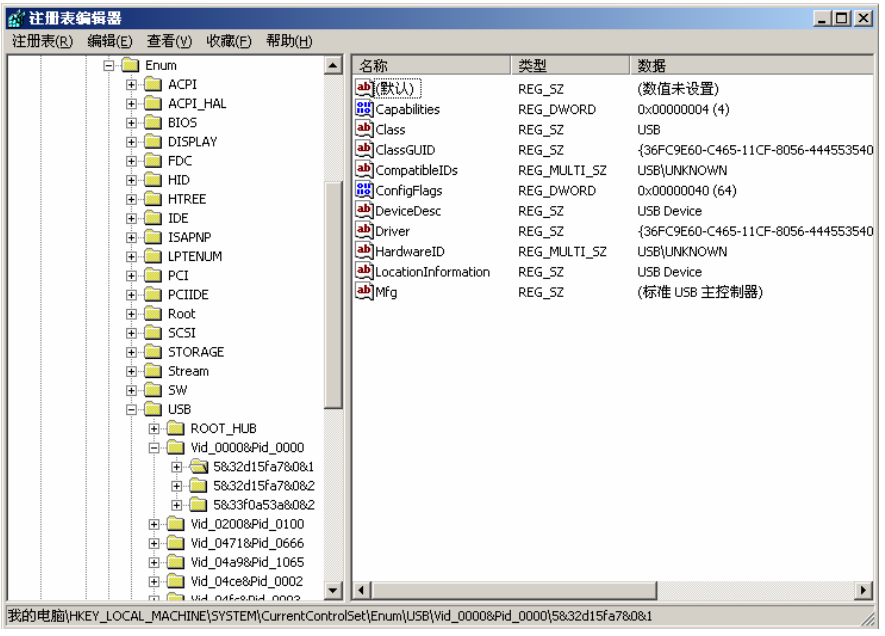


图 12-13　注册表中 Enum\USB 子键的信息

12.5　INF 文件

由上述安装过程可以得知 INF 文件的重要性，因此，这里将会对 INF 文件进行详细的讲解。

12.5.1　INF 文件格式要求

一个 INF 文件是以段组织的简单的文本文件。一些段有系统定义（System-Defined）的名称，而另一些段由 INF 文件的编写者命名。每个段包含特定的条目和命令，这些命令用于引用 INF 文件其他地方定义的附加段。下面，将简单讨论一下 INF 文件的语法规则。

❑　要求的内容：在特定的 INF 文件中所要求的必选段和可选段、条目及命令依赖于所要安装的设备或组件。端点顺序可以是任意的，大多数的 INF 文件安装惯用的次序来安排各个段。

❑　段名：INF 文件的每个段从一个括在方括号[ ]中的段名开始。段名可以由系统定义或 INF 编写者定义。

在 Windows 2000 中，段名的最大长度为 255 个字符。而在 Windows 98 中，段名不应该超过 28 个字符。如果 INF 设计要在两个平台上运行，必须遵守最小的限制。段名、条目和命令不分大小写。在一个 INF 文件中如果有两个以上的段有相同的名字，系统将把其条目和命令合并成一个段。每个段以另一个新段的开始或文件的结束为结束。

❑　使用串标记：在 INF 文件中的许多值，包括 INF 编写者定义的段名都可以表示成

%strkey%形式的标记。每个这样的 strkey 必须在 INF 文件的 Strings 段中定义为一系列显式可见字符组成的值。

❑　行格式、续行及注释：段中的每个条目或命令以回车或换行符结束。在条目或命令中，“\”可以被用作一个显式的续行符；分号（;）表示后面的内容是注释；可以用逗号（,）分隔条目和命令中提供的多个值。

把一个大的 INF 文件看成是一个树结构的线形描述，这样可以更容易理解 INF 文件。一个段就是树上的一个节点，而每个指令就是指向另一个段的指针，如图 12-14 所示。

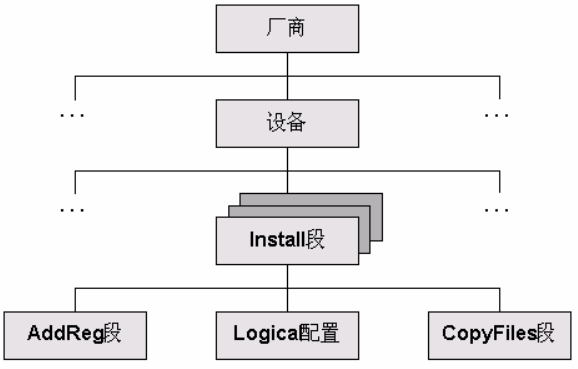


图 12-14　INF 文件的树型结构

12.5.2　INF 文件举例

下面就是一个完整的.inf 文件，它是 Windows 2000 DDK 提供的 USB 批量传输驱动程序范例中所附的.inf 文件：

```
;    Installation inf for the Intel 82930 USB Bulk IO Test Board

;    (c) Copyright 1999 Microsoft
;
[Version]
Signature="$CHICAGO$"
Class=USB
ClassGUID={36FC9E60-C465-11CF-8056-444553540000}
provider=%MSFT%
DriverVer=08/05/1999

[SourceDisksNames]
1="BulkUsb Installation Disk",,,

[SourceDisksFiles]
BULKUSB.sys = 1
```

```
BULKUSB.inf = 1

[Manufacturer]
%MfgName%=Microsoft

[Microsoft]
%USB\VID_045E&PID_930A.DeviceDesc%=BULKUSB.Dev, USB\VID_045E&PID_930A

;  [PreCopySection]
;  HKR,,NoSetupUI,,1

[DestinationDirs]
BULKUSB.Files.Ext = 10,System32\Drivers
BULKUSB.Files.Inf = 10,INF

[BULKUSB.Dev]
CopyFiles=BULKUSB.Files.Ext, BULKUSB.Files.Inf
AddReg=BULKUSB.AddReg

[BULKUSB.Dev.NT]
CopyFiles=BULKUSB.Files.Ext, BULKUSB.Files.Inf
AddReg=BULKUSB.AddReg

[BULKUSB.Dev.NT.Services]
Addservice = BULKUSB, 0x00000002, BULKUSB.AddService

[BULKUSB.AddService]
DisplayName      = %BULKUSB.SvcDesc%
ServiceType      = 1                  ; SERVICE_KERNEL_DRIVER
StartType        = 3                  ; SERVICE_DEMAND_START
ErrorControl     = 1                  ; SERVICE_ERROR_NORMAL
ServiceBinary    = %10%\System32\Drivers\BULKUSB.sys
LoadOrderGroup = Base

[BULKUSB.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,BULKUSB.sys
HKLM,"System\Currentcontrolset\Services\BulkUsb\Parameters",
"MaximumTransferSize",0x10001,4096
HKLM,"System\Currentcontrolset\Services\BulkUsb\Parameters","DebugLevel",
```

```
0x10001,2

[BULKUSB.Files.Ext]
BULKUSB.sys

[BULKUSB.Files.Inf]
BulkUsb.Inf
;  ------------------------------------------------------------;
[Strings]
MSFT="Microsoft"
MfgName="Intel"
USB\VID_045E&PID_930A.DeviceDesc="BulkUsb.Sys Intel 82930 USB Bulk IO Test Board"
BULKUSB.SvcDesc="BulkUsb.Sys i82930 Bulk IO test driver"
```

12.5.3　INF 文件详解

从上面这个完整的例子来看，可以对 INF 文件有一个总体的印象，包括 INF 文件包括的段，以及各段的书写格式。下面将详细地介绍构成 INF 文件的各个段。

1. [Version]段

习惯上，每个 INF 文件都开始于一个 Version 段，该段确定文件中描述的设备类型，上述范例中的 Version 段有如下几条语句：

```
[Version]
Signature="$CHICAGO$"
Class=USB
ClassGUID={36FC9E60-C465-11CF-8056-444553540000}
provider=%MSFT%
DriverVer=08/05/1999
```

Signature 指定使用此 INF 文件的操作系统，可以是$Chicago$、$Windows NT$（含有一个空格）或$Windows 95$（也含有一个空格）之一，定界符$必不可少，且这些串是不分大小写的。如果 Signature 的值不是这些有效的串值之一，该 INF 文件就被认为无效。如果一个 INF 文件用来向 Windows 98 和 Windows 2000 两个平台上安装设备驱动程序，它必须通过 DDInstall 段来增加系统定义的扩展指定任意操作系统特有的安装信息，而不管 Signature 是何值。

Class 指定设备的类名，此范例中指定的是 USB 类。ClassGUID 指定设备注册表的 GUID，GUID 是一个 128 位的标志符，DDK 头文件 DEVGUID.H 定义了标准设备类的 GUID。

Provide 标志该 INF 文件的提供者。%MSFT%的具体内容将在 Strings 段中定义，范例中的定义是 MSFT="Microsoft"，表明该 INF 的提供者是 Microsoft。

DriverVer 条目提供整个 INF 文件的版本信息，在每个 Install 段中加上 DriverVer 条目，为驱动程序提供版本信息。Install 段的驱动程序版本条目更具有专用性，并且比 Version 段的

全局 DriverVer 条目日期具有更高的优先级。当操作系统搜索驱动程序时，它会选择一个具有更近的 DriverVer 日期的驱动程序代替一个较早的驱动程序。如果一个 INF 没有 DriverVer 条目，操作系统将会使用缺省的日期 00/00/0000。

2. [SourceDisksNames]段

该段指定并且命名一个或多个包含源文件的磁盘，这些源文件用于文件拷贝或者重命名操作。该段可以有任意条目，每个条目对应一个源盘。条目的通式如下：

diskid=%strkey% |"disk-description",[tagfile],[unused],[path]。

diskid 是标志一个源盘的非负整数。这个值可以以十进制或者十六进制的形式表示，但它不能占用多于 4 个字节的存储单元。等式右边规定一个%strkey%标记或者一个引号引起来的串，描述由 diskid 所标识盘符的内容或目的。在安装过程中安装程序可以给终端用户显示这个串值。Tagfile 是一个可选的值，规定一个所带磁盘上提供的特征文件名，或者在磁盘根目录中，或者在给定的子目录中，这个值应该只规定文件名，不规定任何目录和子目录。安装程序使用特征文件核对用户插入正确的安装盘。特征文件只能用于可移动的介质。

Unused 值不用在 Windows 2000 中，只用在 Windows 9x 中。Path 也是个可选项，用于标识磁盘上包含源文件的目录路径。范例中 SourceDisksNames 段的内容如下：

```
[SourceDisksNames]
1= “BulkUsb Installation Disk” ,,,
```

范例规定了源盘为磁盘 1，在安装期间，安装程序可以给终端用户显示字符串“BulkUsb Installation Disk”。

3. [SourceDisksFiles]段

该段命名安装过程中所用的源文件，标志包含这些源文件的磁盘（或者 CD-ROM），并提供在所带磁盘上包含的每个文件的目录路径。一个 SourceDisksFiles 段可以有任意多条目，磁盘上每一个文件都有一个条目。它所包含的条目的通式如下：

filename=diskid[,[subdir][,size]]。

filename 规定磁盘上源文件的名称；Diskid 规定一个整数来标志包含源文件的磁盘，即在 SourceDisksNames 段中规定的 Diskid；Subdir 是个可选值，它规定了原磁盘上的文件所在的子目录，如果该条目省略，指定的源文件或者在给定磁盘的根目录中，或者在由 SourceDisksNames 段的 path 条目所指定的目录；Size 也是个可选值，规定了给定文件的非压缩长度，其以字节表示。范例中 SourceDisksFiles 段的内容如下：

```
[SourceDisksFiles]
BULKUSB.sys = 1
BULKUSB.inf = 1
```

范例中为两个文件建立了条目，这两个文件都在磁盘 1 中，并且在根目录下。

4. [Manufacturer]段

该段标志一个或者多个用 INF 文件安装的设备的制造商，它也为制造商的设备及驱动程序的安装定义 Models 段名。每一个 INF 文件都必须有 Manufacturer 段。

上述范例的 Manufacturer 段有一个条目：

```
[Manufacturer]
%MfgName%=Microsoft
```

在 Strings 段中可以找到%MfgName%定义的字符串，本范例中定义的是 MfgName =

"Intel"，表明设备制造商是 Intel。而等式右边 Microsoft 也是制造商的 Models 段的段名。在 INF 文件中，为每个制造商的 Models 段规定了一个 INF 编写者定义的名字，这个名称要在 Manufacturer 段中加以应用。范例中的制造商 Models 段如下：

```
[Microsoft]
%USB\VID_045E&PID_930A.DeviceDesc%=BULKUSB.Dev,USB\VID_045E&PID_930A
```

该段属于制造商的 Models 段，段名是由 INF 编写者自行定义的，不属于系统段名。每个制造商的 Models 段至少标识一个设备，并规定设备的厂商 ID（VID）和产品 ID（PID）同时引用这个设备 INF 文件的 Install 段。该段也可以规定一个或者多个附加设备 ID，因为有多个与初始硬件 ID 所识别的设备兼容，同时由相同的驱动程序驱动。当设备管理器发现从检测设备所得到的 ID，符合此段定义的 ID 时，设备管理器就知道找到了正确的 INF 文件。

范例中只规定了一个设备，设备的 VID 是 0x045E，而 PID 是 0x930A。VID 是由 USB 管理委员会给每个 USB 芯片厂商统一分配的，例如 Philips 公司的 VID 是 0x0471，Cypress 公司的 VID 是 0x0547 等，范例中的 VID 是 Intel 公司的。产品 ID（PID）是由各个厂商自己定义的，这里的 VID 0x930A 是 Intel 的一个 USB 批量传输的试验板。

范例中该条目等式右边的 BULKUSB.Dev 给设备标识了一个 INF 文件编写者定义的 Install 段。

5. [DestinationDirs]段

该段为所有的硬件拷贝、删除和改名操作规定目标目录。范例中的 DestinationDirs 段如下：

```
[DestinationDirs]
BULKUSB.Files.Ext = 10,System32\Drivers
BULKUSB.Files.Inf = 10,INF
```

DestinationDirs 段中条目等式的左边规定 INF 文件编写者定义的段名，这些段中的文件将会被存入等式右边指定的目录中，并且这些文件可以被 INF 文件中其他地方的 CopyFiles、RenFiles 和 DelFiles 命令引用。例如，范例中这两个段的定义如下：

```
[BULKUSB.Files.Ext]
BULKUSB.sys
[BULKUSB.Files.Inf]
BulkUsb.Inf
```

DestinationDirs 段中条目等号右边的 10 是一个逻辑磁盘标识符（Logical Disk Identifier，LDID），它规定了对文件操作的目标目录的标志符，后面跟随的是子目录，作为文件操作的目标地址。范例中第一个文件 BULKUSB.sys 的目标地址是 Winnt\Syste-m32\Drivers；第二个文件 BulkUsb.Inf 的目标地址是 Winnt\INF。

DestinationDirs 段也可以包含一个缺省目标目录 DefaultDestDird 的条目，为所有对文件的拷贝、删除和更名操作规定缺省的目标路径，这些文件没有明确地列在其他条目所引用的文件列表中。Windows DDK 的 Device Information File Reference 文件有其他的 LDID 数值的定义，如表 12-2 所示。

表 12-2　　常用 LDID 定义

LDID 数值	目标目录
00	NULL LDID，此 LDID 可以用来创建新的 LDID
01	源磁盘驱动器:\路径
02	临时安装文件夹，只在安装期间有效
03	解除安装目录
10	Windows 目录
11	系统目录
12	IO 子系统目录
13	Command 目录
14	控制面板目录
15	打印机目录
16	工作类别目录
17	INF 目录
18	Help 目录
19	登记
20	字体
21	Viewers
22	VMM32
23	Color 目录
24	应用程序目录
25	共享目录
26	Winboot
28	主机 Winboot
30	引导驱动器的根目录
31	虚拟引导驱动器的主机驱动器的根目录

6. Install 段

Install 段包含指导安装程序安装所需软件的实际指示。该段必须在一个设备或者型号（Model）的条目中引用。每个 Models 的每个 Install 段包括一个可选的 DriverVer 条目和一个或多个引用 INF 中附加命令段的命令，这些命令所引用的段包含安装驱动程序文件及向注册表写入设备专用的和驱动程序专用信息命令。这里，首先列出这个段的常用通式：

```
[install-section-name]
```

```
[install-section-name.nt]
[install-section-name.ntx86]

[DriverVer=mm/dd/yyyy]
[CopyFiles=@filename | file-list-section]
Addreg=add-registry-section
[include=filename.inf]
……
……
```

首先，看到有 3 个可选的段名，在 Windows 9x 平台中，提供了一个无修饰的 Install 段名[install-section-name]，其中规定设备安装条目。在 Windows 2000 平台中，提供相应的[install-section-name.nt]段名，如果还要限定在以 x86 为基础的 Windows 2000 平台上，则提供[install-section-name.ntx86]段名。设备安装程序将搜寻最合适后缀的 install 段，例如，假设有 3 个 install 段，第一个无后缀，后两个分别带有.NT 后缀和.NTx86 后缀。如果安装到 x86 平台的 Windows 2000 系统中，安装程序会使用.NTx86 段；如果安装到非 Intel 平台上运行的 Windows 2000 中，安装程序将使用.NT 段；如果安装到 Windows 98 中，安装程序使用无后缀段。DriverVer 条目在前面的版本段中已经讲述过了，此处就不再重述。

- [CopyFiles=@filename | file-list-section]：或者指定一个要从源媒体拷贝到目标设备的文件名，或者引用一个或多个 INF 编写者定义的段，其中列出了源媒体上要拷贝到目标设备的相关文件。这个命令是可选的，但是大多数的段都包括这个命令。
- Addreg=add-registry-section：这个命令引用一个或多个 INF 编写者定义的段，其中规定了要写入注册表的新的子键，这个新关键字可能有初始条目，在这些段中也可以修改已有关键字的值。范例中所引用的段如下：

```
[BULKUSB.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,BULKUSB.sys
HKLM,"System\Currentcontrolset\Services\BulkUsb\Parameters","MaximumTransferSize",0x10001,4096
HKLM,"System\Currentcontrolset\Services\BulkUsb\Parameters","DebugLevel",0x10001,2
```

HKLM 是注册表的根键，它是 HKEY_LOCAL_MACHINE 的缩写。而 HKR 与注册表关键字相关，最适合用于 AddReg 命令出现的地方。比如，注册表中每个设备的“硬件”子键：..\Enum\枚举器 ID\设备 ID，与此相对的注册表中每个驱动程序指定的“软件”子键：..CLASS\类 GUID\设备 ID 等。除此之外还有 HKCR、HKCU、HKLM 和 HKU。

前面所举范例的 Install 段如下：

```
[BULKUSB.Dev]
CopyFiles=BULKUSB.Files.Ext, BULKUSB.Files.Inf
AddReg=BULKUSB.AddReg
```

所要拷贝的两个文件在 INF 编写者定义的 BULKUSB.Files.Ext 和 BULKUSB.Files.Inf 段中，而要添加到注册表的项目则在 BULKUSB.AddReg 段中。

7. Install.Service 段

该段包含一个或者多个 AddService 命令，用以控制一个特定驱动程序的服务装载的时间和方式，控制本服务对其他服务或下一级驱动程序的依赖等，其范例的定义如下：

```
[BULKUSB.Dev.NT.Services]
Addservice = BULKUSB, 0x00000002, BULKUSB.AddService
```

其中，BULKUSB 是指定的所要安装服务的名称。对于一个设备，这个值通常是该设备驱动程序的属名。0x00000002 是一个系统指定的标记（Flag），不同的标记有不同的服务功能。最后一项 BULKUSB.AddService 是 INF 文件编写者定义的特定段，其中包括了所要添加的服务项目，其范例的定义如下：

```
[BULKUSB.AddService]
DisplayName     = %BULKUSB.SvcDesc%
ServiceType     = 1                     ; SERVICE_KERNEL_DRIVER
StartType       = 3                     ; SERVICE_DEMAND_START
ErrorControl    = 1                     ; SERVICE_ERROR_NORMAL
ServiceBinary   = %10%\System32\Drivers\BULKUSB.sys
LoadOrderGroup = Base
```

8. [Strings]段

[Strings]段就是字符串段。该段定义了其他段内所指定的字符串，段内的每一个项目都符合一个在其他段内使用百分比符号（%）包括起来的字符串。范例中的定义如下：

```
[Strings]
MSFT="Microsoft"
MfgName="Intel"
USB\VID_045E&PID_930A.DeviceDesc="BulkUsb.Sys Intel 82930 USB Bulk IO Test Board"
BULKUSB.SvcDesc="BulkUsb.Sys i82930 Bulk IO test driver"
```

12.5.4　用 geninf 实用程序生成一个 INF 文件

Windows 2000 DDK 提供了一个实用程序 geninf，如图 12-15 所示。它是一个 INF 文件生成向导工具，可以帮助用户产生一个自定义的 Windows 2000 驱动程序的 INF 文件。用户只需要按照向导的提示填写必要的信息，该实用程序就可以自动生成一个 INF 文件。但是，INF 只能生成单结构的 INF 文件，而并不支持多结构的 INF 文件。另外，geninf 程序并不一定总是能够生成一个完整的或者功能完备的 INF 文件。因此，用户还必须作一些必要的修改。尽管如此，也要比从头开始写方便得多。

用 geninf 生成 INF 文件之后，还可以使用 Windows 2000 DDK 提供的 ChkINF 实用程序来检查所生成的 INF 文件的结构和语法是否正确。ChkINF 程序是一个 Perl script，因此，需要有 Perl 解释器才能运行它。检测的结果以 HTML 的格式保存，包括所有错误和警告的列表。

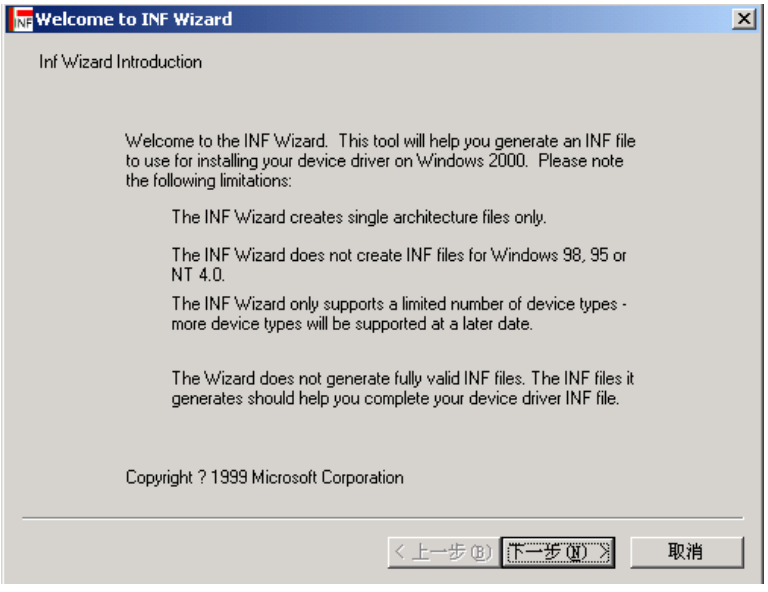


图 12-15　geninf 实用程序运行界面

12.6　本章小结

在本章中介绍了 USB 设备驱动程序的开发流程，并且利用 Windows 2000 DDK 作为开发工具，给出了驱动程序的部分代码。同时，详细介绍了设备驱动程序引导文件（.INF）的编写步骤。