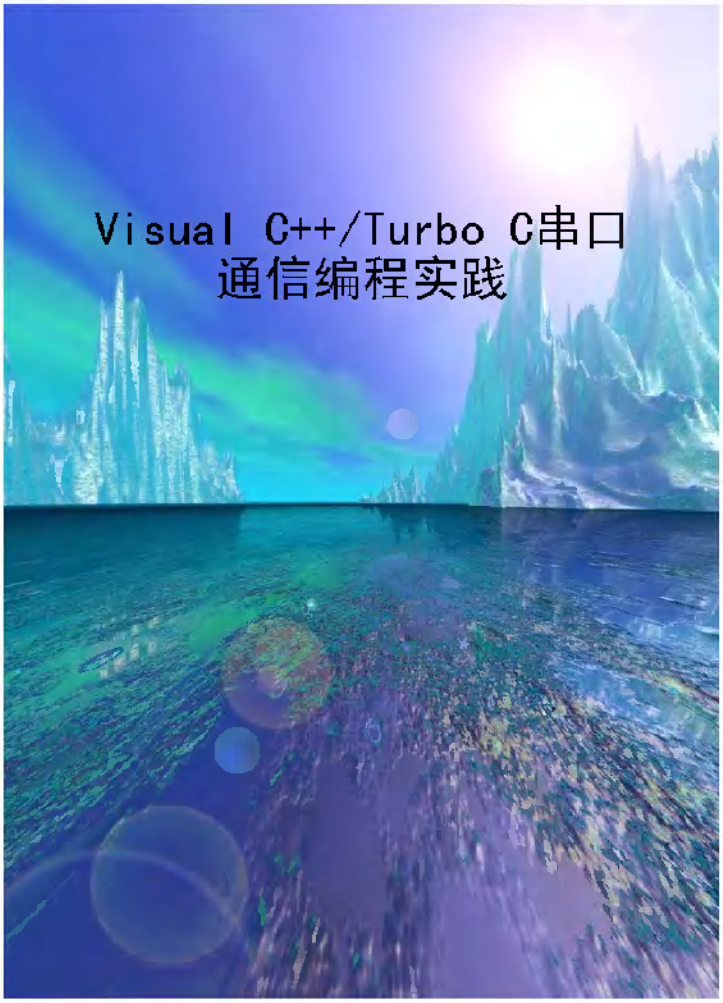


Visual C++/Turbo C串口通信编程实践

Visual C++/Turbo C 串口通信编程实践

龚建伟　熊光明　编著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

本书从编程实践的角度详细介绍了 Windows 环境下和 DOS 环境下的串口通信的基本方法，并根据当前串口与网络结合的发展趋势，介绍了串口与网络 TCP/IP、远程控制与监测相结合的一些解决方案和编程要点。由于编程步骤详尽，初学 Visual C++/C（甚至是以前完全没有接触过 Visual C++）的读者也能很快编写出 Visual C++的串口通信程序。本书配光盘，书中实例源程序和相关资料可在对应章节的文件夹中找到。

本书是从事串口及网络通信的技术人员和学习者的极佳参考资料，也可以作为数据通信课程的辅助教材。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Visual C++/Turbo C 串口通信编程实践 / 龚建伟，熊光明编著. —北京：电子工业出版社，2004.10
ISBN 7-121-00253-1

Ⅰ.V… Ⅱ.①龚…②熊… Ⅲ.C语言—程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2004）第 083810 号

责任编辑：朱沐红
印　　刷：北京东光印刷厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
经　　销：各地新华书店
开　　本：787×1 092　1/16　印张：24.5　字数：604 千字
印　　次：2004 年 10 月第 1 次印刷
印　　数：5000 册　定价：55.00 元（含光盘 1 张）

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

前　言

本书内容

本书从编程实践的角度详细介绍了 Windows 环境下和 DOS 环境下的串口通信的基本方法，并根据当前串口与网络结合的发展趋势，介绍了串口与网络 TCP/IP、远程控制与监测相结合的一些解决方案和编程要点。

本书共分三部分，第一部分是编程实例，包括前十章，从体验轻松的串口编程开始，在有了一定的感性知识后，由浅入深地介绍了各种编程方法，并将一些串口通信的基础知识融入其中，使读者不知不觉中掌握了许多基本概念。考虑到当前 DOS 环境下的串口编程还应用很广，本书也深入地介绍了 DOS 环境下 Turbo C 的编程。串口与网络通信总是伴随着数据包的处理和存储，因此本书也介绍了各种协议的数据通信包处理实例。有不少编程知识在 Windows 和 DOS 环境下是通用的。串口通信与网络通信相结合是当前串口通信的发展趋势，所以本书以具体产品实例介绍了串口网络化的解决方案。

第二部分为第 11 章，汇集了常用串行通信的基本概念和规范，介绍了各种接线匹配方法，将第一部分中分散的基础知识进行了总结。

第三部分是一些较深层次技术问题的探讨，其中对不占用串口资源的串口数据捕捉给出了实例程序，并简要介绍了虚拟串口的应用，并对这些问题的相关知识进行了必要的说明。

附录介绍了 Turbo C 3.0/2.0 的用法，列出了常用的 ASCII 码表，供通信编程的较高层次学习者使用。本书光盘中收录了书中可独立运行的源代码和相关程序资料，方便读者学习。

本书特色

在本书写作中，作者有意识地重点介绍了一些应用简单但功能很强的类，如果读者不想花很多时间了解串口通信的底层，而急于在短时间内（比如 1 个小时）完成编程任务，不管在 Windows 或 DOS 环境下，您应用这些类来解决问题无疑是最佳的选择。

本书读者对象为从事串口及网络通信的技术人员和学习者，由于编程步骤详尽，初学 Visual C++/C（甚至是以前完全没有接触过 Visual C++）的读者也能很快编写出 Visual C++ 的串口通信程序，同时，本书也可以作为数据通信课程的辅助教材。

串口调试助手是作者自己编制的软件，是一个工程技术人员和学习者广泛应用的串口调试工具，本书中放入了该软件的源代码和详细编程步骤。

最后要强调的是：数据通信技术无论是串行通信还是网络通信，其本质内容都是类似的，

即数据处理及用户层的通信协议是一致的；学习过程中经常能做到触类旁通，这一点，我们在通信编程中要注意体会。

致谢

除署名作者外，本书第 10 章的 10.1 节完全来自科脑工作室，还摘录了一些参考文献的内容，同时还得到了 MOXA 公司（http://www.moxa.com.cn）的串口联网产品信息和技术支持，在此谨表谢意。

技术支持

作者近几年在自己的日访问量近千次的个人技术主页（http://www.gjwtech.com）中收到很多朋友的问题，部分典型的问题可以在本书中找到答案。欢迎读者到作者个人技术主页上进行交流。

作　者

2004.07 于北京

目　录

第 1 章　轻松体验串口通信编程与调试 1
1.1　使用串口调试助手来体验串口通信 1
1.2　体验 Windows 环境下的 Visual C++串口通信编程 4
1.3　体验 DOS 环境下 Turbo C 串口通信编程 12
第 2 章　多线程串口编程工具 CSerialPort 类 16
2.1　CSerialPort 类的功能及成员函数介绍 16
2.2　应用 CSerialPort 类编制基于对话框的应用程序 30
2.3　应用 CSerialPort 类编制基于单文档的应用程序 35
2.4　对 CSerialPort 类的改进 40
2.4.1　改进一：ASCII 文本和二进制数据发送方式兼容 40
2.4.2　改进二：也许能解决内存泄漏 43
2.4.3　改进三：彻底关闭串口，释放串口资源 44
第 3 章　控件 MSComm 串口编程 46
3.1　MSComm 控件介绍 46
3.1.1　VC 中应用 MSComm 控件编程步骤 46
3.1.2　MSComm 控件串行通信处理方式 47
3.1.3　MSComm 控件的属性说明 48
3.1.4　MSComm 控件错误信息 55
3.2　使用 MSComm 控件的几个疑难问题 56
3.2.1　使用 VARIANT 和 SAFEARRAY 数据类型从串口读写数据 56
3.2.2　MSComm 控件能离开对话框独立存在吗 59
3.2.3　如何发送接收 ASCII 值为 0 和大于 128 的字符 60
3.2.4　在同一程序中用 MSComm 控件控制多个串口的具体操作方法 62
3.2.5　解决使用控件编程时程序占用的内存会不断增大的问题 62
3.2.6　在 MSComm 控件串口编程时遇到的其他问题 63
3.3　在基于单文档（SDI）程序中应用 MSComm 控件 63
3.4　应用 MSComm 控件控制多个串口实例 69
3.5　串口与 MODEM 拨号应用简例 76
3.5.1　创建工程 76
3.5.2　代码分析 78

3.5.3 应用 85
第 4 章 Windows API 串口编程 87
4.1 Windows API 串口编程概述 87
4.2 API 串口编程中用到的结构及相关概念说明 89
4.2.1 DCB（Device Control Block）结构 89
4.2.2 超时设置 COMMTIMEOUTS 结构 92
4.2.3 OVERLAPPED 异步 I/O 重叠结构 94
4.2.4 通信错误与通信设备状态 95
4.2.5 串行通信事件 96
4.3 Windows API 串行通信函数 97
4.4 Win32 API 串口通信编程的一般流程和特殊实例 116
4.4.1 Win32 API 串口通信编程的一般流程 116
4.4.2 用查询方式读串口 116
4.4.3 同步 I/O 读写数据 117
4.4.4 关于流控制的设置问题 118
4.5 CSerialPort 类中的 API 函数编程应用剖析 119
4.6 Win32 API 串口编程 TTY（虚拟终端）实例 128
4.6.1 建立程序工程 128
4.6.2 建立串口设置对话框 129
4.6.3 编写 CTermDoc 类的相关代码 132
4.6.4 小结 141
4.6.5 在 CTermView 类中字添加符键入处理代码与串口接收处理代码 142
第 5 章 串口调试助手 V2.2 编程 147
5.1 建立 SCOMM 程序工程实现界面功能 147
5.2 串口的初始化及关闭 150
5.3 串口数据的发送与接收及十六进制数据的处理 151
5.3.1 十六进数据发送处理 152
5.3.2 手动发送处理 152
5.3.3 自动发送处理 153
5.3.4 接收处理及十六进制显示 154
5.4 其他辅助功能的实现 156
5.4.1 接收数据的文件保存 156
5.4.2 实现小文件发送 158
5.4.3 图钉按钮功能使程序能浮在最上层 161

5.4.4 对话框动画图标的实现 162
5.4.5 超链接功能的实现 164
5.4.6 如何打开帮助网页文件 164
第 6 章 DOS 环境下的 Turbo C 串口编程及通用实例 GSerial 类 168
6.1 PC 机异步通信适配器 8250 及其编程操作 169
6.1.1 INS8250 内部寄存器及其选择方式 169
6.1.2 波特率设置 169
6.1.3 数据位、奇偶校验、停止位等数据格式设置 170
6.1.4 查询 I/O 方式相关设置 171
6.1.5 中断 I/O 通信方式相关设置 171
6.1.6 MODEM 寄存器 172
6.2 COMRXTX 程序实例 173
6.3 通用实例程序 GSerial 类 175
6.4 用 GSerial 类控制多串口 186
6.5 多串口编程 PC 机高号中断 8259A 可编程中断控制器的控制 195
第 7 章 串口通信用户层协议的编制与数据处理方法 197
7.1 通信协议的编制 197
7.1.1 为什么要编制用户通信协议 197
7.1.2 串口通信中用户层协议编制原则 199
7.1.3 在串口通信中几种常用的用户层协议 200
7.2 串口通信数据包处理方法编程实例 202
7.2.1 编程任务 203
7.2.2 编程步骤 203
7.2.3 程序测试 216
第 8 章 单片机串口通信 218
8.1 单片机串口硬件系统及 C51 程序开发 218
8.1.1 较典型的单片机硬件系统实例 218
8.1.2 C51 语言及程序简介 220
8.1.3 开发 C51 程序的利器 Keil C51 uVision2 及串口程序仿真 221
8.2 C51 单片机串口通信程序实例 226
8.2.1 实例一 226
8.2.2 实例二 227
第 9 章 串口与网络结合的解决方案及编程 230
9.1 串口与网络结合的硬件解决方案 230

9.2 典型串口与联网的设备 231
9.2.1 NPort5400 系列产品的特点 231
9.2.2 NPort 5400 系列产品的典型应用介绍 233
9.2.3 NPort5400 系列产品的设置与编程测试 235
9.3 与 Access 数据库结合的串口通信实例 237
9.3.1 微机网络检测系统说明 237
9.3.2 创建 ODBC 数据源 238
9.3.3 创建工程 239
9.3.4 程序简介 244
9.4 与 WinSock 结合的串口通信实例 246
9.4.1 客户端应用程序 247
9.4.2 服务器应用程序 252
9.5 在已经编好的串口通信程序中加入网络通信功能 260
9.5.1 参照 MFC AppWizard 创建 WinSockets 程序 261
9.5.2 利用 Windows Sockets API 和第三方提供的类进行编程 262
9.6 串口通信用于遥控操作简例 262
第 10 章 计算机串口与其他设备通信编程实例 266
10.1 通过串口收发短消息 266
10.1.1 SMS 编码规范及编码与解码例程 266
10.1.2 AT 命令收发短消息实例 273
10.1.3 “实时”接收短消息的方法 281
10.1.4 用串口收发 SMS 短信编程的一些讨论 283
10.2 计算机与 Rabbit 2000 嵌入式系统通信编程实例 286
10.2.1 Rabbit 2000 微处理器介绍 286
10.2.2 动态 C（Dynamic C）语言介绍 287
10.2.3 某车载无线调度系统实例介绍 288
10.3 计算机与 PLC 通信程序实例 294
10.4 MATLAB 环境串口编程通信实例 295
10.4.1 MATLAB 串口类 Serial 应用 295
10.4.2 通过串口使 MATLAB Simulink 与下位机通讯进行控制 299
10.4.3 xPC 目标环境下串口通信实现 299
第 11 章 串口通信基本概念及标准 302
11.1 串口通信基本概念 302
11.1.1 串行通信概述 302

11.1.2 单工、半双工和全双工的定义 305
11.1.3 同步传送与异步传送 306
11.1.4 串行通信协议 306
11.2 RS-232-C 串口标准 309
11.2.1 RS-232-C 标准 309
11.2.2 RS-232-C 串行通信接线实例 312
11.3 RS-422/485 串口标准 314
11.3.1 概述 314
11.3.2 RS-422 与 RS-485 串行接口标准 315
11.3.3 RS-422 与 RS-485 的网络安装注意要点 317
11.3.4 RS-232、RS422、RS485 电气参数对比 318
11.4 串口调试注意事项 318
11.5 常用数据校验法 318
11.5.1 奇偶校验 318
11.5.2 循环冗余码校验 319
11.6 串口连接和 TCP/IP 连接对比 320
11.7 现场总线与 RS-232、RS-485 的本质区别 320
11.8 MODEM 通信技术 320
11.8.1 MODEM 的基本工作原理 320
11.8.2 MODEM 的功能 322
11.8.3 MODEM 的分类 322
11.8.4 MODEM 的安装 324
11.8.5 MODEM V.92 标准介绍 326
11.8.6 MODEM 的速度 327
11.8.7 MODEM 优化方法 328
11.8.8 MODEM 命令/AT 命令 329
第 12 章 不占用串口的串口数据捕捉 338
12.1 驱动程序的基本概念：VxD 与 WDM 338
12.1.1 虚拟设备驱动程序 VxD 338
12.1.2 Win32 驱动程序模型 WDM 340
12.1.3 在不同操作系统下选用哪种驱动程序模式 341
12.2 VxD 示例程序介绍——VToolsD 中的 CommHook 341
12.3 串口数据捕捉实例程序 351
12.3.1 编程任务 351

12.3.2 编程步骤 351
12.4 虚拟串口简介 364
附录 A Turbo C 说明 366
附录 B ASCII 码表 376

第 1 章　轻松体验串口通信编程与调试

[内容提要]

首先，我们用串口调试助手来体验串口通信，熟悉这个调试工具是必要的，以后所有的实例都可以通过它来测试。

然后，编写一个串口通信小程序，没有用 VC 编写过串口通信程序的读者可以来感受一下用 VC 编写串口程序是多么简单的事情。您不需要对 VC 很熟练，只要能启动 VC 就可以了；以前从未接触过串口数据通信也没关系，只要我们一步一步编写并运行完这个程序，您就能有“不过如此”的感觉了，至于其他的概念，我们慢慢在编程中掌握。

本章最后，我们还要编写一个 DOS 环境下的串口程序，体验一下 Windows 环境与 DOS 环境下串口编程的区别。了解这些，对在各种应用场合编写串口通信程序是有帮助的。

当然，还应该具备以下条件：

■　**硬件环境**

能运行 Windows 9X/2000/XP 的 PC 机，配有两个串行口（可以是同一台 PC 机或分属于两台 PC 机的两个串口，串口连接线（若身边没有，可以参考第 11.2.2 节 3 线制连线方法自己制作一根连接线）。

■　**软件环境**

Visual C++ 6.0, Windows 9X/2000/XP

对于初次接触串口通信的读者，特别要注意：

注意：串口线插拔时，请关闭计算机电源，以免烧毁计算机的串行电路。自己制作的串口连线，使用前，用万用表测试一下焊接是否有错。

1.1　使用串口调试助手来体验串口通信

首先来看串口调试助手的功能。一旦熟悉了这个工具，你以后就会省很多事，所以值得花点时间。

串口调试助手是一个专门用来调试串口程序的工具软件，由本书作者编写，目前有大量工程人员和软件编程人员在使用，在作者技术主页和各大软件下载网站均可下载。其界面如图 1.1.1 所示，使用平台为 Windows 9X/NT/2000/XP，2.2 版本的程序仅供三线制（NONMODEM）串口调试之用，所有功能均置于界面上，一目了然，其义自明。界面分为串口设置区（左上角）、接收显示区和发送输入区，下面分别说明。

图 1.1.1 串口调试助手 V2.2

■ 串口设置区

在串口设置区可以设置串口、波特率、检验位、数据位及停止位等串口通信参数。参数设置好后，可以“打开串口”，若正常，则指示灯亮，显示为 ；否则会提示“串口不存在或被其他设备占用”，指示灯灭，显示为 。

■ 接收显示区

接收区可设置为自动清空，也可手动清空。设置为自动清空时，每接收一定行数后，清空接收区。为了看清楚接收的数据，可以让显示暂时停止（但数据仍在接收），之后可以继续显示。默认显示格式为 ASCII 码显示，也可以设置为十六进制显示，十六进制显示时，每个十六进制之间会有一个空隔，如 01 23 00 34 A5。接收的数据可以保存成文本文件，单击“保存显示数据”即可实现这一功能。默认保存文件夹为 C:\COMDATA，可以通过“更改”按钮改变保存的文件夹，以文件名 REC01.TXT，REC02.TXT，REC03.TXT 等（后面序号自动递增）保存。

■ 发送输入区

发送输入区可以填写需要发送的数据。输入要发送的字符数据后，单击“手动发送”按钮，则普通文本可以发送一次；若选中了“自动发送”，则会每隔设定的发送周期内发送一次，直到去掉“自动发送”为止。值得注意的一点是，选中“十六进制发送”后，发送框中所填字符每两个字符之间应有一个空隔，如：01 23 00 34 45。还有一些特殊的字符，如回车换行，则直接敲入回车即可。

另外，为了方便调试，程序设置了一些小功能。如窗口悬浮功能：单击程序左下角的图钉按钮，可以使程序置于最上层，保持可见，这时图钉按钮形状由 变为 。

程序还可放大至全屏，当需要扩大接收窗口以方便观看数据时，可以单击右上角最大化按钮；在程序的最下端，还有目前串口状态及接收与发送的字符数，计数可以清零。

现在，我们就要利用串口调试助手来体验一下串口通信了。普通计算机一般使用 9 针串口，如图 1.1.2 所示，如果有两台计算机，则可以用串口线连接起来，若是一台计算机上有两个串口，也可以把它们直接连接起来就可以了。

注意： 再次提醒插拔串口线时，别忘记关掉计算机电源。

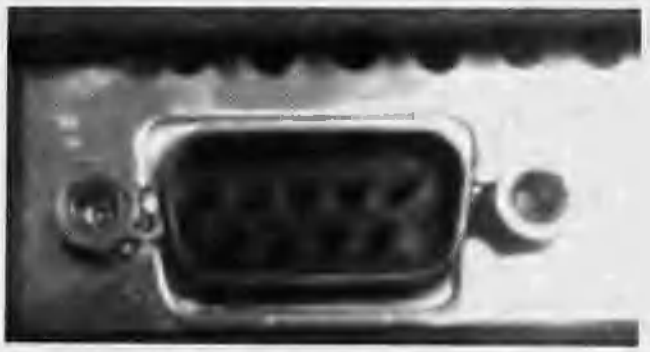


图 1.1.2　PC 机上 9 针串口的形状

接好串口线后，打开计算机，启动串口调试助手；若在同一计算机上，则将先启动的串口号设置为 COM2，再次启动串口调试助手（等于在同一计算机上启动两个程序），并将串口号设置为 COM1，实际设置时，串口号要根据自己的计算机配置设定相应的串口号。通信双方的串口参数要一致。下面，在一个串口调试助手的输入框输入 12345678ABCDEFG（加上回车换行），并选中“自动发送”，一切顺利的话，程序运行情况应如图 1.1.3 所示。读者还可以自己测试一下其他的功能：十六进制发送与接收、数据保存等，这些在今后实际调试中经常会用到。

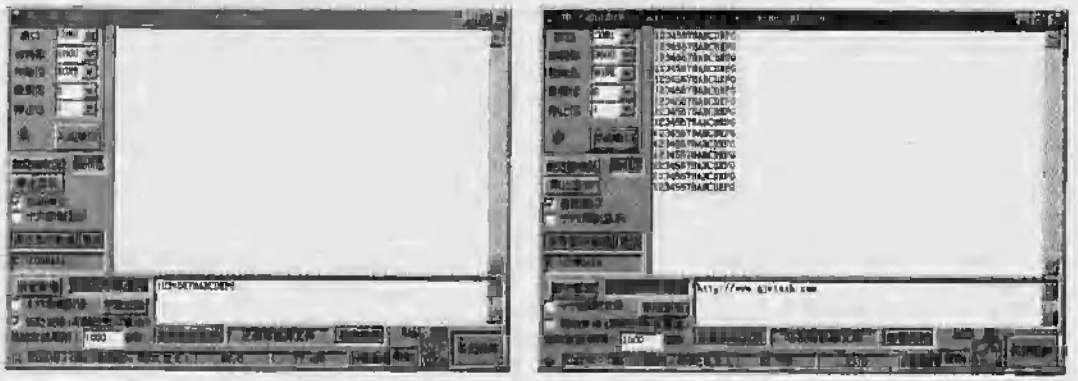


图 1.1.3　串口调试助手发送接收测试

到这里，我们可以对自己说：“我能用串口调试助手接收发送数据了。”这在以后调试许多设备时都是有用的，当然，更便于我们调试自己编写的程序。接下来，我们就要花半小时来做自己的串口程序了。我希望读者读完这本书后，会在几分钟之内写出自己的串口程序，就像我一样，有了这个基础，我们以后编写 TCP/IP 的程序，会一样很快的。是啊，有什么能

比自己编写的程序顺利运行更能让我们高兴的呢！

1.2　体验 Windows 环境下的 Visual C++串口通信编程

好的，让我们行动起来。如果你是初次接触串口通信编程，就跟着我一步一步来编写这个程序，为了方便学习，尽量与我的源代码保持一致。在这个程序中，应用了 VC 自带的 MSComm 控件，先不必去管 MSComm 控件的具体特性，第 3 章中会对其做详细介绍。

1．建立应用程序工程 SCommTest

打开 Visual C++ 6.0，建立一个基于对话框的 MFC 应用程序：SCommTest。然后在主对话框中添加控件，最后效果如图 1.2.1 所示。其中的电话状图标是 MSComm 控件，参照第 2 步的方法添加到对话框中。

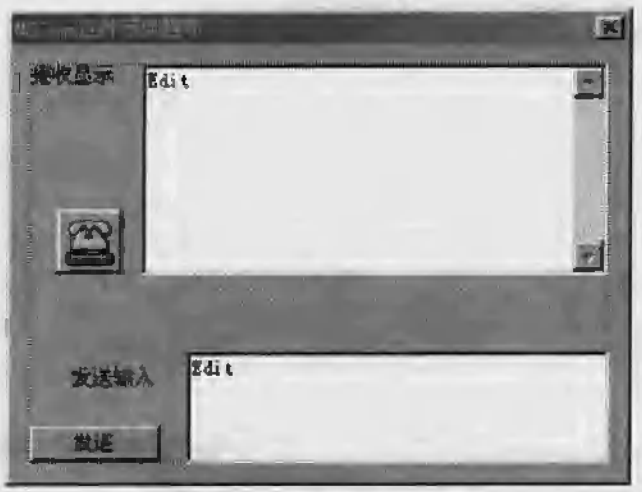


图 1.2.1　对话框添加控件后的状态

然后用 ClassWizard 为相应控件添加变量，控件的属性设置情况如表 1-2-1 所列。

表 1-2-1 控件及其属性设置情况

控件	控件 ID	Caption	需要添加的变量及变量类型
静态文本	IDC_STATIC	接收显示	
静态文本	IDC_STATIC	发送输入	
编辑框	IDC_EDIT_RXDATA		m_strEditRXData Value CString
编辑框	IDC_EDIT_TXDATA		m_strEditTXData Value CString
按钮	IDC_BUTTON_MANUALSEND	发送	
MSComm 控件	IDC_MSCOMM1		m_ctrlComm　control

2．在当前工程中添加 MSComm 控件

单击菜单 Add To Project-> Components and Controls...，就打开了如图 1.2.2 所示的添加组（控）件对话框。

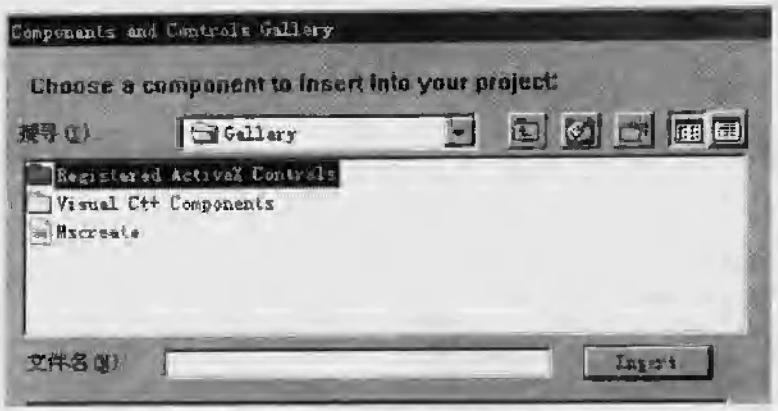


图 1.2.2　添加组件、控件对话框

再在其中双击“Registered ActiveX Controls”项（这时要稍等一会，这个过程较慢），就出现了如图 1.2.3 所示的控件选择（Component and Controls Gallery）对话框。在该对话框中选择“Microsoft Communications Control, version 6.0”控件。

注意：如果在控件列表中看不到 Microsoft Communications Control, version 6.0，那可能是你在安装 VC6 时没有把 ActiveX 一项选上，这时需要重新安装 VC6，选上 ActiveX 就可以了。

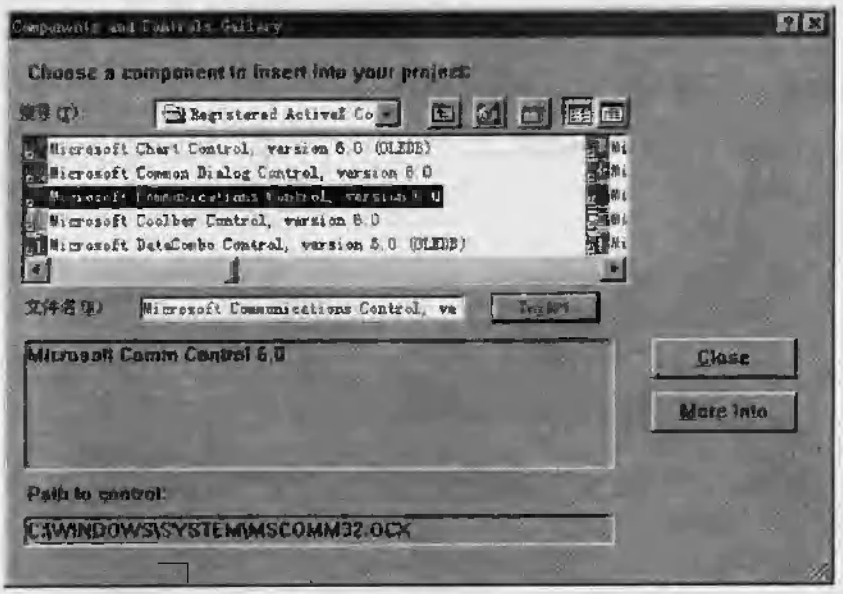


图 1.2.3　控件选择对话框

再单击“Insert”按钮，提示“Insert this component?”，确认后，可以看到如图 1.2.4 所示的加入 CMSComm 类的确认（Confirm Class）对话框，提示加入到当前工程中的 CMSComm 类头文件为 MSComm.h，实现文件为 MSComm.cpp。单击“OK”按钮，（Confirm Class）对话框关闭。再单击“Close”关闭（Component and Controls Gallery）对话框。在 VC 集成环境

中，当前工程的 ClassView 中就出现了 CMSComm 类。同时，在对话框资源控件中出现了一个电话机形状的控件（如图 1.2.5 所示），这就是 MSComm 控件。要在对话框中应用该控件，还需要将这个控件图标用鼠标拖入对话框中，这个对话框就成了 MSComm 控件的“宿主”。

提示：利用这种添加控件的方法，对之后的串口消息事件处理会提供很大的方便，ClassWizard 会自动在当前程序工程中进行消息类的映射，这一点我们在以下的编程中会体会到。

图 1.2.4　添加 CMSComm 类的确认对话框

图 1.2.5　MSComm 控件出现在工程资源中

3. 初始化串口：设置 MSComm 控件的属性

打开 ClassWizard->Member Variables 页，如图 1.2.6 所示，选中控件 IDC_MSCOMM1，再单击“Add Varialbe...”按钮，在 CSCommTestDlg 类中为控件 IDC_MSCOMM1 添加 CMSComm 控制变量 m_ctrlComm。

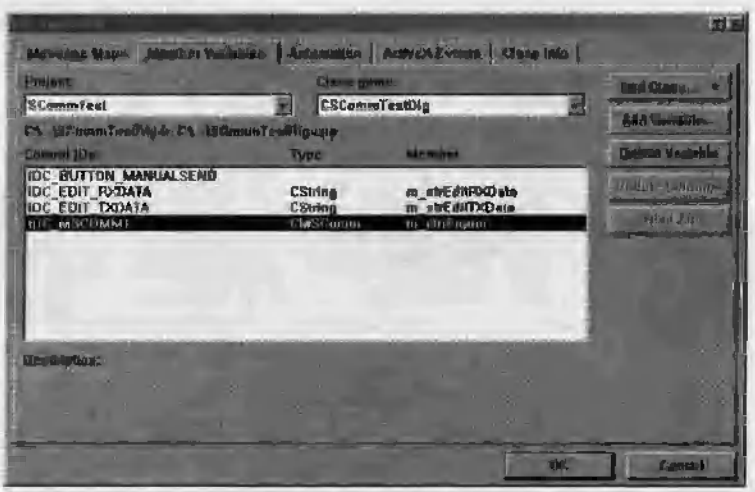


图 1.2.6　为控件 IDC_MSCOMM1 添加控制变量

通过以上操作，ClassWizard 自动在 SCommTestDlg.h 中加入了#include "mscomm.h"语句。

```
//{{AFX_INCLUDES()
#include "mscomm.h"
//}}AFX_INCLUDES
```

下面，在 CSCommTestDlg::OnInitDialog()函数中写入对串口的初始化语句，串口初始化语句由 IDC_MSCOMM1 的 CMSComm 控制变量 m_ctrlComm 来设置串口控件属性。代码如下：

```
BOOL CSCommTestDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
      … …
    // TODO: Add extra initialization here
    m_ctrlComm.SetCommPort(1);   //选择 COM1
    //波特率 9600，无校验，8 个数据位，1 个停止位
    m_ctrlComm.SetInputMode(1); //输入方式为二进制方式
    m_ctrlComm.SetInBufferSize(1024); //设置输入缓冲区大小
    m_ctrlComm.SetOutBufferSize(512); //设置输出缓冲区大小
    //波特率 9600，无校验，8 个数据位，1 个停止位
    m_ctrlComm.SetSettings("9600,n,8,1");
    //参数 1 表示每当串口接收缓冲区中有多于
    //或等于 1 个字符时将引发一个接收数据的 OnComm 事件
    if(!m_ctrlComm.GetPortOpen())
        m_ctrlComm.SetPortOpen(TRUE);//打开串口
    m_ctrlComm.SetRThreshold(1);
    m_ctrlComm.SetInputLen(0);  //设置当前接收区数据长度为 0
    m_ctrlComm.GetInput();    //先预读缓冲区以清除残留数据

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

4．添加串口事件消息处理函数 OnComm()

MSComm 控件一般用事件驱动方式从串口接收数据，也就是消息处理，当串口有事件发生时，程序调用消息函数来处理数据。打开 ClassWizard->Member Variables 页，如图 1.2.7 所示，打开 ClassWizard—>Message Maps，在 Class Name 中选择类 CSCommTestDlg，再在 Object IDs 中选择 IDC_MSCOMM1，然后在 Message 中双击消息 OnComm（或单击“Add Function”按钮），在弹出的对话框中将函数名改为 OnComm（好记而已），单击“OK”，就加入了串口事件的消息处理函数。

我们注意到，这时自动添加了以下代码：

头文件 SCommTestDlg.h 中：

```
// Generated message map functions
//{{AFX_MSG(CSCommTestDlg)
… …
afx_msg void OnComm();
   … …
DECLARE_EVENTSINK_MAP()
//}}AFX_MSG
```

实现文件 SCommTestDlg.cpp 中：

```
BEGIN_EVENTSINK_MAP(CSCommTestDlg, CDialog)
```

```
    //{{AFX_EVENTSINK_MAP(CSCommTestDlg)
     ON_EVENT(CSCommTestDlg, IDC_MSCOMM1, 1 /* OnComm */, OnComm, VTS_NONE)
     //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()

void CSCommTestDlg::OnComm()
{
    // TODO: Add your control notification handler code here
}
```

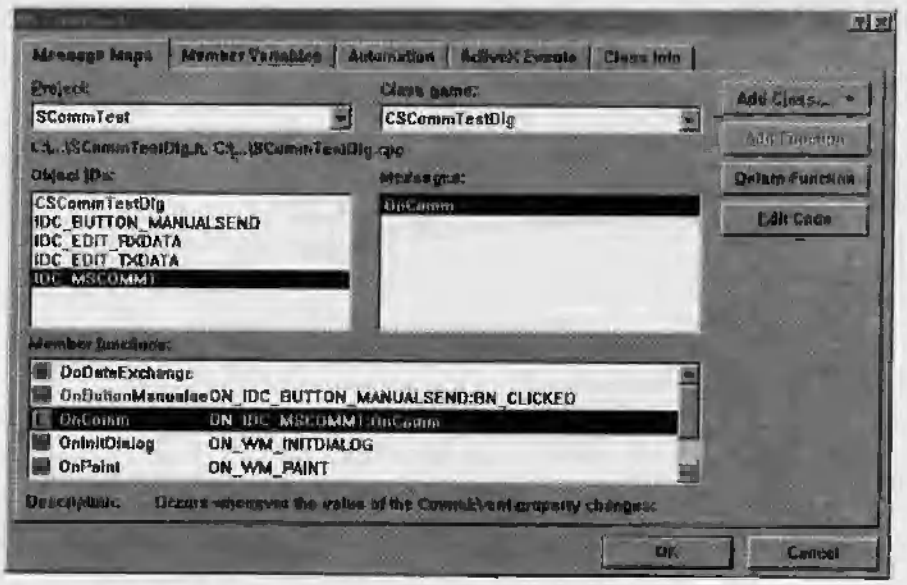


图 1.2.7　为控件 IDC_MSCOMM1 添加消息事件处理函数 OnComm()

下面编写函数 OnComm()中的代码，主要任务是从串口接收数据并显示在接收编辑框中。

```
void CSCommTestDlg::OnComm()
{
    // TODO: Add your control notification handler code here
    VARIANT variant_inp;
    COleSafeArray safearray_inp;
    LONG len,k;
    BYTE rxdata[2048]; //设置BYTE数组
    CString strtemp;
    if(m_ctrlComm.GetCommEvent()==2) //事件值为2表示接收缓冲区内有字符
    {
        variant_inp=m_ctrlComm.GetInput(); //读缓冲区
        safearray_inp=variant_inp;  //VARIANT型变量转换为ColeSafeArray型变量
        len=safearray_inp.GetOneDimSize(); //得到有效数据长度
        for(k=0;k<len;k++)
            safearray_inp.GetElement(&k,rxdata+k);//转换为BYTE型数组
        for(k=0;k<len;k++)            //将数组转换为Cstring型变量
        {
            BYTE bt=*(char*)(rxdata+k);   //字符型
            strtemp.Format("%c",bt);    //将字符送入临时变量strtemp存放
            m_strEditRXData+=strtemp;  //加入接收编辑框对应字符串
        }
    }
```

```
    UpdateData(FALSE); //更新编辑框内容
}
```

5. 发送数据

先为发送按钮添加一个单击消息，即 BN_CLICKED 处理函数，打开 ClassWizard→Message Maps，选择类 CSCommTestDlg，选中 IDC_BUTTON_MANUALSEND，双击 BN_CLICKED 添加 OnButtonManualsend()函数，如图 1.2.8 所示。

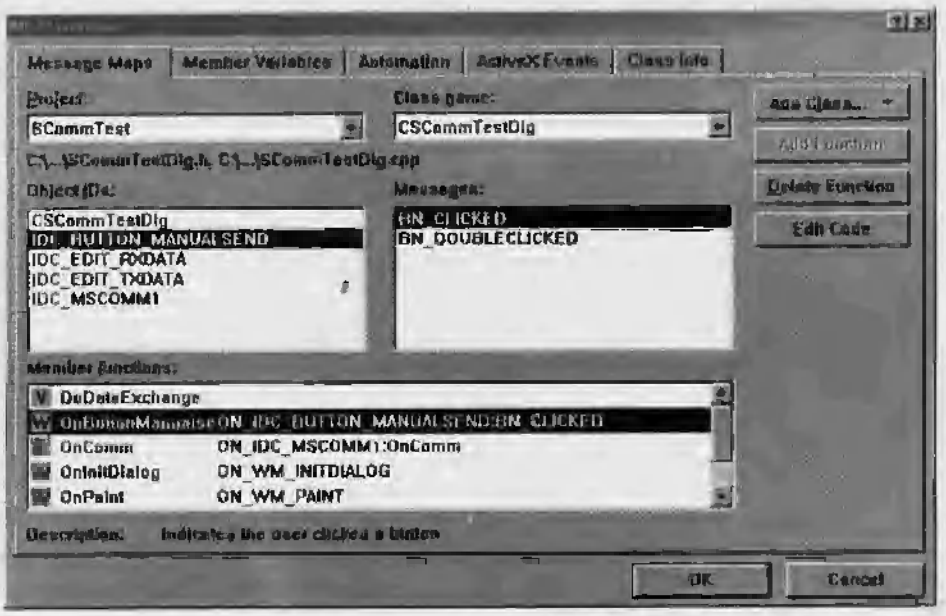


图 1.2.8　添加 OnButtonManualsend()函数

然后，在函数中添加如下代码：

```
void CSCommTestDlg::OnButtonManualsend()
{
  // TODO: Add your control notification handler code here
  UpdateData(TRUE); //读取编辑框内容
  m_ctrlComm.SetOutput(COleVariant(m_strTXData));//发送数据
}
```

全部程序编写完成后，SCommTestDlg.h 文件代码如下：

```
// SCommTestDlg.h : header file
//
//{{AFX_INCLUDES()
#include "mscomm.h"
//}}AFX_INCLUDES

#if !defined(AFX_SCOMMTESTDLG_H__22F2B146_69C2_11D5_870E_00E04C3F78CA__INCLUDED_)
#define
AFX_SCOMMTESTDLG_H__22F2B146_69C2_11D5_870E_00E04C3F78CA__INCLUDED_
```

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

/////////////////////////////////////////////////////////////////////////////
// CSCommTestDlg dialog

class CSCommTestDlg : public CDialog
{
// Construction
public:
    CSCommTestDlg(CWnd* pParent = NULL);  // standard constructor

// Dialog Data
    //{{AFX_DATA(CSCommTestDlg)
    enum { IDD = IDD_SCOMMTEST_DIALOG };
    CMSComm   m_ctrlComm;
    CString   m_strEditRXData;
    CString   m_strEditTXData;
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CSCommTestDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    HICON m_hIcon;

    // Generated message map functions
    //{{AFX_MSG(CSCommTestDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg void OnComm();
    afx_msg void OnButtonManualsend();
    DECLARE_EVENTSINK_MAP()
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
//Microsoft Visual C++ will insert additional declarations immediately before the previous
line.

#endif
// !defined(AFX_SCOMMTESTDLG_H__22F2B146_69C2_11D5_870E_00E04C3F78CA__INCLUDED_)
```

需要两个串口来测试程序。这两个串口可以在一台计算机上，也可以分别在两台计算机上，用串口线将其连接。图 1.2.9 所示的情况为运行本程序控制 COM1，在发送编辑框中输入 1234567890ABCDEFG，再打开串口调试助手控制 COM2，单击“发送”按钮，再单击串口调试助手的“手动发送”按钮，程序运行正常。

图 1.2.9　利用串口调试助手测试串口接收与发送功能

图 1.2.10 所示的测试情况为先将串口号设置为 COM1，运行程序；再将语句：

```
m_ctrlComm.SetCommPort(1); //COM1
```

修改为：

```
m_ctrlComm.SetCommPort(2); //COM2
```

再运行程序，并在两个程序的“发送输入”框中分别填上“串口 1：MSComm 控件测试程序”和“串口 2：MSComm 控件测试程序”，单击“发送”按钮。当然，如果在界面上有串口选择功能，就不必那么麻烦了。

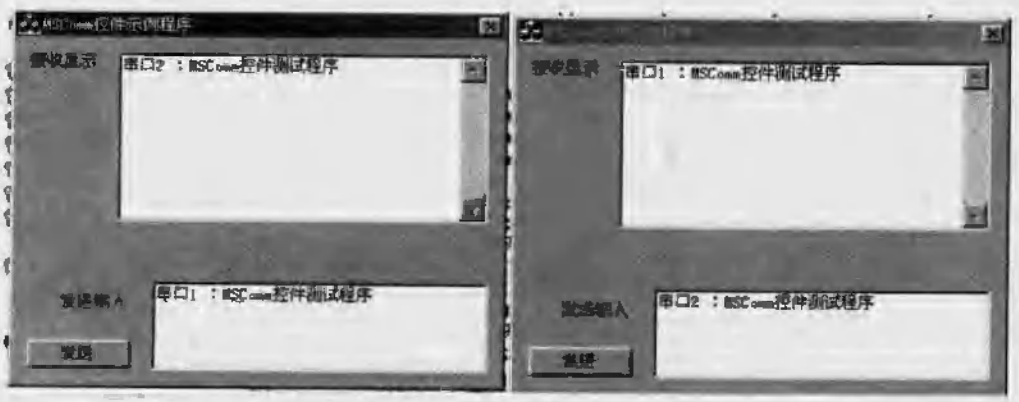


图 1.2.10　利用程序本身进行测试

好的，我们的第一个串口程序就成功运行了，在高兴的同时，还要注意，这只不过是初步的东西，我们真正要做的是如何把数据取出来，把一定格式的命令字符或数据发送出去。编程也是初步的，以后还有一些深层的知识需要了解。

1.3　体验 DOS 环境下 Turbo C 串口通信编程

目前，单片机和嵌入式系统的应用越来越广，且编程语言绝大部分为 C 语言，其编程方式大多与 DOS 环境下 C 语言编程大同小异，所以，了解 DOS 下串口通信编程方式也是十分必要的，这样，我们才可以不惧任何编程环境的挑战了。

在本书中，使用 Turbo C++ 3.0 编译器来调试程序，目前使用较多的是 Turbo C 2.0 和 Turbo C++ 3.0，推荐使用后者。Turbo C++ 3.0 不仅编程界面比 Turbo C 2.0 友好不少，可以使用鼠标，而且 C 语句与 C++语句兼容，如可以在编程中使用类，这就能让我们应用大量的程序源代码资源，这同时也是 C 语言的发展趋势，初学者更是可以通过它来学习 C++语言编程。不熟悉 Turbo C++3.0 编译环境的读者可以先看一下附录 A，了解 Turbo C++ 3.0 和 Turbo C 2.0 的使用方法及安装注意事项。

DOS 下串口通信属于底层编程，一般有中断和查询两种通信方式，作为初始的体验，在这个程序中用了中断方式。这个程序涉及许多硬件的设置问题，读者先不必做详细了解，第 6 章中会有详细介绍，这里只要体验一下 DOS 操作系统下的串口通信。

程序如下（源代码可在本书所附光盘的第 1 章找到：COMRX.CPP）：

```
//////////////////////////////////////////////////////////////////
//COMRX.CPP for asyn serial communication (only RX)
//edited by Xiong Guangming and Gong Jianwei
//Turbo C++3.0
//////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <dos.h>
#include <conio.h>

#define BUFFLEN 1024

void InitCOM();  //初始化串口
void OpenPort();  //打开串口
void ClosePort();  //关闭串口，释放串口资源
//新的中断函数，注意在TC2.0下，下面函数的...要去掉
void interrupt far asyncint(...);
//中断向量：用于保存中断现场
void interrupt(*asyncoldvect)(...);

unsigned char Buffer[BUFFLEN];
int buffin=0;
int buffout=0;
//unsigned char ch;

//COM1 产生的硬件中断号为 IQR4，对应的中断向量为 0CH
//打开 COM1
void OpenPort()
{
    unsigned char ucTemp;
    InitCOM();  //初始化串口
```

```
    //读入由参数给定的中断向量值，并将它作为中断函数的远地址
    asyncoldvect=getvect(0x0c);
    disable();      //关中断
    inportb(0x3f8);
    inportb(0x3fe);
    inportb(0x3fb);
    inportb(0x3fa);
    outportb(0x3fc,0x08|0x0b);
    outportb(0x3f9,0x01);
    ucTemp=inportb(0x21)&0xef;
    outportb(0x21,ucTemp);
    setvect(0x0c,asyncint);
    enable();       //开中断
}

//中断服务程序，从COM1接收数据
//注意在TC2.0下，下面函数的...要去掉
void interrupt far asyncint(...)
{
    //unsigned char ch;
    Buffer[buffin++] = inportb(0x3f8);// 读字符到缓冲区
    if (buffin >= BUFFLEN)  // 缓冲区满
        buffin=0;           // 指针复位
    outportb(0x20,0x20);
}

void ClosePort(void) //关闭中断
{
    disable();
    outportb(0x3f9,0x00);
    outportb(0x3fc,0x00);
    outportb(0x21,inportb(0x21)&0x10);
    enable();
    setvect(0x0c,asyncoldvect);
}

void InitCOM()// 对COM1串口初始化，设置串口参数
{

    outportb(0x3fb,0x80);  //将设置波特率
    /* 设置波特率，低位在前、高位在后；（部分波特率器参数如下）
   波特率    分频器H    分频器L
     ...
     4800      00       18H
     7200      00       10H
     9600      00       0CH
     ....
 */
    outportb(0x3f8,0x0C);   //波特率为9600bps
    outportb(0x3f9,0x00);

    /*设置停止位、奇偶校验位、等
    D7：为1表示设置波特率；为0，其他；
    D6：为1，是，强迫在数据线上输出逻辑0；若为0，则否；
    D5：1，校验位可变；0，不变；
    D4D3：×0，无校验位；01，奇校验位；11，偶校验位；
    D2：0，1个停止位；1，1.5个停止位；
    D1D0：00，5位数据位；01，6位数据位；10，7位数据位；11，8位数据位；
```

```
        */
        outportb(0x3fb,0x03);   //8 个数据位，1 个停止位，无奇偶校验

        outportb(0x3fc,0x08|0x0b);
        outportb(0x3f9,0x01);
    }


    unsigned char read_char(void)
    {
        unsigned unch;
        if(buffout != buffin)
        {
            unch = Buffer[buffout];
            buffout++;
            if(buffout >= BUFFLEN)
                buffout=0;
            return(unch);
        }
        else
            return(0xff);
    }

    //以下为主函数
    void main()
    {
        unsigned char unChar;
        short bExit_Flag=0;

        OpenPort(); //打开串口

        fprintf(stdout, "\n\nReady to Receive DATA\n"
                "press [ESC] to quit...\n\n");

        do {
            if (kbhit())
            {
                unChar=getch();
                /* Look for an ESC key */
                switch (unChar)
                {
                case 0x1B:  //ESC的ASCII 值为27
                    bExit_Flag = 1;  /* Exit program */
                    break;
                //You may want to handle other keys here
                }
            }
            unChar = read_char();  //从缓冲区中读数
            if (unChar != 0xff)
            {
                fprintf(stdout,"%c",unChar);
            }
        } while (!bExit_Flag);

        ClosePort(); //关闭串口
    }
```

下面对程序进行测试。在 Turbo C++ 3.0 中编译运行程序后，得到了 comrx.exe 可执行文件。连接好串口线后（同一台计算机的两个串口或不同计算机的串口），首先在串口调试助

手中，将串口号设置为 COM2（串口 2）、波特率 9600bps、8 个数据位、1 个停止位、无奇偶校验，并在发送输入框中填入“12345678ABCDEFG”，后加换行（直接按回车键即可），选上“自动发送”，自动发送周期为 1000ms。

如图 1.3.1 所示，打开 MS-DOS 窗口（Windows 98 下，从开始→程序→MS-DOS 方式；Windows 2000 下，从开始→程序→附件→命令提示符），运行 comrx 就可以看到 COM1（串口 1）接收的数据了，也可以看到从串口调试助手发来的数据了。

图 1.3.1 comrx 程序的测试情况

第2章　多线程串口编程工具CSerialPort类

[内容提要]

本章介绍一个非常好用的多线程串口编程工具CSerialPort类，用它可以很轻松地完成一般串口编程任务，几分钟就可搭好串口通信框架。编程者可以从烦心的框架编写中解脱出来，只需将精力放在通信协议的编制及数据处理上。初次接触VC串口编程的读者在用这个类做过程序后，一定会喜欢上它的，等熟悉底层Windows API编程后，我们还可以对这个类进行必要的改造，使之更加强健，帮助我们完成特定的任务。

和MSComm控件相比，这个类打包时，不需要加入其他的文件，而且函数都是开放透明的，允许我们进行改造，还有，它不需要我们去理解有些对于初学者较难掌握的数据类型，所以对初学者更适用。在本书里的例程里，也大量应用了这个类。

本章第2.1节里，对CSerialPort类的功能及成员函数进行了介绍，如果读者刚开始只想完成编程任务，则可以不看这一节，等到想对这个类进行改造或者以后学习用API函数来编写串口程序时，再来仔细看看。

在第2.2节和第2.3节中，分别用CSerialPort类做了基于对话框的例程和基于单文档的例程，一步一步地详细介绍了编程过程，相信即使对VC编程环境不太熟悉的读者，也会很快掌握这些内容。这里仍然可以通过串口调试助手来测试本章程序。

第2.4节是对CSerialPort类的改进。在实际应用中，发现了一些CSerialPort类中一些不完善的地方，笔者根据自己以及其他应用者的一些补充，对类的代码进行了改进。

2.1　CSerialPort类的功能及成员函数介绍

CSerialPort类是由Remon Spekreijse提供的免费串口类，作者还为这个类制作了一个例程，原类的地址为http://www.codeguru.com/network/serialport.shtml，Codeguru是一个非常不错的源代码网站，编程的朋友们应该常去看看。

CSerialPort类支持线连接（非MODEM）的串口编程操作，编写的程序在Windows 98/NT/2000/XP操作系统下可很好地运行，但在Windows Me操作系统下会出现死机的现象，作者没有仔细探究过这个问题，这也许是Windows Me对CSerialPort类中用到的API函数不兼容所致(想当然了)。

CSerialPort类是基于多线程的，其工作流程如下：首先设置好串口参数，再开启串口监测工作线程，串口监测工作线程监测到串口接收到的数据、流控制事件或其他串口事件后，就以消息方式通知主程序，激发消息处理函数来进行数据处理，这是对接收数据而言的；发送数据可直接向串口发送。

CSerialPort类定义的消息如表2-1-1所示。

表 2-1-1　类消息说明

消息名称	消息号	功能说明
WM_COMM_BREAK_DETECTED	WM_USER+1	检测到输入中断
WM_COMM_CTS_DETECTED	WM_USER+2	检测到 CTS (清除发送)信号状态改变
WM_COMM_DSR_DETECTED	WM_USER+3	检测到 DSR (数据设备准备就绪)信号状态改变
WM_COMM_ERR_DETECTED	WM_USER+4	发生线状态错误（包括 CE_FRAME, CE_OVERRUN, 和 CE_RXPARITY）
WM_COMM_RING_DETECTED	WM_USER+5	检测到响铃指示信号
WM_COMM_RLSD_DETECTED	WM_USER+6	检测到 RLSD (接收线信号)状态改变
WM_COMM_RXCHAR	WM_USER+7	接收到一个字符并已放入接收缓冲区
WM_COMM_RXFLAG_DETECTED	WM_USER+8	检测到接收到字符（该字符已放入接收缓冲区）事件
WM_COMM_TXEMPTY_DETECTED	WM_USER+9	检测到发送缓冲区最后一个字符已经被发送

这里先重点介绍几个经常要用到的函数：

1．串口初始化函数 InitPort

这个函数是用来初始化串口的，即设置串口的通信参数：需要打开的串口号、波特率、奇偶校验方式、数据位、停止位，这里还可以用来进行事件的设定。

```
BOOL CSerialPort::InitPort(CWnd* pPortOwner,    // 所属主窗口 the owner (CWnd) of the port
                                                //(receives message)
          UINT  portnr,                         // 串口号 portnumber
          UINT  baud,                           // 波特率 baudrate
          char  parity,                         // 校验方式 parity
          UINT  databits,                       // 数据位 databits
          UINT  stopbits,                       // 停止位 stopbits
          DWORD dwCommEvents,                   // EV_RXCHAR、EV_CTS 事件设置
    UINT  writebuffersize)                  // 设置发送缓冲区大小 size to the writebuffer
```

如果串口初始化成功，就返回.TRUE，若串口被其他设备占用、不存在或存在其他故障，就返回 FALSE，编程者可以在这儿提示串口操作是否成功。

如果在当前主窗口中调用这个函数，那么 pPortOwner 可用 this 指针表示。串口号在函数中做了限制，只能用 1，2，3 和 4 四个串口号，而事实上在编程时可能用到更多串口号，可以通过注释掉本函数中的“assert(portnr > 0 && portnr < 5);”语句取消对串口号的限制。

2．启动串口通信监测线程函数 StartMonitoring()

串口初始化成功后，就可以调用 BOOL StartMonitoring()来启动串口监测线程。线程启动成功，返回 TRUE。

```
// start comm watching
BOOL CSerialPort::StartMonitoring()
{
    if (!(m_Thread = AfxBeginThread(CommThread, this)))
        return FALSE;
    TRACE("Thread started\n");
    return TRUE;
}
```

调用 InitPort()和 StartMonitoring()后，串口就被打开，各种串口状态和事件就可以被监测到。

3. 暂停或停止监测线程函数 StopMonitoring()

该函数暂停或停止串口监测，要注意的是，调用该函数后，串口资源仍然被占用。

```
// Suspend the comm thread
BOOL CSerialPort::StopMonitoring()
{
    TRACE("Thread suspended\n");
    m_Thread->SuspendThread();
    return TRUE;
}
```

4. 关闭串口函数 ClosePort()

该函数功能是关闭串口，释放串口资源。调用该函数后，如果要继续使用串口，还需再调用 InitPort()函数。

```
//Close serial port
void CSerialPort::ClosePort()
{
    SetEvent(m_hShutdownEvent);
}
```

5. 通过串口发送字符/写串口函数 WriteToPort()

该函数完成写串口功能，即向串口发送字符。

```
void CSerialPort::WriteToPort(char* string)
```

以上是常用的函数介绍，熟悉该类的使用后，可以仔细去看看其他的函数。对上面介绍的函数，在对串口资源的使用上要记住以下三点：

- 打开串口用调用 InitPort()和 StartMonitoring()；关闭串口用 StopMonitoring()和 ClosePort()。而且以上函数的调用顺序不能乱。
- 通过串口发送字符调用函数 WriteToPort（）。
- 接收串口收到的字符需要自己编写 WM_COMM_RXCHAR 消息处理函数，需要手工添加。消息处理类似于 DOS 环境下的中断处理。具体过程在下一节的例程中将做详细说明。其他串口事件，如 CTS、DSR、RLSD 等的处理也一样。

下一节将通过具体实例来体验一下 CSerialPort 类的编程，完成后，相信读者会体会到：原来编写串口程序要我做的事真得很少。

为了方便查阅 CSerialPort 类代码（在本书以后的章节中将多次用到），将源代码列在下面：

CSerialPort.h 文件：

```
/*
**  FILENAME            CSerialPort.h
**  AUTHOR              Remon Spekreijse
*/

#ifndef __SERIALPORT_H__
```

```
#define __SERIALPORT_H__

#define WM_COMM_BREAK_DETECTED   WM_USER+1 // A break was detected on input.
#define WM_COMM_CTS_DETECTED  WM_USER+2    // The CTS (clear-to-send) signal
                                           //changed state.
#define WM_COMM_DSR_DETECTE  WM_USER+3     //The DSR (data-set-ready) signal
                                           //changed state.
#define WM_COMM_ERR_DETECTED   WM_USER+4   // A line-status error occurred. Line-status
                                 // errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.
#define WM_COMM_RING_DETECTED    WM_USER+5 // A ring indicator was detected.
#define WM_COMM_RLSD_DETECTED  WM_USER+6   //The RLSD (receive- line-signal-
                                           //detect signal changed state.
#define WM_COMM_RXCHAR  WM_USER+7          // A character was received and placed in the
                                           //input buffer.
#define WM_COMM_RXFLAG_DETECTED  WM_USER+8 // The event character was
                                           //received and placed in the input buffer.
#define WM_COMM_TXEMPTY_DETECTED WM_USER+9 // The last character in the
                                           //output buffer was sent.

class CSerialPort
{
public:
    // contruction and destruction
    CSerialPort();
    virtual       ~CSerialPort();

    // port initialisation
    BOOL      InitPort(CWnd* pPortOwner, UINT portnr = 1, UINT baud = 19200, char parity
= 'N', UINT databits = 8, UINT stopsbits = 1, DWORD dwCommEvents = EV_RXCHAR | EV_CTS, UINT
nBufferSize = 512);

    // start/stop comm watching
    BOOL      StartMonitoring();
    BOOL      RestartMonitoring();
    BOOL      StopMonitoring();

    DWORD          GetWriteBufferSize();
    DWORD          GetCommEvents();
    DCB            GetDCB();

    void      WriteToPort(char* string);

protected:
    // protected memberfunctions
    void      ProcessErrorMessage(char* ErrorText);
    static UINT   CommThread(LPVOID pParam);
    static void   ReceiveChar(CSerialPort* port, COMSTAT comstat);
    static void   WriteChar(CSerialPort* port);

    // thread
    CWinThread*             m_Thread;

    // synchronisation objects
    CRITICAL_SECTION   m_csCommunicationSync;
    BOOL               m_bThreadAlive;

    // handles
    HANDLE                  m_hShutdownEvent;
    HANDLE                  m_hComm;
    HANDLE                  m_hWriteEvent;
```

```
        // Event array.
        // One element is used for each event. There are two event handles for each port.
        // A Write event and a receive character event which is located in the overlapped
structure (m_ov.hEvent).
        // There is a general shutdown when the port is closed.
        HANDLE                  m_hEventArray[3];

        // structures
        OVERLAPPED              m_ov;
        COMMTIMEOUTS        m_CommTimeouts;
        DCB                     m_dcb;

        // owner window
        CWnd*                   m_pOwner;

        // misc
        UINT                m_nPortNr;
        char*                   m_szWriteBuffer;
        DWORD                   m_dwCommEvents;
        DWORD                   m_nWriteBufferSize;
    };

    #endif __SERIALPORT_H__
```

CSerialPort.cpp 文件：

```
    /*
    **  FILENAME            CSerialPort.cpp
    */
    #include "stdafx.h"
    #include "SerialPort.h"
    #include <assert.h>

    // Constructor
    CSerialPort::CSerialPort()
    {
        m_hComm = NULL;
        // initialize overlapped structure members to zero
        m_ov.Offset = 0;
        m_ov.OffsetHigh = 0;
        // create events
        m_ov.hEvent = NULL;
        m_hWriteEvent = NULL;
        m_hShutdownEvent = NULL;

        m_szWriteBuffer = NULL;

        m_bThreadAlive = FALSE;
    }

    // Delete dynamic memory
    CSerialPort::~CSerialPort()
    {
        do
        {
            SetEvent(m_hShutdownEvent);
        } while (m_bThreadAlive);

        TRACE("Thread ended\n");
```

```
        delete [] m_szWriteBuffer;
    }


    // Initialize the port. This can be port 1 to 4.
    BOOL CSerialPort::InitPort(CWnd* pPortOwner,//the owner (CWnd) of the port (receives
message)
                                UINT portnr,        // portnumber (1..4)
                                UINT baud,          // baudrate
                                char parity,        // parity
                                UINT databits,      // databits
                                UINT stopbits,      // stopbits
                                DWORD dwCommEvents, // EV_RXCHAR, EV_CTS etc
                                UINT writebuffersize)// size to the writebuffer
    {
        assert(portnr > 0 && portnr < 5);
        assert(pPortOwner != NULL);
        // if the thread is alive: Kill
        if (m_bThreadAlive)
        {
            do
            {
                SetEvent(m_hShutdownEvent);
            } while (m_bThreadAlive);
            TRACE("Thread ended\n");
        }
        // create events
        if (m_ov.hEvent != NULL)
            ResetEvent(m_ov.hEvent);
        m_ov.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

        if (m_hWriteEvent != NULL)
            ResetEvent(m_hWriteEvent);
        m_hWriteEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

        if (m_hShutdownEvent != NULL)
            ResetEvent(m_hShutdownEvent);
        m_hShutdownEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

        // initialize the event objects
        m_hEventArray[0] = m_hShutdownEvent;  // highest priority
        m_hEventArray[1] = m_ov.hEvent;
        m_hEventArray[2] = m_hWriteEvent;

        // initialize critical section
        InitializeCriticalSection(&m_csCommunicationSync);

        // set buffersize for writing and save the owner
        m_pOwner = pPortOwner;

        if (m_szWriteBuffer != NULL)
            delete [] m_szWriteBuffer;
        m_szWriteBuffer = new char[writebuffersize];

        m_nPortNr = portnr;

        m_nWriteBufferSize = writebuffersize;
        m_dwCommEvents = dwCommEvents;

        BOOL bResult = FALSE;
```

```
        char *szPort = new char[50];
        char *szBaud = new char[50];

        // now in critical.
        EnterCriticalSection(&m_csCommunicationSync);

        // if the port is already opened: close it
        if (m_hComm != NULL)
        {
            CloseHandle(m_hComm);
            m_hComm = NULL;
        }

        // prepare port strings
        sprintf(szPort, "COM%d", portnr);
        sprintf(szBaud, "baud=%d parity=%c data=%d stop=%d", baud, parity, databits,
stopbits);

        // get a handle to the port
        m_hComm = CreateFile(szPort, // communication port string (COMX)
                    GENERIC_READ | GENERIC_WRITE,    // read/write types
                    0,  // comm devices must be opened with exclusive access
                    NULL,               // no security attributes
                    OPEN_EXISTING,// comm devices must use OPEN_EXISTING
                    FILE_FLAG_OVERLAPPED,  // Async I/O
                    0);            // template must be 0 for comm devices

        if (m_hComm == INVALID_HANDLE_VALUE)
        {
            // port not found
            delete [] szPort;
            delete [] szBaud;

            return FALSE;
        }

        // set the timeout values
        m_CommTimeouts.ReadIntervalTimeout = 1000;
        m_CommTimeouts.ReadTotalTimeoutMultiplier = 1000;
        m_CommTimeouts.ReadTotalTimeoutConstant = 1000;
        m_CommTimeouts.WriteTotalTimeoutMultiplier = 1000;
        m_CommTimeouts.WriteTotalTimeoutConstant = 1000;

        // configure
        if (SetCommTimeouts(m_hComm, &m_CommTimeouts))
        {
            if (SetCommMask(m_hComm, dwCommEvents))
            {
                if (GetCommState(m_hComm, &m_dcb))
                {
                    m_dcb.fRtsControl = RTS_CONTROL_ENABLE;         // set RTS bit high!
                    if (BuildCommDCB(szBaud, &m_dcb))
                    {
                        if (SetCommState(m_hComm, &m_dcb))
                            ; // normal operation... continue
                        else
                            ProcessErrorMessage("SetCommState()");
                    }
                    else
                        ProcessErrorMessage("BuildCommDCB()");
```

```
                }
                else
                    ProcessErrorMessage("GetCommState()");
            }
            else
                ProcessErrorMessage("SetCommMask()");
        }
        else
            ProcessErrorMessage("SetCommTimeouts()");

        delete [] szPort;
        delete [] szBaud;

        // flush the port
        PurgeComm(m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT  PURGE_TXABORT);

        // release critical section
        LeaveCriticalSection(&m_csCommunicationSync);

        TRACE("Initialisation for communicationport %d completed.\nUse Startmonitor to
communicate.\n", portnr);

        return TRUE;
    }

    // The CommThread Function.
    UINT CSerialPort::CommThread(LPVOID pParam)
    {
        // Cast the void pointer passed to the thread back to
        // a pointer of CSerialPort class
        CSerialPort *port = (CSerialPort*)pParam;

        // Set the status variable in the dialog class to
        // TRUE to indicate the thread is running.
        port->m_bThreadAlive = TRUE;

        // Misc. variables
        DWORD BytesTransfered = 0;
        DWORD Event = 0;
        DWORD CommEvent = 0;
        DWORD dwError = 0;
        COMSTAT comstat;
        BOOL bResult = TRUE;

        // Clear comm buffers at startup
        if (port->m_hComm)      // check if the port is opened
            PurgeComm(port->m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT |
PURGE_TXABORT);

        // begin forever loop. This loop will run as long as the thread is alive.
        for (;;)
        {
        // Make a call to WaitCommEvent(). This call will return immediatly
        // because our port was created as an async port (FILE_FLAG_OVERLAPPED
        // and an m_OverlappedStructerlapped structure specified). This call will cause the
    // m_OverlappedStructerlapped element m_OverlappedStruct.hEvent, which is part of the
    //m_hEventArray to
            // be placed in a non-signeled state if there are no bytes available to be read,
            // or to a signeled state if there are bytes available. If this event handle
            // is set to the non-signeled state, it will be set to signeled when a
```

```
            // character arrives at the port.
            // we do this for each port!

            bResult = WaitCommEvent(port->m_hComm, &Event, &port->m_ov);

            if (!bResult)
            {
                // If WaitCommEvent() returns FALSE, process the last error to determin
                // the reason..
                switch (dwError = GetLastError())
                {
                case ERROR_IO_PENDING:
                    {
                        // This is a normal return value if there are no bytes
                        // to read at the port.
                        // Do nothing and continue
                        break;
                    }
                case 87:
                    {
                        // Under Windows NT, this value is returned for some reason.
                        // I have not investigated why, but it is also a valid reply
                        // Also do nothing and continue.
                        break;
                    }
                default:
                    {
                        // All other error codes indicate a serious error has
                        // occured. Process this error.
                        port->ProcessErrorMessage("WaitCommEvent()");
                        break;
                    }
                }
            }
            else
            {
            // If WaitCommEvent() returns TRUE, check to be sure there are
            // actually bytes in the buffer to read.
            //
            // If you are reading more than one byte at a time from the buffer
            // (which this program does not do) you will have the situation occur
            // where the first byte to arrive will cause the WaitForMultipleObjects()
            // function to stop waiting. The WaitForMultipleObjects() function
            // resets the event handle in m_OverlappedStruct.hEvent to the non-signelead
            // state as it returns.
            //
        // If in the time between the reset of this event and the call to
        // ReadFile() more bytes arrive, the m_OverlappedStruct.hEvent handle will be set
        // again to the signeled state. When the call to ReadFile() occurs, it will
        // read all of the bytes from the buffer, and the program will
        // loop back around to WaitCommEvent().
        //
    // At this point you will be in the situation where m_OverlappedStruct.hEvent is set,
        // but there are no bytes available to read. If you proceed and call
        // ReadFile(), it will return immediatly due to the async port setup, but
        // GetOverlappedResults() will not return until the next character arrives.
        //
        // It is not desirable for the GetOverlappedResults() function to be in
        // this state. The thread shutdown event (event 0) and the WriteFile()
        // event (Event2) will not work if the thread is blocked by GetOverlappedResults().
```

```
//
// The solution to this is to check the buffer with a call to ClearCommError().
// This call will reset the event handle, and if there are no bytes to read
// we can loop back through WaitCommEvent() again, then proceed.
// If there are really bytes to read, do nothing and proceed.

        bResult = ClearCommError(port->m_hComm, &dwError, &comstat);

        if (comstat.cbInQue == 0)
            continue;
    }   // end if bResult

    // Main wait function.  This function will normally block the thread
    // until one of nine events occur that require action.
    Event = WaitForMultipleObjects(3, port->m_hEventArray, FALSE, INFINITE);

    switch (Event)
    {
    case 0:
        {
            // Shutdown event.  This is event zero so it will be
            // the higest priority and be serviced first.

            port->m_bThreadAlive = FALSE;

            // Kill this thread.  break is not needed, but makes me feel better.
            AfxEndThread(100);
            break;
        }
    case 1:   // read event
        {
            GetCommMask(port->m_hComm, &CommEvent);
            if (CommEvent & EV_CTS)
                ::SendMessage(port->m_pOwner->m_hWnd,
WM_COMM_CTS_DETECTED, (WPARAM) 0, (LPARAM) port->m_nPortNr);
            if (CommEvent & EV_RXFLAG)
                ::SendMessage(port->m_pOwner->m_hWnd,
WM_COMM_RXFLAG_DETECTED, (WPARAM) 0, (LPARAM) port->m_nPortNr);
            if (CommEvent & EV_BREAK)
                ::SendMessage(port->m_pOwner->m_hWnd,
WM_COMM_BREAK_DETECTED, (WPARAM) 0, (LPARAM) port->m_nPortNr);
            if (CommEvent & EV_ERR)
                ::SendMessage(port->m_pOwner->m_hWnd,
WM_COMM_ERR_DETECTED, (WPARAM) 0, (LPARAM) port->m_nPortNr);
            if (CommEvent & EV_RING)
                ::SendMessage(port->m_pOwner->m_hWnd,
WM_COMM_RING_DETECTED, (WPARAM) 0, (LPARAM) port->m_nPortNr);

            if (CommEvent & EV_RXCHAR)
                // Receive character event from port.
                ReceiveChar(port, comstat);

            break;
        }
    case 2: // write event
        {
            // Write character event from port
            WriteChar(port);
            break;
```

```
            } // end switch

        } // close forever loop

        return 0;
    }

    // start comm watching
    BOOL CSerialPort::StartMonitoring()
    {
        if (!(m_Thread = AfxBeginThread(CommThread, this)))
            return FALSE;
        TRACE("Thread started\n");
        return TRUE;
    }

    // Restart the comm thread
    BOOL CSerialPort::RestartMonitoring()
    {
        TRACE("Thread resumed\n");
        m_Thread->ResumeThread();
        return TRUE;
    }

    // Suspend the comm thread
    BOOL CSerialPort::StopMonitoring()
    {
        TRACE("Thread suspended\n");
        m_Thread->SuspendThread();
        return TRUE;
    }

    // If there is a error, give the right message
    void CSerialPort::ProcessErrorMessage(char* ErrorText)
    {
        char *Temp = new char[200];

        LPVOID lpMsgBuf;

        FormatMessage(
            FORMAT_MESSAGE_ALLOCATE_BUFFER
    FORMAT_MESSAGE_FROM_SYSTEM,
            NULL,
            GetLastError(),
            MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
            (LPTSTR) &lpMsgBuf,
            0,
            NULL
        );

        sprintf(Temp, "WARNING: %s Failed with the following error: \n%s\nPort: %d\n",
(char*)ErrorText, lpMsgBuf, m_nPortNr);
        MessageBox(NULL, Temp, "Application Error", MB_ICONSTOP);

        LocalFree(lpMsgBuf);
        delete[] Temp;
    }

    // Write a character.
```

```
void CSerialPort::WriteChar(CSerialPort* port)
{
    BOOL bWrite = TRUE;
    BOOL bResult = TRUE;

    DWORD BytesSent = 0;

    ResetEvent(port->m_hWriteEvent);

    // Gain ownership of the critical section
    EnterCriticalSection(&port->m_csCommunicationSync);

    if (bWrite)
    {
        // Initailize variables
        port->m_ov.Offset = 0;
        port->m_ov.OffsetHigh = 0;

        // Clear buffer
        PurgeComm(port->m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT |
PURGE_TXABORT);

        bResult = WriteFile(port->m_hComm,              // Handle to COMM Port
            port->m_szWriteBuffer,  // Pointer to message buffer in calling finction
            strlen((char*)port->m_szWriteBuffer), // Length of message to send
            &BytesSent,    // Where to store the number of bytes sent
            &port->m_ov);       // Overlapped structure

        // deal with any error codes
        if (!bResult)
        {
            DWORD dwError = GetLastError();
            switch (dwError)
            {
                case ERROR_IO_PENDING:
                    {
                        // continue to GetOverlappedResults()
                        BytesSent = 0;
                        bWrite = FALSE;
                        break;
                    }
                default:
                    {
                        // all other error codes
                        port->ProcessErrorMessage("WriteFile()");
                    }
            }
        }
        else
        {
            LeaveCriticalSection(&port->m_csCommunicationSync);
        }
    } // end if(bWrite)

    if (.bWrite)
    {
        bWrite = TRUE;

        bResult = GetOverlappedResult(port->m_hComm,    // Handle to COMM port
                                      &port->m_ov,      // Overlapped structure
```

```
                                                    &BytesSent,        // Stores number of bytes
sent
                                                    TRUE);             // Wait flag

                LeaveCriticalSection(&port->m_csCommunicationSync);

                // deal with the error code
                if (!bResult)
                {
                    port->ProcessErrorMessage("GetOverlappedResults() in WriteFile()");
                }
            } // end if (!bWrite)

            // Verify that the data size send equals what we tried to send
            if (BytesSent != strlen((char*)port->m_szWriteBuffer))
            {
                TRACE("WARNING: WriteFile() error.. Bytes Sent: %d; Message Length: %d\n",
BytesSent, strlen((char*)port->m_szWriteBuffer));
            }
        }

        // Character received. Inform the owner
        void CSerialPort::ReceiveChar(CSerialPort* port, COMSTAT comstat)
        {
            BOOL  bRead = TRUE;
            BOOL  bResult = TRUE;
            DWORD dwError = 0;
            DWORD BytesRead = 0;
            unsigned char RXBuff;

            for (;;)
            {
                // Gain ownership of the comm port critical section.
                // This process guarantees no other part of this program
                // is using the port object.

                EnterCriticalSection(&port->m_csCommunicationSync);

                // ClearCommError() will update the COMSTAT structure and
                // clear any other errors.

                bResult = ClearCommError(port->m_hComm, &dwError, &comstat);

                LeaveCriticalSection(&port->m_csCommunicationSync);

                // start forever loop.  I use this type of loop because I
                // do not know at runtime how many loops this will have to
                // run. My solution is to start a forever loop and to
                // break out of it when I have processed all of the
                // data available.  Be careful with this approach and
                // be sure your loop will exit.
                // My reasons for this are not as clear in this sample
                // as it is in my production code, but I have found this
                // solution to be the most efficient way to do this.

                if (comstat.cbInQue == 0)
                {
                    // break out when all bytes have been read
                    break;
```

```
            EnterCriticalSection(&port->m_csCommunicationSync);

            if (bRead)
            {
                bResult = ReadFile(port->m_hComm,       // Handle to COMM port
                                   &RXBuff,             // RX Buffer Pointer
                                   1,                   // Read one byte
                                   &BytesRead,          // Stores number of bytes read
                                   &port->m_ov);        // pointer to the m_ov structure
                // deal with the error code
                if (!bResult)
                {
                    switch (dwError = GetLastError())
                    {
                        case ERROR_IO_PENDING:
                            {
                                // asynchronous i/o is still in progress
                                // Proceed on to GetOverlappedResults();
                                bRead = FALSE;
                                break;
                            }
                        default:
                            {
                                // Another error has occured.  Process this error.
                                port->ProcessErrorMessage("ReadFile()");
                                break;
                            }
                    }
                }
                else
                {
        // ReadFile() returned complete. It is not necessary to call GetOverlappedResults()
                    bRead = TRUE;
                }
            } // close if (bRead)

            if (!bRead)
            {
                bRead = TRUE;
                bResult = GetOverlappedResults(port->m_hComm,    // Handle to COMM port
                                               &port->m_ov,//Overlapped structure
                                               &BytesRead,    // Stores number of bytes read
                                               TRUE);             // Wait flag

                // deal with the error code
                if (!bResult)
                {
                    port->ProcessErrorMessage("GetOverlappedResults() in ReadFile()");
                }
            } // close if (!bRead)

            LeaveCriticalSection(&port->m_csCommunicationSync);

            // notify parent that a byte was received
            ::SendMessage((port->m_pOwner)->m_hWnd, WM_COMM_RXCHAR, (WPARAM) RXBuff,
(LPARAM) port->m_nPortNr);
        } // end forever loop
```

```
// Write a string to the port
void CSerialPort::WriteToPort(char* string)
{
    assert(m_hComm != 0);
    memset(m_szWriteBuffer, 0, sizeof(m_szWriteBuffer));
    strcpy(m_szWriteBuffer, string);
    // set event for write
    SetEvent(m_hWriteEvent);
}

// Return the device control block
DCB CSerialPort::GetDCB()
{
    return m_dcb;
}

// Return the communication event masks
DWORD CSerialPort::GetCommEvents()
{
    return m_dwCommEvents;
}

// Return the output buffer size
DWORD CSerialPort::GetWriteBufferSize()
{
    return m_nWriteBufferSize;
}
```

2.2　应用 CSerialPort 类编制基于对话框的应用程序

现在，应用 CSerialPort 类来编写一个基于对话框的测试程序。可以按照以下步骤完成程序的编写。

1. 建立程序框架工程

在 VC 6.0 集成开发环境中，新建基于对话框（Dialog based）的 MFC AppWizard(exe)应用程序，工程名为 SerialPortTest。所有步骤保持默认状态。并在主对话框中添加控件，最后的主对话框状态如图 2.2.1 所示。

图 2.2.1　主对话框添加完控件后的状态

然后用 ClassWizard 为相应控件添加变量，控件的属性设置情况如表 2-2-1 所示。

表 2-2-1　控件及其属性设置情况

控件	控件 ID	Caption	需要添加的变量及变量类型
静态文本	IDC_STATIC	接收发送	
静态文本	IDC_STATIC	发送显示	
静态文本	IDC_STATIC	串口号	
组合框	IDC_COMBO_COMPORT		m_ctrlComboComPort Control
编辑框	IDC_EDIT_RECEIVEMSG		m_strEditReceiveMsg Value CString
编辑框	IDC_EDIT_SENDMSG		m_strEditSendMsg Value CString
按钮	IDC_BUTTON_OPEN	打开串口	
按钮	IDC_BUTTON_CLOSE	关闭串口	
按钮	IDC_BUTTON_SEND	发送	

2. 添加类文件

将类文件 SerialPort.h 和 SerialPort.cpp 复制到工程所在的文件夹中，然后单击 VC 菜单 Project->Add to Project->Files...，再在打开的文件选择对话框中选择上 SerialPort.h 和 SerialPort.cpp，单击 OK，就把类文件加入了当前工程，如图 2.2.2 所示。并在 SerialPortTestDlg.h 中将头文件 SerialPort.h 说明：#include "SerialPort.h"。通过以上步骤，就在当前工程中加入了 CSerailPort 类。

图 2.2.2　在工程中添加类文件

3. 通过 CSerialPort 类完成串口操作

首先，在主对话框头文件 SerialPortTestDlg.h 中定义 CSerialPort 类对象，在本例中只需要操作 1 个串口，所以只定义 1 个类对象就可以了。若要操作多个串口，则要为每个串口均定义一个类对象，这可以通过数组方式来定义。这里定义的类对象为 m_SerialPort，再定义一个布尔变量 m_bSerialPortOpened 用来标志串口是否打开。

在 CSerialPort 类中有多个串口事件可以响应，在一般串口编程中，只需要处理 WM_COMM_RXCHAR 消息就可以了，该类所有的消息均需要人工添加消息处理函数。将处理函数名定义为 OnComm()。首先在 SerialPortTestDlg.h 中添加串口字符接收消息 WM_COMM_RXCHAR（串口接收缓冲区内有一个字符）的响应函数声明：

```
// Generated message map functions
//{{AFX_MSG(CSCPortTestView)
afx_msg LONG OnComm(WPARAM ch, LPARAM port);
//}}AFX_MSG
```

然后，在 SerialPortTestDlg.cpp 文件中进行 WM_COMM_RXCHAR 消息映射：

```
BEGIN_MESSAGE_MAP(CSerialPortTestDlg, CDialog)
    //{{AFX_MSG_MAP(CSerialPortTestDlg)
    ON_MESSAGE(WM_COMM_RXCHAR, OnComm)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

接着，在 SerialPortTestDlg.cpp 文件中加入函数 OnComm()的实现，并在其中完成对接收到的字符的处理，将接收到的字符显示在接收编辑框中：

```
LONG CSerialPortTestDlg::OnComm(WPARAM ch, LPARAM port)
{
    m_strEditReceiveMsg += ch;
    UpdateData(FALSE);  //将接收到的字符显示在接收编辑框中
    return 0;
}
```

说明：WPARAM、LPARAM 类型是多态数据类型（polymorphic data type），在 WIN32 中为 32 位，支持多种数据类型，根据需要自动适应，这样，程序有很强的适应性。在此，我们可以分别理解为 char 和 integer 类型数据。每当串口接收缓冲区内有一个字符时，就会产生一个 WM_COMM_RXCHAR 消息，触发 OnComm()函数，这时就可以在函数中进行数据处理，所以，这个消息就是整个程序的“发动机”。

以上完成了对串口接收字符的处理，但还没有打开串口。选择串口 1（将组合框 IDC_COMBO_COMPORT 的选项置为 0 即可：“m_ctrlComboComPort.SetCurSel(0);”。

为按钮“打开串口”添加单击响应函数（利用 ClassWizard 或在对话框 IDD_SERIALPORTTEST_DIALOG 资源中双击按钮）OnButtonOpen()，为“关闭串口”按钮添加单击响应函数 OnButtonClose()，完成对串口的初始化（打开）和关闭操作，串口的参数用最常用的参数设置：波特率 9600、8 个数据位，1 个停止位、无奇偶校验，在以后的例程中如没有特别说明，我们均用此设置。代码如下：

```
void CSerialPortTestDlg::OnButtonOpen()  //打开串口
{
    // TODO: Add your control notification handler code here
    int nPort=m_ctrlComboComPort.GetCurSel()+1; //得到串口号，想想为什么要加1
    if(m_SerialPort.InitPort(this, nPort, 9600,'N',8,1,EV_RXFLAG | EV_RXCHAR,512))
    {
        m_SerialPort.StartMonitoring();
        m_bSerialPortOpened=TRUE;
    }
    else
    {
        AfxMessageBox("没有发现此串口或被占用");
        m_bSerialPortOpened=FALSE;
    }
    GetDlgItem(IDC_BUTTON_OPEN)->EnableWindow(!m_bSerialPortOpened);
    GetDlgItem(IDC_BUTTON_CLOSE)->EnableWindow(m_bSerialPortOpened);
}

void CSerialPortTestDlg::OnButtonClose()   //关闭串口
{
    // TODO: Add your control notification handler code here
    m_SerialPort.ClosePort();//关闭串口
 m_bSerialPortOpened=FALSE;
    GetDlgItem(IDC_BUTTON_OPEN)->EnableWindow(!m_bSerialPortOpened);
    GetDlgItem(IDC_BUTTON_CLOSE)->EnableWindow(m_bSerialPortOpened);
}
```

至此，完成对串口的初始化和打开关闭的操作，本例程还有一个发送数据的任务没有完成。“发送”按钮添加单击响应函数 OnButtonSend()，并写入发送代码：

```
void CSerialPortTestDlg::OnButtonSend()
{
    // TODO: Add your control notification handler code here
    if(!m_bSerialPortOpened) return; //检查串口是否打开
    UpdateData(TRUE); //读入编辑框中的数据
    m_SerialPort.WriteToPort((LPCTSTR)m_strEditSendMsg);//发送数据
}
```

以上程序编写完后，SerialPortTestDlg.h 文件代码应如下：

```
// SerialPortTestDlg.h : header file
……
#include "SerialPort.h"  //添加CSerialPort类的头文件
…
class CSerialPortTestDlg : public CDialog
{
// Construction
public:
    CSerialPort m_SerialPort; //CSerialPort类对象
    BOOL m_bSerialPortOpened; //标志串口是否打开
    CSerialPortTestDlg(CWnd* pParent = NULL);  // standard constructor
// Dialog Data
    //{{AFX_DATA(CSerialPortTestDlg)
    enum { IDD = IDD_SERIALPORTTEST_DIALOG };
    CComboBox m_ctrlComboComPort;
    CString  m_strEditReceiveMsg;
    CString  m_strEditSendMsg;
    //}}AFX_DATA
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CSerialPortTestDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL
// Implementation
protected:
    HICON m_hIcon;
    // Generated message map functions
    //{{AFX_MSG(CSerialPortTestDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg LONG OnComm(WPARAM ch, LPARAM port);
    afx_msg void OnButtonOpen();
    afx_msg void OnButtonClose();
    afx_msg void OnButtonSend();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
```

现在试着编译运行程序，应该没有错误，可以正常运行程序。可以利用串口调试助手来对程序进行测试，接好串口线后，用串口调试助手控制一个串口，用本测试例程控制另一个串口，再在发送编辑框中随意填上几个字符，单击“发送”按钮。或用串口调试助手发送，本例程接收。图 2.2.3 是程序运行情况。

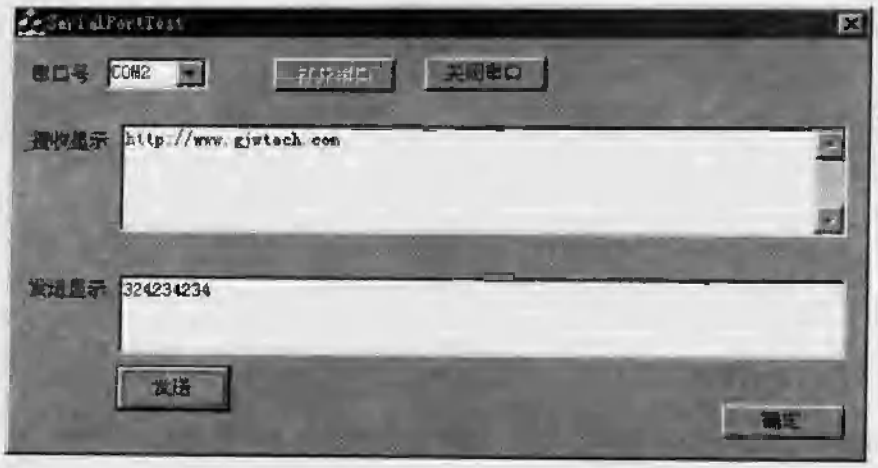


图 2.2.3　测试例程运行情况

2.3　应用 CSerialPort 类编制基于单文档的应用程序

单文档与对话框程序中，应用 CSerialPort 类来操作串口没有多大区别，过程与基于对话框的应用程序基本相同，只是在 VC 编程过程中，具体操作细节上会有所区别。为了方便初学者，我们做了一个例程放在这里，具体代码可以查看本书配带的源代码光盘，这里不一一列出。

在以下建立的例程中，我们完成这样的任务：同时打开两个串口，串口 1（COM1）负责每隔 1 秒钟向 COM2 发送 6 位数据 XXXXXX，为了便于识别，COM1 发送的数据前面加上$，后面加上*，并且每次自动加 1。即：

$XXXXXX*

串口 2（COM2）接收到数据后，将其显示，并向 COM1 发送应答，将$改为 Y，加上接收到的信息，即：

YXXXXXX*

同时，COM1 将发送和接收到的信息显示。运行以上程序，计算机上必须有两个串口，如果只有一个串口，则自己将程序修改一下，只建立一个串口的操作，或者只看看如何操作两个串口就行了。

1．建立程序框架工程并添加类文件

建立基于单文档的工程：SerialPortTest2，将类文件 SerialPort.h 和 SerialPort.cpp 复制到工程所在文件夹中，同第 2.2 节第 2 步一样，在工程中添加类文件，并在 SerialPortTest2View.h 中将头文件 SerialPort.h 说明：#include "SerialPort.h"。为了对串口进行处理，还需要在 SerialPortTest2View.h 定义下列变量：

```
public:
    CSerialPort m_SerialPort[2]; //定义两个 CSerialPort 类对象
    BOOL m_bCOM1Opened; //用于标志 COM1 是否打开
```

```
    BOOL m_bCOM2Opened; //用于标志 COM2 是否打开
    CString m_strRXDataCOM1;   //COM1 接收数据
    CString m_strRXDataCOM2;   //COM2 接收数据
```

在 SerialPortTest2View 的构造函数中，将串口状态设置为没有打开：

```
CSerialPortTest2View::CSerialPortTest2View()
{
    // TODO: add construction code here
    m_bCOM1Opened=FALSE; //COM1 初始状态没有打开
    m_bCOM2Opened=FALSE; //COM2 初始状态没有打开
}
```

2. 完成串口初始化工作

在视图中通过 ClassWizard 添加初始化更新函数 OnInitialUpdate()，如图 2.3.1 所示。

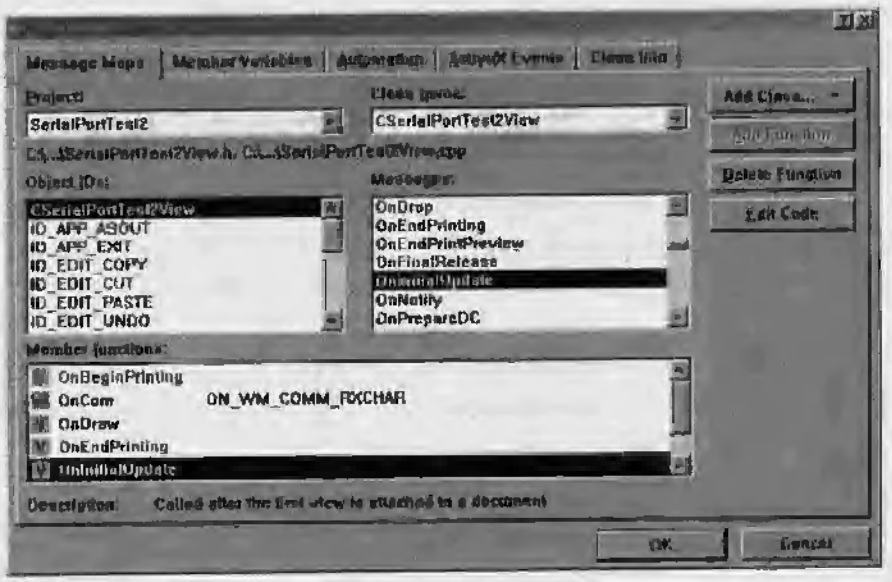


图 2.3.1　添加 OnInitialUpdate()函数

然后在 OnInitialUpdate()函数中完成串口的初始化工作。

```
void CSerialPortTest2View::OnInitialUpdate()
{
    CView::OnInitialUpdate();
    // TODO: Add your specialized code here and/or call the base class
    for(int i=0;i<2;i++)
    {
        if (m_SerialPort[i].InitPort(this,i+1,9600,'N',8,1,EV_RXFLAG |
EV_RXCHAR,512))
        {
            m_ComPort[i].StartMonitoring();   //启动串口监视线程
            if(i==0)          //如果是 COM1 打开成功，则设置定时器
            {
                SetTimer(1,1000,NULL);  //设置定时器，1 秒后发送数据
                m_bCOM1Opened=TRUE;     //COM1 打开
```

```
            }
            else
                m_bCOM2Opened=TRUE;     //COM2 打开
        }
        else
        {
            CString strTemp;
            strTemp.Format("COM%d 没有发现，或被其他设备占用",i+1);
            AfxMessageBox(strTemp);
        }
    }
}
```

3. 添加 WM_TIMER 消息响应在 COM1 定时发送数据

通过 ClassWizard 添加 WM_TIMER 消息响应，如图 2.3.2 所示。

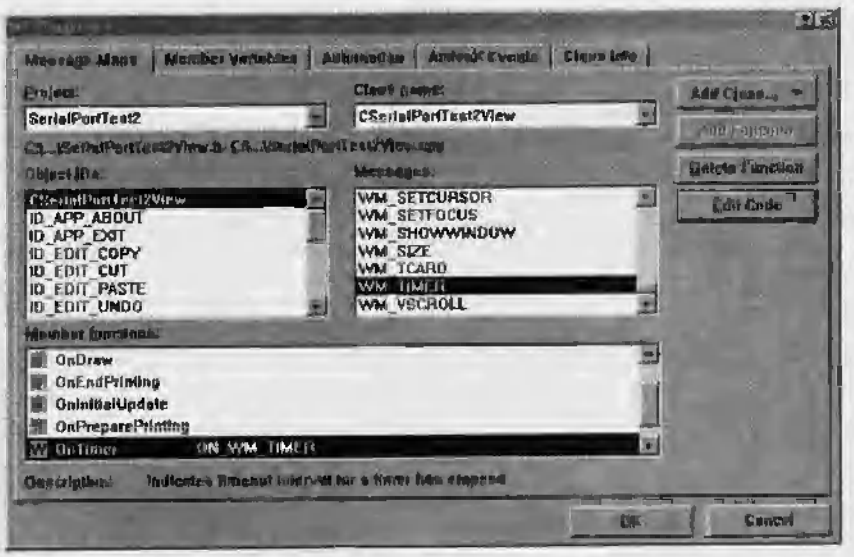


图 2.3.2　添加 WM_TIMER 消息响应

每隔 1 秒钟从 COM1 定时发送数据，为了方便观察，让发送的数据每次自动加 1。代码如下：

```
void CSerialPortTest2View::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    static UINT unCount=0; //定义静态变量
    if(m_bCOM1Opened) //检查 COM1 是否成功打开
    {
        unCount++;
        CString strTemp;
        strTemp.Format("$%06d*",unCount);
        m_SerialPort[0].WriteToPort((LPCTSTR)strTemp); //COM1 发送数据
        CDC* pDC=GetDC();   //准备数据显示
        pDC->TextOut(10,100,"COM1 发送: "+strTemp);//显示接收到的数据
        ReleaseDC(pDC);
    }
```

```
        CView::OnTimer(nIDEvent);
    }
```

4．建立 WM_COMM_RXCHAR 的消息映射处理函数 OnComm()

接着手工建立 WM_COMM_RXCHAR 的消息映射处理函数 OnComm()。首先在 SerialPortTest2View.h 中添加串口字符接收消息 WM_COMM_RXCHAR（串口接收缓冲区内有一个字符）的响应函数声明：

```
//{{AFX_MSG(CSerialPortTest2View)
afx_msg LONG OnComm(WPARAM ch, LPARAM port);
//}}AFX_MSG
```

然后在 SerialPortTest2View.cpp 文件中进行 WM_COMM_RXCHAR 消息映射：

```
BEGIN_MESSAGE_MAP(CSerialPortTest2View, CView)
    //{{AFX_MSG_MAP(CSerialPortTest2View)
    ON_MESSAGE(WM_COMM_RXCHAR, OnCom)
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

再在 SerialPortTest2View.cpp 文件中加入函数 OnComm()的实现，并根据本节编程任务完成函数的编写，代码如下：

```
LONG CSerialPortTest2View::OnComm(WPARAM ch, LPARAM port)
{
    if(port==2)   //COM2 接收到数据
    {
        switch(ch)
        {
        case '$':
            m_strRXDataCOM2=(char)ch;
            break;
        case '*':
            {
            m_strRXDataCOM2+=(char)ch;
            CDC* pDC=GetDC();   //准备数据显示
                                //显示接收到的数据
            pDC->TextOut(10,150,"COM2 接收到"+m_strRXDataCOM2);
                ReleaseDC(pDC);
            m_strRXDataCOM2.Replace('$','Y');
                                //COM2 发送应答信息
            m_SerialPort[1].WriteToPort((LPCTSTR)m_strRXDataCOM2);
            }
            break;
        default:
            m_strRXDataCOM2+=(char)ch;
            break;
        }
    }

    if(port==1)       //COM1 接收到数据
    {
```

```
            switch(ch)
            {
            case 'Y':
                m_strRXDataCOM1=(char)ch;
                break;
            case '*':
                {
                m_strRXDataCOM1+=(char)ch;
                CDC* pDC=GetDC();   //准备数据显示
                //显示接收到的数据
            pDC->TextOut(200,100,"COM1 接收到: "+m_strRXDataCOM1);
            ReleaseDC(pDC);
                }
                break;
            default:
                m_strRXDataCOM1+=(char)ch;
                break;
            }
        }

        return 0;
    }
```

从以上对串口字符的接收处理中，还可以看出，CSerialPort 类在处理多个串口时，在得到接收到的字符时，还能知道接收该字符的串口号。

当以上代码编写完后，SerialPortTest2View.h 中的代码如下：

```
// SerialPortTest2View.h : interface of the CSerialPortTest2View class
… …
#include "SerialPort.h"  //添加 CSerialPort 类头文件
class CSerialPortTest2View : public CView
{
protected: // create from serialization only
    CSerialPortTest2View();
    DECLARE_DYNCREATE(CSerialPortTest2View)
// Attributes
public:
    CSerialPort m_SerialPort[2]; //定义两个 CSerialPort 类对象
    BOOL m_bCOM1Opened; //用于标志 COM1 是否打开
    BOOL m_bCOM2Opened; //用于标志 COM2 是否打开
    CString m_strRXDataCOM1;   //COM1 接收数据
    CString m_strRXDataCOM2;   //COM2 接收数据
    CSerialPortTest2Doc* GetDocument();
// Operations

… …

// Generated message map functions
protected:
    //{{AFX_MSG(CSerialPortTest2View)
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg LONG OnComm(WPARAM ch, LPARAM port);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
… …
```

好了，到这里就完成了本例程的编写，程序运行情况如图 2.3.3 所示。

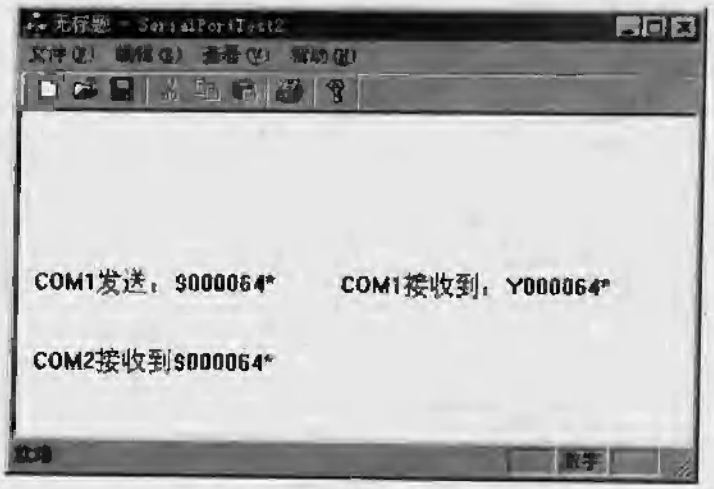


图 2.3.3　程序运行情况

从本章的两个例程中，可以得到这样一个结论：不论在哪种方式的程序中，CSerialPort 类的应用基本相同，即只要在相应的主窗口(CWnd)中建立类对象（指针），就可以对串口进行操作。待到我们熟悉后，用 CSerialPort 类编写一般的串口程序，就用不了几分钟了。

2.4　对 CSerialPort 类的改进

对 CSerialPort 类里面使用的 API 函数说明请参看第 4.5 节，本节对该类进行一些改进，以便它能更好地适应编程。当然，这些改进并不一定是非常合理的，改进后的效果还有待进一步检验。

虽然 CSerialPort 类是一个非常好的类，但毕竟只是集中了作者一个人的智慧和经验，它也有许多缺陷，从目前反馈的情况看，主要有以下一些问题：

- 原类只能发送字符（ASCII 文本），不能处理二进制发送（当能也就不能发送 0X00 了）；
- 该类不能很好地释放串口；
- 存在内存泄漏；

所以，可以进行如下改进。

- 改进一：ASCII 文本和二进制数据发送方式兼容
- 改进二：也许能解决内存泄漏
- 改进三：彻底关闭串口，释放串口资源

2.4.1　改进一：ASCII 文本和二进制数据发送方式兼容

CSerialPort 类中只有一个发送函数 WriteToPort()：

```
void CSerialPort::WriteToPort(char* string)
{
    assert(m_hComm != 0);
    memset(m_szWriteBuffer, 0, sizeof(m_szWriteBuffer));
```

```
        strcpy(m_szWriteBuffer, string);
        SetEvent(m_hWriteEvent); // set event for write
    }
```

调用上面的函数就只能用字符串方式，而 C 语言中，空字符（NULL，其 ASCII 码值为 0，即通常所说的十六进制 0x00 字符）是串结束符，当检测到 NULL 字符后，就认为该字符串结束了。所以 0x00 字符以 ASCII 文本方式是不能从串口发送出去的。那么解决这一问题的方法就是用二进制发送，其实这里说的二进制，只不过是不以我们通常所说的“可见”或“能显示的字符”发送。比如，要发送如下的一组值：

```
char chSend[5 ]={0x33, 0x96, 0x00, 0x31, 0xF1};
```

如果调用 WriteToPort()函数来发送，则语句为：

```
WriteToPort(chSend);
```

这里可以做一个测试程序，现在应该不到 3 分钟，你就能做好这个程序了。作者做了一个放在第 2 章中，工程名为 NNTest，上面就放了一个按钮，Button1，添加 Button1 的单击响应函数：

```
void CNNTestDlg::OnButton1()
{
    // TODO: Add your control notification handler code here
    char chSend[ 5]={0x33, 0x96, 0x00, 0x31, 0xF1};
    m_SerialPort.WriteToPort(chSend);
}
```

用串口线连接两个串口，用这个测试程序 NNTest 控制一个串口，另一串口用串口调试助手接收数据，并选上“十六进制显示”，在测试程序 NNTest 中单击 Button1 按钮，看到在串口调试助手中只显示了 0x33, 0x96 两个数据（如图 2.4.1 所示），而 0x00, 0x31, 0xF1 就没有被发送出去。

图 2.4.1　测试 NNTest 发送含 NULL 的字符串数据

下面来对类做一些改进，解决这个问题。原理就是用字符数据来发送数据，并在发送时指定其长度。这样，数据没有发送完，发送过程就不会停止。CSerialPort 类是用 API 函数编

写的，只要在 WriteFile()函数中指定其实际要发送的长度，就可以将数据全部发送出去，有关 API 函数的使用方法，请参看第 4.3 节。

实现步骤如下。

（1）在 SerialPort.h 文件中为 CSerialPort 类添加一个整型 publicdn 成员变量：int m_nWriteSize; 用于指定发送字符数组的长度。

（2）为 CSerialPort 类添加 3 个公有成员函数：

```
void      WriteToPort(char* string,int n);
void      WriteToPort(LPCTSTR string);
void      WriteToPort(LPCTSTR string,int n);
```

其实现代码如下：

```
void CSerialPort::WriteToPort(char* string,int n)
{
    assert(m_hComm != 0);
    memset(m_szWriteBuffer, 0, sizeof(m_szWriteBuffer));
    memcpy(m_szWriteBuffer, string, n);
    m_nWriteSize=n;
    // set event for write
    SetEvent(m_hWriteEvent);
}

void CSerialPort::WriteToPort(LPCTSTR string)
{
    assert(m_hComm != 0);
    memset(m_szWriteBuffer, 0, sizeof(m_szWriteBuffer));
    strcpy(m_szWriteBuffer, string);
    m_nWriteSize=strlen(string);
    // set event for write
    SetEvent(m_hWriteEvent);
}

void CSerialPort::WriteToPort(LPCTSTR string,int n)
{
    assert(m_hComm != 0);
    memset(m_szWriteBuffer, 0, sizeof(m_szWriteBuffer));
    memcpy(m_szWriteBuffer, string, n);
    m_nWriteSize=n;
    // set event for write
    SetEvent(m_hWriteEvent);
}
```

原有的 void CSerialPort::WriteToPort(char* string)也应进行改写。

原函数为：

```
void CSerialPort::WriteToPort(char* string)
{
    assert(m_hComm != 0);
    memset(m_szWriteBuffer, 0, sizeof(m_szWriteBuffer));
    strcpy(m_szWriteBuffer, string);
    // set event for write
    SetEvent(m_hWriteEvent);
}
```

改写为：

```
void CSerialPort::WriteToPort(char* string)
{
    assert(m_hComm != 0);
        memset(m_szWriteBuffer, 0, sizeof(m_szWriteBuffer));
    strcpy(m_szWriteBuffer, string);
    m_nWriteSize=strlen(string);
    // set event for write
    SetEvent(m_hWriteEvent);
}
```

（3）改写 WriteFile（）函数用于指定发送长度的变量，这个 API 函数位于 void CSerialPort::WriteChar(CSerialPort* port)中：

```
void CSerialPort::WriteChar(CSerialPort* port)
{
……

        bResult = WriteFile(port->m_hComm,          // Handle to COMM Port
            port->m_szWriteBuffer,  // Pointer to message buffer in calling finction
//  原语句          strlen((char*)port->m_szWriteBuffer), // Length of message to send
            port->m_nWriteSize,     // 更改后: Length of message to send
            &BytesSent,         // Where to store the number of bytes sent
           &port->m_ov);        // Overlapped structure

    ……

    //Verify that the data size send equals what we tried to send
//原语句if (BytesSent != strlen((char*)port->m_szWriteBuffer))
    if (BytesSent != port->m_nWriteSize)  // 修改后Length of message to send)
    {
        TRACE("WARNING: WriteFile() error.. Bytes Sent: %d; Message Length: %d\n",
BytesSent, port->m_nWriteSize);
    }
}
```

这样，加上了这三个函数，一共有四个发送函数，就可以适应不同的数据发送要求了。当然，以上的修改中还可以将 char 改为 unsigned char，但不改也是可以的。

改完后，读者可以自己测试一下处理结果。

2.4.2　改进二：也许能解决内存泄漏

在该类的 Init()函数中，有这样一段代码：

```
BOOL CSerialPort::InitPort(CWnd* pPortOwner,   // the owner (CWnd) of the port
                           //(receives message)
            UINT portnr,   // portnumber (1..4)
            UINT baud,     // baudrate
            char parity,   // parity
            UINT databits, // databits
            UINT stopbits, // stopbits
        DWORD dwCommEvents, // EV_RXCHAR, EV_CTS etc
         UINT writebuffersize) // size to the writebuffer
{
……
                           // create events 创建事件
    if (m_ov.hEvent != NULL)
```

```
        ResetEvent(m_ov.hEvent);
    m_ov.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    if (m_hWriteEvent != NULL)
        ResetEvent(m_hWriteEvent);
    m_hWriteEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    if (m_hShutdownEvent != NULL)
        ResetEvent(m_hShutdownEvent);
    m_hShutdownEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
...
}
```

这是用于创建事件，但注意到，如果多次调用 Init()函数来初始化串口（在多串口程序或者改变串口配置参数后都会多次调用），就会引起事件句柄的内存资源泄漏（重复分配）。因为 **ResetEvent()**函数只是将已经存在的事件设置为无信号状态，而该事件本身已经存在，应用上面的语句，就有可能创建多个句柄，这是没有必要的。

ResetEvent()函数声明如下：

函数功能：将指定事件对象设置为无信号（非触发）状态状态。

BOOL ResetEvent(

　HANDLE *hEvent*　　//事件对象句柄 handle to event object

);

参数说明：

hEvent ：由CreateEvent 或 OpenEvent函数返回的事件对象句柄。

可以用下面的修改来解决这个问题：

```
// create events
if (m_ov.hEvent != NULL)
    ResetEvent(m_ov.hEvent);
else
    m_ov.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

if (m_hWriteEvent != NULL)
    ResetEvent(m_hWriteEvent);
else
    m_hWriteEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

if (m_hShutdownEvent != NULL)
    ResetEvent(m_hShutdownEvent);
else
    m_hShutdownEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
```

2.4.3　改进三：彻底关闭串口，释放串口资源

该类没有彻底释放串口资源，在类的释构函数中可添加以下代码：

```
// Delete dynamic memory
//
CSerialPort::~CSerialPort()
```

```
        do
        {
            SetEvent(m_hShutdownEvent);
        } while (m_bThreadAlive);


        // if the port is still opened: close it
        if (m_hComm != NULL)
        {
            CloseHandle(m_hComm);
            m_hComm = NULL;
        }
        // Close Handles
        if(m_hShutdownEvent!=NULL)
            CloseHandle( m_hShutdownEvent);
        if(m_ov.hEvent!=NULL)
            CloseHandle( m_ov.hEvent );
        if(m_hWriteEvent!=NULL)
            CloseHandle( m_hWriteEvent );

        TRACE("Thread ended\n");
        delete [] m_szWriteBuffer;
    }
```

虽然上面的修改在程序关闭时，在控制单个串口的程序中不会有问题了。但应用这个类进行多串口控制时（同时打开多个串口），关闭串口再打开时，程序就会异常。有兴趣的读者可以和作者探讨这个问题。这里不再讨论（因为还没有想出好的办法）。

第 3 章　控件 MSComm 串口编程

[内容提要]

本章介绍微软提供的串口编程控件 MSComm。首先详细介绍了在程序中插入该控件的基本步骤、设置方法及特性，指出了应用这个控件编程时应该注意的事项；然后通过实例介绍了该控件的应用方法。事实上，我们在第 1 章第 1.2 节就体验了如何用 MSComm 控件进行串口编程，但那只是一个粗浅的认识，真正了解这个控件还需要仔细学习本章的内容。

第 3.2 节针对 MSComm 控件在应用中的几个疑难问题做了解释，这些问题在编程中常会遇到。

第 1 章第 1.2 节是 MSComm 控件的第一个实例程序；第 3.3 节是一个如何在单文档中应用 MSComm 控件的一个实例程序；第 3.4 节的实例对在同一程序中利用 MSComm 控件控制多个串口做了说明。

详细了解该控件应该注意以下几点：一是了解如何在工程中添加 MSComm 控件；二是如何利用该控件操作串口，针对各种具体应用如何设置其属性，这就要了解 MSComm 控件本章 3.1 节的属性介绍，熟悉其内部处理机制，并了解在编程中如何应用。

还有一个疑问需要澄清：为什么我们有了 CSerialPort 类，还要在这里介绍 MSComm 控件？这是因为 CSerialPort 类毕竟有局限（非 MODEM 应用），要对 MODEM 进行控制，还需要对类进行改写。而 MSComm 控件是微软提供的，功能较完善，而且可以对中文进行处理。所以熟悉 MSComm 控件有时是必要的。

3.1　MSComm 控件介绍

3.1.1　VC 中应用 MSComm 控件编程步骤

在详细介绍 Microsoft Communications Control（MSComm）之前，可以通过动手编程来应用该控件，对它有一个感性的认识后，更容易理解其他内容。因此，如果读者以前没有用过 MSComm 控件，又没有看过本书第 1 章 1.2 节的例程，那么在继续看下去之前，建议先阅读第 1 章第 1.2 节的例程，因为该例程写得非常详细，即使不太熟悉 VC 6.0 编程集成环境，也可以跟着一步步地将程序编写出来。

这里，再简单总结一下用 MSComm 控件进行串口编程的基本步骤：

(1) 在建立的程序工程中插入 Microsoft Communications Control 控件；

(2) 添加 MSComm 控件 ID 的控制变量（或者对象）；

(3) 对串口进行初始化，设置 MSComm 控件的属性；

(4) 添加串口事件的消息处理函数 OnComm()函数，在函数中根据应用需要，编写数据处理代码；

(5) 编写串口发送等其他代码；

(6) 关闭串口。

3.1.2　MSComm 控件串行通信处理方式

Microsoft Communications Control（以下简称 MSComm）是 Microsoft 公司提供的简化 Windows 下串行通信编程的 ActiveX 控件，为应用程序提供了通过串行接口收发数据的简便方法。MSComm 控件通过串行端口传输和接收数据，为应用程序提供串行通信功能。MSComm 控件在串口编程时比较方便，程序员不必花时间去了解较为复杂的 API 函数，而且在 VC、VB、Delphi 等语言中均可使用。但也要了解一点：那就是本控件通信功能的实现，还是间接调用 Windows API 编程的结果，只是先通过 Comm.drv 解释，然后再传递给设备驱动程序进行的。

它提供了一系列标准通信命令的使用接口，利用它可以建立与串口的连接，并可以通过串口连接到其他通信设备（如调制解调器），发出命令，交换数据以及监视和响应串行连接中发生的事件和错误。MSComm 控件可用于创建电话拨号程序、串口通信程序和功能完备的终端程序。

具体来说，MSComm 控件提供了两种处理通信问题的方法：一是事件驱动（Event-Driven）法，二是查询法。

1. 事件驱动方式

事件驱动方式是处理串行端口交互作用的一种非常有效的方法。在许多情况下，在事件发生时需要得到通知，例如，在串口接收缓冲区中有字符，或者 Carrier Detect (CD) 或 Request To Send (RTS) 线上一个字符到达或一个变化发生时。在这些情况下，可以利用 MSComm 控件的 OnComm 事件捕获并处理这些通信事件。OnComm 事件还可以检查和处理通信错误。所有通信事件和通信错误的列表参阅 CommEvent 属性。在编程过程中，就可以在 OnComm 事件处理函数中加入自己的处理代码。这种方法的优点是程序响应及时，可靠性高。

对于 MSComm 控件的事件，在编程中常在串口事件消息处理函数 OnComm()中进行处理。

2. 查询方式

查询方式实质上还是事件驱动，但在有些情况下，这种方式显得更为便捷。在程序的每个关键功能之后，可以通过检查 CommEvent 属性的值来查询事件和错误。如果应用程序较小，并且是自保持的，那么这种方法可能是更可取的。例如，如果写一个简单的电话拨号程序，则没有必要对每接收一个字符都产生事件，因为惟一等待接收的字符是调制解调器的“确定”响应。

如果有过 DOS 编程经验的读者可以体会到，以上两种方式的编程与 DOS 下的中断和查询机理是一样的。

注意：在使用 MSComm 控件时，1 个 MSComm 控件只能同时对应 1 个串口。如果应用程序需要访问和控件多个串口，那么必须使用多个 MSComm 控件。

由于 MSComm 控件本身没有提供方法，所以 CMSComm 类除了 Create()成员函数外，其他的函数都是 Get/Set 函数对，用来获取或设置控件的属性。与 CSerialPort 类一样，这些函数对就是用 API 函数编写的。MSComm 控件也只有 1 个 OnComm 事件，用来向调用者通知有通信事件发生。

3.1.3 MSComm 控件的属性说明

MSComm 控件有很多重要的属性，但首先必须熟悉几个常用的属性。

- CommPort：设置并返回通信端口号。
- Settings：以字符串的形式设置并返回波特率、奇偶校验、数据位、停止位。
- PortOpen：设置并返回通信端口的状态。也可以打开和关闭端口。
- Input：从接收缓冲区返回和删除字符。
- Output：向传输缓冲区写一个字符串。

对 MSComm 控件通过 Get/Set 函数对来获取或设置控件的属性，每个属性均有与之对应的 Get/Set 函数对。下面对属性分别描述。

为了方便查询，将属性说明序号列表如下：

① CommPort 属性
② RThreshold 属性
③ CTSHolding 属性
④ SThreshold 属性
⑤ Handshaking 属性
⑥ InputMode 属性
⑦ CDHolding 属性
⑧ DSRHolding 属性
⑨ Settings 属性
⑩ InputLen 属性
⑪ InBufferSize 属性
⑫ InBufferCount 属性
⑬ OutBufferSize 属性
⑭ OutBufferCount 属性
⑮ InPut 属性
⑯ OutPut 属性
⑰ PortOpen 属性
⑱ EOFEnable 属性
⑲ DTREnable 属性
⑳ RTSEnalbe 属性
㉑ Break 属性
㉒ CommID 属性
㉓ NullDiscard 属性
㉔ CommEvent 事件属性

1. CommPort 属性

功能：设置并返回通信端口号。

语法：void CMSComm::SetCommPort(short nNewValue)　//设置串口号
　　short CMSComm::GetCommPort()　//查询当前串口号

说明：nNewValue 可以设置成从 1 到 16 的任何数（缺省值为 1）。但是，如果用 PortOpen 属性打开一个并不存在的端口时，MSComm 控件会产生错误 68（设备无效）。

注意： 必须在打开端口之前设置 CommPort 属性。

2. RThreshold 属性

功能：在 MSComm 控件设置 CommEvent 属性为 comEvReceive 并产生 OnComm 之前，设置并返回的要接收的字符数。

语法：void CMSComm::SetRThreshold(short nNewValue)

short CMSComm::GetRThreshold()

说明：接收缓冲区收到 nNewValue 个字符产生 OnComm 事件。当接收字符后，若 nNewValue 设置为 0（缺省值），则不产生 OnComm 事件。例如，设置 nNewValue 为 1，则接收缓冲区收到每一个字符都会使 MSComm 控件产生 OnComm 事件。

3. CTSHolding 属性

功能：确定是否可通过查询 Clear To Send (CTS) 线的状态发送数据。Clear To Send 是调制解调器发送到相连计算机的信号，指示传输可以进行。该属性在设计时无效，在运行时为只读。

语法：void CMSComm::SetCTSHolding(BOOL bNewValue)

BOOL CMSComm::GetCTSHolding()

MSComm 控件的 CTSHolding 属性设置值：

TRUE ：Clear To Send 线为高电平。

FALSE：Clear To Send 线为低电平。

说明：如果 Clear To Send 线为低电平 (CTSHolding = False) 并且超时时，则 MSComm 控件设置 CommEvent 属性为 comEventCTSTO (Clear To Send Timeout)，并产生 OnComm 事件。

Clear To Send 线用于 RTS/CTS (Request To Send/Clear To Send) 硬件握手。如果需要确定 Clear To Send 线的状态，则 CTSHolding 属性给出一种手工查询的方法。

4. SThreshold 属性

功能：MSComm 控件设置 CommEvent 属性为 comEvSend 并产生 OnComm 事件之前，设置并返回传输缓冲区中允许的最小字符数。

语法：void CMSComm::SetSThreshold(short nNewValue)

short CMSComm::GetSThreshold()

nNewValue 整形表达式，代表在 OnComm 事件产生之前在传输缓冲区中的最小字符数。

说明：SThreshold 属性为 0（缺省值），数据传输事件不会产生 OnComm 事件。若设置 SThreshold 属性为 1，当传输缓冲区完全空时，MSComm 控件产生 OnComm 事件。如果在传输缓冲区中的字符数小于 value，则 CommEvent 属性设置为 comEvSend，并产生 OnComm 事件。comEvSend 事件仅当字符数与 SThreshold 交叉时被激活一次。例如，如果 SThreshold 等于 5，仅当在输出队列中字符数从 5 降到 4 时，comEvSend 才发生。如果在输出队列中从没有比 SThreshold 多的字符，则 comEvSend 事件将绝不会发生。

5. Handshaking 属性

功能：设置或返回硬件握手状态，即设定串口通信设备之间的流控制。

语法：void CMSComm::SetHandshaking(long nNewValue)

long CMSComm::GetHandshaking()

说明：属性 nNewValue 值可设定为下表 3-1-1 中的值。

表 3-1-1　硬件握手信号设定值

常数	值	说明
comNone	0	无握手(默认值)
comXonXoff	1	XOn/Xoff 握手
comRTS	2	Request-to-send/clear-to-send 握手
comRTSXOnXOff	3	Request-to-send 和 clear-to-send 握手皆可

Handshaking 是指内部通信协议，通过该协议，数据从硬件端口传输到接收缓冲区。当一个数据字符到达串行端口，通信设备就把它移到接收缓冲区，以使程序可以读它。如果没有接受缓冲区，则程序需要直接从硬件读取每一个字符，这很可能会造成数据丢失，因为字符到达的速度可以非常快。

握手协议保证在缓冲区过载时数据不会丢失，缓冲区过载为数据到达端口太快而使通信设备来不及将它移到接收缓冲区。

6. InputMode 属性

功能：设置或返回传输数据的类型。

语法：void CMSComm::SetInputMode(long nNewValue)

long CMSComm::GetInputMode()

说明：InputMode 属性设定值可设定为表 3-1-2 中的值。

表 3-1-2　InputMode 属性设定值

常数	值	说明
comInputModeText	0	以文本方式取回数据
comInputModeBinary	1	以二进制方式取回数据

7. CDHolding 属性

功能：通过查询 Carrier Detect (CD) 线的状态确定当前是否有传输。Carrier Detect 是从调制解调器发送到相连计算机的一个信号，指示调制解调器正在联机。该属性在设计时无效，在运行时为只读。

语法：void CMSComm::SetCDHolding(BOOL bNewValue)

BOOL CMSComm::GetCDHolding()

说明：bNewValue 为 TRUE 时表示 True Carrier Detect 线为高电平

bNewValue 为 FALSE 时表示 True Carrier Detect 线为低电平

说明：当 Carrier Detect 线为高电平 (CDHolding = True) 且超时时，MSComm 控件设置 CommEvent 属性为 comEventCDTO(Carrier Detect 超时错误)，并产生 OnComm 事件。在主机应用程序中捕获一个丢失的传输是特别重要的，例如一个公告板，因为呼叫者可以随时挂起（放弃传输）。Carrier Detect 也被称为 Receive Line Signal Detect (RLSD)。

8. DSRHolding 属性

功能：确定 Data Set Ready (DSR) 线的状态。Data Set Ready 信号由调制解调器发送到相连计算机，指示做好操作准备。该属性在设计时无效，在运行时为只读。

语法：void CMSComm::SetDTREnable(BOOL bNewValue)
　　BOOL CMSComm::GetDTREnable()

说明：DSRHolding 属性返回以下值

值为 TRUE 表示 Data Set Ready 线为高电平

值为 FALSE 表示 Data Set Ready 线为低电平

当 Data Set Ready 线为高电平 (DSRHolding = True) 且超时时，MSComm 控件设置 CommEvent 属性为 comEventDSRTO（数据准备超时），并产生 OnComm 事件。当为 Data Terminal Equipment (DTE) 机器写 Data Set Ready/Data Terminal Ready 握手例程时，该属性是十分有用的。

9. Settings 属性

功能：设置并返回波特率、奇偶校验、数据位、停止位参数。

语法：void CMSComm::SetSettings(LPCTSTR lpszNewValue)
　　CString CMSComm::GetSettings()

说明：当端口打开时，如果设置值 lpszNewValue 非法，则 MSComm 控件产生错误 380（非法属性值）。

Value 由四个设置值组成，有如下的格式：

"BBBB,P,D,S"

BBBB 为波特率，P 为奇偶校验，D 为数据位数，S 为停止位数。value 的缺省值是："9600,N,8,1"。

10. InputLen 属性

功能：设置并返回 Input 属性从接收缓冲区读取的字符数。

语法：void CMSComm::SetInputLen(short nNewValue)
　　short CMSComm::GetInputLen()

说明：InputLen 属性的缺省值是 0。设置 InputLen 为 0 时，使用 Input 将使 MSComm 控件读取接收缓冲区中全部的内容。

若接收缓冲区中 InputLen 字符无效，Input 属性返回一个零长度字符串 ("")。在使用 Input 前，用户可以选择检查 InBufferCount 属性来确定缓冲区中是否已有需要数目的字符。该属性在从输出格式为定长数据的机器读取数据时非常有用。

11. InBufferSize 属性

功能：设置或返回输入缓冲区的大小。

语法：void CMSComm::SetInBufferSize(short nNewValue)
　　short CMSComm::GetInBufferSize()

说明：设置值的缺省（默认）大小为 1024 字节(byte)。

12. InBufferCount 属性

功能：设置或返回输入缓冲区内等待读取的字节个数。

语法：void CMSComm::SetInBufferCount(short nNewValue)
　　short CMSComm::GetInBufferCount()

说明：当设置 InBufferCount 属性的值为 0 时，可以清除空接收缓冲区，这个功能在很多时候可以用到。

13. OutBufferSize 属性

功能：设置或者返回发送缓冲区的大小。

语法：void CMSComm::SetOutBufferSize(short nNewValue)
　　short CMSComm::GetOutBufferSize()

说明：设置值为字节数，默认值为 512 字节。此值不能太小，否则缓冲区易溢出，但太大则会不必要地占用内存。

14. OutBufferCount 属性

功能：返回发送缓冲区的字节数或者清空发送缓冲区。

语法：void CMSComm::SetOutBufferCount(short nNewValue)
　　short CMSComm::GetOutBufferCount()

说明：设置值为 0 时清空发送缓冲区。

15. InPut 属性

功能：从接收缓冲区内读出数据。

语法：VARIANT CMSComm::GetInPut()

说明：返回数据类型为 VARIANT 型变量，该属性在串口没有打开时不能用，在运行时是只读的。

当 InPutMode 属性值为 0 时（检取数据为文本方式），变量中含 String 型数据。

当 InPutMode 属性值为 1 时（检取数据为二进制方式），变量中含 Byte 数组型数据。

16. OutPut 属性

功能：向发送缓冲区写数据，或返回发送缓冲区当前的数据。

语法：void CMSComm::SetOutPut(const VARIANT& newValue)
　　VARIANT CMSComm::GetOutPut()

说明：变量类型为 VARIANT，该属性在串口没有打开时不可用。OutPut 可以发送文本或二进制数据，当发送文本类型数据时，将字符型数据放入 VARIANT 型变量中；发送二进制数据（按字节发送）时，将字节型数据放入 VARIANT 型变量中。若数据中包含了内嵌控制字符、空字符等，必须将其作为二进制数据发送。

17. PortOpen 属性

功能：用于打开或关闭串口，或者返回串口的开、关状态。

语法：void CMSComm::SetPortOpen(BOOL bNewValue)
　　BOOL CMSComm::GetPortOpen()

说明：bNewValue 值设置为 TRUE，则打开串口；bNewValue 值设置为 FALSE，则关闭串口。编程时可以在程序中打开或关闭串口，当程序终止，MSComm 控件自动关闭串口。

18. EOFEnable 属性

功能：确定在输入过程中 MSComm 控件是否寻找文件结尾 (EOF) 字符。如果找到 EOF 字符，将停止输入并激活 OnComm 事件，此时 CommEvent 属性设置为 ComEvEOF。

语法：void CMSComm::SetEOFEnable(BOOL bNewValue)

BOOL CMSComm::GetEOFEnable()

说明：EOFEnable 属性语法包括下列部分。

bNewValue 布尔表达式，确定当找到 EOF 字符时，OnComm 事件是否被激活，如“设置值”中所描述。

bNewValue 的设置值如下。

TRUE：当 EOF 字符找到时，OnComm 事件被激活。

FALSE：（缺省）当 EOF 字符找到时，OnComm 事件不被激活。OnComm 控件将不在输入流中寻找 EOF 字符。

19. DTREnable 属性

功能：设置或返回 Data Terminal Ready(DTR)线状态。确定在通信时是否使 Data Terminal Ready (DTR) 线有效。Data Terminal Ready 是计算机发送到调制解调器的信号，指示计算机在等待接受传输。

语法：void CMSComm::SetDTREnable(BOOL bNewValue)

BOOL CMSComm::GetDTREnable()

bNewValue 的设置值如下表：

设置值	描述
True	使 Data Terminal Ready 线有效
False	（缺省）使 Data Terminal Ready 线无效

说明：当 DTREnable 设置为 True，当端口被打开时，Data Terminal Ready 线设置为高电平（开），当端口被关闭时，Data Terminal Ready 线设置为低电平（关）。当 DTREnable 设置为 False，Data Terminal Ready 线始终保持为低电平。

注意： 在很多情况下，把 Data Terminal Ready 线设置为低用来挂断电话。

20. RTSEnalbe 属性

功能：确定是否使 Request To Send (RTS) 线有效。一般情况下，由计算机发送 Request To Send 信号到连接的调制解调器，以请示允许发送数据。

语法：void CMSComm::SetRTSEnable(BOOL bNewValue)

BOOL CMSComm::GetRTSEnable()

bNewValue 设置值是：

设置值	描述
True	Request To Send 线有效
False	（缺省）Request To Send 线无效

说明：当 RTSEnable 设置为 True，端口打开时，Request To Send 线设置为高电平，端口关闭时，设置为低电平。

Request To Send 线用在 RTS/CTS 硬件握手。RTSEnable 属性允许手动检测 Request To Send 线以确定其状态。

有关握手协议详细信息，请参阅 Handshaking 属性。

21. Break 属性

功能：设置或清除中断信号的状态。该属性在设计时无效。

语法：void CMSComm::SetBreak(BOOL bNewValue)

BOOL CMSComm::GetBreak()

bNewValue 的设置值为：

设置值	描述
True	设置中断信号状态
False	清除中断信号状态

说明：当设置为 True，Break 属性发送一个中断信号。该中断信号挂起字符传输，并置传输线为中断状态，直到把 Break 属性设置为 False。一般，仅当使用的通信设备要求设置一个中断信号时，才设置一个短时的中断状态。

22. CommID 属性

功能：返回一个说明通信设备的句柄。该属性在设计时无效，在运行时为只读。

语法：void CMSComm::SetCommID(long nNewValue)

long CMSComm::GetCommID()

说明：该值与 Windows API CreateFile 函数返回的值一致。在 Windows API 中调用任何通信例程时使用该值。

23. NullDiscard 属性

功能：确定 Null 字符是否从端口传送到接收缓冲区。

语法：void CMSComm::SetNullDiscard(BOOL bNewValue)

BOOL CMSComm::GetNullDiscard()

bNewValue 设置值是：

设置值	描述
True	Null 字符不从端口传送到接收缓冲区
False	（缺省值）Null 字符从端口传送到接收缓冲区

说明：Null 字符定义为 ASCII 字符 0。

24. OnComm 事件和 CommEvent 属性

功能：设置或返回无论何时当 CommEvent 属性的值变化时，就产生 OnComm 事件，

标志发生了一个通信事件或一个错误。CommEvent 属性值反映错误或者事件类型，通常在程序中的事件消息处理函数中对 CommEvent 事件进行处理。

语法：void CMSComm::SetCommEvent(short nNewValue)

short CMSComm::GetCommEvent()

说明：通常由函数 GetCommEvent()得到当前事件值，再进行相应的处理。读者可以参看相应的例程来了解这一点。

根据应用程序的用途和功能，在连接到其他设备过程中，以及接收或发送数据过程中，可能需要监视并响应一些事件和错误。

可以使用 OnComm 事件和 CommEvent 属性捕捉并检查通信事件和错误的值。

在发生通信事件或错误时，将触发 OnComm 事件，CommEvent 属性的值将被改变。因此，在发生 OnComm 事件的时候，如果有必要，可以检查 CommEvent 属性的值。由于通信（特别是通过电话线的通信）是不可预料的，捕捉这些事件和错误将有助于使应用程序对这些情况做出相应的反应。

表 3-1-3 列出了可能触发 OnComm 事件的通信事件。对应的值将在发生事件时被写入 CommEvent 属性。

表 3-1-3　CommEvent 属性事件设定值和返回值

常数	值	描述
ComEvSend	1	发送缓冲区中的字符数少于 SThreshold
ComEvReceive	2	接收到 Rthreshold 个字符。在使用 Input 属性移去接收缓冲区中的数据之前，该事件将持续产生
ComEvCTS	3	CTS 信号发生变化
ComEvDSR	4	DSR 信号发生变化。该事件仅在 DSR 由 1 变为 0 时触发
ComEvCD	5	CD 信号发生变化
ComEvRing	6	检测到电话振铃。某些 UART（通用异步收发器）可能不支持本事件
ComEvEOF	7	收到文件结束符（ASCII 字符 26）

表 3-1-4 是 CommEvent 属性通信错误事件设定值和返回值说明，这些错误同样会触发 OnComm 事件，并且在 CommEvent 属性中写入相应的值。

表 3-1-4　CommEvent 属性通信错误事件设定值和返回值

设置值	值	描述
ComEventBreak	1001	收到 Break 信号
ComEventFrame	1004	帧错误。硬件检测到帧错误
ComEventOverrun	1006	端口超限。在下一个字符到达端口之前，前一字符还没有从硬件中读走，因而丢失
ComEventRxOver	1008	接收缓冲区溢出。接收缓冲区已没有空间
ComEventRxParity	1009	奇偶校验错误。硬件检测到奇偶校验错误
comEventTxFull	1010	发送缓冲区满。在试图将字符传入发送缓冲区时，该缓冲区已满
ComEventDCB	1011	在为端口获取设备控制块 (DCB) 时，发生不可预料的错误

3.1.4　MSComm 控件错误信息

MSComm 控件可以捕获的错误如表 3-1-5 所示。

表 3-1-5　MSComm 控件可以捕获的错误

常数	值	错误描述
comInvalidPropertyValue	380	无效属性值
comSetNotSupported	383	属性为只读
comGetNotSupported	394	属性为只读
comPortOpen	8000	端口打开时操作不合法
	8001	超时值必须大于 0
comPortInvalid	8002	无效端口号
	8003	属性只在运行时有效
	8004	属性在运行时为只读
comPortAlreadyOpen	8005	端口已经打开
	8006	设备标识符无效或不支持该标识符
	8007	不支持设备的波特率
	8008	指定的字节大小无效
	8009	缺省参数错误
	8010	硬件不可用（被其他设备锁定）
	8011	函数不能分配队列
comNoOpen	8012	设备没有打开
	8013	设备已经打开
	8014	不能使用 comm 通知
comSetCommStateFailed	8015	不能设置 comm 状态
	8016	不能设置 comm 事件屏蔽
comPortNotOpen	8018	仅当端口打开时操作才有效
	8019	设备忙
comReadError	8020	读 comm 设备错误
comDCBError	8021	为该端口检索设备控制块时的内部错误

3.2　使用 MSComm 控件的几个疑难问题

3.2.1　使用 VARIANT 和 SAFEARRAY 数据类型从串口读写数据

使用 MSComm 控件，在发送与接收时都要用到 VARIANT 数据类型，另外，我们还看到 SAFEARRAY（COleSafeArray）类型变量也在处理接收到的数据。这些变量类型，在实际使用中，许多编程者都感到难以理解，这里结合串口通信程序实例仔细讨论一下。

1. VARIANT、_variant_t 与 COleVariant 数据类型

VARIANT 及由之而派生出的 COleVariant 类主要用于在 OLE 自动化中传递数据。实际上，VARIANT 也只不过是一个新定义的结构罢了，它的主要成员包括一个联合体及一个变量。该联合体由各种类型的数据成员构成，而该变量则用来指明联合体中目前起作用的数据类型。

VARIANT 的结构定义如下：

```
typedef struct tagVARIANT {
   VARTYPE vt;
   unsigned short wReserved1;
   unsigned short wReserved2;
   unsigned short wReserved3;
   union {
      unsigned char        bVal;          // VT_UI1.
      Short                iVal;          // VT_I2
```

```
        Long                lVal;           // VT_I4    .
        Float               fltVal;         // VT_R4    .
        Double              dblVal;         // VT_R8    .
        VARIANT_BOOL        boolVal;        // VT_BOOL.
        SCODE               scode;          // VT_ERROR.
        CY                  cyVal;          // VT_CY    .
        DATE                date;           // VT_DATE.
        BSTR                bstrVal;        // VT_BSTR.
        IUnknown            FAR* punkVal;   // VT_UNKNOWN.
        IDispatch           FAR* pdispVal;  // VT_DISPATCH.
        SAFEARRAY           FAR* parray;    // VT_ARRAY|*.
        unsigned char       FAR* pbVal;     // VT_BYREF|VT_UI1.
        Short               FAR* piVal;     // VT_BYREF|VT_I2.
        Long                FAR* plVal;     // VT_BYREF|VT_I4.
        Float               FAR* pfltVal;   // VT_BYREF|VT_R4.
        Double              FAR* pdblVal;   // VT_BYREF|VT_R8.
        VARIANT_BOOL        FAR* pboolVal;  // VT_BYREF|VT_BOOL.
        SCODE               FAR* pscode;    // VT_BYREF|VT_ERROR.
        CY                  FAR* pcyVal;    // VT_BYREF|VT_CY.
        DATE                FAR* pdate;     // VT_BYREF|VT_DATE.
        BSTR                FAR* pbstrVal;  // VT_BYREF|VT_BSTR.
        IUnknown FAR*       FAR* ppunkVal;  // VT_BYREF|VT_UNKNOWN.
        IDispatch FAR*      FAR* ppdispVal; // VT_BYREF|VT_DISPATCH.
        SAFEARRAY FAR*      FAR* pparray;   // VT_ARRAY|*.
        VARIANT             FAR* pvarVal;   // VT_BYREF|VT_VARIANT.
        Void                FAR* byref;     // Generic ByRef.
    };
};
```

因为 Variant 数据类型能存储许多不同类型的数据，所以当希望使用用户定义类型时，在许多情况下也可以使用 Variant 数组。实际上，Variant 数组比用户定义类型更灵活，因为存储在每个元素中的数据的类型可以随时改变，而且还可以将数组定义为动态的，必要时可以改变其大小。但是，Variant 数组使用的内存总是要多于相当的用户定义类型。

对于 VARIANT 变量的赋值：首先给 vt 成员赋值，指明数据类型，再对联合结构中相同数据类型的变量赋值，举个例子：

```
VARIANT vaData;     //定义VARIANT 类型变量
int a=2001;
vaData.vt=VT_I4;    //指明整型数据
vaData.lVal=a;      //赋值
```

对于不马上赋值的 VARIANT，最好先用 Void VariantInit(VARIANTARG FAR* pvarg);进行初始化，其本质是将 vt 设置为 VT_EMPTY。 _variant_t 是 VARIANT 的封装类，其赋值可以使用强制类型转换，其构造函数会自动处理这些数据类型。使用时需加上#include <comdef.h>，例如：

```
long l=222;
ing i=100;
_variant_t lVal(l);
lVal = (long)i;
```

COleVariant 的使用与_variant_t 的方法基本一样，请参考如下例子：

```
COleVariant v3 = "字符串", v4 = (long)1999;
CString str =(BSTR)v3.pbstrVal;
long i = v4.lVal;
```

2. SAFEARRAY（COleSafeArray）数据类型

这里推荐给大家的是指向一个 SAFEARRAY（COleSafeArray）类型变量。数据类型 SAFEARRAY 正如其名字一样，是一个“安全数组”，它能根据系统环境自动调整其 16 位或 32 位的定义，并且不会被 OLE 改变（某些类型，如 BSTR，在 16 位或 32 位应用程序间传递时会被 OLE 翻译，从而破坏其中的二进制数据）。大家无须了解 SAFEARRAY 的具体定义，只要知道它是另外一个结构，其中包含一个 (void *)类型的指针 pvData，其指向的内存就是存放有用数据的地方。简而言之，从 GetInput()函数返回的 VARIANT 类型变量中，找出 parray 指针，再从该指针指向的 SAFEARRAY 变量中找出 pvData 指针，就可以向访问数组一样取得所接收到的数据了。

下面结合在 MSComm 控件中从串口发送数据和接收数据来编写几个具体的实例。

实例一：把 CString 字符串类型数据转换成 Variant 类型数据从串口发送

应用 MSComm 控件，从串口发送数据的函数原型为：

void CMSComm::SetOutput(const VARIANT& newValue)

如果发送 CString 字符串，必须将其他转化为 VARIANT 类型变量才能发送，在第 1 章第 1.2 节的实例工程 ScommTest 中，对发送的 CString 字符串做了强制转换。发送函数如下：

```
void CSCommTestDlg::OnButtonManualsend()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
//m_strEditTXData 是CString类型变量
    m_ctrlComm.SetOutput(COleVariant(m_strEditTXData));
}
```

还有的例程通过把 CString 类型数据转换成为 CbyteArray 类型数据，再转换成为 VARIANT 类型数据，好像还不如上面的代码简单，但发送字符数据时可以参考：

```
void CSCommTestDlg::OnButtonManualsend()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE); //读取编辑框内容
    char TxData[100];
 int Count = m_strEditTXData.GetLength();
    for(int i = 0; i < Count; i++)
    TxData[i] = m_strEditTXData.GetAt(i);
 CByteArray array;
 array.RemoveAll();
 array.SetSize(Count);
 for(i=0;i<Count;i++)
      array.SetAt(i, TxData[i]);
 m_ctrlComm.SetOutput(COleVariant(array)); // 发送数据
}
```

从以上代码可以看到，发送数据时，SetOutPut()函数中需要的参数还可以使用 COleVariant 类的构造函数简单生成， GetInPut()函数的返回值也成了 VARIANT 类型，那么如何从返回的值中提取有用的内容呢？

我们所关心的接收到的数据就存储在 VARIANT 联合体的某个数据成员中。该联合体中包含的数据类型很多，从一些简单的变量到非常复杂的数组和指针。由于通过串口接收到的

内容常常是一个字符串，所以将使用其中的某个数组或指针来访问接收到的数据。

实例二：把 Variant 类型数据从串口读出并转换成 CString 字符串类型数据

还是以第 1 章第 1.2 节的 ScommTest 例程为例，在串口事件处理函数 OnComm()中，应用 SAFEARRAY（COleSafeArray）类型变量。代码如下：

```
void CSCommTestDlg::OnComm()
{
    // TODO: Add your control notification handler code here
    VARIANT variant_inp;
    COleSafeArray safearray_inp;
    LONG len,k;
    BYTE rxdata[2048]; //设置BYTE数组
    CString strtemp;
    if(m_ctrlComm.GetCommEvent()==2) //事件值为2表示接收缓冲区内有字符
    {
        variant_inp=m_ctrlComm.GetInput(); //读缓冲区
        safearray_inp=variant_inp; //VARIANT型变量转换为ColeSafeArray型变量
        len=safearray_inp.GetOneDimSize(); //得到有效数据长度
        for(k=0;k<len;k++)
            safearray_inp.GetElement(&k,rxdata+k);//转换为BYTE型数组
        for(k=0;k<len;k++)          //将数组转换为Cstring型变量
        {
            BYTE bt=*(char*)(rxdata+k);   //字符型
            strtemp.Format("%c",bt);   //将字符送入临时变量strtemp存放
            m_strEditRXData+=strtemp;  //加入接收编辑框对应字符串
        }
    }
    UpdateData(FALSE); //更新编辑框内容
}
```

3.2.2　MSComm 控件能离开对话框独立存在吗

MSComm 控件（几乎是所有的控件）都必须有一个可以寄身的对话框。必须从对话框控件工具栏中把控件图标拖入对话框中，如图 3.2.1 所示。

图 3.2.1　插入 MSComm 控件后出现在对话框控件工具栏上的图标

而对话框工具栏上的图标是不能拖到视图（VIEW）中去的，因此 MSComm 是离不开对话框的。也有人通过变通的方法，如做一个隐藏的对话框，即不让对话框显示，只用 onComm 串口事件处理函数，除非经过特殊处理，否则是行不通的。

下面是一位编程者的出错经历：

“我做了个串口通信的程序(SDI)，在工程中插入 ms communications control,version 6.0，但用其方法时出现 assert failure，用 debug，问题好像是 m_pCtrlSite=FALSE。我是在 CMyView 中加入 public:CMSComm m_ComPort，再在消息映射中 initialising com，但一调用方法，就出错。”

那么，如何在 SDI/MDI 中应用 MSComm 通信控件呢？必须在 VIEW 中的 oncreate()函数中加上一些特殊的处理，并应用 MSComm 通信控件的 Create()函数来创建。MSComm 通信控件的 Create()函数原型为：

```
virtual BOOL Create(LPCTSTR lpszClassName,
     LPCTSTR lpszWindowName, DWORD dwStyle,
     const RECT& rect,
     CWnd* pParentWnd, UINT nID,
     CCreateContext* pContext = NULL)
{ return CreateControl(GetClsid(), lpszWindowName, dwStyle, rect, pParentWnd, nID); }

BOOL Create(LPCTSTR lpszWindowName, DWORD dwStyle,
     const RECT& rect, CWnd* pParentWnd, UINT nID,
     CFile* pPersist = NULL, BOOL bStorage = FALSE,
     BSTR bstrLicKey = NULL)
{ return CreateControl(GetClsid(), lpszWindowName, dwStyle, rect, pParentWnd, nID,
     pPersist, bStorage, bstrLicKey); }
```

其中，UINT nID 为该控件 ID 号。具体的创建方法如下。

假设 ID 为 IDC_MSCOMM1，m_MSComm 为控件对象，那么应该在附属的视类或框架类中用下面的语句来创建控件。

```
m_MSComm.Create(NULL,0,CRect(0,0,0,0),this,IDC_MSCOMM1);
```

具体的例程可以参考本章第 3.3 节。

[illegible]如果一个视类派生自 CFormViewo 类，它还是可以用那个控件的，因为这个类具有许多无模式对话框的特点。另外还有一个类，数据库中 CRecordView 类也依赖于特定的对话框模板资源，是基于 MDI 的，也可以用 MSCOMM 控件。

3.2.3　如何发送接收 ASCII 值为 0 和大于 128 的字符

在 C 语言中，字符串型数据以 0 值作为结束标志，因此许多编程者对如何发送 0 值字符感到困惑，同样，当 ASCII 值大于 128 时，就不是我们能看到的常规字符了。其实这些数据的发送与接收比较简单：将要发送的数据保存在有长度值的字符数组中。为了方便测试，还是用第 1 章的例程来做测试，将发送函数改写成如下的代码：

```
void CSCommTestDlg::OnButtonManualsend()
{
    // TODO: Add your control notification handler code here
//以下内容为第 1 章例程的代码
```

```
/* 注释掉这些代码
    UpdateData(TRUE); //读取编辑框内容
    m_ctrlComm.SetOutput(COleVariant(m_strEditTXData));//发送数据
    */
    unsigned char chData[8];
    chData[0]=0;    chData[1]=1;    chData[2]=13;
    chData[3]=12;   chData[4]=0;    chData[5]=10;
    chData[6]=156;   //156 的 16 进制值为 0X9C
    chData[7]=255;   //255 的 16 进制值为 0XFF

    CByteArray binData; //定义字节数组
    binData.RemoveAll(); //清空 binData
    for(int i=0;i<8;i++)
        binData.Add(chData[i]);
    COleVariant var(binData); //将 binData 转化为 Variant 数据类型
    m_ctrlComm.SetOutput(var); //发送数据
}
```

修改成以上的代码后，可以用以下方法来测试程序。

连接串口线，打开串口调试助手，选择 COM2，将“十六进制显示”选中，然后运行程序，单击“发送”按钮，在串口调试助手的接收窗口中就可以看到 0、1、13、12、0、10、156、255 的十六进制值：00 01 0D 0C 00 0A 9C FF，如图 3.2.1 所示。

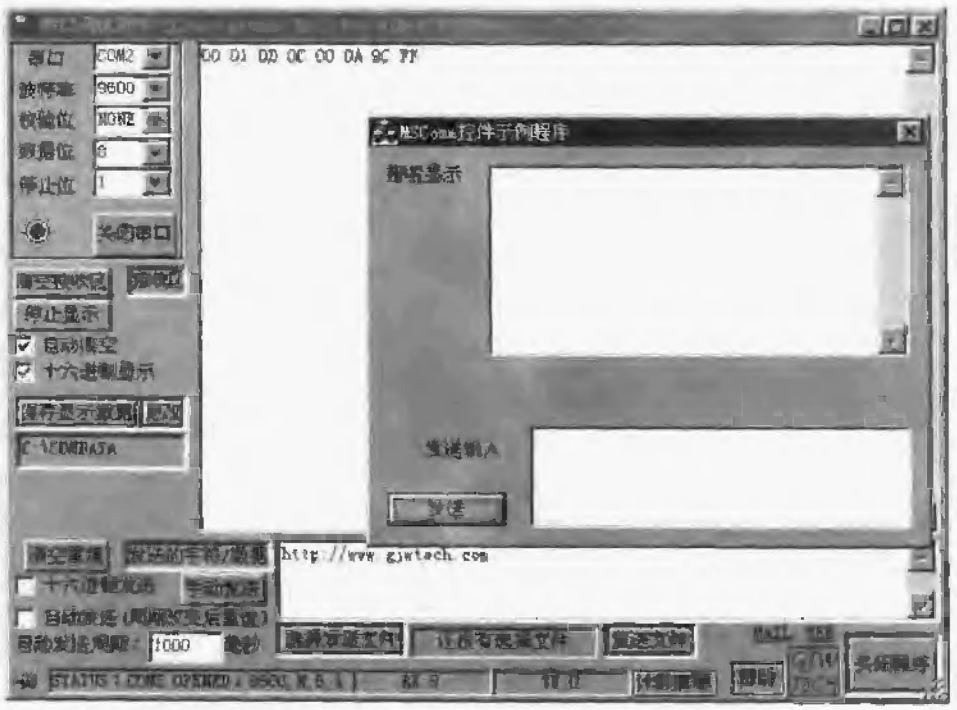


图 3.2.1　测试发送接收 ASCII 值为 0 和大于 128 的字符

接收代码的编写与第 1.2 节例程中的接收代码基本相同，只是将接收到的数据放在字符数组中。

3.2.4　在同一程序中用 MSComm 控件控制多个串口的具体操作方法

在第 3.1.2 节已经提到，在使用 MSComm 控件时，1 个 MSComm 控件只能同时对应 1 个串口。如果应用程序需要访问和控制多个串口，那么必须使用多个 MSComm 控件。具体方法就是在相应对话框中接入相应串口数的 MSComm 控件，对每个 MSComm 控件用不同的 ID，在串口初始化时，一个 MSComm 控件对应着一个串口。

在本章第 3.4 节中做了一个简单的例程来说明这个问题。

3.2.5　解决使用控件编程时程序占用的内存会不断增大的问题

这个问题作者未测试，是在网上的问题讨论中读到此问题的，读者可在实际应用中进行测试。

使用 MSComm 串口控件编写程序的时候，经常会遇到当串口接受数据量比较大时，从 Windows 2000 任务管理器里，可以看到程序占用的内存会不断增大，下面是解决的方法。

编写串口控件接收数据时，一般是这样写的：

```
//接收数据
void OnComm()
{
 VARIANT variant_inp;
 COleSafeArray safearray_inp;
 LONG len,k;
 BYTE rxdata[5]; //设置BYTE数组
 CString strtemp;
 switch(m_msComm.GetCommEvent())
 {
 case 2://事件值为2表示接收缓冲区内有字符
  variant_inp=m_msComm.GetInput(); //读缓冲区**************
  safearray_inp = variant_inp; //VARIANT型变量转换为ColeSafeArray型变量
  len=safearray_inp.GetOneDimSize(); //验证得到有效数据长度
  assert(len == 5);
  for(k=0;k<len;k++)//len is 5 in normal case
   safearray_inp.GetElement(&k,rxdata+k);//转换为BYTE型数组
    HandleCommand(rxdata);//这是接收数据处理函数
 break;
 default:
  break;
 }
}
```

问题出在 VARIANT 型变量上，当执行到*号的这一行时，VARIANT 型变量就会使内存增加，可以用下面的方法解决：

```
void OnComm()
{

 long len,k;
 COleVariant myVar;
 COleSafeArray safearray_inp;
 BYTE rxdata[5]; //设置BYTE数组
 Switch (m_msComm.GetCommEvent())
 {
 case 2://事件值为2表示接收缓冲区内有字符
```

```
    myVar.Attach (m_msComm.GetInput());  //读缓冲区**************
    safearray_inp = myVar;  //COleVariant 型变量转换为 ColeSafeArray 型变量
    len=safearray_inp.GetOneDimSize(); //验证得到有效数据长度
    assert(len == 5);
    for(k=0;k<len;k++)   //len is 5 in normal case
     safearray_inp.GetElement(&k,rxdata+k);   //转换为 BYTE 型数组
    HandleCommand (rxdata); };     //这是接收数据处理函数
    break;
  }
  }
```

3.2.6　在 MSComm 控件串口编程时遇到的其他问题

在应用 MSComm 控件进行串口编程时，还会遇到许多其他问题：比如，如何发送十六进制数据、如何处理数据包、如何应用串口流控制等，这些问题与用 API 进行编程和用其他类编程时同样会遇到，因此会在以后的相关章节对这些问题做出说明和示例。

3.3　在基于单文档（SDI）程序中应用 MSComm 控件

本节为 MSComm 控件在 VC 中 MFC 单文档（或多文档）程序中的应用实例。

编程任务：

在基于单文档的 MFC 应用程序中应用 MSComm 控件编程实现串口接收与发送，COM2（串口号根据实际情况可以更改）每收到 5 个字符触发一个串口接收事件，将接收到的数据显示在视图中，并向发送方返回应答信息。

详细编程步骤如下。

1．建立程序工程并插入 MSComm 控件

利用 MFC 向导建立基于单文档应用程序 SDIComm，所有步骤缺省，在项目中插入 MSCOMM 控件(Project->Add to Project->Components and Control...->Registered Active Controls->Microsoft Commmunications Control,V6.0，单击 INSERT（插入 MSComm 控件的具体图示过程请参考第 1 章第 1.2 节的例程 ScommTest)。

2．在 ABOUT 对话框中拖入 MSComm 控件

如图 3.3.1 所示，在 ResourceView 中选择 IDD_ABOUTBOX 对话框，将控件图标拖入对话框中，控件 ID 号保持缺省的 IDC_MSCOMM1。

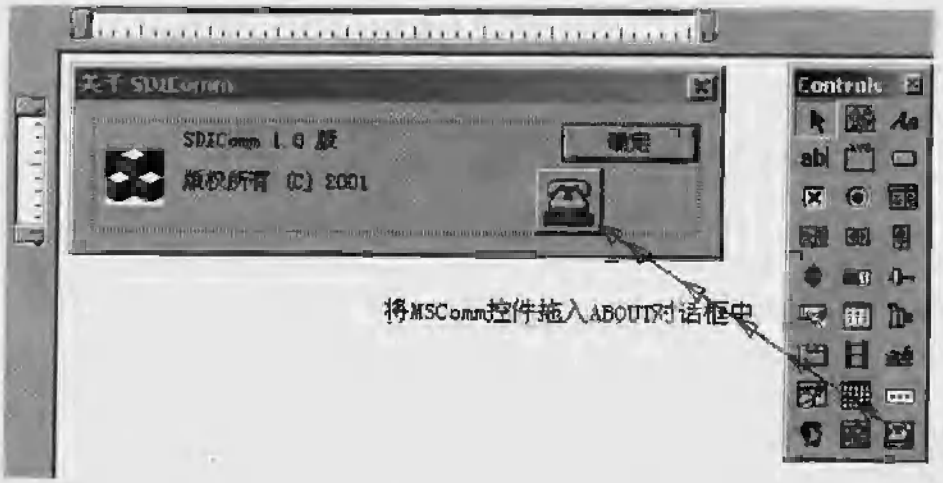


图 3.3.1　在 ABOUT 对话框中拖入 MSComm 控件图标

3. 添加串口事件消息处理函数 OnComm()

在第 1.2 节的例程 ScommTest 中，这一过程可以由 ClassWizard 自动实现，但在这里必须手工添加。

（1）对 SDICommView.h 的处理

首先添加 MSComm 控件类的头文件：#include "mscomm.h"。

再加入 CMSCOMM 类 PUBLIC 对象定义：CMSComm m_MSComm。

接着在//{{AFX_MSG(CSDICommView)和//}}AFX_MSG 之间加入以下两行：

```
afx_msg void OnComm();
DECLARE_EVENTSINK_MAP();
```

加入后的结果是：

```
//{{AFX_MSG(CSDICommView)
afx_msg void OnComm(); //事件处理函数
DECLARE_EVENTSINK_MAP()
//}}AFX_MSG
```

①注意： 别忘了加上 DECLARE_EVENTSINK_MAP().

（2）对 SDICommView.cpp 的处理

添加以下代码：

```
BEGIN_EVENTSINK_MAP(CSDICommView, CView)
    //{{AFX_EVENTSINK_MAP(CAboutDlg)
    ON_EVENT(CSDICommView, IDC_MSCOMM1, 1, OnComm, VTS_NONE)
    //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()

void CSDICommView::OnComm()
{
```

```
        // TODO: Add your control notification handler code here
    }
```

4．创建控件并初始化串口

利用 ClassWizard 为 CSDICommView 类添加 WM_CREATE 函数，该函数在视窗初始化时调用。方法是在 ClassWlizard 中选择 Message Map 卡，在 Object IDs 中选择 CSDICommView，在 Messages 中选择 WM_CREATE，双击添加 int CSDICommView::OnCreate(LPCREATESTRUCT lpCreateStruct)函数，如图 3.3.2 所示。

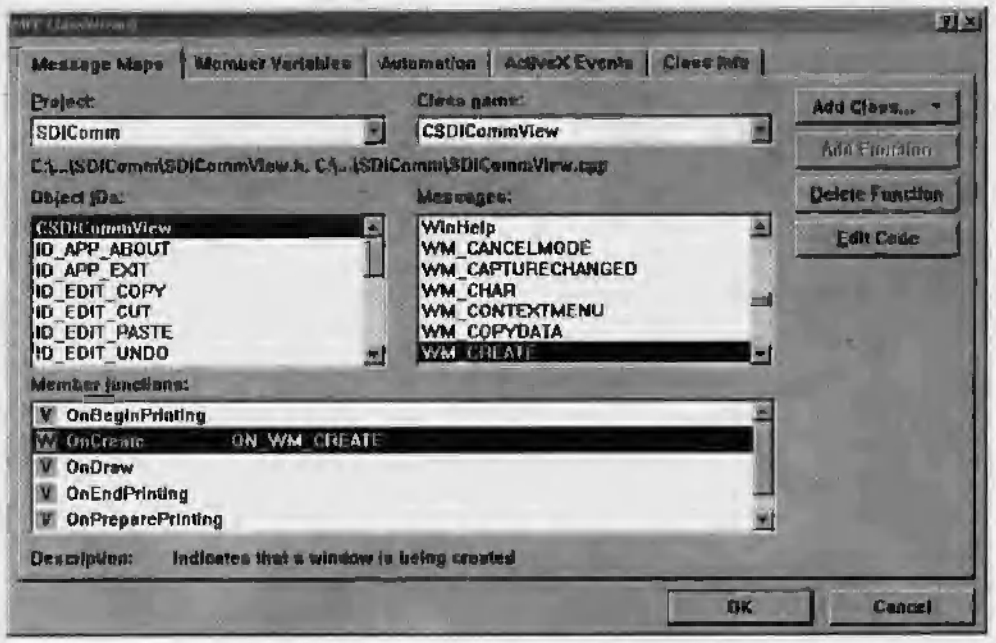


图 3.3.2　在 CSDICommView 中添加 OnCreate()函数

```
int CSDICommView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here
    m_MSComm.Create(NULL,0,CRect(0,0,0,0),this,IDC_MSCOMM1);  //创建控件

    m_MSComm.SetCommPort(2);                  //选择 COM2
    m_MSComm.SetInBufferSize(1024);           //接收缓冲区
    m_MSComm.SetOutBufferSize(1024);          //发送缓冲区
    m_MSComm.SetInputLen(0);  //设置当前接收区数据长度为 0,表示全部读取
    m_MSComm.SetInputMode(1);                 //以二进制方式读写数据
    m_MSComm.SetRThreshold(5);                //接收缓冲区有 5 个及 5 个以上字符时,
                                              //将引发接收数据的 OnComm 事件
    m_MSComm.SetSettings("9600,n,8,1");       //波特率 9600，无检验位，8 个数据位，1 个停止位

    if(!m_MSComm.GetPortOpen())               //如果串口没有打开则打开
        m_MSComm.SetPortOpen(TRUE);           //打开串口
```

```
        else
            AfxMessageBox("Open Serial Port Failure!");
        m_MSComm.GetInput();    //先预读缓冲区以清除残留数据

        return 0;
    }
```

①注意：在以上代码中，应该注意：

m_MSComm.Create(NULL,0,CRect(0,0,0,0), this,IDC_MSCOMM1); 一句，其功能是初始化串口类对象 m_MSComm，这在基于对话框中的程序是由 ClassWizard 自动完成的。注意 IDC_MSCOMM1 还是源于对话框中的控件 ID 规则，而且 IDC_MSCOMM1 必须与串口控件对应，在这里最好利用一个对话框，直接将控件拖入对话框中，就可以省不少事，以上的步骤中（第 2 步）利用了 ABOUT 对话框。

5、根据编程任务完成串口数据的读写并显示在视图中

代码如下：

```
void CSDICommView::OnComm()
{
    // TODO: Add your control notification handler code here
    VARIANT variant_inp;
    COleSafeArray safearray_inp;
    LONG len,k;
    BYTE rxdata[2048]; //设置 BYTE 数组
    CString strtemp;
    CString strDisp; //用于显示接收数据

  if(m_MSComm.GetCommEvent()==2)
    {

        variant_inp=m_MSComm.GetInput(); //读缓冲区
        safearray_inp=variant_inp; //VARIANT 型变量转换为 ColeSafeArray 型变量
        len=safearray_inp.GetOneDimSize(); //得到有效数据长度
        for(k=0;k<len;k++)
            safearray_inp.GetElement(&k,rxdata+k);//转换为 BYTE 型数组
        for(k=0;k<len;k++)          //将数组转换为 Cstring 型变量
        {
            BYTE bt=*(char*)(rxdata+k);   //字符型
            strtemp.Format("%c",bt);   //将字符送入临时变量 strtemp 存放
            strDisp+=strtemp; //加入接收编辑框对应字符串
        }

        CDC* pDC=GetDC();   //准备数据显示
        pDC->TextOut(200,100,"COM2 接收到: "+strDisp);//显示接收到的数据
        ReleaseDC(pDC);

        strtemp.Format("OK,'%s' Received ",strDisp);
        m_MSComm.SetOutput(COleVariant(strtemp));//发送数据
    }
}
```

现在，全部代码已经编写完成，最后，SDICommView.h 代码如下：

```
// SDICommView.h : interface of the CSDICommView class
//
/////////////////////////////////////////////////////////////////////////////
```

```
… …
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "mscomm.h"  //添加MSComm控件类头文件

class CSDICommView : public CView
{
protected: // create from serialization only
    CSDICommView();
    DECLARE_DYNCREATE(CSDICommView)

// Attributes
public:
    CSDICommDoc* GetDocument();

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CSDICommView)
    public:
    virtual void OnDraw(CDC* pDC);  // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    //}}AFX_VIRTUAL

// Implementation
public:
    CMSComm m_MSComm;  //MSComm类对象
    virtual ~CSDICommView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CSDICommView)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnComm();    //事件处理函数
    DECLARE_EVENTSINK_MAP()
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in SDICommView.cpp
inline CSDICommDoc* CSDICommView::GetDocument()
   { return (CSDICommDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////
```

```
    //{{AFX_INSERT_LOCATION}}
    // Microsoft Visual C++ will insert additional declarations immediately before the previous
line.

    #endif
// !defined(AFX_SDICOMMVIEW_H__C97BA3D3_0F5F_4DD7_8237_5227C645AF90__INCLUDED_)
```

程序测试：

程序编写完成后，将串口线连接好后，注意例程的串口设置在 COM2（如果与您的具体情况不一致，可在程序中更改）。运行程序。

再将串口调试助手设置在 COM1，9600，n,8,1，因为在程序中用以下语句

```
m_MSComm.SetRThreshold(5);
```

将串口接收事件设置成接收缓冲区有 5 个及 5 个以上字符时才引发接收数据的 OnComm 事件，因此，可在串口调试助手的发送区中填上单个字符（比如 1），再单击“手动发送”5 次，就可以看到本实例程序 SDIComm 的视图中出现了“COM2 接收到：11111”，同时，在串口调试助手的接收窗口中看到了返回信息：“OK，‘11111’Received”。如图 3.3.3 和图 3.3.4 所示。

图 3.3.3　程序 SDIComm 接收到 5 个字符

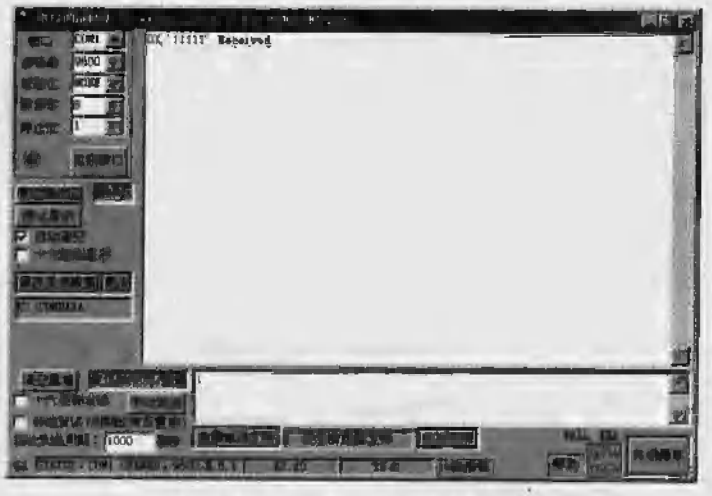


图 3.3.4　串口调试助手接收到返回信息

3.4　应用 MSComm 控件控制多个串口实例

本节为 MSComm 控件在 VC 的 MFC 程序中控制多个串口的应用实例。在使用 MSComm 控件时，1 个 MSComm 控件只能同时对应 1 个串口。如果应用程序需要访问和控制多个串口，那么必须使用多个 MSComm 控件。

编程任务：

在基于对话框的 MFC 应用程序中应用 MSComm 控件编程实现对 COM1 和 COM2 的串口控制，双方同时可以发送和接收数据，并把接收到的数据显示在接收编辑框中。

1. 建立应用程序工程 MSCommMultiPort

打开 VC 6.0，建立一个基于对话框的 MFC 应用程序：MSCommMultiPort。然后在主对话框中添加控件，最后效果如图 3.4.1 所示。其中的电话状图标是 MSComm 控件，参照第 2 步的方法添加到对话框中。

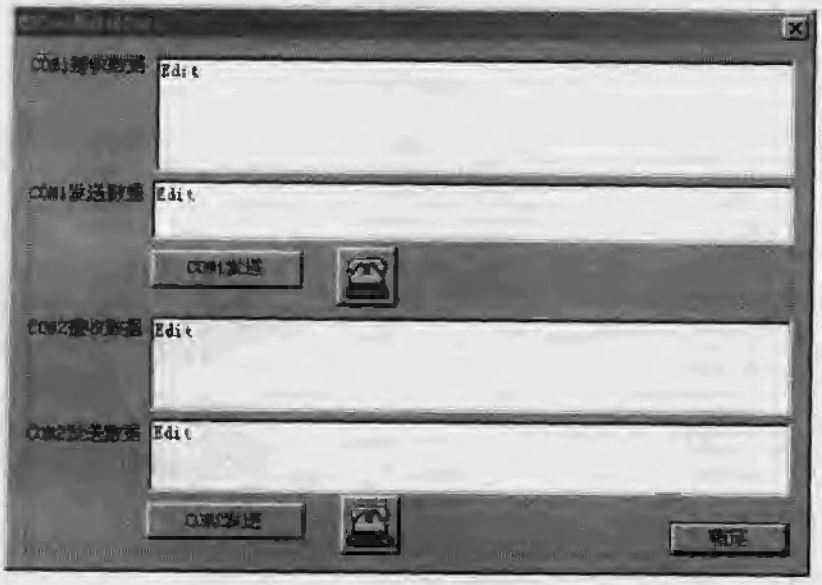


图 3.4.1　对话框添加控件后的状态

然后用 ClassWizard 为相应控件添加变量，如图 3.4.2 所示，控件的属性设置情况如表 3-4-1 所示。

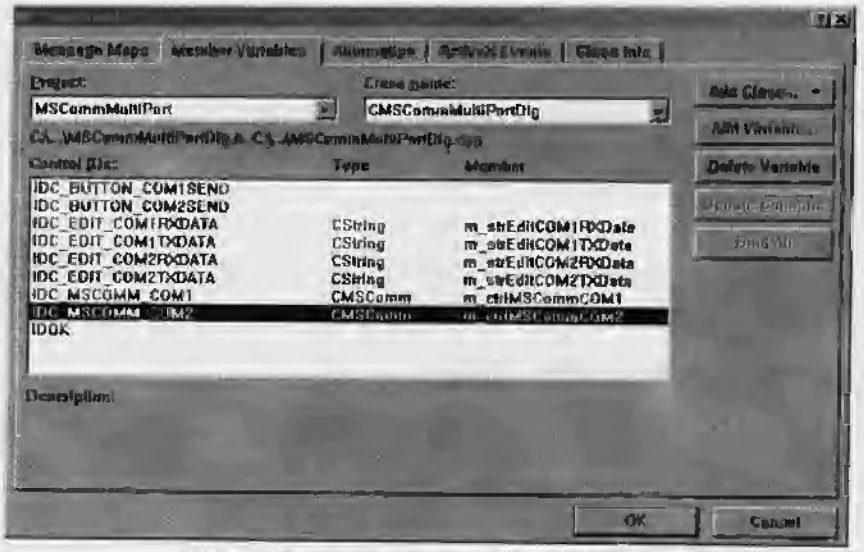


图 3.4.2　在 ClassWizard 中为控件添加相应变量

表 3-4-1　控件及其属性设置情况

控件	控件 ID	Caption	需要添加的变量及变量类型
静态文本	IDC_STATIC	COM1 接收数据	
静态文本	IDC_STATIC	COM1 发送数据	
静态文本	IDC_STATIC	COM2 接收数据	
静态文本	IDC_STATIC	COM2 发送数据	
编辑框	IDC_EDIT_COM1RXDATA		m_strEditCOM1RXData　CString
编辑框	IDC_EDIT_COM1TXDATA		m_strEditCOM1TXData　CString
编辑框	IDC_EDIT_COM2RXDATA		m_strEditCOM2RXData　CString
编辑框	IDC_EDIT_COM2TXDATA		m_strEditCOM2TXData　CString
按钮	IDC_BUTTON_COM1SEND	COM1 发送	
按钮	IDC_BUTTON_COM2SEND	COM2 发送	
MSComm	IDC_MSCOMM_COM1		m_ctrlMSCommCOM1　CMSComm
MSComm	IDC_MSCOMM_COM2		m_ctrlMSCommCOM2　CMSComm

2. 在当前工程中添加 MSComm 控件

单击菜单 Add To Project-> Components and Controls...，在当前工程中插入 MSComm 控件。插入 MSComm 控件的具体方法可以参照第 1 章第 1.2 节。

3. 初始化串口：设置 MSComm 控件的属性

打开 ClassWizard->Member Variables 页，如图 3.4.2 所示，分别为控件 IDC_MSCOMM_COM1 和 IDC_MSCOMM_COM2 添加控制变量 m_ctrlMSCommCOM1 和 m_ctrlMSCommCOM2。

通过以上操作，ClassWizard 自动在 MSCommMultiPortDlg.h 中加入了#include "mscomm.h"语句。

```
//{{AFX_INCLUDES()
#include "mscomm.h"
//}}AFX_INCLUDES
```

下面在 CMSCommMultiPortDlg::OnInitDialog()函数中写入对串口的初始化语句，串口初始化语句分别由 IDC_MSCOMM_COM1 和 IDC_MSCOMM_COM2 的 CMSComm 控制变量 m_ctrlMSCommCOM1 和 m_ctrlMSCommCOM1 来设置串口控件属性。注意这里有两个串口，需要分别对串口进行设置。代码如下：

```
BOOL CMSCommMultiPortDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    // Add "About..." menu item to system menu.
    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);
    CMenu* pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
        }
    }
    // Set the icon for this dialog. The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);          // Set big icon
    SetIcon(m_hIcon, FALSE);         // Set small icon
        // TODO: Add extra initialization here
    ////////////////////初始化 COM1/////////////////////
    m_ctrlMSCommCOM1.SetCommPort(1);  //选择 COM1
    m_ctrlMSCommCOM1.SetInputMode(1); //输入方式为二进制方式
    m_ctrlMSCommCOM1.SetInBufferSize(1024); //设置输入缓冲区大小
    m_ctrlMSCommCOM1.SetOutBufferSize(512); //设置输出缓冲区大小
    //波特率 9600，无校验，8 个数据位，1 个停止位
    m_ctrlMSCommCOM1.SetSettings("9600,n,8,1");
    //参数 1 表示每当串口接收缓冲区中有多于
    //或等于 1 个字符时将引发一个接收数据的 OnComm 事件
    m_ctrlMSCommCOM1.SetRThreshold(1);
    if(!m_ctrlMSCommCOM1.GetPortOpen())
        m_ctrlMSCommCOM1.SetPortOpen(TRUE);//打开串口
    m_ctrlMSCommCOM1.SetInputLen(0); //设置当前接收区数据长度为 0
    m_ctrlMSCommCOM1.GetInput();   //先预读缓冲区以清除残留数据

    ////////////////////初始化 COM2/////////////////////
    m_ctrlMSCommCOM2.SetCommPort(2);  //选择 COM2
    m_ctrlMSCommCOM2.SetInputMode(1); //输入方式为二进制方式
    m_ctrlMSCommCOM2.SetInBufferSize(1024); //设置输入缓冲区大小
    m_ctrlMSCommCOM2.SetOutBufferSize(512); //设置输出缓冲区大小
    //波特率 9600，无校验，8 个数据位，1 个停止位
    m_ctrlMSCommCOM2.SetSettings("9600,n,8,1");
    //参数 1 表示每当串口接收缓冲区中有多于
    //或等于 1 个字符时将引发一个接收数据的 OnComm 事件
    m_ctrlMSCommCOM2.SetRThreshold(1);
    if(!m_ctrlMSCommCOM2.GetPortOpen())
        m_ctrlMSCommCOM2.SetPortOpen(TRUE);//打开串口
    m_ctrlMSCommCOM2.SetInputLen(0); //设置当前接收区数据长度为 0
    m_ctrlMSCommCOM2.GetInput();   //先预读缓冲区以清除残留数据
```

```
        return TRUE;  // return TRUE unless you set the focus to a control
    }
```

4. 添加串口事件消息处理函数 OnComm()

MSComm 控件一般用事件驱动方式来从串口接收数据，也就是消息处理，当串口有事件发生时，程序调用消息函数来处理数据。如图 3.4.3 所示，打开 ClassWizard→Message Maps，在 Class Name 中选择类 CMSCommMultiPortDlg，再在 Object Ids 中选择 IDC_MSCOMM_COM1/ IDC_MSCOMM_COM2，然后在 Message 中双击消息 OnComm（或单击"Add Function"按钮，在弹出的对话框中将函数名改为 OnCommCOM1/OnCommCOM2，单击 OK，就加入了串口事件的消息处理函数（注意，两个函数要分别添加）。

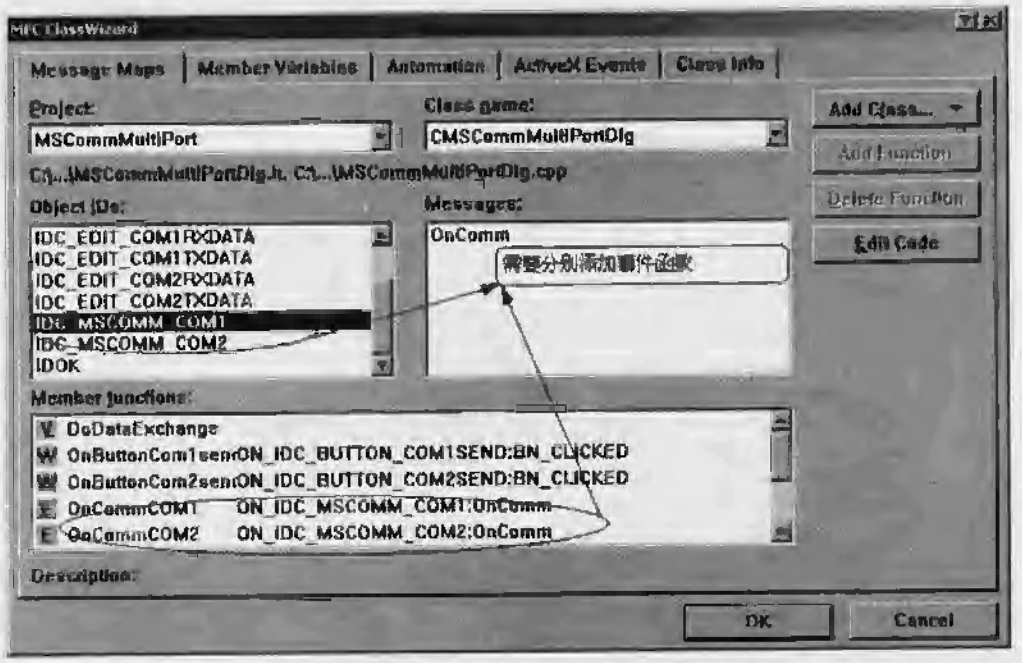


图 3.4.3　为两个 MSComm 控件分别添加消息事件处理函数 OnComm()

注意到，自动添加了以下代码。

头文件 MSCommMultiPortDlg.h 中：

```
// Generated message map functions
//{{AFX_MSG(CMSCommMultiPortDlg)
afx_msg void OnCommCOM1();
afx_msg void OnCommCOM2();
DECLARE_EVENTSINK_MAP()
//}}AFX_MSG
```

实现文件 MSCommMultiPortDlg.cpp 中：

```
BEGIN_EVENTSINK_MAP(CMSCommMultiPortDlg, CDialog)
```

```
    //{{AFX_EVENTSINK_MAP(CMSCommMultiPortDlg)
     ON_EVENT(CMSCommMultiPortDlg, IDC_MSCOMM_COM1, 1 /* OnComm */, OnCommCOM1, VTS_NONE)
     ON_EVENT(CMSCommMultiPortDlg, IDC_MSCOMM_COM2, 1 /* OnComm */, OnCommCOM2, VTS_NONE)
     //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()

void CMSCommMultiPortDlg::OnCommCOM1()
{
// TODO: Add your control notification handler code here
}

void CMSCommMultiPortDlg::OnCommCOM2()
{
   // TODO: Add your control notification handler code here
}
```

下面分别编写函数 OnCommCOM1()和 OnCommCOM2()中的代码。主要任务是从串口接收数据并显示在接收编辑框中。

```
void CMSCommMultiPortDlg::OnCommCOM1()
{
    // TODO: Add your control notification handler code here
    VARIANT variant_inp;
    COleSafeArray safearray_inp;
    LONG len,k;
    BYTE rxdata[2048]; //设置 BYTE 数组
    CString strtemp;
    if(m_ctrlMSCommCOM1.GetCommEvent()==2) //事件值为 2 表示接收缓冲区内有字符
    {
        variant_inp=m_ctrlMSCommCOM1.GetInput(); //读缓冲区
        safearray_inp=variant_inp; //VARIANT 型变量转换为 ColeSafeArray 型变量
        len=safearray_inp.GetOneDimSize(); //得到有效数据长度
        for(k=0;k<len;k++)
            safearray_inp.GetElement(&k,rxdata+k);//转换为 BYTE 型数组
        for(k=0;k<len;k++)          //将数组转换为 Cstring 型变量
        {
            BYTE bt=*(char*)(rxdata+k);   //字符型
            strtemp.Format("%c",bt);   //将字符送入临时变量 strtemp 存放
            m_strEditCOM1RXData+=strtemp; //加入接收编辑框对应字符串
        }
    }
    UpdateData(FALSE); //更新编辑框内容
}

void CMSCommMultiPortDlg::OnCommCOM2()
{
    // TODO: Add your control notification handler code here
    VARIANT variant_inp;
    COleSafeArray safearray_inp;
    LONG len,k;
    BYTE rxdata[2048]; //设置 BYTE 数组
    CString strtemp;
if(m_ctrlMSCommCOM2.GetCommEvent()==2)//事件值为 2 表示接收缓冲区内有字符
    {
        variant_inp=m_ctrlMSCommCOM2.GetInput(); //读缓冲区
        safearray_inp=variant_inp; //VARIANT 变量转换为 ColeSafeArray 型变量
        len=safearray_inp.GetOneDimSize(); //得到有效数据长度
        for(k=0;k<len;k++)
            safearray_inp.GetElement(&k,rxdata+k);//转换为 BYTE 型数组
```

```
            for(k=0;k<len;k++)              //将数组转换为 Cstring 型变量
            {
                BYTE bt=*(char*)(rxdata+k);    //字符型
                strtemp.Format("%c",bt);    //将字符送入临时变量 strtemp 存放
                m_strEditCOM2RXData+=strtemp;  //加入接收编辑框对应字符串
            }
        }
        UpdateData(FALSE);  //更新编辑框内容
    }
```

5. 发送数据

先为发送按钮添加一个单击消息即 BN_CLICKED 处理函数，打开 ClassWizard→Message Maps，选择类 CMSCommMultiPortDlg，选中 IDC_BUTTON_COM1SEND，双击 BN_CLICKED 添加 OnButtonCom1send() 函数，如图 3.4.4 所示。再按同样的方式为按钮 IDC_BUTTON_COM2SEND 添加 OnButtonCom2send()函数。

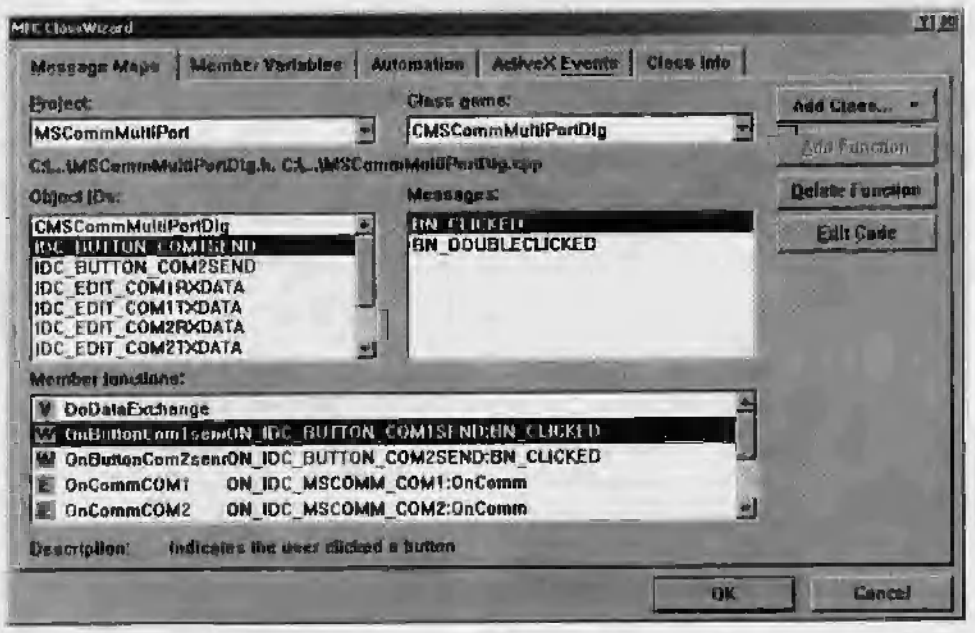


图 3.4.4　添加 OnButtonManualsend()函数

然后，在函数中添加如下代码：

```
void CMSCommMultiPortDlg::OnButtonCom1send()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE); //读取编辑框内容
//发送数据
    m_ctrlMSCommCOM1.SetOutput(COleVariant(m_strEditCOM1TXData));
}

void CMSCommMultiPortDlg::OnButtonCom2send()
{
    // TODO: Add your control notification handler code here
```

```
        UpdateData(TRUE); //读取编辑框内容
    //发送数据
    m_ctrlMSCommCOM2.SetOutput(COleVariant(m_strEditCOM2TXData));}
```

全部程序编写完成后，MSCommMultiPortDlg.h 文件代码如下：

```
    // MSCommMultiPortDlg.h : header file
    //
    //{{AFX_INCLUDES()
    #include "mscomm.h"
    //}}AFX_INCLUDES

    #if !defined(AFX_MSCOMMMULTIPORTDLG_H__FEFC79A6_5B50_11D8_870F_00E04C3F78CA__INCLUDE
D_)
    #define
    AFX_MSCOMMMULTIPORTDLG_H__FEFC79A6_5B50_11D8_870F_00E04C3F78CA__INCLUDED_
    #if _MSC_VER > 1000
    #pragma once
    #endif // _MSC_VER > 1000
    /////////////////////////////////////////////////////////////////////////////
    // CMSCommMultiPortDlg dialog

    class CMSCommMultiPortDlg : public CDialog
    {
    // Construction
    public:
        CMSCommMultiPortDlg(CWnd* pParent = NULL); // standard constructor

    // Dialog Data
        //{{AFX_DATA(CMSCommMultiPortDlg)
        enum { IDD = IDD_MSCOMMMULTIPORT_DIALOG };
        CString  m_strEditCOM1RXData;
        CString  m_strEditCOM1TXData;
        CString  m_strEditCOM2TXData;
        CString  m_strEditCOM2RXData;
        CMSComm  m_ctrlMSCommCOM1;
        CMSComm  m_ctrlMSCommCOM2;
        //}}AFX_DATA

        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CMSCommMultiPortDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
        //}}AFX_VIRTUAL

    // Implementation
    protected:
        HICON m_hIcon;

        // Generated message map functions
        //{{AFX_MSG(CMSCommMultiPortDlg)
        virtual BOOL OnInitDialog();
        afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
        afx_msg void OnPaint();
        afx_msg HCURSOR OnQueryDragIcon();
        afx_msg void OnButtonCom1send();
        afx_msg void OnButtonCom2send();
        afx_msg void OnCommCOM1();
        afx_msg void OnCommCOM2();
        DECLARE_EVENTSINK_MAP()
```

```
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
    };
    //{{AFX_INSERT_LOCATION}}
    //Microsoft Visual C++ will insert additional declarations immediately before the previous
line.
    #endif
// !defined(AFX_MSCOMMMULTIPORTDLG_H__FEFC79A6_5B50_11D8_670F_00E04C3F78CA__INCLUDED_)
```

程序测试：

需要两个串口来测试程序，这两个串口必须在同一台计算机上，用串口线连接。运行程序后，在 COM1 发送编辑框中输入字符，单击“COM1 发送”按钮，可以看到 COM2 接收编辑框中收到了相应字符；同样，在 COM2 发送编辑框中输入字符，单击“COM2 发送”按钮，也可以看到 COM1 接收编辑框中收到了相应字符。图 3.4.5 是本实例的测试效果。

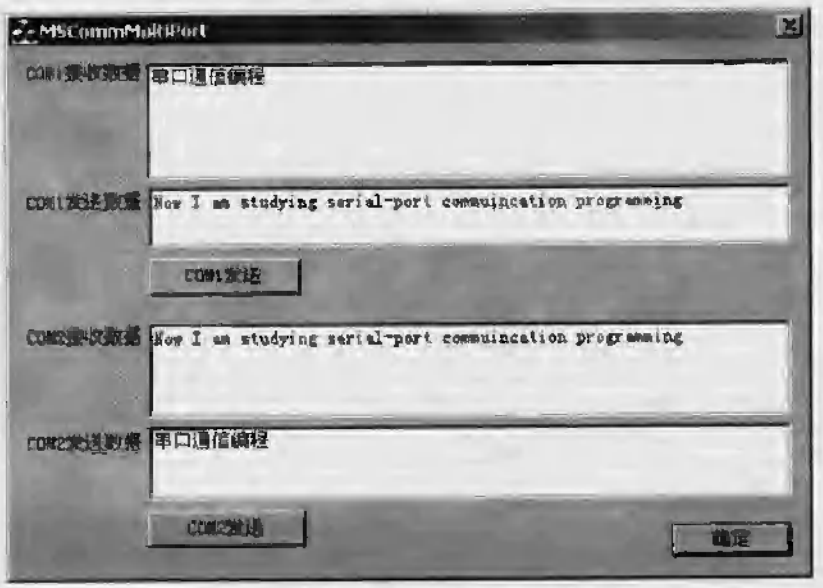


图 3.4.5　程序 MSCommMultiPort 测试效果

3.5　串口与 MODEM 拨号应用简例

本例利用 MSCOMM 控件，通过串口发送 AT 指令来操作 MODEM，使两台计算机通过公用电话网进行数据传送。

3.5.1　创建工程

（1）创建一个基于对话框的工程，并命名为 ModemComm。

（2）单击菜单：工程－添加工程－components and controls，从弹出的对话框中双击 Registered ActiveX Controls，然后在文件夹中选取 Microsoft Communications Control,version 6.0，然后单击 insert 按钮，弹出确认对话框，点确认即可。此时可以看到串口控件图标 已

经添加到控件工具箱中。

（3）应用控件工具箱创建图 3.5.1 所示对话框。

图 3.5.1　对话框

其中，COM1，COM2 控件对象 ID 分别设置为 IDC_RADIO_COM1，IDC_RADIO_COM2；呼叫号码、发送字符、接收区对应的 EDIT 控件 ID 分别设置为 IDC_EDIT_TELPHONENO、IDC_EDIT_SEND，IDC_EDIT_RECEIVE；拨号、发送 BUTTON 控件 ID 分别设置为 IDC_BUTTON_DIAL、IDC_BUTTON_SEND。串口控件 ID 为 IDC_MSCOMM1。

COM1 控件对话框属性如图 3.5.2 所示设置；IDC_EDIT_RECEIVE 控件对话框如图 3.5.3 设置；其余控件对话框则采用系统默认设置。

图 3.5.2　COM1 控件属性设置

图 3.5.3　IDC_EDIT_RECEIVE 控件属性设置

（4）单击菜单查看—建立类向导打开 MFC ClassWizard 对话框，单击 Member Variables 标签，按图 3.5.4 所示内容添加相应的成员变量

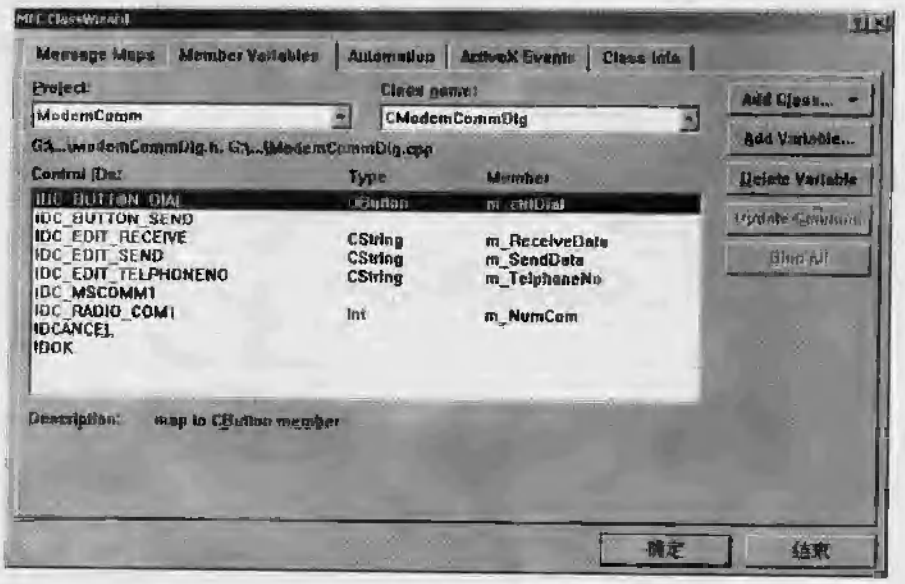


图 3.5.4　添加成员变量

3.5.2　代码分析

■　ModemCommDlg 头文件

```
// ModemCommDlg.h : header file
/*为能使用串口控件，需要在头文件中手动添加如下语句*/
#include "mscomm.h"/////////

#if !defined(AFX_MODEMCOMMDLG_H__548C9AA6_884C_43B0_9505_BC2A41328CB3__INCLUDED_)
#define
AFX_MODEMCOMMDLG_H__548C9AA6_884C_43B0_9505_BC2A41328CB3__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

/////////////////////////////////////////////////////////////////////////////
// CModemCommDlg dialog

class CModemCommDlg : public CDialog
{
// Construction
public:
    CModemCommDlg(CWnd* pParent = NULL);  // standard constructor

   //added begin
    void OpenComm(int number);//
    void SendString(CString m_strSend);//声明发送字符函数

    CMSComm  m_ctrlComm;//为串口控件增加实例变量
```

```
        BOOL bOpen;//added end
    // Dialog Data
        //{{AFX_DATA(CModemCommDlg)
        enum { IDD = IDD_MODEMCOMM_DIALOG };
        CButton   m_ctrlDial;
        CString   m_ReceiveData;
        CString   m_SendData;
        CString   m_TelphoneNo;
        int       m_NumCom;
        //}}AFX_DATA

        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CModemCommDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
        //}}AFX_VIRTUAL

    // Implementation
    protected:
        HICON m_hIcon;

        // Generated message map functions
        //{{AFX_MSG(CModemCommDlg)
        virtual BOOL OnInitDialog();
        afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
        afx_msg void OnPaint();
        afx_msg HCURSOR OnQueryDragIcon();
```

/*单击菜单：查看－建立类向导，打开 MFC ClassWizard 对话框，单击 Message Maps 标签，为 IDC_MSCOMM1 控件的消息 OnComm 添加消息映射函数 OnComm()；类似地，分别为控件 IDC_RADIO_COM1，IDC_RADIO_COM2，IDC_BUTTON_DIAL，IDC_BUTTON_SEND 添加映射函数 OnRadioCom1()、OnRadioCom2()、OnDial()、OnSend()。单击确定后，系统自动完成这些操作，下面阴影部分即为其结果。*/

```
        afx_msg void OnComm();
        afx_msg void OnRadioCom1();
        afx_msg void OnRadioCom2();
        afx_msg void OnDial();
        afx_msg void OnSend();
        DECLARE_EVENTSINK_MAP()
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
    };

    //{{AFX_INSERT_LOCATION}}
    // Microsoft Visual C++ will insert additional declarations immediately before the previous line.

    #endif
    // !defined(AFX_MODEMCOMMDLG_H__548C9AA6_8B4C_43B0_9505_BC2A41328CB3__INCLUDED_)
```

■ ModemCommDlg 应用文件

```
    // ModemCommDlg.cpp : implementation file
    //

    #include "stdafx.h"
    #include "ModemComm.h"
    #include "ModemCommDlg.h"

    #ifdef _DEBUG
```

```
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAboutDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);   // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    //{{AFX_MSG(CAboutDlg)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
        // No message handlers
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CModemCommDlg dialog

CModemCommDlg::CModemCommDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CModemCommDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CModemCommDlg)
    m_ReceiveData = _T("");
    m_SendData = _T("");
    m_TelphoneNo = _T("");
```

```
    m_NumCom = -1;
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CModemCommDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CModemCommDlg)
    DDX_Control(pDX, IDC_BUTTON_DIAL, m_ctrlDial);
    DDX_Text(pDX, IDC_EDIT_RECEIVE, m_ReceiveData);
    DDX_Text(pDX, IDC_EDIT_SEND, m_SendData);
    DDX_Text(pDX, IDC_EDIT_TELPHONENO, m_TelphoneNo);
    DDX_Radio(pDX, IDC_RADIO_COM1, m_NumCom);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CModemCommDlg, CDialog)
    //{{AFX_MSG_MAP(CModemCommDlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_RADIO_COM1, OnRadioCom1)
    ON_BN_CLICKED(IDC_RADIO_COM2, OnRadioCom2)
    ON_BN_CLICKED(IDC_BUTTON_DIAL, OnDial)
    ON_BN_CLICKED(IDC_BUTTON_SEND, OnSend)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CModemCommDlg message handlers

BOOL CModemCommDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Add "About..." menu item to system menu.

    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);

    CMenu* pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
        }
    }

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);          // Set big icon
    SetIcon(m_hIcon, FALSE);         // Set small icon
```

```
    // TODO: Add extra initialization here
/*在对话框初始化函数中选择 COM1 串口，并打开该串口*/
    DWORD style=WS_VISIBLE|WS_CHILD;
    if (!m_ctrlComm.Create(NULL,style,CRect(0,0,0,0),this,IDC_MSCOMM1))
    {
        TRACE0("Failed to create OLE Communications Control\n");
       return -1; //fail to create
    }

    // TODO: Add extra initialization here
    m_ctrlComm.SetCommPort(1); //选择 COM1
    m_ctrlComm.SetInBufferSize(1024); //设置输入缓冲区的大小，Bytes
    m_ctrlComm.SetOutBufferSize(512); //设置输出缓冲区的大小，Bytes

    if(!m_ctrlComm.GetPortOpen()) //打开串口
    {
        m_ctrlComm.SetPortOpen(TRUE);
        SendString("ATS0=1\n");//Modem 自动等待连接
    }
    else
        AfxMessageBox("串口 1 已被占用，请选择其他串口");
    return TRUE; // return TRUE unless you set the focus to a control
}

void CModemCommDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFF0) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {
        CDialog::OnSysCommand(nID, lParam);
    }
}


void CModemCommDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Draw the icon
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CDialog::OnPaint();
```

```
    }
}

// The system calls this to obtain the cursor to display while the user drags
//  the minimized window.
HCURSOR CModemCommDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}

BEGIN_EVENTSINK_MAP(CModemCommDlg, CDialog)
   //{{AFX_EVENTSINK_MAP(CModemCommDlg)
    ON_EVENT(CModemCommDlg, IDC_MSCOMM1, 1 /* OnComm */, OnComm, VTS_NONE)
    //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()

void CModemCommDlg::OnComm() //消息映射函数
{
    // TODO: Add your control notification handler code here
    VARIANT vResponse;
    char *str;
    char *str1;
   int k, nEvent, i;

   nEvent = m_ctrlComm.GetCommEvent();

   switch(nEvent)
    {
    case 2:  //收到大于RIThresshold个字符
         k = m_ctrlComm.GetInBufferCount(); //接收到的字符数目

         if(k > 0)
         {
              vResponse=m_ctrlComm.GetInput(); //read
              //对数据进行其他处理
         str = (char*)(unsigned char*) vResponse.parray->pvData;
         }
         // 接收到字符，MSComm控件发送事件
         i = 0;
         str1 = str;
         while (i < k)
         {
              i++;
              str1++;
         }
         *str1 = '\0';
         m_ReceiveData += (const char *)str;
         //清除字符串中的不必要字符
         break;
    case 3:  //CTS线状态发生了变化
         break;
    case 4:  //DSR线状态发生了变化
         break;
    case 5:  //CD线状态发生了变化
         break;
    case 6:  //Ring Indicator发生变化
         break;
   }

    UpdateData(FALSE);
```

```
}

void CModemCommDlg::OnRadioCom1()
{
    // TODO: Add your control notification handler code here
    OpenComm(1);
}

void CModemCommDlg::OnRadioCom2()
{
    // TODO: Add your control notification handler code here
    OpenComm(2);
}

void CModemCommDlg::OnDial()//鼠标单击“拨号”按钮事件的响应处理函数
{
    // TODO: Add your control notification handler code here
    CString strTemp;
    int Count = m_TelphoneNo.GetLength();
    if(!bOpen)
    {
        if(Count < 7)
            AfxMessageBox("电话号码位数错误");
        else
        {
            bOpen = TRUE;

            //向MODEM发送指令
            strTemp = "ATDT" + m_TelphoneNo + "\n";
            SendString(strTemp);
            m_ctrlDial.SetWindowText("挂断");
        }
    }
    else
    {
        SendString("ATH0");
        bOpen = FALSE;
        m_ctrlDial.SetWindowText("拨号");
    }
}

void CModemCommDlg::OnSend() //“发送”按钮的响应函数
{
    // TODO: Add your control notification handler code here
    CString strTemp = m_ReceiveData;
    SendString(strTemp);
}

void CModemCommDlg::SendString(CString m_strSend)//字符发送函数
{
    char TxData[100];
    int Count = m_SendData.GetLength();

    for(int i = 0; i < Count; i++)
        TxData[i] = m_SendData.GetAt(i);

    CByteArray array;
    array.RemoveAll();
    array.SetSize(Count);
```

```
        for(i = 0; i < Count; i++)
            array.SetAt(i, TxData[i]);
        m_ctrlComm.SetOutput(COleVariant(array));
    }

    void CModemCommDlg::OpenComm(int number)//选择并打开串口
    {
        m_ctrlComm.SetCommPort(number); //选择 COM

        if(!m_ctrlComm.GetPortOpen()) //打开串口
        {
            m_ctrlComm.SetPortOpen(TRUE);//MODEM 自动等待连接
            SendString("ATS0=1\n");

            bOpen = FALSE;
            m_ctrlDial.SetWindowText("拨号");
        }
        else
            AfxMessageBox("串口 1 已被占用，请选择其他串口");
    }
```

3.5.3　应用

将两部 MODEM 分别接到两台计算机的串口，然后连接到公用电话网上，从而使两台计算机建立联系。把上面编制的程序编译、运行，生成可执行程序文件。在两台计算机上分别执行。其中一台计算机作为拨出端，而另一台计算机则作为接收端，这样就可以在两台计算机之间进行数据传送。

选择其中一台计算机为拨出端，在呼叫号码下的编辑框中输入电话号码后，单击“拨号”按钮，程序开始拨号，图 3.5.5 即为拨号状态。

图 3.5.5　MODEM 拨号状态

另一台计算机即为接收端，在线路畅通的条件下，接收端与拨出端连接成功时的程序画面分别如图 3.5.6、图 3.5.7 所示。

图 3.5.6　接收端连接成功画面

图 3.5.7　拨出端连接成功画面

实际上，由于编制的运行程序既可以发送也可以接收，因此，两台计算机既可以是接收端，也可以是拨号端。在发送字符下的编辑框中输入字符，并单击“发送”按钮，就能在另一台计算机上的接收区中收到发送的字符。

第 4 章　Windows API 串口编程

[内容提要]

本章介绍 Windows API 串口编程的相关知识和实例，内容如下：

- Win16 和 Win32 API 函数串口通信编程的区别；
- API 串口编程中用到的结构，如 DCB、COMMTIMEOUTS、OVERLAPPED 等及相关概念；
- Windows API 串行通信函数介绍，以方便查询；
- Win32 API 串口通信编程的一般流程，并简要介绍了几个异于常用方法的特殊实例，如查询方式、同步方式的串口控制；
- CSerialPort 类中的 API 函数编程应用剖析，进一步加强对这个比较好用的串口编程工具类的了解，同时也有利于读者对其进行改进；
- Win32 API 串口编程的一个比较完整的实例：TTY（虚拟终端），列出了详细的编程步骤。

4.1　Windows API 串口编程概述

Windows API (Windows Application Programming Interface，Windows 应用程序编程接口)，是所有 Windows 应用程序的根本之所在。简单地说，API 就是一系列的例程，应用程序通过调用这些例程来请求操作系统完成一些低级服务。在 Windows 这样的图形用户界面中，应用程序的窗口、图标、菜单和对话框等就是由 API 来管理和维护的。

Windows API 具有两种基本类型：Win16 API 和 Win32 API。两者在很多方面非常相像，但是，Win32 API 除了几乎包括了 Win16 API 中的所有内容以外，还包括很多的其他内容。Windows API 依靠三个主要的核心组件提供 Windows 的大部分函数，在 Win16 和 Win32 中，它们具有不同的名称，如表 4-1-1 所示。

表 4-1-1　Win16 和 Win32 的核心组件

Win16 API	Win32 API	说明
USER.EXE	USER32.DLL	负责窗口的管理，包括消息、菜单、光标、通信、计时器和其他控制窗口的显示
GDI.EXE	GDI32.DLL	提供图形设备接口，管理用户界面和图形绘制，包括 Windows 元文件、位图、设备描述表和字体等
KRNL386.EXE	KERNEL32.DLL	处理存储器低层功能、任务和资源管理等 Windows 核心服务

虽然 Win16 API 组件带有.EXE 的扩展名，但是，它们事实上都是动态链接库(.DLL)，不能单独运行。其他一些非核心的 Windows API 由其他组件所提供的 DLL 来实现，这些组件

包括通用对话框、打印、文件压缩、版本控制以及多媒体支持等。

Windows 9x/NT/XP 等操作系统封装了 Windows 串口机制，其串行通信设备驱动程序是 comm.drv，通过调用 API 函数编程来控制驱动程序，对硬件进行操作。

这里，我们先简单回顾一下 Windows 3.x 操作系统下的 16 位 API 串行通信函数。

16 位串口应用程序中，使用的是 16 位的 Windows API 通信函数:

1. 打开与关闭串口

OpenComm() 打开串口资源，并指定输入、输出缓冲区的大小（以字节计）。

CloseComm() 关闭串口。

例：

```
int idComDev;
idComDev = OpenComm("COM1", 1024, 128);
CloseComm(idComDev);
```

2. 初始化串口，设置串口参数

BuildCommDCB() 、setCommState()填写设备控制块 DCB，然后对已打开的串口进行参数配置。例：

```
DCB dcb;
BuildCommDCB("COM1:2400,n,8,1", &dcb);
SetCommState(&dcb);
```

3. 读写串口，收发数据

ReadComm 、WriteComm()对串口进行读写操作，即数据的接收和发送。例：

```
char *m_pRecieve; int count;
ReadComm(idComDev,m_pRecieve,count);
Char wr[30];
int count2;
WriteComm(idComDev,wr,count2);
```

16 位下的串口通信程序最大的特点就在于：串口等外部设备的操作有自己特有的 API 函数。而 32 位程序则把串口操作（以及并口等）和文件操作统一起来了，以上 16 位程序中应用的函数也不再使用。

Windows API 函数中涉及到通信编程的有 20 多个，这些 API 函数在实际编程中并不都是必须用的，例如，要检测当前串口的设置可以只用 SetCommState，而不用 GetCommProperties 和 GetCommConfig，虽然它们返回的信息可能更多。同样，如果有些值想用缺省的，比如缓冲区的大小和超时的时间等，那么 SetupComm 和 BuildCommDCBAndTimeouts、SetCommTimeouts 也可以不用，TransmitCommChar 是马上在发送序列中优先插入发送一个字符用的，平时也很少用到，相关 API 函数将在本章中介绍。

4.2　API 串口编程中用到的结构及相关概念说明

Win32 操作系统中，在设置串口时有许多函数还会用到如设备控制 DCB 以及超时控制（COMMTIMEOUTS）等结构，还有通信错误、通信状态以及通信事件等信息。因此，我们有必要了解这些。

4.2.1　DCB（Device Control Block）结构

在打开通信设备句柄后，常常需要对串口进行一些初始化工作。这需要通过一个 DCB 结构来进行。DCB 结构包含了诸如波特率、每个字符的数据位数、奇偶校验和停止位数等信息。在查询或配置串口的属性时，都要用 DCB 结构来作为缓冲区。

调用 GetCommState 函数可以获得串口的配置，该函数把当前配置填充到一个 DCB 结构中。一般在用 CreateFile 打开串行口后，可以调用 GetCommState 函数来获取串行口的初始配置。要修改串口的配置，应该先修改 DCB 结构，然后再调用 SetCommState 函数用指定的 DCB 结构来设置串行口。

DCB 结构的声明：

```
typedef struct _DCB { // dcb
   DWORD DCBlength;              // DCB 块大小
   DWORD BaudRate;               // 当前波特率
   DWORD fBinary: 1;             // 二进制模式,不检测 EOF
   DWORD fParity: 1;             // 允许奇偶校验
   DWORD fOutxCtsFlow:1;         // CTS 输出流控制
   DWORD fOutxDsrFlow:1;         // DSR 输出流控制
   DWORD fDtrControl:2;          // DTR 流控制类型
   DWORD fDsrSensitivity:1;      //对 DSR 信号线是否敏感
   DWORD fTXContinueOnXoff:1;    // XOFF continues Tx
   DWORD fOutX: 1;               // XON/XOFF 输出流控制
   DWORD fInX: 1;                // XON/XOFF 输入流控制
   DWORD fErrorChar: 1;          // 允许错误替换
   DWORD fNull: 1;               // 允许剥离去掉 NULL 字符
   DWORD fRtsControl:2;          // RTS 流控制
   DWORD fAbortOnError:1;        // 有错误时终止读/写操作
   DWORD fDummy2:17;             // 保留
   WORD wReserved;               // 当前不用
   WORD XonLim;                  // XON 发送字符之前，缓冲区中允许接收的最小字节数
   WORD XoffLim;                 // XON 发送字符之前，缓冲区中允许的最小可用字节数
   BYTE ByteSize;                // 数据位数（字节表示 4～8 位）
   BYTE Parity;                  // 奇偶校验 0～4：表示：不校验,奇校验,偶校验,标号、空格
   BYTE StopBits;                //停止位 0,1,2 = 1, 1.5, 2
   char XonChar;                 //发送和接收的 XON 字符
   char XoffChar;                //发送和接收的 XOFF 字符
   char ErrorChar;               // 用来替换接收到的奇偶校验发生错误的字符
   char EofChar;                 // 输入数据结束字符
   char EvtChar;                 //接收到的事件字符
   WORD wReserved1;              //保留位（不用）
} DCB;
```

DCB 结构参数说明

DCBlength

以字节为单位设置 DCB 结构长度大小。

BaudRate:

设置波特率（串口传输速率），可以以实际数据表示，如 9600 等，也可以是 CBR_110 CBR_300 CBR_600 CBR_1200 CBR_2400 CBR_4800 CBR_9600 CBR_14400 CBR_19200 CBR_38400 CBR_56000 CBR_57600 CBR_115200 CBR_128000 CBR_256000

fBinary

是否允许二进制传输。Win32 API 不支持非二进制方式传输，因此，该参数必须设置为 TRUE，否则不能正常工作。

fParity

是否允许奇偶校验。如果设置为 TRUE，则进行奇偶校验并报告错误信息。

fOutxCtsFlow

指定 CTS (clear-to-send) 信号是否检测输出流控制。该值为 TRUE 且 CTS 为 off 时，发送被暂停，直至 CTS 被重新置为 on。

fOutxDsrFlow

指定 DSR (data-set-ready)信号是否检测输出流控制。该值为 TRUE 且 DSR 为 off 时，发送被暂停，直至 DSR 被重新置为 on。

fDtrControl

设置 DTR (data-terminal-ready) 流控制。该属性可以设置为表 4-2-1 中的任意值。

表 4-2-1 DTR 流控制值描述

值	描述
DTR_CONTROL_DISABLE	当串口被打开时，禁止 DTR 线，并保持禁止状态
DTR_CONTROL_ENABLE	当串口被打开时，允许 DTR 线，并保持允许状态
DTR_CONTROL_HANDSHAKE	允许 DTR 握手。如果 handshaking（握手）被打开，不允许用 EscapeCommFunction 函数调整线路状态

fDsrSensitivity

设置通信驱动程序对 DTR 信号是否敏感，如果该值为 TRUE，且 DSR 信号线为低（OFF）时，接收的任何字节将被忽略。

fTXContinueOnXoff

设置当接收缓冲区已满，且驱动程序已发送出 **XoffChar** 字符时，发送是否停止。若该值为 TRUE，在接收缓冲区接收到了缓冲区已满的字节 **XoffLim**，且驱动程序已经发送出 **XoffChar** 字符去停止接收字节后，发送继续进行；若该值为 FALSE，当接收缓冲区接收到代表缓冲区已空的字节数 **XonLim**，且驱动程序已经发送出恢复发送的 **XonChar** 字符后，发送可以继续进行。

fOutX

设置在发送时是否应用 XON/XOFF 流控制。当该值为 TRUE 时，收到 XoffChar 字符后发送停止，只有在再接收到 XonChar 字符后才能继续发送。

fInX

设置在接收时是否应用 XON/XOFF 流控制。当该值为 TRUE 时，接收缓冲区收到代表接收缓冲区已满的 **XoffLim** 字节数后，**XoffChar** 被发送出去；接收缓冲区接收到

代表接收缓冲区已空的 **XonLim** 字节数后，**XonChar** 字符被发送出去。

fErrorChar

设置是否用 **ErrorChar** 成员指定的错误替换字符来替换当奇偶校验出错时的接收字符，该项选择只有在 **fParity** 为 TRUE 时有效。

fNull

设置是否剥离丢掉接收到的 NULL 字符（ASCII 值为 0）。该值为 TRUE 时，NULL 字符被丢掉，否则保留。

fRtsControl

设置 RTS (request-to-send)流控制。缺省值为 RTS_CONTROL_HANDSHAKE（值为 0）。该成员可以设置为表 4-2-2 中的任意值。

表 4-2-2　RTS 流控制

值	描述
RTS_CONTROL_DISABLE	当串口被打开时，禁止 RTS 线，并保持禁止状态
RTS_CONTROL_ENABLE	当串口被打开时，允许 RTS 线，并保持允许状态
RTS_CONTROL_HANDSHAKE	允许 RTS 握手。当接收缓冲区中字节数小于缓冲区最大值的 1/2 时，将 RTS 线置为高（ON）；当接收缓冲区中字节数大于缓冲区最大值的 3/4 时，将 RTS 线置为低（OFF）；如果允许握手，则不允许调用 EscapeCommFunction 函数调整线路
RTS_CONTROL_TOGGLE	当还有等待发送的字节时，将 RTS 线置高（ON）；全部发送完后，将 RTS 线置低（OFF）

fAbortOnError

当有错误产生，设置是否终止读写操作。当该值为 TRUE 时，驱动程序将终止读写操作，只有当应用程序调用ClearCommError函数处理错误后，通信才能恢复。

fDummy2

保留位，不使用。

wReserved

不使用，必须设置为 0。

XonLim

设置在 XON 字符发送之前输入（接收）缓冲区中允许的最小字节数。

XoffLim

设置在发送 XOFF 字符之前，（接收）缓冲区中允许的最大字节数。最大字节数为接收缓冲区大小减去本值（以字节数）。

ByteSize

设置数据位，（字节表示 4～8 位）。

Parity

设置奇偶校验方式。可设置为如表 4-2-3 中的值。

表 4-2-3　奇偶校验方式

值	描述
EVENPARITY	Even　偶校验
MARKPARITY	Mark　标号校验

续表

值	描述
NOPARITY	No parity　无校验
ODDPARITY	Odd　奇校验
SPACEPARITY	Space　空格校验

StopBits

设置停止位。可设置为表 4-2-4 中的值。

表 4-2-4　停止位设置方式

值	描述
ONESTOPBIT	1 个停止位
ONE5STOPBITS	1.5 个停止位
TWOSTOPBITS	2 个停止位

XonChar

设置发送和接收的 XON 字符。

XoffChar

设置发送和接收的 XOFF 字符。

ErrorChar

设置用来替换接收到的奇偶校验发生错误的字符。

EofChar

设置用来表示数据结束的字符。

EvtChar

设置事件字符。当接收到此字符时，会产生一个事件。

wReserved1

保留位，不用。

注意 当 DCB 用来控制 8250 时，对 ByteSize 和 StopBits 的设置有如下限制：

- 数据位 ByteSize 必须为 5 ~ 8 位
- 5 个数据位配两个停止位，或者 6、7 和 8 个数据位配 1.5 个停止位都是不正确的设置。

4.2.2　超时设置 COMMTIMEOUTS 结构

在用 ReadFile 和 WriteFile 读写串行口时，需要考虑超时问题。如果在指定的时间内没有读出或写入指定数量的字节，那么 ReadFile 或 WriteFile 的操作就会结束。要查询当前的超时设置应调用 GetCommTimeouts 函数，该函数会填充一个 COMMTIMEOUTS 结构。调用 SetCommTimeouts 函数可以用某一个 COMMTIMEOUTS 结构的内容来设置超时。

COMMTIMEOUTS 结构的定义为：

```
typedef struct _COMMTIMEOUTS {
DWORD ReadIntervalTimeout;            // 读间隔超时
DWORD ReadTotalTimeoutMultiplier;     // 读时间系数
DWORD ReadTotalTimeoutConstant;       // 读时间常量
DWORD WriteTotalTimeoutMultiplier;    // 写时间系数
DWORD WriteTotalTimeoutConstant;      // 写时间常量
```

```
} COMMTIMEOUTS, *LPCOMMTIMEOUTS;
```

参数说明

ReadIntervalTimeout 读时间间隔超时

以毫秒(ms)为单位设置通信线路上两个字符到达之间最大时间间隔。在 **ReadFile** 操作期间，从接收到第一个字符开始计时。如果任意两个字符到达之间的时间间隔超过这个最大值，则 **ReadFile** 操作完成，返回缓冲数据。如果该值被置为 **0**，则不使用时间间隔超时。

ReadTotalTimeoutMultiplier 读时间系数

以毫秒(ms)为单位设置一个用来计算读操作总超时时间的时间系数。该时间系数乘要求读出的字节数。

ReadTotalTimeoutConstant 读时间常量

以毫秒(ms)为单位设置一个用来计算读操作总超时时间的时间常量。

WriteTotalTimeoutMultiplier 写时间系数

以毫秒(ms)为单位设置一个用来计算写操作总超时时间的时间系数。该时间系数乘要求写入的字节数。

WriteTotalTimeoutConstant 写时间常量

以毫秒(ms)为单位设置一个用来计算读操作总超时时间的时间常量。

总超时时间 = 读 / 写时间系数 × 要求读 / 写的字节数 + 读 / 写时间常量

即：

读总超时时间＝读时间系数 × 要求读的字节数 + 读时间常量

读总超时时间＝写时间系数 × 要求写的字节数 + 写时间常量

例如，如果要读入 10 个字节数据，那么读操作的总超时的计算公式为：

读总超时＝**ReadTotalTimeoutMultiplier** × 10 + **ReadTotalTimeoutConstant**

同样，要写入 10 个字节数据，那么写操作的总超时的计算公式为：

写总超时＝**WriteTotalTimeoutMultiplier** × 10 + **WriteTotalTimeoutConstant**

超时有间隔超时和总超时两种类型。间隔超时是指在接收时两个字符之间的最大时延，用于从串口读取数据，当接收一个字节时，通信驱动程序启动一个内部定时器开始计时，在下一个字节到达之前，如果定时时间超过了间隔超时时间，读操作就会被放弃。总超时是指读写操作总共花费的最大时间。写操作只支持总超时，而读操作对两种超时均支持。用 COMMTIMEOUTS 结构可以规定读/写操作的超时。

如果应用程序将 **ReadIntervalTimeout** 和 **ReadTotalTimeoutMultiplier** 设置成为 MAXDWORD，并且将 **ReadTotalTimeoutConstant** 设置为大于 0 且小于 MAXDWORD，则调用 **ReadFile** 函数时，会有以下情况：

- 如果接收缓冲区中有字符存在，则 **ReadFile** 函数立即返回这些字符；
- 如果接收缓冲区中没有任何字符，则 **ReadFile** 函数会等待直至一个字符到达，然后立即返回；
- 若在 **ReadTotalTimeoutConstant** 设定的时间内没有任何字符到达，则 **ReadFile** 超时返回。

可以看出，间隔超时和总超时的设置是不相关的，这可以方便通信程序灵活地设置各种超时。

如果所有写超时参数均为 0，那么就不使用写超时。如果 ReadIntervalTimeout 为 0，那么就不使用读间隔超时，如果 ReadTotalTimeoutMultiplier 和 ReadTotalTimeoutConstant 都为 0，则不使用读总超时。如果读间隔超时被设置成 MAXDWORD，并且 ReadTotalTimeoutMultiplier 和 ReadTotalTimeoutConstant 都为 0，那么在读一次输入缓冲区中的内容后，读操作就立即完成，而不管是否读入了要求的字符。

在用重叠方式读写串行口时，虽然 ReadFile 函数和 WriteFile 函数在完成操作以前就可能返回，但超时仍然是起作用的。在这种情况下，超时规定的是操作的完成时间，而不是 ReadFile 函数和 WriteFile 函数的返回时间。

4.2.3　OVERLAPPED 异步 I/O 重叠结构

在用 ReadFile 和 WriteFile 读写串口数据时，既可以同步执行，也可以重叠（异步）执行。在同步执行时，函数直到操作完成后才返回。这意味着在同步执行时线程会被阻塞，从而导致效率下降。在重叠执行时，即使操作还未完成，调用的函数也会立即返回。费时的 I/O 操作在后台进行，这样线程就可以干别的事情。在异步设置时，会用到 OVERLAPPED 异步 I/O 重叠结构。

OVERLAPPED 结构包含了异步输入输出（I/O）操作的相关信息，其定义如下：

```
typedef struct _OVERLAPPED {
   DWORD  Internal; //保留给系统内部使用
   DWORD  InternalHigh; //保留给系统内部使用
   DWORD  Offset;       //指定开始传输数据的位置
   DWORD  OffsetHigh;   //定义 Offset 值的高字节
   HANDLE hEvent;       //当数据传输完成时设置的事件句柄。
} OVERLAPPED;
```

在串口通信编程过程中，与重叠 I/O 操作关系最密切的是串口的读写操作，涉及的两个函数是 ReadFile 和 WriteFile。当 ReadFile 和 WriteFile 返回 FALSE 时，不一定就是操作失败，线程应该调用 GetLastError 函数分析返回的结果。例如，在重叠操作时，如果操作还未完成函数就返回，那么函数就返回 FALSE，而且 GetLastError 函数返回 ERROR_IO_PENDING。

在使用重叠 I/O 时，线程需要创建 OVERLAPPED 结构以供读写函数使用。OVERLAPPED 结构最重要的成员是 hEvent。hEvent 是一个事件对象句柄，线程应该用 CreateEvent 函数为 hEvent 成员创建一个手工重置事件，hEvent 成员将作为线程的同步对象使用。如果读写函数未完成操作就返回，那么就把 hEvent 成员设置成无信号的。操作完成后（包括超时），hEvent 会变成有信号的。

如果 GetLastError 函数返回 ERROR_IO_PENDING，则说明重叠操作还未完成，线程可以等待操作完成。有两种等待办法：一种办法是用像 WaitForSingleObject 这样的等待函数来等待 OVERLAPPED 结构的 hEvent 成员，可以规定等待的时间，在等待函数返回后，调用 GetOverlappedResult。另一种办法是调用 GetOverlappedResult 函数等待，如果指定该函数的 bWait 参数为 TRUE，那么该函数将等待 OVERLAPPED 结构的 hEvent 事件。GetOverlappedResult 可以返回一个 OVERLAPPED 结构来报告包括实际传输字节在内的重叠操作结果。

如果规定了读/写操作的超时，那么当超过规定时间后，hEvent 成员会变成有信号的。因此，在超时发生后，WaitForSingleObject 和 GetOverlappedResult 都会结束等待。WaitForSingleObject 的 dwMilliseconds 参数会规定一个等待超时，该函数实际等待的时间是两个超时的最小值。注意 GetOverlappedResult 不能设置等待的时限，因此，如果 hEvent 成员无信号，则该函数将一直等待下去。

从以上可以看出，异步 I/O 操作有两种方法来获取结果：

- 利用 GetOverlappedResult 函数。
- 利用 WaitForSingleObject 等事件处理函数。
- 以上两种结合起来使用。

相关函数的介绍查看第 4.3 节。

4.2.4 通信错误与通信设备状态

在串口通信过程中，如果奇偶校验、终端等错误，I/O 操作将会终止，此时，如果要进一步进行 I/O 操作，则必须调用 **ClearCommError** 函数来清除错误标志。表 4-2-5 是错误类型说明，**ClearCommError** 函数将在第 4.3 节介绍。

表 4-2-5　通信错误类型

值	描述
CE_BREAK	硬件检测到一个中断条件
CE_DNS	**Win9x**：一个并行设备没有被选择
CE_FRAME	硬件检测到一个帧错误
CE_IOE	发生一个 I/O 操作错误
CE_MODE	设置通信模式出错，或者句柄无效
CE_OOP	**Win9x**：并行设备缺纸
CE_OVERRUN	超速错误，下一个字符丢失
CE_PTO	**Win9x**：并行设备发生超时错误
CE_RXOVER	一个接收缓冲区溢出错误。可能是接收缓冲区已满，也可能是接收到文件结束字符(EOF)后还收到了字符
CE_RXPARITY	硬件检测到一个奇偶校验错误
CE_TXFULL	发送缓冲区满，应用程序无法再发送数据

通信设备状态信息是由一个 COMSTAT 结构来存放的，该结构内容也由 **ClearCommError** 函数来填写。

COMSTAT 结构声明如下：

```
typedef struct _COMSTAT {
   DWORD fCtsHold : 1;  // Tx waiting for CTS signal
   DWORD fDsrHold : 1;  // Tx waiting for DSR signal
   DWORD fRlsdHold : 1; // Tx waiting for RLSD signal
   DWORD fXoffHold : 1; // Tx waiting, XOFF char received
   DWORD fXoffSent : 1; // Tx waiting, XOFF char sent
   DWORD fEof : 1;      // EOF character sent
   DWORD fTxim : 1;     // character waiting for Tx
   DWORD fReserved : 25; // reserved
   DWORD cbInQue;       // bytes in input buffer
   DWORD cbOutQue;      // bytes in output buffer
} COMSTAT, *LPCOMSTAT;
```

参数说明

fCtsHold

指定发送时是否等待 CTS (clear-to-send) 信号。若该值为 TRUE, 则发送等待。

fDsrHold

指定发送时是否等待 DSR (data-set-ready)信号。若该值为 TRUE, 则发送等待。

fRlsdHold

指定发送时是否等待 RLSD (receive-line-signal-detect)信号。若该值为 TRUE, 则发送等待。

fXoffHold

指定当收到 XOFF 字符后发送时是否等待。若该值为 TRUE, 则发送等待。

fXoffSent

指定发送完 XOFF 字符后发送是否等待。若该值为 TRUE, 则发送等待。当把 XOFF 字符发送给一系统，而该系统把下一个字符当做 XON 字符（不管该字符的实际值），此时发送会停止。

fEof

说明文件是否收到结束字符 end-of-file (EOF) 。若该值为 TRUE, 则已收到 EOF 字符。

fTxim

若该值为 TRUE, 则表明在发送队列中有一个由**TransmitCommChar**函数设置的优先发送字符正等待发送，通信设备将把该字符优先发送。

fReserved

保留不用。

cbInQue

说明串行设备接收到的字节数，但这些字节并不已经全部由 ReadFile 函数读出。

cbOutQue

说明设备发送缓冲区还没有发送的用户数据字节数。如果设置为非重叠 I/O 写操作，则该值为 0。

4.2.5 串行通信事件

通信事件是 Windows 进程中监视通信设备中发生的事件，这样应用程序可以不检测端口状态就可以知道某些条件何时发生，这在编程时是很实用的。应用程序可以随时捕捉到通信事件，再通过事件消息处理机制，这样就可以不需要为接收数据而一直查询端口状态，从而节省 CPU 时间。

在 Windows 95/NT 中，WM_COMMNOTIFY 消息已经取消，在串行口产生一个通信事件时，程序并不会收到通知消息。线程需要调用 WaitCommEvent 函数来监视发生在串行口中的各种事件，该函数的第二个参数返回一个事件屏蔽变量，用来指示事件的类型。线程可以用 SetCommMask 建立事件屏蔽（掩码）以指定要监视的事件。调用 GetCommMask 可以查询串行口当前的事件屏蔽（掩码）。

表 4-2-6 是常用串行通信事件。

表 4-2-6　通信事件

值	描述
EV_BREAK	检测到一个输入中断
EV_CTS	CTS 信号发生变化
EV_DSR	DSR 信号发生变化
EV_ERR	发生行状态错误
EV_RING	检测到振铃信号
EV_RLSD	RLSD(CD)信号发生变化
EV_RXCHAR	输入缓冲区接收到新字符
EV_RXFLAG	输入缓冲区收到事件字符
EV_TXEMPTY	发送缓冲区为空

4.3　Windows API 串行通信函数

本节详尽地介绍 Windows 9X/NT/XP 操作系统下的 32 位 API 通信函数。读者在以后的编程实践中用到相关函数后，可以仔细了解各个函数的功能及应用方法。在以下说明中，我们按照 API 编程中使用串口的基本顺序来解释各个函数。

为了查询方便，在这里和本节的最后均列出各个函数的说明顺序号：

（1）**CreateFile** 打开串口函数
（2）**SetupComm** 缓冲区分配函数
（3）**GetCommState** 获取串口当前配置函数
（4）**SetCommState** 配置串口函数
（5）**GetCommProperties** 获取串口属性函数
（6）**BuildCommDCB**　DCB 填充函数
（7）**BuildCommDCBAndTimeouts** DCB 和 Timeouts 填充函数
（8）**GetCommTimeouts** 超时获取函数
（9）**SetCommTimeouts** 超时设置函数
（10）**ReadFile / ReadFileEx** 读串口函数
（11）**WriteFile / WriteFileEx** 写串口函数
（12）**ClearCommError** 清除错误标志函数
（13）**PurgeComm** 终止读写 / 清空缓冲区函数
（14）**FlushFileBuffer** 清空缓冲区函数
（15）**GetOverlappedResult** 异步 I/O 操作结果获取函数
（16）**WaitForSingleObject** 异步 I/O 操作事件获取函数
（17）**SetCommMask** 通信事件设置函数
（18）**GetCommMask** 当前通信事件获取函数
（19）**WaitCommEvent** 通信事件监测函数
（20）**CreateEvent** 事件创建函数
（21）**EscapeCommFunction** 握手信号设置函数
（22）**SetCommBreak** 通信设备挂起函数
（23）**ClearCommBreak** 通信设备恢复函数

下面详细介绍每个函数。

（1）CreateFile 打开串口函数

函数声明：

```
HANDLE CreateFile (
LPCTSTR lpFileName,                          // 文件名 "COM1"、"COM2" 等
DWORD dwDesiredAccess,                       // 访问模式 读、写或同时读写
DWORD dwShareMode,                           // 共享模式，常为 0
LPSECURITY_ATTRIBUTES lpSecurityAttributes,  // 通常为 NULL
DWORD dwCreationDistribution,                // 创建方式：OPEN_EXISTING
DWORD dwFlagsAndAttributes,                  // 文件属性和标志
HANDLE hTemplateFile                         // 临时文件的句柄，常为 NULL
);
```

功能：

在 Win32 操作系统中，文件是一个广泛的概念，包含文件、通信设备、命名管道、邮件槽、磁盘、控制台等，这些都是用 CreateFile 函数来打开或创建的。

返回值：

CreateFile 函数调用成功后，返回值是一个文件句柄（long），在 Windows API 中调用任何通信例程时都使用该值，它与 MSComm 控件中的 CommID 属性值是等同的；若不成功，则返回错误信息 INVALID_HANDLE_VALUE，同时设置GetLastError。即使函数成功，但若文件存在，且指定了 CREATE_ALWAYS 或 OPEN_ALWAYS，GetLastError 也会设为 ERROR_ALREADY_EXISTS。

参数表说明：

表 4-3-1 是 CreateFile 参数说明。

表 4-3-1 CreateFile 函数参数说明

参数	类型及说明
lpFileName	类型为 String，设置要打开的文件的名字，即串口逻辑名，如"COM1"、"COM2"分别表示串口 1、串口 2 特别注意事项：当串口号大于 COM9（不含 9）时，在 C/C++语言中，串口逻辑名应写为"\\.\COMxx"，如串口 10 为"\\.\COM10"，其他编程语言中，写为"\\.\COMxx"
dwDesiredAccess	类型为 long，设置串口访问模式 GENERIC_READ 表示允许对设备进行读访问，即读串口； GENERIC_WRITE 表示允许对设备进行写访问，即写串口； GENERIC_READ| GENERIC_WRITE 表示可同时读写，即读写串口； 零，表示只允许获取与一个设备有关的信息
dwShareMode	类型为 long，设置串口的共享属性 0 表示不共享； FILE_SHARE_READ 和/或 FILE_SHARE_WRITE 表示允许对文件进行共享访问 对于串口，不能设置成共享，因此必须设置为 0，如果当前应用程序调用 CreateFile 函数来打开串口时，串口不存在或者被其他设备占用（已经打开），则会返回错误信息
lpSecurityAttributes	SECURITY_ATTRIBUTES，指向一个 SECURITY_ATTRIBUTES 结构的指针，定义了文件的安全特性（如果操作系统支持的话） 对于串口，设置为 NULL
dwCreationDisposition	类型为 long，设置创建方式，其值为下述常数之一： CREATE_NEW：创建文件，如文件存在则会出错 CREATE_ALWAYS：创建文件，会改写前一个文件 OPEN_EXISTING：文件必须已经存在，由设备提出要求 OPEN_ALWAYS：如文件不存在，则创建它 TRUNCATE_EXISTING：将现有文件缩短为零长度 对于串口，因其总是存在的，该值只能设置为 OPEN_EXISTING

续表

参数	类型及说明	
dwFlagsAndAttributes	类型为 long，设置文件的各种属性，可用一个或多个下述常数	
	FILE_ATTRIBUTE_ARCHIVE	标记归档属性
	FILE_ATTRIBUTE_COMPRESSED	将文件标记为已压缩，或者标记为文件在目录中的默认压缩方式
	FILE_ATTRIBUTE_NORMAL	默认属性
	FILE_ATTRIBUTE_HIDDEN	隐藏文件或目录
	FILE_ATTRIBUTE_READONLY	文件为只读
	FILE_ATTRIBUTE_SYSTEM	文件为系统文件
	FILE_FLAG_WRITE_THROUGH	操作系统不得推迟对文件的写操作
	FILE_FLAG_OVERLAPPED	允许对文件进行重叠操作
	FILE_FLAG_NO_BUFFERING	禁止对文件进行缓冲处理，文件只能写入磁盘卷的扇区块
	FILE_FLAG_RANDOM_ACCESS	针对随机访问对文件缓冲进行优化
	FILE_FLAG_SEQUENTIAL_SCAN	针对连续访问对文件缓冲进行优化
	FILE_FLAG_DELETE_ON_CLOSE	关闭了上一次打开的句柄后，将文件删除，特别适合临时文件
	也可在 Windows NT 下组合使用下述常数标记： SECURITY_ANONYMOUS SECURITY_IDENTIFICATION SECURITY_IMPERSONATION SECURITY_DELEGATION SECURITY_CONTEXT_TRACKING SECURITY_EFFECTIVE_ONLY 对于串口，惟一有意义的设置为 FILE_FLAG_OVERLAPPED，允许对文件进行重叠操作	
hTemplateFile	类型为 long，如果不为零，则指定一个文件句柄，新文件将从这个文件中复制扩展属性	

注意：在打开一个通信端口时，应该以独占方式打开，另外要指定 GENERIC_READ、GENERIC_WRITE、OPEN_EXISTING 和 FILE_ATTRIBUTE_NORMAL 等属性。如果要打开重叠 I/O，则应该指定 FILE_FLAG_OVERLAPPED 属性。

重叠（OVERLAPPED）I/O 操作，就是异步操作或非阻塞操作，即在执行一项操作时，若系统有别的操作请求，可以立即返回执行其他任务，这样程序就不会类似“死机”一样停在那里。而 NonOverLapped（同步）方式则正好相反，程序原因应该在于同步方式下如果有一个通信 API 函数在操作中，另一个会阻塞（等待）直到上一个操作完成，所以当读数据的线程停留在 WaitCommEvent 的时候，写操作 WriteFile 就停在“原地”等待。所以，如果程序没有特殊要求，则在调用 CreateFile 函数打开串口时，一般将操作方式设为使用 OverLapped（异步）方式。有关重叠（OVERLAPPED）I/O 操作的更多详细介绍，请参看第 4.2.3 节。

综上，这里给出一个打开串口 1（COM1）的程序段，一般的程序均可按此编写。

```
HANDLE hCom;
DWORD dwError;
hCom=CreateFile(
        "COM1", // 串口号
        GENERIC_READ | GENERIC_WRITE, // 允许读和写
        0, // 独占方式NULL,
        OPEN_EXISTING, //打开而不是创建
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, // 重叠方式
        NULL
```

```
        );

    if(hCom = = INVALID_HANDLE_VALUE)
    {
        wError=GetLastError();  //得到错误信息
        //. . . // 处理错误
    }
```

注意： 通过 CreateFile 函数得到了打开的串口句柄 hCom（HANDLE 类型），其他函数在操作该串口时，这个句柄就是串口的标志。

（2）SetupComm 缓冲区分配函数

函数声明：

```
BOOL SetupComm(
  HANDLE hFile,     // 通信设备（串口）句柄，通过 CreateFile 返回值得到
  DWORD dwInQueue,  // 输入缓冲区大小
  DWORD dwOutQueue  // 输出缓冲区大小
);
```

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时，可调用GetLastError函数得到错误信息。

说明：

串口打开后，就可以使用 SetupComm 为其设置缓冲区大小，若不设置，则系统自动将其他设置成缺省大小。设置时应避免缓冲区溢出，即比实际大小稍大一点。

（3）GetCommState 获取串口当前配置函数

函数声明：

```
BOOL GetCommState(
  HANDLE hFile,  // 通信设备（串口）句柄，通过 CreateFile 返回值得到
  LPDCB lpDCB    // 指向 DCB（device-control block）结构
);
```

DCB 结构说明查看第 4.2 节有关介绍，该结构中存储有关串口配置信息。

返回值：

如果函数调用成功，返回非 0 值；调用失败，返回 0，这时可调用**GetLastError**得到有关错误信息。

（4）SetCommState 配置串口函数

函数声明：

```
BOOL SetCommState (
  HANDLE hFile,  // 通信设备（串口）句柄，通过 CreateFile 返回值得到
  LPDCB lpDCB    // 指向 DCB（device-control block）结构
);
```

DCB 结构说明查看第 4.2 节有关介绍，该结构中存储有关串口配置信息。

返回值：

如果函数调用成功，返回非 0 值；调用失败，返回 0，这时可调用**GetLastError**得到有关错误信息。

说明：

SetCommState 函数通过 DCB 结构来定义串口配置，当前的串口配置由 GetCommState 函数得到。在设置过程中，有时只需要更改几个变量（不是全部变量）的值，因此，DCB 结构中其他值需要保持原值，以保证 DCB 中的其他成员变量值正确，这些原值就是由 GetCommState 得到的。

如果设置时出现 DCB 成员变量 **XonChar** 与 **XoffChar** 相同，则 SetCommState 设置失败。在设置 8250 时，数据位只能是 5～8 位。

（5）GetCommProperties 获取串口属性函数

函数声明：

```
BOOL GetCommProperties(
  HANDLE hFile,      // 通信设备（串口）句柄，通过CreateFile返回值得到
  LPCOMMPROP lpCommProp   // 指向COMMPROP结构
);
```

返回值：

如果函数调用成功，返回非 0 值；调用失败，返回 0，这时可调用**GetLastError**得到有关错误信息。

说明：

由该函数得到的串口（或其他通信设备，如并口等）信息可用于SetCommState、SetCommTimeouts和SetupComm三个函数去配置串口信息。

COMMPROP结构说明：

COMMPROP结构只涉及到本函数，因此在这里介绍。

COMMPROP结构声明：

```
typedef struct _COMMPROP {
   WORD  wPacketLength;          // 数据簿大小（字节）
   WORD  wPacketVersion;         // COMMPROP结构版本号
   DWORD dwServiceMask;          // 设置一位掩码
   DWORD dwReserved1;            // reserved 保留
   DWORD dwMaxTxQueue;           // 驱动程序发送缓冲区最大允许长度（字节）
   DWORD dwMaxRxQueue;           // 驱动程序接收缓冲区最大允许长度（字节）
   DWORD dwMaxBaud;              // 最大波特率，in bps
   DWORD dwProvSubType;          // 设置设备类型specific provider type
   DWORD dwProvCapabilities;     // 能支持的功能
   DWORD dwSettableParams;       // 可修改的通信参数
   DWORD dwSettableBaud;         // 可设置的波特率
   WORD  wSettableData;          // 可设置的数据位
   WORD  wSettableStopParity;    // 可设置的停止位/奇偶校验
   DWORD dwCurrentTxQueue;       // 当前发送缓冲区大小（字节）
   DWORD dwCurrentRxQueue;       // 当前接收缓冲区大小（字节）
   DWORD dwProvSpec1;            // 未定义
   DWORD dwProvSpec2;            // 未定义
   WCHAR wcProvChar[1];          // 未定义
} COMMPROP;
```

主要参数说明：

wPacketLength

不论要求的数据量大小，该参数指明整个结构数据簿大小。

wPacketVersion

说明COMMPROP结构版本，该结构随 Windows 操作系统不同有所区别。

dwServiceMask

设置一位用来说明通信设备实现了哪种功能的选择码（掩码）；对串口（含 MODEM），该变量通常设置为 SP_SERIALCOMM。

dwReserved1

保留不用。

dwMaxTxQueue

设置通信驱动程序内部的发送缓冲区最大长度（字节），该值为 0，表示设备对缓冲区长度没有限制。

dwMaxRxQueue

设置通信驱动程序内部的接收缓冲区最大长度（字节），该值为 0，表示设备对缓冲区长度没有限制。

dwMaxBaud

设置允许的最大波特率(bit/s 即 bps)。可以设置为表 4-3-2 中的值。

表 4-3-2 最大波特率可用值

值	描述	值	描述
BAUD_075	75 bps	BAUD_7200	7200 bps
BAUD_110	110 bps	BAUD_9600	9600 bps
BAUD_134_5	134.5 bps	BAUD_14400	14400 bps
BAUD_150	150 bps	BAUD_19200	19200 bps
BAUD_300	300 bps	BAUD_38400	38400 bps
BAUD_600	600 bps	BAUD_56K	56K bps
BAUD_1200	1200 bps	BAUD_57600	57600 bps
BAUD_1800	1800 bps	BAUD_115200	115200 bps
BAUD_2400	2400 bps	BAUD_128K	128K bps
BAUD_4800	4800 bps	BAUD_USER	可用于编程的波特率

dwProvSubType

指定通信设备类型。如表4-3-3所示。串口一般为PST_MODEM和PST_MODEM。

表 4-3-3 通信设备类型

值	描述
PST_FAX	传真机 FAX device
PST_LAT	以太网协议设备 LAT protocol
PST_MODEM	调制解调器 Modem device
PST_NETWORK_BRIDGE	未指明的网桥
PST_PARALLELPORT	并口 Parallel port
PST_RS232	RS-232 串口 serial port
PST_RS422	RS-422 串口
PST_RS423	RS-423 串口
PST_RS449	RS-449 串口
PST_SCANNER	扫描仪 Scanner device
PST_TCPIP_TELNET	TCP/IP Telnet® 协仪
PST_UNSPECIFIED	未指定的设备（未知名）
PST_X25	X.25 标准设备

dwProvCapabilities

设置一个位选择码来说明通信设备能够实现的功能。值如表 4-3-4 所示。

表 4-3-4　设备功能

值	描述
PCF_16BITMODE	支持特殊 16 位模式
PCF_DTRDSR	支持 DTR /DSR
PCF_INTTIMEOUTS	支持间隔超时
PCF_PARITY_CHECK	支持奇偶校验
PCF_RLSD	支持 RLSD (receive-line-signal-detect)
PCF_RTSCTS	支持 RTS (request-to-send)/CTS (clear-to-send)
PCF_SETXCHAR	支持设置 XON/XOFF
PCF_SPECIALCHARS	支持提供的特殊字符
PCF_TOTALTIMEOUTS	支持总超时（参看第 4.2 节超时结构说明）
PCF_XONXOFF	支持 XON/XOFF 流控制

dwSettableParams

设置哪种通信参数可以被更改。可以设置成表 4-3-5 中的值。

表 4-3-5　可更改的通信参数

值	描述
SP_BAUD	波特率 Baud rate
SP_DATABITS	数据位 Data bits
SP_HANDSHAKING	握手（流控制）Handshaking (flow control)
SP_PARITY	奇偶位 Parity
SP_PARITY_CHECK	奇偶校验 Parity checking
SP_RLSD	载波检测 RLSD (receive-line-signal-detect)
SP_STOPBITS	停止位 Stop bits

dwSettableBaud

指定哪种波特率可以使用。波特率值见表 4-3-2。

wSettableData

指定可以选择的数据位数。可取表 4-3-6 中的值。

表 4-3-6　数据位

值	描述
DATABITS_5	5 个数据位
DATABITS_6	6 个数据位
DATABITS_7	7 个数据位
DATABITS_8	8 个数据位
DATABITS_16	16 个数据位
DATABITS_16X	通过串行硬件线路的特殊宽路径 Special wide path through serial hardware lines

wSettableStopParity

指定可以选择的停止位数和奇偶校验方式。可取表 4-3-7 中的值。

表 4-3-7 停止位

值	描述
STOPBITS_10	1 个停止位
STOPBITS_15	1.5 个停止位
STOPBITS_20	2 个停止位
PARITY_NONE	无校验
PARITY_ODD	奇校验 Odd parity
PARITY_EVEN	偶校验 Even parity
PARITY_MARK	标号校验 Mark parity
PARITY_SPACE	空格校验 Space parity

dwCurrentTxQueue

设置通信驱动程序内部发送缓冲区大小（字节）。该值为 0 时，表示不设置该值。

dwCurrentRxQueue

设置通信驱动程序内部接收缓冲区大小（字节）。该值为 0 时，表示不设置该值。

dwProvSpec1

设置通信设备特定数据。应用程序一般忽略该值，除非通信设备有特别说明。在调用 **GetCommProperties** 函数前，将该成员参数值设置为 COMMPROP_INITIALIZED 来表明 **wPacketLength**（数据簿大小）已经设置正确有效。

dwProvSpec2

设置通信设备特定数据。应用程序一般忽略该值，除非通信设备有特别说明。

wcProvChar

设置通信设备特定数据。应用程序一般忽略该值，除非通信设备有特别说明。

注意： dwProvSpec1，dwProvSpec2 和 wcProvChar 三个成员参数内容要根据由 dwProvSubType 参数确定的通信设备类型而定，当设备类型为 PST_MODEM 时，它们分别设置如下。

dwProvSpec1：不用

dwProvSpec2：不用

wcProvChar：包含一个MODEMDEVCAPS结构（该结构请参看 **MSDN**）

（6）BuildCommDCB 设备控制块 DCB 填充函数

函数声明：

```
BOOL BuildCommDCB(
  LPCTSTR lpDef,  // 设备控制字符串
  LPDCB lpDCB     // DCB 结构
);
```

DCB 结构说明查看第 4.2 节有关介绍，该结构中存储有关串口配置信息。

返回值：

如果函数调用成功，返回非 0 值；调用失败，返回 0，这时可调用**GetLastError**得到有关错误信息。

说明：

lpDef 参数设置时用模式 Mode 命令格式。例如：设置波特率 9600，无奇偶校验，8 个

数据位，1 个停止位时用格式："9600,N,8,1"。 BuildCommDCB 函数填充好 DCB 结构数据后，由SetCommState函数设置后才能应用。

BuildCommDCB 函数通过对 Mode 语法来设置 DCB 的流控制参数：

- 当 Mode 字符串不以 x 或 p 结束时，如 9600,n,8,1
  fInX, fOutX,fOutXDsrFlow 和 **fOutXCtsFlow** 均被设置为 FALSE
  fDtrControl 设置为 DTR_CONTROL_ENABLE
  fRtsControl 设置为 RTS_CONTROL_ENABLE
- 当 Mode 字符串不以 x 结束时，如 9600,n,8,1,x
  fInX, fOutX 都设置为 TRUE
  fOutXDsrFlow,fOutXCtsFlow 都设置为 FALSE
  fDtrControl 设置为 DTR_CONTROL_ENABLE
  fRtsControl 设置为 RTS_CONTROL_ENABLE
- 当 Mode 字符串不以 p 结束时，如 9600,n,8,1,p
  fInX, fOutX are both set to FALSE
  fOutXDsrFlow,fOutXCtsFlow 设置为 TRUE.
  fDtrControl 设置为 DTR_CONTROL_HANDSHAKE
  fRtsControl 设置为 RTS_CONTROL_HANDSHAKE

（7）BuildCommDCBAndTimeouts 设备控制块 DCB 和超时 Timeouts 填充函数

函数声明：

```
BOOL BuildCommDCBAndTimeouts(
  LPCTSTR lpDef,                  // 设备控制字符串指针
  LPDCB lpDCB,                    // DCB 结构指针
  LPCOMMTIMEOUTS  lpCommTimeouts  // COMMTIMEOUTS 结构
);
```

DCB 结构和 COMMTIMEOUTS 结构说明查看第 4.2 节有关介绍，该结构中存储有关串口配置信息。

返回值：

如果函数调用成功，返回非 0 值；调用失败，返回 0，这时可调用**GetLastError**得到有关错误信息。

说明：

COMMTIMEOUTS 结构指针 lpCommTimeouts 为串行通信设置超时值，函数根据 lpDef 参数字符串中的值来设置超时结构：

- 当 lpDef 参数中包含"TO=ON"时，函数根据 lpCommTimeouts 为串行通信设置超时值；
- 当 lpDef 参数中包含"TO=OFF"时，函数不为串行通信设置超时值；
- 当 lpDef 参数中包含"TO=xxx"时，函数忽略 lpCommTimeouts 结构值，不获取该结构。

（8）GetCommTimeouts 超时获取函数

函数声明：

```
BOOL GetCommTimeouts(
```

```
  HANDLE hFile,    // 通信设备（串口）句柄，通过CreateFile返回值得到
  LPCOMMTIMEOUTS lpCommTimeouts  // COMMTIMEOUTS超时结构
);
```

COMMTIMEOUTS 超时结构说明参看第 4.2 节。

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时可调用GetLastError函数得到错误信息。

说明：

该函数用来获取指定通信设备（如串口）的超时参数值。

（9）SetCommTimeouts 超时设置函数

函数声明：

```
BOOL SetCommTimeouts(
  HANDLE hFile,    // 通信设备（串口）句柄，通过CreateFile返回值得到
  LPCOMMTIMEOUTS lpCommTimeouts  // COMMTIMEOUTS超时结构
);
```

COMMTIMEOUTS 超时结构说明参看第 4.2 节。

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时可调用GetLastError函数得到错误信息。

说明：

该函数用来设置指定通信设备（如串口）的超时参数值。

（10）ReadFile / ReadFileEx 读串口函数

这两个函数用来完成读串口操作。

➢ **ReadFile 函数**

函数声明：

```
BOOL ReadFile(
  HANDLE hFile,  // 通信设备（串口）句柄，通过CreateFile返回值得到
LPVOID lpBuffer,            //指向接收缓冲区
  DWORD nNumberOfBytesToRead,  //指明要从串口读取的字节数
  LPDWORD lpNumberOfBytesRead, //指明实际从串口设备中读出的字节数
  LPOVERLAPPED lpOverlapped    //OVERLAPPED结构
);
```

OVERLAPPED 结构参看第 4.2 节。

参数说明：

hFile

该句柄创建时，必须带有 GENERIC_READ 权限。

nNumberOfBytesToRead

指向实际读出的字节数。ReadFile 在读操作之前，首先将其设置为 0。Windows2000/NT 操作系统下，当 *lpOverlapped* 为 NULL（没有设置）时，*lpNumberOfBytesRead* 不能为 NULL，必须设置；当 *lpOverlapped* 不为 NULL（已经设置）时，*lpNumberOfBytesRead* 可以为 NULL，如果这是一个重叠 I/O 操作，可以通过调用GetOverlappedResult函数得到实际读取的字节数。在 Windows9x 操作系统下，该参数不能为 NULL。

lpOverlapped

如果 hFile 为 FILE_FLAG_OVERLAPPED，该参数不能为 NULL，而应该是一个有效的 OVERLAPPED 结构；否则不需要此结构，设置为 NULL。

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时可调用GetLastError函数得到错误信息。需要注意的是，如果该函数因为超时而返回，返回值仍为 TRUE。而当返回 FALSE 时，也不一定就是操作失败，如重叠 I/O 操作时，如果操作未完成函数就返回，那么 ReadFile 函数返回值为 FALSE，但GetLastError函数返回的值为 ERROR_IO_PENDING，所以在处理错误信息时，最好调用GetLastError函数来得到进一步的错误信息。

说明：

ReadFile 支持同步和异步操作。该函数还有许多其他的规定，在编程时不一定用到，如果遇到问题，可以再进一步查询 MSDN。

➢ **ReadFileEx 函数**

ReadFileEx 函数是 ReadFile 函数的扩展，与后者的区别是：ReadFileEx 只能异步执行，允许应用程序继续执行其他的操作，并异步地报告读操作的完成状态。其声明如下：

```
BOOL ReadFileEx(
  HANDLE hFile,                    //通信设备（串口）句柄，通过 CreateFile 返回值得到
  LPVOID lpBuffer,                 //指向接收缓冲区
  DWORD nNumberOfBytesToRead,      //指明要从串口读取的字节数
  LPOVERLAPPED lpOverlapped,       // OVERLAPPED 结构
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
                                   // 指向一个 I/O 操作完成例程
);
```

其他返回值与 ReadFile 函数相同。

（11）WriteFile / WriteFileEx 写串口函数

这两个函数用来完成写串口操作。WriteFile 函数同时支持同步和异步操作，而 WriteFileEx 函数，则只支持异步操作。

➢ **WriteFile 函数**

函数声明：

```
BOOL WriteFile(
  HANDLE hFile,                      // 通信设备（串口）句柄，通过 CreateFile 返回值得到
  LPCVOID lpBuffer,                  // 指向发送缓冲区
  DWORD nNumberOfBytesToWrite,       // 设置要向串口发送缓冲区写入的字节数
  LPDWORD lpNumberOfBytesWritten,    // 实际写入发送缓冲区的字节数
  LPOVERLAPPED lpOverlapped          // overlapped I/O 结构
);
```

OVERLAPPED 结构参看第 4.2 节。

参数说明：

hFile

该句柄创建时，必须带有 GENERIC_WRITE 权限。

nNumberOfBytesToWrite

指向实际读出的字节数。WriteFile 在读操作之前，首先将其设置为 0。Windows2000/NT 操作系统下，当 *lpOverlapped* 为 NULL（没有设置）时，*lpNumberOfBytesWrite* 不能为 NULL，必须设置；当 *lpOverlapped* 不为 NULL（已经设置）时，*lpNumberOfBytesRead* 可以为 NULL；如果这是一个重叠 I/O 操作，可以通过调用GetOverlappedResult函数得到实际读取的字节数。在 Windows9x 操作系统下，该参数不能为 NULL。

lpOverlapped

如果 hFile 为 FILE_FLAG_OVERLAPPED，该参数不能为 NULL，而应该是一个有效的 OVERLAPPED 结构；否则不需要此结构，设置为 NULL。

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时可调用GetLastError函数得到错误信息。

说明：

WriteFile 支持同步和异步操作。

➢ **WriteFileEx 函数**

WriteFileEx 函数是 WriteFile 函数的扩展，与后者的区别是：WriteFileEx 只能异步执行，允许应用程序继续执行其他的操作，并异步地报告读操作的完成状态。其声明如下：

```
BOOL WriteFileEx(
  HANDLE hFile,        //通信设备（串口）句柄，通过CreateFile 返回值得到
  LPCVOID lpBuffer,            //指向写入数据缓冲区
  DWORD nNumberOfBytesToWrite,  // 设置要写的字节数
  LPOVERLAPPED lpOverlapped,    // OVERLAPPED 结构，指向异步 I/O 数据
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
                          //指向一个 I/O 操作完成例程
);
```

OVERLAPPED 结构参看第 4.2 节。

返回值与 WriteFile 函数相同。

（12）ClearCommError 清除错误标志函数

在调用 ReadFile 和 WriteFile 之前，应用程序应该调用 ClearCommError 函数清除错误标志。该函数负责报告指定的错误和设备的当前状态。

函数声明如下：

```
BOOL ClearCommError(
  HANDLE hFile,   //通信设备（串口）句柄，通过CreateFile 返回值得到
  LPDWORD lpErrors, // 错误掩码类型
  LPCOMSTAT lpStat  // pointer to buffer for communications status
);
```

参数说明：

lpErrors

错误掩误类型（32 位变量）。通信错误类型见表 4-2-5。

lpStat

指向一个 **COMSTAT** 结构，该结构可返回通信设备状态信息。如果 *lpStat* 为 NULL，不返回任何信息。**COMSTAT** 结构介绍见第 4.2.4 节。

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时可调用GetLastError函数得到错误信息。

（13）PurgeComm 终止读写 / 清空缓冲区函数

调用 PurgeComm 函数可以终止正在进行的读写操作，该函数还会清除输入或输出缓冲区中的内容。

函数声明如下：

```
BOOL PurgeComm(
  HANDLE hFile, //通信设备（串口）句柄，通过 CreateFile 返回值得到
  DWORD dwFlags  // 执行的操作
);
```

参数说明：

dwFlags

指定函数执行何种操作。可取表 4-3-8 中的值。

表 4-3-8　PurgeComm 函数执行的操作

值	描述
PURGE_TXABORT	即使发送（写）操作没有完成，也立即终止所有的重叠发送操作，立即返回
PURGE_RXABORT	即使接收（读）操作没有完成，也立即终止所有的重叠接收操作，立即返回
PURGE_TXCLEAR	清空发送缓冲区
PURGE_RXCLEAR	清空接收缓冲区

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时可调用GetLastError函数得到错误信息。

（14）FlushFileBuffer 清空缓冲区函数

注意：PurgeComm 函数可以在读写操作的同时，清空缓冲区，这就不能保证缓冲区内所有字符均被发送。如果要保证缓冲区内的所有字符均被发送，则可以调用 FlushFileBuffer 函数，它在所有的写操作完成后才返回。

```
BOOL FlushFileBuffers(
  HANDLE hFile   //通信设备（串口）句柄，通过 CreateFile 返回值得到
);
```

参数说明：

hFile

要清空缓冲区的通信设备句柄，该句柄必须有 GENERIC_WRITE 权限。

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时可调用GetLastError函数得到错误信息。

（15）GetOverlappedResult 异步 I/O 操作结果获取函数

利用 GetOverlappedResult 函数可以得到异步（overlapped ）I/O 操作结果。

函数声明如下：

```
BOOL GetOverlappedResult(
  HANDLE hFile,                        //通信设备（串口）句柄，通过CreateFile返回值得到
  LPOVERLAPPED lpOverlapped,           // pointer to overlapped structure
  LPDWORD lpNumberOfBytesTransferred,  // pointer to actual bytes count
  BOOL bWait                           // wait flag
);
```

参数说明：

hFile

通信设备（串口）句柄，通过 CreateFile 返回值得到，它与异步 I/O 操作开始调用的ReadFile，WriteFile，ConnectNamedPipe，TransactNamedPipe，DeviceIoControl或WaitCommEvent等函数是一致的。

lpOverlapped

异步 I/O 操作启动时指定的OVERLAPPED 结构

lpNumberOfBytesTransferred

指向一个 32 位变量，该变量值为一个读/写操作实际传输的字节数。

bWait

指明函数是否等待被挂起的异步 I/O 操作完成。若该参数值为 TRUE，则函数要等到异步 I/O 操作完成后才返回；若该参数值为 FALSE，同时操作处于挂起状态，则函数返回 FALSE，并且GetLastError函数返回 ERROR_IO_INCOMPLETE。

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时可调用GetLastError函数得到进一步的错误信息。

（16）WaitForSingleObject 异步 I/O 操作事件获取函数

WaitForSingleObject 函数在以下两种情况出现后返回：

- 检测的对象出现特定的异步 I/O 操作事件；
- 超时时间到。

函数声明如下：

```
DWORD WaitForSingleObject(
  HANDLE hHandle,//OVERLAPPED I/O操作事件hEvent句柄
  DWORD dwMilliseconds  // time-out interval in milliseconds
);
```

参数说明：

hHandle

OVERLAPPED I/O 操作事件 hEvent 句柄

dwMilliseconds

以 ms 为单位设置超时时间。超时时间到后，即使没有引起异步 I/O 操作事件的信号，函数也返回。如果 *dwMilliseconds* 为 0，函数检测对象的状态后，不等待就马上返回；如果 *dwMilliseconds* 为 INFINITE，则超时时间为无穷大。

返回值：

若调用成功，则返回导致本函数返回的通信事件。返回值及其描述如表 4-3-9。不成功则返回 WAIT_FAILED，有错误时可调用GetLastError函数得到进一步的错误信息。

表 4-3-9　WaitForSingleObject 函数返回值

值	描述
WAIT_ABANDONED	等待的对象是互斥对象，且该对象的拥有线程在本身终止时没有将它释放，且对象被设置为无信息状态
WAIT_OBJECT_0	等待的对象处于有信号状态
WAIT_TIMEOUT	超时时间到，且指定的对象仍处于无信号状态

WaitForSingleObject / **WaitForSingleObjectEx** 以及 **WaitForMultipleObjects** / **WaitForMultipleObjectsEx** 的应用都相似，可以参看 MSDN。

WaitForMultipleObjects 的声明如下：

```
DWORD WaitForMultipleObjects(
  DWORD nCount,            // 事件句柄数组中的事件数
  CONST HANDLE *lpHandles,  //事件数组
  BOOL fWaitAll,    // TRUE: 所有事件发生时才返回;
                //FALSE: 只要事件数组中的事件有一个发生，就返回
  DWORD dwMilliseconds      // 以ms为单位设置超时时间
);
```

若调用成功，fWaitAll 为 FALSE，其返回值即为引发的事件号：WAIT_OBJECT_X （X 为事件数据中的事件下标 0～数据事件数减 1，即 WAIT_OBJECT_0 to (WAIT_OBJECT_0 + *nCount* – 1)，其中，*nCount* 为数组大小）。

（17）SetCommMask 通信事件设置函数

SetCommMask 函数可以设置一组在应用程序中监测通信设备通信的事件。函数声明如下：

```
BOOL SetCommMask(
  HANDLE hFile,    // handle to communications device
  DWORD dwEvtMask  // mask that identifies enabled events
);
```

参数说明：

hFile

通信设备（串口）句柄，通过 CreateFile 返回值得到。

dwEvtMask

事件码（掩码）。如果设置该值为 0，则禁止所有事件。通信事件码见表 4-2-6。

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时可调用GetLastError函数得到进一步的错误信息。

（18）GetCommMask 当前通信事件获取函数

GetCommMask 函数可以获取通信设备中当前正在发生的通信事件。

函数声明如下：

```
BOOL GetCommMask(
  HANDLE hFile,     // handle to communications device
  LPDWORD lpEvtMask // pointer to variable to get event mask
);
```

参数说明：

hFile

通信设备（串口）句柄，通过 CreateFile 返回值得到。

dwEvtMask

事件码（掩码）。通信设备中当前正在发生的通信事件掩码将放入这个 32 位变量中。通信事件码见表 4-2-6。

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时，可调用GetLastError函数得到进一步的错误信息。

（19）WaitCommEvent 通信事件监测函数

WaitCommEvent 函数监视（等待）通信设备中发生的通信事件。

函数声明如下：

```
BOOL WaitCommEvent(
  HANDLE hFile,              // handle to communications device
  LPDWORD lpEvtMask,          // pointer to variable to receive event
  LPOVERLAPPED lpOverlapped,   // pointer to overlapped structure
);
```

参数说明：

hFile

通信设备（串口）句柄，通过 CreateFile 返回值得到。

lpEvtMask

事件码（掩码）。通信设备中当前正在发生的通信事件掩码将放入这个 32 位变量中。通信事件码见表 4-2-6。

lpOverlapped

OVERLAPPED 结构，该结构要求设备句柄 *hFile* 设置为 FILE_FLAG_OVERLAPPED。

- 如果设备句柄 *hFile* 被设置为 FILE_FLAG_OVERLAPPED，则 *lpOverlapped* 不能为 NULL，否则将错误地报告操作已完成。
- 如果设备句柄 *hFile* 被设置为 FILE_FLAG_OVERLAPPED，且 *lpOverlapped* 不为 NULL，**WaitCommEvent** 函数将以异步方式执行操作，这种情况下，**OVERLAPPED** 结构中必须含有一个由人工复位的事件对象，这个事件对象由函数 **CreateEvent** 来创建。
- 如果设备句柄 *hFile* 打开时没有被设置为 FILE_FLAG_OVERLAPPED，则 **WaitCommEvent** 函数只有等到特定通信或者错误发生后才返回。

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时，可调用GetLastError函数得到进一步的错误信息。

注意： WaitCommEvent 函数在监视通信事件时，如果异步操作不能立即完成，则该函数返回 FALSE（0），并且 GetLastError 函数返回 ERROR_IO_PENDING，表明该操作正在后台执行。发生这种情况时，系统在 WaitCommEvent 函数返回之前将 OVERLAPPED 结构中的

成员 hEvent 参数值设置为无信号状态，之后等到有特定通信事件或发生错误时，系统再将其置为有信号状态。应用程序可以使用一个如 WaitForSingleObject 的等待函数（wait functions）来确定 WaitCommEvent 函数的操作结果（因为 WaitCommEvent 函数的操作也是异步操作），并用 GetOverlappedResult 函数来报告该操作的成功与否，此时，WaitCommEvent 函数的 lpEvtMask 变量被设置为发生的事件掩码。

如果一个应用程序（进程）在 **WaitCommEvent** 函数操作进行时使用 **SetCommMask** 函数来改变通信设备的事件掩码，则 **WaitCommEvent** 函数将立即返回，并且 *lpEvtMask* 指向的变量被设置为 0。

还有几点要注意的是：

- **WaitCommEvent** 函数只检测在等待开始后的事件。如指定要检测 EV_RXCHAR 事件，只有当接收到字符并将字符放入接收缓冲区后才能满足等待条件，而调用 **WaitCommEvent** 函数之前已在接收缓冲区中的字符则不符合检测条件。
- **WaitCommEvent** 函数用来监视 CTS、DSR 等握手信号状态的改变事件时，只报告信号状态的变动，但不报告当前的具体状态。如果要获取这些信号状态，应用程序可调用 GetCommModemStatus 函数。

（20）CreateEvent 事件创建函数

在 **WaitCommEvent** 函数中，如果设备句柄 *hFile* 被设置为 FILE_FLAG_OVERLAPPED，且 *lpOverlapped* 不为 NULL，则 **WaitCommEvent** 函数将以异步方式执行操作。这种情况下，**OVERLAPPED** 结构中必须含有一个由人工复位的事件对象，这个事件对象由函数 **CreateEvent** 函数来创建。可见，**CreateEvent** 函数在编程中常用于创建接收数据的消息事件。

函数声明如下：

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
                  // pointer to security attributes
  BOOL bManualReset,  // flag for manual-reset event
  BOOL bInitialState, // flag for initial state
  LPCTSTR lpName      // pointer to event-object name
);
```

参数说明：

lpEventAttributes

指向一个**SECURITY_ATTRIBUTES** 结构，这个结构能确定返回的句柄是否可以由子进程来继承。如果 *lpEventAttributes* 为 NULL，那么句柄不能被继承。

说明：在 Windows NT 操作系统中，SECURITY_ATTRIBUTES 结构中的成员参数 lpSecurityDescriptor 为新创建的事件定义了一个安全（security）描述符。如果 lpEventAttributes 为 NULL，则新事件获得一个缺省的安全描述符。

bManualReset

指定新创建的事件是手工复位还是自动复位。如果为 TRUE，那么在应用程序中必须调用**ResetEvent** 函数手工将通信事件状态设为没有信号状态。如果为 FALSE，则系统在一个等待线程结束后自动将事件状态复位为没有信号状态。

bInitialState

指定事件对象的初始状态。*bInitialState* 为 TRUE，则初始状态为有信号状态；否则为无信号状态。

lpName

指向一个以 NULL（空字符）结束字符串命名的事件对象，该对象名最大长度为 MAX_PATH，可以包含除“\”以外的任何字符。对大小写敏感。

返回值：

若调用成功，返回指向事件对象的句柄，若创建的事件对象名已经存在，则返回指向该存在事件句柄，同时，GetLastError函数返回 ERROR_ALREADY_EXISTS。不成功则为 0，有错误时，可调用GetLastError函数得到进一步的错误信息。

注意事项：

用 **SetEvent** 函数置事件对象为有信号状态，调用 **ResetEvent** 函数复位状态（将事件置为无信号状态）。用 **CloseHandle** 函数来关闭事件句柄。

（21）EscapeCommFunction 握手信号设置函数

EscapeCommFunction 函数可以将硬件握手信号置为 ON 或 OFF，也可以设置软件模拟 XON 和 XOFF 字符的接收和发送。同时，**EscapeCommFunction** 函数也可清除终止条件，这与 **SetCommBreak** 函数和 **ClearCommBreak** 函数的功能相同。

函数声明如下：

```
BOOL EscapeCommFunction(
  HANDLE hFile,  // handle to communications device
  DWORD dwFunc   // extended function to perform
);
```

参数说明：

hFile

通信设备（串口）句柄，通过 CreateFile 返回值得到。

dwFunc

指定要执行的握手信号扩展功能码，其值为表 4-3-10 中所列代码之一。

表 4-3-10　握手信号扩展功能码

值	描述
CLRDTR	清除 DTR (data-terminal-ready) 信号
CLRRTS	清除 RTS (request-to-send) 信号
SETDTR	设置发送 DTR (data-terminal-ready) 信号
SETRTS	设置发送 the RTS (request-to-send) 信号
SETXOFF	假设收到一个 XOFF 字符，停止传输
SETXON	假设收到一个 XON 字符，停止传输
SETBREAK	停止字符传输，并使传输线路处于中断（break ）状态，直至调用**ClearCommBreak** 函数（或者**EscapeCommFunction** 函数调用 CLRBREAK 扩展功能码）。SETBREAK 扩展功能码与**SetCommBreak**函数作用是一样的，要注意的是，SETBREAK 扩展功能码并不清除缓冲区中还没有发送的数据
CLRBREAK	恢复字符传输，使传输线路处于非中断（nonbreak ）状态。CLRBREAK 扩展功能码与**ClearCommBreak** 函数作用相同

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时，可调用GetLastError函数得到进

一步的错误信息。

注意事项：

在实际编程中，可以通过设置 CLRDTR/SETDTR，CLRRTS / SETRTS 等功能码来实现手工控制流量。

（22）SetCommBreak 通信设备挂起（暂停）函数

SetCommBreak 函数将指定的通信设备暂停字符传输，处于挂起状态，并使传输线暂处于中断状态（Break）状态，直到调用 **ClearCommBreak** 函数来恢复为止。

函数声明如下：

```
BOOL SetCommBreak(
  HANDLE hFile   //通信设备（串口）句柄，通过 CreateFile 返回值得到
);
```

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时，可调用GetLastError函数得到进一步的错误信息。

（23）ClearCommBreak 通信设备恢复函数

通信设备被 **SetCommBreak** 函数或 **EscapeCommFunction** 函数设置成中断（break）状态后，由 **ClearCommBreak** 函数来恢复。

函数声明如下：

```
BOOL ClearCommBreak(
  HANDLE hFile   //通信设备（串口）句柄，通过 CreateFile 返回值得到
);
```

返回值：

若调用成功，返回非零值；不成功，则为 0，有错误时，可调用GetLastError函数得到进一步的错误信息。

为了查询方便，这里和本节的开始均列出各个函数的说明顺序号：

（1）**CreateFile** 打开串口函数

（2）**SetupComm** 缓冲区分配函数

（3）**GetCommState** 获取串口当前配置函数

（4）**SetCommState** 配置串口函数

（5）**GetCommProperties** 获取串口属性函数

（6）**BuildCommDCB**　DCB 填充函数

（7）**BuildCommDCBAndTimeouts** DCB 和 Timeouts 填充函数

（8）**GetCommTimeouts** 超时获取函数

（9）**SetCommTimeouts** 超时设置函数

（10）**ReadFile / ReadFileEx** 读串口函数

（11）**WriteFile / WriteFileEx** 写串口函数

（12）**ClearCommError** 清除错误标志函数

（13）**PurgeComm** 终止读写 / 清空缓冲区函数

（14）**FlushFileBuffer** 清空缓冲区函数

（15）**GetOverlappedResult** 异步 I/O 操作结果获取函数

（16）**WaitForSingleObject** 异步 I/O 操作事件获取函数
（17）**SetCommMask** 通信事件设置函数
（18）**GetCommMask** 当前通信事件获取函数
（19）**WaitCommEvent** 通信事件监测函数
（20）**CreateEvent** 事件创建函数
（21）**EscapeCommFunction** 握手信号设置函数
（22）**SetCommBreak** 通信设备挂起函数
（23）**ClearCommBreak** 通信设备恢复函数

4.4 Win32 API 串口通信编程的一般流程和特殊实例

在上一节里，我们讲述了大量的 API 函数（头都大了吧），但编程时并不是所有的函数都要用到，在实际编程时，要根据具体情况选用。关于 Win32 串口编程，建议读者再读读 MSDN 中的“**Serial Communications in Win32**”。为方便阅读，将此文放在本书附录，中文翻译将放在作者个人技术主页上（特别是在校生，有空一定要读读，还可以考验你的英文水平）。

在这一节里，介绍应用 API 函数进行串口编程和一般流程。我们在编程中，一般用异步方式、消息处理去读写串口，常用编程方式在随后的几节中都用实例进行了说明。本节并简要介绍了几个异于普通方法的特殊实例。

4.4.1 Win32 API 串口通信编程的一般流程

应用 API 函数进行串口编程时，其一般编程步骤如下：

（1）打开串口：CreateFile 函数。

（2）建立串口通信事件： CreateEvent 函数。

（3）初始化串口，设置串口参数：SetCommState 函数。

（4）建立读数据的线程：建立线程简单程序中并非必要，需要用到的函数较多，如 ReadFile；如果要检测通讯状态，如 CTS 信号，RingIn 等等，则用 SetCommMask、WaitCommEvent、ClearCommError、GetCommModemStatus。

（5）写数据：用 WriteFile。

（6）结束时关闭端口：若程序中打开了其他线程，则先终止线程。停止 WaitCommEvent 的等待以及关闭端口 CloseHandle，平时程序会停留在 WaitCommEvent 的等待中，当要终止线程的时候，程序必须从 WaitCommEvent 中退出来，按照 Win32 手册上的说明，参数为 NULL 的 SetCommMask 会使另一个线程中的 WaitCommEvent 马上返回，然后用 CloseHandle 关闭端口。

4.4.2 用查询方式读串口

查询方式读串口的关键是得到缓冲区的字节数。下面的例程中，通过 ClearCommError() 的调用来确定接收缓区中等待读取的字节数。通过查看第 4.3 节，函数 ClearCommError(HANDLE hFile, LPDWORD lpErrors, LPCOMSTAT lpStat)将返回

个 COMSTAT 结构 lpStat。

```
        typedef struct _COMSTAT { // cst
  ...
  DWORD cbInQue; // bytes in input buffer
  DWORD cbOutQue; // bytes in output buffer
        } COMSTAT, *LPCOMSTAT;
```

其中，cbInQue 和 cbOutQue 即为缓冲区字节，cbInQue 为接收缓冲区字节数。

例程代码如下：

```
COMMTIMEOUTS to;
DWORD ReadThread(LPDWORD lpdwParam)
{
    BYTE inbuff[100];
    DWORD nBytesRead;
    if(!(cp.dwProvCapabilities&PCF_INTTIMEOUTS))
        return 1L;
    memset(&to,0,sizeof(to));
    to.ReadIntervalTimeout = MAXDWORD;
    SetCommTimeouts(hComm,&to);
    while(bReading)
    {
        if(!ReadFile(hComm,inbuff,100,&nBytesRead,NULL))
            locProcessCommError(GetLastError());
        else
            if(nBytesRead)
                locProcessBytes(inbuff,nBytesRead);
    }
    PurgeComm(hComm,PURGE_RXCLEAR);
    return 0L;
}
```

其中，PurgeComm()是一个清除函数，它可以终止任何未决的后台读/写操作，并且可以冲掉 I/O 缓冲区。

4.4.3　同步 I/O 读写数据

《实战串行通讯》（罗云彬）一文中提到下面一段编程经历：同步（NonOverLapped）方式是比较简单的一种方式，编写的代码的长度要明显少于异步（OverLapped）方式。我用同步方式编写了整个子程序，在 Windows 98 下工作正常，但后来在 Windows 2000 下测试，发现接收正常，但一发送数据，程序就会停在那里，原因应该在于同步方式下如果有一个通讯 API 在操作中，另一个会阻塞直到上一个操作完成，所以，当读数据的线程停留在 WaitCommEvent 的时候，WriteFile 就停在那里。我又测试了我手上所有有关串行通信的例子程序，发现所有使用同步方式的程序在 Windows 2000 下全部工作不正常，对这个问题我一直找不到解决的办法，后来在 Iczelion 站点上发现一篇文章提到 NT 下对串行通信的处理和 9x 有些不同，根本不要指望在 NT 或 Windows 2000 下用同步方式同时收发数据，我只好又用异步方式把整个通讯子程序重新写了一遍。所以对于这个问题的建议是：如果程序只打算工作在 Windows 9x 下，为了简单起见，可以用同步方式写程序，如果程序打算在 NT 下也可以工作，就必须用异步方式写。

下面是一段同步读数的代码，读者可以参考一下：

```
COMMTIMEOUTS to;
DWORD ReadThread(LPDWORD lpdwParam)
{
    BYTE inbuff[100];
    DWORD nByteRead,dwErrorMask,nToRead;
    COMSTAT comstat;
    if(.cp.dwProvCapabilities&PCF_TOTALTIMEOUTS)
        return 1L;
    memset(&to,0,sizeof(to));
    to.ReadTotalTimeoutMultiplier = 5;
    to.ReadTotalTimeoutConstant = 50;
    SetCommTimeouts(hComm,&to);
    while(bReading)
    {
        ClearCommError(hComm,&dwErrorMask,&comstat);
        if(dwErrorMask)
            locProcessCommError(dwErrorMask);
        if(comstat.cbInQue >100)
            nToRead = 100;
        else
            nToRead = comstat.cbInQue;
        if(nToRead == 0)
            continue;
        if(!ReadFile(hComm,inbuff,nToRead,&nBytesRead,NULL))
            locProcessCommError(GetLastError());
        else
            if(nBytesRead)
                locProcessBytes(inbuff,nBytesRead);
    }
    return 0L;
}
```

4.4.4 关于流控制的设置问题

流控制的设置可以用 EscapeCommFunction 函数来设置，也可以通过 DCB 结构来设置。

在流控制方式为“无”和“软件控制”的情况下，基本上没有什么问题，但在“硬件控制”下，Win32 手册中说明 RTS_CONTROL_HANDSHAKE 控制方式的含义如下：

Enables RTS handshaking. The driver raises the RTS line when the “type-ahead” (input) buffer is less than one-half full and lowers the RTS line when the buffer is more than three-quarters full. If handshaking is enabled, it is an error for the application to adjust the line by using the EscapeCommFunction function.

也就是说，当缓冲区快满的时候，RTS 会自动 OFF 通知对方暂停发送，当缓冲区重新空出来的时候，RTS 会自动 ON，但我发现当 RTS 变 OFF 以后，即使你已经清空了缓冲区，RTS 也不会自动地 ON，造成对方停在那里不发送了。所以，如果要用硬件流控制的话，则还要在接收后最好加上检测缓冲区大小的判断，具体是使用 ClearCommError 后返回的 COMSTAT.cbInQue，当缓冲区已经空出来的时候，要使用 invoke EscapeCommFunction (hCom, SETRTS) 重新将 RTS 设置为 ON。

4.5　CSerialPort 类中的 API 函数编程应用剖析

他山之石，可以攻玉。我们在编程时要大量学习别人的经验，才能不断提高自己。第 2 章中介绍的 CSerialPort 类是 API 函数串口编程的比较典型的应用。让我们来仔细看看其中 API 函数是如何发挥作用的。

阅读本节时，可以在计算机上同时用 VC 打开 CSerialPort 类文件。

1．打开串口句柄：CreateFile 函数

首先看到，该类中的串口句柄为：

HANDLE　　m_hComm;

该句柄在应用 CreateFile 函数创建后，就得到了指定了的串口句柄，其后，几乎所有的函数要对串口进行操作时，必须通过 m_hComm 进行。CreateFile 函数的应用在 BOOL CSerialPort::InitPort（）函数中：

```
// 为串口得到逻辑名，这样在初始化函数时，只要用数字表示串口号，如1表示COM1
    sprintf(szPort, "COM%d", portnr);
    sprintf(szBaud, "baud=%d parity=%c data=%d stop=%d", baud, parity, databits,
stopbits);

    // 得到要打开的串口句柄，供以后的操作使用
    m_hComm = CreateFile(szPort, // 串口号(COMX)
        GENERIC_READ | GENERIC_WRITE,     // 可以同时读写
        0,                        // 以独占方式打开串口
        NULL,                       // 无安全属性
        OPEN_EXISTING,   //串口设备必须用OPEN_EXISTING，因为它肯定存在
        FILE_FLAG_OVERLAPPED,    // （重叠）异步 I/O 操作方式
        0);          // 对串口设备，模板文件句柄必须为0（NULL）

//以下是打开串口不成功，进行错误处理
    if (m_hComm == INVALID_HANDLE_VALUE)
    {
        // port not found
        delete [] szPort;
        delete [] szBaud;
        return FALSE;
    }
```

因为串口句柄在打开时，设置了异步（重叠） I/O 操作方式，因此，必须对串口的超时做出说明：

定义超时结构：

```
COMMTIMEOUTS  m_CommTimeouts;
```

再对超时变量进行设置：

```
// 设置读间隔超时变量
m_CommTimeouts.ReadIntervalTimeout = 1000;
m_CommTimeouts.ReadTotalTimeoutMultiplier = 1000;
m_CommTimeouts.ReadTotalTimeoutConstant = 1000;
m_CommTimeouts.WriteTotalTimeoutMultiplier = 1000;
m_CommTimeouts.WriteTotalTimeoutConstant = 1000;
```

2. 建立串口通信事件： CreateEvent 函数

用异步方式操作串口必须定义 OVERLAPPED 结构，其中的 hEvent 必须自己建立，要定义两个 OVERLAPPED 结构，分别用于读和写，当然也必须建立两个 Event，把它们放入 OVERLAPPED.hEvent，还是在 BOOL CSerialPort::InitPort()函数中：

```
// create events 创建事件,
// 以下三个CreateEvent 函数应用均设置为手工复位方式
if (m_ov.hEvent != NULL)
        ResetEvent(m_ov.hEvent);
    m_ov.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    if (m_hWriteEvent != NULL)
        ResetEvent(m_hWriteEvent);
    m_hWriteEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    if (m_hShutdownEvent != NULL)
        ResetEvent(m_hShutdownEvent);
    m_hShutdownEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
```

接着为这些事件设定了优先级别。

3. 初始化串口，设置串口参数：SetCommState 函数

```
// configure
if (SetCommTimeouts(m_hComm, &m_CommTimeouts))  //设置超时
{
    if (SetCommMask(m_hComm, dwCommEvents))  //设置通信事件
    {
        if (GetCommState(m_hComm, &m_dcb))  //获取当前的DCB 结构参数
        {
            m_dcb.EvtChar = 'q';  //设置字件字符
            m_dcb.fRtsControl = RTS_CONTROL_ENABLE;    // set RTS bit high!
            if (BuildCommDCB(szBaud, &m_dcb))  //填写DCB 结构
            {
                if (SetCommState(m_hComm, &m_dcb))  //利用填写好的DCB 配置
                    ; // normal operation... continue
                else
                    ProcessErrorMessage("SetCommState()");
            }
            else
                ProcessErrorMessage("BuildCommDCB()");
        }
        else
            ProcessErrorMessage("GetCommState()");
    }
    else
        ProcessErrorMessage("SetCommMask()");
}
else
    ProcessErrorMessage("SetCommTimeouts()");
```

配置好串口参数后，又应用 **PurgeComm** 函数终止读写并清空接收与发送缓冲区：

```
// flush the port
PurgeComm(m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT | PURGE_TXABORT);
```

注意：这时，串口已经打开，但 CSerialPort 类开辟了一个监视线程来接收数据，在监视线程没有启动之前，是不会处理接收事件的。所以，应用这个类时，通常打开串口的方法如下。

以下 m_SerialPort 为 CSerialPort 类对象。

```
if(m_SerialPort.InitPort(this, nPort, 9600,'N',8,1,EV_RXFLAG | EV_RXCHAR,512))
{
    m_SerialPort.StartMonitoring(); //开启线程
}
else
{
    AfxMessageBox("没有发现此串口");
}
```

4. 建立读数据的监视线程

一般我们是在主线程中写数据，因为写是可以控制的，而读的时候我们不知道数据什么时候会到，所以要建立一个线程专门用来读数据。在这个线程中，循环地用 ReadFile 读串口，同时用 WaitCommEvent 检测线路状态。

建立读线程需要用到的函数较多：ReadFile，如果要检测通信状态，如 CTS 信号，RingIn 等等，则用 SetCommMask、WaitCommEvent、ClearCommError、GetCommModemStatus。

CSerailPort 类中，监视线程函数为 CommThread 函数。在以下的线程处理中，程序中多次用到了 Critical Section，这是一种保证在一个时间只有一个线程访问数据集的非常简单的方法，以保证线程数据同步。当你使用 Critical Section，你给了线程一个它们必须共享的对象。任何拥有 Critical Section 对象的线程可以访问被保护起来的数据。其他线程必须等待直到第一个线程释放了 Critical Section 对象，此后，其他线程可以按照顺序抢占 Critical Section 对象，访问数据。因为线程只有拥有 Critical Section 对象才能访问数据，而且在一个时刻只有一个线程可以拥有 Critical Section 对象，所以绝不会出现一个时刻有多个线程访问数据。线程的具体建立方法在此不做详述，读者可以参考有关资料。

```
UINT CSerialPort::CommThread(LPVOID pParam)
{
    // Cast the void pointer passed to the thread back to a pointer of CSerialPort class
    CSerialPort *port = (CSerialPort*)pParam;
    // Set the status variable in the dialog class to TRUE to indicate the thread is
running.
    port->m_bThreadAlive = TRUE;

    // Misc. variables
    DWORD BytesTransfered = 0;
    DWORD Event = 0;
    DWORD CommEvent = 0;
    DWORD dwError = 0;
    COMSTAT comstat;
    BOOL bResult = TRUE;

    // 利用 PurgeComm 函数清空缓冲区 Clear comm buffers at startup
    if (port->m_hComm)      // 检查串口是否打开
        PurgeComm(port->m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT |
PURGE_TXABORT);
```

```
        // 开始一个无循环，只要线程没有终止，就不会停止读数据
        for (;;)
        {
```

/* ////////调用通信事件监测函数 WaitCommEvent ()时，该函数将会立即返回。这是因为串口句柄打开时设置为 FILE_FLAG_OVERLAPPED---（重叠）异步 I/O 操作方式

WaitCommEvent 函数在监视通信事件时，如果异步读操作不能立即完成，则该函数返回 FALSE（0），并且 GetLastError 函数返回 ERROR_IO_PENDING，表明该操作正在后台执行。发生这种情况时，系统在 WaitCommEvent 函数返回之前将 OVERLAPPED 结构中的成员 hEvent 参数值设置为无信号状态，之后等到有特定通信事件或发生错误时，系统再将其置为有信号状态。所以只要有字符到达，就会产生事件。

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

            bResult = WaitCommEvent(port->m_hComm, &Event, &port->m_ov);
    //如果 WaitCommEvent 返回值为 NULL，
    //调用 GetLastError 函数来查询错误信息
            if (!bResult)
            {
                switch (dwError = GetLastError())
                {
                case ERROR_IO_PENDING:
                    {
                        // 这是正常情况，说明现在没有字符可读
                        break;
                    }
                case 87:  //87 错误是系统错误，
    //ERROR_INVALID_PARAMETER 但并不影响数据接收
                    {
                        // Under Windows NT, this value is returned for some reason.
                        // I have not investigated why, but it is also a valid reply
                        // Also do nothing and continue.
                        break;
                    }
                default:
                    {  //如果发生其他错误，则返回错误信息
                        port->ProcessErrorMessage("WaitCommEvent()");
                        break;
                    }
                }
            }
            else   //下面是 WaitCommEvent 函数正确返回的处理
            {
                // 如果 WaitCommEvent() 返回 TRUE，检查接收缓冲区中
                // 是否有字符可读 。
                //
```

/*注意：如果你要一次就从接收缓冲区中读出多个字符（本类没有这样做），就要用 WaitForMultipleObjects 函数，当第一个字符到达缓冲区时，引发 WaitForMultipleObjects 函数，当函数返回时，将异步（重叠）I/O 操作事件句柄 m_OverlappedStruct.hEvent 变量复位，即设置为无信号状态。如果在复位操作后到调用 ReadFile 函数之间有其他字符到达，m_OverlappedStruct.hEvent 会被重新置位，即设置为有信号状态。这时调用 ReadFile 函数，ReadFile 函数将从缓冲区中读出所有数据，然后本循环回到 WaitCommEvent()。

```
            // At this point you will be in the situation where m_OverlappedStruct.hEvent
is set,
            // but there are no bytes available to read.  If you proceed and call
            // ReadFile(), it will return immediatly due to the async port setup, but
            // GetOverlappedResults() will not return until the next character arrives.
            //
            // It is not desirable for the GetOverlappedResults() function to be in
```

```
            // this state. The thread shutdown event (event 0) and the WriteFile()
            // event (Event2) will not work if the thread is blocked by GetOverlappedResults().
            //
            // The solution to this is to check the buffer with a call to ClearCommError().
            // This call will reset the event handle, and if there are no bytes to read
            // we can loop back through WaitCommEvent() again, then proceed.
            // If there are really bytes to read, do nothing and proceed.

                bResult = ClearCommError(port->m_hComm, &dwError, &comstat);

                if (comstat.cbInQue == 0)
                    continue;
            }   // end if bResult

            // 下面是主等待函数，本函数将阻塞线程直到有相应事件发生
    /*******WaitForMultipleObjects 声明如下:
    DWORD WaitForMultipleObjects(
      DWORD nCount,           // 事件句柄数组中的事件数
      CONST HANDLE *lpHandles,  //事件数组：本类为 m_hEventArray 数据 ,
      BOOL fWaitAll,    // TRUE: 所有事件发生时才返回
                  //FALSE: 只要有指定的事件发生就返回
      DWORD dwMilliseconds      // 以 ms 为单位设置超时时间
    );
    */
    ////////////////////////////////
      //在本类中，3 个在事件数组中的事件如下
        //m_hEventArray[0] = m_hShutdownEvent;     // highest priority 关闭串口 event.
        //m_hEventArray[1] = m_ov.hEvent;     //读事件
        //m_hEventArray[2] = m_hWriteEvent;    //写事件
            Event = WaitForMultipleObjects(3, port->m_hEventArray, FALSE, INFINITE);

            switch (Event)
            {
            case 0:
                {
                    // Shutdown event. This is event zero so it will be
                    // the higest priority and be serviced first.
                    CloseHandle(port->m_hComm);
                    port->m_hComm=NULL;
                    port->m_bThreadAlive = FALSE;

                    // Kill this thread. break is not needed, but makes me feel better.
                    AfxEndThread(100);

                    break;
                }
    case 1:   // read event 在读事件中，将定义的各种消息发送出去.
    //这样在相关的消息处理函数 OnComm() 中就可以做出处理
    //注意，这里应用了 GetCommMask 函数来得到通信事件
                {
                    GetCommMask(port->m_hComm, &CommEvent);
                    if (CommEvent & EV_RXCHAR)
                        // Receive character event from port.
                        ReceiveChar(port, comstat);
                    if (CommEvent & EV_CTS)
                        ::SendMessage(port->m_pOwner->m_hWnd, WM_COMM_CTS_DETECTED,
(WPARAM) 0, (LPARAM) port->m_nPortNr);
                    if (CommEvent & EV_BREAK)
                        ::SendMessage(port->m_pOwner->m_hWnd, WM_COMM_BREAK_DETECTED,
(WPARAM) 0, (LPARAM) port->m_nPortNr);
```

```
                        if (CommEvent & EV_ERR)
                            ::SendMessage(port->m_pOwner->m_hWnd, WM_COMM_ERR_DETECTED,
(WPARAM) 0, (LPARAM) port->m_nPortNr);
                        if (CommEvent & EV_RING)
                            ::SendMessage(port->m_pOwner->m_hWnd, WM_COMM_RING_DETECTED,
(WPARAM) 0, (LPARAM) port->m_nPortNr);

                        if (CommEvent & EV_RXFLAG)
                            ::SendMessage(port->m_pOwner->m_hWnd, WM_COMM_RXFLAG_DETECTED,
(WPARAM) 0, (LPARAM) port->m_nPortNr);
                            break;
                    }
                case 2: // 发送数据 write event
                    {
                        // Write character event from port
                        WriteChar(port);
                        break;
                    }
                } // end switch
            } // close forever loop
            return 0;
        }
```

下面再来看看如何接收数据。

在上面的函数中，如果是 EV_RXCHAR 事件，就调用 ReceiveChar()函数，ReceiveChar 函数如下：

```
//
// Character received. Inform the owner
//
void CSerialPort::ReceiveChar(CSerialPort* port, COMSTAT comstat)
{
    BOOL bRead = TRUE;
    BOOL bResult = TRUE;
    DWORD dwError = 0;
    DWORD BytesRead = 0;
    unsigned char RXBuff;

    for (;;)
    {
        // critical section 能保证在同一时间内只有一个线程有操作某一资源的权限
// 下面应用了 EnterCriticalSection 函数来获得串口的 critical section
        // 这一过程可保证本程序（进程）中没有其他函数或线程来使用本串口资源
        EnterCriticalSection(&port->m_csCommunicationSync);

        // 下面使用 ClearCommError() 来更新 COMSTAT 结构，并清除通信错误
        bResult = ClearCommError(port->m_hComm, &dwError, &comstat);

        LeaveCriticalSection(&port->m_csCommunicationSync);

        // start forever loop. I use this type of loop because I
        // do not know at runtime how many loops this will have to
        // run. My solution is to start a forever loop and to
        // break out of it when I have processed all of the
        // data available. Be careful with this approach and
        // be sure your loop will exit.
        // My reasons for this are not as clear in this sample
        // as it is in my production code, but I have found this
        // solution to be the most efficient way to do this.
```

```
// 如果所有字符均被读出，就中断循环
            if (comstat.cbInQue == 0)
            {
                // break out when all bytes have been read
                break;
            }

            EnterCriticalSection(&port->m_csCommunicationSync);

            if (bRead)
            {
            //应用 ReadFile 函数读出缓冲区中的字节
                bResult = ReadFile(port->m_hComm,      // Handle to COMM port
                                   &RXBuff,                   // RX Buffer Pointer
                                   1,                     // Read one byte
                                   &BytesRead,            // Stores number of bytes read
                                   &port->m_ov);          // pointer to the m_ov
structure
                // 若 ReadFile 返回 FALSE，调用 GetLastError()错误处理
    //deal with the error code
                if (!bResult)
                {
                    switch (dwError = GetLastError())
                    {
                        case ERROR_IO_PENDING:
                            {
    // 异步 I/O 操作仍在进行，这时需要调用 GetOverlappedResults 函数来查询
                                //asynchronous i/o is still in progress
                                // Proceed on to GetOverlappedResults();
                                bRead = FALSE;
                                break;
                            }
                        default:
                            {   //其他错误处理
                                // Another error has occured.  Process this error.
                                port->ProcessErrorMessage("ReadFile()");
                                break;
                            }
                    }
                }
                else
                {
    //ReadFile()返回 TRUE，操作完成，这时不需调用 GetOverlappedResults()
                    bRead = TRUE;
                }
            } // close if (bRead)

    // 异步 I/O 操作仍在进行，这时需要调用 GetOverlappedResults 函数来查询
            if (!bRead)
            {
                bRead = TRUE;
                bResult = GetOverlappedResult(port->m_hComm,    // Handle to COMM port
                                              &port->m_ov, // Overlapped structure
                                              &BytesRead, // Stores number of bytes
read
                                              TRUE);            // Wait flag

                // deal with the error code
                if (!bResult)
```

```
                    {
                        port->ProcessErrorMessage("GetOverlappedResults() in ReadFile()");
                    }
                } // close if (!bRead)

                LeaveCriticalSection(&port->m_csCommunicationSync);

                // notify parent that a byte was received
                ::SendMessage((port->m_pOwner)->m_hWnd, WM_COMM_RXCHAR, (WPARAM) RXBuff,
(LPARAM) port->m_nPortNr);
            } // end forever loop

        }
```

5. 写数据：用 WriteFile 函数

在 CSerialPort 类中，写数据调用 WriteToPort 函数，通过 SetEvent 函数设置写事件，通知辅助线程有数据需要发送。

```
void CSerialPort::WriteToPort(char* string)
{
    assert(m_hComm != 0);
    memset(m_szWriteBuffer, 0, sizeof(m_szWriteBuffer));
    strcpy(m_szWriteBuffer, string);
    m_nWriteSize=strlen(string);
    // set event for write
    SetEvent(m_hWriteEvent);
}
```

在线程函数 **CommThread()**中，调用了 **WriteChar()**函数，其中，应用 WirteFile 函数发送数据。

```
void CSerialPort::WriteChar(CSerialPort* port)
{
    BOOL bWrite = TRUE;
    BOOL bResult = TRUE;

    DWORD BytesSent = 0;

    ResetEvent(port->m_hWriteEvent); //复位写事件句柄
    // Gain ownership of the critical section
    EnterCriticalSection(&port->m_csCommunicationSync);

    if (bWrite)
    {
        // Initailize variables
        port->m_ov.Offset = 0;
        port->m_ov.OffsetHigh = 0;

        // 清空缓冲区
        PurgeComm(port->m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT |
PURGE_TXABORT);

        bResult = WriteFile(port->m_hComm,              // Handle to COMM Port
                    port->m_szWriteBuffer,  // Pointer to message buffer in calling
finction
                    port->m_nWriteSize,     // Length of message to send
                    &BytesSent,         // Where to store the number of bytes sent
```

```
                    &port->m_ov); // Overlapped structure

            // 返回 FALSE，错误处理
            if (!bResult)
            {
                DWORD dwError = GetLastError();
                switch (dwError)
                {
                    case ERROR_IO_PENDING:
                        {
// 并非错误，而是异步 I/O 操作正在进行，需要调用 GetOverlappedResults() 查询结果
                            BytesSent = 0;
                            bWrite = FALSE;
                            break;
                        }
                    default:
                        {
                            // all other error codes
                            port->ProcessErrorMessage("WriteFile()");
                        }
                }
            }
            else
            {
                LeaveCriticalSection(&port->m_csCommunicationSync);
            }
        } // end if(bWrite)

        if (!bWrite)
        {
            bWrite = TRUE;
            bResult = GetOverlappedResult(port->m_hComm,    // Handle to COMM port
                                          &port->m_ov, // Overlapped structure
                                          &BytesSent,// Stores number of bytes sent
                                          TRUE);       // Wait flag
            LeaveCriticalSection(&port->m_csCommunicationSync);

        } // end if (!bWrite)

        ::SendMessage((port->m_pOwner)->m_hWnd, WM_COMM_TXEMPTY_DETECTED,0,(LPARAM)
port->m_nPortNr);

    }
```

6. 结束时关闭端口

若程序中打开了其他线程，则先终止线程，停止 WaitCommEvent 的等待以及关闭端口 CloseHandle，平时程序会停留在 WaitCommEvent 的等待中，当要终止线程的时候，必须是程序从 WaitCommEvent 中退出来，按照 Win32 手册上的说明，参数为 NULL 的 SetCommMask 会使另一个线程中的 WaitCommEvent 马上返回，然后用 CloseHandle 关闭端口。

在本类中，关闭串口在辅助线程事件处理时，设置了最高的优先级。在线程函数 **CommThread()**中实现。

4.6　Win32 API 串口编程 TTY（虚拟终端）实例

为了使读者更好地掌握本章的概念，这里再举一个具体实例来说明问题。这个程序不像 CSerialPort 类，没有写成可重用的代码，是一步一步编写过来的，好像更符合一般初学者的编程习惯，是一个编写得比较全面的实例《注：初始的源程序参考《Visual C++网络教程》，是一个简单的 TTY（虚拟终端）程序》。这个程序适用于当读者想编写自己的 API 程序时参考。包括硬件流控制的设置，因此可以用于对 MODEM 进行控制。

编程任务：

TTY 原意为电传打字机终端(在此称为虚拟终端)，顾名思义，程序的任务就是要做到像电传打字机一样，键盘上打出什么字符，就要把这些字符通过串口发送出去。

现在开始编程。

4.6.1　建立程序工程

用 AppWizard 建立一个名为 Term 的 MFC 应用程序。在 MFC AppWizard 对话框的第 1 步选择 Single document(单文档)，在第 4 步去掉 Docking toolbar 等项的选择。在第 6 步将 **CTermView 类的基类（Base Classes）更改为 CEditView。**

在 Term 工程的资源视图中打开 IDR_MAINFRAME 菜单资源。去掉 Edit 菜单和 View 菜单，并去掉 File 菜单中除 Exit 以外的所有菜单项。然后在 File 菜单中加入三个菜单项，如表 4-6-1 所示。

表 4-6-1　新菜单项

标题	ID
串口设置	ID_COMM_SETTINGS
串口连接	ID_COMM_CONNECT
断开串口	ID_COMM_DISCONNECT

用 ClassWizard 为 CTermDoc 类创建三个与上表菜单消息对应的 COMMAND 命令处理函数，使用缺省的函数名。为 ID_COMM_CONNECT 和 ID_COMM _DISCONNECT 命令创建命令更新处理函数。同时，用 ClassWizard 为该类加入 CanCloseFrame 成员函数。以上操作方法如图 4.6.1 所示。这些函数的处理代码稍后再来添加。

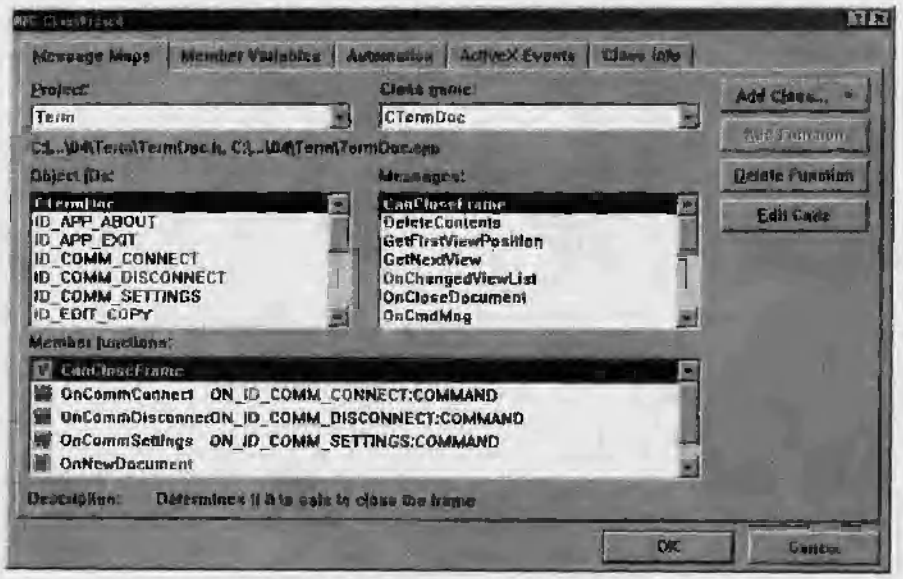


图 4.6.1　用 ClassWizard 添加菜单命令等函数

4.6.2　建立串口设置对话框

如图 4.6.2 所示方法，新建一个对话框模板资源，令其 ID 为 IDD_DIALOG_COMSETTINGS，Caption 为“串口设置”。请按图 4.6.2 所示设计对话框的控件。并按表 4-6-2 为这些控件设置属性或变量。

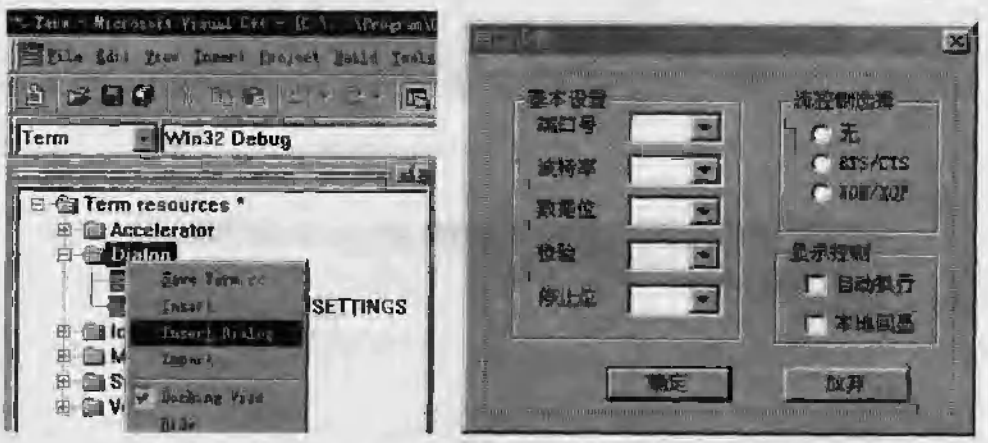


图 4.6.2　添加串口设置对话框

表 4-6-2　通信设置对话框中的主要控件

控件	ID	属性设置
基本设置组框	缺省	标题为基本设置
端口号组合框	IDC_PORT	Drop List，不选 Sort，初始列表为 COM1、COM2、COM3、COM4
波特率组合框	IDC_BAUD	Drop List，不选 Sort，初始列表为 300、600、1200、2400、4800、9600、14400、19200、38400、57600
数据位组合框	IDC_DATABITS	Drop List，不选 Sort，初列表为 5、6、7、8
奇偶校验组合框	IDC_PARITY	Drop List，不选 Sort，初列表为 None、Even、Odd
停止位组合框	IDC_STOPBITS	Drop List，不选 Sort，初列表为 1、1.5、2
流控制选择组框	缺省	标题为流控制选择
None 单选按钮	IDC_FLOWCTRL	标题为 None，选择 Group 属性
RTS/CTS 单选按钮	缺省	标题为 RTS/CTS
XON/XOFF 单选按钮	缺省	标题为 XON/XOFF
显示控制组框	缺省	标题为显示控制
自动换行检查框	IDC_NEWLINE	标题为自动换行
本地回显检查框	IDC_ECHO	标题为本地回显

下面为 IDD_DIALOG_COMSETTINGS 模板创建一个名为 CSetupDlg 的对话框类。打开 ClassWizard，出现图 4.6.3 所示的提示。选择 Create a new class（建立一个新类）。单击 OK，出现如图 4.6.4 所示的“New Class”对话框，在 Name（类名）中填入“CsetupDlg”。

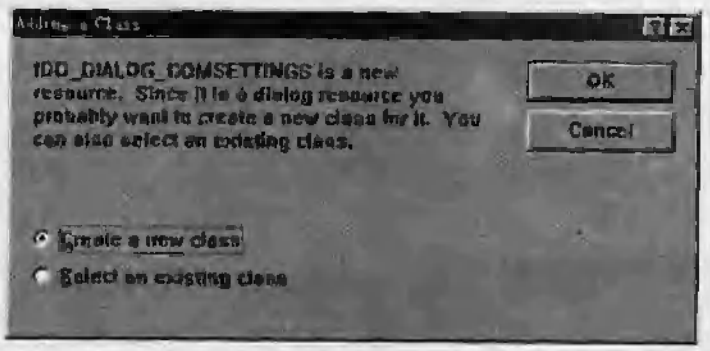


图 4.6.3　ClassWizard 提示为 IDD_DIALOG_COMSETTINGS 建立类

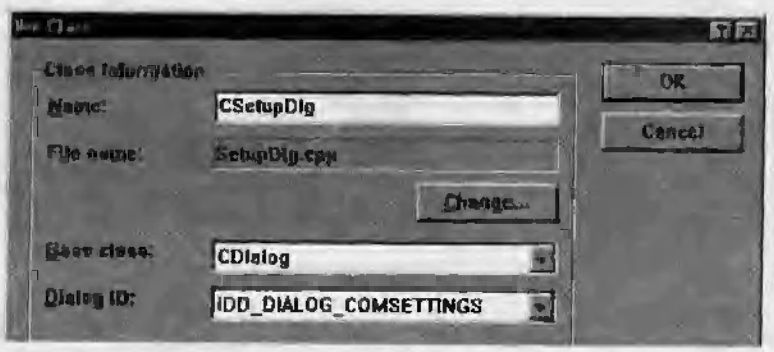


图 4.6.4　为新类命名

接着，打开 ClassWizard，按图 4.6.5 中所示为各控件添加变量。

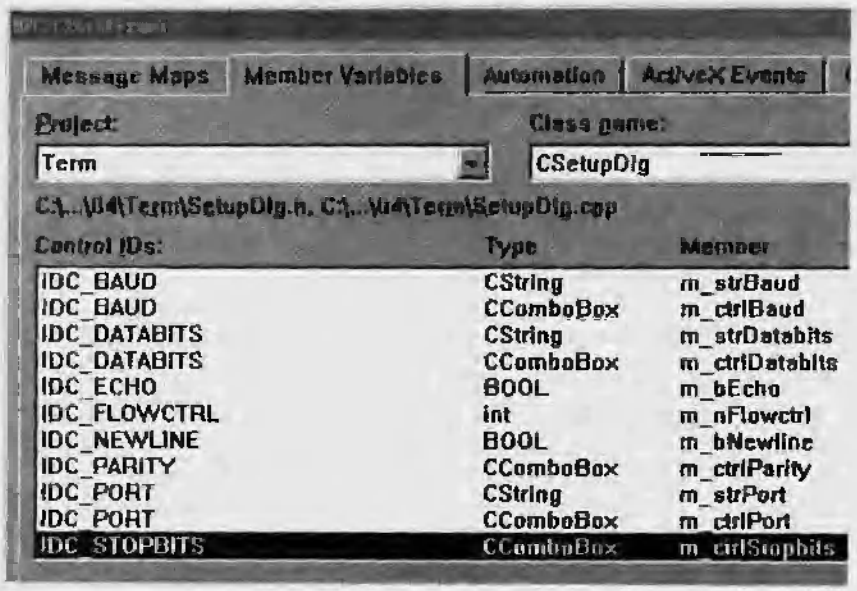


图 4.6.5　在 CSetupDlg 类中为控件添加变量

再在 ClassWizard 中为 CSetupDlg 类添加为 OnInitDialog 成员函数（如图 4.6.6 所示），并加入以下代码：

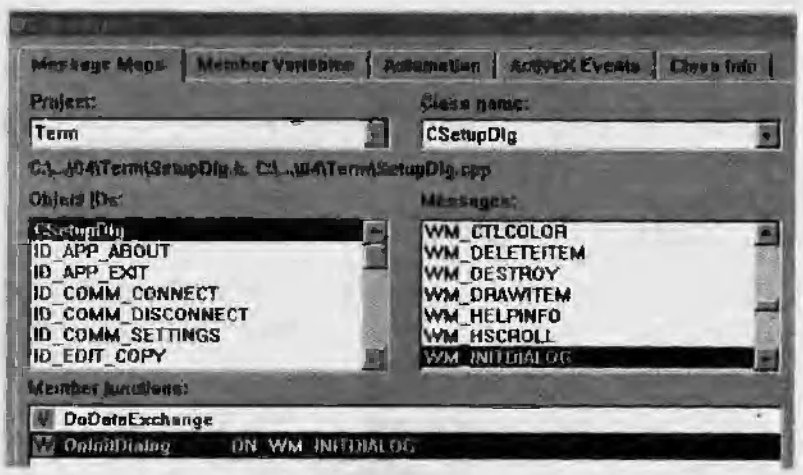


图 4.6.6　在 CSetupDlg 类中添加 OnInitDialog 成员函数

```
BOOL CSetupDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // TODO: Add extra initialization here
    //对对话框的显示情况进行初始化
```

```
    m_ctrlPort.AddString(_T("COM1"));
    m_ctrlPort.AddString(_T("COM2"));
    m_ctrlPort.AddString(_T("COM3"));
    m_ctrlPort.AddString(_T("COM4"));

    m_ctrlParity.AddString(_T("NONE"));
    m_ctrlParity.AddString(_T("EVEN"));
    m_ctrlParity.AddString(_T("ODD"));
    m_ctrlParity.SetCurSel(m_nParity);

    m_ctrlDatabits.AddString(_T("5"));
    m_ctrlDatabits.AddString(_T("6"));
    m_ctrlDatabits.AddString(_T("7"));
    m_ctrlDatabits.AddString(_T("8"));

    m_ctrlBaud.AddString(_T("300"));
    m_ctrlBaud.AddString(_T("600"));
    m_ctrlBaud.AddString(_T("1200"));
    m_ctrlBaud.AddString(_T("2400"));
    m_ctrlBaud.AddString(_T("4800"));
    m_ctrlBaud.AddString(_T("9600"));
    m_ctrlBaud.AddString(_T("14400"));
    m_ctrlBaud.AddString(_T("19200"));
    m_ctrlBaud.AddString(_T("38400"));
    m_ctrlBaud.AddString(_T("57600"));

    m_ctrlStopbits.AddString(_T("1"));
    m_ctrlStopbits.AddString(_T("1.5"));
    m_ctrlStopbits.AddString(_T("2"));
    m_ctrlStopbits.SetCurSel(m_nStopBits);
    GetDlgItem(IDC_PORT)->EnableWindow(!m_bConnected);
    UpdateData(FALSE);
    return TRUE;  // return TRUE unless you set the focus to a control
              // EXCEPTION: OCX Property Pages should return FALSE
}
```

CSetupDlg 的主要任务是配置通信参数。在 OnInitDialog 函数中，要根据当前是否连接来允许/禁止 Port 组合框。因为在打开一个连接后，显然不能随便改变端口。

4.6.3　编写 CTermDoc 类的相关代码

CTermDoc 类是本程序的重点，请读者认真体会。该类负责 Term 的通信任务，主要包括设置通信参数、打开和关闭串口、建立和终止辅助工作线程、用辅助线程监视串口等。

在 TermDoc.h 中添加下列变量：

```
CWinThread* m_pThread;  // 代表辅助线程
volatile BOOL m_bConnected;  //串口是否连接
volatile HWND m_hTermWnd;    //保存视图的窗口句柄
volatile HANDLE m_hPostMsgEvent; // 用于WM_COMMNOTIFY消息的事件对象
OVERLAPPED m_osRead, m_osWrite; // 用于重叠读/写
volatile HANDLE m_hCom; // 串行口句柄
int m_nBaud;    //波特率
int m_nDataBits; //停止位
int m_nParity;  //校验位
CString m_strPort;    //串口号
int m_nStopBits; //停止位
int m_nFlowCtrl; //流控制选项
```

```
BOOL m_bEcho;    //是否在本地回显
BOOL m_bNewLine; //是否自动换行
```

上面有些变量是用 volatile 关键字声明的，这是为下面建立辅助线程准备的。当两个线程都要用到某一个变量且该变量的值会被改变时，应该用 volatile 声明，该关键字的作用是防止优化编译器把变量从内存装入 CPU 寄存器中。如果变量被装入寄存器，那么两个线程有可能一个使用内存中的变量，一个使用寄存器中的变量，这会造成程序的错误执行。

成员 m_bConnected 用来表明当前是否存在一个通信连接。m_hTermWnd 用来保存视图的窗口句柄。m_hPostMsgEvent 事件对象用于 WM_COMMNOTIFY 消息的允许和禁止。m_pThread 用来指向 AfxBeginThread 创建的 CWinThread 对象，以便对线程进行控制。OVERLAPPED 结构 m_osRead 和 m_osWrite 用于串口的重叠读/写，程序应该为它们的 hEvent 成员创建事件句柄。

接着，为三个菜单命令函数添加相应的代码：

```
//串口联接
void CTermDoc::OnCommConnect()
{
    // TODO: Add your command handler code here
    if(!OpenConnection())
        AfxMessageBox("无法建立串口联接");
}

//断开串口联接
void CTermDoc::OnCommDisconnect()
{
    // TODO: Add your command handler code here
    CloseConnection();
}

//串口设置
void CTermDoc::OnCommSettings()
{
    // TODO: Add your command handler code here
    CSetupDlg dlg;
    CString str;
    dlg.m_bConnected = m_bConnected;
    dlg.m_strPort = m_strPort;

    str.Format("%d", m_nBaud);
    dlg.m_strBaud = str;
    str.Format("%d", m_nDataBits);
    dlg.m_strDatabits = str;
    dlg.m_nParity = m_nParity;
    dlg.m_nStopBits = m_nStopBits;
    dlg.m_nFlowctrl = m_nFlowCtrl;
    dlg.m_bEcho = m_bEcho;
    dlg.m_bNewline = m_bNewLine;
    if(dlg.DoModal() == IDOK)
    {
        m_strPort = dlg.m_strPort;
        m_nBaud = atoi(dlg.m_strBaud);
        m_nDataBits = atoi(dlg.m_strDatabits);
        m_nParity = dlg.m_nParity;
        m_nStopBits = dlg.m_nStopBits;
        m_nFlowCtrl = dlg.m_nFlowctrl;
```

```
            m_bEcho=dlg.m_bEcho;
            m_bNewLine=dlg.m_bNewline;
            if(m_bConnected)
                if(!ConfigConnection())
                    AfxMessageBox("无法按指定的参数设置串口!");
        }
    }

    //UPDATE_COMMAND_UI 断开串口联接命令更新函数，设置命令是否可用
    void CTermDoc::OnUpdateCommDisconnect(CCmdUI* pCmdUI)
    {
        // TODO: Add your command update UI handler code here
        pCmdUI->Enable(m_bConnected);
    }

    //UPDATE_COMMAND_UI 串口联接命令更新函数，设置命令是否可用
    void CTermDoc::OnUpdateCommConnect(CCmdUI* pCmdUI)
    {
        // TODO: Add your command update UI handler code here
        pCmdUI->Enable(!m_bConnected);
    }
```

接着，建立读/写函数 ReadComm 和 WriteComm：

```
    // 将指定数量的字符从串行口输出
    DWORD CTermDoc::WriteComm(char *buf, DWORD dwLength)
    {
        BOOL fState;
        DWORD length=dwLength;
        COMSTAT ComStat;
        DWORD dwErrorFlags;

        ClearCommError(m_hCom,&dwErrorFlags,&ComStat);
        fState=WriteFile(m_hCom,buf,length,&length,&m_osWrite);
        if(!fState)
        {
            if(GetLastError()==ERROR_IO_PENDING)
            {
                GetOverlappedResult(m_hCom,&m_osWrite,&length,TRUE);// 等待
            }
            else
                length=0;
        }
        return length;
    }

    // 从串行口输入缓冲区中读入指定数量的字符
    DWORD CTermDoc::ReadComm(char *buf, DWORD dwLength)
    {
        DWORD length=0;
        COMSTAT ComStat;
        DWORD dwErrorFlags;
        ClearCommError(m_hCom,&dwErrorFlags,&ComStat);
        length=min(dwLength, ComStat.cbInQue);
        ReadFile(m_hCom,buf,length,&length,&m_osRead);
        return length;
    }
```

在以下的函数调用中，分别用到了下列函数，先添加相应函数，并写入相关代码：

```
// 打开并配置串口，建立辅助线程
BOOL CTermDoc::OpenConnection()
{
    COMMTIMEOUTS TimeOuts;
    POSITION firstViewPos;
    CView *pView;

    firstViewPos=GetFirstViewPosition();
    pView=GetNextView(firstViewPos);

    m_hTermWnd=pView->GetSafeHwnd();

    if(m_bConnected)
        return FALSE;

    m_hCom=CreateFile(m_strPort,
        GENERIC_READ | GENERIC_WRITE,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, // 重叠方式
        NULL);

    if(m_hCom==INVALID_HANDLE_VALUE)
        return FALSE;

    SetupComm(m_hCom,MAXBLOCK,MAXBLOCK);
    SetCommMask(m_hCom, EV_RXCHAR);


    // 把间隔超时设为最大，把总超时设为 0 将导致 ReadFile 立即返回并完成操作
    TimeOuts.ReadIntervalTimeout=MAXDWORD;
    TimeOuts.ReadTotalTimeoutMultiplier=0;
    TimeOuts.ReadTotalTimeoutConstant=0;

    /* 设置写超时以指定 WriteComm 成员函数中的 GetOverlappedResult 函数的等待时间*/
    TimeOuts.WriteTotalTimeoutMultiplier=50;
    TimeOuts.WriteTotalTimeoutConstant=2000;

    SetCommTimeouts(m_hCom, &TimeOuts);

    if(ConfigConnection())
    {

        m_pThread=AfxBeginThread(CommProc, this, THREAD_PRIORITY_NORMAL,
            0, CREATE_SUSPENDED, NULL); // 创建并挂起线程

        if(m_pThread==NULL)
        {
            CloseHandle(m_hCom);
            return FALSE;
        }
        else

        {
            m_bConnected=TRUE;
            m_pThread->ResumeThread(); // 恢复线程运行
        }
    }
    else
```

```
    {
        CloseHandle(m_hCom);
        return FALSE;
    }

    return TRUE;
}

//配置串口连接
BOOL CTermDoc::ConfigConnection()
{
    DCB dcb;

    if(!GetCommState(m_hCom, &dcb))
        return FALSE;

    dcb.fBinary=TRUE;
    dcb.BaudRate = m_nBaud; // 数据传输速率
    dcb.ByteSize = m_nDataBits; // 每字节位数
    dcb.fParity = TRUE;

    switch(m_nParity) // 校验设置
    {
    case 0:
        dcb.Parity=NOPARITY;
        break;
    case 1:
        dcb.Parity=EVENPARITY;
        break;
    case 2:
        dcb.Parity=ODDPARITY;
        break;
    default:;
    }

    switch(m_nStopBits) // 停止位
    {
    case 0:
        dcb.StopBits=ONESTOPBIT;
        break;

    case 1:
        dcb.StopBits=ONE5STOPBITS;
        break;

    case 2:
        dcb.StopBits=TWOSTOPBITS;
        break;

    default:;
    }

    switch(m_nFlowCtrl)  //流控制选项设置
    {
    case 0:
        // 硬件流控制设置
        dcb.fOutxCtsFlow = FALSE;
        dcb.fRtsControl = FALSE;
        // XON/XOFF 流控制设置
        dcb.fInX=dcb.fOutX = FALSE;
```

```
            //dcb.XonChar = XON;
            //dcb.XoffChar = XOFF;
            ////dcb.XonLim = 50;
            //dcb.XoffLim = 50;
            break;
        case 1:
            // 硬件流控制设置
            dcb.fOutxCtsFlow = TRUE;
            dcb.fRtsControl = TRUE;

            // XON/XOFF 流控制设置
            dcb.fInX=dcb.fOutX = FALSE;
            break;
        case 2:
            // 软件流控制设置
            dcb.fOutxCtsFlow = FALSE;
            dcb.fRtsControl = FALSE;
            // XON/XOFF 流控制设置
            dcb.fInX=dcb.fOutX = TRUE;
            dcb.XonChar = XON;
            dcb.XoffChar = XOFF;
            dcb.XonLim = 50;
            dcb.XoffLim = 50;
            break;
        default:
            break;
        }
        return SetCommState(m_hCom, &dcb);
    }

    //关闭连接，关闭辅助线程
    void CTermDoc::CloseConnection()
    {
        if(!m_bConnected)
            return;

        m_bConnected=FALSE;

        //结束 CommProc 线程中 WaitSingleObject 函数的等待
        SetEvent(m_hPostMsgEvent);

        //结束 CommProc 线程中 WaitCommEvent 的等待
        SetCommMask(m_hCom, 0);

        //等待辅助线程终止
        WaitForSingleObject(m_pThread->m_hThread, INFINITE);
        m_pThread=NULL;
        CloseHandle(m_hCom);
    }
```

在 CTermDoc 中，要添加一个辅助线程用来接收数据，辅助线程通过发送该消息来通知视图有通信事件发生。要注意的是，这是一个全局函数：

```
    // 辅助线程，负责监视串行口
    UINT CommProc(LPVOID pParam)
    {
        OVERLAPPED os;
        DWORD dwMask, dwTrans;
        COMSTAT ComStat;
```

```
    DWORD dwErrorFlags;

    CTermDoc *pDoc=(CTermDoc*)pParam;

    memset(&os, 0, sizeof(OVERLAPPED));
    os.hEvent=CreateEvent(NULL, TRUE, FALSE, NULL);

    if(os.hEvent==NULL)
    {
        AfxMessageBox("无法创建事件对象!");
        return (UINT)-1;
    }

    while(pDoc->m_bConnected)
    {
        ClearCommError(pDoc->m_hCom,&dwErrorFlags,&ComStat);

        if(ComStat.cbInQue)
        {
            // 无限等待 WM_COMMNOTIFY 消息被处理完
            WaitForSingleObject(pDoc->m_hPostMsgEvent, INFINITE);
            ResetEvent(pDoc->m_hPostMsgEvent);

            // 通知视图
            PostMessage(pDoc->m_hTermWnd, WM_COMMNOTIFY, EV_RXCHAR, 0);

            continue;
        }

        dwMask=0;

        if(!WaitCommEvent(pDoc->m_hCom, &dwMask, &os)) // 重叠操作
        {

            if(GetLastError()==ERROR_IO_PENDING)
            // 无限等待重叠操作结果
                GetOverlappedResult(pDoc->m_hCom, &os, &dwTrans, TRUE);
            else
            {
                CloseHandle(os.hEvent);
                return (UINT)-1;
            }
        }
    }
    CloseHandle(os.hEvent);
    return 0;
}
```

OnNewDocument 成员函数创建了三个事件对象：

```
BOOL CTermDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    ((CEditView*)m_viewList.GetHead())->SetWindowText(NULL);

    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)
    // 为 WM_COMMNOTIFY 消息创建事件对象，手工重置，初始化为有信号的
```

```
        if((m_hPostMsgEvent=CreateEvent(NULL, TRUE, TRUE, NULL))==NULL)
            return FALSE;

        memset(&m_osRead, 0, sizeof(OVERLAPPED));
        memset(&m_osWrite, 0, sizeof(OVERLAPPED));

        // 为重叠读创建事件对象，手工重置，初始化为无信号的
        if((m_osRead.hEvent=CreateEvent(NULL, TRUE, FALSE, NULL))==NULL)
            return FALSE;

        // 为重叠写创建事件对象，手工重置，初始化为无信号的
        if((m_osWrite.hEvent=CreateEvent(NULL, TRUE, FALSE, NULL))==NULL)
            return FALSE;


        return TRUE;
    }
```

注意：不要忘记在 CTermDoc 的构造函数中初始化有关变量，并在析构函数关闭串口并删除事件对象句柄。

```
    CTermDoc::CTermDoc()
    {
        // TODO: add one-time construction code here
        m_bConnected=FALSE;//断开连接菜单项无效
        m_pThread=NULL;

        m_nBaud = 9600;
        m_nDataBits = 8;
        m_bEcho = TRUE;   //初始设置为本地回显
        m_bNewLine = TRUE;  //初始设置为自动换行
        m_nParity = 0;     //无奇偶校验
        m_strPort = "COM2";  //选择 COM2
        m_nStopBits = 0;    //
        m_nFlowCtrl=0;

    }

    CTermDoc::~CTermDoc()
    {
        //程序结束时删除线程、关闭串口的操作
        if(m_bConnected)
            CloseConnection();

        // 删除事件句柄
        if(m_hPostMsgEvent)
            CloseHandle(m_hPostMsgEvent);

        if(m_osRead.hEvent)
            CloseHandle(m_osRead.hEvent);

        if(m_osWrite.hEvent)
            CloseHandle(m_osWrite.hEvent);
```

以上所有操作完成后，**TermDoc.h** 文件代码如下：

```
// TermDoc.h : interface of the CTermDoc class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_TERMDOC_H__957DC8EA_654F_11D8_870F_00E04C3F78CA__INCLUDED_)
#define AFX_TERMDOC_H__957DC8EA_654F_11D8_870F_00E04C3F78CA__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

//定义最大数据块、XON 字符和 XOFF 字符
#define MAXBLOCK 4096
#define XON 0x11
#define XOFF 0x13

class CTermDoc : public CDocument
{
protected: // create from serialization only
    CTermDoc();
    DECLARE_DYNCREATE(CTermDoc)

// Attributes
public:
    CWinThread* m_pThread; // 代表辅助线程
    volatile BOOL m_bConnected; //串口是否连接
    volatile HWND m_hTermWnd;   //保存视图的窗口句柄
    volatile HANDLE m_hPostMsgEvent; // 用于 WM_COMMNOTIFY 消息的事件对象
    OVERLAPPED m_osRead, m_osWrite; // 用于重叠读/写
    volatile HANDLE m_hCom; // 串行口句柄
    int m_nBaud;    //波特率
    int m_nDataBits; //停止位
    int m_nParity;  //校验位
    CString m_strPort;    //串口号
    int m_nStopBits; //停止位
    int m_nFlowCtrl; //流控制选项
    BOOL m_bEcho;   //是否在本地回显
    BOOL m_bNewLine; //是否自动换行

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CTermDoc)
    public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    virtual BOOL CanCloseFrame(CFrameWnd* pFrame);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CTermDoc();
    void CloseConnection();  //关闭连接
    BOOL ConfigConnection(); //配置串口通信参数
    BOOL OpenConnection();   //建立连接
```

```
        DWORD ReadComm(char *buf,DWORD dwLength);//读串口
        DWORD WriteComm(char *buf,DWORD dwLength);//写串口

    #ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
    #endif

    protected:

    // Generated message map functions
    protected:
        //{{AFX_MSG(CTermDoc)
        afx_msg void OnCommConnect();
        afx_msg void OnCommDisconnect();
        afx_msg void OnCommSettings();
        afx_msg void OnUpdateCommDisconnect(CCmdUI* pCmdUI);
        afx_msg void OnUpdateCommConnect(CCmdUI* pCmdUI);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
    };

    /////////////////////////////////////////////////////////////////////////////

    //{{AFX_INSERT_LOCATION}}
    // Microsoft Visual C++ will insert additional declarations immediately before the previous
line.

    #endif
    // !defined(AFX_TERMDOC_H__957DC8EA_654F_11D8_870F_00E04C3F78CA__INCLUDED_)
```

4.6.4　小结

下面再对 CTermDoc 类的编程思路做一个总结说明。

CTermDoc 类的构造函数主要完成一些通信参数的初始化工作。OnNewDocument 成员函数创建了三个事件对象，CTermDoc 的析构函数关闭串口并删除事件对象句柄。

OnFileSettings 是 File->Settings...的命令处理函数，该函数弹出一个 CSetupDlg 对话框来设置通信参数。实际的设置工作由 ConfigConnection 函数完成，在 OpenConnection 和 OnFileSettings 中都会调用该函数。

OpenConnection 负责打开串口并建立辅助工作线程，当用户选择了 File->Connect 命令时，消息处理函数 OnFileConnect 将调用该函数。该函数调用 CreateFile 以重叠方式打开指定的串口，并把返回的句柄保存在 m_hCom 成员中。接着，函数对 m_hCom 通信设备进行各种设置。需要注意的是对超时的设定，将读间隔超时设置为 MAXDWORD 并使其他读超时参数为 0 会导致 ReadFile 函数立即完成操作并返回，而不管读入了多少字符。设置超时就规定了 GetOverlappedResult 函数的等待时间，因此有必要将写超时设置成适当的值，这样，如果不能完成写串口的任务，GetOverlappedResult 函数会在超过规定超时后结束等待并报告实际传输的字符数。

如果对 m_hCom 设置成功，则函数会建立一个辅助线程并暂时将其挂起。在最后，调用 CWinThread:: ResumeThread 使线程开始运行。

OpenConnection 调用成功后，线程函数 CommProc 就开始工作。该函数的主体是一个 while 循环，在该循环内，混合了两种方法监视串口输入的方法。先是调用 ClearCommError 函数查询输入缓冲区中是否有字符，如果有，就向视图发送 WM_COMMNOTIFY 消息通知其接收字符。如果没有，则调用 WaitCommEvent 函数监视 EV_RXCHAR 通信事件，该函数执行重叠操作，紧接着调用的 GetOverlappedResult 函数无限等待通信事件，如果 EV_RXCHAR 事件发生（串口收到字符并放入输入缓冲区中），那么函数就结束等待。

上述两种方法的混合使用兼顾了线程的效率和可靠性。如果只用 ClearCommError 函数，则辅助线程将不断耗费 CPU 时间来查询，效率较低。如果只用 WaitCommEvent 来监视，那么由于该函数对输入缓冲区中已有的字符不会产生 EV_RXCHAR 事件，因此，在通信速率较高时，会造成数据的延误和丢失。

注意到辅助线程用 m_PostMsgEvent 事件对象来同步 WM_COMMNOTIFY 消息的发送。在发送消息之前，WaitForSingleObject 函数无限等待 m_PostMsgEvent 对象，WM_COMMNOTIFY 的消息处理函数 CTermView::OnCommNotify 在返回时会把该对象置为有信号，因此，如果 WaitForSingleObject 函数返回，则说明上一个 WM_COMMNOTIFY 消息已被处理完，这时才能发下一个消息，在发消息前还要调用 ResetEvent 把 m_PostMsgEvent 对象置为无信号的，以供下次使用。

由于 PostMessage 函数在消息队列中放入消息后会立即返回，所以，如果不采取上述措施，那么辅助线程可能在主线程未处理之前重复发出 WM_COMMNOTIFY 消息，这会降低系统的效率。

可能有读者会问，为什么不用 SendMessage？该函数在发送的消息被处理完毕后才返回，这样不就不用考虑同步问题了吗？是的，本例中也可以使用 SendMessage，但该函数会阻塞辅助线程的执行，直到消息处理完毕，这会降低效率。如果用 PostMessage，那么在函数立即返回后，线程还可以干别的事情，因此，考虑到效率问题，这里使用了 PostMessage 而不是 SendMessage。

函数 ReadComm 和 WriteComm 分别用来从 m_hCom 通信设备中读/写指定数量的字符。ReadComm 函数很简单，由于对读超时的特殊设定，ReadFile 函数会立即返回并完成操作，并在 length 变量中报告实际读入的字符数。此时，没有必要调用等待函数或 GetOverlappedResult。在 WriteComm 中，调用 GetOverlappedResult 来等待操作结果，直到超时发生。不管是否超时，该函数在结束等待后都会报告实际的传输字符数。

CloseConnection 函数的主要任务是终止辅助线程并关闭 m_hCom 通信设备。为了终止线程，该函数设置了一系列信号，以结束辅助线程中的等待和循环，然后调用 WaitForSingleObject 等待线程结束。

4.6.5 在 CTermView 类中字添加符键入处理代码与串口接收处理代码

如图 4.6.7 所示，用 ClassWizard 为 CTermView 类创建 OnChar 函数，该函数用来把用户键入的字符向串口输出。该函数用来获取键盘键入的字符。同时向串口输出该字符。

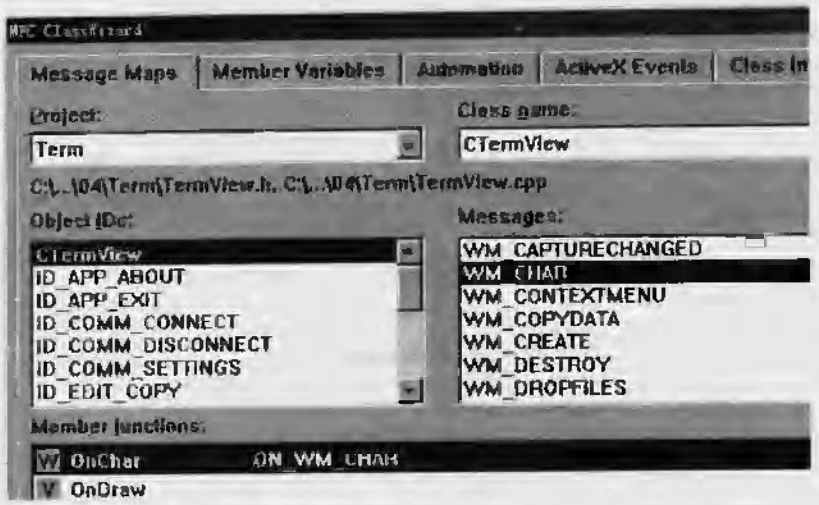


图 4.6.7　为 CTermView 添加 OnChar 函数

```
void CTermView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
    CTermDoc* pDoc=GetDocument();
    char c=(char)nChar;
    if(!pDoc->m_bConnected)
    {
        AfxMessageBox("串口没有联接");
        return;
    }
    pDoc->WriteComm(&c, 1);
    if(pDoc->m_bEcho)
        CEditView::OnChar(nChar, nRepCnt, nFlags); // 本地回显
}
```

下面再为串口事件消息添加处理函数 OnComm()。首先在 CTermView.h 中添加如下代码：

```
// Generated message map functions
protected:
    //{{AFX_MSG(CTermView)
    afx_msg LRESULT OnComm(WPARAM wParam, LPARAM lParam);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
```

再在 CTermView.cpp 中做消息映射：

```
BEGIN_MESSAGE_MAP(CTermView, CEditView)
    //{{AFX_MSG_MAP(CTermView)
    ON_MESSAGE(WM_COMMNOTIFY, OnComm)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

再添加 OnComm()函数实现代码：

```
LRESULT CTermView::OnComm(WPARAM wParam, LPARAM lParam)
{
```

```
        char buf[MAXBLOCK/4];
        CString str;
        int nLength, nTextLength;
        CTermDoc* pDoc=GetDocument();
        CEdit& edit=GetEditCtrl();
        if(!pDoc->m_bConnected || (wParam & EV_RXCHAR)!=EV_RXCHAR) // 是否是 EV_RXCHAR 事件?
        {
            SetEvent(pDoc->m_hPostMsgEvent); // 允许发送下一个 WM_COMMNOTIFY 消息
            return 0L;
        }
        nLength=pDoc->ReadComm(buf,100);
        if(nLength)
        {
            nTextLength=edit.GetWindowTextLength();
            edit.SetSel(nTextLength,nTextLength); //移动插入光标到正文末尾
            for(int i=0;i<nLength;i++)
            {
                switch(buf[i])
                {
                case '\r': // 回车
                    if(!pDoc->m_bNewLine)
                        break;
                case '\n': // 换行
                    str+="\r\n";
                    break;
                case '\b': // 退格
                    edit.SetSel(-1, 0);
                    edit.ReplaceSel(str);
                    nTextLength=edit.GetWindowTextLength();
                    edit.SetSel(nTextLength-1,nTextLength);
                    edit.ReplaceSel("");  //回退一个字符
                    str="";
                    break;
                case '\a': // 振铃
                    MessageBeep((UINT)-1);
                    break;
                default :
                    str+=buf[i];
                }
            }
            edit.SetSel(-1, 0);
            edit.ReplaceSel(str); // 向编辑视图中插入收到的字符
        }
        SetEvent(pDoc->m_hPostMsgEvent); // 允许发送下一个 WM_COMMNOTIFY 消息
        return 0L;
    }
```

要注意的是，以上 OnComm()函数实现均需要手工完成。

以上操作完成后，CTermView.h 文件的完整代码如下：

```
// TermView.h : interface of the CTermView class
//
……
class CTermView : public CEditView
{
protected: // create from serialization only
    CTermView();
    DECLARE_DYNCREATE(CTermView)
// Attributes
```

```
public:
    CTermDoc* GetDocument();
// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CTermView)
    public:
    virtual void OnDraw(CDC* pDC);  // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CTermView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:
// Generated message map functions
protected:
    //{{AFX_MSG(CTermView)
    afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg LRESULT OnComm(WPARAM wParam, LPARAM lParam);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
#ifndef _DEBUG  // debug version in TermView.cpp
inline CTermDoc* CTermView::GetDocument()
  { return (CTermDoc*)m_pDocument; }
#endif
/////////////////////////////////////////////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous
line.
#endif
// !defined(AFX_TERMVIEW_H__957DC8EC_654F_11D8_870F_00E04C3F78CA__INCLUDED_)
```

CTermView 是 CEditView 的派生类，利用 CEditView 的编辑功能，可以大大简化程序的设计。

OnChar 函数对 WM_CHAR 消息进行处理，它调用 CTermDoc::WriteComm 把用户键入的字符从串口输出。如果设置了 Local echo，那么就调用 CEditView::OnChar 把字符输出到视图中。

OnCommNotify 是 WM_COMMNOTIFY 消息的处理函数。该函数调用 CTermDoc::ReadComm 从串口输入缓冲区中读入字符并把它们输出到编辑视图中。在输出前，函数会对一些特殊字符进行处理。在函数返回时，要调用 SetEvent 把 m_hPostMsgEvent 置为有信号。

至此，现在所有的代码都写完了，下面来测试程序。

连接好串口线，注意，如果用的是三线制（非 MODEM）连接，在测试时，串口配置就将流控制选项选为“无”。否则字符发不出去。

运行程序后，单击“串口操作”→“串口设置”，将 Term 程序设置为如图 4.6.8 所示的配置。即以 Term 程序控制 COM1。单击“确定”后，再建立串口连接：“串口操作”→“串口连接”。

图 4.6.8　串口设置对话框

同时，打开串口调试助手，选择 COM1。现在可以在 Term 终端仿真程序中键入一些字符，可以看到，串口调试助手的接收框中显示了同样的字符。单击串口调试助手的“手动发送”按钮，也可以看到 Term 程序中也能接收到发来的字符，如图 4.6.9 所示。

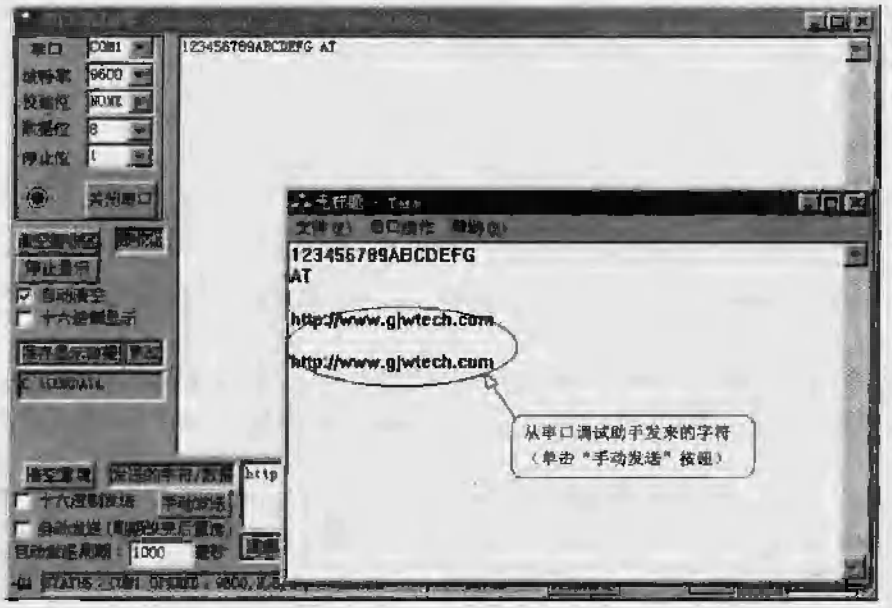


图 4.6.9　Term 终端仿真程序测试

第 5 章　串口调试助手 V2.2 编程

[内容提要]

本章介绍串口调试助手的详细编程过程，自从串口调试助手作为免费调试工具提供给工程技术人员使用后，许多朋友向作者索取了源代码。现在将其源代码放在这里，并写出详细编程过程。有兴趣的读者可以根据自己的需要对程序进行进行某些功能的改造，拥有自己的专门测试程序也是一件非常自豪的事情。

关于串口调试助手的使用方法请参看第 1 章，本章中不做介绍。

在本章中，还涉及到在串口编程过程中的许多问题：即如何发送与接收十六进制数等一些问题，以及 VC 编程中的一些常用方法，如数据显示、保存等，就是 VC 的初学者，也能从中领会一些 VC 编程的技巧，帮助 VC 不太熟悉的读者完成串口通信编程也是本书的目的之一。

这里介绍的串口调试助手中，我们用改进的 CSerialPort 类（见第 2.4 节）来编写程序。在本章的程序中，没有描述放大至全屏这一功能，其他功能与串口调试助手 V2.2 相同。

5.1　建立 SCOMM 程序工程实现界面功能

1. 建立基于对话框的程序工程 SCOMM

打开 Visual C++ 6.0，用 MFC AppWizard 建立一个名为 SCOMM 的 MFC 应用程序，S 是 Serial（串行）的字头，COMM 则是 communication(通信)的缩写。在 AppWizard 对话框的第 1 步选择 Dialog based(对话框)，其余依照缺省设置即可，单击“完成”。

2. 在对话框中添加控件并设置控件属性

串口调试助手的界面分为三个区：接收区、发送区和其他一些扩展功能区。首先按图 5.1.1 所示的界面添加控件。

由于控件较多，有些控件的属性没有做详细说明，读者可以通过 VC 编程环境打开程序工程，从对话框资源模板中查看。下面按功能说明各按件及其属性，注意变量前带有 ctrl 的为控制型变量。

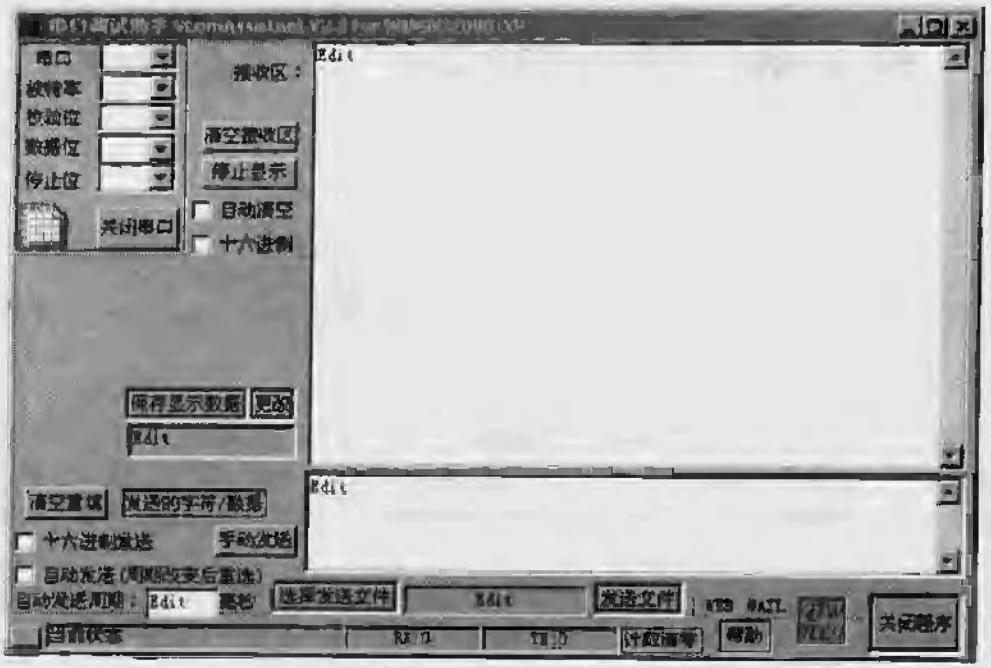


图 5.1.1　串口调试助手程序界面控件

➢　接收区

串口号组合框 IDC_COMBO_COMSELECT, m_Com

波特率组合框 IDC_COMBO_SPEED, m_Speed

停止位组合框 IDC_COMBO_STOPBITS, m_StopBits

数据位组合框 IDC_COMBO_DATABITS, m_DataBits

停止位组合框 IDC_BUTTON_STOPDISP, m_ctrlStopDisp

校验位组合框 IDC_COMBO_PARITY, m_Parity

十六制显示（接收）IDC_CHECK_HEXRECIEVE, m_ctrlHexReceieve

接收编辑框　IDC_EDIT_RECIVE, m_ReceiveData　m_ctrlReceiveData Style: Vertical Scroll，MultiLine

打开串口　IDC_BUTTON_OPENPORT, m_ctrlOpenPort

串口开关标志图标 IDC_STATIC_OPENOFF, m_ctrlIconOpenoff

数据文件保存路径 IDC_EDIT_SAVEPATH, m_strCurPath

保存显示数据文件路径 IDC_EDIT_SAVEPATH,　m_ctrlSavePath

接收计数 IDC_STATIC_RXCOUNT,　m_ctrlRXCOUNT

➢　发送区

发送计数 IDC_STATIC_TXCOUNT,　m_ctrlTXCount

串口状态 IDC_STATIC_STATUS,　m_ctrlPortStatus

手动发送 IDC_BUTTON_MANUALSEND, m_ctrlManualSend

自动发送 IDC_CHECK_AUTOSEND, m_ctrlAutoSend

十六进制发送 IDC_CHECK_HEXSEND, m_ctrlHexSend

发送编辑框 IDC_EDIT_SEND, m_strSendData　Style: Vertical Scroll，MultiLine
要发送的文件名 IDC_EDIT_SENDFILE, m_strSendFilePathName
发送文件 IDC_BUTTON_SENDFILE,　　m_ctrlSendFile
文件名编辑框 IDC_EDIT_SENDFILE,　　m_ctrlEditSendFile
清空重填（发送区）IDC_BUTTON_CLEARRECASENDA,　m_ctrlClearTXData
作者标志图标 IDC_STATIC_XFS,　ICON,　m_ctrlStaticXFS
发送输入编辑框 IDC_EDIT_SEND, m_ctrlEditSend
自动清空（接收框）IDC_CHECK_AUTOCLEAR, m_ctrlAutoClear

➢ **其他**

计数清零IDC_BUTTON_COUNTRESET, m_ctrlCounterReset
图钉按钮 IDC_BUTTON_PUSHPIN,　m_ctrlPushPin
作者主页链接 IDC_STATIC_XFS2, m_ctrlHyperLinkWWW
作者主页链接 IDC_STATIC_GJW, m_ctrlHyperLink2
帮助 IDC_BUTTON_HELP,　m_ctrlHelp
关闭程序 IDC_BUTTON_CLOSE, m_ctrlClose

3. 添加类文件

将类文件 SerialPort.h 和 SerialPort.cpp 复制到工程所在文件夹中（选择改进后的类文件），然后单击 VC 菜单 Project->Add to Project->Files...，再在打开的文件选择对话框中选择 SerialPort.h 和 SerialPort.cpp，单击 OK，就把类文件加入了当前工程（参考图 2.2.2）。并在 SCOMMDlg.h 中将头文件 SerialPort.h 说明：#include "SerialPort.h"。通过以上步骤，就在当前工程中加入了 CSerailPort 类。

4. 完成串口消息处理函数 OnCommunication

在 CSerailPort 类中有多个串口事件可以响应，在一般串口编程中，只需要处理 WM_COMM_RXCHAR 消息就可以了，该类所有的消息均需要人工添加消息处理函数。我们将处理函数名定义为 OnComm()，首先在 SCOMMDlg.h 中添加串口字符接收消息 WM_COMM_RXCHAR（串口接收缓冲区内有一个字符）的响应函数声明：

```
// Generated message map functions
//{{AFX_MSG(CSCOMMDlg)
afx_msg LONG OnCommunication(WPARAM ch, LPARAM port);
//}}AFX_MSG
```

然后，在 SCOMMDlg.cpp 文件中进行 WM_COMM_RXCHAR 消息映射：

```
BEGIN_MESSAGE_MAP(CSCOMMDlg, CDialog)
    //{{AFX_MSG_MAP(CSCOMMDlg)
    ON_MESSAGE(WM_COMM_RXCHAR, OnCommunication)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

接着，在 SCOMMDlg.cpp 文件中加入函数 **OnCommunication** ()的实现，暂不添加代码。

```
LONG SCOMMDlg:: OnCommunication (WPARAM ch, LPARAM port)
```

```
        return 0;
    }
```

以上步骤需要手工完成。

至此，完成了程序的对话框模板，在工程中插入了串口操作类 CSerialPort 类。

5.2　串口的初始化及关闭

程序中有两种方法打开串口，一是程序启动，调用 OnInitDialog()函数，就可以打开串口，缺少的串口号为 COM1，如果 COM1 不存在成被占用，就给出提示；另外，单击“打开串口”按钮也可打开串口。

```
//在OnInitDialog()函数中打开串口
BOOL CSCOMMDlg::OnInitDialog()
{
   BOOL b = CDialog::OnInitDialog();
    // Add "About..." menu item to system menu.
...
    m_nBaud=9600; //波特率
    m_nCom=1;    //串口号
    m_cParity='N'; //奇偶校验
    m_nDatabits=8; //数据位
    m_nStopbits=1; // 停止位
    m_dwCommEvents = EV_RXFLAG | EV_RXCHAR; //串口事件
    //if (m_Port.InitPort(this, 1, 9600,'N',8,1,dwCommEvents,512))
    CString strStatus;
    if (m_Port.InitPort(this, m_nCom, m_nBaud,m_cParity, m_nDatabits, m_nStopbits,
m_dwCommEvents, 512))
    {
        m_Port.StartMonitoring(); //启动监测辅助线程
        strStatus.Format("STATUS: COM%d OPENED, %d,%c,%d,%d",m_nCom,
m_nBaud,m_cParity,m_nDatabits,m_nStopbits); //打印串口状态及参数
        m_ctrlIconOpenoff.SetIcon(m_hIconRed);
        //m_ctrlIconOpenoff.SetIcon(m_hIconOff);

        //"当前状态: 串口打开, 无奇偶校验, 8 数据位, 1 停止位");
    }
    else
    {
        AfxMessageBox("没有发现此串口");
        m_ctrlIconOpenoff.SetIcon(m_hIconOff);
    }
    m_ctrlPortStatus.SetWindowText(strStatus); //显示串口状态及参数
...
    return b;
}
```

在 ClassWizard 中为按钮“打开串口”控制 IDC_BUTTON_OPENPORT 添加单击响应函数。

```
//打开/关闭串口
void CSCOMMDlg::OnButtonOpenport()
{
    // TODO: Add your control notification handler code here
    m_bOpenPort=!m_bOpenPort;
    if(m_bOpenPort)  //关闭串口
```

```
            if(m_ctrlAutoSend.GetCheck())
            {
                m_bOpenPort=!m_bOpenPort;
                AfxMessageBox("请先关掉自动发送");
                return;
            }
            m_ctrlOpenPort.SetWindowText("打开串口");
            m_Port.ClosePort();//关闭串口
            m_ctrlPortStatus.SetWindowText("STATUS: COM Port Closed");
            m_ctrlIconOpenoff.SetIcon(m_hIconOff);
        }
        else  //打开串口
        {
            m_ctrlOpenPort.SetWindowText("关闭串口");
            CString strStatus;
            if(m_Port.InitPort(this,m_nCom,m_nBaud,m_cParity,m_nDatabits,m_nStopbits,
m_dwCommEvents, 512))
            {
                m_Port.StartMonitoring();
                m_ctrlIconOpenoff.SetIcon(m_hIconRed);
                strStatus.Format("STATUS: COM%d OPENED, %d,%c,%d,%d", m_nCom, m_nBaud,
m_cParity, m_nDatabits, m_nStopbits);
                //"当前状态：串口打开，无奇偶校验，8 数据位，1 停止位");
            }
            else
            {
                AfxMessageBox("没有发现此串口或被占用");
                m_ctrlIconOpenoff.SetIcon(m_hIconOff);
            }
            m_ctrlPortStatus.SetWindowText(strStatus);
        }
    }
```

另外，为了在程序关闭时通过关闭串口并释放占用资源，在 ClassWizard 中为 CSCOMMDlg 添加了 WM_DISTROY 的消息响应函数 OnDestroy()，该函数在主窗口即将销毁时调用。

```
void CSCOMMDlg::OnDestroy()
{
    CDialog::OnDestroy();
    m_ctrlAutoSend.SetCheck(0);  //强行关闭自动发送
    KillTimer(1);   //关闭定时器
    KillTimer(4);
    m_Port.ClosePort();  //关闭串口
    m_ReceiveData.Empty();  //清空接收数据字符串
}
```

5.3　串口数据的发送与接收及十六进制数据的处理

在本节里，为程序完成发送与接收编程任务。在串口编程中，经常遇到十六进制的处理，发送时，要读入十六进数据，而接收时，则要将数据转换为十六进制数据显示。

5.3.1　十六进数据发送处理

首先为 CSCOMMDlg 类添加两个成员函数 Str2Hex()和 HexChar()，前者对后者进行了调用。Str2Hex()的作用是将一个字符串作为十六进制串转化为一个字符数组，其中，data 即为返回的数组，函数的返回值为 data 数组的长度。该函数具有一定的通用性，读者根据实际情况修改后可用在自己的程序里。至于以下函数的功能如何实现，读者用这样一段数据去测试一下就明白了：[illegible]。注意，这两个十六进制数之间要有一个空格，看看程序代码中空格是如何消除的。

```
//将一个字符串作为十六进制串转化为一个字节数组，字节间可用空格分隔，
//返回转换后的字节数组长度，同时字节数组长度自动设置。
int CSCOMMDlg::Str2Hex(CString str, char* data)
{
    int t,t1;
    int rlen=0, len=str.GetLength();
    for(int i=0;i<len;)
    {
        char l, h=str[i];
        if(h==' ')     //如果有空格
        {
            i++;
            continue;
        }
        i++;
        if(i>=len) break;
        l=str[i];
        t=HexChar(h);
        t1=HexChar(l);
        if((t==16)||(t1==16))
            break;
        else
            t=t*16+t1;
        i++;
        data[rlen]=(char)t;
        rlen++;
    }
    return rlen;
}

char CSCOMMDlg::HexChar(char c)
{
    if((c>='0')&&(c<='9'))
        return c-0x30;
    else if((c>='A')&&(c<='F'))
        return c-'A'+10;
    else if((c>='a')&&(c<='f'))
        return c-'a'+10;
    else
        return 0x10;
}
```

5.3.2　手动发送处理

在 ClassWizard 中为手动发送按钮 IDC_BUTTON_MANUALSEND 添加单击处理函数(或

直接在对话框模板中双击该控件）OnButtonManualsend()，添加如下代码：

```
long TX_count=0;
void CSCOMMDlg::OnButtonManualsend()
{
    // TODO: Add your control notification handler code here
    if(m_Port.m_hComm==NULL)  //发送时要检测串口是否打开，否则会出错
    {
        m_ctrlAutoSend.SetCheck(0);
        AfxMessageBox("串口没有打开，请打开串口");
        return;
    }
    else
    {
        UpdateData(TRUE);
        if(m_ctrlHexSend.GetCheck())  //发送十六进制数据
        {
            char data[512];
            int len=Str2Hex(m_strSendData,data);
            m_Port.WriteToPort(data,len);
            TX_count+=(long)((m_strSendData.GetLength()+1)/3);
            //计算发送的十六进制数据，注意这里的计算方法，
            //只有严格按照规则输入才能正确计算
        }
        else  //发送ASCII文本
        {
            m_Port.WriteToPort((LPCTSTR)m_strSendData);      //发送数据
            TX_count+=m_strSendData.GetLength();             //发送计数
        }
        CString strTemp;
        strTemp.Format("TX:%d",TX_count);
        m_ctrlTXCount.SetWindowText(strTemp);                //显示计数
    }
}
```

5.3.3　自动发送处理

自动发送时，需要用到定时器。打开 ClassWizard，为 CSCOMMDlg 类添加 WM_TIMER 消息处理函数 OnTimer(UINT nIDEvent)。要注意的是，在 VC 中，每个定时器都有自己的 ID 号，所有定时处理均在该函数中，因此，必须事先为相应的定时器设置 ID 号，OnTimer(UINT nIDEvent)函数则根据调用的 nIDEvent 值来确定是哪个定时器的定时时间到，再做相应处理。

```
void CSCOMMDlg::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    switch(nIDEvent)
    {
    case 1:  //定时器ID=1为自动发送时间到
        OnButtonManualsend();   //自动发送周期到后，
                                //只需简单地调用OnButtonManualsend
        break;
    case 2:  //其他定时器
        ...
    default:
        break;
```

```
        CDialog::OnTimer(nIDEvent);
    }
```

我们再来看看定时器 1 如何设置。在 ClassWizard 中，为手动发送按钮自动发送 IDC_CHECK_AUTOSEND 添加单击处理函数（或直接在对话框模板中双击该控件）OnCheckAutosend ()，添加如下代码：

```
void CSCOMMDlg::OnCheckAutosend()
{
    // TODO: Add your control notification handler code here
    m_bAutoSend=!m_bAutoSend;  //标志是否打开自动发送
    if(m_bAutoSend)
    {
        if(m_Port.m_hComm==NULL)
        {
            m_bAutoSend=!m_bAutoSend;
            m_ctrlAutoSend.SetCheck(0);
            AfxMessageBox("串口没有打开，请打开串口");
            return;
        }
        else
            SetTimer(1,m_nCycleTime,NULL);  //设置定时器1
    }
    else
    {
        KillTimer(1);  //“杀”掉定时器1
    }
}
```

在发送时，当发送输入编辑框中的内容改变时，要及时将串口发送的内容改变。在 ClassWizard 中为编辑框 IDC_EDIT_SEND 添加 EN_CHANGE 响应函数：

```
void CSCOMMDlg::OnChangeEditSend()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
}
```

同样，当自动发送周期改变后，也需要将发送周期更新，但实际的发送周期需要重新用 SetTimer()函数对定时器的定时时间进行设置。在这个程序里，必须先关闭串口的“自动发送”，然后再打开，新设的自动发送周期才能生效。

```
void CSCOMMDlg::OnChangeEditCycletime()
{
    // TODO: Add your control notification handler code here
    CEdit* pEdit=(CEdit*)GetDlgItem(IDC_EDIT_CYCLETIME);
    CString strText;
    pEdit->GetWindowText(strText);
    m_nCycleTime=atoi(strText);
}
```

5.3.4　接收处理及十六进制显示

接收处理均在串口事件消息处理函数 OnCommunication()函中实现。其中，十六进制的接收显示时并不像发送那样麻烦，只要将数据直接以十六制打印输出就可以了，注意，中间

插入一个空格。

```
static long rxdatacount=0; //该变量用于接收字符计数
LONG CSCOMMDlg::OnCommunication(WPARAM ch, LPARAM port)
{
    if (port <= 0 || port > 4)
        return -1;
    rxdatacount++;  //接收的字节计数
    CString strTemp;
    strTemp.Format("%ld",rxdatacount);
    strTemp="RX:"+strTemp;
    m_ctrlRXCOUNT.SetWindowText(strTemp); //显示接收计数

    if(m_bStopDispRXData)  //如果选择了"停止显示"接收数据，则返回
        return -1;         //注意，这种情况下，计数仍在继续，只是不显示
    //若设置了"自动清空"，则达到50行后，自动清空接收编辑框中显示的数据
    if((m_ctrlAutoClear.GetCheck())&&(m_ctrlReceiveData.GetLineCount()>=50))
    {
        m_ReceiveData.Empty();
        UpdateData(FALSE);
    }
    //如果没有"自动清空"，数据行达到400后，也自动清空
    //因为数据过多，影响接收速度，显示是最费CPU时间的操作
    if(m_ctrlReceiveData.GetLineCount()>400)
    {
        m_ReceiveData.Empty();
        m_ReceiveData="***The Length of the Text is too long, Emptied Automaticly !!!
*** \r\n";
        UpdateData(FALSE);
    }

    //如果选择了"十六进制显示"，则显示十六进制值
    CString str;
    if(m_ctrlHexReceieve.GetCheck())
        str.Format("%02X ",ch);
    else
        str.Format("%c",ch);
    //以下是将接收的字符加在字符串的最后，这里费时很多
    //但考虑到数据需要保存成文件，所以没有用List Control
    int nLen=m_ctrlReceiveData.GetWindowTextLength();
    m_ctrlReceiveData.SetSel(nLen, nLen);
    m_ctrlReceiveData.ReplaceSel(str);
    nLen+=str.GetLength();

    m_ReceiveData+=str;
    return 0;
}
```

接收显示是影响程序性能的一个大问题，当接收大量数据时，串口调试助手响应不太及时，这可能与类中使用的机制有关，但显示也没能很好地处理。读者如果有兴趣，可以在这里改进测试一下。

以下是"清空接收区"和"停止/继续显示"两个按钮的单击响应函数。

```
//清空接收区
void CSCOMMDlg::OnButtonClearReciArea()
{
    // TODO: Add your control notification handler code here
    m_ReceiveData.Empty();
```

```
    UpdateData(FALSE);
}

//停止/继续显示接收数据
void CSCOMMDlg::OnButtonStopdisp()
{
    // TODO: Add your control notification handler code here
    m_bStopDispRXData=!m_bStopDispRXData;
    if(m_bStopDispRXData)
        m_ctrlStopDisp.SetWindowText("继续显示");
    else
        m_ctrlStopDisp.SetWindowText("停止显示");
}
```

5.4 其他辅助功能的实现

本节来实现串口调试助手的一些辅助功能。虽然这些功能的编程不属于串口编程的范畴，但作为一个整体程序，有不少本程序的使用者问到了这些功能的实现。其实，编程不可能全部是程序员本人的劳动，我们编程时要时刻记住去借助前人的劳动成果。本程序的几个功能几乎全部借用了一些免费的类，编程实现非常简单。

5.4.1 接收数据的文件保存

串口设备调试时，有时需要把接收到的数据保存成文件，以对这些数据做进一步的分析。本程序将接收编辑框中的数据保存成文本文件。缺少的路径为 C:\COMDATA，使用者可以通过单击“更改”按钮来选择其他文件夹。文件名为 Rec**.txt，程序自动检测文件名是否存在，若存在，则后面序号自动递增，例如，当程序检测到 Rec00.txt Rec01.txt 文件存在，则会自动为正要保存的文件命名为 Rec02.txt。

在 ClassWizard 中为按钮“保存显示数据”IDC_BUTTON_SAVEDATA 添加单击响应函数 OnButtonSavedata():

```
//保存显示数据
void CSCOMMDlg::OnButtonSavedata()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);

    int       nLength;
    nLength = m_strCurPath.GetLength();

    for( int nCount = 0; nCount < nLength; nCount++ )
    {
        if( m_strCurPath.GetAt( nCount ) == '\\' )
            CreateDirectory( m_strCurPath.Left( nCount + 1 ), NULL );
    }
    CreateDirectory( m_strCurPath, NULL );

    CFile m_rFile;
    LPCSTR    lpszPath =m_strCurPath;// "c:\\comdata";
    SetCurrentDirectory( lpszPath );
```

```
        //文件名为Rec**.txt,以下代码自动检测文件名是否存在，若存在，则后面序号
        //自动递增，如Rec00.txt Rec01.txt,程序自动为正要保存的文件命名为Rec02.txt.
        char buf[20];
        for(int j=0;j<100;j++)
        {
            sprintf(buf,"Rec%02d.txt",j);
            if( (access( buf, 0 )) == -1 )
                break;
        }

        if(!m_rFile.Open(buf,CFile::modeCreate | CFile::modeWrite ))
        {
            AfxMessageBox( "创建记录文件失败！");
            return;
        }
        if((access(buf,0))==-1)
        {
            AfxMessageBox("failed");
            return;
        }
    //在文件开始处写上保存日期
        CTime t = CTime::GetCurrentTime();
        CString str=t.Format("%Y年%m月%d日%H时%M分%S秒\r\n");
        m_rFile.Write((LPCTSTR)str,str.GetLength());
    //保存显示数据
        m_rFile.Write((LPCTSTR)m_ReceiveData,m_ReceiveData.GetLength());
        m_rFile.Flush();
        m_rFile.Close(); //关闭文件

       str="OK,";
        for(int i=0;i<5;i++)
           str+=buf[i];
        str+=".txt saved";
        m_ctrlSavePath.SetWindowText(str);
        SetTimer(2,5000,NULL);  //在定时器中显示保存文件状态
    }
```

上面设置的定时器 ID 为 2，看看我们在定时器的 OnTimer()函数中做了什么：

```
void CSCOMMDlg::OnTimer(UINT nIDEvent)
{
    switch(nIDEvent)
    {
-----
    case 2:
        m_ctrlSavePath.SetWindowText(m_strCurPath); //重新显示路径
        KillTimer(2);  //关闭定时器
        break;
-----
    default:
        break;
    }
    CDialog::OnTimer(nIDEvent);
}
```

接着，还要在 ClassWizard 中为按钮“更改”（改变保存文件的缺省路径）IDC_BUTTON_DIRBROWSER 添加单击响应函数 OnButtonDirbrowser()：

```
//改变文件保存路径
void CSCOMMDlg::OnButtonDirbrowser()
{
    // TODO: Add your control notification handler code here
    static char displayname[MAX_PATH];
    static char path[MAX_PATH];
  LPITEMIDLIST pidlBrowse;   // PIDL selected by user
    BROWSEINFO bi;
    bi.hwndOwner = this->m_hWnd;
    bi.pidlRoot = NULL;
    bi.pszDisplayName = displayname;
    bi.lpszTitle = "请选择要保存接收数据的文件夹";
    bi.ulFlags = BIF_EDITBOX ;
    bi.lpfn = NULL;
    pidlBrowse=SHBrowseForFolder( &bi);
    if(pidlBrowse!=NULL)
    {
        SHGetPathFromIDList(pidlBrowse,path);
    }
    CString str=path;  //得到路径
    if(str.IsEmpty()) return;  //如果没有选择，就返回
    m_strCurPath=str;  //接收路径编辑框对应变量
    UpdateData(FALSE);
}
```

5.4.2 实现小文件发送

严格来讲，三线制的串口线用来发送文件有很大困难，一般需要用到流控制，但只要文件长度不太长，还是可以发送的，这在工控领域可以用得上，例如，可以做一个文件，随时改变文件中的内容，通过串口发送出去。读者可以用几个小文本文件测试一下。

首先，选择要发送的小文件。在 ClassWizard 中为按钮“选择发送文件” IDC_BUTTON_FILEBROWSER 添加单击响应函数 OnButtonFilebrowser()：

```
//选择要发送的文件
void CSCOMMDlg::OnButtonFilebrowser()
{
    // TODO: Add your control notification handler code here
    LPCSTR   lpszPath = "c:\\comdata";
    SetCurrentDirectory( lpszPath );
    static char BASED_CODE szFilter[] = "文本文件(*.txt)|*.txt|所有文件(*.*)|*.*||";

    CFileDialog FileDlg( TRUE,  NULL,  NULL,OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
         szFilter );
    FileDlg.m_ofn.lpstrInitialDir = lpszPath ;
    if( FileDlg.DoModal() ==IDOK )
    {
        CString strFileName = FileDlg.GetFileName( );
        CString strFileExt = FileDlg.GetFileExt( );
        CString lpstrName =  FileDlg.GetPathName( );
        m_strSendFilePathName=lpstrName;
        UpdateData(FALSE);
    }
}
```

再在 ClassWizard 中为按钮“发送文件” IDC_BUTTON_SENDFILE 添加单击响应函数 OnButtonSendfile()：

```
//发送文件
void CSCOMMDlg::OnButtonSendfile()
{
    // TODO: Add your control notification handler code here
    CFile fp;
    if(!(fp.Open((LPCTSTR)m_strSendFilePathName,CFile::modeRead)))
    {
        AfxMessageBox("Open file failed!");
        return;
    }
    fp.SeekToEnd();
    unsigned long fplength=fp.GetLength();
    m_nFileLength=fplength;
    char* fpBuff;
    fpBuff=new char[fplength];
    fp.SeekToBegin();
    if(fp.Read(fpBuff,fplength)<1)
    {
        fp.Close();
        return;
    }
    fp.Close();

    CString strStatus;
    if (m_Port.InitPort(this, m_nCom, m_nBaud, m_cParity, m_nDatabits, m_nStopbits,
m_dwCommEvents, fplength))
    {
        m_Port.StartMonitoring();
        strStatus.Format("STATUS: COM%d OPENED, %d,%c,%d,%d",m_nCom,
m_nBaud,m_cParity,m_nDatabits,m_nStopbits);
        m_ctrlIconOpenoff.SetIcon(m_hIconRed);
        m_bSendFile=TRUE;
        m_strTempSendFilePathName=m_strSendFilePathName;
        m_ctrlEditSendFile.SetWindowText("正在发送.....");
        //发送文件时，以下功能不能使用
        m_ctrlManualSend.EnableWindow(FALSE);
        m_ctrlAutoSend.EnableWindow(FALSE);
        m_ctrlSendFile.EnableWindow(FALSE);
        m_Port.WriteToPort((LPCTSTR)fpBuff,fplength);
    }
    else
    {
        AfxMessageBox("Failed to send file!");
        m_ctrlIconOpenoff.SetIcon(m_hIconOff);
    }
    delete fpBuff;
}
```

那么，文件什么时候发送完了呢？可以通过 CSerialPort 类的 WM_COMM_TXEMPTY_DETECTED 消息来判断。下面手工添加该消息的处理函数。

首先，在 SCOMMDlg.h 中添加串口字符接收消息 WM_COMM_TXEMPTY_DETECTED（串口接收缓冲区内有一个字符）的响应函数声明：

```
    // Generated message map functions
    //{{AFX_MSG(CSCOMMDlg)
    afx_msg LONG OnFileSendingEnded(WPARAM wParam,LPARAM port);
//}}AFX_MSG
```

然后，在 SCOMMDlg.cpp 文件中进行 WM_COMM_TXEMPTY_DETECTED 消息映射：

```
BEGIN_MESSAGE_MAP(CSCOMMDlg, CDialog)
    //{{AFX_MSG_MAP(CSCOMMDlg)
    ON_MESSAGE(WM_COMM_TXEMPTY_DETECTED, OnFileSendingEnded)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

接着，在 SCOMMDlg.cpp 文件中加入函数 OnFileSendingEnded ()的实现。

```
//检测文件是否发送完毕
LONG CSCOMMDlg::OnFileSendingEnded(WPARAM wParam,LPARAM port)
{
    if(m_bSendFile)
    {
        m_ctrlEditSendFile.SetWindowText("发送完毕!");//m_strSendFilePathName
        TX_count+=m_nFileLength;
        SetTimer(3,3000,NULL);
        CString strTemp;
        strTemp.Format("TX:%d",TX_count);
        m_ctrlTXCount.SetWindowText(strTemp);
        m_bSendFile=FALSE;
    }
    return 0;
}
```

上面设置了定时器 3，看看在 OnTimer()函数中对该定时的处理代码：

```
void CSCOMMDlg::OnTimer(UINT nIDEvent)
{
    switch(nIDEvent)
    {
...
        case 3:
        m_ctrlManualSend.EnableWindow(TRUE);
        m_ctrlAutoSend.EnableWindow(TRUE);
        m_ctrlSendFile.EnableWindow(TRUE);
        m_strSendFilePathName=m_strTempSendFilePathName;
        m_ctrlEditSendFile.SetWindowText(m_strSendFilePathName);
        //m_strSendFilePathName
        KillTimer(3);
        if(!(m_ctrlAutoSend.GetCheck()))
        {
            if (m_Port.InitPort(this, m_nCom, m_nBaud, m_cParity, m_nDatabits,
m_nStopbits, m_dwCommEvents, 512))
            {
                m_Port.StartMonitoring();
                strStatus.Format("STATUS:COM%dOPENED,%d,%c,%d,%d",m_nCom,m_nBaud,
m_cParity, m_nDatabits, m_nStopbits);
                m_ctrlIconOpenoff.SetIcon(m_hIconRed);
            }
            else
            {
                AfxMessageBox("Failed to reset send buffer size!");
                m_ctrlIconOpenoff.SetIcon(m_hIconOff);
            }
            m_ctrlPortStatus.SetWindowText(strStatus);
        }
        break;
...
```

```
        default:
            break;
        }
        CDialog::OnTimer(nIDEvent);
    }
```

要注意的是，读者如果在实际编程中，要用到文件发送，则可对文件长度在程序中做出限制。

5.4.3　图钉按钮功能使程序能浮在最上层

在观察串口数据时，有时需要将串口调试助手置于最上层，即“浮”在最上面，始终可见。我们可以模仿 VC 开发环境中的属性（Properties）对话框中左上角的图钉按钮功能来实现这一要求。

幸运的是，这个功能的实现有一个第三方提供的免费类，不需要写多少代码。首先在当前程序工程中加入这个类。SCOMM 工程文件夹中的 PushPin.cpp 和 PushPin.h 两个文件就是 PushPin 类的类文件（如果读者应用，可将这两个类文件复制到工程所在文件夹中），单击 VC 菜单 Project->Add to Project->Files...，再在打开的文件选择对话框中选择上 PushPin.cpp 和 PushPin.h，单击 OK，就把类文件加入了当前工程，如图 5.4.1 所示。在 ClassView 中就可以看到 CPushPinButton 类了。

图 5.4.1　在工程中插入 PushPin 类

然后，在 SCOMMDlg.h 中将头文件 PushPin.h 说明：#include "PushPin.h"。接着，设置按钮控件 IDC_BUTTON_PUSHPIN 的属性，在 Properties—>Style 中，选上 Ower draw、Bitmap 和 Notify 属性。

现在我们要做的就是将两个代表“钉住”（浮在最上层，保持可见）和“松开”（不浮在最上层）的位图放到工程文件夹中的 res 文件夹中，将位图 pinned.bmp,unpinned.bmp 导入（Import）到项目中，对应 ID 分别为：IDB_PINNED_BITMAP，IDB_UNPINNED_BITMAP。

再在 ClassWizard->Member Variable 中为按钮控件 IDC_BUTTON_PUSHPIN 添加一个 CPushPinButton 控制变量 m_ctrlPushPin.，如图 5.4.2 所示。

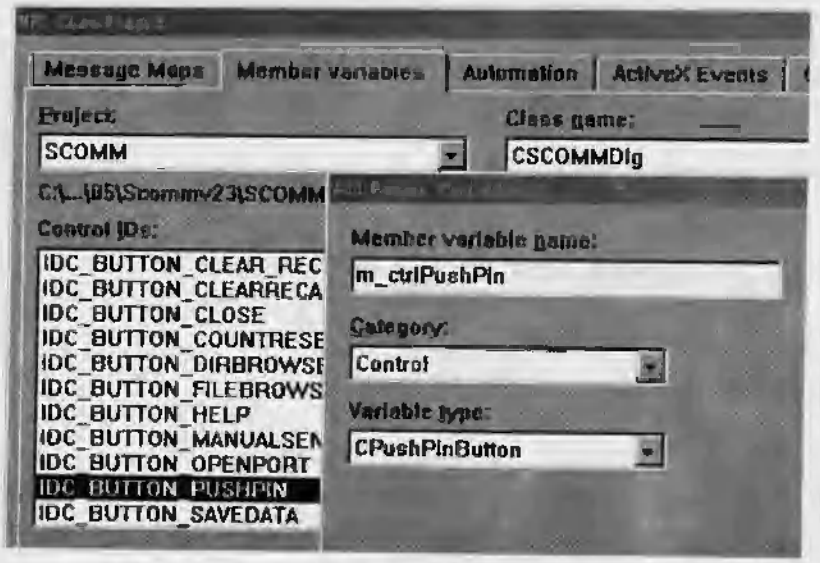


图 5.4.2　为 IDC_BUTTON_PUSHPIN 按钮添加控制变量

利用 ClassWizard 为 IDC_BUTTON_PUSHPIN 添加 BN_CLICKED 消息处理函数，名称用缺省名称 OnButtonPushpin ()，在函数中加入如下代码：

```
void CSCOMMDlg::OnButtonPushpin()
{
    // TODO: Add your control notification handler code here
    m_ctrlPushPin.ProcessClick();
    m_bVisible=!m_bVisible;
    if(m_bVisible)
    {
        SetWindowPos(&wndTopMost, 0, 0, 0, 0, SWP_NOMOVE|SWP_NOSIZE);
    }
    else
    {
        SetWindowPos(&wndBottom, 0, 0, 0, 0, SWP_NOMOVE |SWP_NOSIZE | SWP_NOREDRAW);
        BringWindowToTop();
    }
}
```

5.4.4　对话框动画图标的实现

对话框左上角动画图标对于串口功能并无影响，但程序发布后，也有使用者问到如何编程实现，在这里做简要说明。

AnimateDlgIcon.cpp 和 AnimateDlgIcon.h 是类文件，复制到当前工程文件的文件中，并把类加入到当前工程（方法参考第 5.4.3 节）。并在 SCOMMDlg.h 文件中加入类的对象说明：

```
#include "AnimateDlgIcon.h"
……
public:
CAnimateDlgIcon m_animIcon;
```

需要准备一幅位图 IDB_ANIM_IMGLIST，制作顺序按动画顺序而定。可以直接在 VC

集成环境中制作，如图 5.4.3 所示，注意，其宽×高为 64×18（实际为 4 幅 16×16 的位图），颜色值为 16。

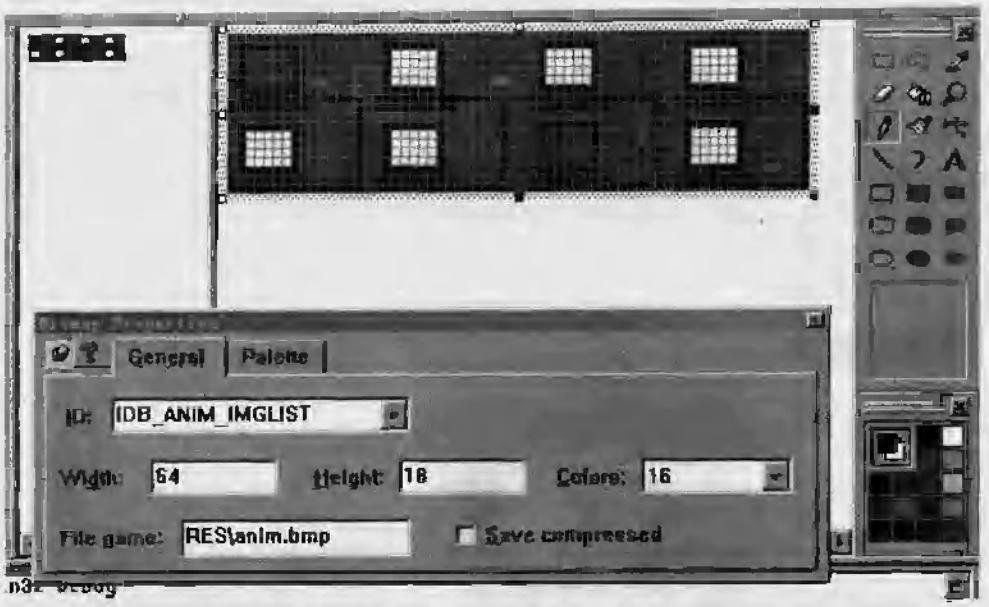


图 5.4.3　制作 IDB_ANIM_IMGLIST 位图

设置为程序一启动，图标就开始动作，在 OnInitDialog()函数中加入以下代码：

```
BOOL CSCOMMDlg::OnInitDialog()
{
- -
    m_animIcon.SetImageList(IDB_ANIM_IMGLIST,4,RGB(0,0,0));
    SetTimer(4,200,NULL);   //设置定时器，定时时间控制动作的快慢
- -
}
```

上面代码中设置了定时器（ID 为 4），定时时间控制动作的快慢，在定时处理函数中，定时时间到后就显示下一幅图像：

```
void CSCOMMDlg::OnTimer(UINT nIDEvent)
{
    switch(nIDEvent)
    {
- -
    case 4:
        m_animIcon.ShowNextImage();  //显示一下幅图像
        break;
    default:
        break;
    }
    CDialog::OnTimer(nIDEvent);
}
```

5.4.5 超链接功能的实现

这里我们同样用到了两个常用来做超链接的类 CHyperLink 和 CLabel 类，这里只就 HyperLink 给出说明。

在对话框中已添加了两个静态文本控件 WEB 和 MAIL，这里不能用缺省的 IDC_STATIC，分别修改为 IDC_STATIC_XFS2 和 IDC_STATIC_GJW，前者用于链接本书作者主页，后者用于链接 E-mail，再在 ClassWizard 中为它们添加控制变量 CHyperLink 型控制变量 m_ctrlHyperLinkWWW 和 m_ctrlHyperLink2，其添加方法可参考图 5.4.2。

一般，将鼠标点到超链接时，鼠标形状发生变化，这里也可以将一个手形鼠标导入（Import）工程中，如图 5.4.4 所示。

图 5.4.4　在工程中添加手形鼠标

```
BOOL CSCOMMDlg::OnInitDialog()
{
...
    m_ctrlHyperLink2.SetURL(_T("mailto:webmaster@gjwtech.com"));
    m_ctrlHyperLink2.SetUnderline(TRUE);  //是否有下划线
    m_ctrlHyperLink2.SetLinkCursor(AfxGetApp()->LoadCursor(IDC_CURSOR_HAND));

    m_ctrlHyperLinkWWW.SetURL(_T("http://www.gjwtech.com"));
    m_ctrlHyperLinkWWW.SetUnderline(TRUE);  //是否有下划线
    m_ctrlHyperLinkWWW.SetLinkCursor(AfxGetApp()->LoadCursor(IDC_CURSOR_HAND));
...
}
```

5.4.6 如何打开帮助网页文件

实现帮助的方法有很多种，本程序仅用了一个简单的网页文件，即单击“帮助”按钮后用 IE 打开一个网页文件。实现代码如下：

```
void CSCOMMDlg::OnButtonHelp()
{
    // TODO: Add your control notification handler code here
//首先要得到应用程序所在的路径
    TCHAR exeFullPath[MAX_PATH];
    GetModuleFileName(NULL,exeFullPath,MAX_PATH);
    CString strlpPath;
    strlpPath.Format("%s",exeFullPath);
    strlpPath.MakeUpper();
    strlpPath.Replace("串口调试助手V2.2.EXE","");
    ShellExecute(NULL,NULL,_T("help.htm"),NULL,strlpPath,SW_SHOW);
}
```

以上代码编写完毕后，程序中主要部分 CSCOMMDlg 类的头文件的全部代码如下：

```
// SCOMMDlg.h : header file
//
#include "SerialPort.h"
#include "HyperLink.h"
#include "label.h"
#include "PushPin.h"
#include "AnimateDlgIcon.h"

#if !defined(AFX_SCOMMDLG_H__666127A8_FEE4_40AA_9309_1B3B55EEDAFC__INCLUDED_)
#define AFX_SCOMMDLG_H__666127A8_FEE4_40AA_9309_1B3B55EEDAFC__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

/////////////////////////////////////////////////////////////////////////////
// CSCOMMDlg dialog

class CSCOMMDlg : public CDialog
{
// Construction
public:
    BOOL m_bVisible;  //程序是否浮在最上面 用于图钉按钮功能
    BOOL m_bStopDispRXData; //是否显示接收字符
    CString m_strTempSendFilePathName; //发送文件路径名
    long m_nFileLength;  //文件长度
    BOOL m_bSendFile;   //是否发送文件
    HICON m_hIconRed;   //串口打开时的红灯图标句柄
    HICON m_hIconOff;   //串口关闭时的指示图标句柄
    HICON m_hIconGreen;

    int m_nBaud;      //波特率
    int m_nCom;       //串口号
    char m_cParity;   //校验
    int m_nDatabits;   //数据位
    int m_nStopbits;   //停止位
    CSerialPort m_Port;  //CSerialPort 类对象
    CAnimateDlgIcon m_animIcon;  //动画图标
    CSCOMMDlg(CWnd* pParent = NULL); // standard constructor

// Dialog Data
    //{{AFX_DATA(CSCOMMDlg)
    enum { IDD = IDD_SCOMM_DIALOG };
    CButton  m_ctrlHelp;
    CPushPinButton m_ctrlPushPin;
```

```
    CButton  m_ctrlSendFile;
    CEdit    m_ctrlEditSendFile;
    CStatic  m_ctrlTXCount;
    CStatic  m_ctrlPortStatus;
    CStatic  m_ctrlRXCOUNT;
    CEdit    m_ctrlSavePath;
    CButton  m_ctrlManualSend;
    CHyperLink    m_ctrlHyperLink2;
    CButton  m_ctrlClearTXData;
    CStatic  m_ctrlStaticXFS;
    CButton  m_ctrlClose;
    CButton  m_ctrlCounterReset;
    CEdit    m_ctrlEditSend;
    CEdit    m_ctrlReceiveData;
    CButton  m_ctrlAutoClear;
    CStatic  m_ctrlIconOpenoff;
    CHyperLink    m_ctrlHyperLinkWWW;
    CComboBox m_StopBits;
    CComboBox m_DataBits;
    CComboBox m_Parity;
    CButton  m_ctrlAutoSend;
    CButton  m_ctrlHexSend;
    CButton  m_ctrlStopDisp;
    CButton  m_ctrlOpenPort;
    CButton  m_ctrlHexReceieve;
    CComboBox m_Speed;
    CComboBox m_Com;
    CString  m_ReceiveData;
    CString  m_strSendData;
    CString  m_strCurPath;
    CString  m_strSendFilePathName;
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CSCOMMDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    int m_nCycleTime;
    BOOL m_bAutoSend;
    int Str2Hex(CString str, char *data);
    char HexChar(char c);
    DWORD m_dwCommEvents;
    BOOL m_bOpenPort;
    HICON m_hIcon;

    // Generated message map functions
    //{{AFX_MSG(CSCOMMDlg)
    virtual BOOL OnInitDialog();
    afx_msg LONG OnCommunication(WPARAM ch, LPARAM port);
    afx_msg LONG OnFileSendingEnded(WPARAM wParam,LPARAM port);
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg void OnButtonClearRecIArea();
    afx_msg void OnButtonOpenport();
    afx_msg void OnButtonStopdisp();
```

```
        afx_msg void OnButtonManualsend();
        afx_msg void OnCheckAutosend();
        afx_msg void OnTimer(UINT nIDEvent);
        afx_msg void OnChangeEditCycletime();
        afx_msg void OnChangeEditSend();
        afx_msg void OnButtonClearrecasenda();
        afx_msg void OnSelendokComboComselect();
        afx_msg void OnSelendokComboSpeed();
        afx_msg void OnSelendokComboParity();
        afx_msg void OnSelendokComboDatabits();
        afx_msg void OnSelendokComboStopbits();
        afx_msg void OnButtonSavedata();
        afx_msg void OnButtonDirbrowser();
        afx_msg void OnButtonSendfile();
        afx_msg void OnButtonCountreset();
        afx_msg void OnButtonClose();
        afx_msg void OnButtonFilebrowser();
        afx_msg void OnButtonPushpin();
        afx_msg void OnDestroy();
        afx_msg void OnButtonHelp();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
        //DECLARE_DYNAMIC_MAP()
    private:
    };
    //{{AFX_INSERT_LOCATION}}
    // Microsoft Visual C++ will insert additional declarations immediately before the previous
line.
    #endif
    // !defined(AFX_SCOMMDLG_H__666127A8_FEE4_40AA_9309_1B3B55EEDAFC__INCLUDED_)
```

第6章　DOS环境下的Turbo C串口编程及通用实例GSerial类

[内容提要]

DOS下的串口编程我们在第1章1.3节已经初步尝试过了，但也留下了许多疑问：

- *代码到底是如何控制串口的？*
- 要是对程序进行修改，还需要了解哪些知识？
- 就像Windows操作系统下一样，有没有像CSeiralPort类一样的更通用一点的可重用编程代码，让DOS下的串口编程更轻松呢？

  这正是本章要回答的问题。

由于DOS操作系统下，不能像Windows操作系统一样有API（应用编程接口），而是直接操作硬件端口来实现编程。所以，本章首先介绍PC机异步通信适配器（核心器件为INS8250）及其编程考虑。

接着，我们来尝试改进第1章的程序comrx.cpp，由于该程序只实现了中断接收功能，本章将为它增加发送功能，并将程序命名为comrxtx.cpp，再进行程序测试。

然后，介绍一个较为通用的可重用代码GSerial类，并对代码进行了详细的说明，以利于读者对其进行改进，去适应自己的程序。并利用这段代码编写了一个DOS环境下的串口终端程序，该终端程序也可以作为一个简单的串口调试工具。

6.4节是利用GSerial类进行多串口编程控制，在同一台计算机上对多串口的操作进行了说明。

最后，还对多串口编程中用到的高号中断编程，对可编程控制器8259的处理给出了代码实例，说明了注意事项。

学习完本章知识，就能做到对DOS环境下的编程应付自如了。

*MS-DOS的16位编程模式可以直接访问PC机硬件，对其进行读写操作；Windows 9x的32位编程模式（实际语句仍为16位编程语言）也可以对硬件端口进行读写；但Windows 2000/NT/XP操作系统下的32位编程模式下不支持这种对硬件的直接访问，只有通过设备驱动程序或其他接口来进行。

*DOS程序编写调试小技巧：因为Visual C++ 6.0有强大易用的编辑功能，因此，编写时可在Visual C++ 6.0集成环境下进行，调试时再回到Turbo C的DOS模式下。

6.1　PC 机异步通信适配器 8250 及其编程操作

6.1.1　INS8250 内部寄存器及其选择方式

PC 机主板上负责串行通信的核心器件为 8250（或其兼容元器件）异步通信适配器（UART）。程序通过对 8250 内部的寄存器进行读写进而控制 8250。表 6-1-1 为 8250 内部寄存器的定义和描述。

表 6-1-1　8250 内部寄存器

寄存器	缩写	偏移	COM1	COM2	A2A1A0	D7
接收数据寄存器	RXR	0	3F8H	2F8H	000	0
发送数据寄存器	TXR	0	3F8H	2F8H	000	
中断允许寄存器	IER	1	3F9H	2F9H	001	
中断标识寄存器	IIR	2	3FAH	2FAH	010	×
线路控制寄存器	LCR	3	3FBH	2FBH	011	
MODEM 控制寄存器	MCR	4	3FCH	2FCH	100	
线路状态寄存器	LSR	5	3FDH	2FDH	101	
MODEM 状态寄存器	MSR	6	3FEH	2FEH	110	
波特率除数锁存器低	LSB	0	3F8H	2F8H	000	1
波特率除数锁存器高	MSB	1	3F9H	2F9H	001	

从表中可以看出，INS8250 借助线控制寄存器的最高位 D7，通过三条寄存器选通线 A2A1A0，实现对寄存器进行寻址。当 D7 为 0 时，寄存器 0 和 1 分别为接收/发送数据寄存器和中断允许寄存器；当 D7 为 1 时，则分别为低 8 位除数锁存器 LSB 和高 8 位除数锁存器 MSB。D7 也通常被称为除数锁存器访问位 DLAB。

6.1.2　波特率设置

由于波特率时钟是主数据时钟的十六分之一，而 INS8250 使用频率为 1.8432MHz 的基准时钟输入信号作为主数据时钟，因此对于要求的波特率可用如下公式计算除数因子：

除数因子＝（主数据时钟频率/16）/波特率＝115200/波特率

表 6-1-2 列出了常用波特率及其除数锁存器之关系。

表 6-1-2　常用波特率及其除数锁存器

波特率	波特率除数锁存器 MSB	波特率除数锁存器 LSB
50	09H	00H
300	01H	80H
1200	00H	60H
1800	00H	4DH
2000	00H	3AH
2400	00H	30H
3600	00H	20H
4800	00H	18H
7200	00H	10H
9600	00H	0CH
19200	00H	06H

实际应用时，须先将除数锁存器访问位 DLAB 置 1，然后按照低位在前、高位在后的顺序将除数因子写入相应寄存器中。如要求设置串口 1 波特率为 4800bps，则语句如下：

```
outportb(0x3fb,0x80);  //将D7（DLAB）置1，选择除数锁存器
outportb(0x3f8,0x18);
outportb(0x3f9,0x00);
```

6.1.3　数据位、奇偶校验、停止位等数据格式设置

数据格式是通过线路控制寄存器 LCR 进行设置的，通常放在初始化函数中。串行数据格式如图 6.1.1 所示。数据位 D0 是发送或接收的第 1 位，适配器 8250 自动地插入起始位、正确的奇偶位（如果程序这样安排），以及停止位（1、1.5 或 2 位，取决于线路控制寄存器 LCR 中的命令）。

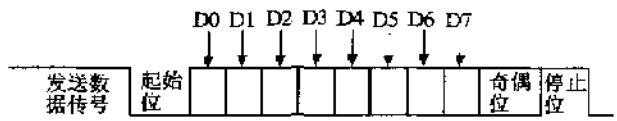


图 6.1.1　串行数据格式

线控制寄存器各位的定义如下。

D7：该位即前面提到的 DLAB，它与数据格式无关。当 D7 为 0 时，寄存器 0 和 1 分别为接收/发送数据寄存器；D7 为 1 时，则分别为低 8 位除数锁存器 LSB 和高 8 位除数锁存器 MSB。

D6：BREAK 位。当该位为 1 时，允许发送器持续一个完整帧时间以上的空号状态。

D5 D4 D3:奇偶检验位设置，其组合含义见表 6-1-3。

表 6-1-3　奇偶校验设置位组合

D5D4D3	含义
000 / 010 / 100 / 110	无奇偶校验
001	奇校验
011	偶校验
101	标志(MARK)奇偶校验， 奇偶位保持传号（恒为 1）
111	空白(SPACE)奇偶校验， 奇偶位保持空号（恒为 0）

D2：停止位设置。当该位为 0 时，停止位数为 1 位；该位为 1，且 D1D0=00 时，停止位为 1.5，当 D1D0≠00 时，停止位为 2。

D1 D0：表示数据位位数。其组合含义如表 6-1-4 所示。

表 6-1-4　数据位设置组合

D1D0	数据位数
00	5
01	6
10	7
11	8

显然，对数据位进行设置时，必须先将除数锁存器访问位 DLAB 置 0。如要求串口 1 通信数据格式设置为 8 位数据位、偶校验和 2 位停止位，则语句如下：

```
outportb(0x3fb,0x00);
outportb(0x3fb,0x1f);
```

6.1.4　查询 I/O 方式相关设置

当串口编程采用查询 I/O 方式时，需要定时读取线路状态寄存器。线路状态寄存器提供串行数据发送和接收时的状态，包括接收器错误、发送器和接收器关键寄存器的就绪状态等。在发送字节前，需要检测发送器是否就绪，当发送保持寄存器为空时，就可以发送下一个字节。表 6-1-5 为线状态寄存器各位的含义。

表 6-1-5　线状态寄存器各位含义

状态内容	位	状态含义
保留	D7	恒为零（未用）
操作是否就绪	D6	发送器空："所有字节都已发送"标志，该位为 1 时，表示发送器的发送寄存器和移位寄存器中都没有字节。通常在结束发送之前，查询该位以防止丢失待发送的数据
	D5	发送器传输保持寄存器空：发送器该位为 1 时，表示发送的字节从发送器的保持寄存器移入发送移位寄存器，即发送操作就绪，准备好接收 CPU 下一个要发送的字节
	D0	数据就绪：该位为 1 时，表示接收移位寄存器接收到的新字节完全移入接收缓冲寄存器，即接收操作就绪，可供 CPU 读取；读取线状态寄存器时，该位被置 0（复位）
接收时是否检测到错误或条件	D4	检测中断：当接收器检测到空号状态已持续一个完整帧传输时间时，该位置 1；它表明发送方处于间断状态。读取该寄存器，该位被置 0
	D3	帧同步错误：一段输入字符没有有效的终止位，检测到停止位不正确（不完整、丢失或变成空号）时，该位置 1；它表明出现了一帧格式错，可能是噪声行，或者发送器与接收器的波特率不匹配。读取该寄存器，该位被复位
	D2	奇偶校验错误：当一个接收到的字节的奇偶校验位与线控制寄存器中设置的奇偶校验位的校验值不匹配时，该位置 1；表明接收到的字节出现奇偶校验错。读取该寄存器，该位被复位
	D1	溢出错误：当前一字节尚未被 CPU 取走，又接收到新字节时，该位置 1；表明接收缓冲寄存器中的一个字节已被一个最新收到的字节覆盖，导致前一字节丢失，即出现超越错

6.1.5　中断 I/O 通信方式相关设置

1. 中断允许

通过对中断允许寄存器的低 4 位操作，可产生 UART 支持的四类中断。由前述内容已知，当线路控制寄存器 D7 位为 0 时，寄存器 1 即为中断允许寄存器，其各位定义如表 6-1-6 所示。

表 6-1-6　中断允许寄存器各位含义

位	含义
D7～D4	保留，恒为零
D3	该位为 1 表示允许 MODEM 状态变化中断：即在任何一个计算机的 RS-232C 输入端改变状态时，产生一个中断
D2	该位为 1 时，表示当接收到有错信息（包括奇偶校验错、超越错、帧格式错）或间断条件时，允许接收器产生中断
D1	该位为 1 时，表示允许发送保持寄存器空中断，即当从发送器的保持寄存器中移一个字节到发送器的移位寄存器时，产生一个中断
D0	该位为 1 时，表示允许接收器数据就绪中断，即当接收器的缓冲寄存器中有待读的有效字节时，产生一个中断

若要禁止某类中断，只需将相应位置 0 即可。对于采用查询 I/O 通信方式的寄存器，应将中断允许寄存器置 0。

2. 中断标识

中断标识寄存器用来保持优先级最高的中断请求的标识码，在 CPU 处理这个特定的中断请求之前，不接受其他的中断请求。因此，在中断服务程序内部必须通过读取中断标识寄存器才能识别出确切的中断源，然后转入相应的中断处理子程序。

中断标识寄存器是只读寄存器，**D7～D3** 恒为零，其余各位如表 6-1-7 所示。

表 6-1-7　中断标识寄存器

D2D1D0	优先级	中断源类型	中断源	寄存器复位
001		无	无	
110	1（最高）	接收有错或间断条件	奇偶校验错、超越错、帧格式错、间断条件	读线路状态寄存器
100	2	接收的数据就绪	接收数据就绪	读接收缓冲寄存器
010	3	发送保持寄存器空	发送数据就绪	写发送保护寄存器或读中断标识寄存器
000	4	MODEM 状态变化	输入状态变化	读 MODEM 状态寄存器

表 6-1-5-2 中包含了 UART 的 8250 系列 4 个不同的中断类型，如果程序中设置了中断允许，在中断服务程序中，就可以对相应中断进行处理。

注意：要注意的是，程序中的中断处理代码应尽量短，一般只修改数据值，而不做复杂的处理，这样会使得系统中断开销少，否则容易引起程序异常。

6.1.6　MODEM 寄存器

MODEM 控制寄存器与 MODEM 状态寄存器提供了 UART 与 RS-232C(或 MODEM)之间联络应答信号的输入输出状态，也就是完成对 MODEM 控制逻辑的管理。MODEM 控制逻辑共包括 8 个信号：4 个控制输出信号 $\overline{RTS}$、$\overline{DTR}$、$\overline{OUT1}$、$\overline{OUT2}$ 和 4 个控制输入信号 $\overline{DCD}$、$\overline{RI}$、$\overline{DSR}$、$\overline{CTS}$。

1. MODEM 控制寄存器

MODEM 控制寄存器用于设置输出信号，D7～D5 恒为零，其余各位如表 6-1-8 所示。

表 6-1-8　MODEM 控制寄存器

位	含义
D4	该位提供一个循环反馈特性，供诊断测度 INS8250
D3	该位置 1 时，$\overline{OUT2}=0$，使得允许 INTRPT 中断请求输出；为 0 时，则禁止 INTRPT 中断请求输出。因此，对于中断方式必须将该位置 1；而查询方式则清 0
D2	该位置 1 时，$\overline{OUT1}=0$
D1	该位置 1 时，$\overline{RTS}=0$，输出有效，即 UART 请求发送
D0	该位置 1 时，$\overline{DTR}=0$，输出有效，即 UART 数据中断就绪

2. MODEM 状态寄存器

MODEM 状态寄存器（MSR：MODEM Status Register）用于提供状态输入信号。其各位定义如表 6-1-9 所示。

表 6-1-9　MODEM 状态寄存器

位	表征内容	定义
D7	反映 MODEM 控制逻辑的四个输入信号的当前状态	该位为 1，表示载波检测有效
D6		该位为 1，表示振铃指标有效
D5		该位为 1，表示数据设备就绪有效
D4		该位为 1，表示清除发送有效
D3	反映 MODEM 控制逻辑的四个输入信号的状态变化（记为 δ）	该位为 1，表示 δ 载波检测，CD 中发生变化
D2		该位为 1，表示 δ 振铃指示，RI 中发生变化
D1		该位为 1，表示 δ 数据设备就绪，DSR 中发生变化
D0		该位为 1，表示 δ 清除发送，CTS 中发生变化

当 MODEM 状态寄存器的任何一位为 1，且中断允许寄存器的 D3＝1 时，将产生 MODEM 状态变化中断。执行 8250 驱动程序时，在任一时刻，如果在 4 个调制解调器的状态线路中，某一个状态发生变化，则都将产生中断。在中断服务例程中，读入 MSR 的内容，并且通过查看 0～3 位来确定哪些位发生了变化。如果打开了硬件握手，那第 DSR 位和 CTS 位的任一个事件标志都能够触发产生或终止一次发送器中断。

如同线路状态寄存器（LSR）一样，状态位将永远反映输入线的当前状态，但是，在读过事件位一次后将清除它。

6.2　COMRXTX 程序实例

学习了上一节的 DOS 环境下 PC 机的串口设置基本知识后，现在尝试利用这些知识来改进第 1 章的程序 comrx.cpp。由于该程序只实现了中断接收功能，这里的具体目标就是为它增加发送功能，并将程序命名为 comrxtx.cpp，然后进行程序测试。

发送数据一般情况下可不用中断，而是直接将数据送入发送缓冲区，但送入之前，必须知道发送发送器是否为空，即发送器中的所有字节都已发送。如果还没有空就送入字符，则会造成字符丢失。查看表 6-1-5 可知线路状态寄存器 LSR 各位的含义，当 D6 为 1 时，表示发送器的发送寄存器和移位寄存器中都没有字节。通常在结束发送之前，查询该位以防止丢失待发送的数据。

再查表 6-1-5 可知，线路状态寄存器 LSR 对基地址的偏移值为 5，对于 COM1，就是 0x3F8+5。只要检测到线路状态寄存器 LSR 的 D6 位不为 1，就等待，否则，向发送器送入待发数据。因此，可以为程序增加如下的发送函数 send_char()，一次发送一个字符。

```
/* send char 发送字符函数 */
void send_char(unsigned char unch)
{
  while ( ((inp( 0x3f8 + 5)) & 0x40 ) == 0);  //和 0x40 相与，可取出 D6 位进行判断
  outportb(0x3f8,unch);
}
```

同时，在主程序中还需要增加发送字符的代码，下面列出主函数代码：

```
//////////////////////////////////////////////////////////////////
//COMRXTX.CPP for asyn serial communication (RX/TX)
//edited by Xiong Guangming and Gong Jianwei
//Turbo C++3.0
//////////////////////////////////////////////////////////////////

//以下为主函数
void main()
{
    unsigned char unChar;
    short bExit_Flag=0;

    OpenPort(); //打开串口

    fprintf(stdout, "\n\nReady to Receive DATA\n"
            "press [ESC] to quit...\n\n");

    do {
        if (kbhit())
        {
            unChar=getch();
            /* Look for an ESC key */
            switch (unChar)
            {
            case 0x1B:  //ESC的ASCII值为27
                bExit_Flag = 1; /* Exit program */
                break;
            //You may want to handle other keys here
            }
            if(!bExit_Flag)
                send_char(unChar); //发送键入的字符
        }
        unChar = read_char(); //从缓冲区中读数
        if (unChar != 0xff)
        {
            fprintf(stdout,"%c",unChar);
        }
    } while (!bExit_Flag);

    ClosePort(); //关闭串口
}
```

程序的测试仍然与 1.3 节相似，在 Turbo C++ 3.0 中编译运行程序后，得到了 comrxtx.exe 可执行文件。连接好串口线后（同一台计算机的两个串口或不同计算机的串口），首先在串口调试助手中，将串口号设置为 COM2(串口 2)、波特率 9600bps、8 个数据位、1 个停止位、无奇偶校验，并在发送输入框中填入“123456789ABCDEFG”，后加换行（直接按回车键即可），选上自动发送，自动发送周期为 1000ms。

如图 6.2.1 所示，打开 MS-DOS 窗口（Windows 98 从开始→程序→MS-DOS 方式；Windows 2000 从开始→程序→附件→命令提示符），运行 comrx 就可以看到 COM1（串口 1）接收的数据，就可以看到从串口调试助手发来的数据了。

同时，在键盘上随意键入一些字符，可以看到，在串口调试助手的接收编辑框中就看到了这些字符（注意要在激活 MS-DOS 窗口时键入字符），如图 6.2.1 所示。

图 6.2.1　在 MS-DOS 窗口中测试 Comrxtx 程序

6.3　通用实例程序 GSerial 类

1.3 节的 comrx 和 6.2 节的 comrxtx 程序缺少通用性，本节将介绍一个通用性较好的可重用代码 GSerial 类。首先详细说明该类的成员函数和编程考虑，然后将编写一个单串口控制实例。下一节将应用该代码控制多串口。

GSerial 类由 2 个文件构成：GSerial.h 和 GSerial.cpp，GSerial.h 对串口编程中要用到的常量和类进行了定义。代码如下：

```
/*-----------------------------------------------------------------------
  FILENAME: GSERIAL.H
  This file is used to define const and class for GSERIAL.CPP
  此文件用于定义常量和serial类的变量
-----------------------------------------------------------------------*/

#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>


//Define Serial port Const 定义串口逻辑名常量
#define COM1          1
#define COM2          2
```

```
#define COM3          3
#define COM4          4
#define COM5          5
#define COM6          6

#define COM1BASE      0x3F8   /* 基地址 COM1  */
#define COM2BASE      0x2F8   /* 基地址 COM2  */
#define COM3BASE      0x3E8   /* 基地址 COM3  */
#define COM4BASE      0x2E8   /* 基地址 COM4  */
#define COM5BASE      0x3A8   /* 基地址 COM5  */
#define COM6BASE      0x2A8   /* 基地址 COM6  */

/*
8250 异步通信适配器（UART）通过 7 个端口地址控制 10 个寄存器。
```

表 6-3-1　8250 内部寄存器

寄存器	缩写	偏移	COM1	COM2	A2A1A0	D7
接收数据寄存器	RXR	0	3F8H	2F8H	000	0
发送数据寄存器	TXR	0	3F8H	2F8H	000	
中断允许寄存器	IER	1	3F9H	2F9H	001	
中断标识寄存器	IIR	2	3FAH	2FAH	010	×
线路控制寄存器	LCR	3	3FBH	2FBH	011	
MODEM 控制寄存器	MCR	4	3FCH	2FCH	100	
线路状态寄存器	LSR	5	3FDH	2FDH	101	
MODEM 状态寄存器	MSR	6	3FEH	2FEH	110	
波特率除数锁存器低	LSB	0	3F8H	2F8H	000	1
波特率除数锁存器高	MSB	1	3F9H	2F9H	001	

```
*/

/*The following is the adress of the registers          DLAB status   */
#define TXR        0     /* Transmit register (WRITE)  0   */
#define RXR        0     /* Receive register  (READ)   0   */
#define IER        1     /* Interrupt Enable              x   */
#define IIR        2     /* Interrupt ID                  x   */
#define LCR        3     /* Line control                  x   */
#define MCR        4     /* Modem control                 x   */
#define LSR        5     /* Line Status                   x   */
#define MSR        6     /* Modem Status                  x   */
#define DLL        0     /* Divisor Latch Low             1   */
#define DLH        1     /* Divisor latch High            1   */


/*-------------------------------------------------------------------*
 Bit values held in the Line Control Register  LCR 控制数据格式，可对照 6.1.3 节进行理解
    位        意义
    ---       -------
    0-1       00=5 bits, 01=6 bits, 10=7 bits, 11=8 bits.     /数据位
    2         Stop bits.                                      /
    3         0=parity off,  1=parity on.                  /Parity Enable=1
    4         0=parity odd,  1=parity even.                   /Odd or Even select
    5         Sticky parity.                                  /
    6         Set break.
    7         Toggle port addresses.                          /1:access
 *-------------------------------------------------------------------*/
#define LCR_NO_PARITY       0x00
#define LCR_EVEN_PARITY     0x18
#define LCR_ODD_PARITY      0x08
```

```
/*-----------------------------------------------------------------*
 Bit values held in the Line Status Register (LSR).线路状态寄存器
    bit       meaning
    ---       -------
    0         Data ready.
    1         Overrun error - Data register overwritten.
    2         Parity error - bad transmission.
    3         Framing error - No stop bit was found.
    4         Break detect - End to transmission requested.
    5         Transmitter holding register is empty.
    6         Transmitter shift register is empty.
    7            Time out - off line.
 *-----------------------------------------------------------------*/
#define LSR_RCVRDY        0x01
#define LSR_OVRERR        0x02
#define LSR_PRTYERR       0x04
#define LSR_FRMERR        0x08
#define LSR_BRKERR        0x10
#define LSR_XMTRDY        0x20
#define LSR_XMTRSR        0x40
#define LSR_TIMEOUT          0x80

/*-----------------------------------------------------------------*
 Bit values held in the Modem Output Control Register (MCR).
    bit       meaning
    ---       -------
    0         Data Terminal Ready. Computer ready to go.
    1         Request To Send. Computer wants to send data.
    2         auxillary output #1.
    3         auxillary output #2.(Note: This bit must be
              set to allow the communications card to send
              interrupts to the system)
    4         UART ouput looped back as input.
    5-7       not used.
 *-----------------------------------------------------------------*/
#define MCR_DTR           0x01
#define MCR_RTS           0x02
#define MCR_INT                   0x08


/*-----------------------------------------------------------------*
 Bit values held in the Modem Input Status Register (MSR).
    bit       meaning
    ---       -------
    0         delta Clear To Send.
    1         delta Data Set Ready.
    2         delta Ring Indicator.
    3         delta Data Carrier Detect.
    4         Clear To Send.
    5         Data Set Ready.
    6         Ring Indicator.
    7         Data Carrier Detect.
 *-----------------------------------------------------------------*/
#define MSR_CTS           0x10
#define MSR_DSR           0x20
```

```
/*-----------------------------------------------------------------*
 Bit values held in the Interrupt Enable Register (IER).
    bit       meaning
    ---       -------
    0         Interrupt when data received.
    1         Interrupt when transmitter holding reg. empty.
    2         Interrupt when data reception error.
    3         Interrupt when change in modem status register.
    4-7       Not used.
 *-----------------------------------------------------------------*/
#define IER_RX_INT        0x01
#define IER_TX_INT        0x02


/*-----------------------------------------------------------------*
 Bit values held in the Interrupt Identification Register (IIR).
    bit       meaning
    ---       -------
    0         Interrupt pending
    1-2          Interrupt ID code
              00=Change in modem status register,
              01=Transmitter holding register empty,
              10=Data received,
              11=reception error, or break encountered.
    3-7       Not used.
 *-----------------------------------------------------------------*/
#define IIR_MS_ID         0x00  // Modem status
#define IIR_RX_ID         0x04  // Received data CK,and to read the receive buffer
#define IIR_TX_ID         0x02  // Transmitter holding register empty
#define IIR_MASK          0x07  // Get the low 3 bits of IIR


/*
    These are the port addresses of the 8259 Programmable Interrupt
   Controller (PIC).
*/
#define PIC8259_IMR          0x21  /* Interrupt Mask Register port */
#define PIC8259_ICR          0x20  /* Interrupt Control Port      */


/*
   An end of interrupt needs to be sent to the Control Port of
   the 8259 when a hardware interrupt ends.
*/
#define PIC8259_EOI          0x20  /* End Of Interrupt */


/*
    The (IMR) tells the (PIC) to service an interrupt only if it
    is not masked (FALSE).
```

以下之所以将有的中断号全部定义上，是因为有时进行多串口编程时要用到这些中断号，比如在 PC104 上，只要没有用完的中断，就可以用于串口编程。

```
"COM? "中的"? "号表示串口号的中断号可根据情况而定，但 PC 机的串口中断号一般已确定。
*/
#define IRQ0          0xFE  // COM? 1111 1110
#define IRQ1          0xFD  // COM? 1111 1101
#define IRQ2          0xFB  // COM? 1111 1011
#define IRQ3          0xF7  // COM2 1111 0111  /* COM2 */
#define IRQ4          0xEF  // COM1 1110 1111  /* COM1 */
```

```
#define IRQ5          0xDF  // COM? 1101 1111
#define IRQ6          0xBF  // COM? 1011 1111
#define IRQ7          0x7F  // COM? 0111 1111

#define IRQ8          0xFE  // COM? 1111 1110
#define IRQ9          0xFD  // COM? 1111 1101
#define IRQ10         0xFB  // COM? 1111 1011
#define IRQ11         0xF7  // COM? 1111 0111
#define IRQ12         0xEF  // COM? 1110 1111
#define IRQ13         0xDF  // COM? 1101 1111
#define IRQ14         0xBF  // COM? 1011 1111
#define IRQ15         0x7F  // COM? 0111 1111


#define FALSE                 0
#define TRUE                  1

#define ESC                       0x1B   /* ASCII Escape character */
#define ASCII                 0x007F  /* Mask ASCII characters  */


#define NO_ERROR              0      /* 无错误No error            */

#define BUF_OVFL              1      /* 缓冲区溢出Buffer overflowed     */

#define SBUFSIZ                   512  /* Serial buffer size    */
#define IBUF_LEN              2048   // 接收缓冲区Incoming buffer
#define OBUF_LEN              1024   // 发送缓冲区Outgoing buffer

    unsigned int PortBaseAddr[6]= {COM1BASE, COM2BASE, COM3BASE, COM4BASE, COM5BASE,
COM6BASE};

            // 70-8  71-9  72-10  73-11  74-12  75-13 76-14 77-15
    int InterruptNo[6]= { 0x0C, 0x0B, 0x0D,  0x72,  0x73,  0x77};//4,3,5,10,11,15
    int ComIRQ[6] =    { IRQ4, IRQ3, IRQ5,  IRQ10, IRQ11, IRQ15};


//////////////////////GSerial 类//////////////////////////////////////////////////
 class GSerial{
    int flag;

public:

    unsigned int  m_unPortNo; //串口号: COM1, COM2 etc.
    unsigned int  m_unPortBase; //串口基地址: 0x3f8,0x2f8 etc.
    GSerial(void);  //类构造函数
    ~GSerial(void); //类析构函数

    int InitSerialPort(int Port, int Speed, int Parity, int Bits, int StopBit); //初
始化串口
    void CloseSerialPort(void); //关闭串口, 释放串口资源
    int SetDataFormat(int Parity, int Bits, int StopBit); //设置数据模式: 奇偶位等
    int SetSpeed(int Speed); //设置波特率: 串口传输速率
    int SetPortBaseAddr(int Port); //设置串口基地址
    void CommOn(void);     //开启串口
    void CommOff(void);    //关闭串口

    int ReadStatus(void);  //读串口状态LSR
    void SendChar(unsigned char unCh); //发送单个字符
    void SendString(int nStrlen, unsigned char *unChBuf); //发送字符串
```

```
    void interrupt(*OldVects)(...);  //中断向量，用于存放老中断向量，中断前保护好现场

    void SetVects(void interrupt(*New_Int)(...)); //设置中断服务程序的中断向量
    void ResetVects(void);   //恢复中断向量，程序结束或关闭串口前恢复现场

};
```

GSerial.cpp 文件中 GSerial 类的成员函数的实现代码，对一些串口的数据变量进行了初始化和说明：

```
/*-----------------------------------------------------------------*
    GSERIAL.CPP
    For asynchronous serial port communications
    适用于DOS环境下异步串口通信编程
   ATTENTION: Compile this program with Test Stack Overflow OFF.
   在Turbo C++3.0中选项设置 Options/Compiler/Entry 中关闭Test Stack Overflow
*-----------------------------------------------------------------*/

#include "GSerial.h"

    char  inbuf[IBUF_LEN];        // 接收数据buffer
    char  outbuf[OBUF_LEN];       // 发送数据 buffer
//数据缓冲区有关变量
    unsigned int  startbuf = 0;
    unsigned int  endbuf  = 0;
    unsigned int  inhead  = 0;
    unsigned int  intail   = 0;
    unsigned int  outhead  = 0;
    unsigned int  outtail  = 0;
//串口基地址:
    unsigned int  PortBase = 0;

GSerial::GSerial()
{
}

GSerial::~GSerial()
{
}

//* 读串口LSR状态 */
int GSerial::ReadStatus(void)
{
 return(inp(m_unPortBase+5));
}

/*  发送一个字符  */
void GSerial::SendChar(unsigned char unCh)
{
  while ((ReadStatus() & 0x40) == 0);
  outportb(m_unPortBase,unCh);
}

/*  发送一个字符串 */
void GSerial::SendString(int nStrlen, unsigned char *unChBuf)
{
    int k=0;
    do {
```

```
            SendChar(*(unChBuf + k));
            k++;
        } while ((k < nStrlen));
}

/* 装入新的中断向量：中断服务程序地址 */
void GSerial::SetVects(void interrupt(*New_Int)(...))
{
    disable();
    OldVects = getvect(InterruptNo[m_unPortNo-1]);
    setvect(InterruptNo[m_unPortNo-1], New_Int);
    enable();
}

/* 恢复中断向量 */
void GSerial::ResetVects(void)
{
    setvect(InterruptNo[m_unPortNo-1], OldVects);
}

/*启动串口 */
void GSerial::CommOn(void)
{
    int temp;
    disable();
    //temp = inportb(m_unPortBase + MCR) | MCR_INT;
    //outportb(m_unPortBase + MCR, temp);
    outportb(m_unPortBase + MCR, MCR_INT);
    //temp = inportb(m_unPortBase + MCR) | MCR_DTR | MCR_RTS;
    //outportb(m_unPortBase + MCR, temp);
    temp = (inportb(m_unPortBase + IER)) | IER_RX_INT;//|IER_TX_INT;
    outportb(m_unPortBase + IER, temp);
    temp = inportb(PIC8259_IMR) & ComIRQ[m_unPortNo-1];
    outportb(PIC8259_IMR, temp);
    enable();
}

/* 关闭串口 */
void GSerial::CommOff(void)
{
     int  temp;
    disable();
    temp = inportb(PIC8259_IMR) | ~ComIRQ[m_unPortNo-1];
    outportb(PIC8259_IMR, temp);
    outportb(m_unPortBase + IER, 0);
    outportb(m_unPortBase + MCR, 0);
    enable();
}

/* 设置串口号 */
int GSerial::SetPortBaseAddr(int Port)
{
    if((Port<1)||(Port>6))
        return(-1);
    m_unPortNo = Port;
    m_unPortBase = PortBaseAddr[m_unPortNo-1];
    return (0);
}

/* 设置波特率 */
```

```
/* Setting the speed requires that the DLAB be set on.      */
int GSerial::SetSpeed(int Speed)
{
    char c;
    int      divisor;
    if (Speed == 0)          /* Avoid divide by zero */
        return (-1);
    else
        divisor = (int) (115200L/Speed);
    if (m_unPortBase == 0)
        return (-1);

    disable();
    c = inportb(m_unPortBase + LCR);
    outportb(m_unPortBase + LCR, (c   0x80)); /* Set DLAB */
    outportb(m_unPortBase + DLL, (divisor & 0xC0FF));
    outportb(m_unPortBase + DLH, ((divisor >> 8) & 0x00FF));
    outportb(m_unPortBase + LCR, c);        /* Reset DLAB */
    enable();
    return (0);
}

/* 设置数据格式: 奇偶校验、数据位、停止位 */
int GSerial::SetDataFormat(int Parity, int Bits, int StopBit)
{
    int setting;
    if (m_unPortBase == 0)
        return (-1);
    if (Bits < 5 || Bits > 8)
        return (-1);
    if (StopBit != 1 && StopBit != 2)
        return (-1);
    if (Parity != LCR_NO_PARITY && Parity != LCR_ODD_PARITY && Parity != LCR_EVEN_PARITY)
        return (-1);
    setting = Bits-5;
    setting |= ((StopBit == 1) ? 0x00 : 0x04);
    setting |= Parity;

    disable();
    outportb(m_unPortBase + LCR, setting);
    enable();
    return (0);
}

//关闭串口
void GSerial::CloseSerialPort(void)
{
    CommOff();
    ResetVects(); //恢复中断向量
}

/* 初始化串口: 设置串口号、数据格式 */
int GSerial::InitSerialPort(int Port, int Speed, int Parity, int Bits, int StopBit)
{
    int flag = 0;
    if (SetPortBaseAddr(Port))
      flag = -1;
    if (SetSpeed(Speed))
      flag = -1;
    if (SetDataFormat(Parity, Bits, StopBit))
```

```
        flag = -1;
        return(flag);
    }

    //注意：本中断服务不是 GSerial 类成员函数
    void interrupt ComIntServ(...)
    {
        int temp;
        disable();
        temp = (inportb(PortBase+IIR)) & IIR_MASK;              // why interrupt was
called
        switch(temp)
        {
            case 0x00: // modem status changed
                inportb(PortBase+MSR);  // read in useless char
                break;
            case 0x02: // 可以发送数据（该功能本例程没有用上，可根据需要添加
                if (outhead != outtail)                                // 在数据要发送
                {
                    outportb(PortBase+TXR,outbuf[outhead++]); // send the character
                    if (outhead == OBUF_LEN)
                        outhead=0;                    // if at end of buffer, reset pointer
                }
                break;
            case 0x04: // 串口接收器中有数据
                inbuf[inhead] = inportb(PortBase+RXR); // 读取字符数据
                inhead++;
                if (inhead == IBUF_LEN) // if at end of buffer
                    inhead=0;         // reset pointer
                break;
            case 0x06: // line status has changed
                inportb(PortBase+LSR);    // read in useless char
                break;
            default:
                break;
        }
        outportb(PIC8259_ICR, PIC8259_EOI);   // 硬件中断结束
        enable();     // reenable interrupts at the end of the handler

    }

    //本函数也不是 GSerial 类成员函数，这是笔者为方便多串口编程而设计的
    char ReadChar(void)
    {
        char ch;
        if (inhead != intail)    // there is a character
        {
            disable();                    // disable irqs while getting char
            ch = inbuf[intail++];           // get character from buffer
            if (intail == IBUF_LEN)               // if at end of in buffer
                intail=0;                         // reset pointer
            enable();                     // re-enable interrupt
            return(ch);                 // return the char
        }
        ch = -1;
        return(ch);                     // return nothing
    }
```

以下部分为程序主函数，在 Turbo C 的 DOS 程序中，必须定义一个主函数。在主函数中，

首先定义了一个 GSerial 类对象 gs，再调用 GSerial 类成员函数初始化串口、打开串口，在保护好中断现场后（保存中断向量），打开串口。用一个无穷循环读取串口数据，也可以从键盘敲入字符，从打开的串口发送出去，检测到 Esc 键退出循环。退出循环后，在结束程序之前，必须恢复现场（将初始化时保存的中断向量再送入中断表中）。

```
///////主函数//////////////////////////////////////////
main()
{
    /* 通信参数 Communications parameters */
    int     port    = COM1;
    int     speed   = 9600;
    int     parity  = LCR_NO_PARITY;
    int     bits    = 8;
    int     stopbits = 1;
    int     done = FALSE;
    char    c;
    int temp;
    int SError=0;

    GSerial gs; //定义 GSerial 类对象

//以下代码调用 GSerial 类成员函数初始化串口、打开串口
    if (!gs.InitSerialPort(port, speed, parity, bits, stopbits))
    {
        PortBase = PortBaseAddr[port-1]; //得到串口基地址：中断服务程序中要用到
        gs.SetVects(ComIntServ); //装入中断服务程序向量
        gs.CommOn();   //打开串口
    }
    else
      SError=2; //如果串口打开出错，则设置错误代号，这会导致退出程序

//以下代码打印出串口号、基地址、中断号信息
    fprintf(stdout, "\nCOM%d, PortBase=0X%x, IntVect=0X%x\n\n", gs.m_unPortNo,
gs.m_unPortBase, ComIRQ[gs.m_unPortNo-1]);

  //打印出终端状态，程序按 Esc 键退出
    fprintf(stdout, "TURBO C TERMINAL\n"
            "...You're now in terminal mode, "
            "press [ESC] to quit...\n\n");

    /*
      以下用一循环读取串口数据，也可以从键盘敲入字符，从打开的串口发送出去
    */
    do {
        if (kbhit())
        {
            c = getch();
            /* 是否有 Esc 键，如有，则退出循环*/
            switch (c)
            {
                case ESC:
                    done = TRUE; /* Exit program */
                    break;
            //这里可对其他键进行响应处理
            }
            if (!done)
            {
              gs.SendChar( c ); //如果不是 Esc 键，则从串口将键入的字符发送
```

```
                fprintf(stdout,"%c",c); //同时，在屏幕上显示该字符
            }
        }
        c = ReadChar();  //读接收数据缓冲区
        if (c != -1)    //'-1' is the END signal of a string
        {
            fprintf(stdout,"%c",c);  //在屏幕上显示接收到的字符
        }
    // fprintf(stdout,"%d",testtemp);

    } while ((!done) && (!SError));

    gs.CloseSerialPort();  //关将串口，恢复现场

    /* 下面是错误显示或正常退出*/
    switch (SError)
    {
        case NO_ERROR: fprintf(stderr, "\nbye.\n");
                       return (0);

        case BUF_OVFL: fprintf(stderr, "\nBuffer Overflow.\n");
                       return (99);
        case 2:  fprintf(stderr,"\n Cannot init serial port");
                       return(2);
        default:     fprintf(stderr, "\nUnknown Error, SError = %d\n",
                                SError);
                       return (99);
    }
}
```

程序的测试仍然与 1.3 节和上一节相似，在 Turbo C++ 3.0 中编译运行程序后，得到了 GSerial.exe 可执行文件，连接好串口线后（同一台计算机的两个串口或不同计算机的串口，这里采用在同一台计算机上进行测试），首先在串口调试助手中，将串口号设置为 COM2(串口 2)、波特率 9600bps、8 个数据位、1 个停止位、无奇偶校验，并在发送输入框中填入“123456789ABCDEFG”，后加换行（直接按回车键即可），选上自动发送，自动发送周期为 1000ms。

如图 6.3.1 所示，打开 MS-DOS 窗口（Windows 98 从开始→程序→MS-DOS 方式；Windows2000 从开始→程序→附件→命令提示符），运行 GSerial 就可以看到 COM1（串口 1）接收的数据，就可以看到从串口调试助手发来的数据了。同时，在键盘上随意键入一些字符，可以看到，在串口调试助手的接收编辑框中就看到了这些字符（注意要在激活 MS-DOS 窗口时键入字符）。

图 6.3.1　GSerial 单串口控制程序测试

6.4　用 GSerial 类控制多串口

GSerial 类是一个 DOS 下可重用串口通信程序代码，程序结构较好，可以方便地编写出用中断方式接收/发送的多串口通信程序。笔者曾用它编写过 7 串口通信程序，需要在 Turbo C++ 3.0 编译器下运行，适应的操作系统为 MS DOS 或 Windows 3x/9x/Me/NT/2K 的 DOS 环境。本节以一个 2 串口通信的程序作为实例，如果在中断编程中用到了高号中断（IRQ8~IRQ15），则要注意对 8259A 可编程中断控制器的控制方式，这可参考 6.5 节的实例说明。

应用 GSerial 类来控制多串口，要掌握以下几个原则。

- **每个串口对应一个 GSerial 类对象**：GSerial 类的代码是可重用代码，因此只要为每个串口定义一个 GSerial 类对象。
- **每个串口都有自己的中断服务函数**：由于各串口接收数据时，处理代码会有所不同，会有各自存放数据的变量，这些都需要在自己的中断服务程序中完成。定义好 GSerial 类对象后，只要将中断服务函数的入口地址指针送到相应的中断号中。

在本节的实例程序中，GSerial.h 文件不做任何改变，但读者如果要改变中断号，则要自己根据实际情况来设置以下一些变量，以改变缓冲区的大小、中断号等。

```
#define SBUFSIZ                512  /* Serial buffer size    */
#define IBUF_LEN             2048   // 接收缓冲区 Incoming buffer
#define OBUF_LEN             1024   // 发送缓冲区 Outgoing buffer

    unsigned int PortBaseAddr[6]= {COM1BASE, COM2BASE, COM3BASE, COM4BASE, COM5BASE,
COM6BASE};

    // 70-8  71-9  72-10  73-11  74-12  75-13 76-14 77-15
    int InterruptNo[6]= { 0x0C, 0x0B, 0x0D, 0x72, 0x73, 0x77};//4,3,5,10,11,15
    int ComIRQ[6] =    { IRQ4, IRQ3, IRQ5, IRQ10, IRQ11, IRQ15};
```

为了方便了解程序结构，我们先明确编程任务。

编程任务：

程序为终端方式。在配有两个串口的计算机上，连接好串口线。在键盘上任意键入一个字符，该字符从 COM1 发送去，COM2 收到该字符后，显示在屏幕上，并把该字符再返回发送给 COM1，COM1 收到后显示在屏幕上。为方便观察，在屏幕上标明串口号及收发标志，如串口 1 发送用[COM1:TX]标明，接收用[COM1:RX]标明。

现在开始编程。与上一节不同的是，为了让每个串口方便地设置自己的变量，我们加入了一个结构 COMPORT_VAR，以定义两个串口共同的变量 comport1 和 comport2。

再根据以上的控制原则，为每个串口定义了一个中断服务程序，分别为 ComIntServ_comport1 和 ComIntServ_comport2，再各自定义一个读接收缓冲区的函数 ReadChar_comport1 和 ReadChar_comport2。其他编程的考虑在以下的代码中说明，GSerial.h 文件没有做任何改变，与上一节相同。

```
/*-----------------------------------------------------------------*
    GSERIAL.CPP
    Edited by Gong jianwei  http://www.gjwtech.com
    For asynchronous serial port communications
    适用于DOS环境下异步串口通信编程 多串口控制程序
  ATTENTION: Compile this program with Test Stack Overflow OFF.
  在Turbo C++3.0中选项设置 Options/Compiler/Entry 中关闭Test Stack Overflow
 *-----------------------------------------------------------------*/
#include "GSerial.h"

struct COMPORT_VAR{
    char  inbuf[IBUF_LEN];        // in buffer
    char  outbuf[OBUF_LEN];       // out buffer
    unsigned int  startbuf ;
    unsigned int  endbuf  ;
    unsigned int  inhead  ;
    unsigned int  intail  ;
    unsigned int  outhead  ;
    unsigned int  outtail  ;
    unsigned int  PortBase ;
};

//定义结构COMPORT_VAR变量，为每个串口定义一个变量
COMPORT_VAR comport1,comport2;


//////////////////////////////////GSerial类////////
GSerial::GSerial(){}
GSerial::~GSerial(){}
```

```
//* 读串口LSR状态 */
int GSerial::ReadStatus(void)
{
  return(inp(m_unPortBase+5));
}

/*  发送一个字符  */
void GSerial::SendChar(unsigned char unCh)
{
  while ((ReadStatus() & 0x40) == 0);
  outportb(m_unPortBase,unCh);
}

/*  发送一个字符串 */
void GSerial::SendString(int nStrlen, unsigned char *unChBuf)
{
     int k=0;
     do {
          SendChar(*(unChBuf + k));
          k++;
     } while ((k < nStrlen));
}

/* 装入新的中断向量: 中断服务程序地址 */
void GSerial::SetVects(void interrupt(*New_Int)(...))
{
     disable();
     OldVects = getvect(InterruptNo[m_unPortNo-1]);
     setvect(InterruptNo[m_unPortNo-1], New_Int);
     enable();
}

/* 恢复中断向量 */
void GSerial::ResetVects(void)
{
     setvect(InterruptNo[m_unPortNo-1], OldVects);
}

/*启动串口 */
void GSerial::CommOn(void)
{
     int temp;
     disable();
     //temp = inportb(m_unPortBase + MCR) | MCR_INT;
     //outportb(m_unPortBase + MCR, temp);
     outportb(m_unPortBase + MCR, MCR_INT);
     //temp = inportb(m_unPortBase + MCR) | MCR_DTR | MCR_RTS;
     //outportb(m_unPortBase + MCR, temp);
     temp = (inportb(m_unPortBase + IER)) | IER_RX_INT;//|IER_TX_INT;
     outportb(m_unPortBase + IER, temp);
     temp = inportb(PIC8259_IMR) & ComIRQ[m_unPortNo-1];
     outportb(PIC8259_IMR, temp);
     enable();
}

/* 关闭串口 */
void GSerial::CommOff(void)
{
     int temp;
```

```
    disable();
    temp = inportb(PIC8259_IMR) | ~ComIRQ[m_unPortNo-1];
    outportb(PIC8259_IMR, temp);
    outportb(m_unPortBase + IER, 0);
    outportb(m_unPortBase + MCR, 0);
    enable();
}

/* 设置串口号 */
int GSerial::SetPortBaseAddr(int Port)
{
    if((Port<1)||(Port>6))
        return(-1);
    m_unPortNo = Port;
    m_unPortBase = PortBaseAddr[m_unPortNo-1];
    return (0);
}

/* 设置波特率 */
/* Setting the speed requires that the DLAB be set on.      */
int GSerial::SetSpeed(int Speed)
{
    char c;
    int      divisor;
    if (Speed == 0)          /* Avoid divide by zero */
        return (-1);
    else
        divisor = (int) (115200L/Speed);
    if (m_unPortBase == 0)
        return (-1);

    disable();
    c = inportb(m_unPortBase + LCR);
    outportb(m_unPortBase + LCR, (c | 0x80)); /* Set DLAB */
    outportb(m_unPortBase + DLL, (divisor & 0x00FF));
    outportb(m_unPortBase + DLH, ((divisor >> 8) & 0x00FF));
    outportb(m_unPortBase + LCR, c);        /* Reset DLAB */
    enable();
    return (0);
}

/* 设置数据格式：奇偶校验、数据位、停止位 */
int GSerial::SetDataFormat(int Parity, int Bits, int StopBit)
{
    int setting;
    if (m_unPortBase == 0)
        return (-1);
    if (Bits < 5 || Bits > 8)
        return (-1);
    if (StopBit != 1 && StopBit != 2)
        return (-1);
    if (Parity != LCR_NO_PARITY && Parity != LCR_ODD_PARITY && Parity != LCR_EVEN_PARITY)
        return (-1);
    setting = Bits-5;
    setting |= ((StopBit == 1) ? 0x00 : 0x04);
    setting |= Parity;

    disable();
    outportb(m_unPortBase + LCR, setting);
    enable();
```

```
    return (0);
}

//关闭串口
void GSerial::CloseSerialPort(void)
{
    CommOff();
    ResetVects(); //恢复中断向量
}

/* 初始化串口: 设置串口号、数据格式 */
int GSerial::InitSerialPort(int Port, int Speed, int Parity, int Bits, int StopBit)
{
    int flag = 0;
    if (SetPortBaseAddr(Port))
      flag = -1;
    if (SetSpeed(Speed))
      flag = -1;
    if (SetDataFormat(Parity, Bits, StopBit))
      flag = -1;
    return(flag);
}
//////////////////////////////////类说明结束//////////////////////////////////////////

//////////////////////////////COM1//////////////////////////////////////////////////
void interrupt ComIntServ_comport1(...)
{
    int temp;
    disable();
    temp = (inportb(comport1.PortBase+IIR)) & IIR_MASK; // why interrupt was called
    switch(temp)
    {
        case 0xC0:  // modem status changed
            inportb(comport1.PortBase+MSR);   // read in useless char
            break;
        case 0x02:  // Request To Send char
            if (comport1.outhead != comport1.outtail)  // there's a char to send
            {
            outportb(comport1.PortBase+TXR,comport1.outbuf[comport1.outhead++]);
            // send the character
                if (comport1.outhaad == OBUF_LEN)
                    comport1.outhead=0;          // if at end of buffer, reset pointer
            }
            break;
        case 0xD4:  // character ready to be read in
            //inbuf[inhead++] = inportb(m_unPortBase+RXR);
            // read character into inbuffer
            comport1.inbuf[comport1.inhead] = inportb(comport1.PortBase+RXR);
// read character into inbuffer
            comport1.inhead++;
            if (comport1.inhead == IBUF_LEN) // if at end of buffer
                comport1.inhead=0;          // reset pointer
            break;
        case 0x06:  // line status has changed
            inportb(comport1.PortBase+LSR);    // read in useless char
            break;
        default:
            break;
```

```
        outportb(PIC8259_ICR, PIC8259_EOI);   // Signal end of hardware interrupt
        enable();          // reenable interrupts at the end of the handler
    }

    //COM1 串口 1 读接收缓冲区
    char ReadChar_comport1(void)
    {
        char ch;
        if (comport1.inhead != comport1.intail)    // there is a character
        {
            disable();                    // disable irqs while getting char
            ch = comport1.inbuf[comport1.intail++];  // get character from buffer
            if (comport1.intail == IBUF_LEN)      // if at end of in buffer
                comport1.intail=0;                // reset pointer
            enable();                   // re-enable interrupt
            return(ch);                 // return the char
        }
        ch = -1;
        return(ch);                     // return nothing
    }

    ////////////////////////////////COM1END/////////////////////////////////////////////
    ////////////////////////////////COM2/////////////////////////////////////////////
    void interrupt ComIntServ_comport2(...)
    {
        int temp;
        disable();
        temp = (inportb(comport2.PortBase+IIR)) & IIR_MASK; // why interrupt was called
        switch(temp)
        {
            case 0x00:  // modem status changed
                inportb(comport2.PortBase+MSR);   // read in useless char
                break;
            case 0x02:  // Request To Send char
                if (comport2.outhead != comport2.outtail)  // there's a char to send
                {

    outportb(comport2.PortBase+TXR,comport2.outbuf[comport2.outhead++]);
// send the character
                    if (comport2.outhead == CBUF_LEN)
                        comport2.outhead=0;          //if at end of buffer, reset pointer
                }
                break;
            case 0x04:  // character ready to be read in
                //inbuf[inhead++] = inportb(w_unPortBase+RXR);
                // read character into inbuffer
                comport2.inbuf[comport2.inhead] = inportb(comport2.PortBase+RXR);
    // read character into inbuffer
                comport2.inhead++;
                if (comport2.inhead == IBUF_LEN) // if at end of buffer
                    comport2.inhead=0;          // reset pointer
                break;
            case 0x06:  // line status has changed
                inportb(comport2.PortBase+LSR);    // read in useless char
                break;
            default:
                break;
        }
        outportb(PIC8259_ICR, PIC8259_EOI);   // Signal end of hardware interrupt
        enable();      // reenable interrupts at the end of the handler
```

```
}

//COM2 串口2 读缓冲函数
char ReadChar_comport2(void)
{
    char ch;
    if (comport2.inhead != comport2.intail)    // there is a character
    {
        disable();                    // disable irqs while getting char
        ch = comport2.inbuf[comport2.intail++];     // get character from buffer
        if (comport2.intail == IBUF_LEN)      // if at end of in buffer
            comport2.intail=0;                          // reset pointer
        enable();                     // re-enable interrupt
        return(ch);            // return the char
    }
    ch = -1;
    return(ch);                // return nothing
}
```

以下为主函数。首先对两个结构中的变量进行了初始化，然后定义通信参数，当然，两个串口的通信参数是一致的，否则会收不到数据。

```
//////
main()
{
    /* 初始化结构变量 */
//COM1
    comport1.startbuf =0;   comport1.endbuf   =0;
    comport1.inhead   =0;   comport1.intail   =0;
    comport1.outhead  =0;   comport1.outtail  =0;
    comport1.PortBase =0;
  //COM2
    comport2.startbuf =0;   comport2.endbuf   =0;
    comport2.inhead   =0;   comport2.intail   =0;
    comport2.outhead  =0;   comport2.outtail  =0;
    comport2.PortBase =0;
  //定义通信参数
    int  port    = COM1;
    int  speed   = 9600;
    int  parity  = LCR_NO_PARITY;
    int  bits    = 8;
    int  stopbits = 1;

    int  done  = FALSE;  //退出循环标志
    char c;
    int  temp;
    int  SError=0;  //错误标志

    GSerial gsCOM1, gsCOM2;  //定义两个GSerial对象

    //初始化COM1，串口1
    if (!gsCOM1.InitSerialPort(port, speed, parity, bits, stopbits))
    {
        comport1.PortBase = PortBaseAddr[port-1];
        gsCOM1.SetVects(ComIntServ_comport1);
        gsCOM1.CommOn();
    }
    else
      SError=2;
```

```
//初始化 COM2，串口 2
port = COM2; //改变端口号
if (!gsCOM2.InitSerialPort(port, speed, parity, bits, stopbits))
{
    comport2.PortBase = PortBaseAddr[port-1];
    gsCOM2.SetVects(ComIntServ_comport2);
    gsCOM2.CommOn();
}
else
  SError=2;

//打印串口地址及中断向量地址，这是为了方便查看设置是否正确
fprintf(stdout, "\nCOM%d, PortBase=0X%x, IntVect=0X%x\n\n",
    gsCOM1.m_unPortNo,gsCOM1.m_unPortBase,ComIRQ[gsCOM1.m_unPortNo-1]);

fprintf(stdout, "\nCOM%d, PortBase=0X%x, IntVect=0X%x\n\n",
    gsCOM2.m_unPortNo,gsCOM2.m_unPortBase,ComIRQ[gsCOM2.m_unPortNo-1]);

fprintf(stdout, "Now we are ready to go: \n\n"); //准备好了 READY!

//开始无穷循环，检测键入字符，检测到 Esc 键退出，其他字符则从 COM1 发送
do {
    if (kbhit())
    {
        c = getch();
        /* 是否有 Esc 键*/
        switch (c)
        {
            case ESC:
                done = TRUE; /* 退出程序 Exit program */
                break;
      //这里可以处理其他特殊键代码
        }
        if (!done) //不退出则将键入字符从 COM1 发送到 COM2
        {
          gsCOM1.SendChar( c );
          fprintf(stdout,"\n\n[COM1:TX]: %c\n",c);
        }
    }
    c = ReadChar_comport1();  //COM1 接收缓冲区中是否有数据
    if (c != -1)    //'-1' is the END signal of a string
    {
        fprintf(stdout,"[COM1:RX]: %c\n",c); //COM1 接收的字符显示在屏幕上
    }

    delay(50); //延迟以方便观察

    c = ReadChar_comport2(); //COM2 接收缓冲区中是否有数据
    if (c != -1)    //'-1' is the END signal of a string
    {
        fprintf(stdout,"[COM2:RX]: %c\n",c); // //COM2 接收的字符显示在屏幕上
        gsCOM2.SendChar( c ); //COM2 把接收的字符发送出去
        fprintf(stdout,"[COM2:TX]: %c\n",c); //在屏幕上显示发送标志
    }

} while ((!done) && (!SError));

//关闭打开的串口
gsCOM1.CloseSerialPort(); //关闭串口 1
gsCOM2.CloseSerialPort(); //关闭串口 2
```

```
        /*检查错误类型 */
        switch (SError)
        {
            case NO_ERROR: fprintf(stderr, "\nbye.\n");
                           return (0);

            case BUF_OVFL: fprintf(stderr, "\nBuffer Overflow.\n");
                           return (99);
            case 2:  fprintf(stderr,"\n Cannot init serial port");
                           return(2);
            default:     fprintf(stderr, "\nUnknown Error, SError = %d\n",
                                 SError);
                           return (99);
        }
    }
```

现在进行程序测试。在 Turbo C++ 3.0 中编译运行程序后，得到了 GSerial.exe 可执行文件，连接好串口线（必须是同一台计算机的两个串口，如果串口号不同，则做出相应改变）。

如图 6.4.1 所示，打开 MS-DOS 窗口（Windows 98 从开始→程序→MS-DOS 方式；Windows2000 从开始→程序→附件→命令提示符），运行 GSerial，在键盘上随意键入几个字符，就可以看到运行效果了。

图 6.4.1　多串口 GSerial 程序测试

6.5　多串口编程PC机高号中断8259A可编程中断控制器的控制

本节说明 PC 机中高号中断（IRQ8~IRQ15）编程时，如何初始化 8259 可编程中断控制器和中断服务程序处理，给出了 Turbo C++编写的 8259 初始化程序和中断服务程序实例。有时在处理多串口时，要对中断进行扩展，很典型的实例就是在 PC104 中对多串口的控制，PC104 与 IBM PC 是兼容的，但留了多余的中断号供编程者使用。

在 IBM PC 及其兼容机中，通过 CPU 的 NMI（非屏蔽中断）和两个 8259A 可编程中断控制器芯片，为系统提供了 16 级中断，硬件中断结构如图 6.5.1 所示。

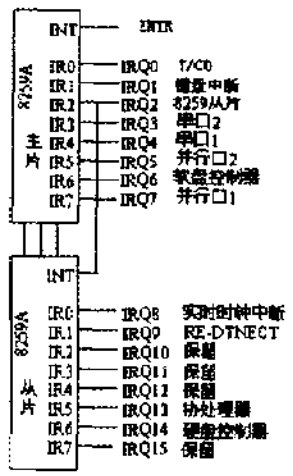


图 6.5.1　PC 机硬件中断结构

两片 8259A 构成主从式级联控制结构，与 CPU 相连的称为主片，下一层的称为从片，从片中断请求信号 INT 与主片的 IRQ2 相连。IBM PC 机中保留给用户可随意编程的中断号有 IRQ10、IRQ11、IRQ12 和 IRQ15，这些中断信号都在 8259A 从片上。8259A 的详细资料请参阅有关手册，本节仅列出大多数 PC 硬件手册中未提及的编程资料，然后说明 PC 机中 8259A 中断控制器的编程初始化过程和如何处理中断服务程序。

IBM PC 机中由 8259A 管理的 16 级中断均有规定的中断向量存储地址，主片中 IRQ0～IRQ7 分别对应 08H～0FH，从片中 IRQ8～IRQ15 分别对应 70H～77H。主片的中断控制寄存器 ICR 和中断屏蔽寄存器 IMR 的口地址分别为 20H 和 21H，从片的相应寄存器口地址分别为 A0H 和 A1H。

中断初始化编程时，当用主片中 IRQ0～IRQ7 时，只需在屏蔽寄存器中打开相应中断，在中断服务程序中，中断结束后，发一次中断结束命令 EOI；而涉及从片中 IRQ7～IRQ15 高号中断时，除在从片中的屏蔽寄存器中打开相对应的中断，还须打开主片中的 IRQ2，且在中断服务程序中，中断结束时，要发两次 EOI 命令，分别使主片和从片执行中断结束命令。下面就 IRQ11 的初始化和中断服务程序处理给出 Turbo C 的源代码编程说明。

首先说明一个中断指针 oldvect 以保存原来的中断向量，在中断服务程序 ser_program() 结束后，分别向主片和从片的中断控制寄存器 ICR 送中断结束信号 EOI。

```
void interrupt(*oldvect)(...); //设置原中断向量保存指针
void interrupt ser_program(...)
{
 {... ...}     //中断服务程序代码
     outportb(0xA0,0x20); //向从片 ICR 送 EOI 命令
     outportb(0x20,0x20); //向主片 ICR 送 EOI 命令
}
```

中断初始化时，要先保存 IRQ11 对应的地址 73H 存储的原中断向量，然后将自己的中断服务程序入口地址装入，再分别打开主片 IRQ2 和从片 IRQ11。

```
void Interrupt_Enable(void)
{    int temp;
     {... ...}      //其他初始化代码
     oldvect = getvect(0x73);      //保存原中断向量
     setvect(0x73,ser_program);    //装入中断服务程序入口地址
     temp = inportb(0x21) & 0xFB;//打开主片 IRQ2
outportb(0x21, temp);
     temp = inportb(0xA1) & 0xF7;//打开从片 IRQ11
     outportb(0xA1, temp);
}
```

最后，不要忘记在程序关闭前关中断和恢复原中断向量。

```
void Interrupt_Disable(void)
{    int temp;
     setvect(0x73, oldvect);   //恢复原中断向量
     temp = inportb(0x21) | ~(0xFB) ;//关主片 IRQ2
     outportb(0x21, temp);
     temp = inportb(0xA1) | ~(0xF7) ;//关从片 IRQ11
     outportb(0xA1, temp);
}
```

以上仅给出了初始化和中断服务程序处理的必要源代码。结合 6.4 节的程序，我们就可以编写出 DOS 操作系统下的多串口控制程序，而且不管中断号高低，都可以应付自如了。

第 7 章　串口通信用户层协议的编制与数据处理方法

[内容提要]

串口通信程序中，为什么要编制用户通信协议呢？为什么要进行数据处理？

在实际工作中编写过串口通信程序的读者马上就能回答这两个问题：为了按规定格式从串口发送数据，也为了从接收到的数据中将需要的信息提取出来。没有在实际工作中体验过串口通信编程的读者也不必着急，本章完全是从实际应用需要来讲述这一问题的，只要您认真读完，处理实际工作中的编程应该是轻车熟路了。

串口通信协议分为底层通信协议和用户层协议。底层协议一般由计算机硬件提供商和设备厂家提供，在一般性的通信编程中很少涉及，而用户层协议则是面向使用者的，也就是我们在编程中通常说到的通信协议。这种用户层的通信协议，简单来说，就是数据以何种格式发送出去，或者如何从接收到的某种格式的数据串中提取出需要的数据，以及在发送和接收过程中如何保证这些数据的正确性，即数据校验。

本章首先以实例方式介绍几种常见的用户层通信协议，以及数据处理方法，然后通过程序实例来演示这些协议在编程实践中的应用方法。

7.1　通信协议的编制

7.1.1　为什么要编制用户通信协议

有过实际工程编程经验的读者当然很容易理解编制通信协议的重要性，但初次接触通信编程的读者大多会提出这个问题：为什么需要编制通信协议？

我们在前面的例程中，大多只是从串口接收数据，并显示在程序界面中，同时也向串口发送数据，至于如何从接收到的数据中提取出有用的信息，以及我们发送出去的数据对方如何处理，我们还没有关心过，但这些在实际工程编程中是必须考虑的，提取有用的信息正是我们进行通信编程的目标所在。

在大多数编程实践中，接收与发送的数据并不需要直接显示在程序界面中，而只是显示对我们有利用价值的几个数据，或者根本不显示，只是在程序内部进行处理，这时，数据发送方和数据接收方就必须事先约定数据发送的格式。这种数据发送格式的约定就是数据通信协议的编制过程。

下面我们利用一个 GPS（Global Positioning System，全球定位系统）接收模块数据处理过程来说明为什么在串口通信中要编制用户协议。

近年来，GPS 系统已经在大地测绘、海上渔用、车辆定位监控、建筑、农业等各个领域得到广泛应用。目前，市场上的大部分 GPS 接收模块都是通过 RS-232 串口与 PC/MCU（计算机/单片机）进行数据传输的。这些数据包括经度、纬度、海拔高度、时间、卫星使用情况等基本信息。我们要从 GPS 接收模块发送出来的数据中提取出我们需要的信息，比如利用经度和纬度进行定位。这时，我们就必须了解 GPS 接收模块发出来的数据格式，也就是用户层的通信协议。

例如，一个 GPS 接收模块发出的 GGA 定位信息如下：

```
$GPGGA,hhmmss,dddmm.mmmm,a,dddmm.mmmm,a,x,xx,x.x,x.x,M,,M,x.x,xxxx*hh<CR><LF>
```

这是一种名叫 NMEA-0183 无线通信输出格式（NMEA：National Marine Electronics Association，全国海洋电子协会[美]）。每次发出如上面格式的一个数据串，由于这是一个整体，所以也把这个数据串称为一个数据包，其中：

- $是串(包)头，GPGGA 是串(包)名；
- hhmmss,dddmm.mmmm,a,dddmm.mmmm,a,x,xx,x.x,x.x,M,,M,x.x,xxxx 是数据内容；
- *是串(包)尾；
- hh 为校验；
- <CR>是回车符，ASCII 码值为 13（十六进制值为 0D）；
- <LF>是换行符，ASCII 码值为 10（十六进制值为 0A）。

之所以加上回车符和换行符，是为了使用接收调试工具（如串口调试助手）观察时方便。平时，我们说把一些数据“打包”发送，实际上也就是发送方与接收方商定一个数据通信格式，依据这个商定的格式（协议）来对要发送的数据加上包头、包名（有时包名并非必要）、包尾、校验方式等信息，接收方收到这些信息后，也根据规定的格式来判断一个数据包的开始与结束，并校验是否正确，如果正确，就把自己需要的信息提取出来。

下面是该 GPS 接收模块发送出来的一个具体的 GPGGA 数据串实例：

```
$GPGGA,033744,2446.5241,N,12100.1536,E,1,10,0.8,133.4,M,,,,*1F
```

因为加上了回车的换行符，在串口调试助手中显示情况如图 7.1.1 所示。如果我们编写的程序是产品级的，也要考虑为使用者调试提供这样的方便。

图 7.1.1 在串口调试助手中显示接收到的 GPS 数据串

但在实际应用中，并不是能够显示这些数据信息就可以了，我们还要实时将有关的数据提取出来，或者在程序里使用，或者将这些数据发送给第三方。为了将这些数据提取出来，

我们就要知道该数据包的格式定义。上面的 GPGGA 数据包中厂家规定的对应描述如表 7-1-1 所示。

表 7-1-1　一个 GAGGA 数据串的说明

区域	名称	例	单位	说明
1	信息 ID	$GPGGA		GGA 协议开始
2	UTC 时间	033744		hhmmss
3	纬度	2446.5241		dddmm.mmmm
4	南/北半球指示	N		N=north ； S=south
5	经度	12100.1536		dddmm.mmmm
6	东/西半球指示	E		E=east ； W=west
7	定位指示	1		0 =未定位； 1=定位 SPS 模式； 2=定位 DGPS, SPS 模式
8	应位卫星数	10		00-12
9	HDOP	0.8	米	
10	海拔高度	133.4	米	
11	海拔高度单位	M	米	
12	WGS84 水准面划分			
13	WGS-84 水准面划分单位			
14	累计 GPS 数据微分			本模块中不可用
15	参考工作站 ID			本模块中不可用
16	校验位	*1F		

了解了厂家提供的数据通信协议格式，我们在编程时就能从上面数据串中读出位置信息：北纬 24 度 46.5241 分、西经 121 度 00.1536 分、格林尼治时间 3 点 37 分 44 秒、应位卫星数 10 以及其他需要利用的信息。

通常把这种通信双方必须遵循的数据描述格式称为通信用户层协议。这种协议可以是一方单独建立，另一方必须依据这样的协议来进行数据接收与发送，通常是产品生产方提供这样的协议；也可以通信双方临时协商约定，双方都按照这样的约定来编程。

读者仔细读完本章后，会对在串口通信编程（其他通信也一样）中为什么要编制用户协议有一个更深刻的认识。

7.1.2　串口通信中用户层协议编制原则

在串口用户层的通信协议中，一般是围绕发送方如何建立数据包和接收方如何处理数据包并从数据包中提取出关心的信息，通信协议也必须是有利于这一目的。编制用户层的通信协议具有很大的随意性，但有几个原则是必须要遵循的。

（1）数据包必须有包头。包头是供接收方判断一个数据包开始传输的重要标志，接收方从接收到的数据中判断接收到了包头，就认为接收的数据已经开始，真正的数据信息马上就会到达。包头字符必须有别于数据信息，这种特征是数据包中其他数据没有的，否则就会引起混乱。

（2）非定长数据包必须有包尾。所谓非定长，是指没有指明数据包的长度。对于非定长的数据包，接收方只能根据包尾标志判断数据包是否结束。同包头一样，包尾字符必须也有别于数据信息，这种特征是数据包中其他数据没有的，否则也会引起混乱。

（3）定长数据包应该指明长度。对于长度不变的数据包，数据长度可以事先约定，也可以在数据包中的约定位置定义；对于长度可变的数据包，则必须每次在数据包中的指定位

置说明。接收方在知道了接收长度后，就能够判断接收的数据包是否结束。

（4）一般应该对数据进行校验。串口通信底层协议（由机器硬件实现）已经设置了奇偶校验方式，在用户层加入校验，可以对数据进行进一步的排错，更好地保证数据的正确性，因为在重要的场合，数据出错可能会引发严重的后果。

（5）便于观察的数据应该在结尾加入换行等符号。对于产品化的设备输出数据，或者需要经常调试的数据，应该在结尾加上换行，以便于在调试工具中查看数据，方便调试程序时观察数据。

（6）更新快的数据，应尽量简短。传输的数据越多，需要的时间越长，但传输速率是有限制的，针对具体的硬件系统有不同的限制，因此，如果要求数据更新快，就要让每次传送的数据尽量短。在下面的用户层协议常用实例中，会对这个问题做进一步的说明。

以上几点是我们在编制用户层协议时应该考虑的，通常我们编程时，还要去适应既定的协议，这些协议也基本上是按照以上规则编制的，因此，了解这些很有用处。

7.1.3　在串口通信中几种常用的用户层协议

在常见的用户层协议中，按照输出数据的可读性，可以分为完整型协议和简单型协议，上面提到的 NMEA-0183 无线通信输出格式协议，包含了包头、包尾、校验、换行，而且数据之间还有逗号分隔，观察数据非常方便，但数据包的长度增加，发送需要的时间也会增加，这在要求更新快的场合是不合适的；简单型协议则去掉了在程序功能中不需要利用的换行及其他分隔符，有时甚至连校验也省掉了。

根据字符的可见性，还可以把用户层协议分成可显示字符型和非可见字符型，如果用串口调试助手等工具来接收，前者可以用 ASCII 字符才显示；后者以 ASCII 字符方式显示时，有时什么也看不到，必须用十六进制方式来显示，有时，以这种方式传输时，能大大缩短传输长度，例如，数据 246 如果全用字符传输，需要占三个字符，而把 246 赋给一个字符以十六进制方式传送出去，就只要占用一个字符。

以上的分类是相对的，有时我们会看到，一些用户层协议具备多种特征。下面是几个实例，说明实际编程过程中如何根据需要来设计通信协议。

1. NMEA-0183 无线通信协议

在这里之所以还要对 NMEA-0183 无线通信协议进行说明，是因为这是一种比较典型的用户层协议。上面已经对 GPS 输出格式中应用该协议，一般 NMEA 字符串（ASCII 字符）格式如下：

$XXXXX,①,②,③,……*$h_1h_2$<CR><LF>

其中：

- $：串头
- XXXXX：串名，一般为英文大写字母，表达一定的意义
- ①：数据字段，英文字母或数字，按约定方式表达
- ②：数据字段，英文字母或数字，按约定方式表达
- ③：数据字段，英文字母或数字，按约定方式表达
- ……

- ，：逗号，分隔符
- *：星号，串尾
- <CR>：回车控制符
- <LF>：换行控制符
- h_1h_2：$与*之间所有字符代码的校验和（Checksum），要注意的是，校验和 h_1h_2 为半 Byte 校验，h_1 表示高 4 位校验和，h_2 表示低 4 位校验和。得到校验值后，再转换成 ASCII 字符。下面给出 C 语言程序来说明这个校验和是如何得到的。

假设串字符数组为：*str* 中包括 XXXXX，,①,②,③,……（注意不含$ 和*）。设其长度为 strlen，则可以通过以下程序段得到校验和：

```
int k;
char cc, h1, h2;
     cc = str[0];
      for (k = 1; k < strlen; k++ )
          cc ^= str[k];          //异或
      h2 = cc & 0x0f;
h1 = (cc >> 4) & 0x0f;
      if (h1 < 10)            //将h1转换为ASCII 字符
          h1 += '0';
      else
          h1 += 'A' - 10;
      if (h2 < 10)            //将h2转换为ASCII 字符
          h2 += '0';
      else
          h2 += 'A' - 10;
```

这种半 Byte 校验和主要是为了以 ASCII 字符（可显示的字符）显示出来，形成一个完整的便于观察的数据串。

NMEA-0183 无线通信协议上面已有 GPS 定位数据串作为实例，这里不再举例。

2. 简单自定义通信协议

自定义的通信协议可以设置得非常简单，也可以为了查看方便，设置得比较复杂。简单的协议一般用在需要传送的信息比较简单，而且不需要直接观察的情况下。

下面列出一个由 4 个字节组成的通信协议。首先，设想这样一个数据信息传送任务：要把 9 个开关状态、32 种位置状态和一个最大值为 120km/h 的速度值传送出去。

制定一个比较简单的数据通信协议来完成以上数据信息的传送任务。第一字节为首字节，其最高位（第 7 位）为 1，其他三个字节的最高位为 0，这样，接收方只要判断每个字节的最高位就可以知道一个数据包（串）的首字节，第四字节是校验字节，其最高位为 0，其余 7 位是前面三个字节的异或值。下面是具体的通信协议。

a. 第一字节：最高位（第 7 位）为 1，表示首字节，其他字节该位为 0；第 6 位和第 5 位为开关状态，1 表示开，0 代表关；第 0 至 5 位可以组成 32 种状态，表示各种位置。

第 7 位	第 6 位	第 5 位	第 4 位	第 3 位	第 2 位	第 1 位	第 0 位
标志位	开关 1 状态	开关 2 状态	32 种位置状态				
1 为首字节标志	0 代表关 1 代表开	0 代表关 1 代表开	00000，00001，00010，…，11111 代表 32 种位置状态				

b. 第二字节：第 7 位为 0，表示非第一字节。其余各位为开关状态，1 表示开，0 代表关。

第 7 位	第 6 位	第 5 位	第 4 位	第 3 位	第 2 位	第 1 位	第 0 位
标志位	开关 3 状态	开关 4 状态	开关 5 状态	开关 6 状态	开关 7 状态	开关 8 状态	开关 9 状态
0 表示非第一字节	0 代表关 1 代表开	0 代表关 1 代表开	0 代表关 1 代表开	0 代表关 1 代表开	0 代表关 1 代表开	0 代表关 1 代表开	0 代表关 1 代表开

c. 第三字节：第 7 位为 0，表示非第一字节。需要传送一个速度值，速度要求精确到 1km/h，最大速度值为 120km/h，由于第 0～7 位可以表示的最大值为 127，可满足要求。（注意，如果大于 127 的数，可以通过多个字节来传送，或者通过改变精度要求来满足要求。）

第 7 位	第 6 位	第 5 位	第 4 位	第 3 位	第 2 位	第 1 位	第 0 位
标志位	速度值						
0 表示非第一字节	数值表示实际速度：0000000，0000001，…，1111000 表示从 0～120 的值						

d. 第四字节：该字节为指令校验字节，第 7 位为 0，表示非第一字节，后 7 位是前二个个字节非最高位（第 0～6 位）的各位数逐个进行“异或”计算的结果，据以判断指令传送是否有错。

从这个简单协议的例子可以看出，4 个字节传送了大量的信息，而且还加入了校验。在实际编程中，可以根据自己的需要对协议进行扩充。

在实际工作中，还可以看到各种各样的通信协议，但其实质一般是对数据进行“打包”发送，再在接收方对数据进行“拆包”，编制通信协议就是要如何使数据发送和接收方便，并不容易产生歧义。下一节里介绍如何程序中处理数据包，也就是如何在编程中处理通信协议。为了方便大家更好地理解本节的两种协议，下面的例程还是以处理这两个协议来编写实例程序。

7.2　串口通信数据包处理方法编程实例

通信协议在串口通信编程中的实现，就是如何处理数据包，即发送时如何根据通信协议对数据进行“打包”，接收时如何根据协议对接收到的数据信息“拆包”，并取出自己想要的数据信息。

在程序中，对数据包的处理的一般过程是一边接收一边处理，对接收到的每一个字符进行判断。在 VC 程序中，每当串口缓冲区中有一个或一个以上字符时触发串口通信事件（字符个数可以在程序中进行设定），该事件就驱动（调用）串口通信事件处理函数，对接收到的数据进行处理，判断是否为包（串）头，即第一个字符，再判断是不是结束字符，若是结束字符，则还要对数据进行校验计算，并与接收到的校验值比较，如两者相同，则断定接收到的数据包是正确的，否则是错误的。若是正确的数据包，则将数据包“拆开”，依照一定的规则取出数据信息。

下面的实例程序是来处理 7.1 节的列出的两个协议，并结合 CSerialPort 串口通信类来编写程序。程序要完成的任务如下。

7.2.1　编程任务

在配置有两个串口的同一台计算机上（如果没有两个串口，可对程序稍做修改，在两台计算机上运行程序）编写程序，为方便调试，我们把两个串口的控制都放在同一个程序中，该程序既可在同一台计算上运行，也可以两台计算机上单独运行。程序要求做到单击“发送 NMEA 数据包”按钮，就从串口 1 发送一个 NEMA 通信协议格式的数据包，该数据包发送的内容对应 7.1 节的“简单自定义通信协议”。

NMEA 数据包通信协议如下：

$①②③④⑤⑥⑦⑧⑨⑩⑪⑫⑬⑭*hh<CR><LF>

其中：

- $：启头，首字符。
- ①②③④⑤⑥⑦⑧⑨：控制“简单自定义通信协议”的 9 个开关状态，如①为 1，则表示开关 1 状态为开，为 0，则表示开关 9 状态为关；同理，9 为 1 则表示开关 9 状态为开，为 0 则表示关。
- ⑩⑪：两位数的数据字段，值的范围为 0～31，表示 32 种位置状态，如位置 28 时，⑩为 2，⑪为 8。
- ⑫⑬⑭：三位数的数据字段，值的范围为 0～120，表示速度值。
- *：星号，串尾。
- <CR>：回车控制符。
- <LF>：换行控制符。

为了更明白地说明以上协议，假定要控制开关 2 和开关 5 为打开状态，其他开关状态为关，位置状态为 20，要求的速度值为 110 km/h，则相应的数据包为：

```
$01001000020110* hh<CR><LF>
```

串口 2 收到该数据包后，对相应数据进行处理，将数据显示在程序界面上，为了直观，直接用信号灯表示，位置状态和速度值分别表示，按第 7.1 节的“简单自定义通信协议”将数据打包，再从串口 2 发送给串口 1，串口 1 收到数据后，显示数据。

看了这么一大串的编程任务，是不是觉得很复杂？别怕，因为实际工作编程中，往往也这么复杂，认真编写了这个程序，下回再有这样的数据通信编程任务（包括非串口如 TCP/IP 网络通信程序），你就“手到擒来”了。

7.2.2　编程步骤

本节的程序是在 2.2 节的实例程序基础上进行改进，如果读者还没有编写过该程序，那就必须回过头去先把程序编写出来，也可以直接在 2.2 节的例程基础上直接进行改写。本程序中，还是沿用原来的程序项目工程名 SerialPortTest。按以下步骤进行改写。

（1）建立程序并在主对话框添加控件

我们约定当串口 2 接收到控制数据后，用信号灯来表示灯的开关状态。首先找到代表开关状态的 2 个信号灯图标文件，并将其复制到项目文件里的资源文件夹\Res 中，如图 7.2.1 所示。

OFF	1 KB	图标
RED	1 KB	图标

图 7.2.1　将 RED（代表开）和 OFF（代表关）两个图标文件复制到\Res 文件夹中

然后，在 VC 集成环境中将图标导入程序工程项目中，方法为在 ResourceView 窗口中右击“ICON”图标项，并在弹出的菜单中单击“Import”，然后在\Res 文件夹中选择 RED.ico 和 OFF.ico 两个图标文件，单击“import”命令，如图 7.2.2 所示。并在属性对话框中将其 ID 设置为 IDI_ICON_RED 和 IDI_ICON_OFF。

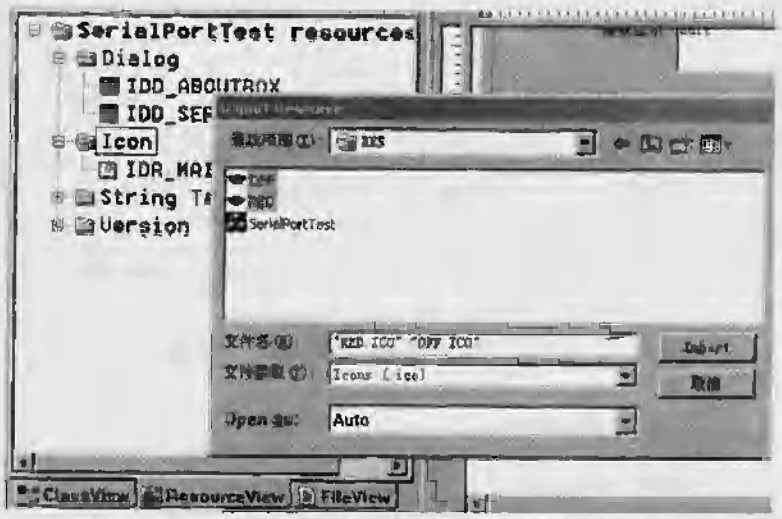


图 7.2.2　导入 RED 和 OFF 两个图标

再将程序主对话框设置成如图 7.2.3 所示的形式，其中，新添加了 9 个复选框（Check Box）控件用于手工设置灯的开关状态，并加入位置和速度的设置编辑框，串口 1 在发送前先读入这些设置值，再按 NEMA 协议将数据打包发送出去。串口 2 接收后，拆包取出数据，再按收到的数据值设置灯的状态，并将位置值和速度值显示在相应对话框中，因此，需要添加 9 个 Picture 控件，在属性中将类型项（Type）设置为 ICON。

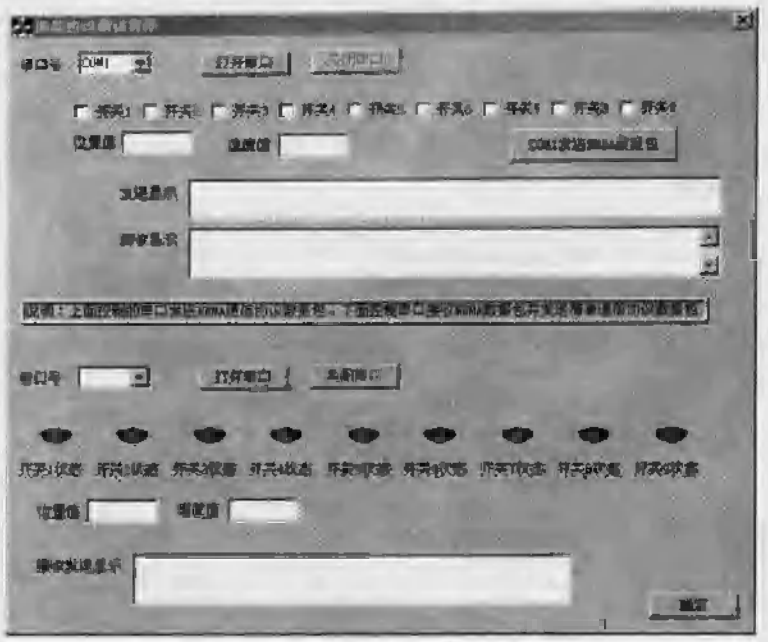


图 7.2.3　为主对话框添加相应控件

以上控件相应的变量及属性如表 7-2-1 所列。按表中所列添加完变量后，试着运行程序，应该没有出错现象。

表 7-2-1 控件及其属性设置情况

控件	控件 ID	Caption	需要添加的变量及变量类型
静态文本	IDC_STATIC	接收发送	
静态文本	IDC_STATIC	发送显示	
静态文本	IDC_STATIC	串口号	
静态文本	IDC_STATIC	位置值	
静态文本	IDC_STATIC	速度值	
组合框	IDC_COMBO_COMPORT		m_ctrlComboComPort Control
编辑框	IDC_EDIT_RECEIVEMSG		m_strEditReceiveMsg CString
编辑框	IDC_EDIT_SENDMSG		m_strEditSendMsg CString
编辑框	IDC_EDIT_POSITION		m_unEditPosition UINT（0-31）
编辑框	IDC_EDIT_VELOCITY		m_unEditVelocity UINT（0-120）
按钮	IDC_BUTTON_OPEN	打开串口	
按钮	IDC_BUTTON_CLOSE	关闭串口	
按钮	IDC_BUTTON_SEND	发送 NMEA 数据包	
复选框	IDC_CHECK_SWITCH1-9	开关 1-9	m_ctrlCheckSwitch1-9 Control
静态文本	IDC_STATIC	串口号	
按钮	IDC_BUTTON_OPEN	打开串口	
按钮	IDC_BUTTON_CLOSE	关闭串口	
按钮	IDC_BUTTON_SEND	发送	
组合框	IDC_COMBO_COMPORT2		m_ctrlComboComPort2 Control
编辑框	IDC_EDIT_POSITION		m_unEditPosition2 UINT
编辑框	IDC_EDIT_VELOCITY		m_unEditVelocity2 UINT
编辑框	IDC_EDIT_DISPDATA2		m_strEditDispData2 CString
图形控件	IDC_STATIC_ICONS1-9	开关 1-9 状态	m_ctrlStaticIconS1-9 Control

（2）编写复选框和图形控件的函数

由于在程序中需要用复选框来设置开关状态，还需要用到信号灯来指示开关状态，因为有可能要多次用到这个功能，所以把它写成函数。另外还要查看复选框的设置状态。

添加 CSerialPortTestDlg 类的两个成员函数（方法如图 7.2.4 所示，似乎我写得太详细了，但好多初学 VC 的朋友还嫌不够啰嗦呢）。

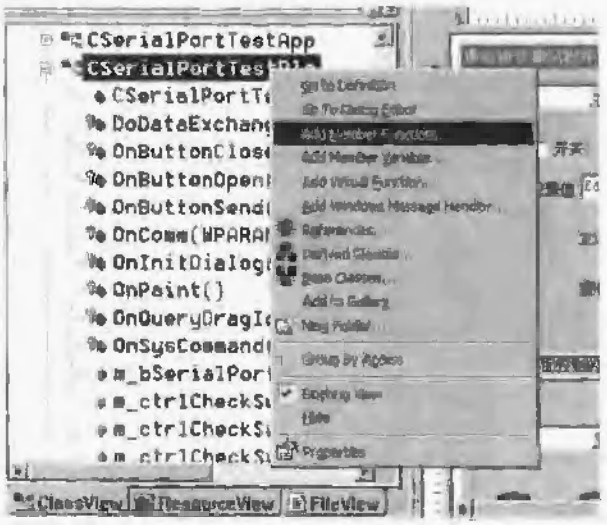


图 7.2.4　添加 CSerialPortTestDlg 类的两个成员函数

```
//该函数用于设置开关的信号灯状态，如开关 5 开启，
//可设置为 SetSwitchStatus(5,TRUE);
void CSerialPortTestDlg::SetSwitchStatus(UINT unSwtich,BOOL bStatus)
{
    if(bStatus)   //设置开关打开状态
    {
        switch(unSwtich) {
        case 1:
            m_ctrlStaticIconS1.SetIcon(m_hIconRed);
            break;
        case 2:
            m_ctrlStaticIconS2.SetIcon(m_hIconRed);
            break;
        case 3:
            m_ctrlStaticIconS3.SetIcon(m_hIconRed);
            break;
        case 4:
            m_ctrlStaticIconS4.SetIcon(m_hIconRed);
            break;
        case 5:
            m_ctrlStaticIconS5.SetIcon(m_hIconRed);
            break;
        case 6:
            m_ctrlStaticIconS6.SetIcon(m_hIconRed);
            break;
        case 7:
            m_ctrlStaticIconS7.SetIcon(m_hIconRed);
            break;
```

```
            case 8:
                m_ctrlStaticIconS8.SetIcon(m_hIconRed);
                break;
            case 9:
                m_ctrlStaticIconS9.SetIcon(m_hIconRed);
                break;
            default:
                break;
            }
        }
        else    //设置开关关闭状态
        {
            switch(unSwtich) {
            case 1:
                m_ctrlStaticIconS1.SetIcon(m_hIconOff);
                break;
            case 2:
                m_ctrlStaticIconS2.SetIcon(m_hIconOff);
                break;
            case 3:
                m_ctrlStaticIconS3.SetIcon(m_hIconOff);
                break;
            case 4:
                m_ctrlStaticIconS4.SetIcon(m_hIconOff);
                break;
            case 5:
                m_ctrlStaticIconS5.SetIcon(m_hIconOff);
                break;
            case 6:
                m_ctrlStaticIconS6.SetIcon(m_hIconOff);
                break;
            case 7:
                m_ctrlStaticIconS7.SetIcon(m_hIconOff);
                break;
            case 8:
                m_ctrlStaticIconS8.SetIcon(m_hIconOff);
                break;
            case 9:
                m_ctrlStaticIconS9.SetIcon(m_hIconOff);
                break;
            default:
                break;
            }
        }
    }
```

下面的函数用于查看复选框设置状态：

```
//该函数用于得到设置的开关状态值
BOOL CSerialPortTestDlg::GetSwitchStatus(UINT unSwitch)
{
    BOOL bStatus=FALSE;
    switch(unSwitch) {
    case 1:
        bStatus = m_ctrlCheckSwitch1.GetCheck();
        break;
    case 2:
        bStatus = m_ctrlCheckSwitch2.GetCheck();
        break;
    case 3:
```

```
            bStatus = m_ctrlCheckSwitch3.GetCheck();
            break;
        case 4:
            bStatus = m_ctrlCheckSwitch4.GetCheck();
            break;
        case 5:
            bStatus = m_ctrlCheckSwitch5.GetCheck();
            break;
        case 6:
            bStatus = m_ctrlCheckSwitch6.GetCheck();
            break;
        case 7:
            bStatus = m_ctrlCheckSwitch7.GetCheck();
            break;
        case 8:
            bStatus = m_ctrlCheckSwitch8.GetCheck();
            break;
        case 9:
            bStatus = m_ctrlCheckSwitch9.GetCheck();
            break;
        default:
            break;
        }
        return bStatus;
    }
```

（3）处理串口控制

在程序中加入了一个串口，需要对串口控制进行处理。首先添加 CSerialPortTestDlg 类公有成员变量 m_bSerialPortOpened2，用于标志串口 2 是否打开，并在类构造函数中将其初始值置为 FALSE；再在类中加入 CSerialPort 类对象 m_SerialPort2，用于新加入串口的控制句柄。由于程序中进行了多串口，且串口号有可能不固定，所以再添加 2 个变量用于记录打开的串口号。

```
CSerialPort m_SerialPort;          //CSerailPort 类对象，串口 1
CSerialPort m_SerialPort2;         //CSerailPort 类对象，串口 2
BOOL m_bSerialPortOpened;          //标志串口 1 是否打开
BOOL m_bSerialPortOpened2;         //标志串口 2 是否打开
UINT m_unPort;                     //记录串口 1 的串口号
UINT m_unPort2;                    //记录串口 2 的串口号
```

组合框 IDC_COMBO_COMPORT2 有个初始值，可以在 CSerialPortTestDlg:: OnInitDialog() 加入一行代码：

```
m_ctrlComboComPort2.SetCurSel(1); //初始选择串口 2
```

为按钮“打开串口 2”添加单击响应函数 CSerialPortTestDlg::OnButtonOpen2()，方法是双击按钮控件或在类向导 CLASSWIZARD 中添加。代码如下：

```
void CSerialPortTestDlg::OnButtonOpen2()
{
    // TODO: Add your control notification handler code here
    int nPort=m_ctrlComboComPort2.GetCurSel()+1; //得到串口号
    if(m_SerialPort2.InitPort(this, nPort, 9600,'N',8,1,EV_RXFLAG | EV_RXCHAR,512))
    {
        m_SerialPort2.StartMonitoring();
        m_bSerialPortOpened2=TRUE;
```

```
m_unPort2=nPort;   //记录打开的串口号
    }
    else
    {
        AfxMessageBox("没有发现此串口或被占用");
        m_bSerialPortOpened2=FALSE;
    }
    GetDlgItem(IDC_BUTTON_OPEN2)->EnableWindow(!m_bSerialPortOpened2);
    GetDlgItem(IDC_BUTTON_CLOSE2)->EnableWindow(m_bSerialPortOpened2);
}
```

注意，OnButtonOpen2()函数同样要记录打开的串口号，可以在相同位置加入一行程序：

```
m_unPort=nPort;,
```

再用同样的方法为按钮控件“关闭串口 2”加入单击响应函数，代码如下：

```
void CSerialPortTestDlg::OnButtonClose2()
{
    // TODO: Add your control notification handler code here
    m_SerialPort2.ClosePort();//关闭串口2
  m_bSerialPortOpened2=FALSE;
    GetDlgItem(IDC_BUTTON_OPEN2)->EnableWindow(!m_bSerialPortOpened2);
    GetDlgItem(IDC_BUTTON_CLOSE2)->EnableWindow(m_bSerialPortOpened2);
}
```

（4）NMEA 通信协议发送处理

下面是进行协议处理的关键代码了。首先添加一个通用的将字符串按 NMEA-0183 无线通信协议格式打包的函数（这个函数今后可以直接用在实际程序中）。

```
//用于将字符串打包成NMEA通信协议包
void CSerialPortTestDlg::SendNMEAData(CString &strData)
{
    char checksum=0,cr=13,ln=10;
  char c1,c2; //2个半Byte校验值
    for(int i=0;i<strData.GetLength();i++)
        checksum = checksum^strData[i];
  c2=checksum & 0x0F;  c1=((checksum >> 4) & 0x0F);
  if (c1 < 10) c1+= '0';   else c1 += 'A' - 10;
  if (c2 < 10) c2+= '0';   else c2 += 'A' - 10;
    CString strNMEAData;
    //加上包头，尾和校验值和回车换行符
    strNMEAData='$'+ strData +'*'+c1+c2+cr+ln;
    //以下几行程序关不通用，在自己的程序中注意修改
    m_SerialPort.WriteToPort((LPCTSTR)strNMEAData);
    m_strEditSendMsg.Format("发送的数据包为: %s",strNMEAData);
    UpdateData(FALSE);  //在发送显示编辑框中显示发送的数据包
}
```

再改写按钮控件“发送 NMEA 数据包”的 OnButtonSend()单击响应函数，代码如下：

```
void CSerialPortTestDlg::OnButtonSend()
{
    // TODO: Add your control notification handler code here
    if(!m_bSerialPortOpened) //检查串口是否打开
    {
        AfxMessageBox("串口没有打开");
        return;
```

```
        UpdateData(TRUE); //读入编辑框中的数据
        CString strSend="$"; //要发送的NEMA字符串
        //以下读入9个开关的设置状态
        for(int i=1;i<=9;i++)
        {
            if(GetSwitchStatus(i))
                strSend+='1';
            else
                strSend+='0';
        }
        CString strTemp;
        strTemp.Format("%02d",m_unEditPosition);
        strSend+=strTemp;
        strTemp.Format("%03d",m_unEditVelocity);
        strSend+=strTemp;
        SendNMEAData(strSend);
    }
```

到这里，连接好串口线（连线时别忘了关机），编译运行程序，把串口调试助手设置在COM2（或在另一台计算机运行），在本例程界面上选择几个开关处于打开状态，写上设置的位置和速度值，打开串口后，单击“发送NMEA数据包”按钮。就可以看到发送的数据包了（如图7.2.5所示）。这说明前面的编程是没有问题的，下面接着处理其他任务。

图7.2.5　利用串口调试助手测试程序

（5）对CSerialPort类做一点改进

在发送字符中，将字符定义为无符号字符，而在CSerialPort类中，没有发送无符号字符的函数。在类中添加一个函数（对这个类，在实际应用中，可以做许多改进）：

```
void CSerialPort::WriteToPort(unsigned char* string,int n)
{
    assert(m_hComm != 0);
    memset(m_szWriteBuffer, 0, sizeof(m_szWriteBuffer));
    memcpy(m_szWriteBuffer, string, n);
    m_nWriteSize=n;
    // set event for write
    SetEvent(m_hWriteEvent);
}
```

（6）串口事件处理函数中的接收与发送处理

在串口事件处理函数中，还要处理的事项如下。

- 串口 2 接收 NMEA 数据包的处理：判断包头、包尾并进行校验，如果校验正确，取出数据；根据收到的设置值对 9 个开关的信号灯进行设置，显示位置值和速度值；将设置值按 7.1 节的“简单自定义通信协议”打包并发送出去。
- 串口 1 要接收串口 2 发送来的“简单自定义通信协议”数据包，并判断数据是否正确，同时将返回的数据显示在编辑框中。

注意：由于用到多串口控制，需要在收到数据时确定是哪个串口来的数据。

下面是串口事件处理函数代码：

```
#define CR 0X0D   //回车
#define LF 0X0A   //换行
LONG CSerialPortTestDlg::OnComm(WPARAM ch, LPARAM port)
{
    static char checksum=0,checksum1=0;
    static int count1=0;//,count2=0,count3=0;
    static unsigned char buf[20];
   static char c1,c2;   //用于计算半Byte校验
    static int flag;    //用于接收阶段标记
    static int twoflag=0;

    if(port == m_unPort)
    {
        if(ch>127)  //是不是首字节
        {
            count1=0; //记录接收字符的个数
            buf[count1]=ch;
            checksum1= ch-128; //开始计算校验值
        }
        else
        {
            count1++;
            buf[count1]=ch;
            checksum1 = checksum1^ch;
            if(count1==3)  //包括校验字节在内的全部接收完毕
            {
                if(checksum1)  //校验错
                {
                    m_strEditReceiveMsg = "接收校验出错";
                    UpdateData(FALSE);
                }
                else
                {
                    CString str;
                    unsigned char * temp=(unsigned char*)buf;
                    m_strEditReceiveMsg ="接收到的简单通信协议字节为: ";
                    for(int i=0;i<4;i++)
                    {
                        str.Format("%02X ",*(buf+i));
                        m_strEditReceiveMsg += str;
                    }
                    UpdateData(FALSE);
                }
                if(count1>5)   //防止出错
                    count1=0;
```

```
            }
        }
    }

    if(port==m_unPort2)   //串口 2 的数据处理
    {
        m_strPortRXData2 += (char)ch;
        switch(ch)
        {
        case '$':
            checksum=0;                //开始计算 CheckSum
            flag=0;
            break;
        case '*':  //有效数据结束，可以$和*之间数据的半 Byte 校验值了
            flag=2;
            c2=checksum & 0x0f;      c1=((checksum >> 4) & 0x0f);
            if (c1 < 10) c1+= '0';  else c1 += 'A' - 10;
            if (c2 < 10) c2+= '0';  else c2 += 'A' - 10;
            break;
        case CR:   //这句必须加上，否则会出错的
            break;
        case LF:   //数据包的最后一个字符
            m_strPortRXData2.Empty();
            break;
        default:
            if(flag>0)  //注意：只有在接收到'*'后，flag 才大于 0
            {
                m_strChecksum2 += ch;
                if(flag==1)
                {
                    CString strCheck="";
                    strCheck.Format("%c%c",c1,c2);
                    if(strCheck!=m_strChecksum2)
                    {          //校验计算不正确，说明接收数据出错
                        m_strPortRXData2.Empty();
                    }
                    else  //校验计算正确则处理数据
                    {
                        CString strSwitchSetData;
                        strSwitchSetData = m_strPortRXData2.Mid(1,9);
                        //以下设置信号灯状态
                        for(int i=0;i<9;i++)
                        {
                            if(strSwitchSetData.Mid(i,1)=="1")
                                SetSwitchStatus(i+1,TRUE);
                            else
                                SetSwitchStatus(i+1,FALSE);
                        }
                        //以下取出位置与速度数据
                        CString strTemp;
                        strTemp = m_strPortRXData2.Mid(10,5);
                        char *temp=(char*)(LPCTSTR)strTemp);
                        char tbuf[4];
                        tbuf[0]=temp[0]; tbuf[1]=temp[1]; tbuf[2]=0;
                        m_unEditPosition2 = atoi(tbuf);  //得到位置状态值
                        tbuf[0]=temp[2]; tbuf[1]=temp[3];
                        tbuf[2]=temp[4]; tbuf[3]=0;
                        m_unEditVelocity2 = atoi(tbuf);  //得到速度值
                        //将接收到的数据包内容显示
                        m_strEditDispData2="接收到NMEA 数据包："+m_strPortRXData2;
```

```
                    //下面准备发送"简单自定义通信协议"数据包
                    unsigned char ucChar[4];
                    //首字节
                    ucChar[0]=0x80;   //首字节最高位置 1
                    if(strSwitchSetData.Mid(0,1)=="1")  // 开关 1 状态
                         ucChar[0] |= 0x40;          //0100 0000
                    else
                         ucChar[0] &= 0xBF;          //1011 1111
                    if(strSwitchSetData.Mid(1,1)=="1")  // 开关 2 状态
                         ucChar[0] |= 0x20;          //0010 0000
                    else
                         ucChar[0] &= 0xDF;          //1101 1111
                    if(m_unEditPosition2>31)      //对位置值进行限值
                         m_unEditPosition2=31;
                    ucChar[0] &= 0xE0;    //将首字节的低 5 位置 0
                    ucChar[0] += m_unEditPosition2; //再加上位置值
                    ucChar[3] = ucChar[0];    //同时计算校验值
                    //第二字节
                    unsigned char ucTemp=0x40;
                    for(i=0; i<7; i++)
                    {
                         if(strSwitchSetData.Mid(2+i,1)=="1")
                              ucChar[1] |= ucTemp;
                         else
                              ucChar[1] &= (~ucTemp);
                         ucTemp >>= 1;
                    }
                    //前面做了那么多运算，用这条语句保证一下最高位为 0
                    ucChar[1] &= 0x7F;
                    ucChar[3] ^= ucChar[1];   //计算校验值
                    //第三字节
                    if(m_unEditVelocity2>120)
                         m_unEditVelocity2=120;
                    ucChar[2] = m_unEditVelocity2;
                    ucChar[3] ^= ucChar[2];   //计算校验值
                    //第四字节
                    ucChar[3] &= 0x7F;
                    //把"简单自定义通信协议"数据包发送出去
                    if(m_bSerialPortOpened2)
                         m_SerialPort2.WriteToPort(ucChar,4);
                    //同时把发送的内容显示
                    CString strTemp1;
                    strTemp=  _T("发送的内容为: ");
                    for(int j=0;j<4;j++)
                    {
                         strTemp1.Format("0x%02X",ucChar[j]);
                         strTemp += strTemp1 + ",";
                    }
                    m_strEditDispData2 += strTemp;

                    UpdateData(FALSE);
                }
                m_strChecksum2.Empty();
            }
//从'*'后收，flag-2,依次减 1 操作，正好将数据包的校验值保存在 m_strChecksum 中
            flag--;
        }
        else
            checksum=checksum^ch;  //当 flag<=0 时，计算校验值
        break;
```

```
            }
        }
        return 0;
    }
```

（7）其他变量的设置说明

程序编写完了，添加的变量在 CSerialPortTestDlg 类头文件中，在类的构造文件中变量初始化，并在类函数 OnInitDialog()中做了初始化处理。

CSerialPortTestDlg 类头文件代码如下：

```
// SerialPortTestDlg.h : header file
#include "SerialPort.h"  //添加CSerailPort 类的头文件
class CSerialPortTestDlg : public CDialog
{
// Construction
public:
    void SendNMEAData(CString &strData);
    BOOL GetSwitchStatus(UINT unSwitch);
    void SetSwitchStatus(UINT unSwtich,BOOL bStatus);
    HICON m_hIconRed;    //开关打开时的红灯图标句柄
    HICON m_hIconOff;    //开关关闭时的指示图标句柄

    CSerialPort m_SerialPort; //CSerailPort 类对象，串口1
    CSerialPort m_SerialPort2; //CSerailPort 类对象，串口2
    BOOL m_bSerialPortOpened; //标志串口1是否打开
    BOOL m_bSerialPortOpened2; //标志串口2是否打开
    UINT m_unPort;        //记录串口1的串口号
    UINT m_unPort2;       //记录串口2的串口号
    CString m_strPortRXData;  //用于储存串口1收到的数据
    CString m_strPortRXData2;  //用于储存串口2收到的数据
    CString m_strChecksum2;  //用于串口2接收数据计算校验值
    CSerialPortTestDlg(CWnd* pParent = NULL); // standard constructor
// Dialog Data
    //{{AFX_DATA(CSerialPortTestDlg)
    enum { IDD = IDD_SERIALPORTTEST_DIALOG };
    CStatic  m_ctrlStaticIconS1;
    CStatic  m_ctrlStaticIconS2;
    CStatic  m_ctrlStaticIconS3;
    CStatic  m_ctrlStaticIconS4;
    CStatic  m_ctrlStaticIconS5;
    CStatic  m_ctrlStaticIconS6;
    CStatic  m_ctrlStaticIconS7;
    CStatic  m_ctrlStaticIconS8;
    CStatic  m_ctrlStaticIconS9;
    CButton  m_ctrlCheckSwitch1;
    CButton  m_ctrlCheckSwitch2;
    CButton  m_ctrlCheckSwitch3;
    CButton  m_ctrlCheckSwitch4;
    CButton  m_ctrlCheckSwitch5;
    CButton  m_ctrlCheckSwitch6;
    CButton  m_ctrlCheckSwitch7;
    CButton  m_ctrlCheckSwitch8;
    CButton  m_ctrlCheckSwitch9;
    CComboBox m_ctrlComboComPort2;
    CComboBox m_ctrlComboComPort;
    CString  m_strEditReceiveMsg;
    CString  m_strEditSendMsg;
    UINT m_unEditPosition;
```

```
        UINT m_unEditVelocity;
        UINT m_unEditVelocity2;
        UINT m_unEditPosition2;
        CString  m_strEditDispData2;
        //}}AFX_DATA

        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CSerialPortTestDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
        //}}AFX_VIRTUAL

    // Implementation
    protected:
        HICON m_hIcon;
        // Generated message map functions
        //{{AFX_MSG(CSerialPortTestDlg)
        virtual BOOL OnInitDialog();
        afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
        afx_msg void OnPaint();
        afx_msg HCURSOR OnQueryDragIcon();
        afx_msg LONG OnComm(WPARAM ch, LPARAM port);
        afx_msg void OnButtonOpen();
        afx_msg void OnButtonClose();
        afx_msg void OnButtonSend();
        afx_msg void OnButtonOpen2();
        afx_msg void OnButtonClose2();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
    };
    …
    #endif
// !defined(AFX_SERIALPORTTESTDLG_H__556B3D06_5375_11D8_870F_00E04C3F78CA__INCLUDED_)
```

在类构造函数中，对变量进行了初始化。

```
    CSerialPortTestDlg::CSerialPortTestDlg(CWnd* pParent /*=NULL*/)
        : CDialog(CSerialPortTestDlg::IDD, pParent)
    {
        //{{AFX_DATA_INIT(CSerialPortTestDlg)
        m_strEditReceiveMsg = _T("");
        m_strEditSendMsg = _T("");
        m_unEditPosition = 0;
        m_unEditVelocity = 0;
        m_unEditVelocity2 = 0;
        m_unEditPosition2 = 0;
        m_strEditDispData2 = _T("");
        //}}AFX_DATA_INIT
        // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
        m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
        m_bSerialPortOpened=FALSE; //初始状态：串口 1 没有打开
        m_bSerialPortOpened2=FALSE; //初始状态：串口 2 没有打开
        m_unPort=1;
        m_unPort2=2;
        m_strPortRXData2="";
        m_strChecksum2="";
    }
```

在类函数 OnInitDialog()中做的初始化处理工作：

```
BOOL CSerialPortTestDlg::OnInitDialog()
{
    ...
    // TODO: Add extra initialization here
    m_ctrlComboComPort.SetCurSel(0); //初始选择串口1
    m_ctrlComboComPort2.SetCurSel(1); //初始选择串口2

    //以下两句分别设置"打开串口"、"关闭串口"两个按钮的使能状态
    GetDlgItem(IDC_BUTTON_OPEN)->EnableWindow(!m_bSerialPortOpened);
    GetDlgItem(IDC_BUTTON_CLOSE)->EnableWindow(m_bSerialPortOpened);
    GetDlgItem(IDC_BUTTON_OPEN2)->EnableWindow(!m_bSerialPortOpened2);
    GetDlgItem(IDC_BUTTON_CLOSE2)->EnableWindow(m_bSerialPortOpened2);

    //载入图标
    m_hIconRed  = AfxGetApp()->LoadIcon(IDI_ICON_RED);
    m_hIconOff    = AfxGetApp()->LoadIcon(IDI_ICON_OFF);

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

到这里，程序全部编写完了。我们可以开始测试程序了。

7.2.3　程序测试

如果计算机上有两个串口，就可以在同一台计算机上测试，否则，就在两台计算机上分别打开程序。首先，连接好串口线（连线时不要忘记关机），然后运行程序。程序运行情况如图 7.2.6 所示。图示情况为串口 1 选在 COM1，串口 2 先在 COM2，分别打开串口，然后将开关 1、开关 3 和开关 5 的复选框选上，位置值设为 22，速度值设置为 68，单击"发送 NMEA 数据包"按钮后，下面的相应开关状态指示信号灯变红，而且，位置和速度值也与设置值一致，在各个编辑框中也显示了接收和发送信息。这说明编程是成功的。

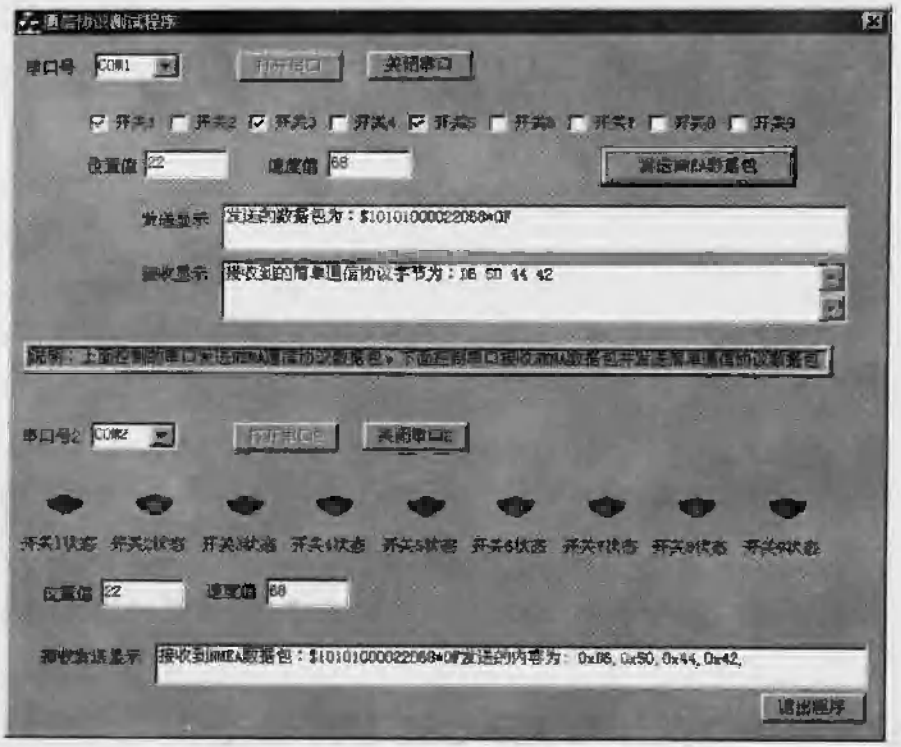


图 7.2.6　测试通信协议程序

但是，该程序还有一点没有处理：串口 1 只是显示了接收到的“简单通信协议”数据信息，并没有将数据提取出来，读者可以试着完成数据的提取。

第8章　单片机串口通信

[内容提要]

单片机与 PC 机（或者单片机之间）的串口通信，在实际工作中，也是经常遇到的编程任务。本章假定读者熟悉单片机编程（包括程序烧写）的前提下，以实例程序的演示方式阐述了单片机与 PC 机的编程工作。单片机编程目标为通用的 8051 系列微处理器，并采用 C51 语言编程，程序在 Keil C51 V7.0 单片机设计软件中仿真并编译后，烧写到单片机中与 PC 机进行通信。

单片机与 PC 机通信时，PC 机串口程序编写方式与本书其他章节的串口程序是一样的，只是在收发数据时要根据通信协议对数据进行打包或拆包。所以只简单讲述单片机的程序实例。

本章 8.1 节对单片机中与串口相关的硬件构成做了简要描述，并简单介绍了 C51 语言程序，以及如何利用 Keil C51 编译器对 C51 串口程序做编译和仿真通信；8.2 节是 C51 的单片机串口实例程序，对程序的编写方式做了详细说明。

8.1　单片机串口硬件系统及 C51 程序开发

单片机以其体积小、价格低、抗干扰性好等特点，在现代控制系统中常用在操作现场进行数据采集，以及实现现场控制。但是，由于其数据存储容量和数据处理能力都较低，所以一般情况下要通过通信手段使它与 PC 机相连，把所采集到的数据传送到 PC 机上，再在 PC 机上进行数据处理，充分发挥两者各自的优势。PC 微机—单片机系统是一种广泛应用的主从式计算机控制系统，其信号的交互往往采用串口或网络通信。本节以 MCS-51 单片机和 PC 机为例，从单片机硬件系统、底层驱动软件层应用软件设计等方面来说明这种系统的实现方法。

8.1.1　较典型的单片机硬件系统实例

为了便于初学的读者（可能会没有接触过单片机实际工作）了解单片机硬件系统，这里介绍一个典型的低速信号数据采集系统，其结构框图如图 8.1.1 所示。在这个系统中，按键模块的作用主要是设定系统的工作模式，其设定结果由显示模块显示出来。从图中可以很清楚地看出数据的流向。要采集的低速信号，经前端放大处理，由 A/D 采样后送入单片机，单片机把采集到的数据整理成一帧（数据包），最后通过 RS-232 串行口发送到 PC 机。

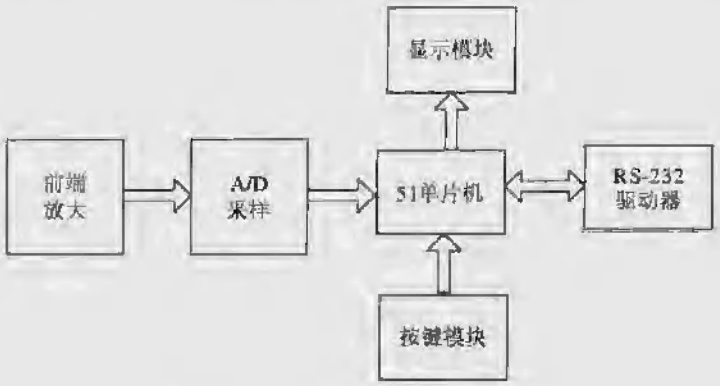


图 8.1.1　单片机硬件系统结构框图

由于单片机输入、输出电平是 TTL 电平，而 PC 机配置的是 RS-232 标准串行接口，二者的电气规范不一致，因此，要完成单片机与 P C 机的数据通信，必须对单片机输出的 TTL 电平进行电平转换。以前常用的 TTL 与 RS-232 电平转换芯片为 MC1488 和 MC1489。MC1488 将 TTL 电平转换为 RS-232 电平，其供电电压为±12V，MC1489 则把 RS-232 标准电平转换为 TTL，供电电压为+5V，因此电路中除系统的+5V 电源外，另需±12V 电源。这对于不具备±12V 电源的单片机系统很麻烦，因此本电路用一种标准的 RS-232 芯片 MAX232。MAX232 是 MAXIM 公司生产的芯片，使用单一电源电压 *Vcc*，该芯片与单片机的接口电路非常简单，只需外接 5 个 0.1μF 的电容，即可实现 TTL 与 RS-232 两种电平的转换，具体接口电路框图和电路图如图 8.1.2 所示（注意，两者并不一致）。采用 3 线制双工通信连接方式，单片机串行口的 TXD、RXD 和 GND 经电平转换分别与微机的 RXD、TXD 和 SG 相连。

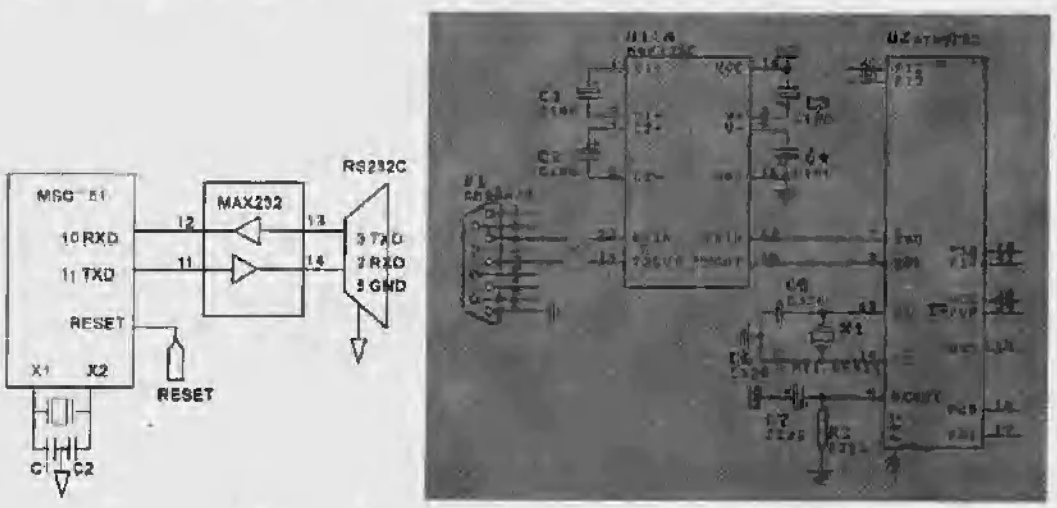


图 8.1.2　PC 机与单片机串口通信系统框图和电路图

8.1.2　C51 语言及程序简介

C51 是一种针对 MCS-51 系列单片机的软件开发工具，是开发 MCS-51 系列单片机的高级语言。C51 编译器为用户利用 C 语言来开发单片机提供了支撑环境。利用传统的汇编语言开发单片机系统虽然具有代码紧凑、实时性好等优点。但缺点也十分突出：程序可读性、可移植性差；不易进行复杂的逻辑数学运算；大程序的软件开发周期长等。现在广为普及并流行的 C 语言则恰好弥补了汇编语言的缺点：一方面，丰富的指令集直接面向硬件操作，位指令、逻辑、关系表达式均可直接针对外设端口；另一方面，高级语言的描述，可使用户摆脱与硬件无必要的接触。语言描述可由编译器编译直接生成面向硬件的机器码，用于下装 EPROM 或 E^2PROM。在一般情况下，由 C51 编译生成的代码不论长度还是程序运行速度均能适应程序要求。利用 C51 开发单片机系统，不但可以使编程工作量大为减少，而且使软件维护、修改亦变得非常方便。

1. C51 程序结构设计方法

C 语言具有良好的程序结构，适用于模块化程序设计，C51 继承了 C 语言的思想，因此，在实际 C51 程序设计中,其主程序结构一般均采用如下结构：

```
main()
{
int i,j;      /*  整型变量声明    */
func1();/*  功能函数说明 1*/
func2();/*  功能函数说明 2*/
func3();/*  功能函数说明 3*/
}
```

- 与 C 语言不同，包含文件中多了一组与 MCS-51 系列硬件相关的头文件如：IO51.H、IO52.H 等。在这些预说明文件中，一般是对 MCS-51 系列单片机的内部功能寄存器及端口进行说明，它包括字节型变量和位型变量，以便用户在编程时直接使用。
- 在 main 外部声明的变量为全局变量，一般是将在整个程序都有效的数据和标志存储在全局变量中，局部变量一般用做存储一些中间结果的变量，相当于汇编语言程序设计中使用的一些缓冲单元。
- C51 支持 ANSI 标准 C 的 signedchar/unsignedchar、signedint/unsignedint、float、double 等数据类型，扩充了位寻址的 bit、sfr 简化对 8051 特殊功能寄存器的访问，变量可组合为结构和联合，可定义为多维数组，也可通过指针访问变量。
- 存储模式:如果省略存储类型，编译时设置的存储模式将自动决定变量的默认存储类型。
  - TINY——全局变量和局部变量均放入内部存储区(data)
  - SMALL——参数及局部变量放入可直接寻址的内部存储区(data 及 idata)
  - LARGE——参数及局部变量直接放入外部数据存储区(xdata)
  - BANKED——组模式全局及局部变量放入外部数据存储区(xdata)

利用 C51 开发单片机程序时，应当根据单片机硬件系统和相应的程序大小选用适当的存储模式。例如：硬件为 8031 系统并且程序带有复杂浮点运算和大量输入/输出，程序同时要管理键盘和 LED 显示时，存储模式就可选用 LARGE 模式。

2. 使用 C51 开发单片机系统应用软件

（1）复杂应用程序的分块编译。

当应用程序较大时，由于硬件环境的限制，计算机在编译大程序时会产生“内存不够”的提示，这时可以将大程序分割成几个较小的程序，需要调用全局变量和外部函数时，只要声明其为 extem 即可。

（2）编制.XCL 文件。

一个良好的.XCL 文件能够很好地控制系统硬件。通过修改.XCL 文件，可以确定系统 CPU 类型、外部 RAM 开始地址、掉电保护区域 RAM 地址、堆栈指针起始地址和堆栈区大小、片内 RAM 的起始地址、欲连接的程序块名称、连接完毕后生成的输出文件格式。从生成(编译连接)的.MAP 文件中可以确切知道每一变量及函数在 ROM 及 RAM 中的地址，从而可以达到跟踪调试的目的。

（3）在 C 源程序级上更改默认 I/O 输出，以便适应不同的特定的硬件要求。

例如，printf 函数默认输出为串行口，现为适应硬件要求，需要定时输出至微型打印机，若采用汇编语言，则需将整型数进行转换才能输出，而利用 C51，在修改后可以直接使用 printf 函数，详细情况参见下例子：

```
/*  修改默认 printf 的输出至打印机  */
#include  (io51.h)
#define  Tpup24    ((char*)0x0180000)  /*  定义微型打印机输出端口  */
static  void  low_level_put(char c)
{
do{}
while(P1.6==1);    /*判断打印机是否为忙状态  */
Tpup24=C;/*将字符输出至打印机端口  */
}
```

（4）C51 允许编程者对中断的所有方面的控制和寄存器组的使用，可创建高效的中断服务程序，产生最合适的代码。

简便而强有力的中断函数，使 C51 极其适合于工业中具有复杂逻辑判断、运算的智能控制仪表，比如研制专家预测模糊 PID 控制的智能温控器中，程序要管理键盘，LED 数码管显示，大量的开关量输入/输出，同时程序还需要进行大量的条件转移判断及复杂的浮点数学运算。用汇编语言实现会相当繁杂，且极容易出错，程序很难调试和维护。采用 C51 后，复杂算法则很容易实现，并且程序的可读性、可维护性明显增强。

8.1.3　开发 C51 程序的利器 Keil C51 uVision2 及串口程序仿真

Keil C51 uVision2 支持众多不同公司的 MCS-51 架构的芯片，它集编辑、编译和程序仿真等于一体，同时还支持 PLM、汇编和 C 语言的程序设计。它的界面和本书用的 VC 的界面十分相似，界面友好且易学易用，在调试程序、软件仿真方面也有很强大的功能。因此，很多开发 51 应用的工程师或普通的单片机爱好者都使用 Keil C51 uVision2 开发单片机程序。要说明的是，Keil C51 是一个商业的软件，但免费提供能编译 2KB 程序的演示版，对于初学者这就足够了，要继续看本章，就赶紧去下载一个来安装在自己的计算机上吧，一般的单片机开发网站上都可以找到下载文件。

在下面的讲解中，利用 Keil C51 7.0 中来编译一个简单的程序，通过对程序的项目创建、

器件选择、仿真、程序生成等步骤，来熟悉 Keil C51 的使用。

软件的开发流程如下：

- 创建一个项目，从器件库中选择目标器件，配置工具设置；
- 用 C 语言或汇编语言创建源程序；
- 用项目管理器生成程序；
- 修改源程序中的错误；
- 测试（仿真），连接应用；
- 生成 HEX 文件，烧写到单片机芯片中。

下面按这个流程利，用 Keil C51 7.0 来体会一下运行在单片机芯片中的代码是如何生成的。

（1）启动 uVision2 并创建新项目，选择目标 CPU 器件。

启动 Keil C51 uVision2，单击 Project-> New Project 菜单命令，新建一个 scommtest 项目，如图 8.1.3 所示，在弹出的“Create New Project”对话框中填好 scommtest 后，单击“保存”即可。

图 8.1.3 新建项目 scommtest

再在弹出的“Select Device for Target 1”对话框中选择一种单片机，这里选用 Intel 的 8051AH，如图 8.1.4 所示。

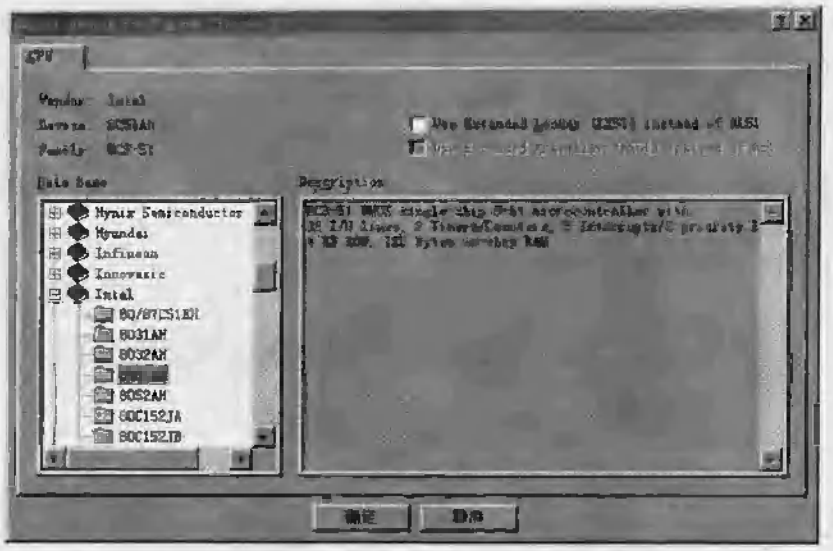


图 8.1.4　选择目标单片机器件

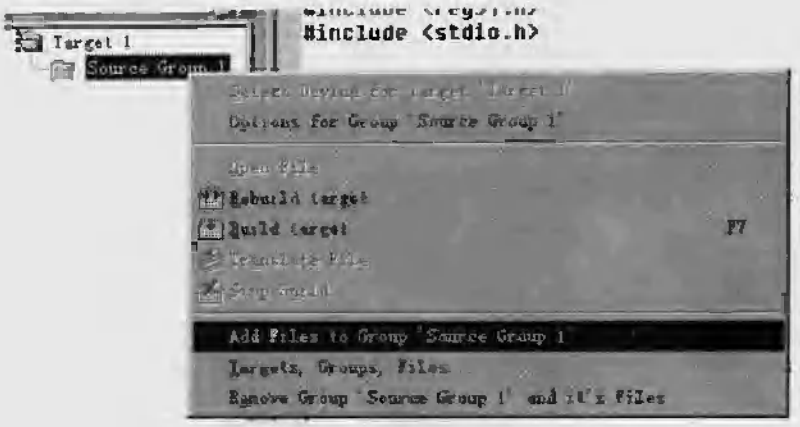
完成以上工作后，就可以开始编写程序了。

（2）用项目管理器生成程序，并加入项目工程。

单击 File->New 命令，建立新的程序，输入下面的简单代码后，把程序保存为 test0c.c。

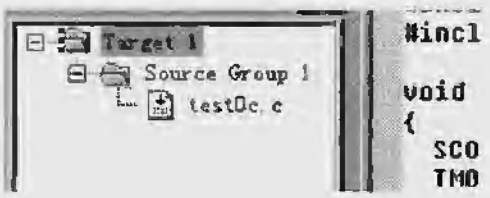
```
#include <reg51.h>
#include <stdio.h>

void main(void)
{
  SCON = 0x50; //串口方式1,允许接收
  TMOD = 0x20; //定时器1定时方式 2
  TCON = 0x40; //设定定时器开始计数
  TH1 = 0xE8;  //11.0592MHz 1200 波特率
  TL1 = 0xE8;
  TI = 1;
  TR1 = 1; //启动定时器

  while(1)
 {
     printf ("Hello World!\n"); //显示Hello World
  }
}
```

现在，可以将程序 test01.c 加入工程 scommtest，方法如图 8.1.5 所示，在 Source Group 1 上单击鼠标右键，在弹出的环境菜单中选择“Add File to Group ‘Source Group 1’”，弹出文件选择窗口，选择刚刚保存的文件 test0c.c，单击 Add 按钮，关闭文件窗，程序文件已加到项目中了。这时在 Source Group1 文件夹图标左边出现了一个小+号说明，文件组中有了文件。



(a)

(b)



(c)

图 8.1.5　在项目工程中添加程序

程序加入项目后，最好先编译一下，看看程序有没有语法错误。编译的方法是单击图 8.1.6 中的快捷按钮 1，系统会提示程序中有没有错误。按钮 2 是编译项目，按钮 3 是重新编译项目（所有的程序）。

图 8.1.6　编译程序

（3）项目程序的仿真调试。

Keil uVision2 仿真调试项目程序时，其输出是串口，并且输出窗口中可以清楚地看到所有的输出结果，所以在调试单片机串口时非常方便，这也是本书中编入这部分内容的重要原因。

单击开始/关闭调试快捷按钮（或者是 Debug 菜单中的相应菜单命令），出现如图 8.1.7（a）所示的“调试命令”和“调试仿真输出窗口”的工具条菜单。单击“RUN”命令，并把串行输出窗口“Serial Window#1”打开，就可以从串口输出信息了，如图 8.1.7（b）所示。

(a)

(b)

图 8.1.7　调试仿真项目程序并从串口输出

（4）生成供烧写项目程序的十六进制文件（HEX 文件）。

HEX 文件格式是 Intel 公司提出的按地址排列的数据信息，数据宽度为字节，所有数据使用十六进制数字表示，常用来保存单片机的目标程序代码，一般的编程器都支持这种格式。Keil uVision2 来编译生成用于烧写芯片的 HEX 文件后，就可以将这个 HEX 文件通过烧写器写入目标处理器。

鼠标右击图 8.1.8 中的项目文件夹，弹出项目功能菜单，选择 Options for Target ‘Target1’，出现项目选项设置窗口（注意：如果这时是在调试状态，请先关闭调试状态），选择 Output 项，选上“Create HEX File”选项，同时，可以更改输出文件的名称和保存文件夹。

(a)

(b)

图 8.1.8　选择输出 HEX 文件

这时再重新编译项目，就会在项目文件夹（或者设定的文件夹）中看到 HEX 文件了，将这个文件烧写到单片机器件中，就可以在单片机中运行程序了。

8.2　C51 单片机串口通信程序实例

在这一节中，介绍两个单片机串口通信程序实例。两个程序都采用中断方式接收数据，只是通信任务有所不同，读者可以通过读这两个程序，加深对单片机串口程序的理解。

8.2.1　实例一

本例上位 PC 机采用 VC 6.0 编制上位机通信程序，单片机采用 51 系列，语言用 C51。通信功能是实现 PC 机键盘输入字符串，发送给单片机，单片机收到 PC 机发来的数据后，回送同一数据给 PC 机，并在 PC 机屏幕上显示出来。如果通信正常，两次字符应该相同。双方约定:波特率为 9600；信息格式为 8 个数据位，一个停止位。传送方式为 PC 机采用查询方式收发数据，51 单片机采用中断方式接收数据。以字符“$”为发送和接收的停止标志。

MCS-51 单片机通过中断接收 PC 机发送过来的字符,并回送给主机。具体程序如下：

```
#include <reg51.h>
#define uchar unsigned char
uchar xdata rt_buf[32];
uchar r_in,t_out;
bit r_full,t_empty;

serial() interrupt4  //串口中断程序
{
```

```
        if(RI && r_full)
        {
            rt_buf[r_in]=SBUF;
            RI=0;
          if(rt_buf[r_in]==0x24)
          {
             r_full=1;
             SBUF=tr_buf[t_out];
             t_empty=0;
          }
          /*接收字符为$，则接收结束；设置接收结束标志，开始发送*/
          r_in=++r_in;
        }
        else if(TI && t_empty)
        {
            TI=0;
            t_out=++t_out;
            SBUF=rt_buf[t_out];
            if(t_out==r_in)
                 t_empty=1;
            /*t_out=r_in 则发送完，设发送完标志 t_empty*/
        }
    }

    main()
    {
        uchar a;
    /*设置定时器 T1 工作于方式 2，计数常数为 0xfdH */
       TMOD=0x20;
        TL1=0xfd;
        TH1=0xfd;
        SCON=0x50;//在 11.0592MHz 下，设置串行口波特率为 9600，方式 1
        PCON=0xD0;
        IE=0x10;
        TR1=1;
        EA=1;
        r_in=t_out=0;
        t_empty=1;
        r_full=0;
        for(;;)
        {
            //相应的任务处理功能函数
        }
    }
```

8.2.2　实例二

通信协议：第 1 字节，MSB 为 1，为第 1 字节标志，第 2 字节，MSB 为 0，为非第一字节标志，其余类推……，最后一个字节为前几个字节后 7 位的异或校验和。

测试方法：可以将串口调试助手的发送框写上 95 10 20 25，并选上 16 进制发送，接收框选上 16 进制显示，如果每发送一次就接收到 95 10 20 25，则说明测试成功。

```
//这是一个单片机 C51 串口接收（中断）和发送例程，可以用来测试 51 单片机的中断接收
//和查询发送，另外我觉得发送没有必要用中断，因为程序的开销是一样的
#include <reg51.h>
#include <string.h>
```

```
#define INBUF_LEN 4  //数据长度

unsigned char inbuf1[INBUF_LEN];
unsigned char checksum,count3;
bit        read_flag=0;

void init_serialcomm(void)
{
   SCON  = 0x50;      //SCON: serail mode 1, 8-bit UART, enable ucvr
   TMOD |= 0x20;      //TMOD: timer 1, mode 2, 8-bit reload
   PCON |= 0x80;      //SMOD=1;
   TH1   = 0xF4;      //Baud:4800  fosc=11.0592MHz
   IE   |= 0x90;      //Enable Serial Interrupt
   TR1   = 1;         // timer 1 run
  // TI=1;
}

//向串口发送一个字符
void send_char_com(unsigned char ch)
{
   SBUF=ch;
   while(TI==0);
   TI=0;
}

//向串口发送一个字符串，strlen为该字符串长度
void send_string_com(unsigned char *str,unsigned int strlen)
{
   unsigned int k=0;
   do
   {
      send_char_com(*(str + k));
      k++;
   } while(k < strlen);
}


//串口接收中断函数
void serial () interrupt 4 using 3
{
   if(RI)
   {
      unsigned char ch;
      RI = 0;
      ch=SBUF;
      if(ch>127)
      {
          count3=0;
          inbuf1[count3]=ch;
          checksum= ch-128;
      }
      else
      {
          count3++;
          inbuf1[count3]=ch;
          checksum ^= ch;
          if( (count3==(INBUF_LEN-1)) && (!checksum) )
          {
            read_flag=1;  //如果串口接收的数据达到INBUF_LEN个，
```

```
                                    //且校验没错，就置位取数标志
                }
            }
        }
    }

    //主程序
    main()
    {
        init_serialcomm();  //初始化串口
        while(1)
        {
            if(read_flag)  //如果取数标志已置位，就将读到的数从串口发出
            {
                read_flag=0; //取数标志清 0
                send_string_com(inbuf1,INBUF_LEN);
            }
```

第 9 章　串口与网络结合的解决方案及编程

[内容提要]

串口通信作为一种简单而传统的通信方式，由于传统设备的使用“惯性”，虽然在相当长的一段时间内不可能被淘汰（这一点，只要去看看那么多高技术公司还在应用着这项技术，还有那么多新出现的设备还保留着串口就可以看出来），但随着 TCP/IP 等网络通信的出现，使用串口通信的传统设备应用场合与网络通信结合的趋势越来越明显，这是保护用户的既往投资和整体利益的一种有效方法。本章讨论几种串口与网络结合的解决方案，并从各种角度进行编程，这些方法可以在实际编程中根据条件和需要选用。串口设备联网的方法有以下几种：

- 使用硬件设备，如 MOXA 公司的串口设备联网服务器，这种方法的优点是不需要对应用程序做出很大修改，开发成本低，周期短。
- 对应用程序进行修改，加入网络通信功能。这种方法适用于拥有源代码资源和人才资源，而又不想付出购置硬件成本的场合，或者两者对比起来，修改程序较为合算。

第 9.1 节是串口与网络结合的硬件解决方案介绍；第 9.2 节介绍了一款较典型的串口网络连接的服务器和应用实例；第 9.3 节介绍了串口数据的数据库存储实例，稍作修改就可应用于远程数据数据存储；第 9.4 节介绍了与 WinSock 结合的串口通信实例，这是 TCP/IP 编程中常用的网络编程方法；第 9.5 说明了如何在已经编好的串口通信程序中加入网络通信功能；第 9.6 节以一个简单实例说明如何用串口或网络通信进行遥控操作。

9.1 串口与网络结合的硬件解决方案

串口通过硬件联网，实现串口设备的网络化，简单的连接方法如图 9.1.1 所示，多/单个串口数据设备（如计算机、单片机、GPS 接收器等）连接在串口联网设备上，再通过串口联网设备接到 TCP/IP（或其他协议）的网络上，只要是连接网络上的控制主机，就可以控制这些串口设备，接收到串口联网设备转发过来的数据。

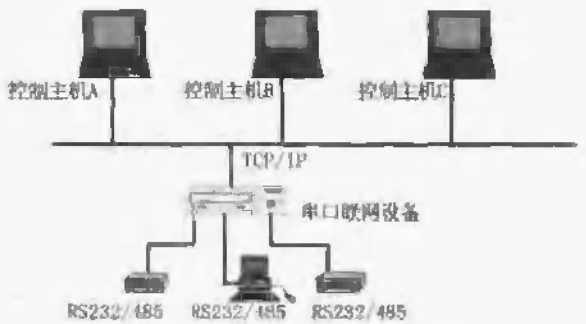


图 9.1.1 串口设备联网方法示意图

串口联网设备与控制主机是通过以太网口连接的，因此，一般需要设定控制主机和串口联网设备的 IP 地址。同时，在控制主机上，安装了网络与串口转换的驱动程序，这样，在控制主机上控制串口与传统方法就没有区别了，编程方法也一样，本书前面介绍的方法都可以应用，这些串口，我们可以理解为虚拟串口。

与传统的串口连接相比，它具备一些明显的优点。

- 形成多主系统：即多个主机控制一台串口设备，共享串口设备数据，可以不再是传统设备的一对一连接，这在某些应用场合非常有用，如需要进行主机备份以保证安全使用的场合；
- 延长了传统串口设备的数据传输范围：串口通信距离较短，就是 RS485 也不可能太长，但与网络结合后，这一限制基本上不再存在，只要是连接在网络上的设备，不论处在何处，都可以控制这些串口设备，这对于实现远程控制是非常方便的。

在串口联网设备产品中，目前系列齐全、应用较广的是台湾四零四科技（MOXA 公司 http://www.moxa.com.cn）的产品，公司网页上有更为详尽的解决方案，限于篇幅，这里不做详细介绍，作为实例，在下一节中我们介绍一款该公司应用范围较广的产品。

9.2 典型串口与联网的设备

本节我们介绍一款较典型的串口联网设备，MOXA 公司（http://www.moxa.com.cn）生产的 NPort 5400 系列串口联网服务器。NPort 5400 系列串口联网服务器的外形结构如图 9.2.1 所示。

图 9.2.1　NPort 5400 系列串口联网服务器的外观结构

在以下的介绍中，有些名词或专用词汇的理解可以去 MOXA 网站查阅咨询，这里只要了解其概貌就可以了。

9.2.1　NPort5400 系列产品的特点

NPort 5400 串口联网服务器为设置 IP 地址提供了简单易用的 LCM（液晶模块），具备界面自适应 10/100Mbps 的以太网接口，配置有 4 个 RS-232 或 RS-422/485 接口，所有串口信号带突波保护（15kV ESD），支持 TCP Server，TCP Client，UDP 和 Driver 模式，也支持 Web，

telnet 和 console 口进行设置，支持 SNMP MIB-II 网络管理，其中，NPort 5430I 带 2kV 光电隔离保护。产品的这些性能特点基本支持了联网需求，同时，也可以作为多串口卡使用。下面对这些特点做详细介绍。

➢ 让 4 个 RS-232 或 RS-422/485 串口设备立即联网

NPort 5400 系列为串口设备连接到以太网提供了便捷的传输方式，只需简单地配置就可将现存的串口设备连接上网络。既保障现有硬件投资，同时确保将来网络的扩展性能。此外，NPort 系列可以在串口和以太网接口中进行双向传输数据。可以通过网络同时集中管理串口设备和分散的主机管理。

➢ 简单易用的串口设备联网服务器

内建的 LCM 人机界面显示了所有的配置参数，并且配备了 4 个按钮来进行配置，或用来选择操作模式，设置服务器参数简单，包括 IP 地址、网络掩码和网关地址。并且，还可以监视服务器状态和数据传输，在安装过程中不需要额外的计算机来操作。

➢ Real COM/TTY 口

NPort 5400 系列就像给 Windows 主机增加额外的多串口卡一样，只是它是利用 TCP/IP 网络这个主要的优点。有了 NPort 的 Real COM/TTY 端口驱动，NPort 5400 系列产品上的串口被 Windows 操作系统认为 Real COM 端口，或者被 Linux 操作系统认为是 Real TTY 口。提供了基本的发送/接收数据功能，如 RTS，CTS，DTR，DSR，DCD（输入）和 Break 等控制信号。

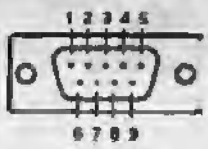
➢ 4 个独立的串口可以分别被配置成为不同的操作模式

不同的串口可以被配置成不同的模式，这确保了操作的多样性。NPort 5400 系列产品可以通过 TCP/IP 网络来连接不同的设备获取远程数据、事件处理或者数据广播等。例如，可以配置 NPort 5400 的端口 1 为 driver 模式，端口 2 为 TCP Server 模式，端口 3 和 4 为 TCP Client 模式下工作。

➢ 规格与接口规范

串行接口：NPort 5400 的串行接口与普通串口一致。

RS-232, DB9 针式: TxD, RxD, RTS, CTS, DTR, DSR, GND, DCD，针脚安排如图 9.2.2 所示。



Pin	RS-232
1	DCD (输)
2	RxD (输)
3	TxD (出)
4	DTR (出)
5	GND
6	DSR (输)
7	RTS (出)
8	CTS (输)
9	—

图 9.2.2　DB9-RS232 接口

NPort 5430 和 NPort 5430I 为 RS-422/485 协议。

RS-422: Tx+, Tx-, Rx+, Rx-, GND

RS-485 (4 线): Tx+, Tx-, Rx+, Rx-, GND

RS-485 (2 线): Data+, Data-, GND 针

串行通信参数与标准串口一致。校验位: None, Even, Odd, Space, Mark；数据位: 5, 6, 7, 8；停止位: 1, 1.5, 2；流量控制: RTS/CTS, XON/XOFF；速率: 50 bps ~ 230.4 Kbps。

网络接口

以太网 TCP/IP: 10/100 Mbps, RJ45。

安全措施

为以太网配置内嵌 1.5 kV 电磁隔离保护；为串行线所有的信号 15 KV ESD 保护提供保护，NPort 5430I 提供 2kV 光电隔离；为电源线 4 kV/2kV 电源突波 EFT 保护。

RS-485 数据流向

享有专利保护的数据流向自动控制。

软件特点

协议: ICMP, IP, TCP, UDP, DHCP, BootP, Telnet, DNS, SNMP, HTTP, SMTP, SNTP

工具: NPort 管理员，应用于 Windows 95/98/ME/NT/2000/XP/2003

Real COM/TTY 驱动: Windows 95/98/ME/2000/XP

Real COM 驱动, Linux Real TTY 驱动

配置: Web 浏览器, 串口 telnet console 口或 Windows 工具

电源输入

12 ~ 48 VDC

电源消耗

5410: 385 mA (at 12V max.)；5430: 380 mA (at 12V max.)；5430I: 600 mA (at 12V max.)

工作环境

操作温度：0~55° C (32 ~ 131° F), 5 ~ 95% RH

储存温度：-20~85° C (-4 ~ 185° F), 5 ~ 95% RH

通过认证：EMC: FCC Class A, CE Class A

其他特点

(ADDC™)功能，内嵌的 HMI，带有 4 个按键的 LCM 显示器，内嵌蜂鸣器，内嵌实时时钟，内嵌看门狗计时器

9.2.2　NPort 5400 系列产品的典型应用介绍

这里介绍种 Nport 5400 系列产品的典型应用方案。

（1）只使用一个 IP 地址在以太网络上控制多串口设备

如图 9.2.3 所示，为了自动地获取远程的数据， NPort 5400 系列可以与 4 个串行设备连接而仅使用一个 IP 地址。通过指定 IP 地址和 TCP 端口号，一个主机可以通过网络来对不同的串行设备进行存取。例如：连接到 192.168.10.2:4001 可以通过 NPort 5410 的第一个串口进行数据存取。

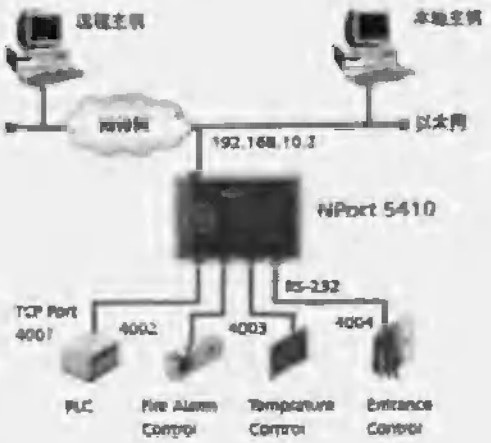


图 9.2.3　典型应用 1：以一个 IP 地址联网控制多串口设备

（2）通过集中连接来管理串口设备，扩展服务器共享的灵活性

如图 9.2.4 所示，不同的主机可以共享同一个 NPortr 5400 系列，来控制不同的设备。例如，NPort 5410 的端口 1 和 2 可以配置成为主机 A 的 COM3 和 COM4，NPort 5410 的端口 3 和 4 可以配置成为主机 B 的 COM3 和 COM4。

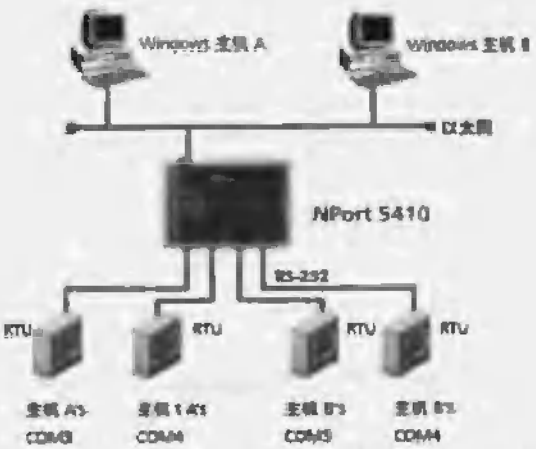


图 9.2.4　典型应用 2：通过集中连接来管理串口设备

（3）集中的 RS-422/485 串口设备控制

如图 9.2.5 所示，最多可有 31 个 RS-485（2 线或 4 线）设备或 9 个 RS-422（4 线）设备可以通过 NPort 5430 的串口连接到以太网上。使用 Web console 口或 NPort 5000 系列 Windows 工具可以为 RS-422/485 的每一个串口进行配置。

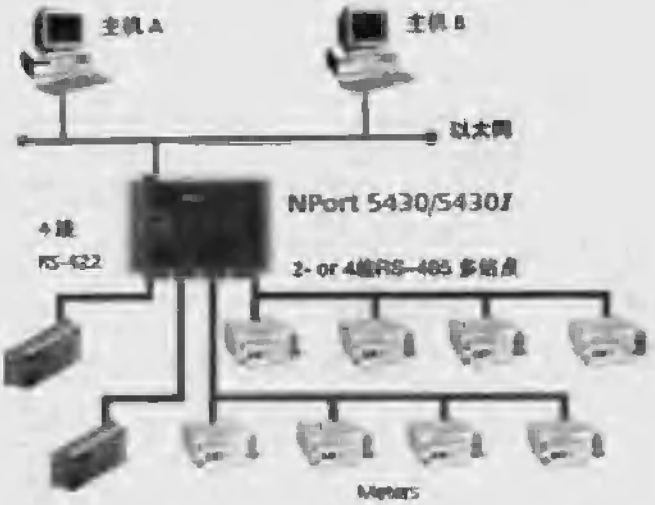


图 9.2.5　典型应用 3：集中的 RS-422/485 串口设备控制

9.2.3　NPort5400 系列产品的设置与编程测试

NPort5400 系列产品的编程，在产品附带的管理软件 NPort5400setup.exe 中有非常详尽的编程实例，针对的编程环境有 VB、VC 和 Dephi。

NPort5400setup.exe 是一个安装应用程序，与普通的安装程序一样，安装后打开的调试设置界面如图 9.2.6 所示。在管理程序中可以对串口号、服务器 IP 地址、通信参数等进行设置，也可以测试通信功能，程序中有详尽的帮助信息。具体的应用在此不做详述。

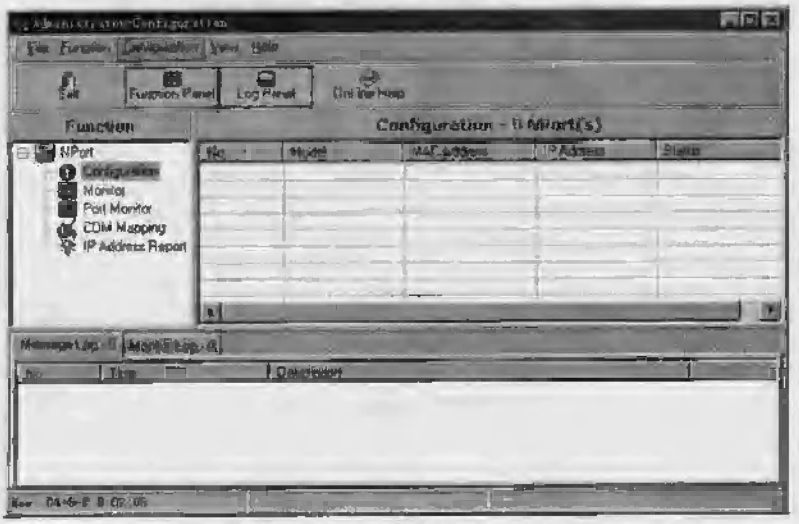
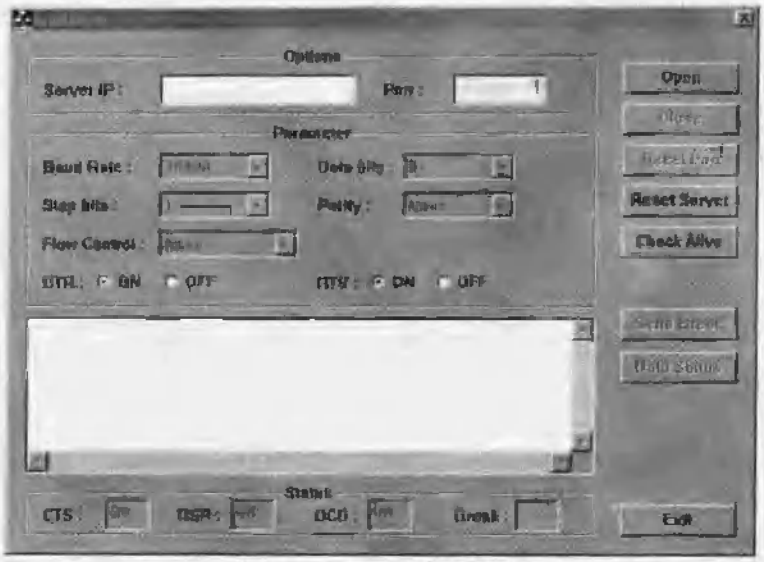


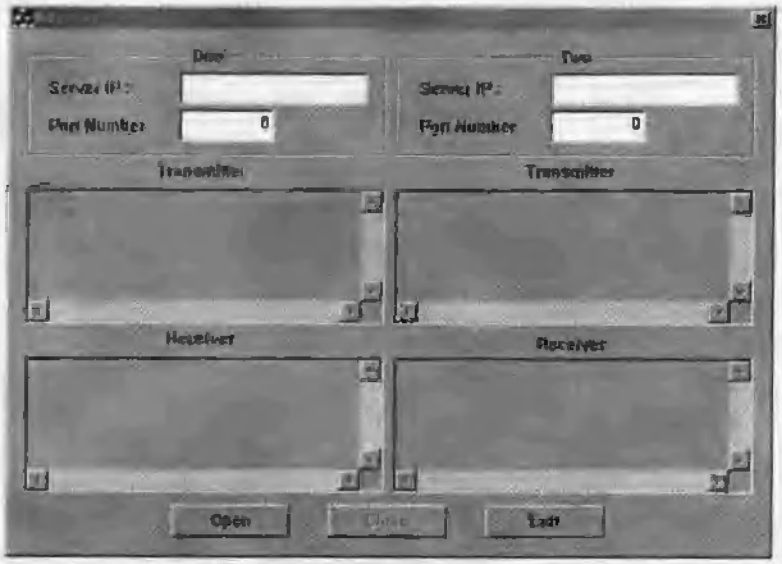
图 9.2.6　NPort5400 系列产品管理器

在安装文件夹中，找到 Example 文件夹（一般在 C:\ Program Files\NPortAdminSuite

\IPSerial\Examples)，在其中的 VC 文件夹中会看到 SinglePort 和 MultiPort 两个实例，可以将相应工程打开，在 VC 6.0 中运行程序，就可以详细按照实例编程了。图 9.2.7(a)和(b)分别是 SinglePort 和 MultiPort 运行后的程序界面。



（a）SinglePort 示例程序界面



（b）MultiPort 示例程序界面

图 9.2.7　产品示例程序界面

当然，也可以按照我们前面介绍的普通方法编程来实现对串口的控制，只要在设备管理器中（Windows 设备管理器或产品自带的管理器均可）设置好串口号，就可以以自己熟悉的方式来编写程序。

9.3　与 Access 数据库结合的串口通信实例

在有些工程应用中，不仅需要进行串口通信，还需要存储串口获取的数据，以备查询或后续处理。因此，本节将结合 Access 数据库对某排气工艺微机网络检测系统实际开发应用进行说明。

编程任务：

本例中，我们首先介绍 ODBC 数据源链接方法及 VC++中对数据库的管理，然后根据界面的实际要求，介绍了静态三叉切分窗口的方法；在串口通信方面运用了 MSCOMM 控件，并编写了通信处理函数。

9.3.1　微机网络检测系统说明

1．检测网络简介

本检测网络由 55 台复合计、55 台温度数显仪，RS422 接口总线及上位计算机构成。

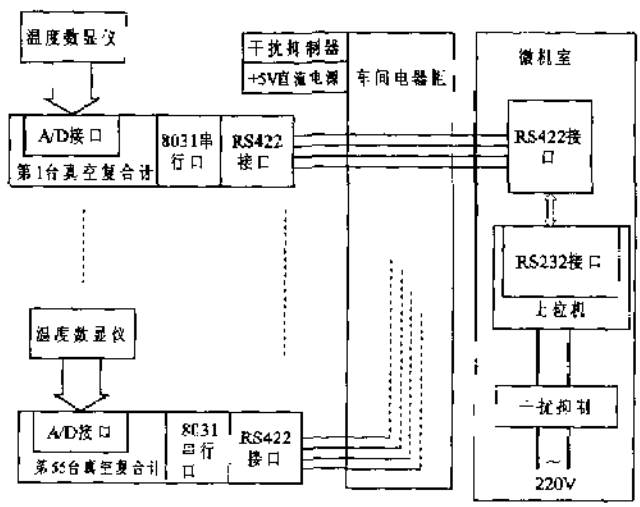


图 9.3.1　网络构成示意图

图 9.3.1 为其构成示意图：上位机分时选通 No.01 到 No.55 各复合计，并采集被选通复合计的温度和真空度值，并以一定的格式存于文件中。当某工况结束时，由上位机对其温度和真空度流程进行数据分析，总结出各关键工艺点和数据，显示于屏幕上，同时也可打印存储。

2．对上位微机的使用说明

图 9.3.2 为上位机构成示意图。上位机界面显示能实时显示采样数据，并动态显示变化曲线。由于有存储要求，所以本例中通过 ODBC 链接 Access 数据库源实时存储数据并随时可供调用。

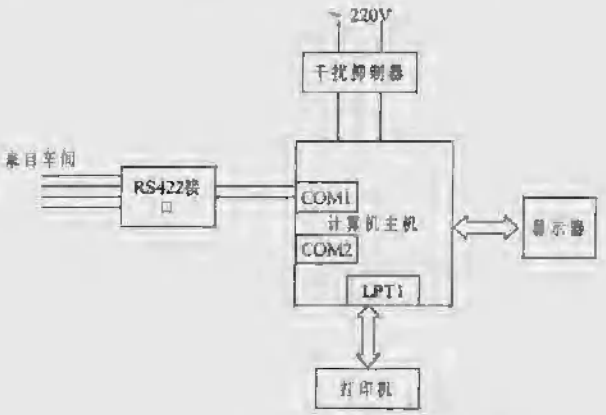


图 9.3.2　上位机构成示意图

经分析，确定上位机界面如图 9.3.3 所示。

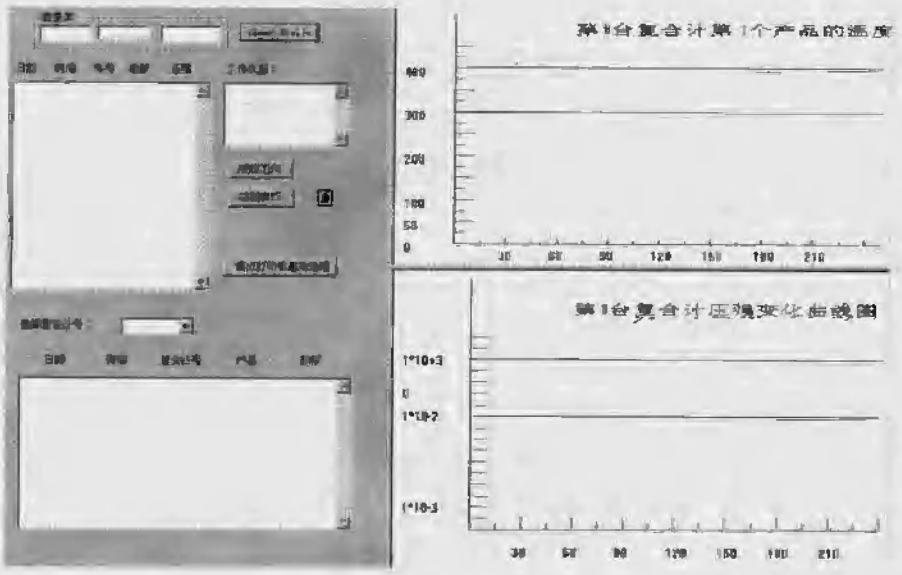


图 9.3.3　上位机界面

9.3.2　创建 ODBC 数据源

（1）首先在控制面板中，打开 ODBC 数据源，单击用户 DSN 选项，在出现的界面中单击添加，如图 9.3.4 所示。

图 9.3.4　创建新数据源

然后，单击完成按钮，出现图 9.3.5，填入数据源名（本例中设为 biao），单击确定即可。

图 9.3.5　安装数据源

（2）在 Access 中创建数据库，本例中为 data（只包含 number,name,score 三列）。

9.3.3　创建工程

（1）打开 File 菜单的 New 选项，选取 Projects，选择 MFC AppWizard (exe)，填入工程名，本例为 xxx。

（2）把数据库文件 data 拷入新建的工程目录。

（3）应用程序的类型指定为 MDI，在 Step2 对话框中选择 Header Files Only 选项。

（4）在“Advanced Options”对话框的“Window Styles”选项卡中单击“Use split window”选项，如图 9.3.6 所示。

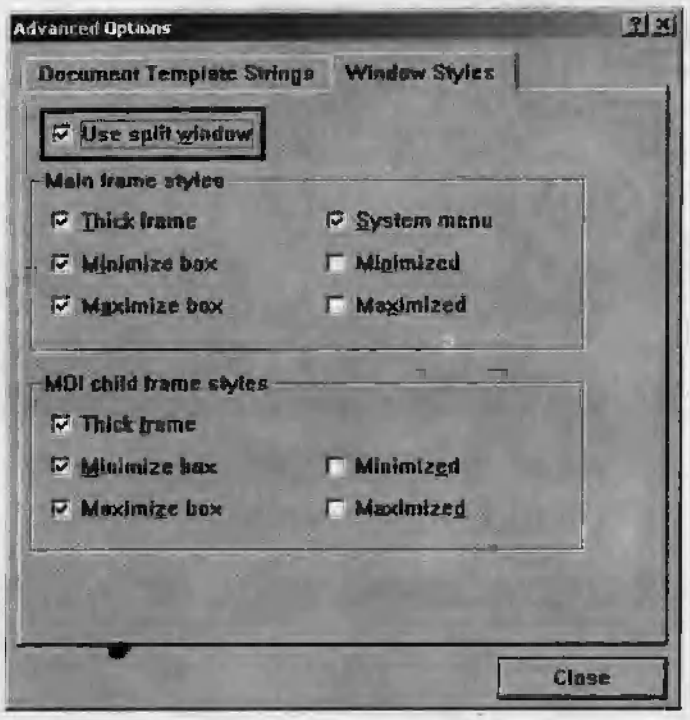


图 9.3.6　“Advanced Options”对话框

（5）将视图基类指定为 CscrollView，如图 9.3.7 所示。

图 9.3.7　指定基类

单击“Finish”按钮即完成创建工程。

（6）用 ClassWizard 创建记录集类。从 Add Class 菜单中选择 New，并按图 9.3.8 填充对话框。

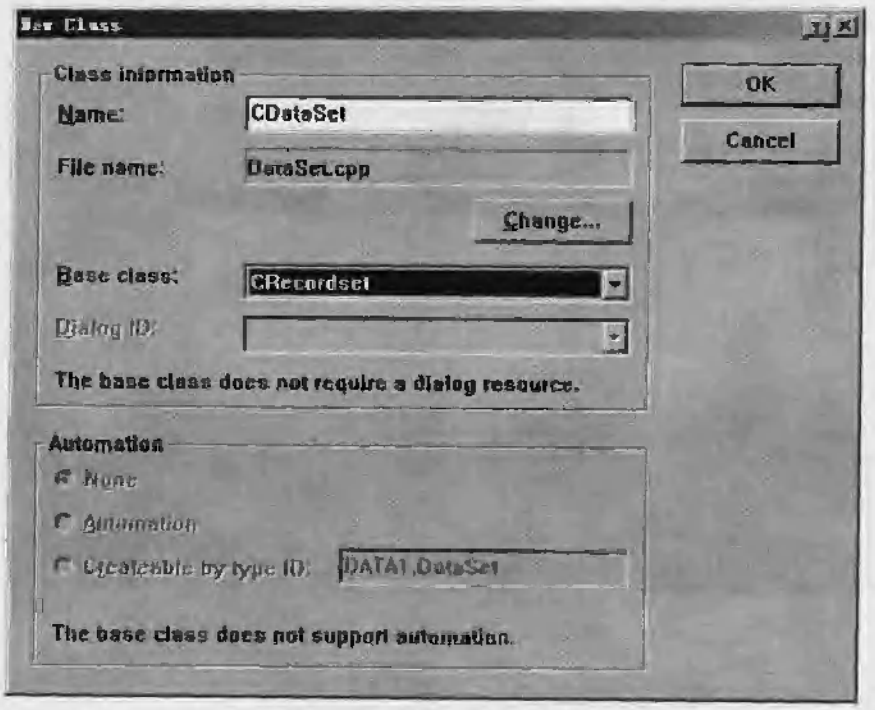


图 9.3.8　创建记录集类

（7）单击 OK，进入 Data Source，选择 biao 数据源,并选择 Dynast 选项，如图 9.3.9 所示。

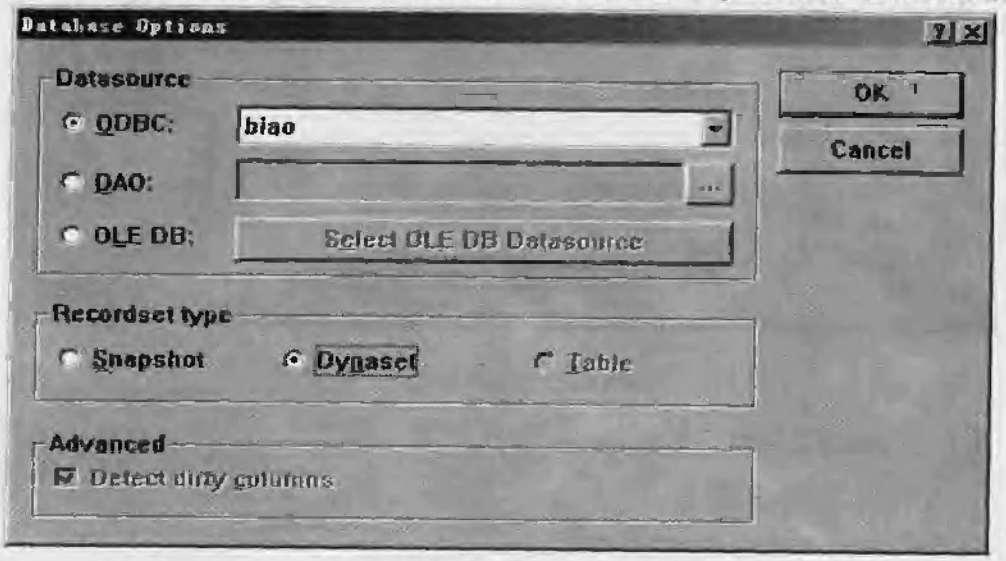


图 9.3.9　Data Source 对话框

在选择了数据源之后，按 ClassWizard 提示选择需要的表。

（8）在 ClassWizard 中，针对新产生的 CDataSet 类,单击 Member Variables 标签。此时，ClassWizard 应按照数据库列的名称产生了如图 9.3.10 所示的数据成员。

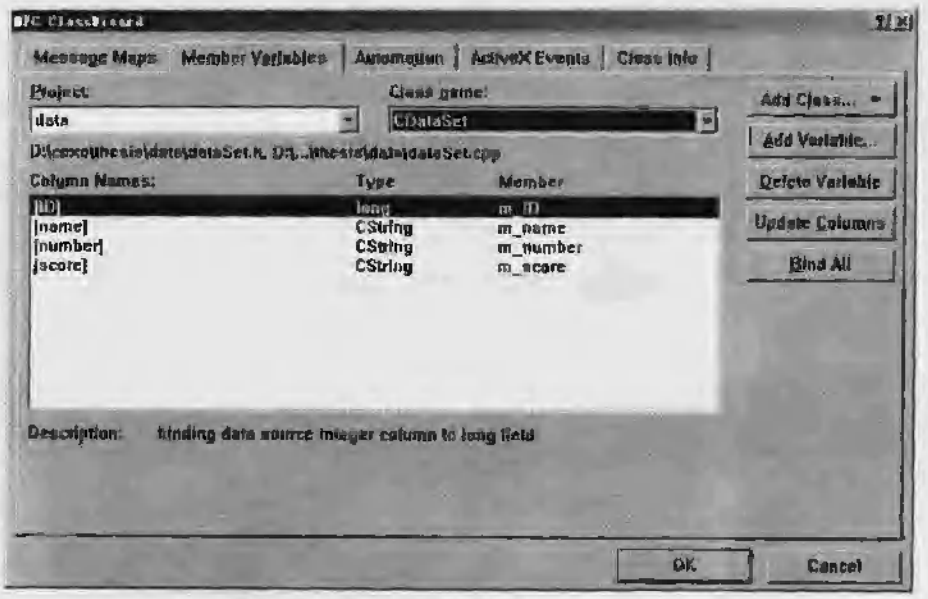


图 9.3.10　ClassWizard 对话框

（9）在 CXxxView 类中添加数据成员，如图 9.3.11 所示。

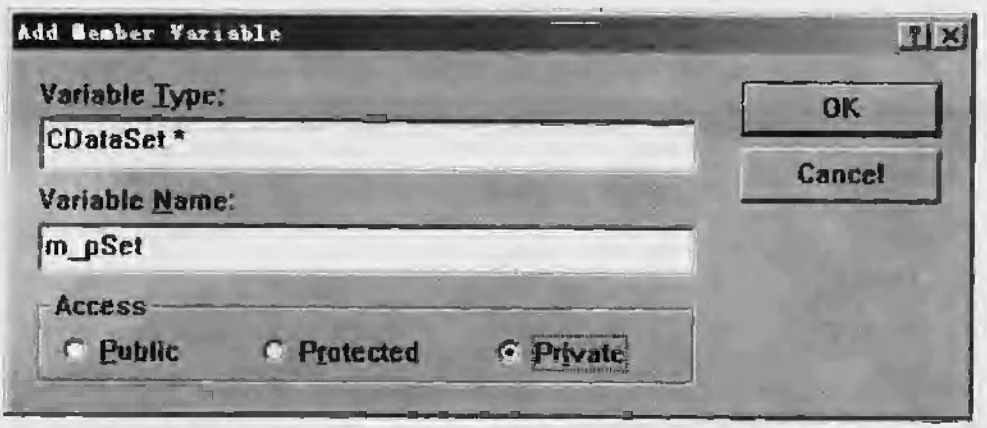


图 9.3.11　添加数据成员

（10）在 CXxxDoc 中添加成员，如图 9.3.12 所示。

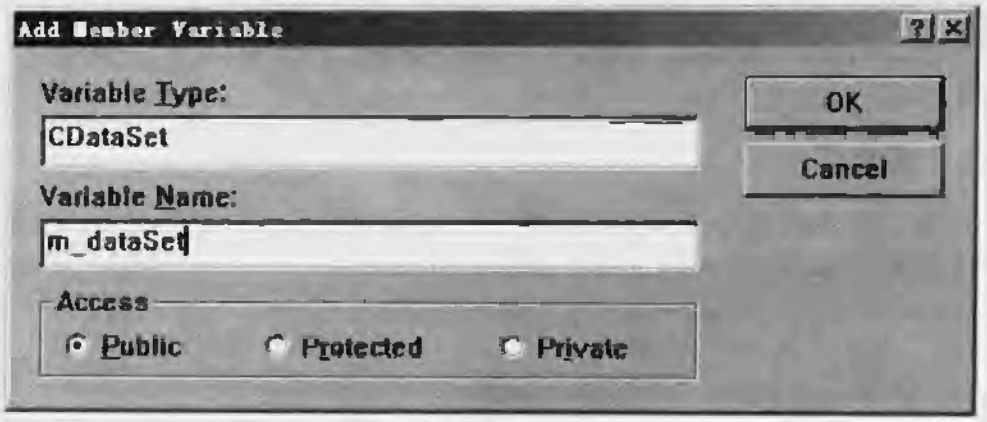


图 9.3.12　添加成员

（11）按前述上位机界面要求，创建一个如图 9.3.13 所示的对话框，其中使用了串口控件，并使该对话框与新类 Cmycomm 对应起来，该类的基类为 CrecordView。此外创建两个新类 CmyDraw1,CmyDraw2，其基类均为 CView，用来完成温度及压强实时曲线图的绘制。

图 9.3.13　串口通讯对话框

（12）添加串口事件消息处理函数 OnComm() 。

打开 ClassWizard—>Message Maps，为串口控件对象添加 OnComm 消息处理函数，并命名为 OnComm。

9.3.4 程序简介

（1）由于在 Step4 的“Advanced”对话框的“Window Styles”选项卡中单击了“Use split window”选项，可以看到 OnCreateClient 函数内容如下所示：

```
BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
    CCreateContext* pContext)
{
    return m_wndSplitter.Create( this,
        2, 2,             // TODO: adjust the number of rows, columns
        CSize( 10, 10 ),     // TODO: adjust the minimum pane size
        pContext );
}
```

该语句表明，当前窗口切分为 2 行 2 列。而本例中，设计界面是要求左半部分为通讯对话框，右半部分则一分为二，因此必须修改 OnCreateClient 函数内容。

首先定义如下成员变量 m_wndSplitter2：

```
protected:
CSplitterWnd m_wndSplitter2;
```

然后修改函数内容如下：

```
BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
    CCreateContext* pContext)
{
    m_wndSplitter.CreateStatic( this,1,2);//窗口为1行2列
    m_wndSplitter.CreateView(0,0,pContext->m_pNewViewClass,
        CSize(512,600),pContext);

   m_wndSplitter2.CreateStatic( &m_wndSplitter,2,1,//将第2列再分成2行1列
        WS_CHILD|WS_VISIBLE,m_wndSplitter.IdFromRowCol(0,1));
m_wndSplitter2.CreateView(0,0,RUNTIME_CLASS(CmyDraw1),
        CSize(308,300),pContext);
    m_wndSplitter2.CreateView(1,0,
        RUNTIME_CLASS(CmyDraw2),
        CSize(308,300),pContext);
    return 1;
}
```

（2）本例使用了 MSCOMM 控件，其成员变量定义如下所示：

```
CMSComm  m_ctrlComm;
```

打开串口：

```
void CXXXView::OnOpencomm()
{
    m_bOpenPort=!m_bOpenPort;
    if(m_bOpenPort)
    {
        m_ctrlOpenPort.SetWindowText("打开串口");
```

```
            if(m_ctrlComm.GetPortOpen())
          m_ctrlComm.SetPortOpen(FALSE);
            m_ctrlIconOpenoff.SetIcon(m_hIconOff);
        }
        else
        {
            m_ctrlOpenPort.SetWindowText("关闭串口");
          m_ctrlComm.SetCommPort(1); //选择 com1
        if( !m_ctrlComm.GetPortOpen())
          m_ctrlComm.SetPortOpen(TRUE);//打开串口
        else
            {
                AfxMessageBox("没有发现此串口或被占用");
              m_ctrlIconOpenoff.SetIcon(m_hIconOff);
            }
    m_ctrlComm.SetInputMode(1);   //1: 表示以二进制方式检取数据
    m_ctrlComm.SetRThreshold(1); //参数 1 表示每当串口接收缓冲区中有多于或等于 1 个字符时将引发一个
                                 //接收数据的 OnComm 事件
    m_ctrlComm.SetInputLen(0);   //设置当前接收区数据长度为 0
    m_ctrlComm.GetInput();       //先预读缓
    m_ctrlIconOpenoff.SetIcon(m_hIconRed);
        }
    }
```

消息处理函数如下：

```
void CXXXXView::OnComm()
{
      VARIANT variant_inp;
   COleSafeArray safearray_inp;//COleSafeArray 操作任意类型，任意维数的数组
   LONG len,k;
   BYTE rxdata[2048]; //设置 BYTE 数组 An 8-bit integerthat is not signed.
   CString strtemp;

  if(m_ctrlComm.GetCommEvent()==2) //事件值为 2 表示接收缓冲区内有字符
      {        ////////以下内容，用户可以根据自己的通信协议加入相应的处理代码
      if(count<=2)
          {
           variant_inp=m_ctrlComm.GetInput(); //读缓冲区
         safearray_inp=variant_inp; //VARIANT 型变量转换为 COleSafeArray 型变量
         len=safearray_inp.GetOneDimSize(); //得到有效数据长度，返回一维 COleSafeArray
                                            //对象中的元素数
         for(k=0;k<len;k++)
         safearray_inp.GetElement(&k,rxdata+k);//转换为 BYTE 型数组
         count++;//在构造函数已置初值为 0
          }
    }

  UpdateData(FALSE); //更新编辑框内容
     CString m_strRXData1;
switch(count)
{
case 1:
     {
       ...//握手处理，判断是哪台下位机，并发信号要求该下位机做相应操作
     }
  case 2:
  {
    if(!m_pSet->IsBOF()) m_pSet->MoveLast();  //更新 Access 数据库记录集
```

```
        m_pSet->AddNew();
        UpdateData(TRUE);

        m_strRXData1.Format("%c%c%c",rxdata[0],rxdata[1],rxdata[2]);
    char* temp=(char*)(LPCTSTR)m_strRXData1);
    char tbuf[10];

    tbuf[0]=temp[0]; tbuf[1]=temp[1]; tbuf[2]=temp[2];tbuf[3]=0;
    m_pSet->m_temperature=atoi(tbuf);//得到温度值并写入记录

        m_pSet->m_pressure.Format("%c*10-%c",rxdata[3],rxdata[4]);//把压强写入记录
                                                                  //(用字符串形式)
        m_pSet->m_number=(long)m_address;//把地址写入记录
        UpdateData(FALSE);
    ...//把获取的数据做适当处理后存入数组中
        if (m_pSet->CanUpdate())
{                                   //记录处理语句
            if(!m_pSet->Update()) AfxMessageBox(_T("增加新记录失败!"));
        }
        if(!m_pSet->IsEOF()){
            m_pSet->MoveLast();
        }

    count=0;//标志位清 0

    CXxxDoc *pDoc=GetDocument();
        pDoc->UpdateAllViews(NULL);// 更新视图
      }
  }
  }
```

限于篇幅，温度及压强曲线的绘制程序此处不再赘述。

9.4　与 WinSock 结合的串口通信实例

事实上，串口通信也是网络通信的一种，只是随着以太网的飞速发展，通常意义上的网络通信就只指以太网的通信了。在主从结构的通信中，串口通信中只能有一个主设备，而网络通信则可以是多主系统。这一点，与现场总线和 RS232、RS485 的区别是类似的。但不管编程采用什么通信方式，编程要完成的任务就是要把数据、消息等信息发送出去或接收到本地，因此，各种程序中，数据的处理方式基本是一致的，前面我们提到的各种数据处理方法基本上是通用的，所以，不管将来硬件系统如何发展，只要我们掌握了串口通信的编程，就能以不变应万变，在各种编程方式中均能得心应手。

编程任务

本例中，我们首先要用 WinSockets 类来编写一个 C/S(Client/Server)程序，C/S 程序包含一个服务器端和一个客户端，服务器程序接收到客户端程序发来的数据后，将数据从串口 1（COM1）发送出去，我们可以用串口调试助手来接收这些数据。同时，为了调试方便，我们把每个程序发送和接收的数据都显示出来。

在正式编程之前，有必要简单了解一下 WinSockets 类，它包括 CSocket 类和 CAsyncSocket 类，前者支持阻塞 I/O 操作，后者使用非阻塞操作。所谓阻塞就是一个函数或语句在完成本

身的任务之前不允许调用其他函数或语句；而非阻塞操作则是函数或语句启动后如能得到执行结果就近回结果，否则就返回错误代码，是异步操作。

CSocket 类是从 CAsyncSocket 类继承来的，它为编程者提供了更高的抽象性，使程序员从编程的底层细节中解脱出来，因而更容易使用。

在 VC/MFC 环境下，编写网络通信程序有几种方法供选择，若要详细了解，读者可以阅读有关的参考书，这里我们应用 CSocket 类来编写这个程序。C/S 程序需要分别编写 Client 端和 Server 端程序，下面分别描述。

9.4.1 客户端应用程序

客户端程序要完成以下功能：

- 建立与服务器程序的连接
- 向服务器程序发送数据信息，接收从服务器程序回传的信息
- 关闭与服务器程序的连接

1. 建立程序框架工程

在 VC 6.0 集成开发环境中，新建基于对话框（Dialog based）的 MFC AppWizard(exe)应用程序，工程名为 Sclient。

在应用程序向导第 2 步中（MFC AppWizard – Step 2 of 4）选中 Windows Sockets 复选框，如图 9.4.1 所示。其他选项保持不变。选中 Windows Sockets 后，MFC AppWizard 就会自动在建立的应用程序框架中加入 WinSocket 初始化代码。如果有的程序已经编写好了，需要临时加入网络的功能，可以参考第 9.5 节解决这个问题。

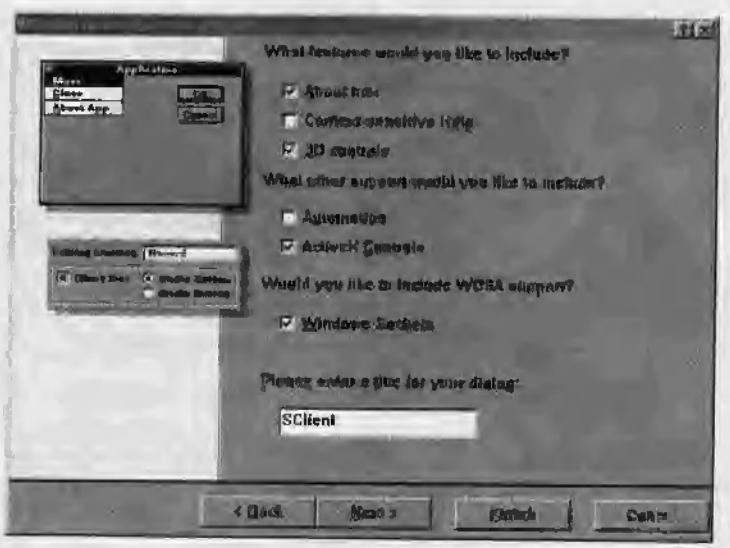


图 9.4.1　选中 Windows Sockets 复选框

2. 添加对话框控件

为主对话框添加如图 9.4.2 所示的控件，并设置其属性，并用 ClassWizard 为相应控件添加变量。添加后如表 9-4-1 所示。

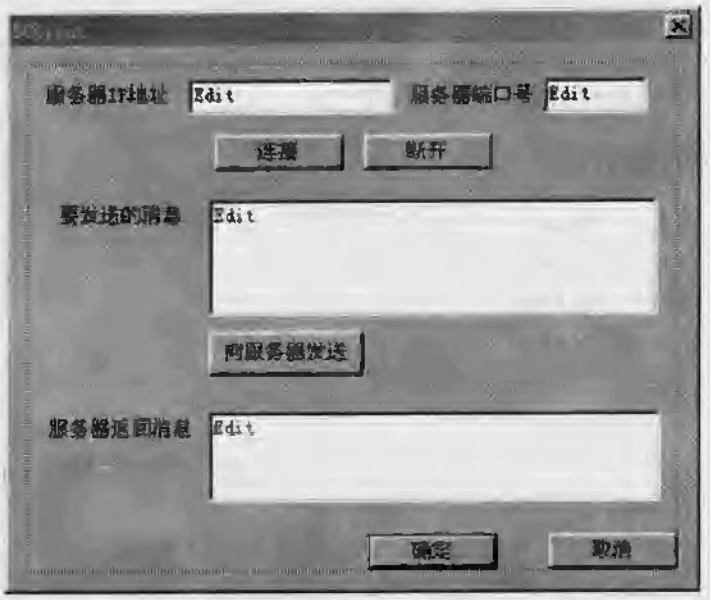


图 9.4.2　对话框界面设置情况

表 9-4-1　控件及其属性设置情况

控件	控件 ID	Caption	需要添加的变量及变量类型
静态文本	IDC_STATIC	服务器 IP 地址	
静态文本	IDC_STATIC	服务器端口号	
静态文本	IDC_STATIC	要发送的消息	
编辑框	IDC_EDIT_IPADDR		m_strEditIPAddr Value CString
编辑框	IDC_EDIT_PORTNO		m_unEditPortNO　Value UINT
编辑框	IDC_EDIT_SENDDATA		m_strEditSendData　Value CString
编辑框	IDC_EDIT_RECVDATA		m_strEditRecvData　Value CString
按钮	IDC_BUTTON_LINK	连接	
按钮	IDC_BUTTON_UNLINK	断开	
按钮	IDC_BUTTON_SEND	向服务器发送	

3. 编写 Csocket 派生类——CMySocket 类

从菜单 Insert->New Class 命令来添加 CMySocket 类，设置情况如图 9.4.3 所示。注意，新添加的类的基类应该为 Csocket 类。添加 CMySocket 类后，ClassWizard 生成了 MySocket.h 和 MySocket.cpp 两个文件。

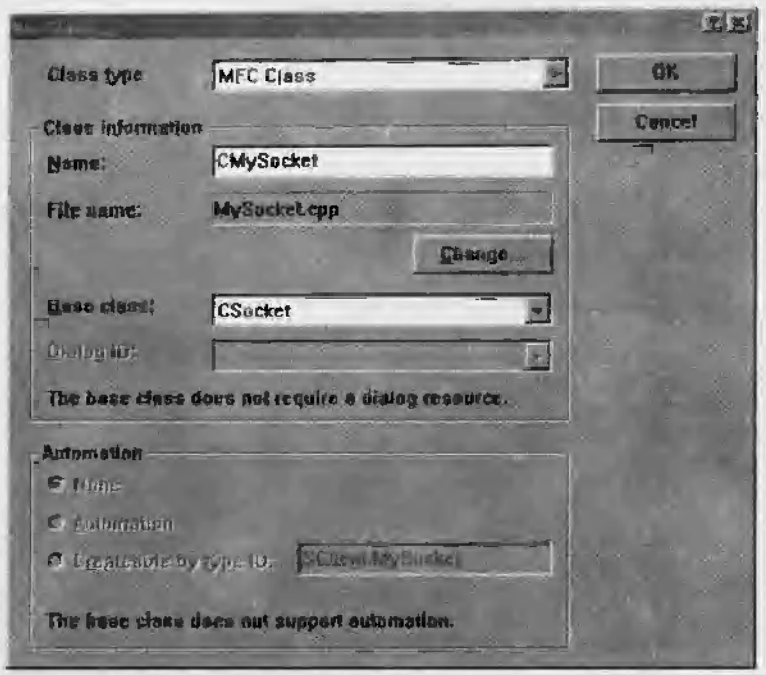


图 9.4.3　添加 CMySocket 类

在这个添加的类中，我们要完成对从服务器接收数据的处理，通过 ClassWizard 添加 OnReceive()函数，如图 9.4.4 所示。关于网络程序的数据接收，我们在这里应注意，OnReceive()函数实质上是一个消息处理响应函数，它重载了 Receive()函数，就像 DOS 程序的中断处理一样，我们也可以自己用查询方式来接收函数，或者新开一个线程来处理。

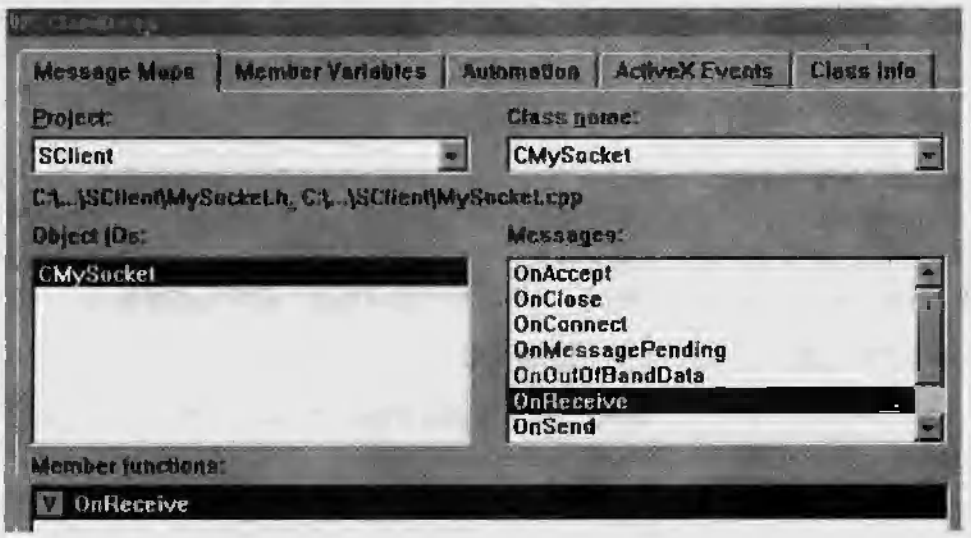


图 9.4.4　添加 OnReceive()函数

```
// MySocket.cpp : implementation file
#include "stdafx.h"
#include "SClient.h"
#include "MySocket.h"
#include "SClientDlg.h"  //添加的主对话框类

……(此处略)

/////////////////////////////////////////////////////////////////////////////
// CMySocket member functions

void CMySocket::OnReceive(int nErrorCode)
{
    // TODO: Add your specialized code here and/or call the base class
    char chMsg[5120], chMsgTemp[1024];
    UINT unRXCharNum;   //每次读取的字符数
    BOOL bEndFlag=0;  //接收完毕标志
    strcpy(chMsg,"");
    do
    {
        strcpy(chMsgTemp,"");
        unRXCharNum=Receive(chMsgTemp,1000);
        if(unRXCharNum>1000||unRXCharNum<=0)
        {
            AfxMessageBox("接收数据中出错",MB_OK);
            return;
        }
        else if(unRXCharNum<1000 && unRXCharNum>0)
        {
        bEndFlag=1;
        }
        chMsgTemp[unRXCharNum]=0; //加上字符串结束位
        strcat(chMsg,chMsgTemp);
    }while(bEndFlag==0);
  //CSClientDlg* pDlg=(CSClientDlg*)GetParent();
    CSClientDlg* pDlg=(CSClientDlg*)AfxGetMainWnd(); //得到主框口(对话框)指针
    pDlg->m_strRXDataTemp.Format("%s",chMsg);
    pDlg->UpdateRXData();  //在主对话的接收编辑框中显示接收到的数据

    CSocket::OnReceive(nErrorCode);
}
```

在 SClientDlg.h 中添加 CMySocket 类头文件“MySocket.h”，再添加指针 CMySocket 类的指针 m_pMySocket，用于建立客户端。这里注意一下，必须用指针，而不能用类对象，这是因为 OnReceive 是虚函数，而指针具有多态性，可以保证接收到数据后激发 OnReceive（）函数。再添加布尔变量 m_bLinked，以判断网络是否连接，初始状态置为 FALSE。

4. 编写主对话框类 CSClientDlg 的相关函数

首先为对话框的“连接”、“断开”和“向服务器发送”按钮分别添加单击响应函数 OnButtonLink()、OnButtonUnlink()和 OnButtonSend()函数。

```
// SClientDlg.cpp : implementation file
//
……(此处略)

CSClientDlg::CSClientDlg(CWnd* pParent /*=NULL*/)
```

```
    : CDialog(CSClientDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CSClientDlg)
    m_strEditIPAddr = _T("10.1.37.170"); //地址读者可以更改成自己的IP
    m_unEditPortNO = 5050; //注意了：如果安装了病毒防火墙，要把相应端口打开
    m_strEditSendData = _T("");
    m_strEditRecvData = _T("");
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_bLinked=FALSE;  //网络是否连接，初始状态置为断
}

……(此处略)

BOOL CSClientDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

……(此处略)

    // TODO: Add extra initialization here
    GetDlgItem(IDC_BUTTON_LINK)->EnableWindow(!m_bLinked);
    GetDlgItem(IDC_BUTTON_UNLINK)->EnableWindow(m_bLinked);

    return TRUE; // return TRUE unless you set the focus to a control
}


void CSClientDlg::OnButtonLink()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    m_pMySocket=new CMySocket;
    m_pMySocket->Create();
    m_bLinked=m_pMySocket->Connect(m_strEditIPAddr, m_unEditPortNO);
    if(!m_bLinked)
    {
        AfxMessageBox("连接服务器失败");
    }
    GetDlgItem(IDC_BUTTON_LINK)->EnableWindow(!m_bLinked);
    GetDlgItem(IDC_BUTTON_UNLINK)->EnableWindow(m_bLinked);
}

void CSClientDlg::OnButtonUnlink()
{
    // TODO: Add your control notification handler code here
    if(!m_bLinked) return; //如果没有连接，就返回
    if(m_pMySocket->ShutDown(2))//关闭接收与发送
    {
        m_bLinked=FALSE;
        Sleep(50);
        m_pMySocket->Close();
        if(m_pMySocket) delete m_pMySocket;
    }
    GetDlgItem(IDC_BUTTON_LINK)->EnableWindow(!m_bLinked);
    GetDlgItem(IDC_BUTTON_UNLINK)->EnableWindow(m_bLinked);
}

void CSClientDlg::OnButtonSend()
```

```
{
    // TODO: Add your control notification handler code here
    if(!m_bLinked)
    {
        AfxMessageBox("没有连接网络");
        return;
    }
    UpdateData(TRUE);
    m_pMySocket->Send(m_strEditSendData,m_strEditSendData.GetLength(),0);
}

void CSClientDlg::UpdateRXData()
{
    m_strEditRecvData=m_strRXDataTemp;
    UpdateData(FALSE);
}
```

要注意一点，我们在网络程序中一般应将端口号设置为大于 1024 的值，这是因为从 1 至 1024 号的端口是系统保留端口号。

到这里，我们就完成了网络程序客户端的编程，下面开始编制服务器程序。

9.4.2 服务器应用程序

服务器程序要完成以下功能：

- 创建监听 Socket 进行监听，当有客户端请求连接时，建立一个新的接收 Socket 处理这个客户的数据发送与接收；
- 接收客户端发来的数据，并向客户端返回相应信息，本程序向客户端返回信息为接收到的信息；
- 将接收的信息通过串口 1(COM1)发送出去。

下面是具体编程步骤，由于本程序与客户端程序有许多相同之处，这里不做详细说明。

1. 建立工程

工程名为 Sserver,其余同第 9.4.1 节客户端程序，在工程向导中别忘了选中 Windows Sockets。

2. 添加对话框控件

为主对话框添加如图 9.4.5 所示的控件，并设置其属性，并用 ClassWizard 为相应控件添加变量。添加后如表 9-4-2 所示。

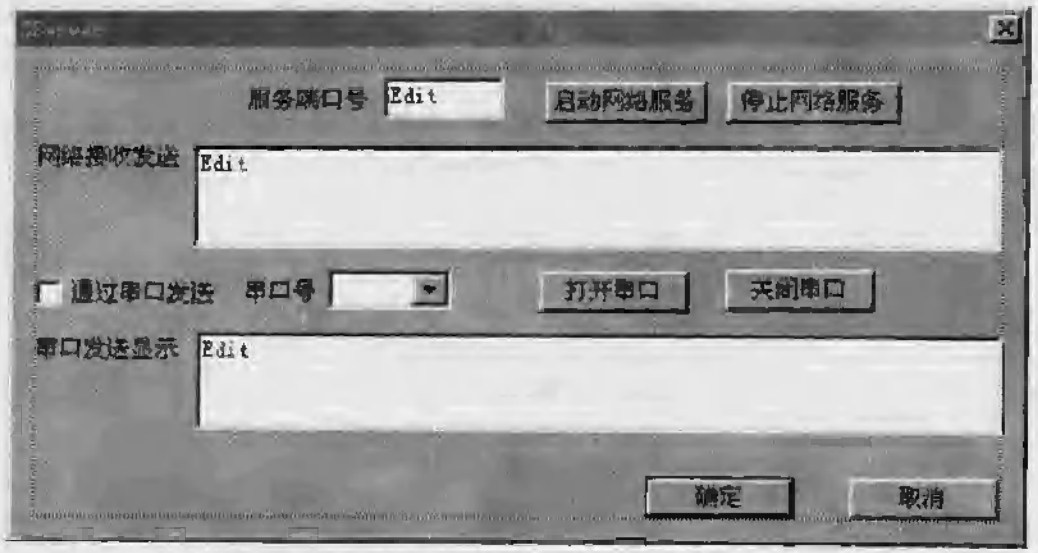


图 9.4.5　服务器程序对话框界面设置情况

表 9-4-2　控件及其属性设置情况

控件	控件 ID	Caption	需要添加的变量及变量类型
静态文本	IDC_STATIC	服务端口号	
静态文本	IDC_STATIC	网络接收发送	
静态文本	IDC_STATIC	串口发送显示	
静态文本	IDC_STATIC	串口号	
组合框	IDC_COMBO_COMPORT		m_ctrlComboComPort Control
编辑框	IDC_EDIT_NETMSG		m_strEditNetMsg Value CString
编辑框	IDC_EDIT_PORTNO		m_unEditPortNO　Value UINT
编辑框	IDC_EDIT_COMMSG		m_strEditComMsg Value CString
按钮	IDC_BUTTON_START	启动网络服务	
按钮	IDC_BUTTON_STOP	停止网络服务	
按钮	IDC_BUTTON_OPEN	打开串口	
按钮	IDC_BUTTON_CLOSE	关闭串口	
按钮	IDC_BUTTON_SEND	向服务器发送	
复选框	IDC_CHECK_SENDDATA	通过串口发送	m_ctrlCheckSendData Control

3. 编写 CSocket 派生类——CListenSocket 类和 CAcceptSocket 类

从菜单 Insert->New Class 命令来添加 CListenSocket 类和 CAcceptSocket 类，设置情况参考图 9.4.1.3 所示操作方法。注意，新添加的类的基类应该为 CSocket 类。

CListenSocket 类响应 FD_ACCEPT 事件，调用 OnAccept()函数创建客户请求，并把请求放入 Socket 接收队列中。OnAccept()重载操作方法参考图 9.4.4。

ListenSocket.cpp 文件的编写如下：

```
// ListenSocket.cpp : implementation file
#include "stdafx.h"
#include "SServer.h"
#include "ListenSocket.h"
#include "AcceptSocket.h"  //添加 CAcceptSocket 类头文件
#include "SServerDlg.h"    //添加的主对话框类头文件
```

```
.......
void CListenSocket::OnAccept(int nErrorCode)
{
    // TODO: Add your specialized code here and/or call the base class
    CAcceptSocket *pSocket=new CAcceptSocket;
    CSServerDlg* pDlg=(CSServerDlg*)AfxGetMainWnd(); //得到主框口(对话框)指针
    if(Accept(*pSocket))
    {
        pDlg->m_pAcceptList.AddTail(pSocket);
        CString strAddr,strAddr1;
        UINT unPort,unPort1;
//得到远程IP地址和端口号
        pSocket->GetPeerName(strAddr1,unPort1);
        pSocket->GetSockName(strAddr,unPort);
//得到本地IP地址和端口号
pDlg->m_strNetMsg.Format("本地IP%s端口%d连接上远程客户IP%s端口%d",
strAddr,unPort,strAddr1,unPort1);
        pDlg->UpdateMsgData();
    }
    else
    {
        delete pSocket;
    }
    CSocket::OnAccept(nErrorCode);
}
```

CAcceptSocket 类响应接收事件，调用 OnReveive()函数接收数据，并把接收到的数据显示在主框口，向客户端回传信息，同时，将接收到的信息从打开的串口传送出去。

AcceptSocket.cpp 文件的编写如下：

```
// AcceptSocket.cpp : implementation file
#include "stdafx.h"
#include "SServer.h"
#include "AcceptSocket.h"
#include "SServerDlg.h"   //添加的主对话框类头文件

.......

void CAcceptSocket::OnReceive(int nErrorCode)
{
    // TODO: Add your specialized code here and/or call the base class
    char chMsg[5120], chMsgTemp[1024];
    UINT unRXCharNum;   //每次读取的字符数
    BOOL bEndFlag=0;  //接收完毕标志
    strcpy(chMsg,"");
    do
    {
        strcpy(chMsgTemp,"");
        unRXCharNum=Receive(chMsgTemp,1000);
        if(unRXCharNum>1000||unRXCharNum<=0)
        {
            AfxMessageBox("接收数据中出错",MB_OK);
            return;
        }
        else if(unRXCharNum<1000 && unRXCharNum>0)
        {
        bEndFlag=1;
        }
```

```
        chMsgTemp[unRXCharNum]=0; //加上字符串结束位
        strcat(chMsg,chMsgTemp);
    }while(bEndFlag==0);
    CSServerDlg* pDlg=(CSServerDlg*)AfxGetMainWnd(); //得到主框口(对话框)指针
    pDlg->m_strNetMsg.Format("接收到: %s",chMsg);
    pDlg->UpdateMsgData();   //在主对话的接收编辑框中显示网络端口接收到的数据
    CString strtemp;
    strtemp.Format("%s",chMsg);
    pDlg->SerialSendData(strtemp);
    strtemp="服务器已收到" + strtemp; //组成回传信息
    Send(strtemp,strtemp.GetLength(),0); //发送回传信息
    CSocket::OnReceive(nErrorCode);
}
```

4. 编写主对话框类代码

首先，我们要把串口发送功能加入进去，还是用最省事的 CserialPort 类，把类文件 SerialPort.h 和 SerialPort.cpp 复制到工程所在文件夹中，然后单击 VC 菜单 Project->Add to Project->Files...，再在打开的文件选择对话框中选择上 SerialPort.h 和 SerialPort.cpp，单击 OK，就把类文件加入了当前工程，如图 9.4.2.2 所示。

接着为对话框的“启动网络服务”、“停止网络服务”、“打开串口” 和“关闭串口”按钮分别添加单击响应函数 OnButtonStart()、OnButtonStop()、OnButtonOpen()和 OnButtonClose()函数。然后为复选框添加单击响应函数 OnCheckSenddata()，为了供外面调用完成串口发送，还要添加公共函数 SerialSendData(CString strSendData)。

图 9.4.6　服务器程序对话框界面设置情况

SServerDlg.h 的编写如下：

```
// SServerDlg.h : header file
#include "SerialPort.h"    //添加CSerialPort类头文件
#include "ListenSocket.h"  //添加CListenSocket类头文件
/////////////////////////////////////////////////////////////////////////////
```

```
// CSServerDlg dialog
class CSServerDlg : public CDialog
{
// Construction
public:
    void SerialSendData(CString strSendData);
    void UpdateMsgData(); //用于更新网络接收与发送信息
    CListenSocket* m_pListenSocket;
    BOOL m_bListened; //用于标志服务器是否处于开启网络服务功能
    CPtrList m_pAcceptList; //用于保存接收 socket 的队列
    CString m_strNetMsg; //用于传递端口号和服务的 IP 地址
    BOOL m_bSerialPortOpened; //用于标记串口是否打开
    CSerialPort m_SerialPort; //定义 CSerialPort 类对象
    CSServerDlg(CWnd* pParent = NULL);    // standard constructor
// Dialog Data
    //{{AFX_DATA(CSServerDlg)
    enum { IDD = IDD_SSERVER_DIALOG };
    CButton  m_ctrlCheckSendData;
    CComboBox m_ctrlComboComPort;
    UINT m_unEditPortNO;
    CString  m_strEditNetMsg;
    CString  m_strEditComMsg;
    //}}AFX_DATA
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CSServerDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    HICON m_hIcon;

    // Generated message map functions
    //{{AFX_MSG(CSServerDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg void OnButtonStart();
    afx_msg void OnButtonStop();
    afx_msg void OnButtonOpen();
    afx_msg void OnButtonClose();
    afx_msg void OnCheckSenddata();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
//{{AFX_INSERT_LOCATION}}
//Microsoft Visual C++ will insert additional declarations immediately before the previous
line.
#endif
// !defined(AFX_SSERVERDLG_H__BFEB8166_42F6_11D8_870F_00E04C3F78CA__INCLUDED_)
```

SServerDlg.cpp 文件的编写如下：

```
// SServerDlg.cpp : implementation file
.........
CSServerDlg::CSServerDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CSServerDlg::IDD, pParent)
{
```

```
    //{{AFX_DATA_INIT(CSServerDlg)
    m_unEditPortNO = 5050;
    m_strEditNetMsg = _T("");
    m_strEditComMsg = _T("");
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_pAcceptList.RemoveAll(); //初始化接收 socket 队列
    m_bListened=FALSE; //没有开启网络服务
    m_bSerialPortOpened=FALSE; //没有打开串口
}

void CSServerDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CSServerDlg)
    DDX_Control(pDX, IDC_CHECK_SENDDATA, m_ctrlCheckSendData);
    DDX_Control(pDX, IDC_COMBO_COMPORT, m_ctrlComboComPort);
    DDX_Text(pDX, IDC_EDIT_PORTNO, m_unEditPortNO);
    DDX_Text(pDX, IDC_EDIT_NETMSG, m_strEditNetMsg);
    DDX_Text(pDX, IDC_EDIT_COMMSG, m_strEditComMsg);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CSServerDlg, CDialog)
    //{{AFX_MSG_MAP(CSServerDlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_BUTTON_START, OnButtonStart)
    ON_BN_CLICKED(IDC_BUTTON_STOP, OnButtonStop)
    ON_BN_CLICKED(IDC_BUTTON_OPEN, OnButtonOpen)
    ON_BN_CLICKED(IDC_BUTTON_CLOSE, OnButtonClose)
    ON_BN_CLICKED(IDC_CHECK_SENDDATA, OnCheckSenddata)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CSServerDlg message handlers

BOOL CSServerDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    ........
    // TODO: Add extra initialization here
    GetDlgItem(IDC_BUTTON_START)->EnableWindow(!m_bListened);
    GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(m_bListened);
    GetDlgItem(IDC_BUTTON_OPEN)->EnableWindow(!m_bSerialPortOpened);
    GetDlgItem(IDC_BUTTON_CLOSE)->EnableWindow(m_bSerialPortOpened);
    m_ctrlComboComPort.SetCurSel(0); //初始选择串口 1
    m_ctrlCheckSendData.SetCheck(0); //没有打开串口之前不选中
    return TRUE; // return TRUE unless you set the focus to a control
}
........
//开启网络服务
void CSServerDlg::OnButtonStart()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    m_pListenSocket=new CListenSocket(); //新建监听 Socket
```

```
    if(m_pListenSocket->Create(m_unEditPortNO))
    {
        if(!m_pListenSocket->Listen(10))
        {
            AfxMessageBox("设置监听 Socket 失败",MB_ICONINFORMATION);
            delete m_pListenSocket; //释放指针空间
        }
        else
        {
            m_bListened=TRUE;
        }
    }
    else
    {
        AfxMessageBox("创建监听 Socket 失败",MB_ICONINFORMATION);
        delete m_pListenSocket; //释放指针空间
    }
    GetDlgItem(IDC_BUTTON_START)->EnableWindow(!m_bListened);
    GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(m_bListened);
}

//关闭网络服务
void CSServerDlg::OnButtonStop()
{
    // TODO: Add your control notification handler code here
    if(!m_bListened) return;
    if(m_pListenSocket == NULL) return;
    m_pListenSocket->ShutDown(2);
    Sleep(50);
    m_pListenSocket->Close();
    m_pListenSocket = NULL;
    m_bListened = FALSE;
    GetDlgItem(IDC_BUTTON_START)->EnableWindow(!m_bListened);
    GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(m_bListened);
}

void CSServerDlg::UpdateMsgData()
{
    m_strEditNetMsg=m_strNetMsg;
    UpdateData(FALSE);   //更新网络通信信息
}

//打开串口
void CSServerDlg::OnButtonOpen()
{
    // TODO: Add your control notification handler code here
    int nPort=m_ctrlComboComPort.GetCurSel()+1; //得到串口号
    if(m_SerialPort.InitPort(this,nPort,9600,'N',8,1,EV_RXFLAG| EV_RXCHAR,512))
    {
        m_SerialPort.StartMonitoring();
        m_bSerialPortOpened=TRUE;
    }
    else
    {
        AfxMessageBox("没有发现此串口或被占用");
        m_bSerialPortOpened=FALSE;
    }
    GetDlgItem(IDC_BUTTON_OPEN)->EnableWindow(!m_bSerialPortOpened);
    GetDlgItem(IDC_BUTTON_CLOSE)->EnableWindow(m_bSerialPortOpened);
}
```

```
//关闭串口
void CSServerDlg::OnButtonClose()
{
    // TODO: Add your control notification handler code here
    if(m_ctrlCheckSendData.GetCheck())
    {
        AfxMessageBox("请先关掉串口发送功能");
        return;
    }
    m_SerialPort.ClosePort();//关闭串口
  m_bSerialPortOpened=FALSE;
    GetDlgItem(IDC_BUTTON_OPEN)->EnableWindow(!m_bSerialPortOpened);
    GetDlgItem(IDC_BUTTON_CLOSE)->EnableWindow(m_bSerialPortOpened);
}

//检查串口是否打开，否则不能通过串口发送数据
void CSServerDlg::OnCheckSendddata()
{
    // TODO: Add your control notification handler code here
    if(m_ctrlCheckSendData.GetCheck())
    {
        if(!m_bSerialPortOpened)
        {
            AfxMessageBox("串口没有打开，请先打开串口");
            m_ctrlCheckSendData.SetCheck(0);
            return;
        }
    }
}

//通过串口发送数据
void CSServerDlg::SerialSendData(CString strSendData)
{
    if(!m_bSerialPortOpened) return;
    if(!m_ctrlCheckSendData.GetCheck()) return;
    m_SerialPort.WriteToPort((LPCTSTR)strSendData);//发送数据
    m_strEditComMsg.Format("向COM%d发送: %s",
        m_ctrlComboComPort.GetCurSel()+1,strSendData);
    UpdateData(FALSE);
}
```

现在，服务程序与客户端程序都编写完了，可以测试程序了，装有网卡的计算机如果配置有 2 个串口，就可以在一台计算机上完成测试，当然，联网的两台计算机更不在话下了。下面是在同一台计算机上完成测试的情况，该计算机的 IP 地址为 10.1.37.185，服务端口设在 5050，步骤如下：

- 计算机的两个串口用串口线相连，开启计算机后，打开客户端应用程序 SClient.exe，打开服务器应用程序 SServer.exe，打开串口调试助手，并设置在 COM2；
- 单击服务器应用程序的“开启网络服务”，然后单击客户端应用程序的“连接”，此时服务器程序的“接收发送信息”框内显示“本地 IP10.1.37.185 端口 5050 连接上远程客户 IP10.1.37.185 端口 1484”，单击服务器程序的“打开串口”，并选中“通过串口发送”复选框；
- 在客户端应用程序的“要发送的信息”框内填上“123456789ABCDEFG”(因串口类不支持汉字，不要填写汉字)，单击“向服务器发送”。运行情况如图 9.4.7 所示。

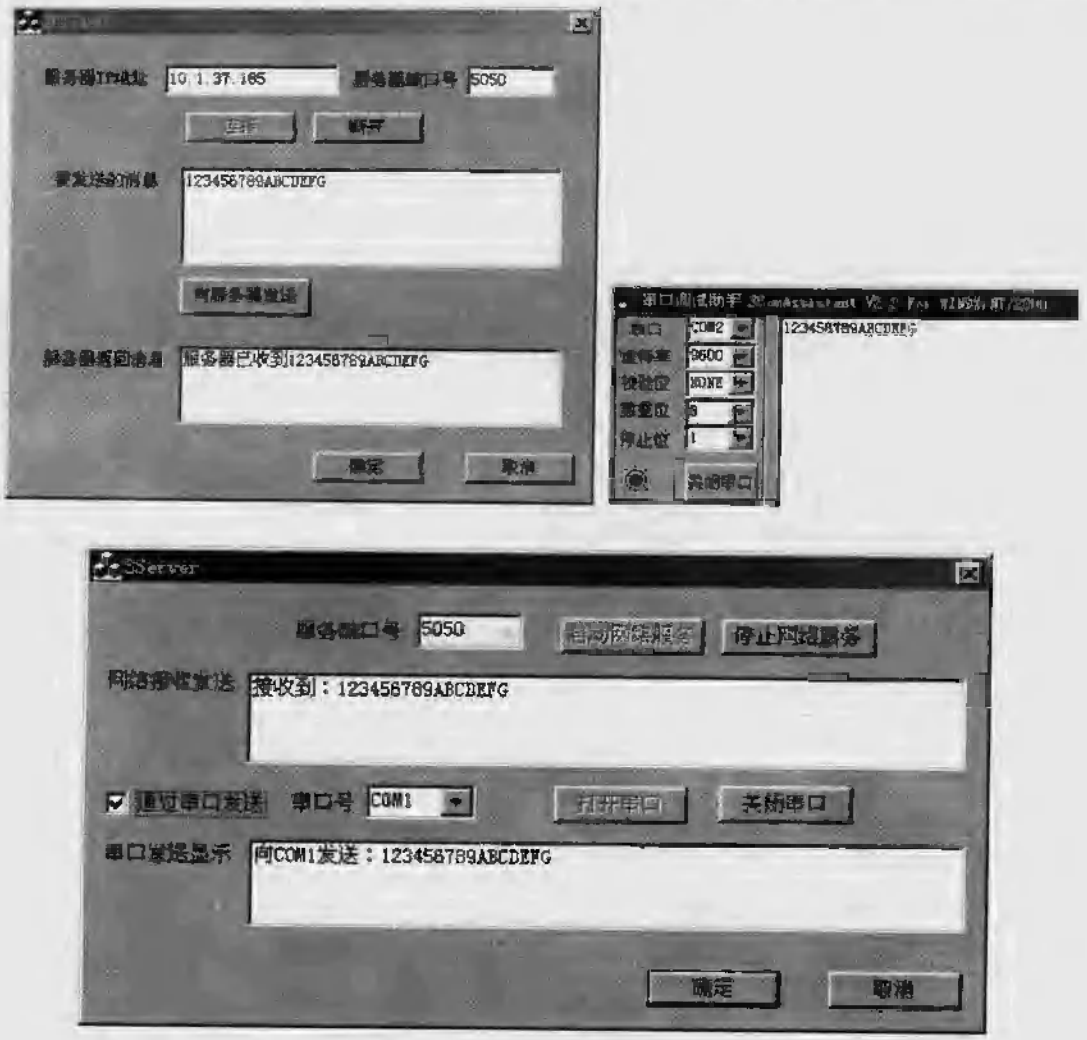


图 9.4.7　程序运行情况

9.5　在已经编好的串口通信程序中加入网络通信功能

随着网络通信的快速发展，传统的串口通信程序（还有其他功能的程序）需要加入网络通信的功能，因此，这个问题不断有编程的朋友提出来。我们这里只就 VC 6.0 程序如何加入基于 WinSocket 的网络通信功能做一简单的说明。其他类型的网络通信应用也可类似地对程序进行改造。

一般而言，有三种方法加入网络通信部分的程序代码：一是参照 MFC AppWizard(exe) 如何创建 WinSockets 程序对已有的程序进行改造；二是自己利用 Windows Sockets API 进行编程；三是利用一些第三方编写的网络通信类，这些类有的很成熟，经过很多人试用，在编写程序时很方便。

9.5.1 参照 MFC AppWizard 创建 WinSockets 程序

在 VC 6.0 集成开发环境中，当新创建工程时，如果是编写 MFC AppWizard(exe)应用程序，选中 Windows Sockets 后，MFC AppWizard 就会自动在建立的应用程序框架中加入 WinSocket 初始化代码。已经编写好的程序是不具备这一功能的，需要我们手工加入网络初始化代码。可以参照以下步骤进行。

1. 加入 WinSocket 初始化头文件

VC 中对 WinSocket 初始化的头文件为 afxsock.h，一般在 stdafx.h 中进行定义，stdafx.h 是对系统标准头文件进行定义的文件，如果在搭建框架时选中了 Windows Sockets 后，则在 stdafx.h 加入 afxsock.h：

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

........
#define VC_EXTRALEAN        // Exclude rarely-used stuff from Windows headers

#include <afxwin.h>     // MFC core and standard components
#include <afxext.h>     // MFC extensions
#include <afxdisp.h>    // MFC Automation classes
#include <afxdtctl.h>   // MFC support for Internet Explorer 4 Common Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>         // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT
#include <afxsock.h>        // MFC socket extensions    要添加的代码

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.
#endif
// !defined(AFX_STDAFX_H__05807408_3669_11D8_8707_00E04C3F78CA__INCLUDED_)
```

因此，我们也只需在 stdafx.h 加入 afxsock.h。

2. 在 App 实现文件中加入 WinSocket 初始化代码

App 实现文件名和创建的工程名相同，假如我们创建了名为 SClient 的工程，那么 App 实现文件名为 SClient.cpp，在这个文件的 InitInstance()函数中加入以下代码：

```
BOOL CSClientApp::InitInstance()
{
  //以下需要添加的代码
     if (!AfxSocketInit())
     {
          AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
          return FALSE;
     }
//添加代码结束
........
```

3. 添加 Csocket 对象或相应的类

这可参考第 9.4 节来完成。

9.5.2　利用 Windows Sockets API 和第三方提供的类进行编程

这两种方法有一个共同的特点，在编译的过程中，必须在 Project->settings 的 Link 标签 Object/library modules 中添加上 wsock32.lib 或者 ws2_32.lib 库。

API 编程具有很大的灵活性，但需要编程者对底层很熟悉，并且要掌握较多的相关知识，程序调试起来也很麻烦，但要真正了解并掌握网络通信过程，这种尝试是值得的。读者可以仔细研读相关网络编程的书籍，对其进行全面了解，在此不做详细介绍。

如果刚接触网络编程，直接用一些好用的封装类，也是一种很好的选择，多用别人的东西，既可以加快入门的进程，提高学习的兴趣，又为深入学习提供一些参考样例。这里我们介绍一个 Socket 封装类 CWSocket 类，包括了服务端、客户端、超时控制等功能，使用较方便。我们不做详细说明，具体编程方法参见本节源代码，其中 Wsocket.cpp、Wsocket.h 两个文件是类文件，在例程中，服务器程序和客户程序做在同一程序中，可在同一台计算机或两台计算机上进行测试。

9.6　串口通信用于遥控操作简例

在某些应用场合，比如比赛的实时计分系统中需要现场实时地显示比赛成绩，此时可以通过串口通信方式实现由上位机操作遥控与显示屏相连的下位机。本例将通过一个十分简单的例子对这一编程方法进行说明。

上位机界面如图 9.6.1 所示，其中，编辑框由工作人员实时输入红蓝队的成绩；下位机界面如图 9.6.2 所示，它不需要人员操作，只是通过串口接收来自上位机的指令，实时地显示红蓝队的成绩。当上位机按“下一场”按钮，表示下一场开始，红蓝队双方置 0。

图　9.6.1

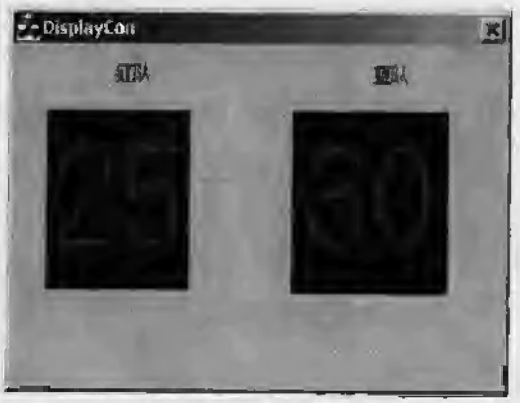


图　9.6.2

上位机程序介绍如下：

（1）建立一个基于对话框的工程，并命名为 RemoteCon。

（2）按上位机界面制作对话框，并在对话框中加入串口通信控件对象，命名成员变量 m_ctrlComm；同时对红蓝队得分编辑框增加 Cstring 类型的成员变量 m_redscore、m_bluescore。

（3）对红蓝队得分框下部的“发送”按钮分别映射对应的函数 OnButtonRedsend()、OnButtonBluesend()；对“下一场”按钮映射函数 OnButtonnext()。

（4）初始化对话框程序。为简单起见，直接在 CRemoteConDlg::OnInitDialog()函数中添加如下内容，打开串口 1，并设置波特率 9600，无校验，8 个数据位，1 个停止位。

```
// TODO: Add extra initialization here
    m_ctrlComm.SetCommPort(1); //选择 com1
if( !m_ctrlComm.GetPortOpen())
m_ctrlComm.SetPortOpen(TRUE);//打开串口
else
AfxMessageBox("cannot open serial port");

m_ctrlComm.SetSettings("9600,n,8,1"); //波特率 9600，无校验，8 个数据位，1 个停止位

m_ctrlComm.SetInputMode(1); //1: 表示以二进制方式检取数据
m_ctrlComm.SetRThreshold(1);
//参数 1 表示每当串口接收缓冲区中有多于或等于 1 个字符时将引发一个接收数据的 OnComm 事件
m_ctrlComm.SetInputLen(0); //设置当前接收区数据长度为 0
m_ctrlComm.GetInput();
```

（5）编写发送程序，发送时红队得分添加标记“r”，蓝队得分标记为“b”，而下一场标记为“n”，具体内容如下：

```
void CRemoteConDlg::OnButtonRedsend()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    m_redscore='r'+m_redscore;
    m_ctrlComm.SetOutput(COleVariant(m_redscore));

}

void CRemoteConDlg::OnButtonBluesend()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    m_bluescore='b'+m_bluescore;
    m_ctrlComm.SetOutput(COleVariant(m_bluescore));
}

void CRemoteConDlg::OnButtonnext()
{
    // TODO: Add your control notification handler code here
     m_redscore="0";
        m_bluescore="0";
        UpdateData(FALSE);
        m_ctrlComm.SetOutput(COleVariant("n"));
}
```

下位机程序介绍如下：

（1）建立一个基于对话框的工程，并命名为 DisplayCon。

（2）按下位机界面制作对话框，包括两个 edit 控件 IDC_EDIT_REDSCORE、IDC_EDIT_BLUESCORE 和两个 static text 控件。（注意，图 9.6.2 中 edit 控件框中字体大小是通过类修改的，下面将要讲述）；在对话框中加入串口通信控件对象，命名成员变量 m_ctrlComm；同时对红蓝队得分编辑框增加 Cstring 类型的成员变量 m_redscore、m_bluescore。

（3）为串口控件对象添加消息处理函数 OnComm()。

（4）初始化对话框程序。在 CRemoteConDlg::OnInitDialog()函数中添加如下内容，打开串口 2，并设置波特率 9600，无校验，8 个数据位，1 个停止位。

```
// TODO: Add extra initialization here
    m_ctrlComm.SetCommPort(2); //选择 com1
if( !m_ctrlComm.GetPortOpen())
m_ctrlComm.SetPortOpen(TRUE);//打开串口
else
AfxMessageBox("cannot open serial port");

m_ctrlComm.SetSettings("9600,n,8,1"); //波特率 9600，无校验，8 个数据位，1 个停止位

m_ctrlComm.SetInputMode(1); //1: 表示以二进制方式检取数据
m_ctrlComm.SetRThreshold(1);
//参数 1 表示每当串口接收缓冲区中有多于或等于 1 个字符时将引发一个接收数据的 OnComm 事件
m_ctrlComm.SetInputLen(0); //设置当前接收区数据长度为 0
m_ctrlComm.GetInput();
```

（5）编写消息处理函数，具体内容如下：

```
void CDisplayConDlg::OnComm()
{
    // TODO: Add your control notification handler code here
    VARIANT variant_inp;
  COleSafeArray safearray_inp;//COleSafeArray 操作任意类型任意维数的数组
  LONG len,k;
  BYTE rxdata[2048]; //设置 BYTE 数组 An 8-bit integerthat is not signed.
  CString strtemp,m_strRXData1;//,m_strRXData2;

 if(m_ctrlComm.GetCommEvent()==2) //事件值为 2 表示接收缓冲区内有字符
 {

      variant_inp=m_ctrlComm.GetInput(); //读缓冲区
    safearray_inp=variant_inp; //VARIANT 型变量转换为 ColeSafeArray 型变量
     len=safearray_inp.GetOneDimSize(); //得到有效数据长度，返回一维 COleSafeArray
                                        //对象中的元素数
     for(k=0;k<len;k++)
      safearray_inp.GetElement(&k,rxdata+k);//转换为 BYTE 型数组

        m_strRXData1.Format("%c%c%c",rxdata[0],rxdata[1],rxdata[2]);
      char* temp=(char*)((LPCTSTR)m_strRXData1);
   char tbuf[10];
   tbuf[0]=temp[1]; tbuf[1]=temp[2]; tbuf[2]=0;
      if(temp[0]=='r')
          m_redscore=tbuf;
      if(temp[0]=='b')
          m_bluescore=tbuf;
   if(temp[0]=='n')
      {
          m_bluescore="0";
     m_redscore="0";
```

```
            }
        UpdateData(FALSE); //更新编辑框内容
      }
  }
```

（6）修改edit控件框中字体大小。

首先，在工程中添加如下三个文件：EditEx.cxx、EditEx.hxx、LogFont.hxx（可从本书附带的光盘中找到），添加完成后即可发现工程中多了三个类：CeditEx 、CeditMask 、ClogFont。

然后，在 DisplayConDlg.h 文件中添加如下成员变量：

```
public:
CEditEx  m_edit;
CEditEx  m_edit2;
```

注意，别忘了引用如下头文件：

```
#include "editex.hxx"
```

最后，在CDisplayConDlg::OnInitDialog()函数中添加如下内容，以给定红蓝队得分字体大小及其背景着色。

```
COLORREF tx = RGB( 255,  0, 0); //文本着色
   COLORREF bk = RGB( 0,  0, 0);//背景色

      m_edit.SubclassDlgItem( IDC_EDIT_REDSCORE, this );
   m_edit.bkColor( bk );
   m_edit.textColor( tx );
   m_edit.setFont( -80 );

      m_edit2.SubclassDlgItem( IDC_EDIT_BLUESCORE, this );
   m_edit2.bkColor( bk );
   m_edit2.textColor( tx );
   m_edit2.setFont( -80 );
```

第 10 章　计算机串口与其他设备通信编程实例

[内容提要]

前面的章节对 VC 6.0 串口通信程序相关内容做了比较详细的介绍。作为串口通信编程实践及工程应用的补充，本章对计算机串口与其他设备通信编程进行了介绍。10.1 节介绍了如何通过串口收发短消息，通过程序实例说明了用串口连接 GSM 手机时短消息的接收与发送。10.2 节介绍在嵌入式系统中广为应用的 Rabbit 2000 系统及其编程语言，通过一个具体的工程实例介绍了其具体用法，并且对语法知识做了解释。本章的后两节则分别对计算机与 PLC 通信、MATLAB 环境下串口编程进行了介绍。

10.1 通过串口收发短消息

本节内容原作为科脑工作室（http://www.kernelstudio.com）的bhw98，示例程序文件为：SmsTest.rar，可在本书所附光盘中第 10 章程序中找到（注意，文中代码以示例程序为准，示例程序中的代码根据实际应用情况做了修正）。

10.1.1　SMS 编码规范及编码与解码例程

随着移动通信的迅速发展，用串口接收手机短消息也在很多场合得到应用，那么在串口应用程序中，如何编程实现 GSM 手机发送和接收短消息？如何利用 SMS 进行数据通信？

首先，我们要对由 ESTI 制定的 SMS 规范有所了解。与我们讨论的短消息收发有关的规范主要包括 GSM 03.38、GSM 03.40 和 GSM 07.05，前两者着重描述 SMS 的技术实现(含编码方式)，后者则规定了 SMS 的 DTE-DCE 接口标准(AT 命令集)。

一共有三种方式来发送和接收 SMS 信息：Block Mode, Text Mode 和 PDU Mode。Block Mode 已是昔日黄花，目前很少用了。Text Mode 是纯文本方式，可使用不同的字符集，主要用于欧美地区，从技术上说也可用于发送中文短消息，但国内手机基本上不支持。PDU Mode 被所有手机支持，可以使用任何字符集，这也是手机默认的编码方式。Text Mode 比较简单，而且不适合做自定义数据传输，这里就不讨论了。下面介绍的内容，是在 PDU Mode 下发送和接收短消息的实现方法。

PDU 串表面上是一串 ASCII 码，由 0～9、A～F 这些数字和字母组成。它们是 8 位字节的十六进制数，或者 BCD 码十进制数。PDU 串不仅包含可显示的消息本身，还包含很多其他信息，如 SMS 服务中心号码、目标号码、回复号码、编码方式和服务时间等。发送和接收的 PDU 串，结构是不完全相同的。下面用两个实例说明 PDU 串的结构和编排方式。

例 1 发送：SMSC 号码是+8613800250500，对方号码是 13851872468，消息内容是“Hello!”。从手机发出的 PDU 串可以是：

```
08 91 68 31 08 20 05 05 F0 11 00 0D 91 68 31 58 81 27 64 F8 00 00 06 C8 32 9B FD 0E 01
```

对照规范，以上数据信息的含义见表 10-1-1-1。

表 10-1-1-1　PDU 串数据信息说明

分段	含义	说明
08	SMSC 地址信息的长度	共 8 个 8 位字节(包括 91)
91	SMSC 地址格式(TON/NPI)	用国际格式号码(在前面加“+”)
68 31 08 20 05 05 F0	SMSC 地址	8613800250500，补“F”凑成偶数个
11	基本参数(TP-MTI/VFP)	发送，TP-VP 用相对格式
00	消息基准值(TP-MR)	0
0D	目标地址数字个数	共 13 个十进制数(不包括 91 和“F”)
91	目标地址格式(TON/NPI)	用国际格式号码(在前面加“+”)
68 31 58 81 27 64 F8	目标地址(TP-DA)	8613851872468，补“F”凑成偶数个
00	协议标识(TP-PID)	是普通 GSM 类型，点到点方式
00	用户信息编码方式(TP-DCS)	7-bit 编码
00	有效期(TP-VP)	5 分钟
06	用户信息长度(TP-UDL)	实际长度 6 个字节
C8 32 9B FD 0E 01	用户信息(TP-UD)	“Hello!”

例 2 接收：SMSC 号码是+8613800250500，对方号码是 13851872468，消息内容是“你好!”。手机接收到的 PDU 串可以是：

```
08 91 68 31 08 20 05 05 F0 84 0D 91 68 31 58 81 27 64 F8 00 08 30 30 21 8C 63 54 80 06
4F 60 59 7D 00 21
```

对照规范，PDU 串数据信息说明见表 10-1-1-2。

表 10-1-1-2　PDU 串数据信息说明

分段	含义	说明
08	地址信息的长度	共 8 个 8 位字节(包括 91)
91	SMSC 地址格式(TON/NPI)	用国际格式号码(在前面加“+”)
68 31 08 20 05 05 F0	SMSC 地址	8613800250500，补“F”凑成偶数个
84	基本参数(TP-MTI/MMS/RP)	接收，无更多消息，有回复地址
0D	源地址数字个数	共 13 个十进制数(不包括 91 和“F”)
91	源地址格式(TON/NPI)	用国际格式号码(在前面加“+”)
68 31 58 81 27 64 F8	源地址(TP-OA)	8613851872468，补“F”凑成偶数个
00	协议标识(TP-PID)	是普通 GSM 类型，点到点方式

续表

分段	含义	说明
08	用户信息编码方式(TP-DCS)	UCS2 编码
30 30 21 80 63 54 80	时间戳(TP-SCTS)	2003-3-12 08:36:45　+8 时区
06	用户信息长度(TP-UDL)	实际长度 6 个字节
4F 60 59 7D 00 21	用户信息(TP-UD)	“你好!”

[illegible]号码和时间的表示方法不是按正常顺序来的，而且要以“F”将奇数补成偶数。

上面两例中已经出现了 **7-bit** 和 **UCS2** 编码，下面详细介绍这些编码方式。

在 PDU Mode 中，可以采用三种编码方式来对发送的内容进行编码，它们是 7-bit、8-bit 和 UCS2 编码。7-bit 编码用于发送普通的 ASCII 字符，它将一串 7-bit 的字符(最高位为 0)编码成 8-bit 的数据，每 8 个字符可“压缩”成 7 个；8-bit 编码通常用于发送数据消息，比如图片和铃声等；而 UCS2 编码用于发送 Unicode 字符。在这三种编码方式下，PDU 串的用户信息(TP-UD)段最大容量(可以发送的短消息的最大字符数)分别是 160、140 和 70。这里，将一个英文字母、一个汉字和一个数据字节都视为一个字符。

需要注意的是，PDU 串的用户信息长度(TP-UDL)，在各种编码方式下意义有所不同。7-bit 编码时，指原始短消息的字符个数，而不是编码后的字节数。8-bit 编码时，就是字节数。UCS2 编码时，也是字节数，等于原始短消息的字符数的两倍。如果用户信息(TP-UD)中存在一个头(基本参数的 TP-UDHI 为 1)，在所有编码方式下，用户信息长度(TP-UDL)都等于头长度与编码后字节数之和。如果采用 GSM 03.42 所建议的压缩算法(TP-DCS 的高 3 位为 001)，则该长度也是压缩编码后字节数或头长度与压缩编码后字节数之和。

1．7-bit 编码

下面以一个具体的例子说明 7-bit 编码的过程。我们对英文短信“Hello!”进行编码（图 10.1.1.1 是编码过程）。

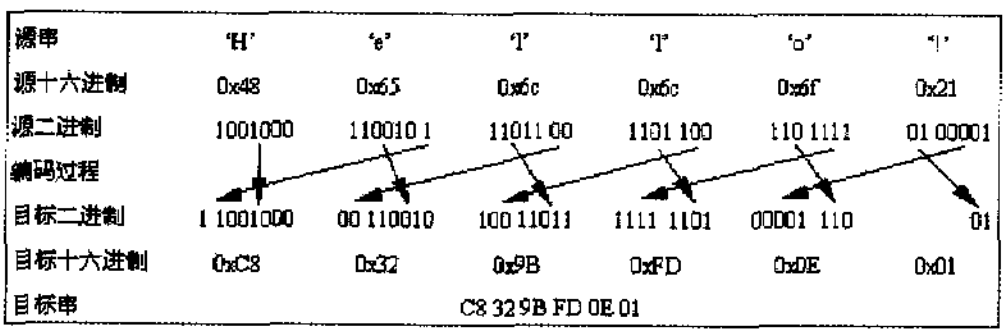


图 10.1.1.1　“Hello!”的编码过程

将源串每 8 个字符分为一组(这个例子中不满 8 个)进行编码，在组内字符间压缩，但每组之间是没有什么联系的。

实现 7-bit 编码和解码的算法程序如下。

编码算法：

```
// 7-bit 编码
// pSrc: 源字符串指针
// pDst: 目标编码串指针
// nSrcLength: 源字符串长度
// 返回: 目标编码串长度
int gsmEncode7bit(const char* pSrc, unsigned char* pDst, int nSrcLength)
{
    int nSrc;        // 源字符串的计数值
    int nDst;        // 目标编码串的计数值
    int nChar;       // 当前正在处理的组内字符字节的序号，范围是 0～7
    unsigned char nLeft;    // 上一字节残余的数据

    // 计数值初始化
    nSrc = 0;
    nDst = 0;

    // 将源串每 8 个字节分为一组，压缩成 7 个字节
    // 循环该处理过程，直至源串被处理完
    // 如果分组不到 8 字节，也能正确处理
    while (nSrc < nSrcLength)
    {
        // 取源字符串的计数值的最低 3 位
        nChar = nSrc & 7;

        // 处理源串的每个字节
        if (nChar == 0)
        {
            // 组内第一个字节，只是保存起来，待处理下一个字节时使用
            nLeft = *pSrc;
        }
        else
        {
            // 组内其他字节，将其右边部分与残余数据相加，得到一个目标编码字节
            *pDst = (*pSrc << (8 - nChar)) | nLeft;
              // 将该字节剩下的左边部分，作为残余数据保存起来
            nLeft = *pSrc >> nChar;
            // 修改目标串的指针和计数值
            pDst++;
            nDst++;
        }
        // 修改源串的指针和计数值
        pSrc++;
        nSrc++;
    }

    // 返回目标串长度
    return nDst;
}
```

解码算法：

```
// 7-bit 解码
// pSrc: 源编码串指针
// pDst: 目标字符串指针
// nSrcLength: 源编码串长度
// 返回: 目标字符串长度
int gsmDecode7bit(const unsigned char* pSrc, char* pDst, int nSrcLength)
```

```
{
    int nSrc;        // 源字符串的计数值
    int nDst;        // 目标解码串的计数值
    int nByte;       // 当前正在处理的组内字节的序号，范围是0～6
    unsigned char nLeft;    // 上一字节残余的数据

    // 计数值初始化
    nSrc = 0;
    nDst = 0;

    // 组内字节序号和残余数据初始化
    nByte = 0;
    nLeft = 0;

    // 将源数据每7个字节分为一组，解压缩成8个字节
    // 循环该处理过程，直至源数据被处理完
    // 如果分组不到7字节，也能正确处理
    while (nSrc < nSrcLength)
    {
        // 将源字节右边部分与残余数据相加，去掉最高位，得到一个目标解码字节
        *pDst = ((*pSrc << nByte) | nLeft) & 0x7f;

        // 将该字节剩下的左边部分，作为残余数据保存起来
        nLeft = *pSrc >> (7 - nByte);

        // 修改目标串的指针和计数值
        pDst++;
        nDst++;

        // 修改字节计数值
        nByte++;

        // 到了一组的最后一个字节
        if (nByte == 7)
        {
            // 额外得到一个目标解码字节
            *pDst = nLeft;

            // 修改目标串的指针和计数值
            pDst++;
            nDst++;

            // 组内字节序号和残余数据初始化
            nByte = 0;
            nLeft = 0;
        }
        // 修改源串的指针和计数值
        pSrc++;
        nSrc++;
    }
    *pDst = 0;
    // 返回目标串长度
    return nDst;
}
```

需要指出的是，7-bit 的字符集与 ANSI 标准字符集不完全一致，在 0x20 以下也排布了一些可打印字符，但英文字母、阿拉伯数字和常用符号的位置两者是一样的。用上面介绍的算法收发纯英文短消息，一般情况应该是够用了。如果是法语、德语、西班牙语等，含有“à”、

“é”这一类字符，则要按上面编码的输出去查表，请参阅 GSM 03.38 的规定。

8-bit 编码其实没有规定什么具体的算法，这里不再介绍。

2. UCS2 编码

UCS2 编码是将每个字符(1～2 个字节)按照 ISO/IEC10646 的规定，转变为 16 位的 Unicode 宽字符。在 Windows 系统中，特别是在 2000/XP 中，可以简单地调用 API 函数实现编码和解码。如果没有系统的支持，比如用单片机控制手机模块收发短消息，则只好用查表法解决了。

Windows 环境下，实现 UCS2 编码和解码的算法如下。

UCS2 编码：

```
// UCS2 编码
// pSrc: 源字符串指针
// pDst: 目标编码串指针
// nSrcLength: 源字符串长度
// 返回:目标编码串长度
int gsmEncodeUcs2(const char* pSrc, unsigned char* pDst, int nSrcLength)
{
   int      nDstLength; // UNICODE 宽字符数目
   WCHAR     wchar[128]; // UNICODE 串缓冲区

   //字符串-->UNICODE 串
   nDstLength = MultiByteToWideChar(CP_ACP, 0, pSrc, nSrcLength, wchar, 128);

   // 高低字节对调，输出
   for (int i = 0; i < nDstLength; i++)
   {
      // 先输出高位字节
      *pDst++ = wchar[i] >> 8;

      // 后输出低位字节
      *pDst++ = wchar[i] & 0xff;
   }

   // 返回目标编码串长度
   return nDstLength * 2;
}
```

UCS2 解码：

```
// UCS2 解码
// pSrc: 源编码串指针
// pDst: 目标字符串指针
// nSrcLength: 源编码串长度
// 返回: 目标字符串长度
int gsmDecodeUcs2(const unsigned char* pSrc, char* pDst, int nSrcLength)
{
   int nDstLength;       // UNICODE 宽字符数目
   WCHAR wchar[128];     // UNICODE 串缓冲区

   // 高低字节对调，拼成 UNICODE
   for (int i = 0; i < nSrcLength/2; i++)
   {
      // 先高位字节
```

```
            wchar[i] = *pSrc++ << 8;

            // 后低位字节
            wchar[i]  = *pSrc++;

        }

        //UNICODE 串-->字符串
        nDstLength=WideCharToMultiByte(CP_ACP, 0, wchar, nSrcLength/2, pDst, 160, NULL, NULL);

        // 返回目标字符串长度
        return nDstLength;
    }
```

用以上编码和解码模块，还不能将短消息字符串编码为 PDU 串需要的格式，也不能直接将 PDU 串中的用户信息解码为短消息字符串，因为还差一个在可打印字符串和字节数据之间相互转换的环节。可以循环调用 sscanf 和 sprintf 函数实现这种变换。下面提供不用这些函数的算法，它们也适用于单片机、DSP 编程环境。

可打印字符串转换为字节数据：

```
// 可打印字符串转换为字节数据
// 如: "C8329BFD0E01" --> {0xC8, 0x32, 0x9B, 0xFD, 0x0E, 0x01}
// pSrc: 源字符串指针
// pDst: 目标数据指针
// nSrcLength: 源字符串长度
// 返回: 目标数据长度
int gsmString2Bytes(const char* pSrc, unsigned char* pDst, int nSrcLength)
{
    for (int i = 0; i < nSrcLength; i++)
    {
        // 输出高 4 位
        if (*pSrc >= '0' && *pSrc <= '9')
        {
            *pDst = (*pSrc - '0') << 4;
        }
        else
        {
            *pDst = (*pSrc - 'A' + 10) << 4;
        }

        pSrc++;

        // 输出低 4 位
        if (*pSrc >= '0' && *pSrc <= '9')
        {
            *pDst |= *pSrc - '0';
        }
        else
        {
            *pDst |= *pSrc - 'A' + 10;
        }

        pSrc++;
        pDst++;
    }

    // 返回目标数据长度
    return nSrcLength / 2;
```

```
}
```

字节数据转换为可打印字符串：

```
// 字节数据转换为可打印字符串
// 如：{0xC8, 0x32, 0x9B, 0xFD, 0x0E, 0x01} --> "C8329BFD0E01"
// pSrc：源数据指针
// pDst：目标字符串指针
// nSrcLength：源数据长度
// 返回：目标字符串长度
int gsmBytes2String(const unsigned char* pSrc, char* pDst, int nSrcLength)
{
   const char tab[]="0123456789ABCDEF";   // 0x0~0xf 的字符查找表

   for (int i = 0; i < nSrcLength; i++)
   {
      // 输出高 4 位
      *pDst++ = tab[*pSrc >> 4];

      // 输出低 4 位
      *pDst++ = tab[*pSrc & 0x0f];

      pSrc++;
   }

   // 输出字符串加个结束符
   *pDst = '\0';

   // 返回目标字符串长度
   return nSrcLength * 2;
}
```

关于 GSM 03.42 中的压缩算法，至今还没有发现哪里用过，这里就不讨论了。有兴趣的读者，可深入研究一下。

10.1.2　AT 命令收发短消息实例

PDU 的核心编码方式已经清楚了，如何实现用 AT 命令收发短消息呢？

上面已经讨论了 7bit、8bit 和 UCS2 这几种 PDU 用户信息的编码方式，并且给出了实现代码，现在重点描述 PDU 全串的编码和解码过程，以及 GSM 07.05 的 AT 命令实现方法。这些是底层的核心代码，为了保证代码的可移植性，尽可能不用 MFC 的类，必要时用 ANSI C 标准库函数。

首先，定义如下常量和结构：

```
// 用户信息编码方式
#define GSM_7BIT      0
#define GSM_8BIT      4
#define GSM_UCS2      8

// 短消息参数结构，编码/解码共用
// 其中，字符串以'\0'结尾
typedef struct {
   char SCA[16];      // 短消息服务中心号码(SMSC 地址)
   char TPA[16];      // 目标号码或回复号码(TP-DA 或 TP-RA)
   char TP_PID;       // 用户信息协议标识(TP-PID)
```

```
    char TP_DCS;        // 用户信息编码方式(TP-DCS)
    char TP_SCTS[16];   // 服务时间戳字符串(TP_SCTS), 接收时用到
    char TP_UD[161];    // 原始用户信息(编码前或解码后的TP-UD)
    char index;         // 短消息序号, 在读取时用到
} SM_PARAM;
```

大家已经注意到，PDU 串中的号码和时间都是两两颠倒的字符串。利用下面两个函数可进行正反变换。

正常顺序的字符串转换为两两颠倒的字符串：

```
// 正常顺序的字符串转换为两两颠倒的字符串, 若长度为奇数, 补F凑成偶数
// 如: "8613851872468" --> "683158812764F8"
// pSrc: 源字符串指针
// pDst: 目标字符串指针
// nSrcLength: 源字符串长度
// 返回: 目标字符串长度
int gsmInvertNumbers(const char* pSrc, char* pDst, int nSrcLength)
{
    int nDstLength;   // 目标字符串长度
    char ch;          // 用于保存一个字符

    // 复制串长度
    nDstLength = nSrcLength;

    // 两两颠倒
    for (int i = 0; i < nSrcLength; i += 2)
    {
        ch = *pSrc++;        // 保存先出现的字符
        *pDst++ = *pSrc++;   // 复制后出现的字符
        *pDst++ = ch;        // 复制先出现的字符
    }

    // 源串长度是奇数吗?
    if (nSrcLength & 1)
    {
        *(pDst-2) = 'F';     // 补'F
        nDstLength++;        // 目标串长度加1
    }

    // 输出字符串加个结束符
    *pDst = '\0';

    // 返回目标字符串长度
    return nDstLength;
}
```

两两颠倒的字符串转换为正常顺序的字符串：

```
// 两两颠倒的字符串转换为正常顺序的字符串
// 如: "683158812764F8" --> "8613851872468"
// pSrc: 源字符串指针
// pDst: 目标字符串指针
// nSrcLength: 源字符串长度
// 返回: 目标字符串长度
int gsmSerializeNumbers(const char* pSrc, char* pDst, int nSrcLength)
{
```

```
    int nDstLength;   // 目标字符串长度
    char ch;          // 用于保存一个字符

    // 复制串长度
    nDstLength = nSrcLength;

    // 两两颠倒
    for (int i = 0; i < nSrcLength; i += 2)
    {
       ch = *pSrc++;        // 保存先出现的字符
       *pDst++ = *pSrc++;   // 复制后出现的字符
       *pDst++ = ch;        // 复制先出现的字符
    }

    // 最后的字符是'F'吗?
    if (*(pDst-1) == 'F')
    {
       pDst--;
       nDstLength--;        // 目标字符串长度减1
    }

    // 输出字符串加个结束符
    *pDst = '\0';

    // 返回目标字符串长度
    return nDstLength;
}
```

以下是 PDU 全串的编解码模块。为简化编程，有些字段用了固定值。

PDU 全串的编码：

```
// PDU编码，用于编制、发送短消息
// pSrc：源PDU参数指针
// pDst：目标PDU串指针
// 返回：目标PDU串长度
int gsmEncodePdu(const SM_PARAM* pSrc, char* pDst)
{
   int nLength;              // 内部用的串长度
   int nDstLength;           // 目标PDU串长度
   unsigned char buf[256];   // 内部用的缓冲区

   // SMSC地址信息段
   nLength = strlen(pSrc->SCA);    // SMSC地址字符串的长度
   buf[0] = (char)((nLength & 1) == 0 ? nLength : nLength + 1) / 2 + 1;
   // SMSC地址信息长度
   buf[1] = 0x91;        // 固定：用国际格式号码
   nDstLength = gsmBytes2String(buf, pDst, 2);        // 转换2个字节到目标PDU串
   nDstLength += gsmInvertNumbers(pSrc->SCA, &pDst[nDstLength], nLength);
   // 转换SMSC到目标PDU串

   // TPDU段基本参数、目标地址等
   nLength = strlen(pSrc->TPA);    // TP-DA地址字符串的长度
   buf[0] = 0x11;                  // 是发送短信(TP-MTI=01)，TP-VP用相对格式(TP-VPF=10)
   buf[1] = 0;                     // TP-MR=0
   buf[2] = (char)nLength;         // 目标地址数字个数(TP-DA地址字符串真实长度)
   buf[3] = 0x91;                  // 固定：用国际格式号码
   nDstLength += gsmBytes2String(buf, &pDst[nDstLength], 4);  // 转换4个字节到目标PDU串
```

```
    nDstLength += gsmInvertNumbers(pSrc->TPA, &pDst[nDstLength], nLength);
    // 转换 TP-DA 到目标 PDU 串

    // TPDU 段协议标识、编码方式、用户信息等
    nLength = strlen(pSrc->TP_UD);    // 用户信息字符串的长度
    buf[0] = pSrc->TP_PID;            // 协议标识(TP-PID)
    buf[1] = pSrc->TP_DCS;            // 用户信息编码方式(TP-DCS)
    buf[2] = 0;                       // 有效期(TP-VP)为 5 分钟
    if (pSrc->TP_DCS == GSM_7BIT)
    {
    // 7-bit 编码方式
       buf[3] = nLength;              // 编码前长度
       nLength = gsmEncode7bit(pSrc->TP_UD, &buf[4], nLength+1) + 4;
    // 转换 TP-DA 到目标 PDU 串
    }
    else if (pSrc->TP_DCS == GSM_UCS2)
    {
    // UCS2 编码方式
       buf[3] = gsmEncodeUcs2(pSrc->TP_UD, &buf[4], nLength);
    // 转换 TP-DA 到目标 PDU 串
       nLength = buf[3] + 4;       // nLength 等于该段数据长度
    }
    else
    {
    // 8-bit 编码方式
       buf[3] = gsmEncode8bit(pSrc->TP_UD, &buf[4], nLength);   // 转换 TP-DA 到目标 PDU 串
       nLength = buf[3] + 4;       // nLength 等于该段数据长度
    }
    nDstLength += gsmBytes2String(buf, &pDst[nDstLength], nLength); // 转换该段数据到目标
                                                                     //PDU 串

    // 返回目标字符串长度
    return nDstLength;
}
```

PDU 全串的解码：

```
// PDU 解码，用于接收、阅读短消息
// pSrc: 源 PDU 串指针
// pDst: 目标 PDU 参数指针
// 返回: 用户信息串长度
int gsmDecodePdu(const char* pSrc, SM_PARAM* pDst)
{
   int nDstLength;          // 目标 PDU 串长度
   unsigned char tmp;       // 内部用的临时字节变量
   unsigned char buf[256];  // 内部用的缓冲区

   // SMSC 地址信息段
   gsmString2Bytes(pSrc, &tmp, 2);    // 取长度
   tmp = (tmp - 1) * 2;    // SMSC 号码串长度
   pSrc += 4;              // 指针后移
   gsmSerializeNumbers(pSrc, pDst->SCA, tmp);    // 转换 SMSC 号码到目标 PDU 串
   pSrc += tmp;        // 指针后移

   // TPDU 段基本参数、回复地址等
   gsmString2Bytes(pSrc, &tmp, 2);    // 取基本参数
   pSrc += 2;        // 指针后移
```

```
        if (tmp & 0x80)
        {
            // 包含回复地址，取回复地址信息
            gsmString2Bytes(pSrc, &tmp, 2);    // 取长度
            if (tmp & 1) tmp += 1;    // 调整奇偶性
            pSrc += 4;          // 指针后移
            gsmSerializeNumbers(pSrc, pDst->TPA, tmp);    // 取 TP-RA 号码
            pSrc += tmp;        // 指针后移
        }
```

TPDU 段协议标识、编码方式、用户信息等：

```
        // TPDU 段协议标识、编码方式、用户信息等
        gsmString2Bytes(pSrc, (unsigned char*)&pDst->TP_PID, 2); // 取协议标识(TP-PID)
        pSrc += 2;        // 指针后移
        gsmString2Bytes(pSrc, (unsigned char*)&pDst->TP_DCS, 2); // 取编码方式(TP-DCS)
        pSrc += 2;        // 指针后移
        gsmSerializeNumbers(pSrc, pDst->TP_SCTS, 14);  // 服务时间戳字符串(TP_SCTS)
        pSrc += 14;       // 指针后移
        gsmString2Bytes(pSrc, &tmp, 2);    // 用户信息长度(TP-UDL)
        pSrc += 2;        // 指针后移
        if (pDst->TP_DCS == GSM_7BIT)
        {
            // 7-bit 解码
            nDstLength = gsmString2Bytes(pSrc, buf, tmp & 7 ? (int)tmp * 7 / 4 + 2 : (int)tmp
* 7 / 4);  // 格式转换
            gsmDecode7bit(buf, pDst->TP_UD, nDstLength);    // 转换到 TP-DU
            nDstLength = tmp;
        }
        else if (pDst->TP_DCS == GSM_UCS2)
        {
            // UCS2 解码
            nDstLength = gsmString2Bytes(pSrc, buf, tmp * 2);    // 格式转换
            nDstLength = gsmDecodeUcs2(buf, pDst->TP_UD, nDstLength);// 转换到 TP-DU
        }
        else
        {
            // 8-bit 解码
            nDstLength = gsmString2Bytes(pSrc, buf, tmp * 2);        // 格式转换
            nDstLength = gsmDecode8bit(buf, pDst->TP_UD, nDstLength); // 转换到 TP-DU
        }
        // 返回目标字符串长度
        return nDstLength;
    }
```

依照 GSM 07.05，发送短消息用 AT+CMGS 命令，阅读短消息用 AT+CMGR 命令，列出短消息用 AT+CMGL 命令，删除短消息用 AT+CMGD 命令。但 AT+CMGL 命令能够读出所有的短消息，所以我们用它实现阅读短消息功能，而没用 AT+CMGR。

下面是发送、读取和删除短消息的实现代码：

```
// 发送短消息
// pSrc：源 PDU 参数指针
BOOL gsmSendMessage(const SM_PARAM* pSrc)
{
```

```
    int nPduLength;        // PDU 串长度
    unsigned char nSmscLength;    // SMSC 串长度
    int nLength;           // 串口收到的数据长度
    char cmd[16];          // 命令串
    char pdu[512];         // PDU 串
    char ans[128];         // 应答串

    nPduLength = gsmEncodePdu(pSrc, pdu);    // 根据 PDU 参数，编码 PDU 串
    strcat(pdu, "\x01a");        // 以 Ctrl-Z 结束

    gsmString2Bytes(pdu, &nSmscLength, 2);    // 取 PDU 串中的 SMSC 信息长度
    nSmscLength++;        // 加上长度字节本身

    // 命令中的长度，不包括 SMSC 信息长度，以数据字节计
    sprintf(cmd, "AT+CMGS=%d\r", nPduLength / 2 - nSmscLength);    // 生成命令

    WriteComm(cmd, strlen(cmd));    // 先输出命令串

    nLength = ReadComm(ans, 128);   // 读应答数据

    // 根据能否找到"\r\n> "决定成功与否
    if (nLength == 4 && strncmp(ans, "\r\n> ", 4) == 0)
    {
       WriteComm(pdu, strlen(pdu));        // 得到肯定回答，继续输出 PDU 串

       nLength = ReadComm(ans, 128);       // 读应答数据

       // 根据能否找到"+CMS ERROR"决定成功与否
       if (nLength > 0 && strncmp(ans, "+CMS ERROR", 10) != 0)
       {
          return TRUE;
       }
    }

    return FALSE;
}
```

读取短消息：

```
// 读取短消息
// 用+CMGL 代替+CMGR，可一次性读出全部短消息
// pMsg：短消息缓冲区，必须足够大
// 返回：短消息条数
int gsmReadMessage(SM_PARAM* pMsg)
{
   int nLength;        // 串口收到的数据长度
   int nMsg;           // 短消息计数值
   char* ptr;          // 内部用的数据指针
   char cmd[16];       // 命令串
   char ans[1024];     // 应答串

   nMsg = 0;
   ptr = ans;

   sprintf(cmd, "AT+CMGL\r");    // 生成命令

   WriteComm(cmd, strlen(cmd));    // 输出命令串
```

```
    nLength = ReadComm(ans, 1024);    // 读应答数据

    // 根据能否找到"+CMS ERROR"决定成功与否
    if (nLength > 0 && strncmp(ans, "+CMS ERROR", 10) != 0)
    {
        // 循环读取每一条短消息，以"+CMGL:"开头
        while ((ptr = strstr(ptr, "+CMGL:")) != NULL)
        {
            ptr += 6;        // 跳过"+CMGL:"
            sscanf(ptr, "%d", &pMsg->index);    // 读取序号

            ptr = strstr(ptr, "\r\n");    // 找下一行
            ptr += 2;        // 跳过"\r\n"

            gsmDecodePdu(ptr, pMsg);    // PDU 串解码

            pMsg++;        // 准备读下一条短消息
            nMsg++;        // 短消息计数加 1
        }
    }

    return nMsg;
}
```

删除短消息：

```
// 删除短消息
// index: 短消息序号，从 1 开始
BOOL gsmDeleteMessage(int index)
{
    int nLength;        // 串口收到的数据长度
    char cmd[16];       // 命令串
    char ans[128];      // 应答串

    sprintf(cmd, "AT+CMGD=%d\r", index);    // 生成命令

    // 输出命令串
    WriteComm(cmd, strlen(cmd));

    // 读应答数据
    nLength = ReadComm(ans, 128);

    // 根据能否找到"+CMS ERROR"决定成功与否
    if (nLength > 0 && strncmp(ans, "+CMS ERROR", 10) != 0)
    {
        return TRUE;
    }

    return FALSE;
}
```

以上发送 AT 命令过程中用到了 WriteComm 和 ReadComm 函数，它们是用来读写串口的，依赖于具体的操作系统。在 Windows 环境下，除了用 MSComm 控件，以及某些现成的串口通信类之外，也可以简单地调用一些 Windows API 来实现。以下是利用 API 实现的主要代码，注意我们用的是超时控制的同步(阻塞)模式。

```
// 串口设备句柄
HANDLE hComm;

// 打开串口
// pPort: 串口名称或设备路径，可用"COM1"或"\\.\COM1"两种方式，建议用后者
// nBaudRate: 波特率
// nParity: 奇偶校验
// nByteSize: 数据字节宽度
// nStopBits: 停止位
BOOL OpenComm(const char* pPort, int nBaudRate, int nParity, int nByteSize, int nStopBits)
{
   DCB dcb;       // 串口控制块
   COMMTIMEOUTS timeouts = {    // 串口超时控制参数
      100,       // 读字符间隔超时时间: 100 ms
      1,         // 读操作时每字符的时间: 1 ms (n个字符总共为n ms)
      500,       // 基本的(额外的)读超时时间: 500 ms
      1,         // 写操作时每字符的时间: 1 ms (n个字符总共为n ms)
      100};      // 基本的(额外的)写超时时间: 100 ms

   hComm = CreateFile(pPort,    // 串口名称或设备路径
         GENERIC_READ | GENERIC_WRITE,    // 读写方式
         0,                  // 共享方式: 独占
         NULL,               // 默认的安全描述符
         OPEN_EXISTING,      // 创建方式
         0,                  // 不需设置文件属性
         NULL);              // 不需参照模板文件

   if (hComm == INVALID_HANDLE_VALUE) return FALSE;       // 打开串口失败

   GetCommState(hComm, &dcb);       // 取DCB

   dcb.BaudRate = nBaudRate;
   dcb.ByteSize = nByteSize;
   dcb.Parity = nParity;
   dcb.StopBits = nStopBits;

   SetCommState(hComm, &dcb);       // 设置DCB

   SetupComm(hComm, 4096, 1024);    // 设置输入输出缓冲区大小

   SetCommTimeouts(hComm, &timeouts);    // 设置超时

   return TRUE;
}

// 关闭串口
BOOL CloseComm()
{
   return CloseHandle(hComm);
}

// 写串口
// pData: 待写的数据缓冲区指针
// nLength: 待写的数据长度
void WriteComm(void* pData, int nLength)
{
   DWORD dwNumWrite;    // 串口发出的数据长度

   WriteFile(hComm, pData, (DWORD)nLength, &dwNumWrite, NULL);
```

```
}

// 读串口
// pData: 待读的数据缓冲区指针
// nLength: 待读的最大数据长度
// 返回: 实际读入的数据长度
int ReadComm(void* pData, int nLength)
{
   DWORD dwNumRead;    // 串口收到的数据长度

   ReadFile(hComm, pData, (DWORD)nLength, &dwNumRead, NULL);

   return (int)dwNumRead;
}
```

10.1.3　"实时"接收短消息的方法

在 10.1.2 节中，以上程序利用扩展的 AT 指令控制 ME(手机，GSM 模块等)发送和接收短消息的基本方法。其中，接收/阅读短消息采用主动查询的方法。TE 发送"AT+CMGL"指令，ME 则输出已接收和存储的短消息。这比较符合 AT 指令集的精神：有问有答，一问一答。但在实际应用中，发现这种方法存在一定的缺点：

- ME 接收到消息并存储起来，待查询时再传输到 TE，中间总会有一段时间的延迟。这个延迟取决于查询间隔。
- 发出该指令后，不论 ME 里有没有，有多少条消息，总要经过长时间的延迟，TE 才能收到最终的"OK"。完整的过程一般持续 5～10 秒。

本文介绍一种通过串口"实时"接收短消息的方法。当 ME 收到一条消息时，主动发出通知给 TE，或者直接将消息转发到 TE。与查询机制相比，它类似于中断机制。

先简要说明一下短消息类(class)的概念：根据指定储存的位置，短消息分为 class 0～3 四个类。也可以不指定类(no class)，由 ME 按默认设置进行处理，存储到内存或者 SIM 卡中。在 TPDU 的 TP-DCS 字节中，当 bit7-bit4 为 00x1, 01x1, 1111 时，bit1-bit0 指出消息所属类：

- 00 – class 0：只显示，不储存
- 01 – class 1：储存在 ME 内存中
- 02 – class 2：储存在 SIM 卡中
- 03 – class 3：直接传输到 TE

GSM MODEM 一般都支持一条"AT+CNMI"指令，可用于设定当有某类短消息到达时，如何处置它：只储存在指定的内存(易失的/非易失的)中，先储存后通知 TE，还是直接转发到 TE，等等。

"AT+CNMI"指令语法为：

```
AT+CNMI=[<mode>[,<mt>[,<bm>[,<ds>[,<bfr>]]]]]
```

mode - 通知方式：

- 0 – 不通知 TE。
- 1 – 只在数据线空闲的情况下，通知 TE；否则不通知 TE。
- 2 – 通知 TE。在数据线被占用的情况下，先缓冲起来，待数据线空闲，再行通知。
- 3 – 通知 TE。在数据线被占用的情况下，通知混合在数据中一起传输。

mt - 消息储存或直接转发到 TE：

- 0 – 储存到默认的内存位置(包括 class 3) 。
- 1 – 储存到默认的内存位置，并且向 TE 发出通知(包括 class 3) 。
- 2 – 对于 class 2，储存到 SIM 卡，并且向 TE 发出通知；对于其他 class，直接将消息转发到 TE 。
- 3 – 对于 class 3，直接将消息转发到 TE；对于其他 class，同 mt=1。

bm, ds, bfr 的含义，请参考相关标准文档。一般不需要去关心它们。

在程序中具体实现时，使用 mode=2, mt=1，比较简单。对所有类型的短消息，只要在收到 ME 送来的“+CMTI”通知后，用“AT+CMGR”指令读取消息内容就行了。TE 与 ME 之间的通信过程，举例如下：

```
(初始化)
AT+CNMI=? (查看能支持的设置范围)
+CNMI: (0-2),(0-3),(0,2,3),(0,1),(0,1)
OK

AT+CNMI?    (查看当前设置)
+CNMI: 0,0,0,0,0
OK

AT+CNMI=2,1    (设置为 mode=2, mt=1)
OK

AT+CNMI?    (再查看当前设置)
+CNMI: 2,1,0,0,0
OK

(过了一段时间，有一条消息到达)
+CMTI "ME",8   (通知：消息已经存储在 ME 内存中，序号为 8)

AT+CMGR=8   (读第 8 条消息)
+CMGR: 8,27
0891683108200505F0240D91683158812764F8000040205211037380074 1E19058341E01
OK

AT+CMGD=8   (删除第 8 条消息)
OK
```

还有一种方式 mode=2, mt=2 也很令人感兴趣。在这种方式下，除了 class 2 外，消息不存储，直接转发到 TE。需要处理消息通知和内容两种情况，复杂一些。但如果发送方也由程序控制，则可以只发 no class 或 class 1 的消息，这样不存储在接收方 ME 内存(一般是闪存，非易失性的)中，肯定能延长它的使用寿命。TE 与 ME 之间的通信过程，举例如下：

```
AT+CNMI=2,2  (设置为 mode=2, mt=2)
OK

(过了一段时间，有一条消息到达)
+CMT: ,26
0891683108200505F0040D91683158812764F8000840205211639180064F60597D0021
```

10.1.4　用串口收发 SMS 短信编程的一些讨论

本节对用串口收发短信时遇到的问题进行一些讨论。

（1）在用 AT 命令同手机通信时，需要注意哪些问题？

任何一个 AT 命令发给手机，都可能返回成功或失败。例如，用 AT+CMGS 命令发送短消息时，如果手机此时正好处于振铃或通话状态，就会返回一个"+CMS ERROR"。所以，应当在发送命令后，检测手机的响应，失败后重发。而且，因为只有一个通信端口，所以发送和接收不可能同时进行。

如果串口通信用超时控制的同步(阻塞)模式，一般做法是专门将发送/接收处理封装在一个工作子线程内。因为代码较多，这里就不详细介绍了。所附的 Demo 中，包含了完整的子线程和发送/接收应用程序界面的源码。

（2）以上 AT 命令，是不是所有厂家的手机都支持？

A ETSI GSM 07.05 规范直到 1998 年才形成最终 Release 版本(Ver 7.0.1)，在这之前及之后一段时间内，不排除各厂商在 DTE-DCE 的短消息 AT 命令有所不同的可能性。我们用到的几个 PDU 模式下的 AT 命令，是基本的命令，从原则上讲，各厂家的手机以及 GSM 模块应该都支持，但可能有细微差别。

（3）用户信息(TPUD)内除了一般意义上的短消息，还可以是图片和声音数据。关于手机铃声和图片格式方面，有什么规范吗？

为统一手机铃声、图片格式，Motorola 和 Ericsson, Siemens, Alcatel 等共同开发了 EMS(Enhanced Messaging Service)标准，并于 2002 年 2 月份公布。这些厂商格式相同。但另一手机巨头 Nokia 未参加标准的制定，手机铃声、图片格式与它们不同。所以没有形成统一的规范。EMS 其实并没有超越 GSM 07.05，只是 TP-UD 数据部分包含一定格式而已。各厂家的手机铃声、图片格式资料，可以查阅相关网站。

（4）用户信息(TP-UD)其实可以是任何的自定义数据，是吗？

是的，尽管手机上会显示乱码。这种情况下，编码方式已经没有任何意义。但注意仍然要遵守规范。比如，若指定 7-bit 编码方式，则 TP-UDL 应等于实际数据长度的 8/7(用进一法，而不是四舍五入)。在利用 SMS 进行点对点或多点对一点的数据通信的应用中，可以传输各种自定义数据，如 GPS 信息，环境监测信息，加密的个人信息，等等。

如果在传输自定义数据的同时还要收发普通短消息，则最简单的办法是在数据前面额外加个识别标志，比如"FFFF"，以区分自定义数据和普通短消息。

（5）我编写的短信发送程序，使用 PDU 格式发送，程序在广州使用一点问题也没有，在河南却怎么也发不出去。不知道为什么，短信“你好吗”格式如下：

```
河南：0891683108200005F011000D91683170031618F20008A9064F60597D5417
广州：0891683108301705F011000D91683170031618F20008A9064F60597D5417
```

发送短信时，要用 SIM 卡属地的 SMSC 号码。如果是在广州办的卡，即使在外地还是要用广州的 SMSC 号码。你的两个短信内 SMSC 号码不同，但用的是同一张 SIM 卡。

（6）短信中心的号码可否直接使用 SIM 卡中的号码，而不要用户输入？我用过的短信软件好像都是不用输 SMSC 号码的。

有一条“AT+CSCA”指令，可用于设置或查询服务中心号码。若手机中已存在此号码，

有两种解决办法。

一是用“AT+CSCA?”指令查询出来，然后自动将此号码写到 PDU 的 SCA 中。

一是 PDU 的 SCA 字段只写一个"00"，"08 91 68 31 ..." -> "00"

可用"AT+CSCA=xxxxxxxx"指令设置服务中心号码。

（7）我在超级终端上，用 at+cmgs 发送短消息，格式好像没有错误，但总返回“ERROR”。我输入的如下：

```
at+cmgs=30
> 0891683108100005F011000D91683118405057F000000006C8329BFD0E01
```

请问是什么原因？

“at+cmgs”指令很特殊，回车后还需要输入数据。此处是“CR”,不是“CRLF”，注意，在超级终端里直接回车是不是生成了两个字符（查看设置）。像“at+cmgl”指令，即使最后输入“CRLF”，也是不要紧的。

问题出在长度上。长度不是随便写的，你的例子中，长度应为 21。除去 SMSC 段（0891683108100005F0），从“11”开始算（即 11000D91683118405057F000000006C8329BFD0E01），除以 2 即得。

正确的写法应该是

```
at+cmgs=21
> 0891683108100005F011000D91683118405057F000000006C8329BFD0E01
```

（“>”是手机提示，不是输入的）

（8）我最近在编一个关于短消息的程序，在你的“通过串口收发短消息”中提到“Text Mode 是纯文本方式，可使用不同的字符集，从技术上说也可用于发送中文短消息，但国内手机基本上不支持，主要用于欧美地区”。是不是说我用 AT 指令“AT+CMGF=1”或“AT+CMGF=0”对我后来的收发短消息没什么影响啊？

Text Mode 写起来简单，直接发原文就行，发送非 ASCII 码内容也能发，但需要手机支持才能正确显示。如法语、德语的很多字符，编码大于 0x80，他们都是用 Text Mode。Text Mode 靠什么区分字符编码方式呢？有专门的字符集设定指令“AT+CSCS”。可以设定为扩充字符集“UCS2”。Siemens TC35/TC37 资料上说，它的“AT+CSCS”支持“UCS2”字符集，但我目前没有机会去亲自试验。正在使用 TC35/TC37 模块的朋友不妨试一下。

据我了解，中文短消息方面，在国内卖的各种手机只支持 PDU Mode，这成了事实上的标准。其实 PDU Mode 真的挺好用，估计以后 Text Mode 会萎缩。我们写的程序，我建议只采用 PDU Mode，即使是发纯英文信息也这样，编码倒是可以灵活采取 7bit 或 UCS2，因为 7bit 能发的长度是 UCS2 的 2 倍(仅对纯英文而言)。如果发送纯数据，不需要手机显示，可用 8bit。

（9）你的 smstraffic 类中的发送接收大循环中，是不是把所有收到的消息都放入消息队列后，然后执行删除程序吗？如果我是并发量很大的话，就是网关有很多短消息等着进入手机，读完所有短消息后，进行删除的过程中，因为短消息的排列顺序，而导致误删除呢（比如说我现在手机里有 1～15 条短消息，然后在我删除第一二条后，第三条自动填补为第一条，而新进来的短消息，16 条排在了第三条，而被 cancel 掉呢？）我试过好像短消息的排列不是

每次都一样啊？（在接收的时候，同一条短消息有时是 14 条，有时是第 15 条），这个怎么解决啊？

手机里消息都有一个物理序号，读出的时候带序号，删除也要根据序号删。“物理”二字很关键。这个序号相当于 ID，无论它前面有没有删除、删除了多少消息，都不会变的。假如原来有 1～15，删除了 1 和 2，又来了一条消息，手机内部的软件有两种处理方式：有的放在第 1 条，有的则放在第 16 条，我都见过。其实，它愿意放到哪个空闲的地方都行。但无论怎样，不会引起混乱的，因为读出是什么序号，就删除什么序号的。在执行删除命令前，消息还是在原来那个地方，不会被后来的覆盖。

如果说网关有很多短消息等着进入手机，量很大，这种处理方式效率不高，因为 AT+CMGL 占用很长时间，这段时间手机无法从 SMSC 接收新消息。采用我说的“实时”接收方法比较好，消息来了直接传出来，不经过写入手机的过程。

（10）我用 Nokia 8210 串口数据线，连上电脑的 com1 口，用 SmsTest 运行提示“没有发现 MODEM”，跟踪发现 gsmInit()检测中串口发 AT 指令没有回应“OK”。按你的提示我安装了 Nokia MODEM 驱动程序，（Windows 2000 Server 系统）虚拟出 com3 口的一个 8210 MODEM 设备，再次调用 smsTest 还是提示“没有发现 MODEM”。但用串口线，手机能通过 LogoManager 手机管理软件进行相应的图片 LOGO，短信发送操作。

Nokia 手机本身没有带 MODEM 功能，用专业术语讲就是不具备 TA(Terminal Adapter)接口，需要驱动转换，不管是真的串口，USB 还是红外接口，反正它能虚拟出“标准 MODEM”串口来。AT 命令只能用标准异步通信。

在我的印象中，Nokia 8210 需用红外线接口同 PC 通信。估计你装的那个驱动是 IR->COM 转换的，而不是驱动串口数据线的，可能你的电脑没有红外接口，所以 com3 也连不上。

要试(虚拟)串口是否连接正确，很简单，用 Windows 自带的“超级终端”在特定虚拟端口连上，敲个“AT”回车，看有没有反应，正确回答应该是“OK”。

Nokia 数据线上跑的是“Nokia 语”，Nokia 专有协议的数据，不是通用/扩展的 AT 命令集。LogoManager 能听、能说“Nokia 语”，所以不需要安装驱动就能工作。Nokia 有一个免费的“Nokia PC Connectivity SDK”，可供开发 Nokia 手机使用。至于 LogoManager 是不是用的这个开发包，那就不得而知了。

（11）在 SmsTest 中，发出 AT 命令，然后接收应答，比如

```
WriteComm("AT+CMGF=0\r", 10);
ReadComm(ans, 128);
```

在 WriteComm 函数后接着就调用 ReadComm，是不是太急？这里的 ReadComm 函数是读返回的这个字符串，还是其中的单个字符或不完全的字符串？

请问超时控制设多少最合适啊？

关于读串口，程序中是这样设定超时控制的：

```
COMMTIMEOUTS timeouts = {  // 串口超时控制参数
   100,        // 读字符间隔超时时间：100 ms
   1,          // 读操作时每字符的时间：1 ms (n个字符总共为n ms)
   500,        // 基本的(额外的)读超时时间：500 ms
   1,          // 写操作时每字符的时间：1 ms (n个字符总共为n ms)
   100};       // 基本的(额外的)写超时时间：100 ms
```

ReadComm 什么时候返回呢？按此 timeout 设定，若 n=128（ReadComm 的第二个参数）：

- 若无任何数据，则等待 500+1*128=628 毫秒返回。也就是说，若没有连上手机，则根本不存在应答，ReadComm 会持续阻塞 628 毫秒，而后返回。
- 若数据连续传输，且字符间隔也未超过了 100 毫秒，但时间已经到了 628 毫秒，则返回已读取的字符(串)。接收到的可能是不完全的字符串。
- 若在 628 毫秒内，字符间隔超过了 100 毫秒（第一个字符之前等待的时间不算），则返回已读取的字符(串)。接收到的应该是完整的字符串。

在手机正确连接的情况下，主要是最后一条起作用。一段数据是连续传输的，若波特率是 9600bps，可以算出字符间隔是 0.1 毫秒左右，远小于 100 毫秒，不会读一个字节或部分数据就返回；通常是数据完毕后才可能出现等待 100 毫秒而返回的情况。举个例子，若在执行 ReadComm(ans, 128)后 150 毫秒收到“OK\r\n”，则还需要额外等 100 毫秒，也就是说函数将在 250 毫秒后返回。这里传输 4 个字节数据的时间被忽略不计了。如果觉得读得太急，可将基本的(额外的)读超时时间调大一些。不过 500 毫秒内还没有应答，可能是连接故障造成的。

需要特别注意的是，“AT+CMGL”指令及其应答。可能是由于需要扫描所有存储区域的缘故，手机在逐条送出短消息后，还需要延迟好几秒的时间才能送出最后的“OK”。当然可以通过设定上面的基本读超时时间和读字符间隔超时时间都很长（比如 20 秒），并且一次读很长的数据（比如 2000），来达到目的。但这样一来，函数阻塞时间太长，若恰好这时要程序退出，你会赫然发现“该程序无响应”。SmsTest 中解决办法是：循环读取串口数据，将每次读取的数据拼接起来，最后得到完整的应答。gsmGetResponse()每次可能读取部分数据，将新数据追加到已读数据后，且检测是否见到“OK”或“ERROR”，以判断是否已经读到完整的数据。

10.2 计算机与 Rabbit 2000 嵌入式系统通信编程实例

10.2.1 Rabbit 2000 微处理器介绍

Rabbit 2000 微处理器是 Rabbit 半导体公司为嵌入式应用开发的一种 8 位微处理器,它和 Z80 系列微处理器有相似的结构，具有高度的兼容性，但它的性能有更大的提高, Rabbit 2000 芯片为 100 针 PQFP 封装。操作电压为 2.7V～5V，最大时钟频率为 30MHz。具有多达 40 个通用 I/O 引脚，内建日历、时钟、看门狗、定时器、多级中断、双 DMA 通道，可外扩 4M 至 8MFlash,用于数据存储。Rabbit 具有突出的计算速度，这是因为源于 Z 80 的指令系统非常精简,并且存储器接口设计允许最大限度地利用存储器带宽。

对于 Rabbit 2000，传统的微处理器硬件和软件的开发已被简化，不再需要复杂的在线仿真器以及 EPROM 写入器，将 PC 串口通过一条简单的接口电缆与基于 Rabbit 2000 处理器的目标系统连接起来就可以进行软件的编辑、编译、调试、下载和运行。

Rabbit 2000 可以实现冷启动，因此，未经编程的 Flash 存储器可以接在适当的地方。其 RabbitCoreRCM2000 模块内含 Rabbit 2000 微处理器、大容量 Flash 及 SRAM，内建以太网接口，可直接通过网络实现监控,具备 RS232/485 接口，可使各种串行设备快速进行网络连接。Rabbit 2000 微处理器的软件开发平台 Dynamic C Premier 集编辑、编译、链接、调试、下载

于一体，并有完善的 TCP/IP 协议栈，支持全功能 RS232/485 通讯，配备各种 I/O 驱动函数库，完善的文件管理系统，可在 Flash 或 SRAM 上建立数据文件便于存储系统或用户数据。

归纳起来，Rabbit 2000 微处理器具有如下一些特点:

- 逻辑上的无胶粘逻辑连接结构使硬件系统设计更为简单;
- 具有大量的串行口，通信速度快，并且能够通过程序控制处理器的速度和功耗;
- 能产生标准、精确的脉冲及边沿触发信号，具有多个中断优先级;
- 即使在 32KHz 的超低功耗模式下，处理器还可继续运行，进行计算和逻辑测试;
- Rabbit 处理器可以作为智能外设或从处理器，如协议栈可以卸载到 Rabbit 从处理器，因此可以使用任何处理器作为主处理器，Rabbit 处理器通过从属口与其进行通信，从而实现网络功能;
- Rabbit 处理器可以冷启动，因此未经编程的闪存可以焊接到电路板上;
- 可以编写严谨的软件，可以是 1000 行，也可以是 50000 行 C 代码，工具是现成的，并且价格低廉。
- 如果了解 Z 80 或 Z 180，就基本上了解了 Rabbit;
- 简单的 10 针编程接口替代了在线仿真器和 PROM 编程器，因此硬件设计和软件开发都简化了。
- 具有带后备电池的时间/日期时钟。

10.2.2　动态 C（Dynamic C）语言介绍

动态 C(Dynamic C)语言是基于 Windows 95/98/NT 及 Linux 平台的 Rabbit 应用软件的完整开发系统。它作为应用程序运行于 IBM PC 兼容机上,是 Z-World 公司为基于 Rabbit 微处理器的嵌入式系统而设计的专门的 C 编译系统。

动态 C 的特点如下:

- 集成了编辑、编译、连接、下载、调试等功能。它的文本编辑器具有 Windows 风格的下拉式菜单,大部分常用命令都有相应的快捷键方式,因而使其简单易用。应用程序可以在源代码与机器码层次交互地执行和调试。
- 支持汇编语言,C 程序可与汇编程序混合在一起进行编译,因此,可以用 C 语言编写中断服务程序。
- 具有超强的调试功能,可使用 printf 命令,可以观察表达式的值,支持单步执行和断点操作以及其他高级调试特性。
- 提供了到 C 语言的扩展以支持实际的嵌入式系统开发,支持协作式和抢先式的多任务进程。
- 提供了许多函数库源代码。这些库支持实时编程和硬件层次的 I/O,并提供了标准的字符串函数和数学函数。
- 直接对存储器进行编译,并且在编译用户程序时,加入了基本输入输出系统(BIOS)。函数和库都是动态地编译、连接、下载的,在快速的 PC 上,在 115200 b / s 波特率下,动态 C 可在 5 s 内下载 30000 B 代码。
- 作为应用程序运行于 PC 机上,可以在无预安装程序的情况下冷启动基于 Rabbit 微处理器的目标系统。目标系统的闪存可以是空白的,也可以存有数据。冷启动能力允许将闪

存焊接于目标系统板上,因此消除了插口、启动模块和 PROM 编程设备。

动态 C 语言有如下几点创新。

- 函数链(Function chaining):动态 C 特有的概念,允许特殊的代码段嵌入到一个或多个程序中。当一个已命名的函数链执行时,所有属于该函数链的代码段都将被执行。函数链可使软件执行初始化操作,进行数据恢复或其他任务。
- 协语句(Costatements):允许在单一程序中模拟并行进程,每一进程主动让出 CPU 的控制权以实现协作式多任务,每一协语句都有一个与其相对应的协数据(CoData)类型的结构体。
- 协函数(Cofunction):像协语句一样,协函数允许在单一程序中模拟协作式多任务进程。但与协语句不同的是,协函数能够实现参数的传递,并能返回计算值,而不是一个结构。
- Slice 语句:能够实现单一程序中的抢先式多任务进程以满足实时性要求较高的情况。
- interrupt 关键字:使编程者可以用 C 语言编写中断服务程序,从而使软件设计变得更简单。
- 动态 C 支持嵌入的汇编程序和独立的汇编程序,使编程更加灵活。
- shared 和 protected 关键字:可使不同程序中共享的和存储在电池供电存储器中的数据得到保护。
- 动态 C 具有一系列的特性允许编程者最大限度地利用扩展存储器。

从以上所述可以看出,动态 C 不同于传统的 C 语言,动态 C 是完全按照嵌入式系统要求设计的,是开发嵌入式应用软件的有力工具。

10.2.3　某车载无线调度系统实例介绍

1. 某车载无线调度系统需求说明

（1）硬件需求

- 显示终端能够满足恶劣环境的使用需要，能够长期经受剧烈振动,能够满足零下 20 度环境的使用；
- 大液晶蓝色显示屏；
- 支持串口连接。

（2）软件需求

- 提供底层编程函数，用户可以自己开发应用程序或修改程序；
- 同时支持以太网和串口通信。

（3）系统需求

车载无线调度系统示意图如图 10.2.3.1 所示。

（4）系统功能描述

- 每个终端可以根据串口寻址；
- 中心主机可以发送任意文字组合的信息（英文和中文简体一、二级字库支持的汉字）到指定的车载终端，同时支持广播方式发送信息；信息可以分多条发送，车载终端的上下移键可以选择任意一条信息进行确认或删除，处理结果可以返回中心，中心可以根据返回的信息知道是哪条信息被处理了；

➢ 车载终端上的功能键的按下或菜单的选定的结果可以返回到中心主机并显示出来其含义和此车载终端地址。

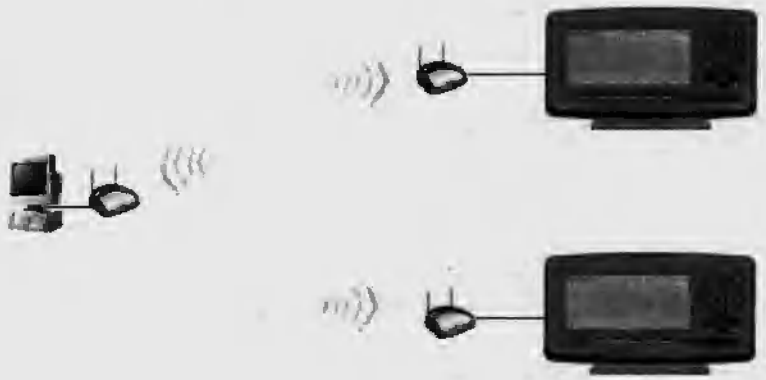


图 10.2.3.1 车载无线调度系统示意图

2. 应用程序介绍

上位机（中心控制系统）以 VC++实现，其人机界面如图 10.2.3.2 所示。

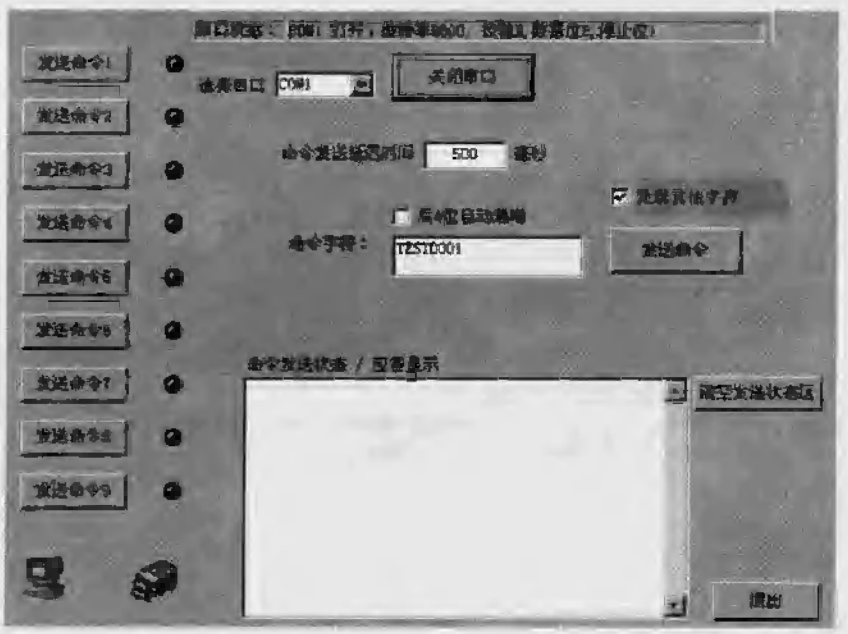


图 10.2.3.2　上位机控制界面图

➢ 接收部分的校验处理：VehicleTerminalDlg.cpp

```
LONG CVehicleTerminalDlg::OnCommunication(WPARAM ch, LPARAM port)
```

➢ 发送部分的校验处理：VehicleTerminalDlg.cpp

```
#define CR 0x0D  //回车符
#define LF 0x0A  //换行符
void CVehicleTerminalDlg::SendString(CString &str)//发送字符串
{
    char checksum=0,cr=CR,lf=LF;
 char c1,c2;
    for(int i=0;i<str.GetLength();i++)
        checksum = checksum^str[i];
 c2=checksum & 0x0f;    c1=((checksum >> 4) & 0x0f);
 if (c1 < 10) c1+= '0'; else c1 += 'A' - 10;
 if (c2 < 10) c2+= '0'; else c2 += 'A' - 10;
    CString str1;
    str1='$'+str+"*"+c1+c2+cr+lf;
    m_Port.WriteToPort((LPCTSTR)str1); //串口发送
}
```

（详细程序有兴趣的读者可根据前面章节介绍的内容自行编写，此处略去）

下位机采用基于 Rabbit 微处理器开发，可用于设备的人机交互及通讯装置。利用 T6963C 点阵式液晶图形显示控制器，提供 240×128 的宽温液晶，一个隔离的 RS232 口作为维护口（Rabbit 的 D 口），一个隔离的 RS422/RS485（可跳线选择，Rabbit 的 C 口），一个隔离的 RS485 口（RABBIT 的 B 口），一个隔离的 CAN 接口（采用 SJA1000CAN 控制器），高达 512KB 数据内存和 512KB 程序内存。

其中，串口实现部分的有关程序介绍如下，所有涉及的动态 C 语句都给出解释：

```
#define TIMEOUT 200L
//用来进行系统初始化的函数
void SYS_INIT(void)
{
WrPortI ( SPCR, &SPCRShadow, 0x84 );
/*WrPortI 内部 I/O 寄存器写；SPCR 为从端口控制寄存器，禁止从属口；//端口 A 输出使能*/
 WrPortI ( PEFR, &PEFRShadow, 0x80);
// PEFR 为端口 E 功能寄存器，设置 PE7 外部 I/O 片选
 WrPortI ( PEDDR, &PEDDRShadow, 0x80);
 // PEDDR 为端口 E 数据方向寄存器，设置 PE7 为输出，其余为输入
 WrPortI ( IB7CR, &IB7CRShadow, 0x48 ); // 设置 PE7 等待状态，片选、//写使能

 keyinit();//键盘初始化，用到 8279 端口
 WrPortI(I0CR, NULL, 0x07);  //在 PE0 引脚使能外部中断 0，下降沿触发
 fnLCMInit();//LCM 初始化，用到 T6963C 点阵式液晶显示屏

 serDopen(9600);
 serDrdFlush();
}
```

语法解释：

```
int serXopen(long baud); /* 其中X = A|B|C|D */
```

打开串口 X。

用户必须为每个端口定义缓冲大小 XINBUFSIZE 和 XOUTBUFSIZE，例：

```
#define XINBUFSIZE 63
#define XOUTBUFSIZE 127
```

定义缓冲大小为 2^n-1，使循环缓冲操作非常有效。如果定义值不等于 2^n-1，则系统使用默认值 31，并提出警告。

参数

- baud：波特率，该值不应小于外围电路时钟频率的 1/8192

返回值：

- 1: Rabbit 上设置波特率与输入值一致
- 0: Rabbit 上设置波特率与输入值不一致

```
void serXrdFlush(); /* 其中 X = A|B|C|D */
```

刷新串口 X 输入缓冲区。

```
void My_Delay()
{
 int flag;
 flag=1;

 while (flag)
   costate {
     waitfor (DelayMs(TIMEOUT));
     flag=0;
   }
}
```

语法解释：

Costate 表示协语句的开始，其语法为：

```
costate [ name [state] ] {
[ statement | yield; | abort; | waitfor( expression ); ] . . .}
```

name 可以是:

- 一个有效的，没有定义过的 C 命名；
- 一个已经定义过的全局或局部 CoData 结构名
- 一个指向 CoData 结构类型的指针

name 也可以是默认的，编译器将创建一个 CoData 类型的“无名”结构。

state 可以是：

- always_on ：costatement 总是激活。这一选项使得编译器不会检查(paused condition)中止条件，并且 CoPause 不能使用。
- init_on ： costatement 初始激活，遇到执行指令(execution thread)自动执行。这一操作完成后 costatement 失效。costatement 可以使用 CoPause 中止。
- 如果 state 缺省，已命名的 costatement 在中止条件下初始化，直到 CoBegin 或 CoResume 语句执行才有效。 costatement 将执行一次后失效。

未命名的 costatements 是 always_on。用户不能指定为 init_on 。

```
int My_Get_Ch()//接收字符
{
  int get_ch;
  CoData Co_Get,Co_Time;

  get_ch=0;
  CoBegin(&Co_Get);
  CoBegin(&Co_Time);
  while (1) {
     costate Co_Get {
       get_ch=serDgetc();
       if (get_ch!=-1) {
         CoPause(&Co_Get);
         CoPause(&Co_Time);
         return(get_ch);
       }
     } //End of costate

     costate Co_Time {
       waitfor(DelayMs(TIMEOUT));
       CoPause(&Co_Get);
       CoPause(&Co_Time);
       return(-1);
     } //End of costate
  } //End of while
}
```

语法解释：

CoData 结构：

```
typedef struct {
   char CSState;
   unsigned int lastlocADDR;
   char lastlocCBR;
   char ChkSum;
   char firsttime;
   union {
      unsigned long ul;
      struct {
         unsigned int u1;
         unsigned int u2;
      } us;
   } content;
   char ChkSum2;
} CoData;
void CoBegin(CoData* p)
```

该函数初始化一个 costatement 的 CoData 结构；因此，运行到这一语句时， costatement 将被执行。

```
void CoPause(CoData* p)
```

该函数改变 CoData， 使相应的 costatement 中止。当调用 CoResume 或 CoBegin，才被释放。

```
int serXgetc(); /* 其中 X = A|B|C|D */
```

从串口读缓冲区获取字符。

返回值：

- Success: 字符在低字节，0 在高字节
- Failure: −1

用户可根据接收字符函数 My_Get_Ch()，编写字符串接收函数 My_Receive_Str，及字符接收检验函数 RX_Check() 和字符发送检验函数 Tx_Check()。

```
int My_Send_Str(send_tx_info)//发送字符串
char send_tx_info[];
{
 int i;
 char Get_RX_Str[127],Get_RX_Info[127],Get_RX_Check[3];
 char Send_TX_Str[127],Send_TX_ANS[10];

 strcpy(Send_TX_Str,"$");
 strcat(Send_TX_Str,send_tx_info);
 Tx_Check(Send_TX_Str);

 for (i=0;i<=20;i++) {
  serDputs(Send_TX_Str);
  My_Delay();
  if (My_Receive_Str(Get_RX_Str,Get_RX_Info,Get_RX_Check)==1)
   switch (Get_RX_Info[0]) {
    case '#' : {
               return(1);
               break;
              }

    case '@' : {
               Send_TX_ANS[0]='$';
               Send_TX_ANS[1]='#';
               Send_TX_ANS[2]=Get_RX_Info[1];
               Send_TX_ANS[3]='\0';
               Tx_Check(Send_TX_ANS);
               serDputs(Send_TX_ANS);
               GET_MESSAGE(Get_RX_Info,-1);//消息获取函数，其内容略去
               break;
              }

    default  : {
               serDputs(Get_RX_Str);
               GET_MESSAGE(Get_RX_Info,-1);
              }

   } // End of case
 } //End of for
 return(0);
}
```

语法解释：

```
int serXputc(char c); /* 其中 X = A|B|C|D */
```

写字符到串口写缓冲区。

参数

- c：待写的字符

返回值：

- 0：缓冲区锁住或已满
- 1：字符被发送

10.3 计算机与 PLC 通信程序实例

可编程序控制器(Programmable Logical Controller，简称 PLC)以其良好的适应性和可扩展能力而得到越来越广泛的应用。采用 PLC 的控制系统或装置具有可靠性高、易于控制、系统设计灵活、能模拟现场调试、编程使用简单、性价比高、有良好的抗干扰能力等特点。但是，PLC 也有不易显示各种实时图表／曲线（趋势线）和汉字、无良好的用户界面、不便于监控等缺陷。随着现场总线技术的发展和控制网络化趋势，PC 机与 PLC 之间越来越多地建立起通信联系。用一台计算机(上位机)去监控下位机（PLC），既能保证系统性能，又能使系统操作简便，便于生产过程的有效监督。这就要求 PC 与 PLC 之间稳定、可靠的数据通讯。

下面以西门子(Siemens)200 系列的 PLC 为例进行说明 PC 与 PLC 之间的串口通信编程。

西门子 S7-200 系统为用户提供了灵活的通讯功能。集成在 S7-200 中的点对点接口(PPI)可用普通的双绞线做波特率高达 9600bit/s 的数据通讯,用 RS485 接口实现的高速用户可编程接口,可使用专用位通讯协议（如 ASCII）做波特率高达 38.4 kbit／s 的高速通讯,并可按步调整。而 PC 的接口为 RS232,两者之间需要进行电平转换。利用西门子公司的 PC/PPI 电缆,可将 S7-200 CPU 与计算机连接起来组成 PC/PPI 网络,实现点对点通讯。

下面仅列出下位机串行通讯相关程序，上位机程序略去。

（1）通讯口初始化

西门子 CPU 214 有一个串行口 Port 0,要实现通讯,必须先进行初始化。

初始化指令为:

```
MOVB 9,SMB30
```

//通讯方式为自由口通讯,9600 波特,无奇偶校验位,数据位为 8 位

CPU 214 使用特殊寄存器 SMB30 存储通讯方式控制字。

（2）PLC 发送命令

PLC 发送命令为:

```
XMT VB99,0
```

此命令向 Port 0 发送字符,字符起始地址为 VB100,字符个数存储在寄存器 VB99 中。PLC 有多种中断源，Port 0 接收字符会引起中断事件 8，在初始化程序中要先打开中断，并设置中断程序号：

```
ATCH  0,8  //中断程序 0 中处理接收字符中断事件 8
ENI  //允许中断从通讯口传来的数据暂存在 8 位寄存器 SMB2 中，在中断程序 0 中将移到指定的缓冲区中。
```

（3）PLC 通讯程序清单

接收程序:

```
LD    SM0.1  //第一扫描（SM0.1=1），通讯口初始化
MOVB  9,SMB30 //自由口通讯模式，9600 波特，无奇偶校验位，
MOVD  &VB100 ,VD56  //地址指针指向 VB100，即设置接收数据缓冲区
MOVB  0,VB200 //VB200 存放接收字符个数，先置 0
ATCH   0,8 //中断程序 0 处理接收字符事件
ENI  //允许中断
MEND //主程序结束
INT  0  //接收中断程序 0
MOVB  SMB2,VD56 //将 SMB2 中的字符放到缓冲区中
INCD VD56 //地址指针加 1
INCB VB200 //接收字符个数加 1
RETI //中断程序 0 结束，返回主程序
```

发送程序：

```
LD  SM0.1 //第一扫描(SM0.1=1)
MOVB 9,SMB30 //通讯口初始化
MOVB 250,SMB34 //设置中断 0 的定时中断时间为 250ms
MOVB 5,VB99 //发送 5 个字符
MOVB 16#41,VB100  //字符“A”，其 ASCII 码为 41(16 进制)
MOVB 16#39,VB101  //字符“9”
MOVB 16#38,VB102 //字符“8”
MOVB 16#37,VB103 //字符“7”
MOVB 16#36,VB104 //字符“6”
ATCH 1,10 //指定定时中断 10 调用中断程序 1
ENI
MEND
INT 1 //定时中断时间到，运行中断程序 1
XMT VB99,0
ATCH 2,9 //指定定时中断 9 调用中断程序 2
RETI
INT  2 //中断程序 2
DTCH 10 //切断定时中断 10 与中断程序 1 的联系
DTCH 9 //切断定时中断 9 与中断程序 2 的联系
RETI
```

10.4　MATLAB 环境串口编程通信实例

MATLAB 以其优秀的仿真和计算性能在工程技术领域和科学计算中的应用日益广泛，如果能够直接把数据从串口接收到 MATLAB 程序中，也是非常诱人的。本节在这方面做了一些探讨。

10.4.1　MATLAB 串口类 Serial 应用

建立串口对象的语法为：

```
S = SERIAL('PORT','P1',V1,'P2',V2,...)
```

其中，PORT 为端口号，PX 为属性名称，VX 为属性值。

例 10.4.1.1：　s=serial('COM1','BaudRate',9600)

执行该语句可看到如下结果：

```
Serial Port Object : Serial-COM1
```

```
    Communication Settings (通讯设置)
      Port:            COM1
      BaudRate:        9600
      Terminator:      'LF'
    Communication State (通讯状态)
      Status:          closed
      RecordStatus:    off
    Read/Write State (读写状态)
      TransferStatus:  idle
      BytesAvailable:  0
      ValuesReceived:  0
  ValuesSent:      0
```

可以通过 set(s)语句查看 serial 对象的全部属性。

常用的属性如下：

- 通讯属性(Communications Properties)

  BaudRate：波特率。

  Parity：检验位，其值可以是奇检验(odd)、偶检验(even)、无检验(none)。

  DataBits：数据位，通常有 6、7、8 位。

  StopBits：停止位，可以是 1、2 位。

  Terminator：指定结束字符。

- 写属性(Write Properties)

  BytesToOutput 指出输出缓冲区当前字节数。

  OutputBufferSize 指定输出缓冲区大小(以字节为单位)。

  Timeout 指定完成数据读写的最大等待时间。

  TransferStatus 指出是否有异步读写操作,其属性值如下。

  Idle：没有异步操作

  Read：有异步读操作

  Write：有异步写操作

  Read & write ：同时有异步读写操作

  ValuesSent 指出实际发送字节数。

- 读属性(Read Properties)

  BytesAvailable 指出输入缓冲区内可用的字符数。

  InputBufferSize 指定输入缓冲区的大小。

  ReadAsyncMode 指定异步方式是连续的还是手动的。

  TransferStatus 指出当前是否有异步读写操作。

  ValuesReceived 指出接收到的总字节数。

例 10.4.1.2：建立一个波特率为 19200，奇检验，8 位数据位，1 位停止位的串口对象。

```
s=serial('com1','BaudRate',19200,'Parity','odd','DataBits',8,'StopBits',1)
```

也可以使用 set 命令，对各属性单独设值，如：

```
set(s,'Parity','none');
```

建立串口对象后，在进行数据通讯之前，必须打开串口，其语法为：

```
fopen(s);
```

通讯结束后，应关闭串口对象，其语句为：

```
fclose(s);
```

关闭对象之后，变量 s 仍存在于系统之中，应将其清除，语句为：

```
delete(s)
```

此时，s 就是一个无效的对象了，可用如下语句将其从 MATLAB 工作空间中清除：

```
clear s;
```

其他常用串口 I/O 函数如下。

Disp 显示串口对象基本信息。

输入建立串口语句时，结尾处不加分号，回车，实际上也执行了该函数。

Fgetl 读一行文本，不包括字符串结束符，有如下几种语法格式：

```
tline = fgetl(obj)
[tline,count] = fgetl(obj)
[tline,count,msg] = fgetl(obj)
```

各参数分别为：

- Obj：串口对象
- Tline：所读的文本内容，不包括结束符
- Count：包括结束符在内的字符个数
- Msg：指出读操作是否成功

Fgets 读一行文本，包括字符串结束符，其语法格式与 Fgetl 类似。

Fprintf 以文本方式发送数据，有如下四种语法格式：

```
fprintf(obj,'cmd')
fprintf(obj,'format','cmd')
fprintf(obj,'cmd','mode')
fprintf(obj,'format','cmd','mode')
```

各参数分别为：

- obj：串口对象
- “cmd”：写字符串
- “format”：C 语言通用格式
- “mode”：指定是异步还是同步方式

Fread 以二进制方式读数据，其语法格式如下：

```
A = fread(obj,size)
A = fread(obj,size,'precision')
[A,count] = fread(...)
[A,count,msg] = fread(...)
```

各参数意义如下：

- Obj：串口对象
- Size：读取个数
- 'precision'：精度，指出是字符型、整型或浮点数
- A：从串口设备读取的二进制数
- Count：读到的个数
- Msg：指出读操作是否成功

Fscanf 以指定格式读入数据，其语法格式如下，各参数意义同前。

```
A = fscanf(obj)
A = fscanf(obj,'format')
A = fscanf(obj,'format',size)
[A,count] = fscanf(...)
[A,count,msg] = fscanf(...)
```

Fwrite 以二进制方式发送数据，其语法格式如下，各参数意义同 Fread 中所述。

```
fwrite(obj,A)
fwrite(obj,A,'precision')
fwrite(obj,A,'mode')
fwrite(obj,A,'precision','mode')
```

Get 获取串口对象属性：

```
get(obj)
out = get(obj)
out = get(obj,'PropertyName')
Readasync 异步方式读取数据。
Stopasync 停止异步读写操作。
```

例 10.4.1.3：建立例 10.4.1.2 的串口对象后，以文本方式发送“this is a matlab comm”。

```
>> fopen(s)
>> fprintf(s,'this is a matlab comm')
```

先打开串口调试助手，设置通讯属性与例 10.4.1.2 一致，则当上述语句执行时，可以看到如图 10.4.1.1 所示结果。

图 10.4.1.1　演示结果

在默认状态下，读写操作是以同步方式进行的，以下语句可使操作以异步方式进行：

```
fprintf(s, 'ok,this is a async', 'async');
```

通过以下语句获取当前串口对象的 TransferStatus 属性值，从而判断其是否为 idle 状态，如果是，说明发送完毕，可以发送新的数据。

```
get( s, 'TransferStatus' )
```

10.4.2　通过串口使 MATLAB Simulink 与下位机通讯进行控制

有时候，控制任务过于复杂，用 PLC 或单片机编写程序过于繁琐甚至不能实现，可以考虑利用 MATLAB Simulink 模块实现复杂控制。

比如用 C51 单片机对某电机进行控制，其控制参数可以从 Simulink 通过 RS232 口发给 C51 单片机，并通过 RS232 口得到 C51 的反馈信息。

方法如下：

（1）编写一个串口读写的 m 程序。

（2）用 S-fuction 进行封装。

（3）加入到 simulink 模型中去。

当然，这样用 M 文件 S-函数写的模块不能生成实时代码，无法利用 RTW 提供的许多强大功能；因此，最好用 C MEX S-函数写。更多的细节请参考 MATLAB 帮助文件。

10.4.3　xPC 目标环境下串口通信实现

xPC 目标可以通过串口线连接来实现上位机与下位机之间的通信。在 xPC 目标环境下，用户可以将装有 MATLAB 软件的 PC 机作为上位机，用 Simulink 模块来创建模型并进行非实时仿真，然后用 RTW 代码生成器等生成执行代码，并将其在与 PC 兼容的下位机上实时地运行。

下面仅对目标启动盘做一简要介绍，更多内容参见 MATLAB 帮助文件。

首先将上下位机用串口线相连，本例中用的均为 COM1 端口。

（1）在 MATLAB 命令行输入如下命令：

```
>> xpcsetup
```

打开图 10.4.3.1 所示对话框，并按图中黑色框选择部分填写，选取串口通信。

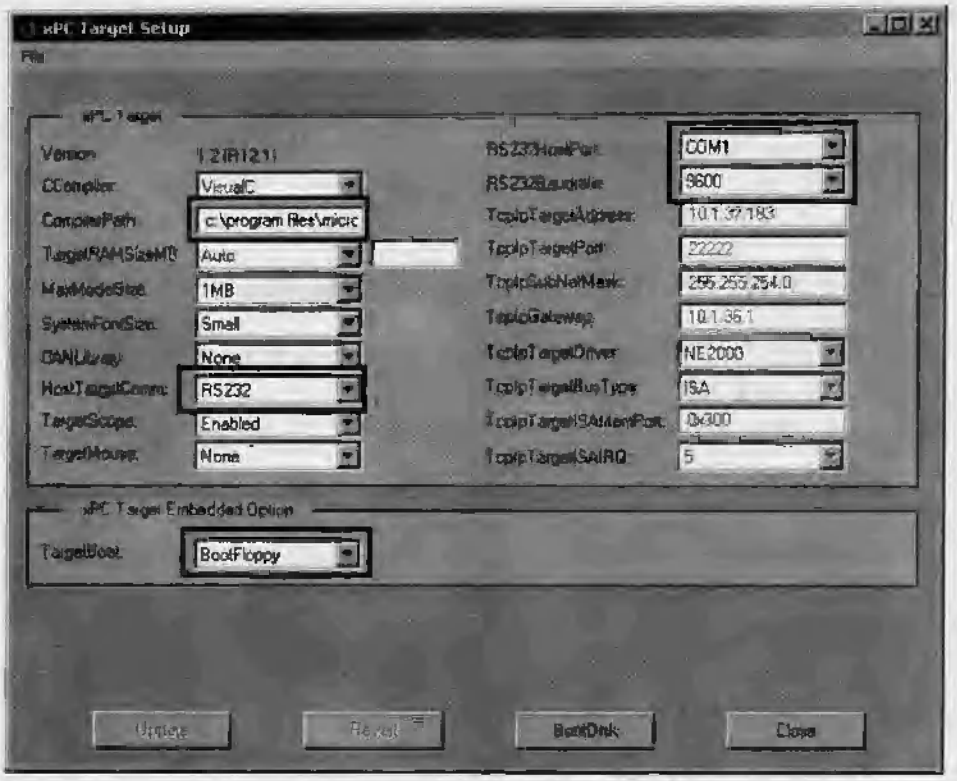


图 10.4.3.1　xPC Target Setup 对话框

（2）在软驱中放入一张软盘，单击图中 BootDisk 按钮即生成下位机启动盘，然后关闭对话框。

（3）在 MATLAB 命令行用如下命令检测目标启动盘：

```
>> xpcbootdisk
MATLAB 显示以下信息:
Insert a formatted floppy disk into your host PC's
disk drive and press a key to continue
```

按任意键，若显示以下信息，则表明启动盘环境设置正确。

```
Inserted floppy is a xPC Target boot floppy disk for the current xPC Target environment
```

（4）将目标启动盘插入下位机软驱中，并重启下位机。

（5）在上位机 MATLAB 命令行输入如下命令：

```
xpctest
```

若连线成功，即显示如下信息：

```
### xPC Target Test Suite 1.2
### Host-Target interface is: RS232 COM1
### Test 1, Ping target system using standard ping: ... SKIPPED
### Test 2, Ping target system using xpctargetping: ... OK
### Test 3, Reboot target using direct call: . OK
```

```
### Test 4, Build and download xPC Target application using model xpcosc: ... OK
### Test 5, Check host-target communication for commands: ... OK
### Test 6, Download xPC Target application using OOP: ... OK
### Test 7, Execute xPC Target application for 0.2s: ... OK
### Test 8, Upload logged data and compare it with simulation: ... OK
### Test Suite successfully finished
```

利用如下语句可以查看目标对象属性及其当前值：

```
tg=xpc

xPC Object

  Connected            = Yes
  Application          = xpcosc
  Mode                 = Real-Time Single-Tasking
  Status               = stopped
  CPUoverload          = none

  ExecTime             = 0.20
  SessionTime          = 399.22
  StopTime             = 0.20
  SampleTime           = 0.000250
  AvgTET               = 0.000008
  MinTET               = 0.000008
  MaxTET               = 0.000009
  ViewMode             = All

  TimeLog              = Vector(801)
  StateLog             = Matrix(801x2)
  OutputLog            = Matrix(801x2)
  TETLog               = Vector(801)
  MaxLogSamples        = 16666
  NumLogWraps          = 0
  LogMode              = Normal

  Scopes               = No Scopes defined
  NumSignals           = 7
  ShowSignals          = Off

  NumParameters      = 7
  ShowParameters     = Off
```

限于篇幅，本节对于 MATLAB 环境串口编程内容只进行了简要的叙述，旨在为拓宽读者知识面，起到抛砖引玉的作用。有兴趣的读者可以进一步地查阅相关资料。

第 11 章　串口通信基本概念及标准

[内容提要]

本章介绍了一些有关串口通信的基本概念，以方便读者在深入了解串口知识，并在编程过程中查阅。本章详细介绍的内容有：RS-232C 标准、RS-422/RS-485 串口标准；根据编程实践列写了串口调试注意事项；常用数据校验法；串口连接与 TCP/IP 连接进行对比；现场总线与 RS232、RS485 的本质区别；MODEM 的有关知识等。

11.1　串口通信基本概念

计算机与外界的信息交换称为通信。基本的通信方式有并行通信和串行通信两种。

并行通信是指一条信息的各位数据被同时传送的通信方式。并行通信的特点是：各数据位同时传送，传送速度快、效率高，但有多少数据位就需多少根数据线，因此传送成本高，且只适用于近距离（相距数米）的通信。

串行通信是指一条信息的各位数据被逐位按顺序传送的通信方式。串行通信的特点是：数据位传送，按位顺序进行，最少只需一根传输线即可完成，成本低但送速度慢。串行通信的距离可以从几米到几千米。

11.1.1　串行通信概述

图 11.1.1.1 为“串行通信”示意图，外设和计算机间使用一根数据信号线(另外需要地线，可能还需要控制线)，数据在一根数据信号线上一位一位地进行传输，每一位数据都占据一个固定的时间长度。这种通信方式使用的数据线少，在远距离通信中可以节约通信成本，当然，其传输速度比并行传输慢。

图 11.1.1.1　“串行通信”示意图

由于 CPU 与接口之间按并行方式传输，接口与外设之间按串行方式传输，因此，在串行接口中，必须要有“接收移位寄存器”（串→并）和“发送移位寄存器”（并→串）。典型的串行接口的结构如图 11.1.1.2 所示。

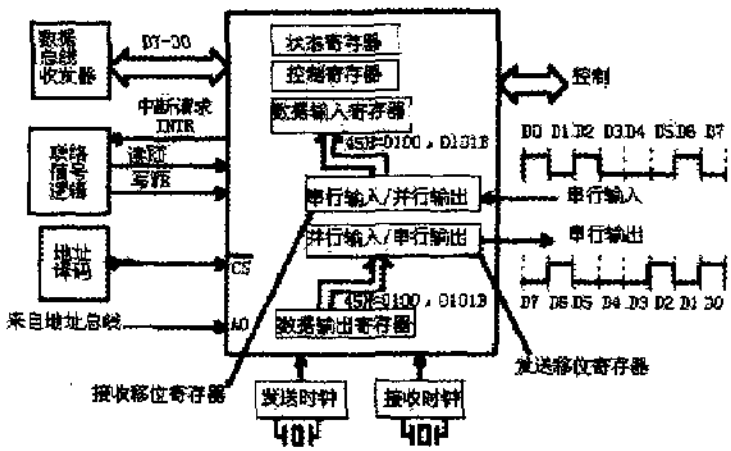


图 11.1.1.2 串行接口的结构

在数据输入过程中，数据 1 位 1 位地从外设进入接口的“接收移位寄存器”，当“接收移位寄存器”中已接收完 1 个字符的各位后，数据就从“接收移位寄存器”进入“数据输入寄存器”，CPU 从“数据输入寄存器”中读取接收到的字符。（并行读取，即 D7～D0 同时被读至累加器中）。“接收移位寄存器”的移位速度由“接收时钟”确定。

在数据输出过程中，CPU 把要输出的字符（并行地）送入“数据输出寄存器”，“数据输出寄存器”的内容传输到“发送移位寄存器”，然后由“发送移位寄存器”移位，把数据 1 位 1 位地送到外设。“发送移位寄存器”的移位速度由“发送时钟”确定。

接口中的“控制寄存器”用来容纳 CPU 送给此接口的各种控制信息，这些控制信息决定接口的工作方式。

“状态寄存器”的各位称为“状态位”，每一个状态位都可以用来指示数据传输过程中的状态或某种错误。例如，用状态寄存器的 D5 位为“1”表示“数据输出寄存器”空，用 D0 位表示“数据输入寄存器满”，用 D2 位表示“奇偶检验错”等。

能够完成上述“串<- ->并”转换功能的电路，通常称为“通用异步收发器”（UART：Universal Asynchronous Receiver and Transmitter），典型的芯片有：Intel 8250/8251,16550。

1．串行通信接口的基本任务

- 实现数据格式化：因为来自 CPU 的是普通的并行数据，所以，接口电路应具有实现不同串行通信方式下的数据格式化的任务。在异步通信方式下，接口自动生成起止式的帧数据格式。在面向字符的同步方式下，接口要在待传送的数据块前加上同步字符。
- 进行串并转换：串行传送，数据是一位一位串行传送的，而计算机处理数据是并行数据。所以当数据由计算机送至数据发送器时，首先把串行数据转换为并行数才能送入计算机处理。因此串并转换是串行接口电路的重要任务。
- 控制数据传输速率：串行通信接口电路应具有对数据传输速率——波特率进行选择和控制的能力。
- 进行错误检测：在发送时，接口电路对传送的字符数据自动生成奇偶校验位或其他

校验码。在接收时，接口电路检查字符的奇偶校验或其他校验码，确定是否发生传送错误。

- 进行 TTL 与 EIA 电平转换：CPU 和终端均采用 TTL 电平及正逻辑，它们与 EIA 采用的电平及负逻辑不兼容，需在接口电路中进行转换。
- 提供 EIA-RS-232C 接口标准所要求的信号线：远距离通信采用 MODEM 时，需要 9 根信号线；近距离零 MODEM 方式，只需要 3 根信号线。这些信号线由接口电路提供，以便与 MODEM 或终端进行联络与控制。

2. 串行通信接口电路的组成

为了完成上述串行接口的任务，串行通信接口电路一般由可编程的串行接口芯片、波特率发生器、EIA 与 TTL 电平转换器以及地址译码电路组成。其中，串行接口芯片，随着大规模继承电路技术的发展，通用的同步(USRT)和异步（UART）接口芯片种类越来越多，它们的基本功能是类似的，都能实现上面提出的串行通信接口基本任务的大部分工作，且都是可编程的。采用这些芯片作为串行通信接口电路的核心芯片，会使电路结构比较简单。

3. 有关串行通信的物理标准

为使计算机、电话以及其他通信设备互相沟通，现在，已经对串行通信建立了几个一致的概念和标准，这些概念和标准属于三个方面：传输率，电特性，信号名称和接口标准。

传输率是指每秒钟传送的二进制位数，单位为 bps(bits per second)：传输率也常叫波特率。它是衡量串行数据速度快慢的重要指标。国际上规定了一个标准波特率系列，标准波特率也是最常用的波特率，标准波特率系列为 110、300、600、1200、4800、9600 和 19200。大多数 CRT 终端都能够按 110 到 9600 范围中的任何一种波特率工作。打印机由于机械速度比较慢而使传输波特率受到限制，所以，一般的串行打印机工作在 110 波特率，点针式打印机由于其内部有较大的行缓冲区，所以可以按高达 2400 波特的速度接收打印信息。大多数接口的接收波特率和发送波特率可以分别设置，而且，可以通过编程来指定。

在波特率指定后，输入/输出移位寄存器在接收/发送时钟控制下，按指定的波特率速度进行移位，一般几个时钟脉冲移位一次。

波特率因子是指，发送/接收 1 个数据位所需要的时钟脉冲个数。如波特率因子为 16，则 16 个时钟脉冲移位 1 次。例：波特率为 9600bps，波特率因子为 32，则接收时钟和发送时钟频率为 9600×32＝307200Hz

串行通信中，数据位信号流在信号线上传输时，会引起畸变，畸变的大小与以下因素有关：

- 波特率——信号线的特征（频带范围）
- 传输距离——信号的性质及大小（电平高低、电流大小）

当畸变较大时，接收方出现误码。

在规定的误码率下，当波特率、信号线、信号的性质及大小一定时，串行通信的传输距离就一定，为了加大传输距离，必须加调制解调器。

RS-232-C 标准对两个方面做了规定，即信号电平标准和控制信号线的定义。RS-232－C 采用负逻辑规定逻辑电平，信号电平与通常的 TTL 电平也不兼容，RS-232-C 将-5V～-15V 规定为“1”，+5V～+15V 规定为“0”。更多内容请见 11.2 节。

11.1.2　单工、半双工和全双工的定义

根据信息的传送方向，串行通信可以进一步分为单工、半双工和全双工三种。如果在通信过程的任意时刻，信息只能由一方 A 传到另一方 B，则称为单工。如果在任意时刻，信息既可由 A 传到 B，又能由 B 传 A，但只能有一个方向上的传输存在，则称为半双工传输。如果在任意时刻，线路上存在 A 到 B 和 B 到 A 的双向信号传输，则称为全双工。电话线就是二线全双工信道。由于采用了回波抵消技术，双向的传输信号不致混淆不清。双工信道有时也将收、发信道分开，采用分离的线路或频带传输相反方向的信号，如回线传输。

串行通信中主要有两个技术问题，一个是数据传送，另一个是数据转换。数据传送主要解决传送中的标准、格式及工作方式等问题。数据转换是指数据的串并行转换。具体说，在发送端，要把并行数据转换为串行数据；而在接收端，却要把接收到的串行数据转换为并行数据。

由于单工目前已很少采用，下面着重介绍半双工和全双工两种方式。

1．全双工方式（full duplex）

当数据的发送和接收分流，分别由两根不同的传输线传送时，通信双方都能在同一时刻进行发送和接收操作，这样的传送方式就是全双工制，如图 11.1.2.1 所示。全双工的串行通信只需要一根输出线和一根输入线。数据的输出又称发送数据（TXD），数据的输入又称接收数据（RXD）。在全双工方式下，通信系统的每一端都设置了发送器和接收器，因此，能控制数据同时在两个方向上传送。全双工方式无须进行方向的切换，因此，没有切换操作所产生的时间延迟，这对那些不能有时间延误的交互式应用（例如远程监测和控制系统）十分有利。这种方式要求通信双方均有发送器和接收器，同时，需要 2 根数据线传送数据信号。（可能还需要控制线和状态线，以及地线）。

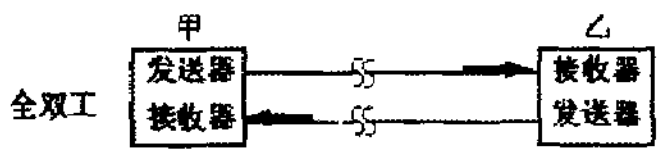


图 11.1.2.1　全双工方式

比如，计算机主机用串行接口连接显示终端，而显示终端带有键盘。这样，一方面，键盘上输入的字符送到主机内存；另一方面，主机内存的信息可以送到屏幕显示。通常，往键盘上打入 1 个字符以后，先不显示，计算机主机收到字符后，立即回送到终端，然后终端再把这个字符显示出来。这样，前一个字符的回送过程和后一个字符的输入过程是同时进行的，即工作于全双工方式。

2．半双工方式（half duplex）

若使用同一根传输线既做接收又做发送，虽然数据可以在两个方向上传送，但通信双方不能同时收发数据，这样的传送方式就是半双工制，如图 11.1.2.2 所示。采用半双工方式时，

通信系统每一端的发送器和接收器，通过收/发开关转接到通信线上，进行方向的切换，因此，会产生时间延迟。收/发开关实际上是由软件控制的电子开关。

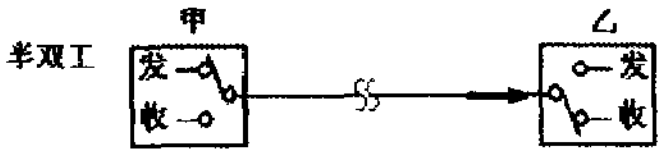


图 11.1.2.2　半双工方式

当计算机主机用串行接口连接显示终端时，在半双工方式中，输入过程和输出过程使用同一通路。有些计算机和显示终端之间采用半双工方式工作，这时，从键盘打入的字符在发送到主机的同时就被送到终端上显示出来，而不是用回送的办法，所以避免了接收过程和发送过程同时进行的情况。

目前多数终端和串行接口都为半双工方式提供了换向能力，也为全双工方式提供了两条独立的引脚。在实际使用时，一般并不需要通信双方同时既发送又接收，像打印机这类的单向传送设备，半双工甚至单工就能胜任，也无须倒向。

11.1.3　同步传送与异步传送

串行传送有两种方式：同步传送方式和异步传送方式。表 11-1-3-1 为其概念及特点对比。

表 11-1-3-1　同步传送方式和异步传送方式对比

	同步传送方式	异步传送方式
基本概念	以数据块（若干个字符）为单位进行传送；在每个数据块（通常称为帧）的开始要加上同步字符，没有信息传送时要加上空字符	以字符为数据传送单位，每个字符都以起始位开始，以停止位结束
同步要求	每一字符中的相邻两位间的时间间隔要相等；而且相邻字符间的时间间隔也要相等	各字符之间允许有间隙；且两个字符之间的间隔不固定
特点	■ 接收方不必对每个字符进行起始与停止的操作，传送效率高 ■ 传送设备复杂，双方时钟允许误差较小 ■ 可用于点和多点之间的数据传送	■ 与同步传送方式相比，传送效率低 ■ 传送设备简单，双方时钟允许一定的误差 ■ 只适用于点到点的数据传送
校验	常用循环冗余码校错；可通过硬件进行纠错或重送	常用奇偶校验码校错；很难重送或纠错

11.1.4　串行通信协议

所谓通信协议是指通信双方的一种约定。约定包括对数据格式、同步方式、传送速度、传送步骤、检纠错方式以及控制字符定义等问题做出统一规定，通信双方必须共同遵守。因此，也叫做通信控制规程，或称传输控制规程，它属于 ISO OSI 七层参考模型中的数据链路层。

目前，采用的通信协议有两类：异步协议和同步协议。同步协议又有面向字符和面向比特以及面向字节计数三种。其中，面向字节计数的同步协议主要用于 DEC 公司的网络体系结构中。

1. OSI 协议和 TCP/IP 协议

（1）OSI 协议

OSI 七层参考模型不是通信标准，它只给出一个不会由于技术发展而必须修改的稳定模型，使有关标准和协议能在模型定义的范围内开发和相互配合。

一般的通信协议只符合 OSI 七层模型的某几层，如：EIA-RS-232-C 实现了物理层。IBM 的 SDLC（同步数据链路控制规程）实现数据链路层。ANSI 的 ADCCP（先进数据通信规程）实现数据链路层。IBM 的 BSC（二进制同步通信协议）实现数据链路层。应用层的电子邮件协议 SMTP 只负责寄信，POP3 只负责收信。

（2）TCP/IP 协议

实现了五层协议：

- 物理层：对应 OSI 的物理层。
- 网络接口层：类似于 OSI 的数据链路层。
- Internet 层：OSI 模型在 Internet 网使用前提出，未考虑网间连接。
- 传输层：对应 OSI 的传输层。
- 应用层：对应 OSI 的表示层和应用层。

2. 串行通信协议

串行通信协议分同步协议和异步协议。

（1）异步通信协议的实例——起止式异步协议

- 起止式异步协议的特点：一个字符一个字符传输，并且传送一个字符总是以起始位开始，以停止位结束，字符之间没有固定的时间间隔要求。每一个字符的前面都有一位起始位（低电平，逻辑值 0），字符本身有 5～7 位数据位组成，接着字符后面是一位校验位（也可以没有校验位），最后是一位，或 1.5 位，或二位停止位，停止位后面是不定长度的空闲位。停止位和空闲位都规定为高电平（逻辑值），这样就保证起始位开始处一定有一个下降沿。这种格式是靠起始位和停止位来实现字符的界定或同步的，故称为起始式协议。传送时，数据的低位在前，高位在后。
- 起 / 止位的作用：起始位实际上是作为联络信号附加进来的，当它变为低电平时，告诉收方传送开始。它的到来，表示下面接着是数据位来了，要准备接收。而停止位标志一个字符的结束，它的出现，表示一个字符传送完毕。这样就为通信双方提供了何时开始收发，何时结束的标志。传送开始前，发收双方把所采用的起止式格式（包括字符的数据位长度，停止位位数，有无校验位以及是奇校验还是偶校验等）和数据传输速率做统一规定。传送开始后，接收设备不断地检测传输线，看是否有起始位到来。当收到一系列的“1”（停止位或空闲位）之后，检测到一个下跳沿，说明起始位出现，起始位经确认后，就开始接收所规定的数据位和奇偶校验位以及停止位。经过处理将停止位去掉，把数据位拼装成一个并行字节，并且经校验后，无奇偶错才算正确地接收一个字符。一个字符接收完毕，接收设备又继续测试传输线，监视“0”电平的到来和下一个字符的开始，直到全部数据传送完毕。
- 由上述工作过程可看到，异步通信是按字符传输的，每传输一个字符，就用起始位来通知收方，以此来重新核对收发双方同步。若接收设备和发送设备两者的时钟频率略

有偏差，这也不会因偏差的累积而导致错位，加之字符之间的空闲位也为这种偏差提供一种缓冲，所以异步串行通信的可靠性高。但由于要在每个字符的前后加上起始位和停止位这样一些附加位，使得传输效率变低了，只有约 80%。因此，起止协议一般用在数据速率较慢的场合（小于 19.2kbit/s）。在高速传送时，一般要采用同步协议。

（2）面向字符的同步协议

- **特点与格式**：这种协议的典型代表是 IBM 公司的二进制同步通信协议(BSC)。它的特点是一次传送由若干个字符组成的数据块，而不是只传送一个字符，并规定了 10 个字符作为这个数据块的开头与结束标志以及整个传输过程的控制信息，它们也叫做通信控制字。由于被传送的数据块是由字符组成，故被称做面向字符的协议。
- **特定字符（控制字符）的定义**：由上面的格式可以看出，数据块的前后都加了几个特定字符。SYN 是同步字符(Synchronous Character)，每一帧开始处都有 SYN，加一个 SYN 的称单同步，加两个 SYN 的称双同步。设置同步字符是起联络作用，传送数据时，接收端不断检测，一旦出现同步字符，就知道是一帧开始了。接着的 SOH 是序始字符（Start Of Header），表示标题的开始。标题中包括起始地址、目的地址和路由指示等信息。STX 是文始字符(Start Of Text)，标志着传送的正文（数据块）开始。数据块就是被传送的正文内容，由多个字符组成。数据块后面是组终字符 ETB（End Of Transmission Block）或文终字符 ETX(End Of Text)，其中，ETB 用在正文很长，需要分成若干个分数据块，分别在不同帧中发送的场合，这时，在每个分数据块后面用文终字符 ETX。一帧的最后是校验码，它对从 SOH 开始到 ETX（或 ETB）字段进行校验，校验方式可以是纵横奇偶校验或 CRC。另外，在面向字符协议中还采用了一些其他通信控制字，它们的名称如表 11-1-4-1 所示。
- **数据透明的实现**：面向字符的同步协议，不像异步起止协议那样，需要在每个字符前后附加起始和停止位，因此，传输效率提高了。同时，由于采用了一些传输控制字，故增强了通信控制能力和校验功能。但也存在一些问题，例如，如何区别数据字符代码和特定字符代码的问题，因为在数据块中完全有可能出现与特定字符代码相同的数据字符，这就会发生误解。比如，正文有个与文终字符 ETX 的代码相同的数据字符，接收端就不会把它当做普通数据处理，而误认为是正文结束，因而产生差错。因此，协议应具有将特定字符作为普通数据处理的能力，这种能力叫做“数据透明”。为此，协议中设置了转移字符 DLE(Data Link Escape)。当把一个特定字符看成数据时，在它前面要加一个 DLE，这样接收器收到一个 DLE 就可预知下一个字符是数据字符，而不会把它当做控制字符来处理了。DLE 本身也是特定字符，当它出现在数据块中时，也要在它前面加上另一个 DLE。这种方法叫字符填充。字符填充实现起来相当麻烦，且依赖于字符的编码。正是由于以上的缺点，故又产生了新的面向比特的同步协议。

表 11-1-4-1　面向字符协议中采用的通信控制字

名 称	ASCII	EBCDIC
序始(SOH)	0000001	00000001
文始（STX）	0000010	00000010
组终(ETB)	0010111	00100110
文终（ETX）	0000011	00000011
同步(SYN)	0010110	00110010
送毕(EOT)	0000100	00110111

续表

名 称	ASCII	EBCDIC
询问(ENQ)	0000101	00101101
确认(ACK)	0000110	00101110
否认（NAK)	0010101	00111101
转义(DLE)	0010000	00010000

（3）面向比特的同步协议

特点与格式：面向比特的协议中最具有代表性的是 IBM 的同步数据链路控制规程 SDLC（Synchronous Data Link Control)，国际标准化组织 ISO(International Standard Organization）的高级数据链路控制规程 HDLC（High Level Data link Control)，美国国家标准协会(American National Standard Institute)的先进数据通信规程 ADCCP(Advanced Data Communication Control Procedure)。这些协议的特点是所传输的一帧数据可以是任意位，而且它是靠约定的位组合模式，而不是靠特定字符来标志帧的开始和结束，故称“面向比特”的协议。

帧信息的分段： SDLC/HDLC 的一帧信息包括以下几个场(Filed)，所有场都从有效位开始传送。

- SDLC/HDLC 标志字符：SDLC/HDLC 协议规定，所有信息传输必须以一个标志字符开始，且以同一个字符结束。这个标志字符是 01111110，称标志场(F)。从开始标志到结束标志之间构成一个完整的信息单位，称为一帧(Frame)。所有的信息是以帧的形式传输的，而标志字符提供了每一帧的边界。接收端可以通过搜索“01111110”来探知帧的开头和结束，以此建立帧同步。
- 地址场和控制场：在标志场之后，可以有一个地址场 A(Address）和一个控制场 C(Control)。地址场用来规定与之通信的次站的地址。控制场可规定若干个命令。SDLC 规定 A 场和 C 场的宽度为 8 位或 16 位。接收方必须检查每个地址字节的第一位，如果为“0”，则后面跟着另一个地址字节；若为“1”，则该字节就是最后一个地址字节。同理，如果控制场第一个字节的第一位为“0”，则还有第二个控制场字节，否则就只有一个字节。
- 信息场：跟在控制场之后的是信息场 I(Information)。I 场包含有要传送的数据，并不是每一帧都必须有信息场。即数据场可以为 0，当它为 0 时，则这一帧主要是控制命令。
- 帧校验信息：紧跟在信息场之后的是两字节的帧校验，帧校验场称为 FC(Frame Check)场或称为帧校验序列 FCS(Frame Check Squence)。SDLC/HDLC 均采用 16 位循环冗余校验码 CRC（Cyclic Redundancy Code)。除了标志场和自动插入的“0”以外，所有的信息都参加 CRC 计算。

11.2　RS-232-C 串口标准

11.2.1　RS-232-C 标准

由于串行通信方式具有使用线路少、成本低，特别是在远程传输时，避免了多条线路特性的不一致而被广泛采用。 在串行通信时，要求通信双方都采用一个标准接口，使不同的设备可以方便地连接起来进行通信。RS-232-C 接口是目前最常用的一种串行通信接口。

RS-232C 标准（协议）的全称是 EIA-RS-232C 标准，其中，EIA(Electronic Industry Association)代表美国电子工业协会，RS（recommended standard）代表推荐标准，232 是标识号，C 代表 RS232 的最新一次修改（1969），在这之前，有 RS232B、RS232A。它是在 1969 年由美国电子工业协会（EIA）联合贝尔系统、 调制解调器厂家及计算机终端生产厂家共同制定的用于串行通信的标准。它的全名是“数据终端设备（DTE）和数据通信设备（DCE）之间串行二进制数据交换接口技术标准”。该标准规定采用一个 25 个脚的 DB25 连接器，对连接电缆和机械、电气特性、信号功能及传送过程加以规定。目前在 IBM PC 机上的 COM1、COM2 接口，就是 RS-232C 接口。

（1）接口的信号内容

RS232C 标准规定设备间使用带“D”型 25 针连接器的电缆通信。DB25 各引脚定义参见图 11.2.1.1。这 25 根引线中，有 20 根用做信号线，其他 3 根未定义用途，2 根备用。

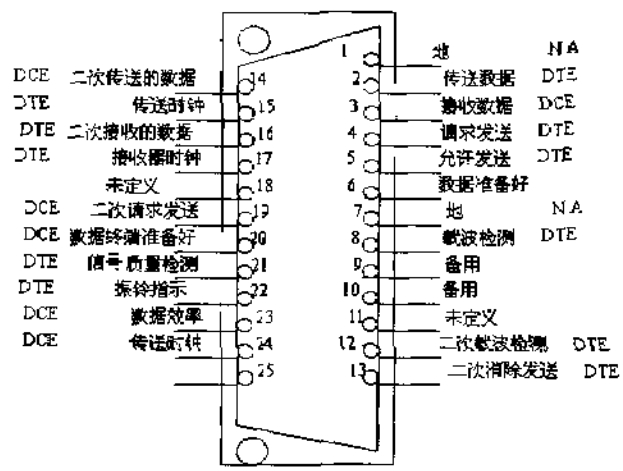


图 11.2.1.1　DB25 引脚定义

实际上，RS-232-C 的 25 条引线中有许多是很少使用的，在计算机与终端通信中一般只使用 3～9 条引线。RS-232-C 最常用的 9 条引线的信号内容见表 11-2-1-1 所示。

表 11-2-1-1　RS-232-C 最常用的 9 条引脚

引脚序号	信号名称	符号	流向	功能
2	发送数据	TXD	DTE→DCE	DTE 发送串行数据
3	接收数据	RXD	DTE←DCE	DTE 接收串行数据
4	请求发送	RTS	DTE→DCE	DTE 请求 DCE 将线路切换到发送方式
5	允许发送	CTS	DTE←DCE	DCE 告诉 DTE 线路已接通可以发送数据
6	数据设备准备好	DSR	DTE←DCE	DCE 准备好
7	信号地			信号公共地
8	载波检测	DCD	DTE←DCE	表示 DCE 接收到远程载波
20	数据终端准备好	DTR	DTE→DCE	DTE 准备好
22	振铃指示	RI	DTE←DCE	表示 DCE 与线路接通,出现振铃

（2）接口的电气特性

EIA-RS-232C 对电器特性、逻辑电平和各种信号线功能都做了规定。

在 TxD 和 RxD 上：逻辑 1(MARK)=-3V～-15V

逻辑 0(SPACE)=+3～+15V

在 RTS、CTS、DSR、DTR 和 DCD 等控制线上：

信号有效（接通，ON 状态，正电压）=+3V～+15V

信号无效（断开，OFF 状态，负电压）=-3V～-15V

以上规定说明了 RS-323C 标准对逻辑电平的定义。对于数据（信息码）：逻辑“1”（传号）的电平低于-3V，逻辑“0”（空号）的电平高于-3V；对于控制信号：接通状态（ON）即信号有效的电平高于+3V，断开状态(OFF)即信号无效的电平低于-3V。也就是当传输电平的绝对值大于 3V 时，电路可以有效地检查出来，介于-3～+3V 之间的电压无意义，低于-15V 或高于+15V 的电压也认为无意义，因此，实际工作时，应保证电平在±(3～15)V 之间。

EIA-RS-232C 与 TTL 转换：EIA-RS-232C 是用正负电压来表示逻辑状态，与 TTL 以高低电平表示逻辑状态的规定不同。因此，为了能够同计算机接口或终端的 TTL 器件连接，必须在 EIA-RS-232C 与 TTL 电路之间进行电平和逻辑关系的变换。实现这种变换的方法可用分立元件，也可用集成电路芯片。目前较为广泛地使用集成电路转换器件，如 MC1488、SN75150 芯片可完成 TTL 电平到 EIA 电平的转换，而 MC1489、SN75154 可实现 EIA 电平到 TTL 电平的转换。MAX232 芯片可完成 TTL←→EIA 双向电平转换。

（3）接口的物理结构

RS-232-C 接口连接器一般使用型号为 DB-25 的 25 芯插头座，通常插头在 DCE 端，插座在 DTE 端。一些设备与 PC 机连接的 RS-232-C 接口，因为不使用对方的传送控制信号，只需三条接口线，即“发送数据”、“接收数据”和“信号地”，所以采用 DB-9 的 9 芯插头座，传输线采用屏蔽双绞线。

（4）传输电缆长度

由于 RS-232C 标准规定在码元畸变小于 4%的情况下，传输电缆长度应为 50 英尺，其实，这个 4%的码元畸变是很保守的，在实际应用中，约有 99%的用户是按码元畸变 10%～20%的范围工作的，所以实际使用中最大距离会远超过 50 英尺，美国 DEC 公司曾规定允许码元畸变为 10%而得出表 11-2-1-2 的实验结果。其中 1 号电缆为屏蔽电缆，型号为 DECP.NO.9107723 。

内有三对双绞线，每对由 22# AWG 组成，其外覆以屏蔽网。2 号电缆为不带屏蔽的电缆。型号为 DECP.NO.9105856-04 是 22#AWG 的四芯电缆。

表 11-2-1-2　DEC 公司的实验结果

波特率	1 号电缆传输距离（英尺）	2 号电缆传输距离（英尺）
110	5000	3000
300	5000	3000
1200	3000	3000
2400	1000	500
4800	1000	250
9600	250	250

注：1 英尺=12 英寸=0.3048 米

11.2.2 RS-232-C 串行通信接线实例

1. 远距离通信

以下是传输距离大于 15m 的通信的例子，一般要加调制解调器 MODEM，因此使用的信号线较多。注意：在图中，DTE 信号为 RS-232-C 信号，DTE 与计算机间的电平转换电路未画出。

（1）采用 MODEM(DCE)和电话网通信时的信号连接

若在双方 MODEM 之间采用普通电话交换线进行通信，除了需要 2～8 号信号线外，还要增加 RI(22 号)和 DTR(20 号)两个信号线进行联络，如图 11.2.2.1 所示。

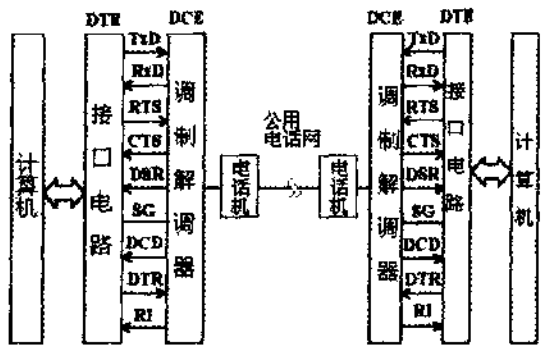


图 11.2.2.1 采用 MODEM(DCE)和电话网通信时的信号连接

DSR、DTR：数传机（DCE）准备好、数据终端（DTE）准备好，只表示设备本身可用。

首先，通过电话机拨号呼叫对方，电话交换台向对方发出拨号呼叫信号，当对方 DCE 收到该信号后，使 RI（振铃信号）有效，通知 DTE，已被呼叫。当对方“摘机”后，两方建立了通信链路。

若计算机要发送数据至对方，首先通过接口电路（DTE）发出 RTS（请求发送）信号。此时，若 DCE（MODEM）允许传送，则向 DTE 回答 CTS（允许发送）信号。一般可直接将 RTS/CTS 接高电平，即只要通信链路已建立，就可传送信号。（RTS/CTS 可只用于半双工系统中做发送方式和接收方式的切换。）

当 DTE 获得 CTS 信号后，通过 TXD 线向 DCE 发出串行信号，DCE（MODEM）将这些数字信号调制成模拟信号（又称载波信号），传向对方。

计算机向 DTE“数据输出寄存器”传送新的数据前，应检查 MODEM 状态和数据输出寄存器为空。当对方的 DCE 收到载波信号后，向对方的 DTE 发出 DCD 信号（数据载波检出），通知其 DTE 准备接收，同时，将载波信号解调为数据信号，从 RXD 线上送给 DTE，DTE 通过串行接收移位寄存器对接收到的位流进行移位，当收到 1 个字符的全部位流后，把该字符的数据位送到数据输入寄存器，CPU 可以从数据输入寄存器读取字符。

（2）采用专用电话线通信

在通信双方的 MODEM 之间采用电话线进行通信，则只要使用 2～8 号信号线进行联络与控制。不需要电话机、振铃信号 RI 和 DTR 信号，其信号线的连接如图 11.2.2.2 那样。

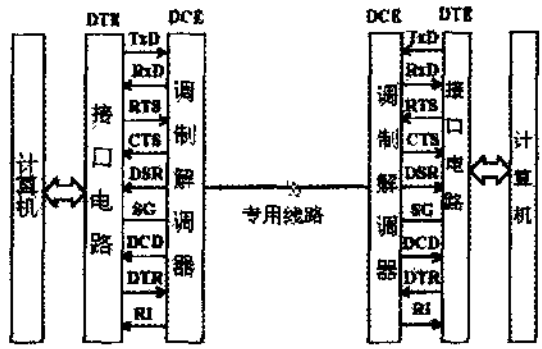


图 11.2.2.2　采用专用电话线通信

2. 近距离通信

当通信距离较近时，可不需要 MODEM，通信双方可以直接连接，这种情况下，只需使用少数几根信号线。最简单的情况，在通信中根本不需要 RS-232C 的控制联络信号，只需三根线（发送线、接收线、信号地线）便可实现全双工异步串行通信。

无 MODEM 时，最大通信距离按如下方式计算：

RS-232C 标准规定：当误码率小于 4%时，要求导线的电容值应小于 2500PF。对于普通导线，其电容值约为 170PF/M。则允许距离 L=2500PF/（170PF/M）=15M 。

这一距离的计算，是偏于保守的，实际应用中，当使用 9600bps，普通双绞屏蔽线时，距离可达 30 米～35 米。

3. 无 MODEM 的最简连线（3 线制）

首先，串口传输数据只要有接收数据针脚和发送针脚就能实现：同一个串口的接收脚和发送脚直接用线相连，两个串口相连或一个串口和多个串口相连。

同一个串口的接收脚和发送脚直接用线相连，对 9 针串口和 25 针串口，均是 2 与 3 直接相连，表 11-2-2-1 为 9 针串口和 25 针串口对应连接方法。

表 11-2-2-1　9 针串口和 25 针串口对应连接方法

9 针—9 针		25 针—25 针		9 针—25 针	
2	3	3	2	2	2
3	2	2	3	3	3
5	5	7	7	5	7

上面表格是对微机标准串行口而言的，还有许多非标准设备，如接收 GPS 数据或电子罗盘数据，只要记住一个原则：接收数据针脚（或线）与发送数据针脚（或线）相连，彼此交叉，信号地对应相接，就能百战百胜。

最后需要说明的是：

- RS-232-C 标准最初是远程通信连接数据终端设备 DTE(Data Terminal Equipment)与数据通信设备 DCE（Data Communication Equipment)而制定的。因此这个标准的制

定，并未考虑计算机系统的应用要求。但目前它又广泛地被借用于计算机（更准确地说，是计算机接口）与终端或外设之间的近端连接标准。显然，这个标准的有些规定和计算机系统是不一致的，甚至是相矛盾的。有了对这种背景的了解，我们对RS-232C 标准与计算机不兼容的地方就不难理解了。

- RS-232C 标准中所提到的“发送”和“接收”，都是站在 DTE 立场上，而不是站在 DCE 的立场来定义的。由于在计算机系统中，往往是 CPU 和 I/O 设备之间传送信息，两者都是 DTE，因此双方都能发送和接收。

11.3 RS-422/485 串口标准

11.3.1 概述

由于 RS-232-C 接口标准出现较早，所以难免有不足之处，主要有以下四点：

- 接口的信号电平值较高，易损坏接口电路的芯片，又因为 TTL 电平不兼容,故需使用电平转换电路方能与 TTL 电路连接。
- 传输速率较低，在异步传输时，波特率为 20kbps。
- 接口使用一根信号线和一根信号返回线而构成共地的传输形式，这种共地传输容易产生共模干扰，所以抗噪声干扰性弱。
- 传输距离有限，最大传输距离标准值为 50 英尺，实际上也只能用在 50 米左右。

RS-422 由 RS-232 发展而来，它是为弥补 RS-232 之不足而提出的。为改进 RS-232 通信距离短、速率低的缺点，RS-422 定义了一种平衡通信接口，将传输速率提高到 10Mb/s，传输距离延长到 4000 英尺（速率低于 100kb/s 时），并允许在一条平衡总线上连接最多 10 个接收器。RS-422 是一种单机发送、多机接收的单向、平衡传输规范，被命名为 TIA/EIA-422-A 标准。

为扩展应用范围，EIA 又于 1983 年在 RS-422 基础上制定了 RS-485 标准，增加了多点、双向通信能力，即允许多个发送器连接到同一条总线上，同时增加了发送器的驱动能力和冲突保护特性，扩展了总线共模范围，后命名为 TIA/EIA-485-A 标准。由于 EIA 提出的建议标准都是以“RS”作为前缀，所以在通信工业领域，仍然习惯将上述标准以 RS 做前缀称谓。

RS-485 具有以下特点：

- RS-485 的电气特性：逻辑“1”以两线间的电压差为+（2～6） V 表示；逻辑“0”以两线间的电压差为-（2～6）V 表示。接口信号电平比 RS-232-C 降低了，就不易损坏接口电路的芯片， 且该电平与 TTL 电平兼容，可方便与 TTL 电路连接。
- RS-485 的数据最高传输速率为 10Mbps。
- RS-485 接口是采用平衡驱动器和差分接收器的组合，抗共模干扰能力增强，即抗噪声干扰性好。
- RS-485 接口的最大传输距离标准值为 4000 英尺，实际上可达 3000 米，另外 RS-232-C 接口在总线上只允许连接 1 个收发器， 即单站能力。而 RS-485 接口在总线上允许连接多达 128 个收发器,即具有多站能力,这样用户可以利用单一的 RS-485 接口方便地建立起设备网络。

因 RS-485 接口具有良好的抗噪声干扰性，长的传输距离和多站能力等上述优点，所以使其成为首选的串行接口。因为 RS485 接口组成的半双工网络，一般只需二根连线，所以 RS485 接口均采用屏蔽双绞线传输。RS485 接口连接器采用 DB-9 的 9 芯插头座，与智能终端 RS485 接口采用 DB-9（孔），与键盘连接的键盘接口 RS485 采用 DB-9（针）。

11.3.2　RS-422 与 RS-485 串行接口标准

1．平衡传输

RS-422、RS-485 与 RS-232 不一样，数据信号采用差分传输方式，也称做平衡传输，它使用一对双绞线，将其中一线定义为 A，另一线定义为 B，如图 11.3.2.1 所示。

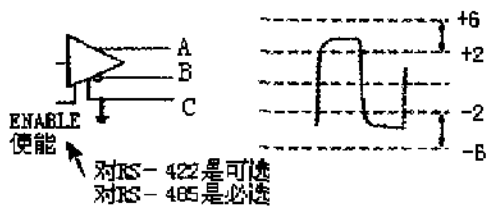


图 11.3.2.1　差分传输方式

通常情况下，发送驱动器 A、B 之间的正电平在+2～+6V，是一个逻辑状态，负电平在-2～6V，是另一个逻辑状态。另有一个信号地 C，在 RS-485 中还有一“使能”端，而在 RS-422 中这是可用可不用的。“使能”端是用于控制发送驱动器与传输线的切断与连接。当“使能”端起作用时，发送驱动器处于高阻状态，称做“第三态”，即它是有别于逻辑“1”与“0”的第三态。

接收器也做与发送端相对的规定，收、发端通过平衡双绞线将 AA 与 BB 对应相连，当在接收端 AB 之间有大于+200mV 的电平时，输出正逻辑电平，小于-200mV 时，输出负逻辑电平。接收器接收平衡线上的电平范围通常在 200mV 至 6V 之间。参见图 11.3.2.2。

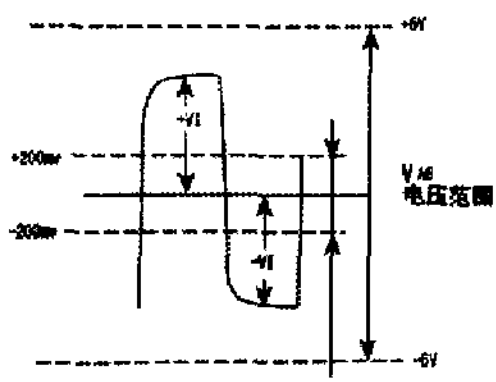


图 11.3.2.2　接收平衡线上的电平范围

2. RS-422 电气规定

RS-422 标准全称是“平衡电压数字接口电路的电气特性”，它定义了接口电路的特性。图 11.3.2.3 是典型的 RS-422 四线接口。实际上还有一根信号地线，共 5 根线。图 11.3.2.4 是其 DB9 连接器引脚定义。由于接收器采用高输入阻抗和发送驱动器比 RS232 更强的驱动能力，故允许在相同传输线上连接多个接收节点，最多可接 10 个节点。即一个主设备（Master），其余为从设备（Salve），从设备之间不能通信，所以 RS-422 支持点对多的双向通信。接收器输入阻抗为 4k，故发端最大负载能力是 10×4k+100Ω（终接电阻）。RS-422 四线接口由于采用单独的发送和接收通道，因此不必控制数据方向，各装置之间任何必需的信号交换均可以按软件方式（XON/XOFF 握手）或硬件方式（一对单独的双绞线）实现。

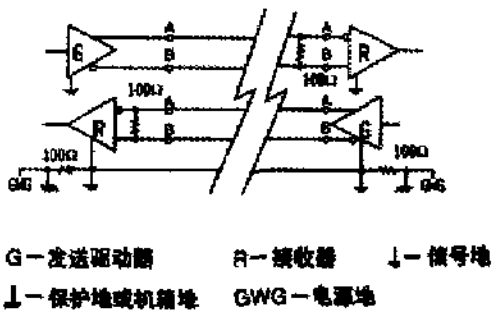


图 11.3.2.3　典型的 RS-422 四线接口

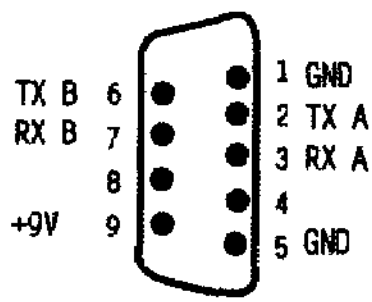


图 11.3.2.4　DB9 连接器引脚定义

RS-422 的最大传输距离为 4000 英尺（约 1219 米），最大传输速率为 10Mb/s。其平衡双绞线的长度与传输速率成反比，在 100kb/s 速率以下，才可能达到最大传输距离。只有在很短的距离下才能获得最高速率传输。一般 100 米长的双绞线上所能获得的最大传输速率仅为 1Mb/s。

RS-422 需要一终接电阻，要求其阻值约等于传输电缆的特性阻抗。在短距离传输时可不需终接电阻，即一般在 300 米以下不需终接电阻。终接电阻接在传输电缆的最远端。

3. RS-485 电气规定

由于 RS-485 是从 RS-422 基础上发展而来的，所以 RS-485 许多电气规定与 RS-422 相仿。

如都采用平衡传输方式、都需要在传输线上接终接电阻等。RS-485 可以采用二线与四线方式，二线制可实现真正的多点双向通信。

而采用四线连接时，与 RS-422 一样只能实现点对多的通信，即只能有一个主（Master）设备，其余为从设备，但它比 RS-422 有改进，无论四线还是二线连接方式，总线上可多接到 32 个设备。

RS-485 与 RS-422 的不同还在于其共模输出电压是不同的，RS-485 是-7V 至+12V 之间，而 RS-422 在-7V 至+7V 之间，RS-485 接收器最小输入阻抗为 12k，而 RS-422 是 4k。RS-485 满足所有 RS-422 的规范，所以 RS-485 的驱动器可以用在 RS-422 网络中应用。

RS-485 与 RS-422 一样，其最大传输距离约为 1219 米，最大传输速率为 10Mb/s。平衡双绞线的长度与传输速率成反比，在 100kb/s 速率以下，才可能使用规定最长的电缆长度。只有在很短的距离下才能获得最高速率传输。一般 100 米长双绞线最大传输速率仅为 1Mb/s。

RS-485 需要 2 个终接电阻，其阻值要求等于传输电缆的特性阻抗。在短距离传输时可不需终接电阻，即一般在 300 米以下不需终接电阻。终接电阻接在传输总线的两端。

11.3.3　RS-422 与 RS-485 的网络安装注意要点

RS-422 可支持 10 个节点，RS-485 支持 32 个节点，因此多节点构成网络。网络拓扑一般采用终端匹配的总线型结构，不支持环型或星型网络。在构建网络时，应注意如下几点：

- 采用一条双绞线电缆做总线，将各个节点串接起来，从总线到每个节点的引出线长度应尽量短，以便使引出线中的反射信号对总线信号的影响最低。图 11.3.3.1 所示为实际应用中常见的一些错误连接方式（a，c，e）和正确的连接方式（b，d，f）。a，c，e 这三种网络连接尽管不正确，但在短距离、低速率仍可能正常工作，但随着通信距离的延长或通信速率的提高，其不良影响会越来越严重，主要原因是信号在各支路末端反射后与原信号叠加，会造成信号质量下降。
- 应注意总线特性阻抗的连续性，在阻抗不连续点，就会发生信号的反射。下列几种情况易产生这种不连续性：总线的不同区段采用了不同电缆，或某一段总线上有过多收发器紧靠在一起安装，再者是过长的分支线引出到总线。

  总之，应该提供一条单一、连续的信号通道作为总线。

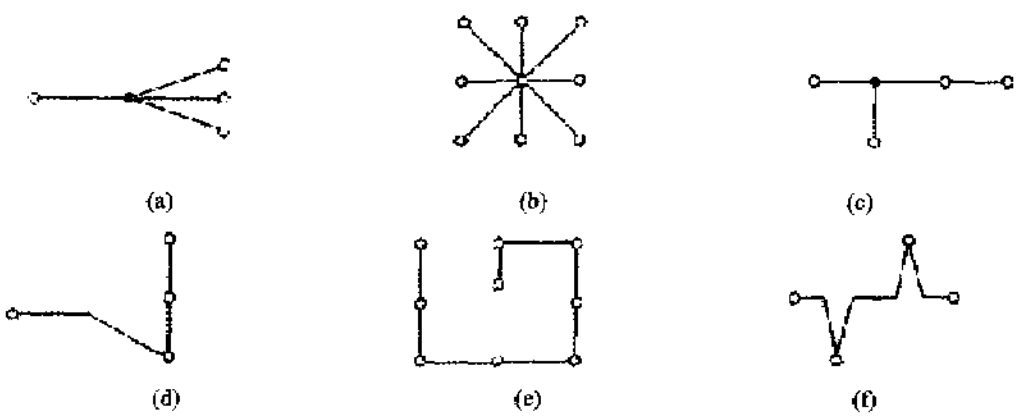


图 11.3.3.1　常见的正误连接方式

11.3.4　RS-232、RS422、RS485 电气参数对比

RS-232、RS422、RS485 有关电气参数对比如表 11-3-4-1 所示。

表 11-3-4-1　电气参数对比表

规定		RS232	RS422	R485
工作方式		单端	差分	差分
节点数		1 收 1 发	1 发 10 收	1 发 32 收
最大传输电缆长度		50 英尺	400 英尺	400 英尺
最大传输速率		20Kb/s	10Mb/s	10Mb/s
最大驱动输出电压		+/-25V	-0.25V～+6V	-7V～+12V
驱动器输出信号电平(负载最小值)	负载	+/-5V～+/-15V	+/-2.0V	+/-1.5V
驱动器输出信号电平(空载最大值)	空载	+/-25V	+/-6V	+/-6V
驱动器负载阻抗(Ω)		3k～7k	100	54
摆率(最大值)		30V/μs	N/A	N/A
接收器输入电压范围		+/-15V	-10V～+10V	-7V～+12V
接收器输入门限		+/-3V	+/-200mV	+/-200mV
接收器输入电阻(Ω)		3k～7k	4k(最小)	≥12k
驱动器共模电压			-3V～+3V	-1V～+3V
接收器共模电压			-7V～+7V	-7V～+12V

11.4　串口调试注意事项

串口调试注意事项如下：

- 不同编码机制不能混接，如 RS232C 不能直接与 RS422 接口相连，市面上有各种转换器卖，必须通过转换器才能连接；
- 线路焊接要牢固，以免因接线问题误事；
- 串口调试时，准备一个好用的调试工具，如串口调试助手等，有事半功倍之效果；
- 强烈建议不要带电插拔串口，插拔时至少有一端是断电的，否则串口易损坏。

11.5　常用数据校验法

由于传输距离、现场状况等诸多可能出现的因素影响，计算机与受控设备之间的通讯数据常会发生无法预测的错误。为了防止错误所带来的影响，一般在通讯时采取数据校验的办法，而奇偶检验与循环冗余码校验就是其中最常用的校验方法。串行数据在传输过程中，由于干扰可能引起信息的出错，出现“误码”，我们把如何发现传输中的错误，叫“检错”。发现错误后，如何消除错误，叫“纠错”。

11.5.1　奇偶校验

最简单的检错方法是“奇偶校验”，即在传送字符的各位之外，再传送 1 位奇/偶校验位。可采用奇校验或偶校验。

奇校验：所有传送的数位（含字符的各数位和校验位）中，“1”的个数为奇数，如：

1 0110，0101

0 0110，0001

偶校验：所有传送的数位（含字符的各数位和校验位）中，“1”的个数为偶数，如：

1 0100，0101

0 0100，0001

奇偶校验能够检测出信息传输过程中的部分误码（1 位误码能检出，2 位及 2 位以上误码不能检出），同时，它不能纠错。在发现错误后，只能要求重发。但由于其实现简单，仍得到了广泛使用。有些检错方法，具有自动纠错能力。如循环冗余码（CRC）检错等。

11.5.2　循环冗余码校验

循环冗余码校验英文名称为 Cyclical Redundancy Check，简称 CRC（有些书中 CRC 专指循环冗余校验码（Cyclic Redundancy Code），本处对两者不做区分）。它是利用除法及余数的原理来做错误侦测（Error Detecting）的。实际应用时，发送装置计算出 CRC 值并随数据一同发送给接收装置，接收装置对收到的数据重新计算 CRC，并与收到的 CRC 相比较，若两个 CRC 值不同，则说明数据通讯出现错误。

根据应用环境与习惯的不同，CRC 又可分为以下几种标准：

①CRC-12 码；

②CRC-16 码；

③CRC-CCITT 码；

④CRC-32 码。

CRC-12 码通常用来传送 6-bit 字符串。CRC-16 及 CRC-CCITT 码则用是来传送 8-bit 字符，其中 CRC-16 为美国采用，而 CRC-CCITT 为欧洲国家所采用。CRC-32 码大都被采用在一种称为 Point-to-Point 的同步传输中。

下面以最常用的 CRC-16 为例来说明其生成过程。

CRC-16 码由两个字节构成，在开始时 CRC 寄存器的每一位都预置为 1，然后把 CRC 寄存器与 8-bit 的数据进行异或，之后对 CRC 寄存器从高到低进行移位，在最高位（MSB）的位置补零，而最低位（LSB，移位后已经被移出 CRC 寄存器）如果为 1，则把寄存器与预定义的多项式码进行异或，否则如果 LSB 为零，则无需进行异或。重复上述的由高至低的移位 8 次，第一个 8-bit 数据处理完毕，用此时 CRC 寄存器的值与下一个 8-bit 数据异或并进行如前一个数据似的 8 次移位。所有的字符处理完成后，CRC 寄存器内的值即为最终的 CRC 值。

下面为 CRC 的计算过程：

（1）设置 CRC 寄存器，并给其赋值 FFFF(hex)。

（2）将数据的第一个 8-bit 字符与 16 位 CRC 寄存器的低 8 位进行异或，并把结果存入 CRC 寄存器。

（3）CRC 寄存器向右移一位，MSB 补零，移出并检查 LSB。

（4）如果 LSB 为 0，重复第三步；若 LSB 为 1，CRC 寄存器与多项式码相异或。

（5）重复第 3 与第 4 步，直到 8 次移位全部完成。此时一个 8-bit 数据处理完毕。

（6）重复第 2 至第 5 步，直到所有数据全部处理完成。

（7）最终 CRC 寄存器的内容即为 CRC 值。

11.6 串口连接和 TCP/IP 连接对比

双机连接通常有两种连接方法:串口连接和 TCP/IP 连接。

与使用基于 TCP/IP 的网络通信相比，使用串口 RS232 具有如下优点：

- 价格低
- 易于安装
- 可用性好

与使用 RS232 串口通信相比，基于 TCP/IP 的网络通信的优点是：

- 数据传输能力强，使用以太网卡的网络通信传输数据速率可达 10Mb/s，而串口通信由于 RS232 端口的限制，其速率最大为 115kb/s；
- 支持远距离的双机通信，使用中继器和网关时，无须考虑双机距离，并可以跨因特网进行通信。
- 可连入局域网。用 TCP/IP 方式连接到局域网上,多台微机都可以加入双机通信系统,而串口方式下只能两台微机互联。

11.7 现场总线与 RS-232、RS-485 的本质区别

PC 与智能设备通信多借助 RS232、RS485、以太网等方式，主要取决于设备的接口规范。但 RS232、RS485 只能代表通信的物理介质层和链路层，如果要实现数据的双向访问，就必须自己编写通信应用程序，但这种程序多数都不符合 ISO/OSI 的规范，只能实现较单一的功能，适用于单一设备类型，程序不具备通用性。在 RS232 或 RS485 设备连成的设备网中，如果设备数量超过 2 台，就必须使用 RS485 做通信介质，RS485 网的设备间要想互通信息，只有通过“主（Master）”设备中转才能实现，这个主设备通常是 PC，而这种设备网中只允许存在一个主设备，其余全部是从（Slave）设备。

而现场总线技术是以 ISO/OSI 模型为基础的，具有完整的软件支持系统，能够解决总线控制、冲突检测、链路维护等问题。现场总线设备自动成网，无主/从设备之分或允许多主存在。在同一个层次上，不同厂家的产品可以互换，设备之间具有互操作性。

11.8 MODEM 通信技术

11.8.1 MODEM 的基本工作原理

调制解调器（MODEM）其实是数据调制器（MODulator）与数据解调器（DEModulator）的合称。它是计算机与计算机之间通过电话线传输信息时不可缺少的设备。图 11.8.1.1 即为 MODEM 连接的计算机间通信示意图。

在典型的数据通信系统中，调制解调器的功能是做数据通信设备(DCE)用，将数据终端设备(DTE)所产生的数字信号，经数字调变转换成适宜线路传输的带宽小于 4kHz 的模拟信号，然后从一般的电话线或数据专线传送到远方的数据通信设备去，这就是“调制”，完成这部分功能的设备称为调制器；远端的调制解调器将电话线传来的模拟信号经解调及数字处理

技巧的处理，恢复成原来的二进制数字信号，这就是“解调”，完成这部分功能的设备则称为解调器。调制解调器正是将调制和解调功能集成在一起，构成一个设备。因此，确切地说，MODEM 是一种专用的、以完成计算机发出的数字信号与电话线路所传输的音频信号相互转换的设备，它是远程的计算机和网络相连所需的设备。

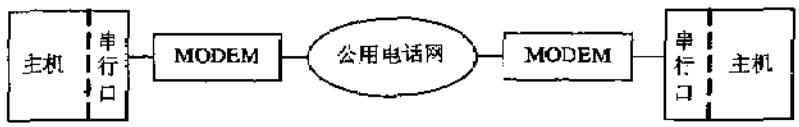


图 11.8.1.1　MODEM 连接的计算机间通信

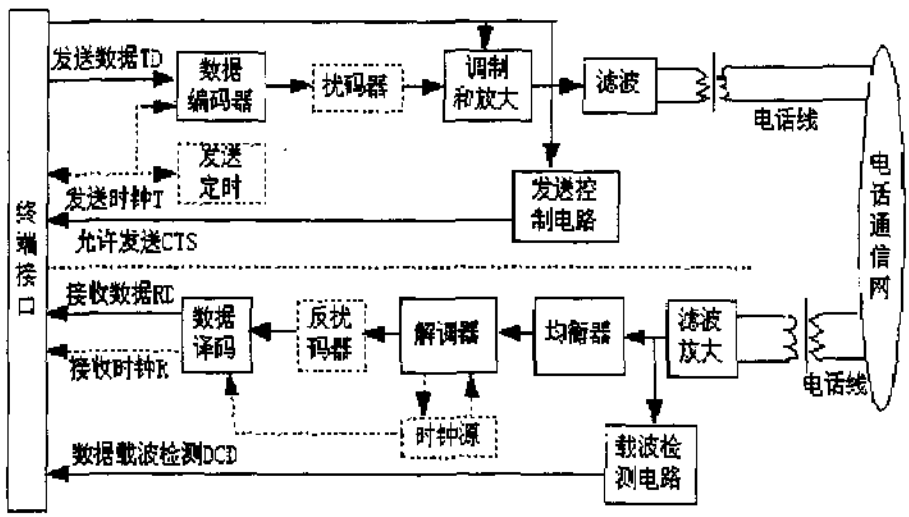


图 11.8.1.2　MODEM 的基本工作原理

图 11.8.1.2 为 MODEM 的基本工作原理。其中，数据终端设备 DTE（串口）与 MODEM 的发送工作流程如下：

- DTE 向 MODEM 发出请求发送信号 RTS。
- MODEM 准备好发送后，向 DTE 返回允许发送信号 CTS。
- DTE 收到允许发送信号 CTS 后，按规定的串行数据协定向 MODEM 通过串行发送口 TD 发出串行数据。
- MODEM 对串行发送口 TD 发来的二进制发送数据进行调制，并发往电话线，直至 RTS 发送请求撤除。

接收工作流程如下：

- MODEM 从电话线上收到有效模拟数据信号后，由载波检测电路向 DTE 发出检测到有效载波信号 DCD。
- DTE 收到有效载波信号 DCD 后，准备接收数据。
- MODEM 将收到的模拟数据信号解调转换为二进制数据送给 DTE 的串口，由 DTE 串行接收。

11.8.2　MODEM 的功能

MODEM 最基本的功能是进行数据通信，现在市面上的 MODEM 大部分还具有以下的功能：

（1）传真功能。目前市场上的高速 MODEM 一般都具备内部传真功能，所不同的是，有时用户需要安装专门的传真软件。带传真功能的 MODEM 有两个速度：一是传真的传送速度；另一个是数据发送速度。需要明确的是，传真功能必须能够以与 MODEM 一样的工作速度发送传真。

（2）语音功能。具备语音功能的 MODEM 可以在同一电话线上传送数据和声音，从而实现个人语音信箱与电话答录等功能。

（3）错误纠正与数据压缩。错误纠正是指侦测出数据错误时，通知对方重新发送数据。数据压缩是指传输时先压缩再传送，以便增加传输量，减少传输时间。

（4）语音数据同传功能（SVD）。SVD 表示语音数据同传功能，即允许用户在发送数据的时候，可以使用同一条线路进行自由的通话，当然在通话的时候，数据的传输速率会降低的。

（5）全双工免提电话功能（FDSP）。现在有的 MODEM 后面不仅有电话插孔，还有麦克风和音箱插孔，它可以不用声卡也能打网络电话。带有 FDSP 功能的 MODEM，可以使你在与对方打电话的同时腾出双手做其他的事情。它与普通的电话机的免提功能不同的是，FDSP 功能允许通话双方同时通话，互不影响。而普通的电话机的免提却做不到这点。

11.8.3　MODEM 的分类

MODEM 在核心结构上主要由处理器和“数据泵”组成。处理器负责 MODEM 的指令控制，“数据泵”负责 MODEM 的底层算法。如果 MODEM 的处理器和“数据泵”全部在卡上实现，则这种 MODEM 卡便是通常所说的“硬猫”。它最主要的特点是不使用计算机主机的资源，可以在 DOS 下使用。市场主要有三大类调制解调器。

1．外置式

外置 MODEM 是将 MODEM 的电路板封装在一个盒子里，外置 MODEM 的造型也是五花八门，由于它的成本较内置 MODEM 高，因此它的价格也比同类型的内置 MODEM 要高。

外置 MODEM 按所采用的接口不同又可分为：串口 MODEM 和 USB 接口 MODEM。串口 MODEM 有独立的电源进行供电，通过串行电缆与计算机的串行口（RS-232）相连接，不用对中断和 COM 口进行设置，安装比较方便，同时，MODEM 的面板上一般有 8 个指示灯，可以显示 MODEM 的工作状况。USB MODEM 的个头比较轻巧，它是通过一根 USB 连线与计算机相连接的，提供高达 12Mbps 的数据带宽，支持即插即用和热插拔的特性，安装十分方便快捷，对计算机操作系统的兼容性也大大优于其他产品。

外置 MODEM 有如下优点。

（1）在前面板上有 LED 指示灯，这些指示灯可用于显示 MODEM 的工作状态，如接收数据、发送数据、准备就绪等，状态观测比较直观，从外接 MODEM 的液晶显示器上可以看到当前是否连通，若连通，则还可显示连接的速度、通信协议、采用何种纠错及压缩协议、

线路的质量等信息，还可以借助于面板上的菜单选择键设置 MODEM 的各种参数、电话号码，以及进行拨号。具体是：

- MR(MODEM 就绪)：当 MODEM 加电时，此灯亮。
- TR(终端就绪)：当 RS - 232DTR 信号给出时，此灯亮。
- SD(发送数据)：本地 MODEM 向远程 MODEM 传送数据时，此灯闪烁。
- RD(接收数据)：本地 MODEM 从远程 MODEM 接收数据时，此灯闪烁。
- OH(摘机)：当 MODEM 摘机时，此灯亮，MODEM 挂机后，此灯灭。
- CD(载波检测)：当本地 MODEM 从远程 MODEM 检测到数据载波信号时，此灯亮。
- AA(自动应答)：当 MODEM 被设置成自动应答方式时，此灯亮。

通过这些指示灯，用户就能够随时掌握 MODEM 的工作情况，便于维护。

（2）如果 MODEM 发生异常而需要 MODEM 复位时，只需开、关 MODEM 就可完成对 MODEM 的复位。

（3）不占用计算机内部的扩展槽。随着多媒体和网络时代的到来，声卡、解压卡、网卡、视频卡等各种插卡都要占用计算机内部的扩展槽。

（4）安装简便，不用打开机箱，易于拆卸。

（5）参数设置比较方便。除了可由通信软件利用 AT 命令实现外，通常还可以通过前面板上的液晶显示器和几个功能键来完成。总之，外接 MODEM 在使用和维护上是十分方便的。

（6）与内置 MODEM 相比，拥有不受机箱内各种连线产生的电磁干扰，不受超频影响。对计算机的 CPU 档次几乎没有要求，不会引起中断、地址冲突，速度比内置 MODEM 快一些。

缺点是：

（1）需要一段电缆和计算机连接，并占用一个计算机串口。

（2）需要一个单独的电源为其供电。

（3）在价格上比相同功能的内置式 MODEM 稍高一些。

2. 内置式

内置 MODEM 只是利用电脑 CPU 强大的运算能力，用软件来接替原来 MODEM 控制模块的功能。这么做的首要目的是省掉 MODEM 的控制芯片及相关电路，从而降低制造成本；另一个目的是更高效地利用系统资源。由于减少了 MODEM 卡上的电子元件，软 MODEM 还节约能源和减少发热量（这一点对便携式电脑来说意义很大）。当然，MODEM 本身的信号处理模块是无法用软件代替的。

内置 MODEM，又称卡式 MODEM，是直接插在主板相应插槽上使用，由主板的电源直接供电。内置的 MODEM 本身包含了串行端口，不需要计算机提供串行端口，制造工艺也比较简单，成本低，所以售价相对于外置来说比较便宜。但是它需要占用主板上的一个扩展槽，并且要对中断和 COM 口进行设置，且安装比较麻烦。

内置 MODEM 按所采用的接口不同又可分为：ISA 接口、PCI 接口、PCMCIA 接口、AMR 接口、CNR 接口与 ACR 接口等。ISA 接口主要应用在早期的 MODEM 上，现在它已逐渐被 PCI 接口的 MODEM 所取代了。目前比较普遍使用的是 PCI 接口，PCI 的 MODEM 可以借助于 PCI 插槽的带宽，而完成 ISA 所无法完成的高速数据运算。PCMCIA 接口应用在笔记本电脑上。PCMCIA 卡除了轻巧、方便携带外，也支持“热插拔”，所以 PCMCIA 规格的设备可

于电脑开机状态时安装插入，并能自动通知操作系统做设备的更新，省去不少安装的麻烦。

优点是：

（1）节省了不必要的电缆连接以及一个计算机串口。

（2）不需要额外的电源进行单独供电，可利用计算机内部电源。

（3）在价格上相对于外置式 MODEM 便宜一些。

（4）内置 MODEM 一般都自带串行接口芯片，而不使用计算机的串行接口电路，因此对那些未带串口的计算机是比较方便的。

（5）内置 MODEM 不占地方，不易受到物理的损坏。

缺点是：

（1）无法观察到 MODEM 的工作状态，而且在使用 MODEM 发生意外情况需要对 MODEM 复位时，通常只能重新启动计算机，这样造成使用上的不便。

（2）占用计算机内部的一个扩展槽。

（3）安装起来有些麻烦，需要打开机箱，有时会发生与 COM1 或 2 口冲突。

（4）不能用手工方法随时设置和修改参数，查错和更换都比较麻烦。

3. PCMCIA 型

适用于笔记本电脑。这种 MODEM 是介于以上两者之间的一种半软半硬的 MODEM。这种 MODEM 没有处理器，却具有硬的“数据泵”，复杂数据算法在卡上实现，简单的控制命令交给计算机处理。这样既可少占用主机资源，又可节省硬件成本，是一种折中方案。内置 MODEM 必须借助 CPU 来完成对通讯数据流的控制，因此，它会占用 CPU。至于内置 MODEM 的速度，一般都略低于外置 MODEM，但差距并不太明显。

现在还有一种 AMR 接口的 MODEM，AMR（Audio/MODEM Riser，声音/调制解调器插卡）是一套开放工业标准，它定义的扩展卡可同时支持声音和 MODEM 功能。采用这种设计，可有效地降低成本，同时解决主板集成声音与 MODEM 子系统后在功能上的一些限制，通过附加的解码器，可以实现软件音频功能和软件调制解调器功能。由于存在电磁干扰以及另外一些不利的因素，MODEM 最重要的模拟 I/O（编码/译码器和 DAA）电路暂时还不能直接做到主板上。Intel 公司之所以制定这套 AMR 规则，很重要的一个目的就是为解决这个问题，将模拟 I/O 电路转移到单独的插卡中，其他部件则留在主板。

11.8.4 MODEM 的安装

1. 外置式调制解调器的安装

将 RS-232 连接线的一端插入 MODEM 的 RS-232 接插口，另一端接入电脑的串行端口；将 12V 或 9V 电源插入电源接口；将电话机用一段电话线与电脑的 Phone 接口连接；将电话线的一端接入 MODEM 的 Line 接口，另一端与电信局提供的电话插座相连接；打开 MODEM 电源开关。至此，硬件安装完毕。

2. 内置式 MODEM 的安装

（1）如果 Fax/MODEM 卡上没有任何跳线设置，则串行端口、I/O 地址以及中断号(IRQ)

是缺省的。这时可以直接安装，但必须避免与其他设备(如声卡、鼠标、解压卡等)使用的系统资源冲突，否则设备将工作不正常。

（2）如果 Fax/MODEM 卡上有跳线设置。在 Windows 2000 中的设置方法是：单击“开始→设置→控制面板”，弹出“控制面板属性”窗口。双击“系统”图标，弹出“系统属性”对话框。单击“硬件”标签，再单击“设备管理器”选项，进入设备管理器，从中可以查看系统资源的使用情况。单击“端口(COM 和 LPT)”旁边的加号，列出系统中的所有串、并口资源。选择一个没有使用的 COM 端口，并确定未被使用的 I/O 地址和中断号(IRQ)。

（3）根据上一步所做的设置，调整 Fax/MODEM 卡的跳线开关。

（4）关闭电脑，打开机箱，在主板上找到空的扩展槽。

（5）将 Fax/MODEM 插入扩展槽，拧紧螺丝，盖上机箱。

（6）将电话线与 Fax/MODEM 卡的接口相连接。

3. 确定 MODEM 是否安装正确

（1）在“调制解调器属性”对话框中，切换到“诊断”选项。

（2）在“端口”列表中选择与调制解调器相连的端口，然后单击“其他信息”按钮，不久可以得到一个检测报告，从中可以判断 MODEM 是否正确安装。

4. MODEM 连接完成后，需要使用某种通信软件来实现数据传输

现对使用中的通信软件分类简述如下。

（1）专用的通信软件

这类通信软件通常是为某种专门的应用（如证券交易、国家信息中心的数据查询、银行信用卡业务联网等）设计的，这类通信软件的发放或出售机构一般对 MODEM 的选购、MODEM 参数的设置和通信软件的使用等方面做出详细的指导或说明。因此在使用时，只要根据要求认真操作即可。

（2）MODEM 随机配发的通信软件

不少 MODEM 厂家都随机配发一张通信软件的软盘（如 Hayes MODEM 就随机配发了通信软件），并附有详细的使用说明书或在线文档。这类通信软件一般都提供文件发送和接收的功能，有的还可以收发传真、处理数字化语音等。由于这类通信软件是由 MODEM 生产厂家自己设计的，因此不存在与 MODEM 不兼容的问题，可在仔细阅读说明书或在 MODEM 销售代理商的指导下正确使用。这类软件的共同缺点就是功能较少，不一定能满足你的实际需要。

（3）通用的通信软件

如 Foxmail 收发电子邮件、IE 浏览器、FTP 传输文件、Windows 95 以上版本所带的传真程序等，这类通信软件不是为某种专门的应用或某种牌子的 MODEM 设计的，其共同特点是功能丰富、适应性强，可在各种应用环境中灵活使用。像 FTP 软件，支持按多种文件传输协议和以各种不同的速度进行文件传输。可以上传文档、图片、语音等信息，也可下载文件、图片、语音等信息。

远程访问，可以浏览远程服务器上的各种信息，并且可以通过 MODEM 控制远方的计算机。当然，也可以通过 MODEM 多进程通信，即可以在一台计算机打开几个窗口来访问不同地域、不同内容的远方终端和近程通信。这些功能的实现，一方面是由于有强大的多任务

操作系统的实现，另一方面是由于 MODEM 性能的不断提高。

11.8.5　MODEM V.92 标准介绍

1998 年 2 月，国际电信联合会（ITU）为了将 X2 和 K56FLEX 撮合在一起，正式推出了 56K 标准，所有的 56K MODEM 的制造商必须放弃自己的 56K 规范，转向统一的 ITU 制定的 56K MODEM 国际标准，该标准就是大名鼎鼎的 V.90 标准。V.90 协议让 MODEM 能够在标准公用电话交换网（PSTN）上以 56K 的理论速率接收数据。尽管 DSL 与 Cable 技术正在蚕食拨号 MODEM 的市场，但由于模拟式 MODEM 具有技术工艺成熟、结构简单、价格低廉、使用方便、速度快、性价比高等诸多优点，因此仍在市场中占主导地位。随着电讯技术的蓬勃发展，用户对 MODEM 的质量要求也越来越高，而现在世界发达国家的网线都得到了大幅度的改进。国际电信联合会决定采取措施加快 MODEM 发展的战略，再次宣布升级其 56K MODEM 的 V.90 标准，于 7 月 3 日前后正式推出了新的 MODEM 标准 V.92。

与 V.90 相比，V.92 标准增加了四大功能：

（1）QuickConnect 的快速连线功能

新型的 V.92 MODEM 与 V.90 MODEM 相比，首先提高了 V.34 的握手速度（就是连通网络前的那段时间），使用户快速登录。新的 V.92 技术，最主要的就是具有 QuickConnect 的快速连线功能，V.90 MODEM 大约需要 20 多秒的时间才能完成整个连线动作，现在新的 V.92 标准可以缩短到 8 秒钟左右。

（2）网络呼叫等待（MODEM-On-Hold）

V.92 MODEM 还有 MODEM-on-hold 的暂停功能，当正在使用 MODEM 时，如果有电话打进来，可最长维持 16 分钟的 MODEM 连接状态。我们可以在 MODEM 连接状态下或在拨号连接过程中暂停拨号或连接，以接听或拨打电话，通话完毕后不需要再重新连线。而 V.90 MODEM 则会切断连接，再重新拨号上网。网络呼叫等待功能可以有足够时间（3 分钟）让你在没有失去连接时便知道有人在呼叫你。

（3）采用 PCM 上行方式

V.92 技术将上行的通信方式改为与下行传输相同的 PCM（脉冲代码调制）方式，加速了用户对 ISP 的数据发送（上行）速度。V.92 MODEM 的发送（上行）速率为 48Kb/s，接受（下行）速率为 56Kb/s。而 V.90 56K MODEM 的发送（上行）速率为 33.6Kb/s，接受（下行）速率为 53Kb/s。PCM 本身的最大传输速度为 64Kb/s（8bit×8KHz）。在 56K MODEM 中下行方向的速度之所以为 56Kb/s，是由于在中继网中有时不能使用 8 位数列中的最下位。而此次在 V.92 中上行速率之所以限制在 48Kb/s，是因为将会受到电信局 A-D 转换器产生的量子化噪声影响的缘故。

（4）采用 V.44 压缩技术

国际电信联合会在宣布 V.92 标准的同时，也宣布了新的 MODEM 数据压缩标准 V.44。V.92 采用全新的 V.44 压缩标准，比原来 V.90 所使用的 V.42 压缩标准（V.42 bis 支持 14 的压缩标准）具有更高的压缩比，将压缩数据的效率提高了 25%，这意味着传输速率将可以借此提升。V.44 技术将客户端和局端拨号 MODEM 之间的实际数据吞吐量提高了 20%～60%（图像与文本混合）。

11.8.6　MODEM 的速度

MODEM 的速度是衡量 MODEM 性能的一个重要指标。一个 MODEM 有两个速度，一个是 DTE 速度，是指与计算机串口之间的通讯速度。一般用比特率表示（bit per second），如 19200bps、38400bps、57600bps、115200bps 等。另一个是 DCE 速度，也叫电话线上速度，是指两个 MODEM 连线时，两者之间的通讯速度，也用比特率表示，如 14.4Kbps、28.8Kbps、33.6Kbps、56Kbps 等，平时谈到选购 MODEM 时，所标明的也就是这个速度，它才是反映实际上网的速度。

随着 ITU-TS（国际通讯联盟）的 V.42bis 和 V.42 压缩和纠错协议的推出，对典型的 ASCII 码文件具有 4:1 的压缩比。同时 V.42bis 还具有智能检测方式，自动判断当前所传送的文件是否为压缩文件，如果是压缩文件，V.42bis 便不再进行压缩而直接传送，节省了时间。在没有推出压缩和纠错协议之前，MODEM 的 DTE 速度与 DCE 速度是完全一样的。而目前各厂家生产的 MODEM 已普遍具备了上述两组协议，因而也就产生了 1DTE 速度=4DCE 速度的概念。

既然知道了 MODEM 有两种速度，而且平时我们所说 MODEM 的速度，如 33.6K 或 56K 都是指 MODEM 的最高传输比特率。因此，在设置 MODEM 的速度时，应综合考虑以下几个方面的因素。

（1）MODEM 所支持的最高数据传输率

在安装 Windows 2000 时，串口的缺省速度是 9600bps，这显然限制了 MODEM 的传输速度。在传输数据过程中，一般将串口速率设置为超过 MODEM 的最高速率，否则 MODEM 建立连接时会受到此值的限制。例如，在“控制面板”→“系统”→“设备管理”→“端口”→“通讯端口 COM2”→“属性”→“端口设置”→“波特率”列表框中选择 115200 或者更高值。

（2）压缩比

不同类型文件的压缩比是不一样的，对于 ASCII 码，文本文件具有最大的压缩比，对于经过压缩软件压缩过的文件，基本上不再进行压缩。

（3）通讯线路的质量

通过国内电话网进行连接的 MODEM，容易受到通讯线路和程控交换机的瓶颈效应的制约，相对来说，市话比长话的效果要好。当电话通讯线路较好时，如果传输的是未经过压缩的文件，可以按 MODEM 最高传输率×压缩比=最高速率来设置。如果电话通讯线路不好，则可以根据线路实际支持的线路速度进行设置。14.4K 的 MODEM 可以设置成 19200bps、28.8K 和 33.6K 的 MODEM 可以设置成 38400bps、56K 的 MODEM 可以设置成 115200bps 以上。

（4）串口 UART 芯片的限制

计算机中，串口 UART 芯片所支持的最大通讯速度限制了外置式 MODEM 的通讯速度。到目前为止，常见的 UART 芯片的类型有三种，所能支持的最大 DTE 速率分别为：8250 型号支持 19200bps、16450 型号支持 38400bps、16550 型号支持 115200 或 557600bps。由于内置式 MODEM 中自带 UART 芯片，因此不受计算机串口芯片的限制。

上网时的速度最好是通过上网下载软件进行测试。利用 FTP 下载工具如网络吸血鬼，下载网上的压缩文件。对于正常的下载速度应该是连接速度除以 9 或 8（包括 8 个数据位，1 个起始位）。例如，以 33600bps 连接时，本地的正常下载速度应在 3.4KB～3.8KB 之间，如果大大低于正常值，则 MODEM 的工作速度不正常。

最后需要说明的两点，一是尽管目前许多 ISP 将它的设备进行了数字化改造，从因特网上上传或下载可以达到 56Kbps 的速度，但用 MODEM 发送数据也只有 33.6Kbps。由于受到电话线路和网络服务商出口带宽的限制，56Kbps 的 MODEM 至少在当前还不能充分发挥其应有的优势。二是目前市场上 MODEM 的品牌很多，大致可以分成两大类。一类是质量稳。这类 MODEM 以传输的质量为根本，传输数据比较可靠，在此基础上进行的连接速度就低不就高，实际上网速度比较快些。另一类是速度快。这类 MODEM 在连接时尽量提供较高的速度，而且有时以牺牲传输质量为代价，最坏的情况是因为传输过程中错误太多而反复传输或干脆断线，结果反而造成实际上网速度很慢。所以选购 MODEM 不能光看 MODEM 的最高传输速度，还要考虑自己的实际情况。

11.8.7　MODEM 优化方法

由于目前国内电话线路质量普遍较差，特别在中小城市采用分机上网，情况更为明显。爱上网的朋友常常要忍受网络速度太慢的煎熬，尤其在下载软件的时候。MODEM 在连线时一般很难达到其标称值，如 33.6K 的猫只有 20K 左右；56K 的猫能上 50K 就不错了。因此，我们不可避免地会遇到如何对 MODEM 进行优化设置的问题。

1. Windows 2000 下的 MODEM 优化方法

（1）优化系统配置

进行正确的系统设置并配置合理的参数，具体过程如下：

- 进入“控制面板”→“电话和调制解调器选项”→“调制解调器（选项卡）”，然后选中你所使用的 MODEM，单击“属性”→“高级”；
- 将 MODEM 特定的初始化参数填入附加值设置栏中；
- 选择“高级端口设置”；
- 把滑块都移至最右端，单击“确定”按钮；
- 选择“改变默认参数…”；
- 将“端口速度”设为 115200；
- 单击“确定”并退出。

（2）注册表优化

在进行如下操作之前，务必将注册表内容做备份，以防万一。其实在 Windows 2000 平台中，一些比较重要的网络方面的设置都被屏蔽掉了，正因为这样，下面所列的一些键及键值可能没有直接出现在注册表里，这就需要你添加相应的主键名和键值，而且即使由此产生了什么问题，也只需将添加的内容删去即可恢复其默认值了。

打开注册表编辑器，并依此路径展开：

编辑—新建—输入主键名—点鼠标右键选择“修改(modify)”→输入键值：

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters]
"EnablePMTUBHDetect"=dword:00000000
"SackOpts"=dword:00000001
"TCPWindowSize"=dword:00000003
"DefaultTTL"=dword:00000020
"EnablePMTUDiscovery"=dword:00000001
```

说明：

- EnablePMTUBHDetect——该项为启动 TCP/IP 协议中的 Black Hole Detection（黑洞侦测）功能，它会检查每个 TCP/IP 堆栈（stacks），从而妨碍浏览速度，所以要关闭它。（注：0－disabled　1－Enabled）
- TCPWindowSize——该项用于设置 TCP 窗口大小的。缺省时窗口为 64k，不过此键对 MODEM 的上传下载并不起作用，所以不必考虑。
- EnablePMTUDiscovery——该项主要针对 Mtu Auto Detect 功能，有激活和关闭两种选择。（注：0－关闭　1－激活）

经过以上设置之后，再拨号上网，相信你就能感觉到 MODEM 速度已经大有提高了。

2．系统监测

用过 Windows 98 的人都知道，微软在系统中集成了许多系统工具和实用程序，用户通过这些工具和程序可轻松自如地实现对系统的管理和维护，并创建一个稳定和安全运行的系统。其中有一个名为 Sysmon 的系统监视工具，它能时刻监测当前正在运行的系统硬件资源使用情况，因此，我们可以利用它来观测 MODEM 优化后的工作效果，方法如下：

（1）单击“开始”→“运行”，打开“运行”提示框，在命令行中输入“Sysmon”；

（2）在弹出的“系统监视器”界面中，单击工具栏上的“编辑（Edit)”→“添加项目…”→“Dial－Up Adapter”，在“项目”中添加“Bytes Received/Second（每秒接收的字节）”、“Bytes Transmitted/Second（每秒传送的字节）”、“Connection Speed（连接速度）”几项。

这样，就可以在 Sysmon 中看到 MODEM 每秒接收的字节数、每秒传送的字节数和连接速度了。

在 Windows 2000 下，检查 MODEM 工作情况和网络状态的方法有两种：

（1）在“运行”提示框中键入“perfmon”命令；

（2）单击“开始”→“设置”→“控制面板”，在控制面板中双击“网络和拨号连接”，打开拨号窗口。

11.8.8　MODEM 命令/AT 命令

所有的 MODEM 命令都由一个特定的“命令前缀”开始，由一个“命令结束标志”结束。命令前缀通常总是由 AT 两个字符组成，它是 Attention 的缩写。因而又称 MODEM 命令为 AT 命令。命令结束标志是一个单字符，其值存在寄存器 S3 中，通常为回车符。S3 中的内容可以用 AT 命令修改，因而命令结束标志是可以改变的。

命令行除非特别情况，一般均由 AT 或 aT 开始。MODEM 从这两个字符串能检测出波特率、字长、奇偶检验。

每个 AT 命令由一个单字符或一个&号后跟一个字符定义组成。基本上所有命令的后继参数均采用十进制形式。每个 AT 命令行可以包括许多条 AT 命令，而只需一个 AT 引导。一个命令行被正确执行的前提是其中每个命令都是正确的，否则将被丢弃。执行完命令行后，MODEM 并不清除命令缓冲区，而一直保存到下一条新的 AT 命令到来为止。惟一没有命令前缀和结束标志的命令是 A/，它使 MODEM 重复执行存于缓冲区中的上一个命令行。

每当 MODEM 执行完一个 AT 命令行后，MODEM 都会返回结果码，以对接收到的命令

做出响应。

下面以列表方式分别从 13 个功能分类对 AT 命令进行介绍。最后还对 S 寄存器和结果码做了简要介绍。

应该说明的是，随着技术的发展以及不同厂家的生产特点，对于每个具体的 MODEM，其 AT 命令可能有所不同，详细情况请参考 MODEM 用户手册。

AT 命令一般格式为：AT+命令+参数（可选）

1. 用户接口命令

当 MODEM 处于命令状态时，每发送一个 AT 命令，MODEM 都会用一个结果码（通常为 OK 或 ERROR）响应并指示当前命令执行情况。用于完成对 AT 命令的回显和结果码控制。

表 11-8-8-1 为用户接口命令及其功能、参数列表。

表 11-8-8-1 接口命令列表

命令	功能描述	参数取值	缺省值
E	该命令控制 MODEM 在命令状态下打开或关闭命令回显	0：关闭回显 1：打开回显	1
V	该命令选择 MODEM 返回给 DTE 的结果码是数字形式还是字符形式	0：以数字形式显示结果码 1：以字符（包括换行符）形式显示结果码	1
Q	结果码控制	0：返回结果码（表 11-8-8-2 为结果码列表） 1：不返回结果码	0
X	结果码类型/呼叫进程	0：允许 0~4 结果码返回，禁止拨号音和忙音检测 1：允许 0~5、10 和以后的结果码返回，允许拨号音和忙音检测 2：允许 0~6、10 和以后的结果码返回，允许拨号音检测，禁止忙音检测 3：允许 0~5、7、10 和以后的结果码返回，允许忙音检测，禁止拨号音检测 4：允许 0~7、10 和以后的结果码返回，允许拨号音和忙音检测	4

表 11-8-8-2 结果码列表

数字码	字码符	含义
0	OK	命令正确执行
1	CONNECT	连接建立
2	RING	检测到振铃信号
3	NO CARRIER	没有接收到载波或载波丢失
4	ERROR	无效命令、检验和错误、命令行错误或错误行超过 255 个字符
5	CONNECT 1200	在 1200bps 速度下建立连接
6	NO DIALTONE	没有检测到拨号音
7	BUSY	检测到忙音
8	NO ANSWER	当拨一个不提供拨号音的系统时，未检测到无声信号
10	CONNECT 2400	在 2400bps 速度下建立连接
11	CONNECT 4800	在 4800bps 速度下建立连接
12	CONNECT 9600	在 9600bps 速度下建立连接
14	CONNECT 19200	在 19200bps 速度下建立连接

2. 拨号呼叫

拨号呼叫命令用于完成 MODEM 向另一个 MODEM 发起呼叫，建立通信链路的任务。连接成功之后，MODEM 通常由命令状态转入在线状态。

表 11-8-8-3 为拨号呼叫命令及其功能、参数列表。

表 11-8-8-3　拨号呼叫命令列表

命令	功能描述	补充描述	缺省值
D	该命令使 MODEM 立即进入摘机状态，并拨出随后的号码（拨号串）以试图建立连接。如果命令后面没有跟拨号串，则 MODEM 将进入在线状态，并确认在呼叫模式	D 命令是基本的拨号命令，它受到其他一些命令的影响。拨号串由拨号修饰符构成。拨号修饰符用于指示 MODEM 何时拨号以及如何拨号操作。表 11-8-8-4 为拨号修饰符。	无
T	设定音频拨号	该命令对 D 命令起补充作用，使得在拨号时使用双音频（DTMF）形式进行拨号	无
P	设定脉冲拨号	该命令对 D 命令起补充作用，使得在拨号时使用脉冲拨号形式进行拨号，MODEM 缺省拨号方式是脉冲拨号方式	无

表 11-8-8-4　为拨号修饰符

拨号修饰符	描述	举例
0~9 A~D，*，#	电话号码中数字和字符；拨号后，MODEM 就一直等待对方 MODEM 传送载波信号，如果在指定的时间内没有检测到载波，则 MODEM 将自动释放线路，返回 NO CARRIER 结果码给 DTE。如果检测到载波，则通信双方进行差错控制的协商，最后返回 CONNECT 9600 等表示连接速率的结果码，表示连接成功。此时，MODEM 自动进入在线状态（拨号串中有“；”修饰符除外），双方可以开始进行通信	如：ATDT（010）86234791 或 ATDT010-62235415，其中连字符和括号只是用来提高命令的可读性。010 为长途区号，86234791 为电话号码
T	选择音频拨号	如：ATDT87654321，其中 87654321 为电话号码
P	选择脉冲拨号	如：ATDP87654321
W	等待拨号音	
,	延迟处理下一字符；延迟时间由 SS 寄存器指定，缺省值为 2 秒	如：ATDT8，65532145
!	挂机闪烁：使 MODEM 短暂摘机 0.5 秒	
@	使 MODEM 进入 5 秒钟的无声回答后再检测线路信号，然后继续执行命令。如果没有监听到 5 秒无声回答（或者出现了一个话音或其他声音信号），则终止呼叫并返回“NO ANSWER”结果码；如果监听到一个占线信息，则返回“BUSY”结果码	
R	使得呼叫方在拨号后切换到应答模式；该修饰符一般加在拨号串的末尾。MODEM 只能以主叫方式进行载波检测，当它们用做被叫方时，就必须用 R 修饰符设置呼叫方的 MODEM 以应答模式进行载波检测	ATDT65523148R
;	指示 MODEM 拨号后返回命令状态，此时 MODEM 只拨号呼叫而不应答对方	ATDT65523148;
sr	用存储在 NVRAM 中的第 r 个号码拨号	如：ATDS0，拨预存储在位置 0 的电话号码

3. 应答呼叫

当 MODEM 检测到远端系统传送来的一个呼叫时，电话铃响，此时 MODEM 有两种应答方式：手工应答和自动应答。

表 11-8-8-5 为两种方式的命令描述。

表 11-8-8-5　应答呼叫命令

命　令	描　述
A	手工应答，使得 MODEM 立即摘机，并等待来自远端 MODEM 的拨号呼叫和载波信号，试图应答呼叫，而不需要等待呼叫振铃信号，对于一个应答，MODEM 等待 S7 寄存器所指定的时间。如果没有检测到载波，则自动挂机。该命令可用来建立一个手工连接或背靠背连接
S0=r	r 值范围为 1～255 时，表示响铃 r 次之后，MODEM 自动摘机并试图连接；只要 S0 为非 0，则 MODEM 就具有自动应答特性。如果 r 设置为 0，则禁止自动应答。当自动应答被禁止后，则每次电话铃响时，MODEM 返回 RING 结果码，但不应答呼叫，除非此时执行 ATA 命令

手工连接是指当听到 MODEM 电话铃响之后，立即发了 A 命令，此时 MODEM 指示进入摘机状态，应答呼叫，与远端系统建立通信链路。如果建立成功，则 MODEM 返回 CONNECT 9600 等指示连接成功的返回码。如果未连接成功，则返回 NO CARRIER 返回码。

背靠背连接是指两台距离较近的 MODEM 不通过电话线，而是直接使用调制电缆将两个 MODEM 连接起来。在这种方式中，因为 MODEM 并未连接到电话线上，因此无须接电话机也无须检测拨号音，也就没有振铃信号去触发自动应答。在连接之前，首先应执行 ATX3 命令，禁止 MODEM 采用应答方式摘机。这样，最终建立起一条通信链路。

4. 电话网络模式选择命令

电话网络分为两种模式：拨号模式和专线模式。

拨号线是指用一个专门号码进行拨号连接的计算机和终端的通信线路，数据通过电话线路传送到计算机，话路质量相对较差、数据传送不稳定、保密性较差。

专线是指在一台计算机和另一台终端间的永久通信连接，完全由租用者支配。两端点固定不变，保密性强，线路质量好，数据传送稳定、可靠、高效。利用专线进行声音通信，不必拨号便可以通话。

MODEM 工作状态设置为专线连接方式命令为：&L1；&L0 则选择拨号线路方式。

5. 状态切换命令

当通信双方建立通信链路之后，就从命令状态进入在线状态，此时双方可通过电话线发送和接收数据。

从在线状态切换到在线命令状态的命令为：+++。这是一个换码序列，而不是一个 AT 命令，因而命令前面不加 AT，后面也不用跟回车符。在发出+++命令之前和之后应停顿一段时间，该时间由 S12 寄存器指定，缺省为 1 秒，以保证 MODEM 接收到该命令。否则，+++将被当做普通数据处理。

从在线命令状态返回到在线数据状态的命令为：O；它有两种不同参数设置。

- O0：返回在线数据状态（缺省设置）；
- O1：重新调整适应型均衡器（重试联机），并返回在线数据状态。

6. 挂机命令

在通信结束后，应挂机、拆除线路。表 11-8-8-6 为挂机命令及其功能、参数列表。

表 11-8-8-6　挂机命令

命令	功能描述	参数取值	缺省值
H	挂机/摘机控制：相当于用手挂上/提起电话听筒的操作	0：挂机，并将 MODEM 置于命令状态 1：摘机	0
Y	长空挂断：该命令允许或禁止长空挂断。长空挂断可用于挂断两个 MODEM	0：禁止长空挂断 1：允许长空挂断	0
Z	软件复位/恢复保存的预置文件	0：软件复位并恢复预置文件 0 1：软件复位并恢复预置文件 0	0

表中所提到的长空挂断是指当检测到任何挂断状态时，MODEM 立即发送一个 4 秒的空信号给对方的 MODEM，该信号通知对方 MODEM 挂断。另一端，如果已执行 ATY1 命令（允许长空挂断），则只要接到 1.6 秒的空号，MODEM 将丢失载波，然后挂断。这可能引起无意的挂断，因此，大多数 MODEM 都缺省禁止长空挂断。

7. MODEM 逻辑接口命令

MODEM 提供一组 AT 命令用来规定逻辑接口的行为。表 11-8-8-7 为 MODEM 逻辑命令及其功能、参数列表。

表 11-8-8-7　逻辑接口命令

命令	功能描述	参数取值	缺省值
&C	数据载波检测（DCD）选择：该命令控制 MODEM 如何处理 DCD 和 RLSD 的关系	0：忽略远端 MODEM 的载波状态，DCD 始终有效 1：跟踪远端 MODEM 的载波状态，当检测到　接收载波时 DCD 有效	视不同 MODEM 而不同
&D	数据终端准备就绪（DTR）选择：该命令控制 MODEM 如何解释从 DTE 到 MODEM 的 DTR 信号的 ON—OFF 跳变	0：忽略 DTR 信号 1：当 DTR 由 ON 到 OFF 时，MODEM 将从在线方式切换到命令方式 2：当 DTR 由 ON 到 OFF 时，MODEM 将挂起，并返回命令状态 3：当 DTR 由 ON 到 OFF 时，MODEM 将重新初始化，执行一个上电检测（不包括 UART 寄存器设置）	0（也有些 MODEM 缺省值为 2）
&R	请求发送/清除发送（RTS/CTS）选择：该命令控制 MODEM 到 DTE 的 CTS 信号的操作	0：当 MODEM 在线时，CTS 跟随 RTS 的变化 1：当 MODEM 在线时，CTS　直有效，忽略 RTS 信号	0（也有些 MODEM 缺省值为 1）
&S	数据设备就绪（DSR）选择：该命令控制由 MODEM 到 DTE 的 DSR 信号的操作	0：DSR　直有效 1：只有在握手时 DSR 有效，即检测到载波时允许 DSR，失去载波时禁止 DSR	0

8. 扬声器控制

在拨号过程和数据传送过程中，可通过编程控制 MODEM 的扬声器音量大小以及开关状态。表 11-8-8-8 为扬声器控制命令及其功能、参数列表。

表 11-8-8-8　扬声器控制命令

命令	功能描述	参数取值	缺省值
L	扬声器音量控制	0：低音量 1：次低音量 2：中音量 3：高音量	2
M	扬声器开关控制：该命令用于选择开关时间	0：扬声器一直关闭 1：建立呼叫时开扬声器，一旦检测到载波后关闭 2：扬声器一直开 3：类似于 1，但在拨号期间扬声器关闭	1

9. 版本信息及自检测试

每一个 MODEM 中都存在版本信息，包括 MODEM 的型号、制造厂家、检验和等信息。另外可通过自检命令对 MODEM 进行测试。表 11-8-8-9 为版本信息及自检测试命令及其功能、参数列表。

表 11-8-8-9　版本信息及自检测试命令

命令	功能描述	参数取值	缺省值
I	MODEM 将产品标识代码、型号和版本号等信息以 ASCII 字符形式发送给 DTE	0：返回产品标识代码 1：返回 ROM 校验和（有些 MODEM 厂家设置为返回 MODEM 芯片版号） 2：验证 ROM 检验和，返回 OK 或 ERROR 结果码	0
&T	建立或中止回送检测，以检测 MODEM 到 MODEM 之间、MODEM 与 DTE 之间通信的完整性	0：中止检测进程 1：本地模拟回送 2：本地数字回送 3：同意远端数字回送请求 4：禁止远端数字回送请求 5：远端数字回送 6：带自检的远端数字回送 7：带自检的本地数字回送	0

10. 配置命令

MODEM 提供一组命令，用于 MODEM 的配置管理。MODEM 一般提供三种配置：厂家配置、当前配置、存储配置。其中，厂家配置存放在 MODEM 的 ROM 芯片中，当前配置存放在 MODEM 的 RAM 中，存储配置存放在 NVRAM（非易失存储器）中。表 11-8-8-10 为配置命令及其功能、参数列表。

表 11-8-8-10　配置命令

命令	功能描述	参数取值	缺省值
&F	MODEM 重新调出“厂家原始配置“作为当前 MODEM 的活动配置，但不保存该配置	0：恢复厂家原始配置文件 0 1：恢复厂家原始配置文件 1	0
&W	将当前 MODEM 的活动配置（当前配置）保存到由 0 或 1 指定的 NVRAM 中，包括保存 S 寄存器的当前设置	0：保存当前配置到预置文件 0 1：保存当前配置到预置文件 1	0
Z	MODEM 执行一个软件复位并恢复指定的存储配置文件为当前配置	0：软件复位并恢复预置文件 0 1：软件复位并恢复预置文件 1	0
&Y	指定启动配置：当 MODEM 上电或执行 ATZ 命令时，MODEM 将自动恢复两个存储配置中的一个作为当前配置	0：上电/软复位时恢复预置文件 0 1：上电/软复位时恢复预置文件 1	0
&V	显示当前配置和存储的预置文件	0：预置文件 0 1：预置文件 1	0
&Z	存储电话号码：MODEM 中的 NVRAM 最多可存储四个常用电话号码	可使用 ATDS=n 命令来拨出存储在位置 n 的电话号码；n=0、1、2、3	无

11. 寄存器操作

MODEM 内部有许多寄存器，称为 S 寄存器。S 寄存器提供了存取 MODEM 设置的专用方法。表 11-8-8-11 为寄存操作命令及其功能、参数列表。

表 11-8-8-11　寄存器操作

命令	功能描述	参数取值	缺省值
Sn=X	写一个十进制数到 S 寄存器 n 中 X=0~255		无
Sn?	读 S 寄存器；该命令使 MODEM 返回 S 寄存器 n 的数值（十进制）	可以使用 AT&V 命令查看所有 S 寄存器中的内容	无

12. 连接性选择命令

表 11-8-8-12 为 AT 命令影响 MODEM 与其他 MODEM（或通信标准）相互作用的方式。

表 11-8-8-12　连接性选择命令

命令	功能描述	参数取值	缺省值
B	该命令允许 MODEM 选择 Bell 或 CCITT 调制方式	0：在 1200bps，选择 CCITT　V22 标准通信 1：在 1200bps，选择 Bell 212A 标准通信	0
&G	选择校正音；该命令只适用于 1200bps(V22)和 2400bps(V22 bis)的连接	0：禁止校正音 1：允许 550Hz 校正音 2：允许 1800Hz 校正音	0
&M	同步/异步模式选择	0：选择异步通信工作方式 1：选择同步方式 1 2：选择同步方式 2 3：选择同步方式 3	0
&Q	同步/异步模式选择	0：选择异步通信工作方式 1：选择同步方式 1 2：选择同步方式 2 3：选择同步方式 3 5：差错控制方式 6：具有自动速率缓冲的异步通信方式	0

13. 其他通用命令

表 11-8-8-13 为其他通用命令列表。

表 11-8-8-13　其他通用命令

命令	功能描述	参数取值	缺省值
A/	MODEM 重复上一次的 AT 命令。主要用于重拨电话号码。该命令前面不加 AT，后面也不用跟回车符		无
&K	定义 MODEM 和 DTE 间的流量控制（流控）方式	0：禁止流控 3：允许 RTS/CTS 流控 4：允许 XON/XOFF 流控 5：允许透明 XON/XOFF 流控 6：允许 RTS/CTS 和 XON/XOFF 两种流控	3
N	允许或禁止自动模式检测	0：禁止自动模式检测，以 S37 寄存器指定的 DCE 速率进行握手； 如果 S37＝0，则与新近检测的 DTE 速度一致。 1：允许自动模式检测，通信双方从 S37 指定的 DCE 线路速率开始协商，必要时降低速率	1
W	设置协商处理结果码	0：不返回协商结果码 1：返回协商结果码 2：不返回协商结果码，且返回的 CONNECT 信息使用 DCE 速率而不是 DTE 速率	0

14. S 寄存器

为了满足用户对 MODEM 的一些高级操作和控制的要求，某些 MODEM 参数被保存到 RAM 区内的寄存中，这些寄存器称为 S 寄存器。S 寄存器提供了存取 MODEM 设置的专用方法以及控制 MODEM 工作的高级技术。不同型号的 MODEM，其 S 寄存器的个数和作用都有所不同。前面介绍了 S 寄存器的操作命令，表 11-8-8-14 列出一些最常用的 S 寄存器。

表 11-8-8-14　常用 S 寄存器

S 寄存器	描述	取值范围	缺省值	单位
S0	该寄存器表示 MODEM 开始自动应答前需要等待的振铃次数。	0~255 0：表示禁止 MODEM 的自动应答特性 如果线路只用于 MODEM 通信，则通常设置为 1；如果还用于话音通信，则应设置为较大的数	0	振铃次数
S1	表示电话已振铃的次数，每响一次加一，振铃结束后自动清零。它是一个只读寄存器	0~255	0	振铃次数
S2	该寄存器含有表示由在线状态到命令状态的换码字符。连续三个换码字符就构成一个换码序列，完成由命令状态到在线状态的转换	0~127	43	十进制 ASCII 码
S3	该寄存器定义了回车字符的值。MODEM 将回车字符作为 AT 命令行的结束并且作为结果码的结束符发送	0~127	13	十进制 ASCII 码
S4	该寄存器定义了换行字符的值。MODEM 将换行字符作为一个回送命令行结束符或一个结果码结束符	0~127	10	十进制 ASCII 码
S5	该寄存器定义了回退字符的值。回退字符完成退格操作	0~32，127	8	十进制 ASCII 码
S6	该寄存器定义了 MODEM 从摘机到开始拨号之间应等待的时间	2~255	2	秒
S7	该寄存器定义了在拨号后或者应答一个呼叫时，MODEM 等待载波的最大时间	1~255	50	秒
S8	该寄存器决定了当 MODEM 在拨号串中遇到“，”应暂停的时间	0~255	2	秒
S9	该寄存器指定了 MODEM 识别远端 MODEM 发来的载波，并确认载波的时间长度	1~255	6	0.1 秒
S10	该寄存器指定了从 MODEM 发现远端载波丢失到挂机之间的时间	1~255	14	0.1 秒
S11	该寄存器指示拨号时音频的持续时间。拨号速率是该寄存器值倒数的一半	50~255	95	0.001 秒（毫秒）
S12	该寄存器指示 MODEM 在换码序列开始之前和之后必须空闲的时间	0~255	50	0.02 秒
S37	该寄存器决定了 MODEM 与远端 MODEM 建立连接时的最高线路速率	0~12	0	无

15. MODEM 返回信息码

当 MODEM 处于命令状态时，每当计算机发送一条 AT 命令，MODEM 至少返回一个结果码，以指示当前命令是否正确执行以及执行结果。结果码有两种形式：一种是文本信息（字符串形式），另一种是数字码。如果程序员自编软件来控制 MODEM，则最好使用数字码形式，它的优点是软件处理结果码简单方便。如果使用 MODEM 的附带软件来控制 MODEM，则可以使用字符串形式，它的优点是结果清楚明了。

AT 命令集中提供了 V 命令来选择结果码返回形式。ATV0 命令表示以数字形式返回结

果码，ATV1 命令表示以字符串形式返回结果码。

随着 MODEM 的速率的提高和功能的增强，结果码随之增加，但各厂家所定义的结果码并不统一。表 11-8-8-2 列出了大多数 MODEM 都支持的结果码。对于每个具体的 MODEM 结果码可能有所不同，详细情况请参考 MODEM 用户手册。

第 12 章 不占用串口的串口数据捕捉

[内容提要]

本章主要探讨不占用串口情况下的串口数据捕捉，并简要介绍了虚拟串口的有关知识。

对较高层次的编程者来说，也许在项目开发中能碰到上面的问题，这些问题在一些编程论坛上也引起较多的讨论。由于解决这些问题需要较多的硬件驱动知识，所以本章我们只做简单的讨论，并给出了 Windows 98 下 VxD 串口数据捕捉实例程序，对虚拟串口，介绍了一般的应用，使读者对这些知识有一个初步的认识，了解进一步努力的方向。

12.1 驱动程序的基本概念：VxD 与 WDM

运行串口调试助手，打开计算机上明明存在的串口有时却会出现这样的提示："该串口不存在或被其他设备占用"，即串口一旦被其他设备占用，就无法再使用这个串口，这说明在普通的编程中，串口硬件资源是无法共享的。但编程者有时需要知道正在工作着的串口发送或接收到了哪些数据，在测试没有显示界面的"黑匣子"串口通信程序时，更是希望能捕捉到串口通信数据。要实现这些操作，涉及到系统级的虚拟设备驱动程序，因此我们首先要熟悉设备驱动程序的有关知识，设备驱动程序应用程序提供访问设备的软件接口，对串口设备而言，进行驱动程序的开发是串口通信程序开发中技术难度较大的一部分。

本节仅从串口编程的应用角度讲述，并给出简单的 VxD 应用实例，要系统地了解设备驱动程序的编程，请参阅相关资料。

要掌握驱动程序的编写，首先就要了解与之相关的两个基本概念。

- VxD：Virtual Device Driver，虚拟设备驱动程序；
- WDM：Win32 Driver Model，Win32 驱动程序模型。

12.1.1 虚拟设备驱动程序 VxD

VxD 是系统用于对各种硬件资源识别、管理、维护运作的扩展。VxD 和 VMM（虚拟机管理器）一起合作，维持着系统的运行。VxD 可以随 VMM 一起静态加载，也可以根据需要动态加载或卸载。正是由于 VxD 与 VMM 之间的紧密协作，才使得 VxD 具有了应用程序所不具备的能力，诸如可以不受限制地访问硬件设备、任意查看操作系统数据结构（如描述符表、页表等）、访问任何内存区域、捕获软件中断、捕获 I/O 端口操作和内存访问等，甚至还可以截取硬件中断。VMM 是通过 VxD 的设备描述符块 DDB（Device Descriptor Block）来识别的。DDB 向 VMM 提供了 VxD 的主入口点，还向应用程序和其他的 VxD 提供了入口点。VMM 利用这个主入口点将 VM 及 Windows 自身的状态通知给 VxD，然后 VxD 通过相应的工作来响应这些事件。由于 VxD 不仅仅服务于一个物理设备（比如多个串口）或仅与一个 VM 发生联系，所以 VxD 需要产生自己支持的数据结构（Supporting Data Structures）来保存

每一个设备、每一个 VM 的配置和状态信息。VxD 用一个或多个设备上下文结构来保存设备信息，如 I/O 端口基址址、中断向量等，VxD 将自己的每个 VM 的状态信息保存在 VMM 的 VM 控制块中。VMM 提供的服务包括：事件服务、内存管理服务、兼容执行和保护模式执行的服务、登录表服务、调度程序服务、同步服务、调试服务、I/O 捕获服务、处理错误和中断服务、VM 中断和回调服务、配置管理程序服务以及其他杂项服务。

这里还要重点谈一个问题，即 VxD 与应用程序间的通信机制，这是我们在理解本章的实例程序时应该有所了解的。由于 VxD 在处理硬件设备的同时，还要向应用程序提供接口，只有通过该接口，应用程序才能控制硬件设备。一般地，我们在 Windows 98 操作系统中，只用 VxD(不用 WDM)进行串口设备驱动程序的编写，Windows 98 操作系统中，Win32 应用程序到 VxD 的通信机制是：VxD 并不像 Win16 应用程序接口那样输出一个特殊的 API 过程（保护模式 API 过程或 V86 模式 API 过程）来支持应用程序，取而代之的是它的控制过程必须能够处理一个特殊的消息：W32_DEVICEIOCONTROL。VMM 代替调用 **DeviceIoControl** 函数（这个函数在下面的实例程序中多次用到）的应用程序向 VxD 发送此消息，消息参数可确定 VxD 消息响应函数、输入输出缓冲区指针及缓冲区大小，并绑定在 DIOCPARAMETERS 结构中，通过这一接口，不仅仅可以读写设备，而且还能在应用程序和 VxD 之间互传指针，从而达到特殊应用的目的。有时，只需调用应用程序与 VxD 间的接口，便能及时获得所需信息和服务。但还有一些特殊情况，必须由 VxD 异步通知应用程序，这就需要用到 VxD 到应用程序的通信机制。

VxD 到应用程序的接口关系要比应用程序到 VxD 的接口关系复杂得多。一般有两种调用方法：一种是使用 PostMessage 函数。通过调用这一由外壳 VxD（SHELL VxD）提供的新服务，便可通知应用程序；另一种是使用特殊的 Win32 技术。这种技术的独到之处在于 Win32 API 支持多线程。

在 Win32 技术中，采用的 APC(Asynchronous Procedure Calls)异步过程调用机制和 Win32 事件机制都依赖于唤醒一个 Win32 应用程序线程，而 VxD 到应用程序最简单的通信机制就是通过 APC。应用程序首先动态加载 VxD（CreateFile），并用 DeviceIoControl 将回调函数的地址传给 VxD，然后应用程序执行。Win32 调用 SleepEx 将其自身置为“挂起”状态时。当应用程序处于“挂起”状态，VxD 能够通过 VWIN32 VxD 提供的 QueueUserApc 服务调用应用程序的回调函数。另一种更有效的方法是使用 Win32 事件机制。如果应用程序运行多个线程，当子线程等待着 VxD 来唤醒它的同时，主线程能够继续做自己的工作。

VToolsD 是专门用于开发 VxD 程序的一种开发工具软件，它包括 VxD 框架代码生成器 QuickVxD、C 运行库、VMM/VxD 服务库、VxD 的 C++类库、VxDLoad 和 VxDView 等实用工具以及大量的 C、C++例程。由 VC++、BC++的 32 位编译器编译生成的 VxD 程序可以脱离 VToolsD 环境运行。VToolsD 的具体使用和开发 VxD 的详细方法请读者参考有关书籍。

DeviceIoControl 函数在以下的编程中需要用到，在此详细说明。

功能：DeviceIoControl 函数可以直接向一个特定的设备驱动程序发更新控制命令，使相关的设备去执行指定的操作。函数原型如下：

```
BOOL DeviceIoControl(
  HANDLE hDevice,
  DWORD dwIoControlCode,
  LPVOID lpInBuffer,
  DWORD nInBufferSize,
```

```
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
  );
```

参数说明：

hDevice　[输入]

要执行操作的设备句柄。调用 CreateFile 函数来得到这个句柄， CreateFile 函数的应用说明请参阅本书第 4 章 API 函数说明，另外还要说明一下，对设备名称，格式如下：

\\.\设备名　（如果是 C 语言，则为\\\\.\\设备名）

对 Windows Me/98/95 操作系统，DeviceIoControl 仅能操作虚拟设备，例如用 CreateFile 打开，则为\\.\设备名，一般地，虚拟设备名一般放在系统文件夹中 C:\Windows\System。对于通信设备（如串口），使用独占模式打开。

dwIoControlCode [输入]

指定要执行的操作内容或方式。如读或者写。

lpInBuffer　[输入]

输入缓冲区指针，如果不需要数据输入，可以设置为 NULL。

nInBufferSize　[输入]

输入缓冲区大小（字节 byte 为单位）。

lpOutBuffer [输出]

输出缓冲区指针，通常对串口数据捕捉程序，就是需要得到返回的数据，因此编程时要设置一个指针变量在这里接收数据。

如果不需要返回数据，可以设置为 NULL。

nOutBufferSize [输入]

输出缓冲区大小（字节 byte 为单位）。

lpBytesReturned [输出]

得到放入输出缓冲区内数据大小的指针变量（字节 byte 为单位）。如果输出缓冲区过小，则调用失败，GetLastError 返回错误码 ERROR_INSUFFICIENT_BUFFER，并且 lpBytesReturned 值为零。

lpOverlapped [输入]

OVERLAPPED 结构指针。

返回值：

调用成功，返回值不为零，如失败则为零，进一步的错误信息可调用 GetLastError 获得。

12.1.2　Win32 驱动程序模型 WDM

WDM（Win32 Driver Model），即 Win32 驱动程序模型，是微软推出的一种较新的驱动程序模式。1997 年微软就提出了该项技术，并使其在 Windows 98 中得到了充分的应用；1999 年推出的 Windows 2000(Windows NT 5.0) 不但继承了 Windows NT 4.0 的种种优点,而且改进了系统结构，使其同时也支持 WDM 驱动程序。也就是说，WDM 是一种跨平台的驱动程序结构。不仅如此，由于引入了 HAL 层（硬件抽象层），WDM 驱动程序还可以在不修改源代码的情况下经过重新编译后在非 Intel 平台上运行。

我们在编写串口 API 程序时，用 CreateFile 来打开串口设备，并用 ReadFile 和 WriteFile 来读写串口设备，因此，每一个物理设备在上层编程时都抽象成了文件。WDM 驱动程序是分层的，不同层上的驱动程序有着不同的优先权。WDM 引入了功能设备对象 FDO（Functional Device Object）与物理设备对象 PDO（Physical Device Object）两个新概念来描述硬件。本质上来说，它们都是系统为帮助软件管理硬件而创建的数据结构。处于堆栈最底层的设备对象称为 PDO。在设备对象堆栈的中间某处有一个对象，称为 FDO。在 FDO 的上面和下面还会有一些过滤器设备对象（Filter Device Object）。一个 PDO 代表一个真实硬件，在驱动程序看来，则是一个 FDO。另外值得注意的是，一个硬件只允许有一个 PDO，但却可以拥有多个 FDO，而在驱动程序中我们不是直接操作硬件，而是操作相应的 PDO 与 FDO。操作系统为每一个用户请求打包成一个 IRP 结构，将其发送至驱动程序，并通过识别 IRP 中的 PDO 来识别是发送给哪一个设备的。

WDM 驱动程序的运行包括安装、加载和执行三个阶段。安装就是根据 inf 文件的信息，把编译好的驱动程序拷贝到操作系统的特定目录，并且修改相应的注册表信息来注册该驱动程序的参数（包括路径、名称和加载方式等）；加载就是根据 inf 文件中制定的加载方式（动态或者静态）在相应的时候加载某个设备的驱动程序。在驱动程序加载以后，上层软件就可以通过驱动程序来访问硬件设备了。

12.1.3　在不同操作系统下选用哪种驱动程序模式

上面讲了一大堆的概念，读者看了肯定头都大了。现在我们只要有这么一个印象就可以了：要做到不占用串口物理资源来进行串口数据的捕捉，必须用到设备驱动程序，而 VxD 和 WDM 就是两种不同的设备驱动程序。那么开发串口设备的驱动程序，针对具体的 Windows 使用系统，到底如何选定驱动程序模式呢？到底该用 VxD 还是 KMD，还是 WDM 呢？

我们在这里讨论目前常用的 Windows 98 和 Windows 2000/XP 操作系统。应该说，这两种操作系统都支持 WDM，而 WDM 又可以突破操作系统的限制，那么 WDM 应该是首选了，但对串口设备来说，还不尽然。

在 Windows 98 下，如果开发串口通信、磁盘等设备的驱动，必须用 VxD，因为 Windows 98 并没有提供这些设备的 WDM 支持。在 Windows 98 下，网络设备、多媒体设备等的硬件支持已经转化到了 WDM，我们在 DDK 里就可以发现这几类驱动程序的样例程序。

而在 Windows 2000/XP 下，WDM 当然是首选。

12.2　VxD 示例程序介绍——VToolsD 中的 CommHook

前面已经提到，VToolsD 是开发 VxD 的工具软件，它自带的例程中就有用于串口数据捕捉的 VxD 例程：CommHook.vxd，因为涉及到 VToolsD 开发软件的操作应用，这里对其详细的编写过程不做描述，仅列出这个“钩子”（HOOK）程序的 C 源代码，使读者有一个印象：VxD 驱动程序也可以用 C 语言编写，这样我们就有进一步去尝试的勇气。

COMMHOOK.cpp 实现文件如下：

```
// COMMHOOK.cpp - main module for VxD COMMHOOK
#define DEVICE_MAIN
#include "commhook.h"
```

```
Declare_Virtual_Device(COMMHOOK)
#undef DEVICE_MAIN
CommhookVM::CommhookVM(VMHANDLE hVM) : VVirtualMachine(hVM) {}
CommhookThread::CommhookThread(THREADHANDLE hThread) : VThread(hThread) {}


//
// These are used to store the original service addresses
// and the thunks that are used by the hook and unhook
// functions
COMMPORTHANDLE(*VCOMM_OpenCommService)(PCHAR,VMHANDLE);
HDSC_Thunk          thunkVCOMM_OpenCommHook;
BOOL            (*VCOMM_CloseCommService)(COMMPORTHANDLE);
HDSC_Thunk          thunkVCOMM_CloseCommHook;
BOOL            (*VCOMM_ReadCommService)(COMMPORTHANDLE,PVOID,DWORD,PDWORD);
HDSC_Thunk          thunkVCOMM_ReadCommHook;
BOOL            (*VCOMM_WriteCommService)(COMMPORTHANDLE,PVOID,DWORD,PDWORD);
HDSC_Thunk          thunkVCOMM_WriteCommHook;


// A pointer to the device context. This is a hack to let
// these non-members of the device class get to its data.
CommhookDevice*CHD;

struct
    {
    char            name[100];
    COMMPORTHANDLE handle;
    } OpenPorts[10];

// My hook for the VCOMM_OpenComm service. I call the
// original handler, check the return value, then see
// if the opened port was the one I was watching for.
COMMPORTHANDLE VCOMM_OpenCommHook( PCHAR szPortName, VMHANDLE VMid )
    {
    COMMPORTHANDLE retval = VCOMM_OpenCommService( szPortName, VMid );

    if ( (retval != IE_BADID && retval != IE_OPEN) &&
         CHD->cpTargetPortName != NULL &&
         !stricmp( szPortName, CHD->cpTargetPortName ) )
        {
        CHD->hPort = retval;
        CHD->sAccessStats.dwOpenCount++;
        }

    return retval;
    }

// My hook for the VCOMM_CloseComm service. I check the port
// handle passed in to see if it was the one I was watching
// (or if I hadn't seen an open yet) then call the original
// service.
BOOL VCOMM_CloseCommHook( COMMPORTHANDLE hPort )
    {
    if ( CHD->hPort == hPort && CHD->hPort != NULL )
        {
        CHD->sAccessStats.dwCloseCount++;
        CHD->hPort = NULL;
        }
    else if ( CHD->hPort == NULL )
```

```
            {
            CHD->sAccessStats.dwNotMineCloseCount++;
            }

        return VCOMM_CloseCommService( hPort );
        }

    // My hook for the VCOMM_ReadComm service. I call the original
    // service, then check the port handle. If it was the one I
    // was watching (or if I hadn't seen an open yet), I add the
    // read data to a list of "trace blocks".
    BOOL VCOMM_ReadCommHook( COMMPORTHANDLE hPort, PVOID buf, DWORD nRequest, PDWORD pNRead )
        {
        BOOL retval = VCOMM_ReadCommService( hPort, buf, nRequest, pNRead );

        if ( CHD->hPort == hPort || CHD->hPort == NULL )
            {
            CHD->sAccessStats.dwReadCount++;
            CHD->sAccessStats.dwReadBytes -= *pNRead;

            CHD->AddTraceBlock( (unsigned char *)buf, false, *pNRead );
            }

        return retval;
        }

    // My hook for the VCOMM_WriteComm service. I call the original
    // service, then check the port handle. If it was the one I
    // was watching (or if I hadn't seen an open yet), I add the
    // written data to a list of "trace blocks".
    // NOTE THE USE OF THE MUTEX. SINCE I WANT TO BE ABLE TO WRITE
    // TO THE PORT (INJECTING MY OWN DATA), I SYNCHRONIZE ACCESS
    // TO THE VCOMM_WriteComm SERVICE. I DON'T REALLY KNOW IF THIS
    // IS NECESSARY.
    BOOL VCOMM_WriteCommHook( COMMPORTHANDLE hPort, PVOID buf, DWORD nRequest, PDWORD
pNWritten )
        {
        if ( CHD->hPort == hPort || CHD->hPort == NULL )
            {
            CHD->mutexWrite->enter();
            }

        BOOL rv = VCOMM_WriteCommService( hPort, buf, nRequest, pNWritten );

        if ( CHD->hPort == hPort || CHD->hPort == NULL )
            {
            CHD->mutexWrite->leave();
            CHD->sAccessStats.dwWriteCount++;
            CHD->sAccessStats.dwWriteBytes += *pNWritten;

            CHD->AddTraceBlock( (unsigned char *)buf, true, *pNWritten );
            }

        return rv;
        }

    // Remove the head of the trace block list and return it, adjusting
    // the total trace size.
    CDataBlock *CommhookDevice::GetNextTraceBlock()
```

```
    mutexTrace->enter();

    CDataBlock *pNext = m_pFirstTraceBlock;
    if ( m_pFirstTraceBlock != NULL )
        m_pFirstTraceBlock = m_pFirstTraceBlock->m_pNextBlock;
    if ( m_pFirstTraceBlock == NULL )
        m_pLastTraceBlock = NULL;

    if ( pNext != NULL )
        m_dwTraceSize -= pNext->m_dwLength;

    mutexTrace->leave();

    return pNext;
    }

// Add a block to the tail of the trace block list, adjusting the size.
void CommhookDevice::AddTraceBlock( unsigned char *pData, bool bWritten, DWORD dwLength )
    {
    if ( dwLength )
        {
        mutexTrace->enter();

        CDataBlock *pNewBlock = new CDataBlock( pData, bWritten, dwLength );
        if ( m_pFirstTraceBlock == NULL )
            m_pFirstTraceBlock = pNewBlock;
        else
            m_pLastTraceBlock->m_pNextBlock = pNewBlock;

        m_pLastTraceBlock = pNewBlock;

        m_dwTraceSize += pNewBlock->m_dwLength;

        mutexTrace->leave();
        }
    }

// Install the hooks. Note that the VCOMM services I'm hooking
// follow the C calling convention, so I use Hook_Device_Service_C
VOID CommhookDevice::DoHooks( void )
{
    // hook the open comm service
    VCOMM_OpenCommService = (COMMPORTHANDLE(*)(PCHAR,VMHANDLE))
            Hook_Device_Service_C( ___VCOMM_OpenComm,
                                   VCOMM_OpenCommHook,
                                   &thunkVCOMM_OpenCommHook);

    // hook the close comm service
    VCOMM_CloseCommService = (BOOL(*)(COMMPORTHANDLE))
            Hook_Device_Service_C( ___VCOMM_CloseComm,
                                   VCOMM_CloseCommHook,
                                   &thunkVCOMM_CloseCommHook);

    // hook the read comm service
    VCOMM_ReadCommService = (BOOL(*)(COMMPORTHANDLE,PVOID,DWORD,PDWORD))
            Hook_Device_Service_C( ___VCOMM_ReadComm,
                                   VCOMM_ReadCommHook,
                                   &thunkVCOMM_ReadCommHook);

    // hook the write comm service
```

```
    VCOMM_WriteCommService = (BOOL(*)(COMMPORTHANDLE,PVOID,DWORD,PDWORD))
            Hook_Device_Service_C( ___VCOMM_WriteComm,
                                   VCOMM_WriteCommHook,
                                   &thunkVCOMM_WriteCommHook);

}

// Uninstall the hooks with Unhook_Device_Service_C
VOID CommhookDevice::DoUnhooks( void )
{
    // unhook the open comm service
    Unhook_Device_Service_C(___VCOMM_OpenComm,
                            &thunkVCOMM_OpenCommHook);

    // unhook the close comm service
    Unhook_Device_Service_C(___VCOMM_CloseComm,
                            &thunkVCOMM_CloseCommHook);

    // unhook the read comm service
    Unhook_Device_Service_C(___VCOMM_ReadComm,
                            &thunkVCOMM_ReadCommHook);

    // unhook the write comm service
    Unhook_Device_Service_C(___VCOMM_WriteComm,
                            &thunkVCOMM_WriteCommHook);

}


void CommhookDevice::ClearStats( void )
    {
    sAccessStats.dwOpenCount =
        sAccessStats.dwCloseCount =
        sAccessStats.dwNotMineCloseCount =
        sAccessStats.dwReadCount =
        sAccessStats.dwReadBytes =
        sAccessStats.dwWriteCount =
        sAccessStats.dwWriteBytes = 0;
    }

void CommhookDevice::SetupData( void )
    {
    ClearStats();

    pHandles = NULL;
    nNumHandles = 0;

    CHD = this;

    mutexWrite = new VMutex;
    mutexTrace = new VMutex;

    cpTargetPortName = NULL;
    hPort = NULL;

    for ( int i=0; i<10; i++ )
        {
        memset( OpenPorts[i].name, 0, 100 );
        OpenPorts[i].handle = NULL;
        }
```

```
    FilteredRead =
        FilteredWrite =
        FilteredCount = 0;

    m_pFirstTraceBlock =
        m_pLastTraceBlock = NULL;
    m_dwTraceSize = 0;
    }


void CommhookDevice::DeleteHandles( void )
    {
    if ( pHandles != NULL )
        {
        int i;
        for ( i=0; i<nNumHandles; i++ )
            VWIN32_CloseVxDHandle( pHandles[i] );
        delete[] pHandles;
        pHandles = NULL;
        }
    }

void CommhookDevice::Cleanup( void )
    {
    DeleteHandles();
    delete mutexWrite;
    delete[] cpTargetPortName;

    mutexTrace->enter();
    CDataBlock *pBlock;
    while ( (pBlock = GetNextTraceBlock()) != NULL )
        delete pBlock;
    mutexTrace->leave();
    delete mutexTrace;
    }


BOOL CommhookDevice::OnSysCriticalInit(VMHANDLE hSysVM, PCHAR pszCmdTail, PVOID refData)
{
    SetupData();
    DoHooks();
    return TRUE;
}

VOID CommhookDevice::OnSysCriticalExit()
{
    DoUnhooks();
    Cleanup();
}

BOOL CommhookDevice::OnSysDynamicDeviceInit()
{
    SetupData();
    DoHooks();
    return TRUE;
}

BOOL CommhookDevice::OnSysDynamicDeviceExit()
{
    DoUnhooks();
```

```
    Cleanup();
    return TRUE;
}

// My client app uses DeviceIoControl() to communicate with
// this VxD.
DWORD CommhookDevice::OnW32DeviceIoControl(PIOCTLPARAMS pDIOCParams)
{

   switch ( pDIOCParams->dioc_IOCtlCode )
        {
        default:
             *pDIOCParams->dioc_bytesret = -1;
             break;

        case 0:   //DIOC_OPEN:
        case -1: //DIOC_CLOSE:
             *pDIOCParams->dioc_bytesret = -2;
             break;

        // Get statistics on the number of opens, closes, etc.
        // pDIOCParams->dioc_OutBuf must point to an _sAccessStats structure.
        case _CommHook_DIOC_AccessStats:
             memcpy( pDIOCParams->dioc_OutBuf, &sAccessStats, sizeof(sAccessStats) );
             *pDIOCParams->dioc_bytesret = sizeof(sAccessStats);
             break;

        // Clear the statistics.
        case _CommHook_DIOC_ClearStats:
             ClearStats();
             *pDIOCParams->dioc_bytesret = 0;
             break;

        // Pass in handles to events that the VxD can use to signal
        // the app. The app must have used OpenVxDHandle() from Kernel32
        // to create the Ring0 handle.
        // pDIOCParams->dioc_cbInBuf is the count of handles
        // pDIOCParams->dioc_InBuf is an array of handles
        case _CommHook_DIOC_Handles:
             {
             DeleteHandles();

             nNumHandles = (int)pDIOCParams->dioc_cbInBuf;
             pHandles = new HANDLE[ nNumHandles ];

             for ( int i=0; i<nNumHandles; i++ )
                  pHandles[i] = ((HANDLE *)pDIOCParams->dioc_InBuf)[i];

             *pDIOCParams->dioc_bytesret = 0;
             }
             break;

        // Set the port name that we want to watch for.
        // pDIOCParams->dioc_cbInBuf is the length of the string not including
        //       the terminating NULL.
        // pDIOCParams->dioc_InBuf points to a NULL-terminated string
        case _CommHook_DIOC_SetTargetPort:
             delete[] cpTargetPortName;

             if ( pDIOCParams->dioc_cbInBuf )
```

```
                    {
                    cpTargetPortName = new char[ pDIOCParams->dioc_cbInBuf + 1 ];
                    strcpy( cpTargetPortName, (char *)pDIOCParams->dioc_InBuf );
                    }
                else
                    cpTargetPortName = NULL;
                *pDIOCParams->dioc_bytesret = 0;
                break;

            // Write a block of data to the open port.
            // pDIOCParams->dioc_cbInBuf is the number of bytes to write
            // pDIOCParams->dioc_InBuf is a pointer to the data.
            // on exit, pDIOCParams->dioc_bytesret contains the number of
            //       bytes actually written.
            case _CommHook_DIOC_WriteToPort:
                *pDIOCParams->dioc_bytesret = 0;
                if ( hPort != NULL )
                    {
                    int offset = 0;
                    DWORD remaining = pDIOCParams->dioc_cbInBuf;
                    DWORD written;
                    mutexWrite->enter();
                    while ( remaining != 0 )
                        {
                        if ( ! VCOMM_WriteCommService( hPort,
                                                        (char *)pDIOCParams->dioc_InBuf +
offset,
                                                        remaining, &written ) )
                            break;
                        remaining -= written;
                        *pDIOCParams->dioc_bytesret += written;
                        }
                    mutexWrite->leave();
                    }
                break;

            // Read filtered data (whose generation is not yet implemented!!!)
            // pDIOCParams->dioc_cbOutBuf is the size of a buffer to fill
            // pDIOCParams->dioc_OutBuf is the app-supplied buffer
            // pDIOCParams->dioc_bytesret will contain the number of bytes
            //   actually read.
            case _CommHook_DIOC_ReadFilteredData:
                *pDIOCParams->dioc_bytesret = 0;
                if ( hPort != NULL &&
                     FilteredCount != 0 )
                    {
                    for ( int i=0; i<pDIOCParams->dioc_cbOutBuf && FilteredCount; i++ )
                        {
                        ((unsigned char *)pDIOCParams->dioc_OutBuf)[i] =
FilteredData[FilteredRead++];
                        FilteredRead &= 0x3FF;
                        FilteredCount--;
                        }
                    *pDIOCParams->dioc_bytesret = i;
                    }
                break;

            // Read trace (Tx and Rx) data from the trace block list.
            // pDIOCParams->dioc_cbOutBuf is the number of bytes to read,
            //       or zero if the app is asking how much data there is.
```

```
            // pDIOCParams->dioc_OutBuf points to the app-supplied buffer. It is
            //       ignored if dioc_cbOutBuf is zero. NOTE THAT THIS BUFFER
            //       MUST ACTUALLY BE TWICE THE SIZE OF THE DATA TO READ, AS WE
            //       GET A WORD FOR EVERY BYTE - THE HIGH BYTE CONTAINS 1 IF THE
            //       LOW BYTE WAS A TRANSMITTED BYTE, OR ZERO IF THE LOW BYTE
            //       WAS A RECEIVED BYTE.
            // On exit, pDIOCParams->dioc_bytesret will contain the number of bytes
            //       actually read, or, if dioc_cbOutBuf was zero, the total number
            //       of bytes of trace data available.
            case _CommHook_DIOC_ReadTraceData:
                 {
                 DWORD remaining = pDIOCParams->dioc_cbOutBuf;
                 if ( ! remaining )      // zero is special - caller wants to know
                      {                        // how much data there is
                      *pDIOCParams->dioc_bytesret = m_dwTraceSize;
                      }
                 else
                      {
                      DWORD offset = 0;
                      CDataBlock *pBlock;
                      while ( (pBlock = GetNextTraceBlock()) != NULL &&
                                pBlock->m_dwLength <= remaining )
                           {
                           for ( int i=0; i<pBlock->m_dwLength; i++ )
                                ((unsigned short int *)pDIOCParams->dioc_OutBuf)
[offset+i] =
                                          ((pBlock->m_bWritten ? 1 : 0) << 8) |
                                          pBlock->m_pData[i];
                           offset += pBlock->m_dwLength;
                           remaining -= pBlock->m_dwLength;
                           delete pBlock;
                           }
                      *pDIOCParams->dioc_bytesret = offset;
                      }
                 }
                 break;
            }

       return DEVIOCTL_NOERROR;
}
```

CommHook 头文件如下：

```
// COMMHOOK.h - include file for VxD COMMHOOK
#ifndef _COMMHOOK_H
#define _COMMHOOK_H

#ifdef   DEVICE_MAIN
#include <vtoolscp.h>
#endif

typedef struct
     {
     DWORD          dwOpenCount;
     DWORD          dwCloseCount;
     DWORD          dwNotMineCloseCount;
     DWORD          dwReadCount;
     DWORD          dwReadBytes;
     DWORD          dwWriteCount;
     DWORD          dwWriteBytes;
```

```
    } _sAccessStats;

#define  _CommHook_DIOC_AccessStats       1
#define _CommHook_DIOC_ClearStats         2
#define _CommHook_DIOC_Handles            3
#define _CommHook_DIOC_SetTargetPort      4
#define _CommHook_DIOC_WriteToPort        5
#define _CommHook_DIOC_ReadFilteredData   6
#define _CommHook_DIOC_ReadTraceData  7

#ifdef   DEVICE_MAIN
#define DEVICE_CLASS         CommhookDevice
#define COMMHOOK_DeviceID        UNDEFINED_DEVICE_ID
#define COMMHOOK_Init_Order UNDEFINED_INIT_ORDER
#define COMMHOOK_Major       1
#define COMMHOOK_Minor       0

class CDataBlock
    {
    private:
        CDataBlock() {}
    public:
        CDataBlock( unsigned char *pData, bool bWritten, DWORD dwLength )
            {
            m_pData = new unsigned char[ m_dwLength = dwLength ];
            memcpy( m_pData, pData, m_dwLength );
            m_bWritten = bWritten;
            m_pNextBlock = NULL;
            }
        ~CDataBlock()
            {
            delete[] m_pData;
            }

        unsigned char *m_pData;
        bool          m_bWritten;
        DWORD              m_dwLength;

        CDataBlock         *m_pNextBlock;
    };

class CommhookDevice : public VDevice
{
public:
    virtual BOOL OnSysCriticalInit(VMHANDLE hSysVM, PCHAR pszCmdTail, PVOID refData);
    virtual VOID OnSysCriticalExit();
    virtual BOOL OnSysDynamicDeviceInit();
    virtual BOOL OnSysDynamicDeviceExit();
    virtual DWORD OnW32DeviceIoControl(PIOCTLPARAMS pDIOCParams);

    VOID DoHooks( void );
    VOID DoUnhooks( void );
    void DeleteHandles( void );
    void SetupData( void );
    void ClearStats( void );
    void Cleanup( void );

    HANDLE             *pHandles;
    int                nNumHandles;
    _sAccessStats sAccessStats;
```

```
    COMMPORTHANDLE hPort;
    char           *cpTargetPortName;

    VMutex    *mutexWrite;

    unsigned char FilteredData[0x400];
    DWORD              FilteredCount;
    DWORD              FilteredRead;
    DWORD              FilteredWrite;

    VMutex             *mutexTrace;
    DWORD              m_dwTraceSize;
    CDataBlock         *m_pFirstTraceBlock;
    CDataBlock         *m_pLastTraceBlock;
    void           AddTraceBlock( unsigned char *pData, bool bWritten, DWORD dwLength );
    CDataBlock         *GetNextTraceBlock();
};

class CommhookVM : public VVirtualMachine
{
public:
    CommhookVM(VMHANDLE hVM);
};

class CommhookThread : public VThread
{
public:
    CommhookThread(THREADHANDLE hThread);
};

#endif    // DEVICE_MAIN
#endif    // _COMMHOOK_H
```

有了以上的源代码，还要在编程环境中编译生成 VxD 驱动程序。需要提醒的是，要想顺利地理解这个程序，需要掌握的知识是较多的，包括硬件和软件知识。

12.3　串口数据捕捉实例程序

12.3.1　编程任务

同一台计算机中，在有其他应用程序或设备占用一个串口或多个串口的情况下，捕捉到这些串口的通信数据，显示在界面上，需要时可以保存成文件。

编程环境：Windows 98，Visual C++ 6.0。

在编程之前，先将 commhook.vxd 复制到系统文件夹：windows/system 中。

详细编程过程如下。

12.3.2　编程步骤

（1）建立应用程序工程 SCommTest，并在主对话框添加控件

打开 Visual C++ 6.0，建立一个基于对话框的 MFC 应用程序：SerialPortVxD。然后，在主对话框中添加控件，最后效果如图 12.3.1 所示。

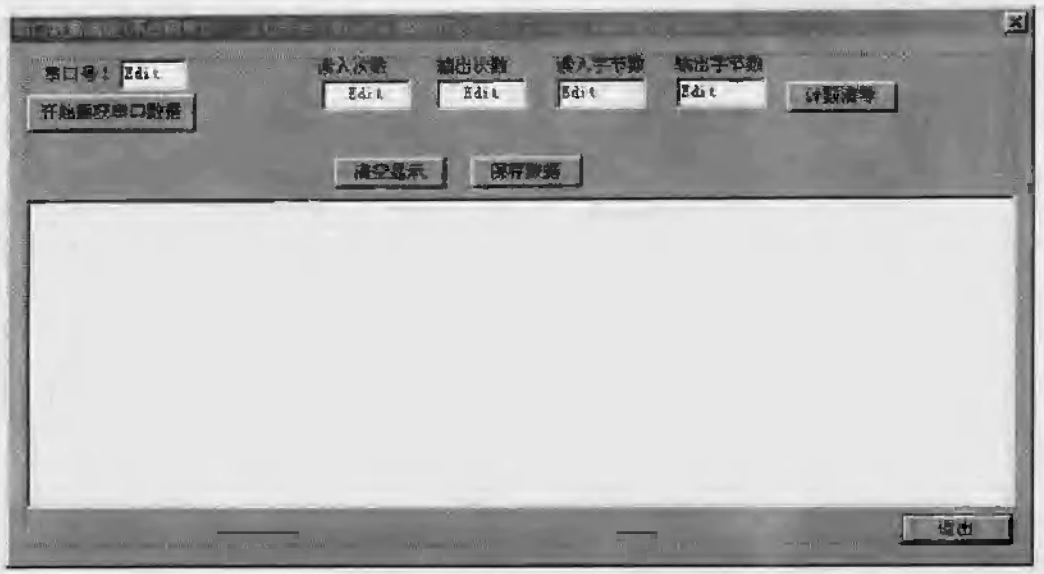


图 12.3.1　主对话框

以上控件相应的变量及属性如表 12-3-1 所示。

表 12-3-1 控件及其属性设置

控件	控件 ID	Caption	需要添加的变量及变量类型
静态文本	IDC_STATIC	串口号	CString　m_strPortName
静态文本	IDC_STATIC	读入次数	
静态文本	IDC_STATIC	输出次数	
静态文本	IDC_STATIC	读入字节数	
静态文本	IDC_STATIC	输出字节数	
编辑框	IDC_EDIT_READS		
编辑框	IDC_EDIT_PORT		
编辑框	IDC_EDIT_WRITES		
编辑框	IDC_EDIT_READBYTES		
编辑框	IDC_EDIT_WRITEBYTES		
按钮	IDC_BUTTON_STARTSTOP	开始截获串口数据	
按钮	IDC_BUTTON_STATRESET	计数清零	
按钮	IDC_BUTTON_CLEARDISP	清空显示	
按钮	IDC_BUTTON_SAVEDATA	保存数据	
列表控件	IDC_LIST_DISPDATA		按图 12.3.2 进行属性设置 CMyListCtrl m_ctrlListDispData

添加的列表控件 List Control，为了下面的显示方便，需要按图 12.3.2 进行属性设置。

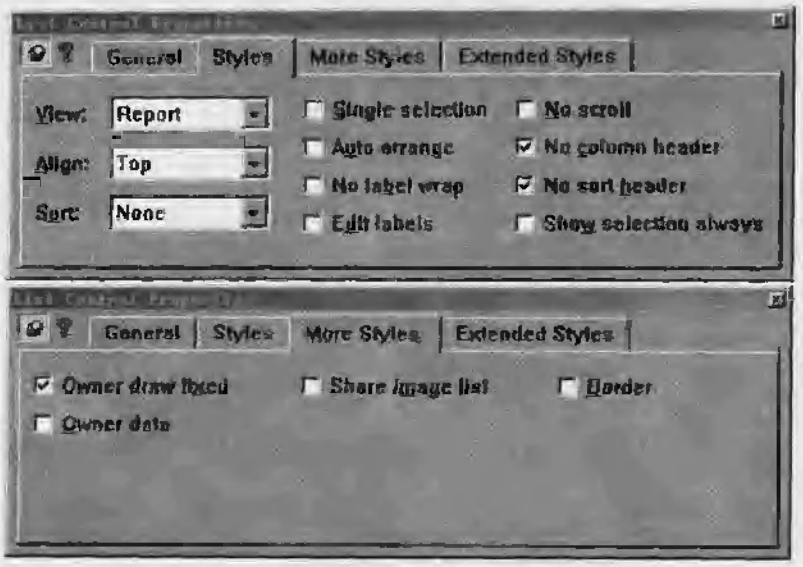


图 12.3.2　列表控件属性设置

（2）添加 VxD 驱动文件，放入系统文件夹中，并加入工程

在程序中，要用到 Commhook.vxd 和 Commhook.h 两个文件，由于需要动态加载设备驱动程序 Commhook.vxd，应将其放到 Windows 系统文件夹中，如操作系统安装在 C 盘，则一般为 C:\Windows\System，可以直接将 Commhook.vxd 复制到系统文件夹中。

再将 Commhook.h 文件复制到工程项目文件夹中。并在 SerialPortVxDDlg.h 中添加：

```
#include "CommHook.h"  //设备驱动头文件
```

（3）添加改进的列表类和记录类

为了解决数据的显示和保存，程序中加入了两个类。从 CListCtrl 类派生的 CMyListCtrl 类。添加方法如图 12.3.3 所示。

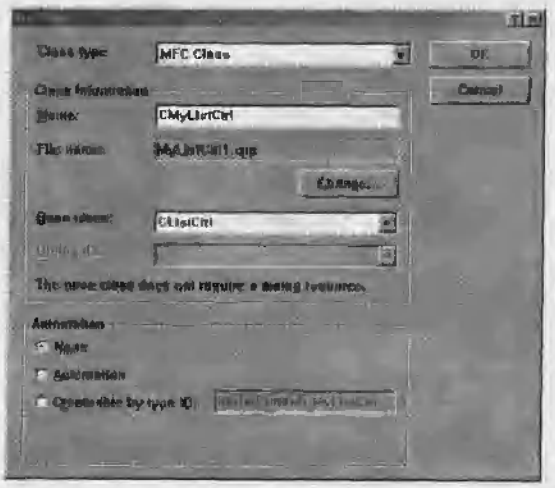


图 12.3.3　添加新类 CListCtrl

CMylistctrl.cpp 的代码如下：

```
// MyListCtrl.cpp : implementation file
//

#include "stdafx.h"
#include "serialportvxd.h"
#include "MyListCtrl.h"
#include "SerialPortVxDDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMyListCtrl

CMyListCtrl::CMyListCtrl()
{
}

CMyListCtrl::~CMyListCtrl()
{
}


BEGIN_MESSAGE_MAP(CMyListCtrl, CListCtrl)
    //{{AFX_MSG_MAP(CMyListCtrl)
    ON_NOTIFY_REFLECT(LVN_DELETEITEM, OnDeleteitem)
    ON_WM_MEASUREITEM_REFLECT()
    ON_WM_ERASEBKGND()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMyListCtrl message handlers

void CMyListCtrl::OnDeleteitem(NMHDR* pNMHDR, LRESULT* pResult)
{
    NM_LISTVIEW* pNMListView = (NM_LISTVIEW*)pNMHDR;
    // TODO: Add your control notification handler code here
    delete (CListRecord *)pNMListView->lParam;
    *pResult = 0;
}

void CMyListCtrl::MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
    CDC dc;
    dc.CreateCompatibleDC(NULL);
    CFont *oldFont = dc.SelectObject(&((CSerialPortVxDDlg
*)GetParent())->m_fontTrace);
    CSize te = dc.GetTextExtent( "0", 1 );
    lpMeasureItemStruct->itemHeight = ((CSerialPortVxDDlg
*)GetParent())->m_nCharHeight = te.cy - 2;
    ((CSerialPortVxDDlg *)GetParent())->m_nCharWidth = te.cx;
    lpMeasureItemStruct->itemWidth = ((CSerialPortVxDDlg *)GetParent())->m_nCharWidth
* 55;
```

```
void CMyListCtrl::DrawItem( LPDRAWITEMSTRUCT lpDrawItemStruct )
{
    int index = lpDrawItemStruct->itemID;
    TRACE("%d\n", index );
    if ( index < 0 )
        return;

    int y = lpDrawItemStruct->rcItem.top;

    HDC hDC = lpDrawItemStruct->hDC;
    HFONT oldFont = (HFONT)::SelectObject( hDC, (HFONT)((CSerialPortVxDDlg
*)GetParent())->m_fontTrace );
    int oldBkMode = ::SetBkMode( hDC, OPAQUE );
    COLORREF oldTextColor = ::SetTextColor( hDC, RGB(0,0,0) );
    COLORREF oldBkColor = ::SetBkColor( hDC, RGB(255,255,255) );

    CListRecord *pRec = (CListRecord *)lpDrawItemStruct->itemData;
    CString str;
    for ( int i=0; i<(int)pRec->m_dwLength; i++ )
        {
        if ( pRec->m_pwData[i] & 0xFF00 )
            {
            ::SetTextColor( hDC, RGB(0,0,0) );
            ::SetBkColor( hDC, RGB(255,255,255) );
            }
        else
            {
            ::SetTextColor( hDC, RGB(255,255,255) );
            ::SetBkColor( hDC, RGB(0,0,0) );
            }
        str.Format( "%2.2X ", pRec->m_pwData[i] & 0x00FF );
        ::TextOut( hDC, ((CSerialPortVxDDlg *)GetParent())->m_nCharWidth*3*i, y, str,
3 );
        }
    ::SetTextColor( hDC, RGB(0,0,0) );
    ::SetBkColor( hDC, RGB(255,0,0) );
    for ( ; i<32; i++ )
        ::TextOut(hDC, ((CSerialPortVxDDlg*)GetParent())->m_nCharWidth*3*i, y, "  ",
3 );

    ::SetTextColor( hDC, oldTextColor );
    ::SetBkColor( hDC, oldBkColor );
    ::SetBkMode( hDC, oldBkMode );
    ::SelectObject( hDC, oldFont );
}


BOOL CMyListCtrl::OnEraseBkgnd(CDC* pDC)
{
    if ( ! GetItemCount() )
        return CListCtrl::OnEraseBkgnd( pDC );
    return TRUE;
}
```

头文件 MyListCtrl.h 代码清单如下：

```
// MyListCtrl.h : header file
/////////////////////////////////////////////////////////////////////////////
// CListRecord: 列表数据记录类
```

```
class CListRecord
    {
    public:
        CListRecord()
            {
            m_dwLength = 0;
            }
        ~CListRecord() {}

        WORD m_pwData[32];
        DWORD    m_dwLength;
    };

/////////////////////////////////////////////////////////////////////////////
// CMyListCtrl window

class CMyListCtrl : public CListCtrl
{
// Construction
public:
    CMyListCtrl();

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMyListCtrl)
    virtual void DrawItem( LPDRAWITEMSTRUCT lpDrawItemStruct );
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMyListCtrl();

   afx_msg BOOL OnEraseBkgnd(CDC* pDC);

    // Generated message map functions
protected:
    //{{AFX_MSG(CMyListCtrl)
    afx_msg void OnDeleteitem(NMHDR* pNMHDR, LRESULT* pResult);
    //}}AFX_MSG
    afx_msg void MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);

    DECLARE_MESSAGE_MAP()
};
```

在上面头文件中，还新添加了一个记录类，直接把它放在列表类 MyListCtrl.h 文件中，实际上，它相当一个数据结构（类本来就是结构的延伸）。

然后，在 ClassWizard 中为列表控件添加控制变量 m_ctrlListDispData，如图 12.3.4 所示。

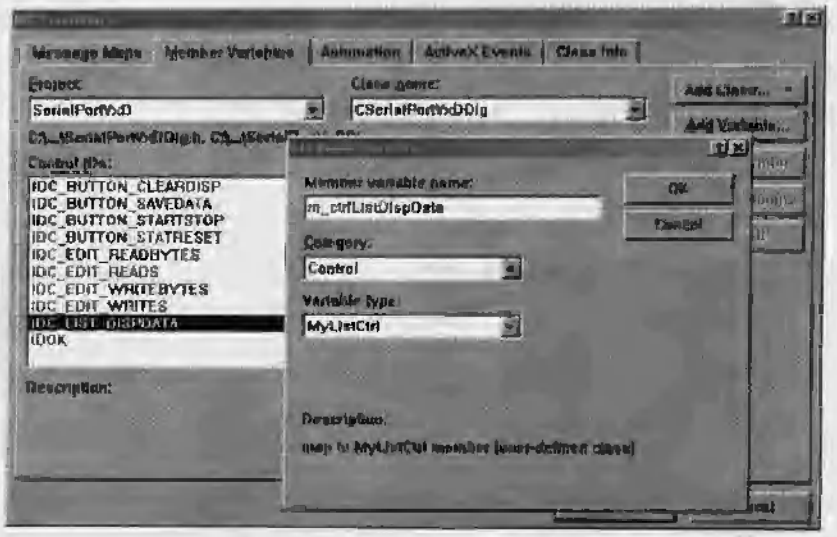


图 12.3.4　为 IDC_LIST_DISPDATA 列表控件添加控制变量

（4）添加 CSerialPortVxDDlg 类成员变量并初始化

在 SerialPortVxDDlg.h 中，为 CSerialPortVxDDlg 类添加以下公有成员变量。在这些变量中，m_hVxD 是串口设备句柄。

```
public:
    bool        m_bRunning;          //是否开始捕捉数据
    HANDLE      m_hVxD;             //虚拟设备句柄
    _sAccessState m_sAccessData;
    CEvent      m_Event;
    DWORD       m_dwEventRing0Handle;
    char *m_cpTargetPortName;
    CFont       m_fontTrace;
    LOGFONT     m_logfontTrace;
    int         m_nCharWidth;
    int         m_nCharHeight;
    CListRecord *m_pCurRec;
```

然后，在类构造函数中对变量进行初始化：

```
CSerialPortVxDDlg::CSerialPortVxDDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CSerialPortVxDDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CSerialPortVxDDlg)
    m_strPortName = _T("");
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

    //变量的初始化
    m_bRunning = false;
    m_cpTargetPortName = NULL;
    m_hVxD = NULL;
    CFont font;
    font.CreatePointFont( 100, _T( "Courier New" ) );
    font.GetLogFont( &m_logfontTrace );
```

```
        m_fontTrace.CreateFontIndirect( &m_logfontTrace );
        m_pCurRec = NULL;
    }
```

并在 OnInitDialog()函数中进行程序的初始化设置：

```
    BOOL CSerialPortVxDDlg::OnInitDialog()
    {
        CDialog::OnInitDialog();
        ……
    // TODO: Add extra initialization here
        // 事件传送
        HINSTANCE hKernel32 = LoadLibrary("KERNEL32");
        DWORD (*fpCVH)(HANDLE) =
            (DWORD (*)(HANDLE)) GetProcAddress(hKernel32, "OpenVxDHandle" );

        FreeLibrary(hKernel32);

        CRect rect;
        m_ctrlListDispData.GetClientRect( &rect );
        m_ctrlListDispData.InsertColumn( 0, NULL, LVCFMT_LEFT,
rect.right-GetSystemMetrics(SM_CXVSCROLL) );

        //列表中的 WM_MEASUREITEM 映射
        m_ctrlListDispData.GetWindowRect( &rect );
        WINDOWPOS wp;
        wp.hwnd = m_ctrlListDispData.m_hWnd;
        wp.cx = rect.Width();
        wp.cy = rect.Height();
        wp.flags = SWP_NOACTIVATE | SWP_NOMOVE | SWP_NOOWNERZORDER | SWP_NOZORDER;
        m_ctrlListDispData.SendMessage( WM_WINDOWPOSCHANGED, 0, (LPARAM)&wp );
        // 准备好第一个记录
        m_pCurRec = new CListRecord;
        int item = m_ctrlListDispData.InsertItem( 0, NULL );
        m_ctrlListDispData.SetItemData( item, (DWORD)m_pCurRec );
        return TRUE;  // return TRUE unless you set the focus to a control
    }
```

（5）启动串口数据捕捉

首先为按钮 IDC_BUTTON_STARTSTOP 添加单击响应函数 OnButtonStartstop()，该函数的功能是打开虚拟设备文件，并用 CreateFile 函数取得设备句柄，若打开成功，就启动一个定时器，每 200ms 去查询一次串口通信数据。

在 OnButtonStartstop()下添加如下代码：

```
    void CSerialPortVxDDlg::OnButtonStartstop()
    {
        // TODO: Add your control notification handler code here
        m_bRunning = !m_bRunning;
        if ( m_bRunning )
        {
            UpdateData( TRUE );
            if ( m_strPortName.IsEmpty() )
                m_strPortName = "NONE";
            delete[] m_cpTargetPortName;
            m_cpTargetPortName = new char[ m_strPortName.GetLength() + 1 ];
            strcpy( m_cpTargetPortName, m_strPortName );

            // 以下的设置是把串口设备驱动文件 CommHook.vxd 放在系统文件夹中
```

```
            // WINSYSDIR 一般在 C:\Windows\System
            //CreateFile 函数中用"\\\\.\\CommHook.vxd"
            m_hVxD = CreateFile( "\\\\.\\CommHook.vxd",
                                 0, 0, NULL, 0,
                                 FILE_FLAG_DELETE_ON_CLOSE, NULL );
            if ( m_hVxD == INVALID_HANDLE_VALUE )
            {
                AfxMessageBox( "无法打开虚拟设备驱动文件 commhook.VxD -\n\n(请检查是否放在系统
文件夹: \Windows\System)", MB_OK );
                m_bRunning = false;
                m_hVxD = NULL;
                return;
            }
            GetDlgItem( IDC_BUTTON_StartStop )->SetWindowText( "停止截获串口数据" );

            DWORD result;

            DeviceIoControl(m_hVxD,
                            _CommHook_DIOC_SetTargetPort,
                            m_cpTargetPortName,
                            strlen(m_cpTargetPortName),
                            NULL,
                            0,
                            &result,
                            NULL );
            //如果打开设备驱动文件成功，就启动定时器每 200ms 查询一次数据
            SetTimer( 1000, 200, NULL );
        }
        else
        {
            KillTimer( 1000 ); //停止定时
            GetDlgItem( IDC_BUTTON_StartStop )->SetWindowText( "开始截获串口数据" );
            CloseHandle( m_hVxD ); //关闭虚拟设备句柄
            m_hVxD = NULL;
        }
    }
```

添加定时消息响应函数 OnTimer，方法如图 12.3.5 所示。

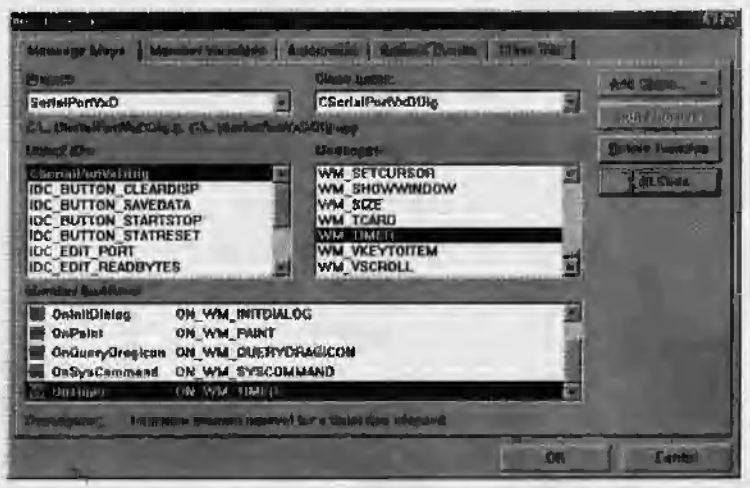


图 12.3.5　为 CSerialPortVxDDlg 类添加定时消息函数

然后添加如下代码：

```
void CSerialPortVxDDlg::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    BOOL status;
    DWORD nBytes;

    status = DeviceIoControl(   m_hVxD,
                                    _CommHook_DIOC_AccessStats,
                                    NULL,
                                    0,
                                    &m_sAccessData,
                                    sizeof(m_sAccessData),
                                    &nBytes,
                                    NULL );

    // 得到串口设备中的数据量的大小（字节数）.
    status = DeviceIoControl(   m_hVxD,
                                    _CommHook_DIOC_ReadTraceData,
                                    NULL,
                                    0,
                                    NULL,
                                    0,
                                    &nBytes,
                                    NULL );
    if ( nBytes )
    {
        ASSERT( nBytes < 10000 );
        DWORD request = nBytes;
        // 这里，每个字节占一个字的容量，高字节为Tx/Rx（发送/接收）标志
        unsigned short int *pData = new unsigned short int[ request ];
        status = DeviceIoControl(   m_hVxD,
                                        _CommHook_DIOC_ReadTraceData,
                                        NULL,
                                        0,
                                        pData,
                                        request,
                                        &nBytes,
                                        NULL );
        ASSERT( request == nBytes );

        for ( int i=0; i<(int)nBytes; i++ )
        {
            m_pCurRec->m_pwData[m_pCurRec->m_dwLength++] = pData[i];

            if ( m_pCurRec->m_dwLength == 32 )
            {
                int n = m_ctrlListDispData.GetItemCount();
                m_ctrlListDispData.RedrawItems( n-1, n-1 );

                m_pCurRec = new CListRecord;
                m_ctrlListDispData.InsertItem( n, NULL );
                m_ctrlListDispData.SetItemData( n, (DWORD)m_pCurRec );
            }
        }
        int n = m_ctrlListDispData.GetItemCount();
        m_ctrlListDispData.RedrawItems( n-1, n-1 );

        for ( i=m_ctrlListDispData.GetItemCount(); i>1000; i-- )
```

```
                m_ctrlListDispData.DeleteItem(0);
            m_ctrlListDispData.EnsureVisible( m_ctrlListDispData.GetItemCount()-1,
FALSE );

            delete[] pData;
        }

        if ( status )
        {
            CString str;
            //str.Format( "%ld", m_sAccessData.dwOpenCount );
            //GetDlgItem( IDC_EDIT_Opens )->SetWindowText( str );
            //str.Format( "%ld", m_sAccessData.dwCloseCount );
            //GetDlgItem( IDC_EDIT_Closes )->SetWindowText( str );
            //str.Format( "%ld", m_sAccessData.dwNotMineCloseCount );
            //GetDlgItem( IDC_EDIT_NotMineCloses )->SetWindowText( str );
            str.Format( "%ld", m_sAccessData.dwReadCount );
            GetDlgItem( IDC_EDIT_READS )->SetWindowText( str );
            str.Format( "%ld", m_sAccessData.dwWriteCount );
            GetDlgItem( IDC_EDIT_WRITES )->SetWindowText( str );
            str.Format( "%ld", m_sAccessData.dwReadBytes );
            GetDlgItem( IDC_EDIT_READBYTES )->SetWindowText( str );
            str.Format( "%ld", m_sAccessData.dwWriteBytes );
            GetDlgItem( IDC_EDIT_WRITEBYTES )->SetWindowText( str );
        }

        CDialog::OnTimer(nIDEvent);
    }
```

（6）实现其他辅助功能

计数清零是为了清除计算次数，并重新开始计数。为按钮 IDC_BUTTON_STATRESET 添加单击响应函数 OnButtonStatreset()，并加入以下代码：

```
void CSerialPortVxDDlg::OnButtonStatreset()
{
    // TODO: Add your control notification handler code here
    if ( m_hVxD != NULL )
    {
        DWORD result;
        DeviceIoControl(m_hVxD,
                        _CommHook_DIOC_ClearStats,
                        NULL,
                        0,
                        NULL,
                        0,
                        &result,
                        NULL);
    }

    GetDlgItem( IDC_EDIT_Opens )->SetWindowText( "0" );
    GetDlgItem( IDC_EDIT_Closes )->SetWindowText( "0" );
    GetDlgItem( IDC_EDIT_READS )->SetWindowText( "0" );
    GetDlgItem( IDC_EDIT_WRITES )->SetWindowText( "0" );
    GetDlgItem( IDC_EDIT_READBYTES )->SetWindowText( "0" );
    GetDlgItem( IDC_EDIT_WRITEBYTES )->SetWindowText( "0" );
}
```

现在来实现清空显示数据功能，为按钮 IDC_BUTTON_CLEARDISP 添加单击响应函数 OnButtonCleardisp()，并加入以下代码：

```
void CSerialPortVxDDlg::OnButtonCleardisp()
{
    // TODO: Add your control notification handler code here
    m_ctrlListDispData.DeleteAllItems();
    m_pCurRec = new CListRecord;
    int item = m_ctrlListDispData.InsertItem( 0, NULL );
    m_ctrlListDispData.SetItemData( item, (DWORD)m_pCurRec );
}
```

将显示数据保存到文件，为按钮 IDC_BUTTON_SAVEDATA 添加单击响应函数 OnButtonSavedata()，并加入以下代码：

```
void CSerialPortVxDDlg::OnButtonSavedata()
{
    // TODO: Add your control notification handler code here
    FILE *fp = fopen( "CHT.log", "wb" );
    int count = m_ctrlListDispData.GetItemCount();
    WORD status = 10;
    int cols = 0;
    bool getLen = true;
    WORD len=0;

    for ( int i=0; i<count; i++ )
    {
        CListRecord *pRec = (CListRecord *)m_ctrlListDispData.GetItemData( i );

        for ( int j=0; j<(int)pRec->m_dwLength; j++ )
        {
            WORD w = pRec->m_pwData[j];

            WORD type = w >> 8;

            if ( type != status || !len)
            {
                fprintf( fp, "\x0D\x0A" );
                status = type;
                if ( status == 0 ) // receive
                    fprintf( fp, "Rx:" );
                else
                    fprintf( fp, "Tx:" );
                cols = 0;
                getLen = true;
            }

            ++cols;
            fprintf( fp, " %2.2X", w & 0xFF );
            len --;

            if ( getLen && cols == 9 )
            {
                len = (w & 0xFF) + 1;
                getLen = false;
            }
        }
    }

    fclose( fp );
}
```

到这里，程序基本编写完成。下面列出 SerialPortVxDDlg.h 的程序清单：

```
// SerialPortVxDDlg.h : header file
//
… …
#include <afxmt.h> //定义 CEVent 类
#include "CommHook.h" //设备驱动头文件
#include "MyListCtrl.h"

/////////////////////////////////////////////////////////////////////////////
// CSerialPortVxDDlg dialog


class CSerialPortVxDDlg : public CDialog
{
// Construction
public:
    CSerialPortVxDDlg(CWnd* pParent = NULL);  // standard constructor

// Dialog Data
    //{{AFX_DATA(CSerialPortVxDDlg)
    enum { IDD = IDD_SERIALPORTVXD_DIALOG };
    CMyListCtrl   m_ctrlListDispData;
    CString  m_strPortName;
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CSerialPortVxDDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
public:
    bool m_bRunning;
    HANDLE    m_hVxD;
    _sAccessStats m_sAccessData;
    CEvent    m_Event;
    DWORD     m_dwEventRing0Handle;
    char *m_cpTargetPortName;
    CFont     m_fontTrace;
    LOGFONT   m_logfontTrace;
    int       m_nCharWidth;
    int       m_nCharHeight;
    CListRecord *m_pCurRec;


protected:
    HICON m_hIcon;

    // Generated message map functions
    //{{AFX_MSG(CSerialPortVxDDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg void OnButtonStartstop();
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void OnButtonStatreset();
    afx_msg void OnButtonCleardisp();
    afx_msg void OnButtonSavedata();
    //}}AFX_MSG
```

```
    DECLARE_MESSAGE_MAP()
};
```

下面对示例程序进行测试。

打开串口调试助手，设置在 COM1，在发送框中填上任意的十六进字符如：12 34 56 35 00 78 90，选上“十六制发送”。再运行本示例程序，单击“开始截获串口数据”按钮，再单击串口调试助手的“手动发送命令”，就可以看到通过串口 1 发出的数据显示在示例程序的数据显示列表框中。测试结果如图 12.3.6 所示。

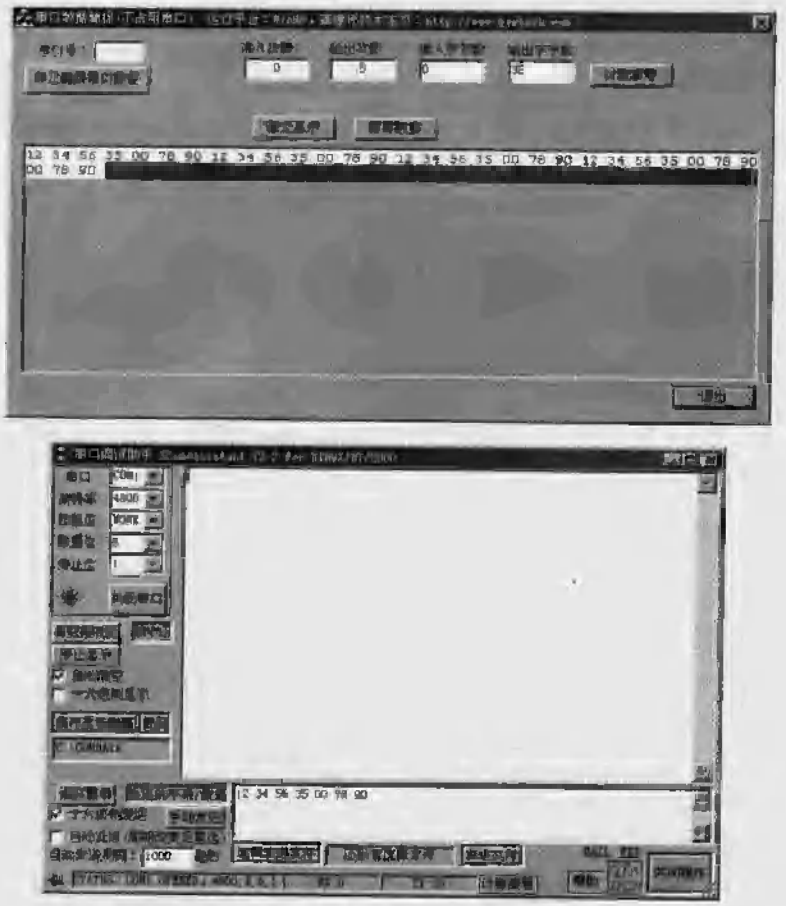


图 12.3.6　示例程序测试结果

最后讨论一下程序的功能。实际上，如果再进一步测试，就可以发现，这个例程运行后，如果同一台计算机上有多个串口都在工作，列表框中将显示所有串口的收发数据，而且我们也注意到，程序中并不需要进行串口参数的设置。

12.4 虚拟串口简介

虚拟串口在这里是相对物理串口而言的，直观地讲，就是用软件模拟出串口，其他应用程序使用这些串口时，就像与应用真正的物理串口一样。本书第 9.2 节提到的 MOXA 公司的

联网服务器也可以理解为一种虚拟串口的。本节我们只提供一些应用思路，当计算机串口资源有限时，在项目设计时就可以用到虚拟串口。

虚拟串口有两种模式：一种是硬件与软件结合来模拟串口，另一种纯软件来模拟串口。应用于 PC 机的多串口卡、串口联网设备、USB 转串口设备等，都是用硬件与软件相结合来模拟串口的，比如，一台笔记本上没有串口，就可以买到一个 USB 转串口装置，再安装驱动软件，就可以通过应用程序进行串口通信编程了。

还有纯软件方式的虚拟串口，比如在同一台计算上用软件模拟出两个串口，这两个串口就能像正常串口一样进行数据通信。对于纯软件方式的虚拟串口，这里推荐一个软件，读者可以去体验一下：商业软件 Virtual Serial Ports Driver （http://www.eltima.com）是一个较优秀的纯软件虚拟串口，有 14 天的试用期。

虚拟串口的编程属于驱动编程，敢于挑战自我的学习者与技术人员可以去尝试这些“高深”的技术。

附录 A　Turbo C 说明

许多网友在下载 Turbo C 2.0 和 Turbo C++ 3.0 后，向我问得最多的是在使用过程中碰到如下问题：

（1）出现找不到 stdio.h conio.h 等 include 文件；

（2）出现 cos.obj 无法连接之类的错误。

这些问题是由于没有设置好路径引起的。目前下载的 TC2，TC3 按安装分类大概有两种版本：一是通过 install 安装，这类已经设置好了路径；二是直接解压后建立 TC.EXE 的快捷方式，在 Windows 下双击即可运行（DOS 下直接运行 TC.EXE），因此下载使用前请注意路径设置。

设置方法为：

```
OPTION->DIRECTORIES:
INCLUDE: [TC2/3 所在目录]/include
LIB: [TC2/3 所在目录]/lib
```

output 输出目录请自己设置一个工作目录，以免混在一起。

最后还提醒一点：FILES 中的 Change dir(改变当前目录）中应设置为当前程序所在目录。

A.1　C 语言的起源

C 语言是 1972 年由美国的 Dennis Ritchie 设计发明的，并首次在 UNIX 操作系统的 DEC PDP-11 计算机上使用。它由早期的编程语言 BCPL(Basic Combind Programming Language) 发展演变而来。在 1970 年，AT&T 贝尔实验室的 Ken Thompson 根据 BCPL 语言设计出较先进的并取名为 B 的语言，最后导致了 C 语言的问世。随着微型计算机的日益普及，出现了许多 C 语言版本。由于没有统一的标准，使得这些 C 语言之间出现了一些不一致的地方。为了改变这种情况，美国国家标准研究所(ANSI)为 C 语言制定了一套 ANSI 标准，成为现行的 C 语言标准。

A.2　C 语言的特点

C 语言发展如此迅速，而且成为最受欢迎的语言之一，主要因为它具有强大的功能。许多著名的系统软件，如 DBase Ⅲ Plus、Dbase Ⅳ 都是由 C 语言编写的。用 C 语言加上一些汇编语言子程序，就更能显示 C 语言的优势了，像 PC- DOS、WordStar 等就是用这种方法编写的。归纳起来 C 语言具有下列特点：

（1）C 是中级语言

它把高级语言的基本结构和语句与低级语言的实用性结合起来。C 语言可以像汇编语言一样对位、字节和地址进行操作，而这三者是计算机最基本的工作单元。

（2）C 是结构式语言

结构式语言的显著特点是代码及数据的分隔化，即程序的各个部分除了必要的信息交流外彼此独立。这种结构化方式可使程序层次清晰，便于使用、维护以及调试。C 语言是以函数形式提供给用户的，这些函数可方便地调用，并具有多种循环、条件语句控制程序流向，从而使程序完全结构化。

（3）C 语言功能齐全

C 语言具有各种各样的数据类型，并引入了指针概念，可使程序效率更高。另外，C 语言也具有强大的图形功能，支持多种显示器和驱动器。而且计算功能、逻辑判断功能也比较强大，可以实现决策目的。

（4）C 语言适用范围大

C 语言还有一个突出的优点就是适合于多种操作系统，如 DOS、UNIX,也适用于多种机型。

A.3　Turbo C 概述

A.3.1　Turbo C 的产生与发展

Turbo C 是美国 Borland 公司的产品,Borland 公司是一家专门从事软件开发、研制的大公司。该公司相继推出了一套 Turbo 系列软件，如 Turbo BASIC, Turbo Pascal, Turbo Prolog, 这些软件很受用户欢迎。该公司在 1987 年首次推出 Turbo C 1.0 产品，其中使用了全然一新的集成开发环境，即使用了一系列下拉式菜单,将文本编辑、程序编译、连接以及程序运行一体化，大大方便了程序的开发。1988 年, Borland 公司又推出 Turbo C 1.5 版本，增加了图形库和文本窗口函数库等，而 Turbo C 2.0 则是该公司 1989 年出版的。Turbo C 2.0 在原来集成开发环境的基础上增加了查错功能，并可以在 Tiny 模式下直接生成.COM (数据、代码、堆栈处在同一 64K 内存中) 文件。还可对数字协处理器 (支持 8087/80287/80387 等)进行仿真。Borland 公司后来又推出了面向对象的程序软件包 Turbo C++，它继承发展 Turbo C 2.0 的集成开发环境，并包含了面向对象的基本思想和设计方法。

1991 年为了适用 Microsoft 公司的 Windows 3.0 版本, Borland 公司又将 Turbo C++ 做了更新，即 Turbo C 的新一代产品 Borland C++也已经问世了。

A.3.2　Turbo C 2.0 基本配置要求

Turbo C 2.0 可运行于 IBM-PC 系列微机，包括 XT, AT 及 IBM 兼容机。此时要求 DOS 2.0 或更高版本支持，并至少需要 448K 的 RAM，可在任何彩、单色 80 列监视器上运行。支持数字协处理器芯片，也可进行浮点仿真，这将加快程序的执行。

A.3.3　Turbo C 2.0 内容简介

Turbo C 2.0 有 6 张低密软盘(或两张高密软盘)。Turbo C 2.0 的主要文件简单介绍如下：

INSTALL.EXE 安装程序文件

TC.EXE 集成编译

TCINST.EXE 集成开发环境的配置设置程序

- TCHELP.TCH 帮助文件
- THELP.COM 读取 TCHELP.TCH 的驻留程序
- README 关于 Turbo C 的信息文件
- TCCONFIG.EXE 配置文件转换程序
- MAKE.EXE 项目管理工具
- TCC.EXE 命令行编译
- TLINK.EXE Turbo C 系列连接器
- TLIB.EXE Turbo C 系列库管理工具
- C0?.OBJ 不同模式启动代码
- C?.LIB 不同模式运行库
- GRAPHICS.LIB 图形库
- EMU.LIB 8087 仿真库
- FP87.LIB 8087 库
- *.H Turbo C 头文件
- *.BGI 不同显示器图形驱动程序
- *.C Turbo C 例行程序(源文件)

其中，上面的“?”分别为:

- T Tiny(微型模式)
- S Small(小模式)
- C Compact(紧凑模式)
- M Medium(中型模式)
- L Large(大模式)
- H Huge(巨大模式)

A.3.4 Turbo C 2.0 的安装和启动

Turbo C 2.0 的安装非常简单，只要将 1#盘插入 A 驱动器中，在 DOS 的“A>”下键入:

```
A>INSTALL
```

即可，此时屏幕上显示三种选择:

（1）在硬盘上创造一个新目录来安装整个 Turbo C 2.0 系统。

（2）对 Turbo C 1.5 更新版本。

这样的安装将保留原来对选择项、颜色和编辑功能键的设置。

（3）为只有两个软盘而无硬盘的系统安装 Turbo C 2.0。

这里假定按第一种选择进行安装。只要在安装过程中按对盘号的提示，顺序插入各个软盘，就可以顺利地进行安装。安装完毕将在 C 盘根目录下建立一个 TC 子目录，TC 下还建立了两个子目录 LIB 和 INCLUDE，LIB 子目录中存放库文件，INCLUDE 子目录中存放所有头文件。

运行 Turbo C 2.0 时，只要在 TC 子目录下键入 TC 并回车，即可进入 Turbo C 2. 0 集成开发环境。

A.3.5　Turbo C 2.0 集成开发环境的使用

进入 Turbo C 2.0 集成开发环境中后，屏幕上显示：

```
File Edit Run Compile Project Options Debug Break/watch
┌──────────────────────────Edit──────────────────────────┐
│ Line 1 Col 1 Insert Indent Tab File Unindent c:NONAME.C │
│                                                         │
│                                                         │
│                                                         │
│                                                         │
│                                                         │
│                                                         │
│                                                         │
│                                                         │
│──────────────────────Message────────────────────────────│
│                                                         │
│                                                         │
└─────────────────────────────────────────────────────────┘
F1-Help  F5-Zoom F6-Switch F7-Trace F8-Step F9-Make F10-Menu
```

其中顶上一行为 Turbo C 2.0 主菜单，中间窗口为编辑区，接下来是信息窗口，最底下一行为参考行。这四个窗口构成了 Turbo C 2.0 的主屏幕，以后的编程、编译、调试以及运行都将在这个主屏幕中进行。下面详细介绍主菜单的内容。

1. 主菜单

主菜单 在 Turbo C 2.0 主屏幕顶上一行，显示下列内容：

```
File Edit Run Compile Project Options Debug Break/watch
```

除 Edit 外，其他各项均有子菜单，只要用 Alt 加上某项中第一个字母(大写字母)，就可进入该项的子菜单中。

（1）File(文件)菜单

按 Alt+F 可进入 File 菜单，该菜单包括以下内容：

➢　Load(加载)

装入一个文件，可用类似 DOS 的通配符(如*.C)来进行列表选择。也可装入其他扩展名的文件，只要给出文件名(或只给路径)即可。该项的热键为 F3，即只要在主菜单中按 F3 即可进入该项，而不需要先进入 File 菜单再选此项。

➢　Pick(选择)

将最近装入编辑窗口的 8 个文件列成一个表让用户选择，选择后将该程序装入编辑区，并将光标置在上次修改过的地方。其热键为 Alt+F3。

➢　New(新文件)

说明文件是新的，默认文件名为 NONAME.C，存盘时可改名。

➢ Save(存盘)

将编辑区中的文件存盘，若文件名是 NONAME.C 时，将询问是否更改文件名，其热键为 F2。

➢ Write to(存盘)

可由用户给出文件名将编辑区中的文件存盘，若该文件已存在，则询问要不要覆盖。

➢ Directory(目录)

显示目录及目录中的文件，并可由用户选择。

➢ Change dir(改变目录)

显示当前目录，用户可以改变显示的目录。

➢ Os shell(暂时退出)

暂时退出 Turbo C 2.0 到 DOS 提示符下，此时可以运行 DOS 命令，若想回到 Turbo C 2.0 中，只要在 DOS 状态下键入 EXIT 即可。

➢ Quit(退出)

退出 Turbo C 2.0，返回到 DOS 操作系统中，其热键为 Alt+X。

说明:

以上各项可用光标键移动色棒进行选择，回车则执行。也可用每一项的第一个大写字母直接选择。若要退到主菜单或从它的下一级菜单列表框退回均可用 Esc 键，Turbo C 2.0 所有菜单均采用这种方法进行操作，以下不再说明。

（2）Edit(编辑)菜单

按 Alt+E 可进入编辑菜单，若再回车，则光标出现在编辑窗口，此时用户可以进行文本编辑。

编辑方法基本与 WordStar 相同，可用 F1 键获得有关编辑方法的帮助信息。

与编辑有关的功能键如下:

➢ F1 获得 Turbo C 2.0 编辑命令的帮助信息
➢ F5 扩大编辑窗口到整个屏幕
➢ F6 在编辑窗口与信息窗口之间进行切换
➢ F10 从编辑窗口转到主菜单

编辑命令简介:

➢ PageUp 向前翻页
➢ PageDn 向后翻页
➢ Home 将光标移到所在行的开始
➢ End 将光标移到所在行的结尾
➢ Ctrl+Y 删除光标所在的一行
➢ Ctrl+T 删除光标所在处的一个词
➢ Ctrl+KB 设置块开始
➢ Ctrl+KK 设置块结尾
➢ Ctrl+KV 块移动
➢ Ctrl+KC 块拷贝
➢ Ctrl+KY 块删除

- Ctrl+KR 读文件
- Ctrl+KW 存文件
- Ctrl+KP 块文件打印
- Ctrl+F1 如果光标所在处为 Turbo C 2.0 库函数，则获得有关该函数的帮助信息

Ctrl+Q[查找 Turbo C 2.0 双界符的后匹配符

Ctrl+Q] 查找 Turbo C 2.0 双界符的前匹配符

说明:

a Turbo C 2.0 的双界符包括以下几种符号:

- 花括符 {和}
- 尖括符 <和>
- 圆括符 (和)
- 方括符 [和]
- 注释符 /*和*/
- 双引号 "
- 单引号 '

b Turbo C 2.0 在编辑文件时还有一种功能，就是能够自动缩进，即光标定位和上一个非空字符对齐。在编辑窗口中，Ctrl+OL 为自动缩进开关的控制键。

（3）Run(运行)菜单

按 Alt+R 可进入 Run 菜单，该菜单有以下各项:

- Run(运行程序)

运行由 Project/Project name 项指定的文件名或当前编辑区的文件。如果对上次编译后的源代码未做过修改，则直接运行到下一个断点(没有断点则运行到结束)。

否则先进行编译、连接后才运行，其热键为 Ctrl+F9。

- Program reset(程序重启)

中止当前的调试，释放分给程序的空间，其热键为 Ctrl+F2。

- Go to cursor(运行到光标处)

调试程序时使用，选择该项可使程序运行到光标所在行。光标所在行必须为一条可执行语句，否则提示错误。其热键为 F4。

- Trace into(跟踪进入)

在执行一条调用其他用户定义的子函数时，若用 Trace into 项，则执行长条将跟踪到该子函数内部去执行，其热键为 F7。

- Step over(单步执行)

执行当前函数的下一条语句，即使用户函数调用，执行长条也不会跟踪进函数内部，其热键为 F8。

- User screen(用户屏幕)

显示程序运行时在屏幕上显示的结果。其热键为 Alt+F5。

（4）Compile(编译)菜单

按 Alt+C 可进入 Compile 菜单，该菜单有以下几个内容:

- Compile to OBJ(编译生成目标码)

将一个 C 源文件编译生成.OBJ 目标文件，同时显示生成的文件名。其热键为

Alt+F9。

➢ Make EXE file(生成执行文件)

此命令生成一个.EXE 的文件，并显示生成的.EXE 文件名。其中，EXE 文件名是下面几项之一。

a. 由 Project/Project name 说明的项目文件名。

b. 若没有项目文件名，则由 Primary C file 说明源文件。

c. 若以上两项都没有文件名，则为当前窗口的文件名。

➢ Link EXE file(连接生成执行文件)

把当前.OBJ 文件及库文件连接在一起生成.EXE 文件。

➢ Build all(建立所有文件)

重新编译项目里的所有文件，并进行装配生成.EXE 文件。该命令不做过时检查（上面的几条命令要做过时检查，即如果目前项目里源文件的日期和时间与目标文件相同或更早，则拒绝对源文件进行编译)。

➢ Primary C file(主 C 文件)

当在该项中指定了主文件后，在以后的编译中，如没有项目文件名则编译此项中规定的主 C 文件，如果编译中有错误，则将此文件调入编辑窗口，不管目前窗口中是不是主 C 文件。

➢ Get info(获得有关当前路径、源文件名、源文件字节大小、编译中的错误数目、可用空间等信息。

（5）Project(项目)菜单

按 Alt+P 可进入 Project 菜单，该菜单包括以下内容:

➢ Project name(项目名)

项目名具有.PRJ 的扩展名，其中包括将要编译、连接的文件名。例如有一个程序由 file1.c, file2.c, file3.c 组成，要将这 3 个文件编译装配成一个 file.exe 的执行文件，可以先建立一个 file.prj 的项目文件，其内容如下:

file1.c

file2.c

file3.c

此时将 file.prj 放入 Project name 项中，以后进行编译时，将自动对项目文件中规定的三个源文件分别进行编译。然后连接成 file.exe 文件。

如果其中有些文件已经编译成.OBJ 文件，而又没有修改过，可直接写上.OBJ 扩展名。此时将不再编译而只进行连接。

例如: file1.obj

file2.c

file3.c

将不对 file1.c 进行编译，而直接连接。

说明:

当项目文件中的每个文件无扩展名时，均按源文件对待，另外，其中的文件也可以是库文件，但必须写上扩展名.LIB。

➢ Break make on(中止编译)

由用户选择是否在有 Warning(警告)、Errors(错误)、Fatal Errors(致命错误)时或 Link(连

接)之前退出 Make 编译。

➢ Auto dependencies(自动依赖)

当开关置为 on，编译时将检查源文件与对应的.OBJ 文件日期和时间，否则不进行检查。

➢ Clear project(清除项目文件)

清除 Project/Project name 中的项目文件名。

➢ Remove messages(删除信息)

把错误信息从信息窗口中清除掉。

（6）Options(选择菜单)

按 Alt+O 可进入 Options 菜单，该菜单对初学者来说要谨慎使用。

➢ Compiler(编译器)

本项选择又有许多子菜单，可以让用户选择硬件配置、存储模型、调试技术、代码优化、对话信息控制和宏定义。这些子菜单如下：

Model 共有 Tiny, small, medium, compact, large, huge 六种不同模式可由用户选择。

Define 打开一个宏定义框，用户可输入宏定义。多重定义可用分号，赋值可用等号。

Code generation 有许多任选项，这些任选项告诉编译器产生什么样的目标代码。

Calling convention 可选择 C 或 Pascal 方式传递参数。

Instruction set 可选择 8088/8086 或 80186/80286 指令系列。

Floating point 可选择仿真浮点、数字协处理器浮点或无浮点运算。

Default char type 规定 char 的类型。

Alignonent 规定地址对准原则。

Merge duplicate strings 做优化用，将重复的字符串合并在一起。

Standard stack frame 产生一个标准的栈结构。

Test stack overflow 产生一段程序运行时检测堆栈溢出的代码。

Line number 在.OBJ 文件中放进行号以供调试时用。

OBJ debug information 在.OBJ 文件中产生调试信息。

Optimize for 选择是对程序小型化还是对程序速度进行优化处理。

Use register variable 用来选择是否允许使用寄存器变量。

Register optimization 尽可能使用寄存器变量以减少过多的取数操作。

Jump optimization 通过去除多余的跳转和调整循环与开关语句的办法，压缩代码。

Source

Identifier length 说明标识符有效字符的个数，默认为 32 个。

Nested comments 是否允许嵌套注释。

ANSI keywords only 是只允许 ANSI 关键字，还是也允许 Turbo C 2.0 关键字。

Error

Error stop after 多少个错误时停止编译，默认为 25 个。

Warning stop after 多少个警告错误时停止编译，默认为 100 个。

Display warning

Portability warning 移植性警告错误。

ANSI Violations 侵犯了 ANSI 关键字的警告错误。

Common error 常见的警告错误。

Less common error 少见的警告错误。

Names 用于改变段(segment)、组(group) 和类(class)的名字，默认值为 CODE,DATA,BSS。

➢ Linker(连接器)

本菜单设置有关连接的选择项，它有以下内容:

Map file menu 选择是否产生.MAP 文件。

Initialize segments 是否在连接时初始化没有初始化的段。

Default libraries 是否在连接其他编译程序产生的目标文件时去寻找其默认库。

Graphics library 是否连接 graphics 库中的函数。

Warn duplicate symbols 当有重复符号时产生警告信息。

Stack warning 是否让连接程序产生 No stack 的警告信息。

Case-sensitive link 是否区分大、小写字。

➢ Environment(环境)

本菜单规定是否对某些文件自动存盘及制表键和屏幕大小的设置 Message tracking。

Current file 跟踪在编辑窗口中的文件错误。

All files 跟踪所有文件错误。

Off 不跟踪。

Keep message 编译前是否清除 Message 窗口中的信息。

Config auto save 选 on 时，在 Run, Shell 或退出集成开发环境之前,如果 Turbo C 2.0 的配置被改过，则所做的改动将存入配置文件中。选 off 时不存。

Edit auto save 是否在 Run 或 Shell 之前，自动存储编辑的源文件。

Backup file 是否在源文件存盘时产生后备文件(.BAK 文件)。

Tab size 设置制表键大小，默认为 8。

Zoomed windows 将现行活动窗口放大到整个屏幕，其热键为 F5。

Screen size 设置屏幕文本大小。

➢ Directories(路径)

规定编译、连接所需文件的路径，有下列各项:

Include directories 包含文件的路径，多个子目录用“;”分开。

Library directories 库文件路径，多个子目录用“;”分开。

Output directories 输出文件(.OBJ, .EXE, .MAP 文件)的目录。

Turbo C directories Turbo C 所在的目录。

Pick file name 定义加载的 pick 文件名，如不定义则从 currentpick file 中取。

➢ Arguments(命令行参数)

允许用户使用命令行参数。

➢ Save options(存储配置)

保存所有选择的编译、连接、调试和项目到配置文件中，默认的配置文件为 TCCONFIG.TC。

➢ Retrive options

装入一个配置文件到 TC 中，TC 将使用该文件的选择项。

（7）Debug(调试)菜单

按 Alt+D 可选择 Debug 菜单，该菜单主要用于查错，它包括以下内容:

Evaluate

Expression 要计算结果的表达式。

Result 显示表达式的计算结果。

New value 赋给新值。

Call stack 该项不可接触。而在 Turbo C debuger 时用于检查堆栈情况。

Find function 在运行 Turbo C debugger 时用于显示规定的函数。

Refresh display 如果编辑窗口偶然被用户窗口重写了，可用此恢复编辑窗口的内容。

(8) Break/watch(断点及监视表达式)

按 Alt+B 可进入 Break/watch 菜单，该菜单有以下内容:

Add watch 向监视窗口插入一监视表达式。

Delete watch 从监视窗口中删除当前的监视表达式。

Edit watch 在监视窗口中编辑一个监视表达式。

Remove all watches 从监视窗口中删除所有的监视表达式。

Toggle breakpoint 对光标所在的行设置或清除断点。

Clear all breakpoints 清除所有断点。

View next breakpoint 将光标移动到下一个断点处。

A.3.6　Turbo C 2.0 的配置文件

所谓配置文件，是包含 Turbo C 2.0 有关信息的文件，其中存有编译、连接的选择和路径等信息。

可以用下述方法建立 Turbo C 2.0 的配置:

（1）建立用户自命名的配置文件

可以从 Options 菜单中选择 Options/Save options 命令，将当前集成开发环境的所有配置存入一个由用户命名的配置文件中。下次启动 TC 时，只要在 DOS 下键入:

tc/c<用户命名的配置文件名>

就会按这个配置文件中的内容作为 Turbo C 2.0 的选择。

（2）若设置 Options/Environment/Config auto save 为 on，则退出集成开发环境时，当前的设置会自动存放到 Turbo C 2.0 配置文件 TCCONFIG.TC 中。Turbo C 在启动时会自动寻找这个配置文件。

（3）用 TCINST 设置 Turbo C 的有关配置，并将结果存入 TC.EXE 中。Turbo C 在启动时，若没有找到配置文件，则取 TC.EXE 中的缺省值。

附录 B　ASCII 码表

ASCII 码（**American Standard Code for Information Interchange**，美国标准信息交换码），是目前计算机中用得最广泛的字符集及其编码。它是由美国国家标准局(ANSI)制定的，现已被国际标准化组织（ISO）定为国际标准，称为 ISO 646 标准。适用于所有拉丁文字字母，ASCII 码有 7 位码和 8 位码两种形式。

因为 1 位二进制数可以表示（2^1=）2 种状态：0、1；而 2 位二进制数可以表示（2^2=）4 种状态：00、01、10、11；以此类推，7 位二进制数可以表示（2^7=）128 种状态，每种状态都唯一地编为一个 7 位的二进制码，对应一个字符（或控制码），这些码可以排列成一个十进制序号 0～127。所以，7 位 ASCII 码是用七位二进制数进行编码的，可以表示 128 个字符。

第 **0～32** 号及第 **127** 号(共 **34** 个)是控制字符或通讯专用字符，如控制符：LF（换行）、CR（回车）、FF（换页）、DEL（删除）、BEL（振铃）等；通讯专用字符：SOH（文头）、EOT（文尾）、ACK（确认）等。

第 **33～126** 号(共 94 个)是字符，其中，第 **48～57** 号为 0～9 十个阿拉伯数字；**65～90** 号为 26 个大写英文字母，**97～122** 号为 26 个小写英文字母，其余为一些标点符号、运算符号等。

注意：在计算机的存储单元中，一个 ASCII 码值占一个字节(8 个二进制位)，其最高位(b7)用做奇偶校验位。所谓奇偶校验，是指在代码传送过程中用来检验是否出现错误的一种方法，一般分奇校验和偶校验两种。奇校验规定：正确的代码一个字节中 1 的个数必须是奇数，若非奇数，则在最高位 b7 添 1；偶校验规定：正确的代码一个字节中 1 的个数必须是偶数，若非偶数，则在最高位 b7 添 1。

ASCII 码在编程中经常要遇到，对于串行通信，我们在编程中也要注意设置程序要传送的是 ASCII 文本还是二进制码，如在 MSComm 控件中，就可以设置为两种方式中的一种。对于中文，在计算机内码中，有这样一个规律：一个中文字符由两个 ASCII 码组成，且码值均为大于 127（DEC）。

在编程过程中，特别是在接收或发送数据的处理过程中，要记住这样一个原则，不管其值是大于 127 还是 127 以下，发送的值为多大，接收时按照同样的数据类型来处理时，其值是相同的。当不需要显示时，我们就只要把传送的值抽取出来，就完成了数据处理。如果要显示，只有常规字符可以显示出来，如果要知道其他字符，则只有显示其十六进制值了（当然，常规的字符同样也有十六进制值），这就是为什么串口调试助手中要将接收字符值以十六进制值显示的原因。

为了便于查询，以下列出 ASCII 码表：

ASCII 表

DEC	HEX	CHAR	CODE	C 程序		DEC	HEX	CHAR	CODE	C 程序
十进制	十六进制	字符	控制码	转义		十进制	十六进制	字符	控制码	转义
0	00		NUL	('\0')		32	20	(space，空格)		
1	01		SOH			33	21	!		
2	02		STX			34	22	"		
3	03		ETX			35	23	#		
4	04		EOT			36	24	$		
5	05		ENQ			37	25	%		
6	06		ACK			38	26	&		
7	07		BEL	('\a')		39	27	'		
8	08		BS	('\b')		40	28	(		
9	09		HT	('\t')		41	29	)		
10	0A		LF	('\n')		42	2A	*		
11	0B		VT	('\v')		43	2B	+		
12	0C		FF	('\f')		44	2C	,		
13	0D		CR	('\r')		45	2D	-		
14	0E		SO			46	2E	.		
15	0F		SI			47	2F	/		
16	10		DLE			48	30	0		
17	11		DC1			49	31	1		
18	12		DC2			50	32	2		
19	13		DC1			51	33	3		
20	14		DC4			52	34	4		
21	15		NAK			53	35	5		
22	16		SYN			54	36	6		
23	17		ETB			55	37	7		
24	18		CAN			56	38	8		
25	19		EM			57	39	9		
26	1A		SUB			58	3A	:		
27	1B		ESC			59	3B	;		
28	1C		FS			60	3C	<		
29	1D		GS			61	3D	=		
30	1E		RS			62	3E	>		
31	1F		US			63	3F	?		

续表

DEC	HEX	CHAR	CODE	C		DEC	HEX	CHAR	CODE	C
十进制	十六进制	字符	控制码	程序转义		十进制	十六进制	字符	控制码	程序转义
64	40	@				96	60	`		
65	41	A				97	61	a		
66	42	B				98	62	b		
67	43	C				99	63	c		
68	44	D				100	64	d		
69	45	E				101	65	e		
70	46	F				102	66	f		
71	47	G				103	67	g		
72	48	H				104	68	h		
73	49	I				105	69	i		
74	4A	J				106	6A	j		
75	4B	K				107	6B	k		
76	4C	L				108	6C	l		
77	4D	M				109	6D	m		
78	4E	N				110	6E	n		
79	4F	O				111	6F	o		
80	50	P				112	70	p		
81	51	Q				113	71	q		
82	52	R				114	72	r		
83	53	S				115	73	s		
84	54	T				116	74	t		
85	55	U				117	75	u		
86	56	V				118	76	v		
87	57	W				119	77	w		
88	58	X				120	78	x		
89	59	Y				121	79	y		
90	5A	Z				122	7A	z		
91	5B	[				123	7B	{		
92	5C	\		(W)		124	7C	|		
93	5D	]				125	7D	}		
94	5E	^				126	7E	~		
95	5F	_				127	7F		DEL	

注：ASCII 码中，127 为 DEL，在 MS-DOS 下，该码与 ASCII 8（BS）相同，DEL 码可以由 Delete 键或 CTRL+BKSP 产生。