Павел Агуров

bhv

Последовательные интерфейсы ПК

Практика программирования

+ CD-ROM

- Протоколы последовательной связи
- Стандарты RS232/RS485
- Работа в DOS, Windows 98/ME/NT/2000/XP
- Примеры на языке Pascal
- Все уровни доступа
- Поддержка Plug and Play

Павел Агуров

Последовательные интерфейсы ПК

Практика программирования

Санкт-Петербург
«БХВ-Петербург»
2004

УДК 681.3.06
ББК 32.973-018
А23

Агуров П. В.

А23 Последовательные интерфейсы ПК. Практика программирования. — СПб.: БХВ-Петербург, 2004. — 496 с.: ил.

ISBN 5-94157-468-1

Книга представляет собой практическое руководство по программированию последовательных интерфейсов. В первой части книги представлены теоретические сведения о последовательных интерфейсах, во второй — практические примеры и листинги программ на языках Pascal и Delphi, а третья содержит справочные данные, облегчающие поиск необходимой информации. В приложениях приведены дополнительные сведения и ответы на часто задаваемые вопросы. Большое количество практических советов, примеров программ, а также последовательность и простота изложения позволят читателю уверенно овладеть изложенным в книге материалом. Для удобства читателей все исходные коды приводятся на прилагаемом к книге компакт-диске.

Для программистов, занятых в сфере промышленной автоматизации

УДК 681.3.06
ББК 32.973-018

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Алексей Семенов*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Оформление серии	*Via Design*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 13.02.04.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 39,99.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02 от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-468-1

© Агуров П. В., 2004
© Оформление, издательство "БХВ-Петербург", 2004

Содержание

Введение **1**

Для кого эта книга 1
Структура книги 2
Краткое описание глав 3
Программные требования 6
О программном коде 6
Обозначения 7
Аппаратные требования 7
Благодарности 7
Обратная связь 7

Часть I. Протоколы и интерфейсы **9**

Глава 1. Стандарты последовательной связи **11**

1.1. Стандарты последовательной связи 11
1.1.1. Протокол RS-232 13
1.1.2. Протокол RS-422A 15
1.1.3. Протокол RS-423A 15
1.1.4. Протокол RS-485 15
1.1.5. Протокол RS-499 16
1.1.6. Протокол RS-562 16
1.1.7. Протокол V.24 17
1.1.8. Протокол V.28 17
1.1.9. Протокол V.35 17
1.1.10. Протокол X.21 18
1.1.11. Рекомендация X.21bis 18
1.1.12. Краткое сравнение RS-протоколов 18
1.2. Принципы последовательной связи 19
1.3. Разъемы коммуникационного порта 20
1.3.1. Сигнальная "земля" (AB/SG) 22
1.3.2. Защитная "земля" (AA) 22
1.3.3. Передаваемые данные (BA/TxD/TD) 23
1.3.4. Принимаемые данные (BB/RxD/RD) 23
1.3.5. Запрос передачи (CA/RTS) 23

1.3.6. Готовность к передаче (CB/CTS) ... 24
1.3.7. Готовность DCE (CC/DSR) ... 25
1.3.8. Готовность DTE (CD/DTR) ... 25
1.3.9. Индикатор вызова (CE/RI) ... 25
1.3.10. Обнаружение несущей (CF/DCD) ... 25
1.3.11. Детектор качества сигнала (CG/SQ) ... 26
1.3.12. Переключатель скорости передачи данных от DTE (CH) ... 26
1.3.13. Переключатель скорости передачи данных от DCE (CI) ... 26
1.3.14. Готовность к приему (CJ) ... 26
1.3.15. Местный шлейф (LL) ... 27
1.3.16. Удаленный шлейф (RL) ... 27
1.3.17. Индикатор тестирования (TM) ... 27
1.3.18. Синхронизация передачи от DTE (DA) ... 28
1.3.19. Синхронизация передачи от DCE (DB/TC) ... 28
1.3.20. Синхронизация приема от DCE (DD/RC) ... 28
1.3.21. Передаваемые данные дополнительного канала (SBA/STD) ... 28
1.3.22. Принимаемые данные дополнительного канала (SBB/SRD) ... 28
1.3.23. Запрос передачи по дополнительному каналу (SCA/SRTS) ... 29
1.3.24. Готовность к передаче по дополнительному каналу (SCB/SCTS) ... 29
1.3.25. Обнаружение несущей дополнительного канала (SCF/SDCD) ... 29
1.4. Ресурсы IBM PC для последовательной связи ... 29
1.4.1. Сервисы BIOS ... 29
1.4.2. Коммуникационные порты ... 30
1.4.3. Использование прерываний ... 30
1.4.4. Прямое программирование портов в Windows ... 30
1.4.5. Функции Windows ... 31

Глава 2. Протоколы ... 32

2.1. Что такое протокол обмена ... 32
2.2. ASCII-протокол передачи данных ... 33
2.3. Бинарный протокол передачи данных ... 35
2.4. Предотвращение потери данных ... 35
2.4.1. Прерывания и потоки ... 35
2.4.2. Буферизация ... 37
Синхронизация в DOS ... 38
Синхронизация в Windows ... 38
2.4.3. Обратная связь ... 40
2.5. Методы обнаружения ошибок передачи ... 40
2.5.1. Контроль четности ... 41
2.5.2. Контрольная сумма ... 42
Простая контрольная сумма ... 42
Контрольная сумма LRC ... 42
Контрольная сумма CRC16 ... 43

2.5.3. Стартовый байт ... 44
2.5.4. Другие способы повышения достоверности данных ... 45

Глава 3. Последовательные протоколы IBM PC ... 46

3.1. Мышь ... 46
3.1.1. Базовый протокол Microsoft Mouse ... 46
3.1.2. Протокол Microsoft Plus ... 47
3.1.3. Протокол 3D Serial Mouse ... 47
3.1.4. Протокол PC Mouse ... 48
3.1.5. Совместимость протоколов ... 48
3.2. Модем ... 48
3.2.1. Команды управления модемом ... 49
3.2.2. Регистры модемов ... 50
3.2.3. Протоколы передачи файлов ... 50
ASCII-протокол ... 50
Протокол XModem ... 51
Протокол XModem-CRC ... 52
Протокол XModem-1K ... 52
Протокол YModem ... 53
Протокол YModem-G ... 54
Протокол ZModem ... 54
Протокол BiModem ... 55
Протокол Kermit ... 55

Глава 4. Промышленные последовательные протоколы ... 56

4.1. Протокол MODBUS ... 56
4.1.1. Протокол MODBUS-ASCII ... 57
4.1.2. Протокол MODBUS-RTU ... 58
4.1.3. Поля MODBUS протокола ... 59
Поле "Адрес абонента" ... 59
Поле "Код функции" ... 60
Поле "Данные" ... 60
Поле "Контрольная сумма" ... 60
4.2. Протокол CAN ... 61
4.2.1. Характеристики протокола CAN ... 62
4.2.2. Обмен данными в протоколе CAN ... 63
4.2.3. Обнаружение ошибок в протоколе CAN ... 63
4.3. Протокол Profibus ... 64

Глава 5. COM-порты и Plug and Play ... 66

5.1. Кратко о Plug and Play ... 66
5.2. Запуск процедуры PnP ... 67
5.3. Если устройство не находится автоматически ... 68

5.4. Где хранится информация о найденных устройствах ... 69
5.4.1. Структура реестра Windows 98 ... 69
5.4.2. Структура реестра Windows 2000 ... 71
5.5. Алгоритм Plug and Play для COM-портов ... 73
5.5.1. Инициализация порта *(Port initialization)* ... 73
5.5.2. Обнаружение устройств *(Check for device)* ... 73
5.5.3. Установка устройства, фаза 1 *(COM port Setup-1)* ... 74
5.5.4. Ожидание ответа, фаза 1 *(Wait for response-1)* ... 74
5.5.5. Установка устройства, фаза 2 *(COM port Setup-2)* ... 74
5.5.6. Ожидание ответа, фаза 2 *(Wait for response-2)* ... 75
5.5.7. Получение идентификатора *(Collect device ID)* ... 75
5.5.8. Проверка отключения (*Verify Disconnect*) ... 76
5.5.9. Дежурное состояние *(Connect Idle)* ... 76
5.5.10. Состояние ожидания отключения *(Disconnect Idle)* ... 76
5.6. Формат данных для передачи PnP-идентификатора ... 77
5.7. Передача PnP-идентификатора ... 77
5.8. Поля PnP-идентификатора ... 78
5.8.1. Поле *Other ID* ... 80
5.8.2. Поле *Begin PnP* ... 80
5.8.3. Поле *PnP Rev* ... 80
5.8.4. Поле *EISA ID* ... 80
5.8.5. Поле *Product ID* ... 81
5.8.6. Поле *Extend* ... 81
5.8.7. Поле *Serial Number* ... 81
5.8.8. Поле *Class Name* ... 81
5.8.9. Поле *Device ID* ... 82
5.8.10. Поле *User Name* ... 82
5.8.11. Поле *Checksum* ... 82
5.8.12. Поле *End PnP* ... 82
5.8.13. Пример PnP-идентификатора ... 83
5.9. INF-файл и его структура ... 84
5.9.1. Секция *Version* ... 84
5.9.2. Секция *Manufacturer* ... 85
5.9.3. Секция *DestinationDirs* ... 88
Ключ *DefaultDescDir* ... 88
Ключи *file-list-section* ... 88
Ключ *dirid* ... 88
Ключ *subdir* ... 89
5.9.4. Секция описания модели ... 90
5.9.5. Секции *xxx.AddReg* и *xxx.DelReg* ... 91
5.9.6. Секция *xxx.LogConfig* ... 92
5.9.7. Секция *xxx.CopyFiles* ... 93
5.9.8. Секция *Strings* ... 94
5.9.9. Связи секций ... 95

Часть II. Практика программирования 97

Глава 6. Использование сервисов BIOS 99

6.1. Подготовка 99
6.2. Первая программа последовательной связи 100
6.3. Другие функции BIOS 102

Глава 7. Прямое программирование портов 104

7.1. Коммуникационные порты 104
7.2. Программа определения наличия COM-портов 105
7.3. Обходим ограничения BIOS 106
7.4. Чтение и запись с помощью модуля *RS232DOS* 111

Глава 8. Использование обработки прерываний 113

8.1. Прерывания 113
8.2. Модуль *RS232Int* для работы с прерываниями 115
8.3. Последовательный обмен с помощью прерываний 124

Глава 9. Переход в Windows 127

9.1. Переходим из DOS в Windows 127
9.2. Первая программа для Windows 131
9.3. Перенос программ из Windows 9*x* в Windows NT/2000 142
9.3.1. Получение доступа к портам в Windows 2000/XP 143
9.3.2. Расширение возможностей GiveIO 148
9.3.3. Работа с драйвером GiveIOEx 157
9.3.4. Еще немного о прямом доступе к портам 165

Глава 10. Использование функций Windows 166

10.1. Обзор функций Windows для работы с последовательными портами 166
10.2. Специальная настройка порта 167
10.3. Получение состояния линий модема 168
10.4. Используем функции Windows 168
10.5. Несколько замечаний о контроле четности 183

Глава 11. Использование потоков 185

11.1. Преимущества потоков 185
11.2. Создание потока опроса порта 185

Глава 12. Функции асинхронного доступа и события 194

12.1. Асинхронный доступ 194
12.2. События последовательного порта 197
12.3. Использование событий 197

Глава 13. Специальные коммуникационные функции 201

13.1. Выполнение дополнительных операций 201
13.2. Прямое управление драйвером 203
13.3. Обнаружение всех портов системы 222
13.4. Имена портов 225
13.5. APC-функции Windows 2000/XP 231

Глава 14. Реализация протоколов обмена 234

14.1. Прием и передача простого пакета 234
14.2. Создание компонента 241
14.3. Реализация ASCII-протокола 254
14.4. Реализация бинарного протокола 262

Глава 15. Вычисление контрольных сумм 273

15.1. Вычисление простой контрольной суммы 273
15.2. Вычисление LCR 274
15.3. Вычисление CRC16 275
15.4. Вычисление CRC32 279

Глава 16. Интерфейс RS-485 282

16.1. Стандарт RS-485 282
16.2. Как работает COM-порт IBM PC 283
16.3. Реализация RS-485 288

Глава 17. Работа Plug and Play 291

17.1. Эмулятор устройства 291
17.2. Установка драйвера устройства 300
17.3. Обнаружение изменений 303

Часть III. Справочник 309

Глава 18. Сервис BIOS INT14H/INT21H 311

18.1. Сервис BIOS INT14H 311
18.1.1. Функция 00: инициализация порта 312
18.1.2. Функция 01: посылка символа в порт 313
18.1.3. Функция 02: получение символа 313
18.1.4. Функция 03: получить статус порта 313
18.1.5. Функция 04: расширенная инициализация (System/2) 314
18.1.6. Функция 05: расширенное управление портом (System/2) 315
18.2. Сервис BIOS INT21H 316
18.2.1. Функция 03H: вспомогательный ввод 317
18.2.2. Функция 04H: вспомогательный вывод 317

18.2.3. Функция 3FH: чтение данных через описатель 317
18.2.4. Функция 40H: запись данных через описатель 318

Глава 19. Порты IBM PC 319

19.1. Порт 3F8H (BasePort+0): данные и делитель 320
19.2. Порт 3F9H (BasePort+1): разрешение прерываний и делитель 320
19.3. Порт 3FAH (BasePort+2): идентификация прерываний 321
19.4. Порт 3FBH (BasePort+3): управление линией 322
19.5. Порт 3FCH (BasePort+4): управление модемом 323
19.6. Порт 3FDH (BasePort+5): статус линии 324
19.7. Порт 3FEH (BasePort+6): статус модема 325
19.8. Порт 3FFH (BasePort+7): заглушка 325
19.9. Скорость передачи 325
19.10. Порты контроллера прерываний 326

Глава 20. Структуры Windows для работы с COM-портами 328

20.1. Структура настроек порта *COMMCONFIG* 328
20.2. Структура *COMMPROP* 330
20.3. Структура тайм-аутов *COMMTIMEOUTS* 336
20.4. Структура статуса порта *COMSTAT* 339
20.5. Структура *DCB* 340

Глава 21. Функции Windows для работы с COM-портами 347

21.1. Функции *CreateFile* и *CloseHandle*: открытие и закрытие порта 347
21.1.1. Дополнительные сведения 349
21.1.2. Возвращаемое значение 349
21.1.3. Пример вызова 349
21.2. Функция *ReadFile*: чтение данных из порта 350
21.2.1. Дополнительные сведения 351
21.2.2. Возвращаемое значение 351
21.2.3. Пример вызова 351
21.3. Функция *WriteFile*: передача данных 352
21.3.1. Дополнительные сведения 353
21.3.2. Возвращаемое значение 353
21.3.3. Пример вызова 353
21.4. Функция *ReadFileEx*: APC-чтение данных 354
21.4.1. Возвращаемое значение 356
21.4.2. Дополнительные сведения 356
21.4.3. Пример вызова 356
21.5. Функция *WriteFileEx*: APC-передача данных 357
21.5.1. Возвращаемое значение 358
21.5.2. Пример вызова 358

21.6. Функция *BuildCommDCB*: создание структуры *DCB* из строки 359
21.6.1. Дополнительные сведения 360
21.6.2. Возвращаемое значение 360
21.6.3. Пример вызова 360
21.7. Функция *BuildCommDCBAndTimeouts*: создание структуры *DCB* и тайм-аутов из строки 361
21.8. Функции *SetCommBreak* и *ClearCommBreak*: управление выводом данных 362
21.8.1. Возвращаемое значение 362
21.9. Функция *ClearCommError*: получение и сброс ошибок порта 363
21.9.1. Возвращаемое значение 364
21.10. Функция *CancelIo*: прерывание асинхронных операций 364
21.10.1. Возвращаемое значение 364
21.11. Функция *EscapeCommFunction*: управление портом 365
21.11.1. Возвращаемое значение 366
21.12. Функции *GetCommMask* и *SetCommMask*: маска вызова событий 366
21.12.1. Возвращаемое значение 367
21.12.2. Пример вызова 367
21.13. Функция *CreateEvent*: создание объекта ожидания 367
21.13.1. Возвращаемое значение 368
21.13.2. Пример вызова 368
21.14. Функция *WaitCommEvent*: ожидание события COM-порта 368
21.14.1. Возвращаемое значение 369
21.14.2. Дополнительные сведения 369
21.14.3. Пример вызова 369
21.15. Функции *GetCommConfig* и *SetCommConfig*: конфигурирование параметров порта 372
21.15.1. Возвращаемое значение 372
21.15.2. Пример вызова 373
21.16. Функция *CommConfigDialog*: диалог конфигурирования порта 374
21.16.1. Возвращаемое значение 375
21.16.2. Дополнительные сведения 375
21.16.3. Пример вызова 375
21.17. Функция *GetCommProperties*: прочитать свойства порта 375
21.17.1. Возвращаемое значение 376
21.17.2. Пример вызова 376
21.18. Функции *GetCommState* и *SetCommState*: состояние порта 376
21.18.1. Возвращаемое значение 377
21.18.2. Пример вызова 377
21.19. Функции *GetCommTimeouts* и *SetCommTimeouts*: тайм-ауты порта 377
21.19.1. Возвращаемое значение 378
21.19.2. Пример вызова 378

21.20. Функция *PurgeComm*: сброс буферов порта 378
21.20.1. Возвращаемое значение 379
21.20.2. Пример вызова 379
21.21. Функция *SetupComm*: конфигурирование размеров буферов 380
21.21.1. Возвращаемое значение 381
21.22. Функции *GetDefaultCommConfig* и *SetDefaultCommConfig*: настройки порта по умолчанию 381
21.22.1. Возвращаемое значение 382
21.23. Функция *TransmitCommChar*: передача специальных символов 382
21.23.1. Возвращаемое значение 382
21.24. Функция *GetCommModemStatus*: статус модема 383
21.24.1. Возвращаемое значение 383
21.24.2. Пример вызова 383
21.25. Функция *WaitForSingleObject*: ожидание сигнального состояния объекта 384
21.25.1. Возвращаемое значение 385
21.26. Функция *WaitForMultipleObjects*: ожидание сигнального состояния объектов 385
21.26.1. Возвращаемое значение 386
21.27. Функция *GetOverlappedResult*: результат асинхронной операции 387
21.27.1. Возвращаемое значение 388
21.28. Функция *DeviceIoControl*: прямое управление драйвером 388
21.28.1. Возвращаемое значение 390
21.29. Функция *EnumPorts*: перечисление портов 390
21.29.1. Дополнительные сведения 391
21.29.2. Возвращаемое значение 391
21.29.3. Пример вызова 392
21.30. Функция *QueryDosDevice*: получение имени устройства по его DOS-имени 393
21.30.1. Возвращаемое значение 393
21.30.2. Пример вызова 394
21.31. Функция *DefineDosDevice*: операции с DOS-именем устройства 395
21.31.1. Возвращаемое значение 395
21.31.2. Пример вызова 395

Глава 22. Функции *DiviceIoControl* для последовательных портов 397

22.1 Функции драйвера последовательного порта 397
22.2. Структуры драйвера последовательного порта 404
22.3. Соответствие функций *DeviceIoControl* и Windows API 406

Глава 23. AT-команды и регистры модемов 409

23.1. Основные AT-команды модемов 409
23.2. Основные регистры модемов 416

Часть IV. Приложения 419

Приложение 1. Микросхемы UART 421

Общее описание 421
Характерные особенности 421
Описание контактов 422
Типы микросхем UART 426

Приложение 2. Библиотека функций для Pascal 427

Приложение 3. Формат *SINGLE* чисел с плавающей точкой 429

Приложение 4. Описание функций Windows для использования в Visual Basic 430

Приложение 5. Использование ActiveX-компонента *MSComm32* 433

Приложение 6. Сервис BIOS — обслуживание INT14H 436

Приложение 7. Как работает GiveIO 442

Приложение 8. Реализация процедуры преобразования SINGLE-чисел в строку 450

Приложение 9. Индикаторы состояния модема 455

Приложение 10. Инструменты 456

Порт-монитор 456
Разработка драйверов Driver Wizard 457
Разработка драйверов NuMega Driver Studio 457
Отладчик NuMega SoftICE 458
Эмулятор компьютера Connectix Virtual PC 458
Терминал HyperTerminal 458

Приложение 11. Формат команды *Mode* 460

Приложение 12. Часто задаваемые вопросы (FAQ) 462

FAQ: Общие вопросы 462
1. Что такое COM и RS и чем они отличаются? 462
2. Что такое порт? 462
3. Чем "протокол RS-232" отличается от "интерфейса RS-232"? 462
4. Что такое контрольная сумма и как ее считать? 463

5. Откуда взялось ограничение числа абонентов в протоколе RS-485? 463
6. Почему нужно подключать COM-устройства при выключенном питании? 463
7. Откуда берется питание для мыши? 463
8. Что означает режим *HANDSHAKING*? 464
9. Что такое дуплексная и полудуплексная передача? 464
FAQ: Программирование в DOS 464
1. Что такое базовый адрес порта? 464
2. Написал программу обработки прерываний порта, а она периодически "виснет". Для других прерываний все нормально работает 464
3. При выводе на экран данных, полученных в обработчике прерывания, выводится "мусор" 465
4. Прерывание и основная программа используют один буфер. Как их синхронизировать в DOS? 465
5. Как бы перехватить вывод программы в COM-порт под DOS? 465
FAQ: Программирование в Windows 466
1. Почему в Delphi пропал оператор *Port* и как тогда обращаться к портам? 466
2. Можно ли использовать прямое обращение к портам в Windows NT/2000? 466
3. Программа асинхронно работает с портом. В Windows 98 все работало, а в Windows 2000 или "виснет", или не работает 466
4. Зачем надо использовать *WaitForSingleObject*, если уже есть вызов *GetOverlappedResult*? 466
5. Как найти все доступные COM-порты? 466
6. Как бы перехватить вывод программы в COM-порт под Windows? 467
7. Как выключить Plug and Play для COM-порта при загрузке Windows? 467
FAQ: Связь, модемы, терминалы 467
1. Что такое FIFO и FOSSIL? 467

Приложение 13. Описание компакт-диска 469

Литература и Интернет-ресурсы 471

Предметный указатель 472

Введение

В книге рассматриваются вопросы программирования, которые по непонятным причинам довольно скудно освещены в доступной литературе, — программирование последовательных коммуникационных устройств. В Интернете и на книжных полках можно найти множество книг, посвященных аппаратной организации последовательной связи. А вот вопросы программирования почему-то ограничиваются описанием битов порта или заголовками функций для работы с портами. Хочется надеяться, что представленная книга заполнит этот пробел.

Почему выбран именно последовательный интерфейс? Простота и низкие аппаратные требования (в сравнении, например, с параллельным интерфейсом) делают его заманчивым методом реализации систем обмена информацией. Видимо по этой причине форумы по программированию заполнены вопросами "Как работать с COM-портом из Windows?", "Как прочитать данные с порта" и т. д. И хотя USB-интерфейс постепенно вытесняет COM, последний остается пока наиболее распространенным, например, в системах промышленной автоматизации.

Операционная система Windows имеет настолько много функций, что начинающий программист просто не знает с чего начать и где искать информацию. Тем более, что функции для работы с портами вовсе не называются `OpenComPort`, а конфигурирование параметров порта ничем не напоминает программирование в DOS. Конечно, MSDN Library (Microsoft Development Network Library) содержит ответы на все эти вопросы, но, во-первых, только на английском языке, и, во-вторых, только на уровне заголовков функций, описания их параметров и скелетов программ. Наша книга содержит и описания, и примеры использования как низкоуровневых функций DOS, так и функций Windows.

Книга построена таким образом, чтобы переход из DOS в Windows и из Windows 98 в Windows 2000/XP стал достаточно простым и минимально трудоемким.

Все исходные коды приведены на прилагаемом компакт-диске.

Для кого эта книга

Мы адресуем книгу нескольким категориям читателей:

- тем, кому необходимо быстро разобраться в принципах работы с последовательными портами и интерфейсами, не углубляясь в дебри теории;

- тем, кого не устраивает стандартный уровень, предоставляемый функциями DOS или Windows, и кому требуется более тонкое программирование порта;
- тем, кому досталась поддержка чужих программ, работающих с оборудованием;
- тем, кому необходимо быстро переделать программу, работающую в старых операционных системах (например, DOS, Windows 95/98) в Windows 2000/XP, а времени на полную переработку нет;
- тем, кто хочет повысить свой профессиональный уровень;
- инженерам, связанным с электроникой, желающим написать программу для работы с оборудованием, не вникая в тонкости программирования;
- ну, и наконец, просто любителям подключать к компьютеру разные устройства и проводить эксперименты.

Даже если вы не относите себя ни к одной из вышеперечисленных категорий, не стоит сразу отставлять книгу на полку. Например, занимаясь не последовательными, а параллельными интерфейсами, вы можете найти необходимую информацию о создании драйверов и INF-файлов для них. Для тех, кого интересуют вопросы программирования "железа", полезным станет описание прямого доступа к портам в Windows 2000/XP и т. п.

Вы не найдете в этой книге ни подробных описаний аппаратной части последовательных интерфейсов, ни рекомендаций по разворачиванию сети на основе RS-485, ни формул для расчета максимальной длины проводника... Зато вы сможете найти листинги и исходные тексты действительно работающих программ с русскими комментариями.

Структура книги

Книга состоит из трех частей. В первой части даются теоретические сведения о последовательных интерфейсах, протоколах, методах коррекции ошибок и т. д. Вторая часть книги — практическая, про нее мы расскажем далее. В третьей части книги приводятся справочные материалы, описание функций, портов, регистров и другие необходимые для работы сведения.

Практически вся вторая часть построена на основе одного примера: передачи данных с одного последовательного порта на другой. В качестве приемника и передатчика можно использовать два компьютера, но это затруднительно как в плане их наличия, так и при отладке программы (если конечно между компьютерами нет локальной сети). В нашей книге мы будем ориентироваться на один компьютер и передавать данные с одного последовательного порта на другой. Все примеры построены по принципу усложнения задачи: от совсем простых примеров в DOS мы дойдем до сложных асинхронных программ в Windows.

Сначала мы будем работать в DOS. Для этого существует несколько причин. Во-первых, эта информация может потребоваться программистам, которым досталась поддержка чужих программ. Примеры будут построены таким образом, что переход в Windows будет достаточно простым делом. Более того, специальные модули позволят обращаться к коммуникационным портам из Windows NT/2000/XP точно так же, как это делается в DOS. Во-вторых, достаточно часто DOS (или его клон) используется в качестве операционной системы промышленных компьютеров. Ну, и наконец, не зная основ, читателю будет сложно ориентироваться в сложных функциях Windows. В DOS мы пройдем полный путь от простейшей программы в несколько строк, использующей сервис BIOS, до программы, обрабатывающей прерывания коммуникационного порта.

Программирование в Windows мы начнем с создания программы, в точности повторяющей структуру программы в DOS. Мы даже заставим работать такую программу в Windows 2000/XP. Затем освоим функции Windows. На этом пути мы познакомимся с синхронной и асинхронной связью, научимся использовать потоки и функции синхронизации. Отдельно рассмотрим функции, которые были добавлены только в Windows 2000/XP.

Программирование в Windows мы завершим созданием программы, демонстрирующей работу Plug and Play. Мы создадим эмулятор устройства, работающий с портом COM1, и заставим Windows найти новое устройство на порту COM2.

В отдельной главе мы рассмотрим интерфейс RS-485, требующий более глубоких знаний о работе коммуникационного порта.

Одна из глав будет посвящена вычислению контрольных сумм и реализации протоколов обмена.

В приложениях вы сможете найти ответы на наиболее часто задаваемые вопросы по программированию коммуникационных портов и некоторые полезные функции.

Краткое описание глав

Главы 1—5 содержат теоретическую часть книги.

- Глава 1 ("Стандарты последовательной связи") дает общие представления об аппаратной части последовательных протоколов, разъемах и о самом стандарте. Специального заострения на аппаратной части не делается. Мы расскажем о многих RS-протоколах, принципах последовательной связи, кратко опишем разъемы и кабели, а также дадим представление о ресурсах и функциях компьютера.
- В главе 2 ("Протоколы") рассматриваются основные методы организации последовательной связи с точки зрения программных протоколов обмена.

Мы расскажем о текстовых и бинарных протоколах, о методах предотвращения и обнаружения потери данных.

- В главе 3 ("Последовательные протоколы IBM PC") мы расскажем об использовании последовательной связи в персональном компьютере. Мы не будем рассматривать аппаратные реализации интерфейсов, а только лишь протоколы обмена устройств с компьютером. Эта глава будет полезна не только тем, кто интересуется работой устройств компьютера, но и тем, кто думает о создании своих протоколов обмена.
- В главе 4 ("Промышленные последовательные протоколы") рассмотрена возможность использования последовательных протоколов обмена в промышленных условиях. Мы познакомимся с протоколами MODBUS, ProfiBus и CAN. Опять же, мы будем рассматривать только программную составляющую протоколов, оставляя аппаратную реализацию за рамками книги.
- В главе 5 ("COM-порты и Plug and Play") мы расскажем о работе системы автоматического обнаружения устройств, дадим некоторые сведения о формате INF-файла, необходимого для установки драйвера устройства, и полностью опишем алгоритм работы Plug and Play для последовательных портов.

Главы 6—17 содержат практическую часть книги.

- Глава 6 ("Использование сервисов BIOS") начинает практическую часть с самой простой программы последовательной связи. Использование сервисов BIOS позволяет написать программу последовательного обмена всего из нескольких строк. К сожалению, такая программа будет очень ограничена: сервис BIOS не позволяет использовать микросхему UART на полную мощность. Эта глава будет интересна тем, кому приходится применять чужие модули последовательной связи, использующие сервисы BIOS. Кроме того, промышленные компьютеры с клоном DOS обычно предоставляют именно такой способ работы с внешними устройствами.
- В главе 7 ("Прямое программирование портов") избавиться от ограничений BIOS нам поможет прямое обращение к портам. Эти сведения будут полезны не только программистам, работающим в DOS, но и тем, кто создает свой драйвер для Windows.
- Использование прямого доступа не избавляет нас от неприятной необходимости постоянного опроса порта. В главе 8 ("Использование обработки прерываний") мы научимся обрабатывать прерывания последовательного порта и на этом завершим программирование в режиме DOS. Сведения этой главы пригодятся программистам, использующим промышленные контроллеры с установленным DOS.
- Глава 9 ("Переход в Windows") демонстрирует переход из DOS в Windows. Приведенные здесь программы еще не используют функций Windows, но

уже работают под управлением операционных систем Microsoft. Эта глава будет полезна тем, кто хочет перенести свои программы из DOS в Windows с минимальными изменениями. Вы сможете работать с портами в Windows 2000/XP так же как в DOS.

- В главе 10 ("Использование функций Windows") мы, наконец, доберемся до непосредственного использования функций Windows. Так же как в DOS мы начнем с простейшей программы обмена, но уже используя функции Windows.
- Глава 11 ("Использование потоков") по смыслу похожая на главу 8 ("Использование обработки прерываний"), рассказывает о том, как разгрузить основной процесс от постоянного опроса порта в Windows.
- Отдельный поток, созданный в главе 11, к сожалению, не избавляет от необходимости постоянного опроса порта, а значит, лишней загрузки процессора. В главе 12 ("Функции асинхронного доступа и события") используются асинхронные функции работы с портом совместно с функциями событий, которые позволяют нам создать поток, "просыпающийся" только в момент появления новых данных. Как мы покажем в нашем примере, загрузка процессора при этом будет практически незаметна.
- В главе 13 ("Специальные коммуникационные функции") мы опишем функции, которые не требовались в предыдущих главах, но потребуются в следующих. Кроме того, будут описаны функции, которые появились только в Windows 2000.
- До сих пор мы занимались приемом и передачей только одного байта. Для реальной программы этого маловато. В главе 14 ("Реализация протоколов обмена") мы создадим специальные компоненты, которые позволят нам разделить транспортные функции последовательного порта и функции работы с конкретным протоколом обмена.
- В главах теоретической части мы описывали протоколы обмена, такие как MODBUS, протоколы модемов XModem, YModem и другие, а в главе 15 ("Вычисление контрольных сумм") приведем примеры вычисления контрольных сумм, применяемых в этих протоколах.
- Несмотря на договоренность не рассматривать аппаратных реализаций протоколов, интерфейс RS-485 заслуживает отдельной главы 16 ("Интерфейс RS-485"), т. к. его программная реализация требует дополнительных знаний и функций.
- Конечно, описание протокола Plug and Play, которое мы привели в главе 5, достаточно интересно. Но ведь всегда гораздо интереснее попробовать самому! Программа из главы 17 ("Работа Plug and Play") будет эмулировать присутствие некоторого устройства, подключенного к порту COM2 с помощью обработки сообщений порта COM1. Также мы напишем INF-файл, устанавливающий "драйверы" нашего устройства.

Главы 18—23 содержат справочную часть книги.

- Глава 18 ("Сервис BIOS INT14H/INT21H") — справочник функций BIOS.
- Глава 19 ("Порты IBM PC") — справочник коммуникационных портов и их значений. С этими сведениями мы пользовались в главах 7—9.
- Глава 20 ("Структуры Windows для работы с COM-портами") — справочник структур Windows и описания полей этих структур.
- Глава 21 ("Функции Windows для работы с COM-портами") — справочник функций Windows.
- Глава 22 ("Функции `DeviceIoControl` для последовательного порта") — описание кодов прямого доступа к драйверу последовательного порта.
- Глава 23 ("AT-команды и регистры модемов") — краткий справочник основных AT-команд и регистров модемов.

Программные требования

Все программы мы будем реализовывать на языках Borland Pascal 7.0 и Borland Delphi 6. Поскольку мы не будем использовать никаких специфических функций, присущих именно этим версиям языков, все примеры могут быть скомпилированы в других версиях практически без модификации.

Версия Windows может быть любой — от Windows 98 до Windows 2000/XP. Конечно, примеры работы с функциями, которые появились только в Windows 2000, не будут работать в Windows 98, но обратной совместимости мы будем придерживаться: даже прямое программирование портов станет доступным в Windows 2000 с помощью специального модуля.

Все примеры, приведенные в книге, тестировались в версиях Windows 98 SE, Windows 2000 Workstation и Windows 2000 Advanced Server.

Для компиляции драйверов, приведенных в книге, потребуется MS Visual Studio и Windows 2000 DDK. На компакт-диске содержатся уже готовые модули драйверов, так что эту часть можно пропустить.

О программном коде

Книга содержит полные исходные коды всех программ, однако многие листинги содержат только изменения кода относительно предыдущего листинга. Такое сокращение позволяет не только экономить место, но и улучшить понимание кода, делая акцент только на новой функциональности. Код на компакт-диске содержит полные тексты без сокращений.

В программах для DOS мы не будем приводить код функций работы со строками. Код этих функций можно найти в приложениях или на компакт-диске к книге. В программах на Delphi мы не приводим код самого

проекта (DPR-файл) и код формы (DFM-файлы). Все исходные коды можно найти на компакт-диске. Еще раз повторим, что такие сокращения не означают отсутствие возможности скомпилировать и попробовать приведенные примеры на своем компьютере, а преследуют цель экономии места. Все необходимые исходные коды и модули находятся на компакт-диске.

Обозначения

При написании чисел мы будем придерживаться следующих правил:

- шестнадцатеричные числа будут иметь префикс "$", например "$45";
- шестнадцатеричные числа могут иметь префикс "0x" или постфикс "H", если того требует контекст изложения или формат строки, например "INT 14H";
- битовые последовательности заключены в угловые скобки, например "<0010>".

Аппаратные требования

Достаточно обычного домашнего компьютера, на котором компиляция программы в Delphi занимает приемлемое для вас время.

Для тестирования последовательной связи необходимо иметь либо два компьютера, либо один компьютер с двумя последовательными портами. В последнем случае нам потребуется PS/2- или USB-мышь, т. к. оба последовательных порта мы займем для экспериментов.

Для соединения портов между собой нам потребуется нуль-модемный кабель, желательно полностью распаянный, но для большей части программ достаточно трехпроводного кабеля.

Благодарности

В первую очередь автор хочет поблагодарить заместителя главного редактора Евгения Рыбакова. Без его советов и критики эта книга никогда бы не была создана. Автор благодарит всех друзей и родных, которые терпели его в процессе написания книги, а также коллектив издательства "БХВ-Петербург". Отдельная благодарность Вадиму Шарикову (компания Bixel, **http://bixel.ru**), дизайнеру сайта PvaSoft.

Обратная связь

Обо всех ошибках или пожеланиях сообщайте по адресу **books@pvasoft.com**.

ЧАСТЬ I

Протоколы и интерфейсы

Глава 1

Стандарты последовательной связи

Стандарт есть легкоусвояемая форма,
маскирующая присутствие или отсутствие содержания[1].

Глава дает общие представления об аппаратной части последовательных протоколов, разъемах и о самом стандарте. Специального заострения внимания на аппаратной части не делается.

1.1. Стандарты последовательной связи

Последовательный интерфейс используется для связи двух устройств между собой. Данные в одну сторону передаются по одному проводу с помощью последовательности битов.

Мы будем рассматривать подключение одного или нескольких устройств к компьютеру, т. к. такая комбинация наиболее часто используется в реальной жизни. Естественно, при подключении нескольких устройств к компьютеру обмен производится только с одним из этих устройств.

Для соединения со стороны компьютера используется интерфейс, называемый *COM-порт* (COMmunication port, коммуникационный порт). Этот порт обеспечивает асинхронный обмен и реализуется на микросхемах универсальных асинхронных приемопередатчиков (UATR, Universal Asynchronous Receiver Transmitter), совместимых с семейством i8250. Хотя стандарт RS-232C предусматривает и асинхронный, и синхронный режимы обмена, COM-порт компьютера поддерживает только асинхронный режим. Для реализации синхронного обмена применяются специальные адаптеры, например, SDLC или V.35.

---

[1] Все афоризмы взяты с юмористического сайта **http://www.anekdot.ru**. Часть эпиграфов принадлежит перу Стаса Янковского **http://syy.narod.ru/wordsd.htm**.

Далее мы опишем основные протоколы последовательной связи, но сначала несколько замечаний.

Слово "протокол" вносит некоторую путаницу. С одной стороны, RS-232 и RS-485 называют протоколами, а с другой, MODBUS, ZModem и CAN — тоже протоколы. Какой же из них настоящий? Дело в том, что последовательная связь состоит из двух составляющих — аппаратной и программной.

Стандарты RS-232, RS-485 и другие описывают аппаратную часть: разъемы, назначение сигналов, уровни напряжений и т. п. Вторая составляющая — программная реализация протоколов, т. е. договоренность о правилах передачи данных. В нашей книге мы не будем подробно рассматривать аппаратные реализации всех интерфейсов. Исключение составит описание разъемов, сигналов и соединительного кабеля, необходимого нам для работы. А вот токовая петля, тонкости организации сетей RS-485 и подробное описание различий между RS-протоколами останутся за рамками книги. Интерфейс RS-485 удостоится отдельного рассмотрения из-за необходимости дополнительных программных решений при его реализации.

Вторую составляющую протокола — программную реализацию — мы будем рассматривать в *гл. 2*, а создание программ отложим до второй части книги.

И еще одно замечание. Все стандарты имеют двойные, а то и тройные названия. Двойственность названий связана с тем, что Международный союз электросвязи ITU-T использует аналогичные стандарты с названиями V.xx. И, вообще говоря, два названия это еще не предел. Так, стандарту RS-232 соответствует ISO 2110, Министерство обороны США выпустило практически идентичный стандарт Mil-Std-188C, а в нашей стране подобный стандарт введен ГОСТом 18145-81.

Для краткости мы будем использовать два специальных обозначения:

- *DTE* (Data Terminal Equipment) — оконечное оборудование, принимающее или передающее данные. В качестве DTE может выступать компьютер, принтер, плоттер или другое периферийное оборудование;
- *DCE* (Data Communications Equipment) — аппаратура канала данных. Функция DCE состоит в обеспечении возможности передачи информации между двумя или большим числом DTE. Для этого DCE должно обеспечить соединение с DTE, с одной стороны, и с каналом передачи — с другой. Роль DCE чаще всего выполняет модем, например, как показано на рис. 1.1.

Устройства DTE могут быть соединены напрямую, без DCE, с помощью *нуль-модемного кабеля*, как показано на рис. 1.2. Разводка такого кабеля показана на рис. 1.3.

Рис. 1.1. Полная схема соединения по RS-232

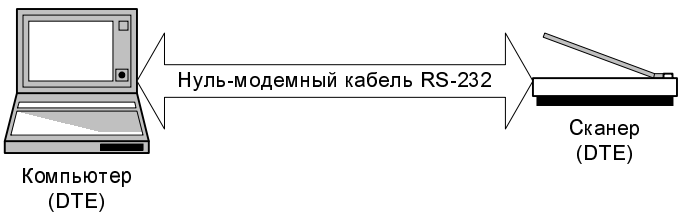

Рис. 1.2. Соединение по RS-232 нуль-модемным кабелем

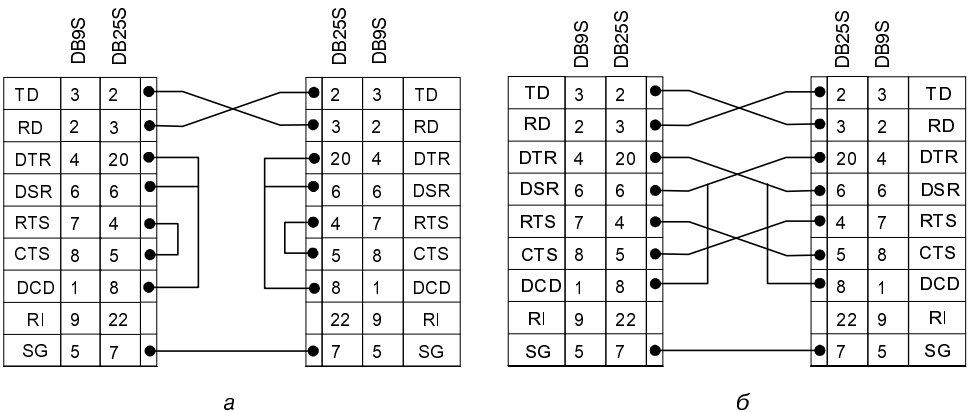

Рис. 1.3. Нуль-модемный кабель:
а — минимальный; б — полный

1.1.1. Протокол RS-232

Из-за простоты и низких аппаратных требований (в сравнении, например, с параллельным интерфейсом), последовательные интерфейсы активно используются в электронной промышленности. В настоящее время наиболее распространенным является стандарт, разработанный Ассоциацией промышленных средств связи (TIA, Telecommunication Industry Association, **http://www.tiaonline.org**) и Ассоциацией электронной промышленности (EIA,

Electronic Industries Alliance, **http://www.eia.org**) "EIA/TIA-232-E", более известный под названием "RS-232".

Стандарт RS-232 (его официальное название "Interface Between Data Terminal Equipment and Data Circuit-Termination Equipment Employing Serial Binary Data Interchange") предназначен для подключения аппаратуры, передающей или принимающей данные, к оконечной аппаратуре каналов данных. Стандарт описывает управляющие сигналы интерфейса, пересылку данных, электрический интерфейс и типы разъемов.

Интерфейс RS-232 используется и во многих устройствах обычного персонального компьютера, начиная с "мыши" и модема до ключей аппаратной защиты. И хотя уже все компьютеры имеют интерфейс USB, интерфейс RS-232 еще жив и активно применяется.

Согласно стандарту RS-232, сигнал (последовательность битов) передается напряжением. Передатчик и приемник являются *несимметричными*: сигнал передается относительно общего провода (в отличие от симметричной передачи протокола RS-485 или RS-422). В табл. 1.1 приведены границы напряжений для сигналов приемника и передатчика. Логическому нулю на входе приемника соответствует диапазон +3...+12 В, а логической единице — диапазон −12...−3 В. Диапазон −3...+3 В — зона нечувствительности, обеспечивающая гистерезис приемника (передатчика). Уровни сигнала на выходах должны быть в диапазоне −12...−5 В для представления логической единицы и +5...+12 В для представления логического нуля.

Вообще говоря, стандарт RS-232 состоит из трех частей. Первая часть, стандарт RS-232C, была принята в 1969 году и содержит описание электрических цепей и сигналов несимметричной последовательной связи. Вторая часть, стандарт RS-232D, принята в 1987 году и определяет дополнительные линии тестирования, а также формально описывает разъем DB-25. Третья часть, RS-232E, принята в 1991 году.

Таблица 1.1. *Границы напряжений COM-порта (стандарт RS-232)*

	Диапазон напряжения входа приемника	**Диапазон напряжения выхода передатчика**	**Состояние управляющего сигнала**	**Состояние линии данных**
Логический 0	от –12 до –3 В	от –12 до –5 В	ON	MARX
Логическая 1	от +3 до +12 В	от +5 до +12 В	OFF	SPACE

Важно!

Интерфейс не обеспечивает гальванической развязки устройств. Подключение и отключение интерфейсных кабелей устройств с независимым питанием должно производиться при отключенном питании.

Один из самых неприятных недостатков стандарта RS-232 — плохая помехозащищенность и, соответственно, короткие линии передачи. Естественно, были созданы стандарты, решающие эти проблемы.

1.1.2. Протокол RS-422A

Стандарт RS-422A (другое название ITU-T V.11) определяет электрические характеристики симметричного цифрового интерфейса. Он предусматривает работу на более высоких скоростях (до 10 Мбит/с) и больших расстояниях (до 1000 м) в интерфейсе DTE-DCE. Для его практической реализации, в отличие от RS-232, требуются два физических провода на каждый сигнал. Реализация симметричных цепей обеспечивает наилучшие выходные характеристики.

Подобно V.28, данный стандарт является простым описанием электрических характеристик интерфейса и не определяет параметры сигналов, типы разъемов и протоколы управления передачей данных. Для линий интерфейсов RS-422A и RS-423A могут быть использованы различные проводники (или пары проводников) одного и того же кабеля.

Стандарт RS-422A был разработан совместно с RS-423A и позволяет размещать линии этих интерфейсов в одном кабеле. Он несовместим с RS-232, и взаимодействие между RS-422A и RS-232 может быть обеспечено только при помощи специального интерфейсного конвертера.

Практически это полнодуплексный протокол RS-485. Прием и передача идут по двум отдельным парам проводов, причем на каждой паре может быть только по одному передатчику.

1.1.3. Протокол RS-423A

Стандарт RS-423A (другое название V.6) определяет электрические характеристики несимметричного цифрового интерфейса. "Несимметричность" означает, что данный стандарт, подобно RS-232, для каждой линии интерфейса использует только один провод. При этом для всех линий используется единый общий провод.

Как и RS-422A, этот стандарт не определяет сигналы, конфигурацию выводов или типы разъемов. Он содержит только описание электрических характеристик интерфейса. Стандарт RS-423A предусматривает максимальную скорость передачи 100 Кбит/с.

1.1.4. Протокол RS-485

Полное название этого стандарта TIA/EIA-485A (аналогичный стандарт описывается в ISO 8482). Он включает в себя 17 страниц и определяет требования к электрическим характеристикам формирователей и приемников.

В нем ничего не говорится ни о качестве сигнала, ни о методах доступа к линиям связи, ни о протоколе обмена, ни о технологической схеме (назначении контактов и т. п.). "Приложение А" этого стандарта описывает принципы работы устройств на основе RS-485.

В дополнение к описанию стандарта был опубликован документ TSB89, называющийся "Application Guidelines for TIA/EIA-485A". TSB89 состоит из 23 страниц и посвящается разъяснению, как применять устройства, определяемые стандартом TIA/EIA-485A для физических сетей.

Подробно этот интерфейс будет обсуждаться в *гл. 16*.

1.1.5. Протокол RS-499

Стандарт RS-449, в отличие от RS-422A и RS-423A, содержит информацию о параметрах сигналов, типах разъемов, расположении контактов и т. п. В этом отношении RS-449 является дополнением к стандартам RS-422A и RS-423A. Стандарту RS-449 соответствует международный стандарт V.36.

Комбинация RS-449, RS-422A и (или) RS-423A первоначально предназначалась для замены RS-232. Однако этого не произошло, хотя данные стандарты нашли достаточно широкое применение в качестве высокоскоростного интерфейса DTE-DCE.

Стандарт RS-449 определяет 30 сигналов интерфейса, большинство из которых имеют эквиваленты в RS-232, но добавлены и новые. Обозначения большинства сигналов были изменены во избежание путаницы.

Десять сигналов RS-449 определены как линии первой категории. Эта группа сигналов включает в себя все основные сигналы данных и синхронизации, такие как "Передаваемые данные", "Принимаемые данные", "Синхронизация терминала". Скорость передачи сигналов первой категории существенно зависит от длины кабеля. Для линий этой категории на скоростях до 20 Кбит/с могут использоваться стандарты RS-422A либо RS-423A; на скоростях выше 20 Кбит/с (до 2 Мбит/с) — только RS-422A.

Оставшиеся 20 линий классифицируются как линии второй категории и используются стандартом RS-423A. К этой категории относятся такие управляющие линии, как "Качество сигнала", "Выбор скорости передачи" и др.

Стандарт RS-449 определяет тип разъема и, в отличие от RS-232, распределение контактов разъема. Используемые разъемы имеют 37 контактов для прямого и 9 контактов для обратного канала.

1.1.6. Протокол RS-562

Протокол RS-562 представляет собой низковольтную реализацию RS-232 с максимальным уровнем сигнала 3,5 В.

1.1.7. Протокол V.24

Рекомендация V.24 содержит описание линий и набора сигналов обмена между DTE и DCE. В RS-232 используются другие обозначения линий, однако линии интерфейса RS-232 и рекомендации V.24 выполняют совершенно одинаковые функции. V.24 определяет большее количество линий, чем RS-232, поскольку стандарт V.24 используется и в других интерфейсах. В этом смысле RS-232 является подмножеством V.24. Рекомендация V.24 не определяет электрические характеристики или другие физические аспекты реализации, такие как тип разъема, расположение контактов, длина кабеля и скорость обмена. Технические вопросы реализации интерфейса подробно изложены в стандарте V.28.

1.1.8. Протокол V.28

Рекомендация V.28 определяет только электрические характеристики интерфейса V.24, обеспечивающего работу по несимметричным двухполярным линиям обмена на скоростях до 20 Кбит/с. К таким характеристикам относятся уровни используемых сигналов, емкостное сопротивление и т. д. Данная рекомендация не содержит требований к длине кабеля, типу разъемов и расположению их контактов. Поэтому рекомендация V.28 может рассматриваться как подмножество стандарта RS-232.

1.1.9. Протокол V.35

Стандарт V.35 появился в начале 1980-х годов как спецификация интерфейса между устройствами доступа к сети (мультиплексором, модемом или др.) и высокоскоростной сетью с коммутацией пакетов. Первоначально эта спецификация использовалась для подключения групповых модемов (модемных пулов) к коммутационному устройству.

Рекомендация V.35 определяет синхронный интерфейс для работы по аналоговым широкополосным каналам с полосой пропускания 60—108 кГц и скоростью передачи до 48 Кбит/с.

В приложении к стандарту определяется вид электрического соединения, обеспечивающего высокоскоростной последовательный интерфейс между мультиплексором и коммутационным оборудованием сети.

Модемы, использующие стандарт V.35, не получили широкого распространения на рынке, но интерфейс в качестве высокоскоростной замены RS-232 прижился. В спецификации стандарта не был определен тип электрического разъема, но фирма IBM в свое время стала выпускать совместимые с V.35 большие прямоугольные разъемы с массивными прижимными винтами. Получилось очень надежное соединение. Остальные производители коммутационной техники стали повторять конструкцию соединителя IBM, который и стал стандартом де-факто, и был принят в качестве рекомендации ISO 2593.

1.1.10. Протокол X.21

Стандарт X.21 впервые был опубликован в 1972 году. Он определяет физические характеристики и процедуры управления для интерфейса DTE-DCE в режиме синхронной передачи данных и может применяться как в сетях с коммутацией каналов, так и в сетях на выделенных линиях. Стандарт предусматривает дуплексную работу при условии, что устройства связаны друг с другом реальными, а не виртуальными цифровыми линиями связи. Функциональные процедуры X.21 формализованы в виде диаграмм состояний, рассмотрение которых выходит за рамки данной книги.

Рекомендация ITU-T X.21 определяет формат передаваемых символов, которые представляются в коде MTK-5 (Международный телеграфный код № 5). Данный интерфейс рассчитан на сквозную цифровую передачу. В нем процесс установления соединения и разъединения полностью автоматизирован при помощи набора сигналов о состоянии соединения и о его неисправностях. В ходе передачи данных через интерфейс могут передаваться любые последовательности битов.

Создатели этого стандарта стремились максимально упростить его. Так, соединение DTE с DCE требует существенно меньшего числа сигнальных линий, чем аналогичное соединение для интерфейса RS-232.

1.1.11. Рекомендация X.21bis

Рекомендация X.21bis была разработана для обеспечения возможности подключения к сетям передачи данных общего пользования тех пользователей, которые используют для этого аналоговые выделенные или коммутируемые каналы и имеют синхронные модемы, работающие согласно рекомендациям серии V.

1.1.12. Краткое сравнение RS-протоколов

В табл. 1.2 мы привели краткое сравнение 4 наиболее распространенных RS-протоколов. Более подробную информацию можно найти в соответствующей литературе.

Таблица 1.2. *Сравнение последовательных интерфейсов*

	RS-232CV.24	RS-422AV.11	RS-423AV.6	RS-485
Максимальная длина кабеля, м	15	1200	1200	1200
Число приемников	1	10	10	32
Число передатчиков	1	1	1	32

Таблица 1.2 *(окончание)*

	RS-232CV.24	RS-422AV.11	RS-423AV.6	RS-485
Максимальная скорость передачи	20 Кбит/с	100 Кбит/с	10 Мбит/с	10 Мбит/с

1.2. Принципы последовательной связи

При асинхронной последовательной связи (напомним, что именно асинхронный интерфейс реализован в IBM PC) одно из устройств посылает или принимает байты информации по одному биту. Интервалы времени между байтами при этом несущественны, но времена между отдельными битами байта очень важны. Сигнал на линии может быть высокого или низкого уровня, что соответствует логическим нулю (SPACE) и единице (MARK). Линия поддерживается в отмеченном состоянии, когда по ней нет передачи данных. При начале передачи байта данных сигнал падает в 0, отмечая стартовый бит. Затем следуют биты данных (от 5 до 8) в виде набора высоких и низких уровней. Последний бит данных может сопровождаться битом четности, используемым для обнаружения ошибок, а затем в последовательность включаются один или более стоп-битов, которым соответствует высокий уровень. Эти стоп-биты начинают отмеченное состояние, которое будет сохраняться до тех пор, пока не начнется передача следующего байта данных. Число используемых стоп-битов существенно, поскольку они устанавливают минимальное время, которое должно пройти перед следующим стартовым битом (рис. 1.4).

Рис. 1.4. Передача байта с помощью последовательности битов

Конечно, передающая и приемная станции должны использовать один и тот же протокол для этих цепочек битов и они должны работать с одной и той же скоростью обмена (измеряемой в битах в секунду, называемых также *бодами*).

1.3. Разъемы коммуникационного порта

Кроме электрических характеристик, RS-стандарт регламентирует разъемы, используемые для соединения устройств (рис. 1.5).

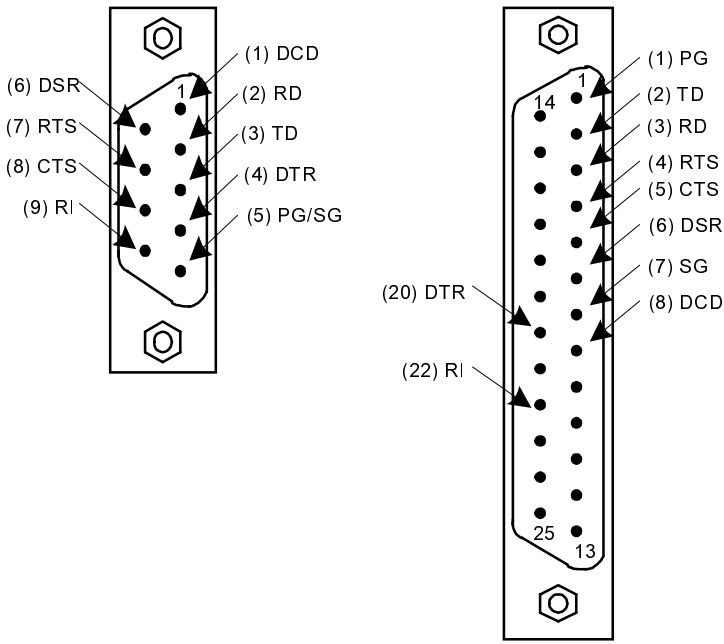

Рис. 1.5. Разъемы DB-9 и DB-25, используемые в COM-портах компьютера

На COM-портах компьютера используются два типа разъемов — DB-25P и более компактный вариант DB-9P. На аппаратуре используются разъемы DB-25S или DB-9S. Соответствие сигналов 9- и 20-штырьковых разъемов и их описания приводятся в табл. 1.3.

Таблица 1.3. *Сигналы интерфейса RS-232*

Контакт		Обозначение		Описание	Аббре-виатура	I/O
DB-25	DB-9	EIA (RS-232)	ITU-T (V.24)			
1		AA		Защитное заземление	GND	
2	3	BA	103	Передаваемые данные	TxD (TD)	O

Таблица 1.3 *(продолжение)*

Контакт		Обозначение		Описание	Аббре-виатура	I/O
DB-25	DB-9	EIA (RS-232)	ITU-T (V.24)			
3	2	BB	104	Принимаемые данные	RxD (RD)	I
4	7	CACJ	105	Запрос передачи	RTS	O
			133	Готовность к приему		
5	8	CB	106	Готовность к передаче	CTS	O
6	6	CC	107	Готовность DCE	DSR	I
7	5	AB	102	Сигнальное заземление	SG	I/O
8	1	CF	109	Обнаружение несущей	DCD	I
9				Резерв для теста DCE, +12 В		I
10				Резерв для теста DCE, –12 В		I
11			126	Выбор частоты передачи		O
12		SCF	122	Обнаружение несущей дополнительного канала	SDCD	I
13		SCB	121	Готовность к передаче по дополнительному каналу	SCTS	I
14		SBA	118	Передаваемые данные дополнительного канала	STD	O
15		DB	114	Синхронизация передачи	TC	I
16		SBB	119	Принимаемые данные дополнительного канала	SRD	I
17		DD	115	Синхронизация приема	RC	I
18		LL	141	Свободный (местный шлейф)		O
19		SCA	120	Запрос передачи дополнительного канала	SRTS	O
20	4	CD	108.1	Готовность DCE	DTR	O
			108.2	Готовность DTE		O
21		CG	110	Детектор качества сигнала		I
		RL	140	Удаленный шлейф		O

Таблица 1.3 *(окончание)*

Контакт		Обозначение		Описание	Аббре-виатура	I/O
DB-25	DB-9	EIA (RS-232)	ITU-T (V.24)			
22	9	CE	125	Индикатор вызова	RI	I
23		CH	111	Переключатель скорости передачи данных (DTE)		O
		CI	112	Переключатель скорости передачи данных (DCE)		I
24		DA	113	Синхронизация передачи (DTE)		O
25		TM	142	Свободный (индикатор тестирования)		I

Все линии обмена между DTE и DCE можно разбить на четыре основные группы:

- ❒ линии данных;
- ❒ линии управления;
- ❒ линии синхронизации;
- ❒ линии "земли".

Далее мы приведем краткое описание сигналов. Подробное описание выходит за рамки книги по программированию.

1.3.1. Сигнальная "земля" (AB/SG)

Эта линия является общим проводом для всех электрических цепей, образуемых линиями физического интерфейса. Стандарт рекомендует присоединять этот общий провод к защитной "земле" путем внутреннего соединения в DCE. Смысл такого соединения заключается в том, что корпуса устройств оказываются заземленными через штепсельную розетку.

1.3.2. Защитная "земля" (AA)

Эта линия присутствует только в интерфейсе с разъемом DB-25 и предполагает соединение с корпусом устройства.

1.3.3. Передаваемые данные (BA/TxD/TD)

Сигналы, которые присутствуют на этой линии, вырабатываются местным (локальным) DTE для передачи местному DCE. Посылаемые сигналы могут быть кодами команд, управляющих работой местного DCE (AT-команды или другие) или данными, которые местное DCE должно передать удаленному DCE-устройству.

Если DTE не передает данные, то оно удерживает эту линию в состоянии логической единицы (MARK). Согласно стандарту, DTE не будет передавать данные до тех пор, пока управляющие линии "Запрос передатчика", "Сброс передатчика", "Готовность DCE" и "Готовность DTE" не будут находиться одновременно в активном (ON) состоянии.

Независимо от того, относится ли данное устройство к DTE или DCE, рассматриваемая линия всегда называется одинаково: "Передаваемые данные". Это выходная линия для DTE и входная для DCE.

1.3.4. Принимаемые данные (BB/RxD/RD)

Сигналы, которые присутствуют на этой линии, вырабатываются местным (локальным) DCE для передачи местному DTE. Передаваемые сигналы могут быть ответами на команды или данными, получаемыми от удаленного DCE.

Если не выполняется операция подтверждения приема команды, стандартное DCE удерживает эту линию в состоянии логической единицы (MARK) при условии, что линия "Указатель несущей" находится в неактивном состоянии (OFF).

При полудуплексной работе эта линия удерживается в состоянии MARK, когда линия "Запрос передачи" находится в активном состоянии, а также в течение короткого промежутка времени после ее перехода из активного состояния в неактивное.

Независимо от того, относится ли данное устройство к DTE или DCE, рассматриваемая линия всегда называется одинаково: "Принимаемые данные". Это выходная линия для DCE и входная для DTE.

1.3.5. Запрос передачи (CA/RTS)

Сигналы на этой линии вырабатывает DTE. В симплексных или дуплексных системах активное состояние этой линии обеспечивает удержание DCE в режиме передачи. Переключение в неактивное состояние приостанавливает передачу. В обоих случаях состояние этой линии никак не влияет на работу DCE-устройства как приемника.

В полудуплексных системах переключение этой линии в активное состояние переводит DCE в режим передачи и приостанавливает его работу на прием. Когда DTE переключает эту линию в неактивное состояние, соответствующее DCE-устройство начинает работать в режиме приема.

Если DTE переключило линию "Запрос передачи" в неактивное состояние, оно не должно снова активизировать эту линию до тех пор, пока DCE-устройство не подтвердит прием этого сигнала путем переключения в такое же неактивное состояние линии "Готовность к передаче".

Переключение линии "Запрос передачи" из неактивного в активное состояние является сигналом на переход DCE в режим передачи. DCE может затем выполнять любые действия, необходимые для подготовки к передаче, и после их завершения устанавливает линию "Готовность к передаче" в активное состояние, сообщая тем самым, что DCE может передавать данные.

Переключение линии "Запрос передачи" из активного в неактивное состояние является сигналом для DCE на завершение обработки любых данных, которые уже получены от DTE-устройства. Затем DCE прекращает передачу или переходит в режим приема. О завершении этого процесса оно сообщает путем переключения линии "Готовность к передаче" в неактивное состояние.

1.3.6. Готовность к передаче (CB/CTS)

Сигналы на этой линии вырабатывает DCE. Эти сигналы сообщают о готовности DCE к приему данных от связанного с ним DTE-устройства. Если линия "Готовность к передаче" находится в неактивном состоянии, DTE не должно передавать данные. Когда DCE переключает эту линию в активное состояние, оно готово принимать данные. Эти данные могут быть командами для DCE или данными, передаваемыми по каналу связи.

Обычно сигнал "Готовность к передаче" является ответом на сигнал "Запрос передачи". Однако DCE может независимо переключить линию "Готовность к передаче" в неактивное состояние, чтобы сообщить DTE о необходимости приостановки передачи данных на некоторый конечный промежуток времени. Любые данные, переданные после переключения линии "Готовность к передаче" в неактивное состояние, могут быть проигнорированы DCE-устройством. DCE может снова активизировать эту линию в любой момент при условии, что линия "Запрос передачи" также находится в активном состоянии. Такая процедура называется "аппаратное управление потоком данных".

Если линия "Запрос передачи" не используется, DCE будет работать так, будто эта линия все время находится в активном состоянии.

1.3.7. Готовность DCE (CC/DSR)

DCE использует эту линию для информирования DTE о своей готовности к работе. Для соответствующего сигнала часто используется название: "Готовность устройства сопряжения" или "Готовность модема". Активное состояние линии означает, что DCE готово обмениваться информацией с DTE и начать передачу данных.

В некоторых реализациях данная линия, в комбинации с линией "Индикатор тестирования", используется для управления обменом сигналами при тестировании и обслуживании DCE. В других случаях эта линия используется вместе с линией "Готовность к передаче" для управления и программирования DCE, поддерживающего последовательную систему автоматического вызова.

1.3.8. Готовность DTE (CD/DTR)

Сигналы на этой линии вырабатывает DTE. Переключение этой линии в активное состояние информирует DCE-устройство о том, что ему нужно приготовиться к соединению с каналом связи. Если DCE может автоматически отвечать на последующие вызовы, оно будет делать это только в том случае, если линия "Готовность DTE" находится в активном состоянии. Состояние данной линии не влияет на сигналы, присутствующие на линии "Индикатор вызова".

Если текущее соединение с каналом связи установлено, то активное состояние линии "Готовность DTE" указывает, что DCE должно поддерживать это состояние. Если эта линия впоследствии переключается в неактивное состояние, DCE отключится от канала связи после завершения текущей передачи данных. После перехода в неактивное состояние линия "Готовность DTE" не должна активизироваться снова до тех пор, пока от DCE не будет получено подтверждение этого перехода путем переключения линии "Готовность DCE" в неактивное состояние.

1.3.9. Индикатор вызова (CE/RI)

DCE использует эту линию для сообщения о том, что по каналу связи принимается сигнал вызова. Сигнал на линии "Индикатор вызова" соответствует состоянию сигнала вызова ON при наличии сигнала вызова, и состоянию OFF — при его отсутствии. Эта линия всегда активна. Однако DTE может игнорировать этот сигнал по своему усмотрению.

1.3.10. Обнаружение несущей (CF/DCD)

DCE активизирует эту линию при получении сигнала, служащего указателем возможности установления соединения с подходящим качеством связи.

Если линия находится в неактивном состоянии, то это означает либо полное отсутствие сигнала, либо наличие сигнала неудовлетворительного качества. Какой сигнал считать подходящим по качеству, DCE определяет само.

Если во время передачи данных возникнут обстоятельства, требующие переключения линии "Обнаружение несущей" в неактивное состояние (означающего потерю несущей), DCE также установит сигнал MARK на линии "Принимаемые данные".

В полудуплексных системах данная линия переключается в неактивное состояние всякий раз, когда активизируется линия "Запрос передачи", а также в течение короткого промежутка времени после переключения линии "Запрос передачи" из активного в неактивное состояние.

1.3.11. Детектор качества сигнала (CG/SQ)

Использование этой линии в настоящее время не рекомендуется.

1.3.12. Переключатель скорости передачи данных от DTE (CH)

По этой линии DTE сигнализирует о том, какую из двух возможных скоростей передачи данных или какой диапазон скоростей передачи должно выбрать DCE. Активное состояние этой линии соответствует выбору более высокой скорости передачи.

1.3.13. Переключатель скорости передачи данных от DCE (CI)

По этой линии DCE сообщает о том, какую из двух возможных скоростей передачи данных или какой диапазон скоростей передачи оно выбирает. Активное состояние этой линии соответствует выбору более высокой скорости передачи.

1.3.14. Готовность к приему (CJ)

Для обеспечения документированного метода аппаратного управления потоком данных стандартом RS-232 предусмотрена линия "Готовность к приему". DTE активизирует эту линию, чтобы сообщить DCE о своей готовности к приему данных.

Напротив, неактивное состояние этой линии означает, что DTE не может принимать данные от DCE. В этом случае DCE должно сохранить не переданные данные. Локальное DCE-устройство может передать удаленному DCE сигнал на приостановку передачи данных по каналу связи.

В системах, использующих линию "Готовность к приему", все остальные линии работают так, как если бы линия "Запрос передачи" постоянно находилась в активном состоянии.

1.3.15. Местный шлейф (LL)

DTE использует эту линию для перевода локального DCE в режим петлевого тестирования. Когда DTE активизирует линию "Местный шлейф", локальное DCE-устройство отключает свой сигнальный выход от канала связи и подключает его к своей собственной входной линии. Затем это DCE активизирует линию "Индикатор тестирования". В результате этого любые данные, передаваемые от DTE к DCE, немедленно возвращаются обратно к DTE. При переключении линии "Местный шлейф" в неактивное состояние DCE реконфигурирует себя для нормальной работы. Состояние линии "Местный шлейф" не влияет на работу линии "Индикатор вызова".

1.3.16. Удаленный шлейф (RL)

DTE использует эту линию для перевода удаленного DCE в режим дистанционного тестирования. Когда DTE активизирует линию "Удаленный шлейф", локальное DCE выдает команду удаленному DCE на установку петлевой конфигурации. Когда установка такой конфигурации завершена, локальное DCE переключает линию "Индикатор тестирования" в активное состояние.

При дистанционном тестировании данные, передаваемые локальным DTE-устройством, проходят через локальное DCE-устройство и далее поступают в канал связи. Удаленное DCE принимает эти данные и сразу же передает их обратно по каналу связи к локальному DCE, а последнее — к локальному DTE. Когда локальное DTE переключает линию "Удаленный шлейф" в неактивное состояние, локальное DCE выдает команду удаленному DCE на окончание тестирования.

Во время дистанционного тестирования удаленное DCE устанавливает линию "Готовность DCE" в неактивное, а линию "Индикатор тестирования" в активное состояние, указывая тем самым, что связь с удаленным DCE невозможна.

1.3.17. Индикатор тестирования (TM)

DCE активизирует эту линию для того, чтобы сообщить DTE о своем переходе в тестовый режим. Активизация этой линии является откликом DCE-устройства на переключение линий "Местный шлейф" или "Удаленный шлейф" в активное состояние. Линия "Индикатор тестирования" активизируется также в том случае, когда DCE отвечает на команду перехода в режим петлевого тестирования, поступающего от удаленного DCE. Неак-

тивное состояние линии "Индикатор тестирования" означает, что DCE готово для нормальной работы.

1.3.18. Синхронизация передачи от DTE (DA)

По этой линии DTE передает сигналы для синхронизации DCE. Моменты переключения этой линии из активного состояния в неактивное номинально соответствуют середине каждого элементарного сигнала (импульса), поступающего от DTE на линию "Передаваемые данные". Если эта линия реализована в интерфейсе, то для поступления на нее синхронизирующей информации обычно достаточно, чтобы DTE-устройство находилось во включенном состоянии.

1.3.19. Синхронизация передачи от DCE (DB/TC)

По этой линии DCE передает сигналы для синхронизации DTE. DCE должно выдавать элементарные сигналы на линию "Передаваемые данные" таким образом, чтобы значащие моменты переходов между соседними элементарными сигналами (битами) соответствовали моментам переключения линии DB из неактивного состояния в активное.

1.3.20. Синхронизация приема от DCE (DD/RC)

По этой линии DCE передает сигналы для обеспечения синхронной работы DTE в режиме синхронной передачи данных. Моменты переключения этой линии из активного состояния в неактивное соответствуют середине каждого элементарного сигнала (бита), поступающего от DCE на линию "Принимаемые данные".

1.3.21. Передаваемые данные дополнительного канала (SBA/STD)

Эта линия эквивалентна линии "Передаваемые данные", но используется для организации дополнительного канала связи.

1.3.22. Принимаемые данные дополнительного канала (SBB/SRD)

Эта линия эквивалентна линии "Принимаемые данные", но используется для организации дополнительного канала связи.

1.3.23. Запрос передачи по дополнительному каналу (SCA/SRTS)

Эта линия эквивалентна линии "Запрос передачи", но используется для организации дополнительного канала связи.

1.3.24. Готовность к передаче по дополнительному каналу (SCB/SCTS)

Эта линия эквивалентна линии "Готовность к передаче", но используется для организации дополнительного канала связи.

1.3.25. Обнаружение несущей дополнительного канала (SCF/SDCD)

Эта линия эквивалентна линии "Обнаружение несущей", но используется для организации дополнительного канала связи.

1.4. Ресурсы IBM PC для последовательной связи

Реализация последовательной связи возможна на нескольких "уровнях":

- низкоуровневый доступ к портам:
  - использование сервисов BIOS;
  - прямое программирование портов;
  - использование прерываний;
- операционная система Windows:
  - прямое программирование портов;
  - использование потоков;
  - использование событий.

1.4.1. Сервисы BIOS

Компьютер может иметь до четырех последовательных портов COM1—COM4 (для обычных компьютеров типично наличие двух портов) с поддержкой на уровне BIOS. Однако сервис BIOS (INT14H) обеспечивает работу с портом только на скоростях 110—9600 бит/с, что чаще всего значительно меньше реально возможной скорости. Для обхода этой проблемы используется прямое программирование коммуникационного порта.

Мы рассмотрим использование сервисов BIOS в *гл. 6*. Эта информация может пригодиться тем, кто вынужден поддерживать чужие программы, написанные с использованием этих сервисов. Кроме того, промышленные компьютеры с встроенным клоном DOS обычно предоставляют именно такой сервис для последовательной связи с внешними устройствами (справедливости ради, отметим, что DOS-клоны промышленных компьютеров имеют функцию INT14H с кодом 80H для работы на скоростях 9600—115200 бит/с).

Справочная информация о функциях BIOS для последовательных портов приводится в *гл. 18*.

1.4.2. Коммуникационные порты

Коммуникационные порты занимают в пространстве ввода/вывода по восемь смежных 8-битовых регистров и могут располагаться по стандартным базовым адресам. Прямое программирование портов предоставляет полный доступ к возможностям UART микросхемы.

Эта информация может пригодиться не только тем, кто пишет программы для микроконтроллеров для DOS, но и использующим Windows DDK, т. к. любые программные надстройки базируются на возможностях аппаратуры.

Прямым программированием портов мы займемся в *гл. 7*, а полное описание портов и их регистров можно найти в *гл. 19*.

1.4.3. Использование прерываний

Порты могут генерировать аппаратные прерывания. Это позволяет создавать свои обработчики прерываний, вызывающиеся только при возникновении некоторых условий (например, при получении нового символа или возникновении ошибок приема).

Использование прерываний максимально использует ресурсы асинхронной микросхемы при программировании последовательной связи на низком уровне. Программу, использующую прерывания, мы создадим в *гл. 8*, а полное описание ресурсов контроллера прерываний приводится в *гл. 19*.

1.4.4. Прямое программирование портов в Windows

Windows 95/98 не блокируют прямой доступ к портам. В *гл. 9* мы покажем, как получить доступ к портам в Windows 95/98, а также как обойти запрет на доступ к портам в Windows NT/2000/XP.

Эта информация необходима программистам, занимающимся поддержкой существующих программ для Windows 95/98 или желающим перенести имеющийся код в Windows 2000 без серьезных изменений.

Более того, мы опишем метод, с помощью которого можно будет запустить любой скомпилированный файл, работающий с портами напрямую в операционных системах Windows NT/2000/XP.

1.4.5. Функции Windows

Конечно, при написании новой программы для Windows правильнее использовать функции Windows. В *гл. 10* мы будем использовать базовые функции Windows. Затем, в *гл. 11*, мы покажем, как использовать потоки, позволяющие существенно разгрузить основной поток программы.

В *гл. 12* мы покажем, как использовать функции асинхронного доступа и события.

В *гл. 13* мы опишем использование дополнительных коммуникационных функций, которые потребуются нам при реализации протоколов обмена, а также функции, добавленные только в Windows 2000/XP.

Глава 2

Протоколы

"Теперь позвольте пару фраз без протокола..."

В. Высоцкий, "Милицейский протокол"

В этой главе описываются типы протоколов обмена, методы организации передачи данных и методы повышения достоверности передачи.

2.1. Что такое протокол обмена

Чтобы принимающее устройство умело правильно интерпретировать полученные данные, необходима договоренность между приемником и передатчиком — *протокол обмена*. Протокол обмена состоит из нескольких частей:

- договоренность о параметрах связи (скорость, четность, число бит в байте и т. д.);
- договоренность о последовательности передачи данных и форматах представления чисел;
- договоренность об инициаторе обмена (если необходимо);
- процедура восстановления связи (если необходимо);
- дополнительные договоренности (например, передача прав главного хоста в сетях RS-485).

Параметры связи обязывают приемник и передатчик работать с одинаковыми аппаратными настройками. Обычно промышленные устройства используют строго определенные настройки обмена, некоторые — позволяют задавать скорость обмена (как программно, так и с помощью перемычек). Настройки четности и число бит в байте чаще всего фиксируются и не могут настраиваться со стороны аппаратуры.

Иногда, для согласования скорости работы, устройства выбирают фиксированную скорость начала обмена, устанавливают соединение и, обмениваясь данными на этой скорости, "договариваются" об использовании других скоростей и параметров обмена.

Для того чтобы принимающее устройство могло воспользоваться полученной информацией, ему необходимо знать, как ее интерпретировать. Для этого приемник и передатчик декларируют последовательность передачи данных. Часто именно эту часть договоренностей называют протоколом обмена, подразумевая наличие остальных частей.

При использовании приемником и передатчиком одной линии связи (как, например, в интерфейсе RS-485), необходима договоренность об инициаторе обмена. Это тем более необходимо в тех случаях, когда приемник один, а передатчиков много.

При разрыве связи одно из устройств может ждать ответа другого, а то, в свою очередь, ответа первого. Таким образом, без договоренности о действиях в случае разрыва связи устройства не смогут связаться друг с другом повторно. Часто выбирается очень простой способ — если в течение некоторого времени устройства не получают данных друг от друга, они переходят в режим "первого запуска". Однако возможны и более экзотические договоренности.

С точки зрения формата передаваемых данных, типы протоколов обмена можно поделить на две категории — строковые и бинарные. В первом случае информация передается обычными ASCII-символами (American Standart Code for Informational Interchange, американский стандартный код для обмена информацией), а во втором — бинарным потоком данных. Далее мы рассмотрим оба эти типа, а их реализацией займемся в *гл. 14*.

2.2. ASCII-протокол передачи данных

В таком протоколе управляющие символы и данные передаются с помощью ASCII-символов.

Пример пакета данных в таком формате приведен в табл. 2.1. Пакет данных начинается со стартовой последовательности — одного или нескольких символов, обозначающих начало данных. Символы стартовой последовательности не должны появляться внутри блока данных. Получив байты начала посылки, все абоненты настраиваются на получение полного пакета данных. Затем, получив и сравнив номер абонента со своим номером, абонент либо продолжает обработку пакета, либо снова переходит в режим ожидания начала посылки. Таким образом, пакет обрабатывает только один абонент сети. После получения байта конца посылки абонент проверяет контрольную сумму и обрабатывает полученные данные. Конец посылки также может представляться несколькими символами. Разумеется, строка конца посылки не должны встречаться внутри самих данных.

Естественно, если связь осуществляется между двумя абонентами, поле "номер абонента" может отсутствовать.

Таблица 2.1. *Структура ASCII-протокола*

Байт	Описание	Примеры	
Начало посылки	Идентификатор начала посылки. Любой символ (или несколько символов), не встречающийся в символах данных	$	!!!
Номер абонента	Две десятичные цифры номера абонента (дополненные при необходимости нулем). В некоторых случаях могут оговариваться специальные номера абонентов, например, "**", означающая посылку всем абонентам сети	01	15
Данные	Последовательность цифр данных. Числа с плавающей точкой разделяются знаком ".". Части данных могут разделяться пробелом	011 03.1	155 17.5
Контроль	Контрольная сумма данных. Обычно один или два байта суммы всех байт данных с переполнением. Байты суммы также передаются в десятичном виде (три десятичные цифры с выравниванием нулями слева)	002	104
Конец посылки	Идентификатор конца посылки. Для ASCII-протоколов обычно символ LR (перевод строки)	$0D	***

Достоинство такого протокола в удобстве отладки: прием и передачу команд и данных можно осуществлять с помощью обычной программы-терминала (например, Telnet).

Основной недостаток — большой объем данных, необходимых для передачи. Например, для передачи одного байта шестнадцатеричной строкой потребуется два байта (число 230 будет передано последовательностью $E, $6). А при передаче обычными десятичными цифрами — три байта (число 230 будет передано тремя цифрами "2,3,0"). Передача же чисел с плавающей точкой потребует количество байт, пропорциональное точности передаваемого числа (плюс десятичный разделитель). Например, число 221,73 будет передаваться с помощью последовательности символов "2,2,1,.,7,3", т. е. с помощью 6 байт.

Второй недостаток — необходимость преобразования двоичных данных в ASCII-строки и обратно. Если на языке высокого уровня (C, Pascal) это не проблема, то, например, для 8-битового контроллера может потребоваться значительный объем кода (пример кода, реализующего такое преобразование приводится в *прил. 8*).

Хорошим примером ASCII-протокола является MODBUS-ASCII *(см. разд. 4.1.1)*, а программной реализацией своего ASCII-протокола мы займемся в *гл. 14*.

2.3. Бинарный протокол передачи данных

Бинарная передача значительно более компактна. Например, для передачи числа с плавающей точкой типа `SINGLE` *(см. прил. 3)* требуется 4 байта независимо от числа знаков после десятичной точки.

Так же как в ACSII-протоколе, выбирается некоторый символ для обозначения начала посылки. Завершение ожидания данных производится либо по получению необходимого числа байт, либо по тайм-ауту: принимающее устройство отсчитывает время с получения последнего байта до приема следующего. Если эта пауза превышает некоторый интервал (обычно 1,5—2 байта), посылка считается потерянной и приемный буфер сбрасывается. После получения необходимого числа байт (или символа конца посылки) проверяется правильность полученных данных и производится их обработка.

Примером бинарного протокола может служить протокол MODBUS-RTU *(см. разд. 4.1.2)*, а реализацией своего бинарного протокола мы займемся в *гл. 14*.

2.4. Предотвращение потери данных

Причины потери данных при передаче или приеме можно разделить на две категории: аппаратные и программные.

К *аппаратным причинам потери данных* относятся помехи на линии, некачественный проводник, несоответствие длины провода и скорости передачи и т. д. Бороться с такими причинами, конечно, надо, но не на программном уровне. Правила прокладки сетей и расчет максимальной длины проводника можно найти в соответствующей литературе, например, см. в [2] и [9]. От программной составляющей в этом случае требуется только определить, что данные были переданы с ошибкой. Способы обнаружения ошибок мы рассмотрим дальше.

Потеря данных может возникать и при неправильной организации программы. Последовательная передача подразумевает, что все биты, посланные с одной стороны, будут приняты с другой. Более того, во время передачи и приема байта нельзя допускать никаких задержек: если байт не считан вовремя, он теряется и его место занимает следующий переданный байт. Аналогично, если хотя бы часть пакета данных будет потеряна, это, чаще всего, означает потерю всего пакета данных.

2.4.1. Прерывания и потоки

Основная причина потери данных — программа занята обработкой других данных и не опрашивает порт. Возможно, в этот момент идет обработка предыдущего полученного пакета данных или программа занята диалогом с

пользователем. Все эти причины сводятся к одной — неправильной организации программы.

Рассмотрим, например, алгоритм, приведенный в листинге 2.1.

Листинг 2.1. Неправильная организация опроса порта

```
Begin
 While (True) do begin
  { Опрос клавиатуры }
  If KeyPressed then
   Case ReadKey of
    #27: Break; {выход}
    'q': Begin
      Write('Введите K'); ReadLn(K);
    End;
   End;
  End;

  { Опрос порта }
  DoReadPort;

  { Есть новые данные! }
  If DataAvailable then begin
   WriteLn('Значение температуры:', Code_ADC * K);
   DataAvailable:= False;
  End;

 End;
End.
```

При всей заманчивой простоте такого кода так делать не надо! Все будет хорошо, пока пользователь не нажмет клавишу <q>, вызывающую диалог ввода нового коэффициента. А вот во время диалога порт опрашиваться не будет. В программе для Windows можно сделать аналогичную ошибку, вызвав метод `DoModal` формы диалога параметров.

Конечно, в Windows можно воспользоваться компонентом `TTimer` и делать опрос порта в обработчике `OnTimer` с некоторой фиксированной частотой. Однако и этот способ далек от совершенства. Дело в том, что, во-первых, при большой скорости передачи данных буфер приема драйвера порта может переполниться, и часть данных будет потеряна. А, во-вторых, таймер

работает с помощью посылки сообщения WM_TIMER окну программы. Это сообщение имеет низший приоритет, и вероятность того, что считывание порта будет произведено вовремя, достаточно мала. В Windows 95/98 достаточно, щелкнув на заголовке окна, удерживать кнопку "мыши" в нажатом состоянии — и сообщения WM_TIMER не будут обрабатываться. Таким образом, если пользователь будет перетаскивать окно на другое место, порт опрашиваться не будет. И еще один недостаток: несмотря на то, что редактор свойств позволяет установить свойство Interval (частоту вызова обработчика таймера) в любое значение, большее нуля, реально работающим минимальным значением будет 70—100 мс.

Раскритиковав все способы работы, стоит предложить какой-нибудь правильный. Для DOS таким способом является обработка прерываний, рассматриваемая в гл. 8. А для Windows правильным способом обработки порта является создание отдельного потока для чтения данных, как будет показано в *гл. 11* и *12*.

2.4.2. Буферизация

При большой скорости передачи данных, очевидно, может возникнуть ситуация, когда время на обработку полученного пакета данных (например, рисование графика, анализ данных и сохранение их в файл) больше, чем время между получением пакетов.

В таких случаях необходимо некоторое промежуточное хранилище полученных данных — буфер.

Буферизация позволяет основному модулю не заботиться о скорости обработки данных. Конечно, размер буфера тоже ограничен, но обычно его выбирают достаточно большим.

На уровне приема одного байта существуют свои буферы приема:

- в DOS можно включить специальный режим буферизации FIFO;
- Windows имеет свои буферы, размер которых можно изменять вызовом функции SetupComm.

Однако чаще всего программе важнее иметь свой буфер, хранящий полные пакеты данных, а системные буферы использовать как дополнительный метод повышения надежности *(см. гл. 14)*.

Единственная сложность буферизации — необходимость синхронизации процедуры выборки данных из буфера и процедуры добавления в буфер. Синхронизация необходима, т. к. в DOS добавление в буфер будет производиться из прерывания, а в Windows — из другого потока. В обоих случаях вероятность того, что основная программа будет прервана на обновление буфера в середине процедуры выборки данных, достаточно высока. Очевидно, в таком случае возникнет ошибка.

Синхронизация в DOS

В DOS синхронизация возможна только одним способом — запрещением прерываний на время выборки данных из порта, как показано в листинге 2.2.

Листинг 2.2. Синхронизация в DOS

```
{Обработчик прерывания}
Procedure OnCOMInterrupt(B : Byte);
Begin
 { Добавить байт в буфер }
 AddByteToBuffer(B);
End;

Var
 B : Byte;
Begin
 {Основной цикл программы}
 While (True) do begin
  {Запретить прерывание}
  DisableIRQ;
  {Забрать байт, если он есть}
  If GetSizeBuffer > 0 then B:= GetByteFromBuffer;
  {Разрешить прерывания}
  EnableIRQ;

... обработка нового байта B ...

 End;
End.
```

Обратите внимание, что обработка байта производится вне блока [`DisableIRQ`, `EnableIRQ`]. Таким образом, запрещение прерываний происходит на очень маленький промежуток времени выборки данных из буфера.

Синхронизация в Windows

В Windows существуют свои методы синхронизации. Для синхронизации двух потоков нужно создать специальный объект синхронизации, например, `TCriticalSection`. Скелет такой программы приводится в листинге 2.3.

Листинг 2.3. Пример синхронизации в Windows

```
Uses SyncObjs; {нужно для использование объектов синхронизации}
Var
 CS : TCriticalSection;

{Создание объекта синхронизации}
procedure TForm1.FormCreate(Sender: TObject);
begin
 CS:= TCriticalSection.Create;
end;

{Уничтожение объекта синхронизации}
procedure TForm1.FormDestroy(Sender: TObject);
begin
 CS.Free;
end;

{Добавление байта в буфер}
procedure TThread1.DoAddToBuffer(:);
begin
 CS. Acquire;
 Try
   :добавить в буфер:
 Finally
  CS. Release;
 End;
end;

{Выборка байта из буфера}
procedure TThread2.DoGetByteFromBuffer(:);
begin
 CS. Acquire;
 Try
   :забрать из буфера:
 Finally
  CS. Release;
 End;
end;
```

Подробнее про объекты синхронизации можно прочитать в [11].

2.4.3. Обратная связь

Одной из причин потери данных является занятость приемника, в то время как передатчик продолжает передавать данные. Избежать переполнения можно, организовав обмен таким образом, чтобы приемник мог сообщать в своей занятости передатчику.

Конечно, обратная связь требует уже полного нуль-модемного провода, содержащего все необходимые линии. Система обратной связи может быть либо аппаратной, либо программной.

Аппаратная обратная связь заключается в выделении некоторой линии (например, DTR/DSR) для уведомления передатчика о возможностях приемника. В этом случае передатчику разрешается передавать данные, только пока DSR находится в состоянии ON. Как только приемник заполнил свой буфер, он сбрасывает DTR в OFF, уведомляя передатчик о необходимости прервать передачу данных. Windows позволяет включать такой режим с помощью флага `fOutxCtsFlow` (управление с помощью CTS) или флага `fOutxDsrFlow` (управление с помощью DSR).

Программная обратная связь осуществляется посылкой специальных символов. При получении символа `Xon` передатчик начинает передачу данных, а при получении символа `Xoff` останавливает передачу. В Windows существует режим автоматического управления посылкой таких символов *(см. гл. 20)*.

2.5. Методы обнаружения ошибок передачи

Протокол RS-232 имеет существенный недостаток — плохую помехозащищенность. Если для символьного протокола некоторой защитой служит ограничение диапазона самих символов (например, в посылке числовых данных могут встречаться только символы "0…9", ".","−" и, может быть, буквы), то для бинарного протокола необходимы дополнительные меры контроля данных.

Конечно, аппаратный контроль четности позволяет в некоторой степени контролировать достоверность данных, но, как мы покажем в *разд. 10.5*, надежность такого способа далека от совершенства.

Способы борьбы с ошибками передачи можно разделить на две категории: не использующие обратную связь и использующие ее.

В первом случае на передающей стороне передаваемые данные кодируются одним из известных кодов с исправлением ошибок. На приемной стороне, соответственно, производится декодирование принимаемой информации и исправление обнаруженных ошибок. Исправляющая возможность применяемого кода зависит от числа избыточных битов, генерируемых кодером.

Если вносимая избыточность невелика, то существует опасность, что принимаемые данные будут содержать необнаруженные ошибки, которые могут привести к ошибкам в работе прикладного процесса. Если же использовать код с высокой исправляющей способностью (большой избыточностью), то это приводит к необоснованно низкой реальной скорости передачи данных.

В системах с обратной связью применяются процедуры обнаружения ошибок и переспроса, также называемые *"обнаружением ошибок с автоматическим запросом повторения"* (ARQ, Automatic Repeat Request). В этом случае к данным добавляется контрольная информация, достаточная только для вычисления достоверности данных. Каждый принятый блок данных *(кадр)* должен быть подтвержден специальным кадром, посылаемым в обратном направлении. Такой кадр содержит уведомление о правильности данных или запрос на повторение кадра. Таким образом, достигается оптимальное сочетание: передаваемый код не содержит избыточных данных, а принимающая сторона может управлять коррекцией ошибок. При этом ARQ также может быть старт-стопного типа (SAW), с возвратом на N шагов (GBN) или селективного повторения (SR).

Для обнаружения ошибок в данных могут применяться несколько методов:

- посимвольный контроль четности (реализуется аппаратно в RS-232 с помощью контроля четности);
- поблочный контроль четности (например, в протоколах ARQ);
- добавление избыточного контрольного кода (CRC);
- увеличение вероятности корректного распознавания начала посылки;
- естественные методы контроля данных.

2.5.1. Контроль четности

Порт RS-232 реализует аппаратный контроль четности (или как его еще называют, *контроль паритета*). Пользователь может конфигурировать порт на проверку четного или нечетного паритета или отсутствия контроля четности.

При включенном контроле четности к передаваемому байту добавляется еще один бит, дополняющий биты основного байта строго определенным образом. Например, пусть производится передача байта <1100 0101>. Общее количество единиц равно четырем. Если используется четный паритет, то бит паритета будет равен 0, и общее количество единиц в этом байте будет по-прежнему четным числом. Если же используется нечетный паритет, то бит паритета будет равен 1. Общее количество единиц в этом случае, вместе с битом паритета, будет равно пяти, т. е. нечетному числу.

Принимающее устройство производит те же вычисления и сравнивает число единиц в полученном байте. При несовпадении генерируется ошибка четно-

сти. Однако, очевидно, что несовпадение четности произойдет только в случае искажения одного бита. Интересные эксперименты с контролем четности мы произведем в *разд. 10.5*.

При ошибке четности порт может генерировать аппаратное прерывание. Мы рассмотрим использование прерываний в *гл. 8*.

2.5.2. Контрольная сумма

Один из способов повышения достоверности получаемых данных заключается в дополнении посылки контрольной суммой. Контрольная сумма — обычно один, два или четыре байта (в случае необходимости может быть и больше), полученные некоторым преобразованием данных посылки, подсчитываемые перед отправкой и включаемые в посылку. Принимающая программа производит аналогичную операцию над данными и сверяет вычисленную и полученную контрольные суммы. Если помеха изменила один или несколько байт посылки, вероятность совпадения контрольных сумм очень мала.

В этой главе мы кратко опишем наиболее распространенные методы вычисления контрольных сумм, а в *гл. 15* займемся реализацией.

Простая контрольная сумма

Самая *простая контрольная сумма* может быть вычислена как исключающее ИЛИ всех информационных байт посылки.

Для вычисления исключающего ИЛИ в языке C (C++) используется оператор ^, а в языке Pascal/Delphi — оператор `XOR`.

Несмотря на примитивность такого метода, вероятность того, что помеха исказит посылку таким образом, что наложение байт останется неизменным, достаточно мала. А очевидным преимуществом является простота реализации.

Контрольная сумма LRC

Контрольная сумма LRC (Longitudinal Redundancy Check) это один байт, который вычисляется передающим устройством и добавляется к концу сообщения. Принимающее устройство также вычисляет LRC в процессе приема и сравнивает вычисленную величину с полем контрольной суммы пришедшего сообщения. Если суммы не совпали, значит, имеет место ошибка.

LRC вычисляется сложением последовательности байтов сообщения, отбрасывая все переносы, и затем двойным дополнением результата, т. е. это 8-битовое поле, где каждое новое прибавление символа, приводящее к результату более чем 255, вызывает простое перескакивание через 0. Так как это поле не является 9-битовым, перенос отбрасывается автоматически.

Язык C не проверяет выход за границы типа. В Pascal (и, естественно, Delphi тоже) это сложнее, т. к. при переполнении вызывается исключение, и программа останавливается с ошибкой "`Run-time error 201: Range check error`". Чтобы этого избежать, следует отключать контроль переполнения при вычислении или программно исключать возможность переполнения.

Таким образом, алгоритм генерации LRC выглядит так:

1. Сложить все байты сообщения, исключая стартовые и конечные символы, складывая их так, чтобы перенос отбрасывался.
2. Отнять получившееся значение от числа $FF — это является первым дополнением.
3. Прибавить к получившемуся значению 1 — это второе дополнение.

Реализацией этого алгоритма мы займемся в *гл. 15*.

Контрольная сумма CRC16

Контрольная сумма CRC16 (Cyclical Redundancy Check) это 16-разрядная величина, т. е. два байта. CRC16 вычисляется передающим устройством и добавляется к сообщению. Принимающее устройство также вычисляет CRC16 в процессе приема и сравнивает вычисленную величину с полем контрольной суммы пришедшего сообщения. Если суммы не совпали, значит, имеет место ошибка.

В 16-битовый регистр CRC предварительно загружается число $FFFF. Процесс начинается с добавления байтов сообщения к текущему содержимому регистра. Для генерации CRC используются только 8 бит данных. Старт- и стоп-биты, бит паритета, если он используется, не учитываются в CRC.

В процессе генерации CRC каждый 8-битовый символ складывается по исключающему ИЛИ (`XOR`) с содержимым регистра. Результат сдвигается в направлении младшего бита, с заполнением 0 старшего бита. Младший бит извлекается и проверяется. Если младший бит равен 1, то содержимое регистра складывается с определенной ранее фиксированной величиной по исключающему ИЛИ. Если младший бит равен 0, то исключающее ИЛИ не делается.

Этот процесс повторяется, пока не будет сделано восемь сдвигов. После последнего (восьмого) сдвига, следующий байт складывается с содержимым регистра, и процесс повторяется снова. Финальное содержание регистра после обработки всех байт сообщения и есть контрольная сумма CRC.

Таким образом, алгоритм генерации CRC выглядит так:

1. 16-битовый регистр загружается числом $FFFF и используется далее как регистр CRC.

2. Первый байт сообщения складывается по исключающему ИЛИ с содержимым регистра CRC. Результат помещается в регистр CRC.
3. Регистр CRC сдвигается вправо (в направлении младшего бита) на один бит, старший бит заполняется 0.
4. (Если младший бит 0): Повторяется шаг 3 (сдвиг).
5. (Если младший бит 1): Делается операция исключающее ИЛИ регистра CRC и полиномиального числа $A001.
6. Шаги 3 и 4 повторяются восемь раз.
7. Повторяются шаги 2—5 для следующего сообщения. Это повторяется до тех пор, пока все байты сообщения не будут обработаны.
8. Финальное содержание регистра CRC и есть контрольная сумма.

Реализацией этого алгоритма мы займемся в *гл. 15*.

2.5.3. Стартовый байт

Для четкого распознавания начала посылки желательно, чтобы стартовый байт (или стартовая последовательность) не встречался в самих данных. Для обеспечения уникальности стартового байта (или, точнее говоря, стартовой посылки) применяют один из способов:

- ❒ программное исключение стартовых символов из данных;
- ❒ аппаратное дублирование битов;
- ❒ программное дублирование байт.

Первый способ заключается в простом анализе данных. Так, если данные состоят из цифр и букв, то логично в качестве стартовой посылки применить, например, символ "$" или последовательность символов "!!!". Появление такой последовательности внутри самих данных исключено. Однако такое исключение можно произвести только в строковых (например, ASCII) протоколах передачи.

Второй способ заключается в принудительном исключении символа начала посылки. Для синхронизации кадров используется комбинация типа "флаг" (двоичный код <01111110>). Началом передаваемого блока данных (кадра) является первый байт после флага, отличный от него. После передачи последнего информационного байта сразу передаются флаги. Для обеспечения прозрачности кода передаваемой информации используется процедура *"бит-стаффингования"*: принудительное добавление нулевого бита после следующих подряд пяти единиц. Таким образом, появление шести единиц подряд (как в коде флага) исключено. Такие алгоритмы реализуются аппаратно, например, в протоколах HDLC (High-level Data Link Control, Высокоуровневый протокол управления каналом передачи данных).

И, наконец, программное обеспечение уникальности стартовой посылки производится на уровне байт или последовательности байт. Один из способов заключается в использовании двух байт и дублировании первого, если он попадается в данных. Например, стартовой последовательностью выбирается последовательность "$10, $01". При этом оговаривается, что если "$10" встретится внутри данных (или контрольной сумме), то этот байт будет дублирован, т. е. будет передано "$10, $10". Таким образом гарантируется, что стартовая последовательность будет уникальна, т. к. такая же последовательность в данных будет передана как "$10, $10, $01".

Другой байтовый способ реализуется, например, в протоколе SLIP (Serial Line IP). В этом случае пакет данных (кадр) должен завершаться символом "$C0" ("конец"). Во многих реализациях кадр и начинается с этого символа, что позволяет принимающей стороне эффективно отбросить все принятое до передачи кадра. Если какой-либо байт кадра равен символу "конец", то вместо него передается 2-байтовая последовательность "$DB, $DC". Если же байт пакета равен "$DB", то вместо него передается последовательность "$DB, $DD".

2.5.4. Другие способы повышения достоверности данных

Кроме контрольной суммы, обеспечивающей, в общем-то, самый надежный способ контроля, можно применять и естественные способы. Например, если передаваемое число представляет собой значение кода АЦП, веса или давления, то логично проверять такие числа на попадание в естественный для них диапазон: код АЦП от 0 до 65535, вес от 0 до максимальной нагрузки, давление — не меньше нуля и т. д.

Еще один способ заключается в увеличении активной паузы, т. е. после включения приемопередатчика в режим передачи следует выдержать паузу в 1,5—2 кадра. При активном передатчике помехе трудно будет "прорваться" к приемникам, а, следовательно, все абоненты зафиксируют тайм-аут приема, сбросят буферы и настроятся на прием новой посылки. Такой способ применяется, например, в протоколе MODBUS *(см. гл. 4)*.

Глава 3

Последовательные протоколы IBM PC

Мышь — это не роскошь, а средство передвижения курсора.

В этой главе будут описаны протоколы, имеющие непосредственное отношение к персональному компьютеру. Мы не будем затрагивать вопросы, связанные с аппаратной реализацией интерфейсов, а рассмотрим только лишь сами протоколы.

3.1. Мышь

Многие еще помнят, что в старых компьютерах манипулятор "мышь" подключался не к PS/2, а к обычному COM-порту. Впрочем, метод подключения ничего не меняет — мышь была и остается устройством, использующим последовательный интерфейс RS-232.

Наиболее распространены два протокола обмена данными с мышью: Microsoft Mouse и PC Mouse. Первый протокол представляет собой группу форматов, основанную на Microsoft-протоколе. В настоящее время именно эта группа стала основной для координационных устройств, подключаемых к последовательному порту.

3.1.1. Базовый протокол Microsoft Mouse

В случае использования базового протокола Microsoft Mouse порт программируется на 1200 бит/с, 7 бит данных без проверки четности и 1 стоп-бит. Протокол обеспечивает передачу информации о двух кнопках. По каждому событию выдается информация из трех байт в весьма запутанном формате, показанном в листинге 3.1.

Листинг 3.1. Протокол Microsoft Mouse

```
1 байт: — 1 LB RB Y7 Y6 X7 X6
2 байт: — 0 X5 X4 X3 X2 X1 X0
3 байт: — 0 Y5 Y4 Y3 Y2 Y1 Y0
```

LB и RB передают состояние левой и правой кнопок соответственно (причем 1 означает "кнопка нажата"), а биты X, Y — относительное перемещение мыши со времени последней посылки. Таким образом, первый байт отличается от остальных установленным в 1 шестым битом. Он служит для синхронизации начала посылки.

Положительное значение приращения по координате X соответствует перемещению указателя вправо, а по координате Y — вниз.

3.1.2. Протокол Microsoft Plus

Для обеспечения нормальной работы с трехкнопочными устройствами протокол Microsoft Mouse пришлось дополнить четвертым байтом, который служит единственной цели — обеспечению передачи состояния средней кнопки мыши (в пятом разряде, обозначенном символом "M"). Следует учитывать, что передача пакета из четырех байт выполняется только в случае изменения состояния средней кнопки, а в остальных случаях передаются первые три байта. Этот протокол получил название Microsoft Plus (листинг 3.2).

Листинг 3.2. Протокол Microsoft Plus

```
1 байт: — 1 LB RB Y7 Y6 X7 X6
2 байт: — 0 X5 X4 X3 X2 X1 X0
3 байт: — 0 Y5 Y4 Y3 Y2 Y1 Y0
4 байт: — 0  M  0  0  0  0  0
```

3.1.3. Протокол 3D Serial Mouse

Появление на рынке более сложных устройств, снабженных дополнительными возможностями, естественно, влечет за собой появление новых протоколов. Например, формат данных трехкоординатной мыши (3D Mouse) включает координату Z (Z0-Z3, целое число со знаком) и состояние специальной кнопки R0 (листинг 3.3).

Листинг 3.3. Протокол 3D Serial Mouse

```
1 байт: — 1 LB RB Y7 Y6 X7 X6
2 байт: — 0 X5 X4 X3 X2 X1 X0
3 байт: — 0 Y5 Y4 Y3 Y2 Y1 Y0
4 байт: — 0  0 R0 Z3 Z2 Z1 Z0
```

3.1.4. Протокол PC Mouse

Протокол PC Mouse обеспечивает передачу информации о трех кнопках и программирует порт на 1200 бит/с, 8 бит данных без контроля четности и 1 стоп-бит. Согласно этому протоколу, по каждому событию выдается информация из 5 байт (листинг 3.4).

Листинг 3.4. Протокол PC Mouse

```
1 байт: 1  0  0  0  0  LB MB RB
2 байт: X7 X6 X5 X4 X3 X2 X1 X0
3 байт: Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
4 байт: X7 X6 X5 X4 X3 X2 X1 X0
5 байт: Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
```

Здесь `LB`, `MB`, `RB` — состояния, соответственно, левой, средней и правой кнопок, причем 0 означает, что кнопка нажата. Дублирование байтов используется для идентификации первого байта посылки.

3.1.5. Совместимость протоколов

Стоит отметить, что протокол для мышей, подключаемых к порту PS/2, отличается (причем даже аппаратно!) от протокола COM-мышей. В манипуляторах "мышь", работающих с обоими протоколами, просто реализованы два интерфейса одновременно. Подключить мышь, работающую только с RS-232, к PS/2 с помощью обычного переходника не получится.

3.2. Модем

Как известно, данные в компьютере представлены в цифровой форме — закодированы в виде нулей и единиц, которым физически соответствует низкий или высокий уровень напряжения. Телефонная же сеть рассчитана на передачу речевых сообщений, представляемых в форме аналоговых электрических сигналов, поэтому непосредственная передача цифровой информации через телефонную сеть невозможна. Для преобразования форм пред-

ставления информации необходимо некоторое устройство, включаемое между компьютером и телефонной линией. Такое устройство называют *модемом* (сокращение от МОдулятор-ДЕМодулятор).

В нашей книге рассматриваются основные команды для работы с модемами и некоторые протоколы, используемые для передачи файлов. Всю аппаратную начинку мы оставим за кадром.

Основными функциями протоколов передачи файлов являются:

- обеспечение безошибочной передачи данных;
- управление потоком передаваемых данных;
- передача вспомогательной информации;
- защита соединения.

Для реализации этих функций данные разбиваются на кадры определенной длины, и в каждый кадр включается контрольная информация *(см. гл. 2)* для обнаружения ошибок. На приемной стороне производится аналогичное вычисление, и полученное значение контрольной информации сравнивается с принятым. При совпадении принимающая сторона посылает *подтверждение* правильного приема блока (ACK, ACKnowledge), а при несовпадении — *запрос на повторную передачу данного блока* (NACK, No ACKnowledge, неподтверждение).

Мы опишем наиболее распространенные протоколы передачи файлов. Все они рассчитаны на отсутствие аппаратной защиты от сбоев, и именно поэтому наиболее интересны для рассмотрения. Эта информация может оказаться полезной при составлении своего протокола обмена.

3.2.1. Команды управления модемом

Для управления модемом используются команды, представляющие собой последовательность определенных символов, воспринимаемых модемом как команды. Такие последовательности называются AT-командами. "AT" (ATtention, внимание) — это префикс, который ставится перед одной или несколькими командами.

Команды можно набирать из окна терминала, предварительно установив соединение с портом, к которому подключен модем. Готовые терминальные программы можно или найти в Интернете, или воспользоваться стандартным терминалом, входящим в поставку Windows.

Команды могут быть набраны прописными или строчными буквами и должны содержать численные параметры, если этого требует формат команды. Если численный параметр пропущен, его значение по умолчанию принимается равным нулю. При успешном выполнении команды модем посылает сообщение "Ok".

Существуют две команды, не требующие префикса "AT". Это команда `A/`, означающая повтор предыдущей команды, и команда `+++`, называемая *Escape-последовательностью*. Escape-последовательность используется для переключения модема в командный режим из режима передачи данных.

Остальные команды являются AT-командами и требуют префикса "AT". Многие команды требуют дополнительного префикса, следующего за префиксом AT. В качестве таких префиксов выступают знаки "&" (логическое И), "\" (слеш), "*" (знак умножения), "%" (знак процента), "#" (решетка), "+" (знак плюс).

Набор AT-команд конкретного модема может отличаться от набора команд других производителей, однако существует список команд, поддерживаемый всеми модемами. Список таких команд приводится в справочной части книги, в *гл. 23*.

3.2.2. Регистры модемов

Список регистров зависит от конкретной модели модема, однако, как и с AT-командами, существует набор регистров, реализуемый во всех моделях модемов.

Регистр представляет собой ячейку в энергонезависимой памяти модема. Ячейки нумеруются с нуля. Запись в регистр производится с помощью AT-команды `ATSr = n`. Например, для установки значения 3 в нулевой регистр нужно выполнить команду `ATS0 = 3`. Чтение регистра производится с помощью AT-команды `ATSr?`. Например, для получения содержимого регистра 7 нужно выполнить команду `ATS7?`.

Список основных регистров модема приводится в справочной части нашей книги, в *гл. 22*.

3.2.3. Протоколы передачи файлов

Конечно, чтобы рассмотреть все появившиеся на сегодняшний день протоколы, потребуется книга в несколько раз больше нашей. Мы же рассмотрим наиболее используемые и интересные с точки зрения реализации протоколы.

ASCII-протокол

Этот протокол работает без коррекции ошибок. В результате, при передаче файлов по телефонным каналам из-за шума принятый файл сильно отличается от передаваемого. Если вы передаете короткие текстовые сообщения, ошибки могут быть легко исправлены. Если же вы передаете выполняемый файл, то ошибки при передаче могут стать роковыми, а полученная программа не будет работать.

Протокол XModem

Протокол XModem, разработанный Вардом Христенсеном (Ward Christensen) в 1977 году, благодаря широкому использованию в справочных службах и введению в недорогие коммуникационные программы для PC, стал фактическим стандартом для связи между персональными компьютерами.

Последовательность операций, выполняемых приемником и передатчиком, следующая. Передающий компьютер начинает передачу файла только после приема от принимающего компьютера знака NAK[1] (Negative AcKnowledge, неподтверждение или отрицательное подтверждение), представляющего собой последовательность <0010101> в кодировке ASCII. Принимающий компьютер передает эту последовательность до тех пор, пока не начнется передача собственно файла. Если переданы девять знаков NAK, а передача файла не началась, процесс должен быть возобновлен вручную.

После приема знака NAK передающий компьютер посылает знак начала блока SOH (Start Of Header, значение $01), два номера блока (сам номер и его двоичное дополнение по "единицам"), блок данных из 128 байт и контрольную сумму блока CS (Check Sum). Блоки нумеруются по модулю 256. Контрольная сумма размером в 1 байт представляет собой остаток от деления на 255 суммы значений кодов ASCII знаков, входящих в блок данных.

Принимающий компьютер тоже вычисляет контрольную сумму и сравнивает ее с принятой. Если сравниваемые значения различны, либо прошло 10 секунд, а прием блока не завершен, принимающий компьютер посылает передатчику знак NAK, означающий запрос на повторную передачу последнего блока. Если блок принят правильно, приемник передает подтверждение его приема знаком ACK (код $06). В случае если следующий блок не поступил в течение 10 секунд, то передача знака ACK повторяется до тех пор, пока блок не будет принят правильно. После девяти неудачных попыток передачи блока связь прерывается.

В протоколе используется двукратная передача номера. Это исключает повторную передачу одного и того же блока из-за потери подтверждающего сообщения. Принимающий компьютер контролирует уникальность номеров принимаемых блоков. Если блок ошибочно передан повторно, то он сбрасывается. После успешной передачи всех данных передающий компьютер посылает знак завершения передачи EOT (End Of Transmission, код $04), сообщающий об окончании передачи файла.

Перерыв в передаче блока свыше одной секунды считается перерывом связи.

---

[1] Следует различать NAK — символ, который посылается принимающей стороной для обозначения готовности к приему, и NACK, который обозначает запрос на повторную передачу ошибочного блока.

Преимущества данного протокола перед другими заключаются в его доступности для разработчиков программных средств, простоте реализации на языках высокого уровня, малом объеме приемного буфера (256 байт) и возможности передачи не только символьных (в кодах ASCII), но и бинарных файлов. Последнее возможно благодаря тому, что конец файла определяется подсчетом переданных байтов и использованием вместо знака файлового маркера (<Ctrl>+<Z>) специального сигнала завершения.

К основным недостаткам протокола XModem можно отнести: низкую производительность, обусловленную в основном использованием механизма ARQ с подтверждением; большую вероятность необнаруженных ошибок; необходимость задания имени файла при приеме и относительно большой объем передаваемой служебной информации.

Последующие модификации протокола XModem были направлены на устранение этих и некоторых других его недостатков.

Протокол XModem-CRC

Более защищенным от ошибок является протокол XModem-CRC (Cyclic Redundancy Check, циклический избыточностный контроль). Xmodem-CRC — протокол с проверкой циклическим избыточным кодом. В нем 8-битовая контрольная сумма заменена 16-битовым циклическим избыточным кодом. Этот протокол гарантирует вероятность обнаружения ошибок, равную 99,9984%, т. е. только один из 700 биллионов "плохих" пакетов будет иметь правильный CRC-код. Протокол XModem-CRC также передает данные пакетами по 128 байт.

Протокол XModem-1K

Протокол XModem-1K представляет собой модификацию протокола XModem-CRC с блоками длиной до 1024 байт. Использование блоков длиной 1 Кбайт позволяет снизить задержки при передаче файлов по системам связи с временным уплотнением, с использованием современных модемов и в сетях с коммутацией пакетов, где длина пакета, как правило, равна величине 1024 байт либо кратна ей. Кроме того, по сравнению с обычным протоколом XModem, уменьшена относительная доля заголовков в общем объеме передаваемой информации.

Основной особенностью протокола XModem-1K является возможность изменения длины пакета. Если передача идет без ошибок, протокол XModem-1K увеличивает размер пакета со 128 до 1024 байт. При увеличении числа ошибок размер пакета снова уменьшается. Такое изменение длины пакета позволяет увеличить скорость передачи файлов.

Для передачи приемнику сообщения об увеличении длины передаваемого блока вместо знака SOH (код $01) в начале блока ставится знак STX (Start of TeXt, начало текта, код $02).

Передатчик не должен изменять длину блока до тех пор, пока не будет принят знак ACK для текущего блока. Игнорирование этого ограничения может привести к тому, что ошибки не будут обнаружены.

Протокол YModem

Протокол YModem разработал Чак Форсберг (Chuck Forsberg) в 1984—1985 годах. Протокол YModem похож на протокол XModem-1K, но имеет два отличия:

- протокол YModem может передавать или принимать за один заход несколько файлов;
- вместе с файлом передаются все его атрибуты. В результате как минимум имя файла и дата остаются неизменными.

Имя файла (возможно, с указанием пути) передается как строка кодов ASCII, завершаемая нуль-символом. Этот формат имени файла используется в функциях, ориентированных на операционные системы типа MS-DOS (например, в функции `fopen` библиотеки языка C). В имя файла не включаются пробелы. Имя диска источника (например, "A:", "B:" и т. д.) не передается. Если передатчик не поддерживает передачу знаков в обоих регистрах, имя передается в строчном регистре. Если в имя файла включен каталог, его название должно ограничиваться знаками "/".

Передача длины файла и каждого последующего поля произвольна. Длина файла представляется в блоке как десятичная строка, обозначающая количество байт в файле. Если передаваемый файл увеличивается во время передачи, то параметр "Длина файла" должен иметь значение, соответствующее максимально ожидаемому размеру или не передаваться вовсе.

Дата модификации файла является необязательным параметром. Дата модификации представляется в виде восьмеричного числа, указывающего время последнего изменения файла, и измеряется в секундах по Гринвичу, начиная с 1 января 1970 года. Дата 0 обозначает, что дата модификации неизвестна, и должна быть оставлена дата приема файла. Этот формат используется для исключения неопределенности при передачах файлов между различными временными зонами.

Если блок имени файла принят с ошибкой, необходим запрос на повторную передачу. Прием блока с именем файла, успешно открытого для записи, подтверждается знаком ACK. Если файл не может быть открыт для записи, то приемник прерывает передачу с помощью знака CAN.

Далее приемник инициирует передачу содержимого файлов в соответствии с протоколом XModem-CRC. После того как содержимое файла передано, приемник запрашивает имя следующего файла.

Передача нулевого имени файла может означать, во-первых, что групповая передача завершена и, во-вторых, что запрошенные у передатчика файлы не могут быть открыты для чтения.

Протокол YModem-G

Вариант G протокола YModem обеспечивает высокую эффективность передачи данных. Он используется приемником, который инициирует групповую передачу путем посылки знака "g" вместо "c". Передатчик, распознавший этот знак, прекращает ожидание обычных подтверждений по каждому переданному блоку и передает последовательные блоки на полной скорости с использованием метода управления потоком, такого как XON/XOFF.

Прежде чем передавать файл, передатчик ожидает поступления последовательности знаков "g", а в конце передачи каждого файла — подтверждающего знака ACK. Такой способ синхронизации позволяет приемнику открывать и закрывать файлы в нужное время.

Протокол YModem-G предназначен для использования с модемами, автоматически осуществляющими коррекцию ошибок на аппаратном уровне, и, соответственно, в нем упрощена защита от ошибок, т. к. ее выполняет сам модем. Не следует использовать этот протокол, если ваш модем не поддерживает аппаратной коррекции ошибок.

Протокол ZModem

Протокол ZModem для общего применения был разработан в 1986 году компанией Telnet и введен в большинство коммуникационных программ. В настоящее время этот протокол получил наиболее широкое распространение.

Представляя собой развитие протоколов XModem и YModem, ZModem устраняет их недостатки и, при соблюдении совместимости, имеет ряд преимуществ:

- высокое быстродействие;
- динамическая адаптация к качеству канала связи посредством изменения в широких пределах размера передаваемых блоков;
- возможность возобновления прерванной передачи файла с того места, на котором произошел сбой;
- повышенная достоверность передачи благодаря использованию 32-разрядной проверочной контрольной суммы (CRC32);
- возможность отключения функции контроля ошибок передаваемых блоков при использовании модемов с аппаратной коррекцией ошибок.

Протокол ZModem, так же как и протокол XModem-1K, может изменять длину пакета (блока) от 64 до 1024 байт в зависимости от качества линии.

Кроме того, протокол обладает следующей полезной особенностью: если при передаче файла произошел сбой на линии, а вы не успели передать весь файл, то в следующий раз при передаче этого же файла он автоматически начнет передаваться с того же места, где произошел обрыв связи. Таким образом, очень большие файлы вы можете передавать по частям.

Протокол BiModem

Особенностью протокола BiModem является возможность одновременной передачи двух файлов в разных направлениях. Кроме того, одновременно с передачей файлов вы можете побеседовать с оператором удаленного компьютера при помощи клавиатуры.

Протокол Kermit

Этот протокол был разработан в Колумбийском университете в 1981 году для связи между различными типами компьютеров, включая большие компьютеры, мини-компьютеры и персональные компьютеры. В отличие от протоколов XModem и Zmodem, он использует для передачи данных пакеты переменной длины и максимальным размером 94 байт.

Так же как и YModem, протокол Kermit может передавать или принимать несколько файлов за один сеанс, и, в дополнение, Kermit использует предварительную компрессию данных для увеличения эффективной скорости обмена данными.

Глава 4

Промышленные последовательные протоколы

Цель работы программиста — намагнитить быстро вращающиеся металлические пластинки в правильных местах.

В этой главе будут кратко описаны современные промышленные протоколы последовательной связи. Эта информация будет интересна не только программистам, занимающимся промышленной автоматизацией, но и всем тем, кто занимается построением собственных протоколов обмена. Мы не будем подробно рассматривать аппаратные реализации интерфейсов, а акцентируем внимание на принципах построения протоколов.

4.1. Протокол MODBUS

Протокол MODBUS является международным стандартом, который поддерживают многие фирмы — производители контроллеров технологического оборудования. Протокол предполагает одно активное (запрашивающее) устройство в линии *(мастер)* и позволяет опрашивать устройства, обращаясь к ним по уникальному адресу. Синтаксис команд протокола позволяет адресовать 254 устройства, соединенные в линию. Обычно подразумевается, что физический уровень линии должен соответствовать стандарту RS-422 (четырехпроводный, дуплексный) или RS-485 (двухпроводный, полудуплексный, с захватом линии), однако при соединении "точка-точка" тот же формат команд может быть использован на любом последовательном асинхронном физическом интерфейсе, в том числе RS-232.

Контроллеры соединяются, используя технологию "главный-подчиненный", при которой только одно устройство (master, главный) может инициировать передачу (сделать запрос). Другие устройства (slaves, подчиненные) передают запрашиваемые главным устройством данные или производят запрашиваемые действия. Типичное главное устройство включает в себя ведущий (host) процессор и панели программирования. Типичное подчиненное уст-

ройство — программируемый контроллер. Контроллеры могут быть соединены напрямую или через модем.

Главное устройство может адресоваться к индивидуальному подчиненному или может инициировать широкую передачу сообщения на все подчиненные устройства. Подчиненное устройство возвращает сообщение в ответ на запрос, адресуемый именно ему. Ответы не возвращаются при широковещательном запросе от главного.

Форматы запроса и ответа имеют вид:

- адрес устройства;
- код функции;
- 8-битные байты данных;
- контрольная сумма.

Код функции в запросе говорит подчиненному устройству, какое действие необходимо произвести. Байты данных содержат информацию, необходимую для выполнения запрошенной функции. Например, код функции 3 подразумевает запрос на чтение содержимого регистров подчиненного. Если подчиненный дает нормальный ответ, код функции в ответе повторяет код функции в запросе, и в байтах данных содержится затребованная информация. Если имеет место ошибка, то код функции модифицируется, и в байтах данных передается причина ошибки. В сетях MODBUS может быть использован один из двух способов передачи: ASCII или RTU (Remote Terminal Units, удаленные терминальные устройства). Пользователь выбирает необходимый режим вместе с другими параметрами (скорость передачи, режим паритета и т. д.) во время конфигурирования каждого контроллера.

4.1.1. Протокол MODBUS-ASCII

При использовании ASCII-режима каждый байт сообщения передается как два ASCII-символа, а именно, две шестнадцатеричные цифры (символы 0—9, A—F). Каждое сообщение передается непрерывным потоком. Главное преимущество этого способа заключается в том, что время между передачей символов может быть до 1 секунды, без возникновения ошибок при передаче. Байты могут передаваться как с контролем четности, так и без него. В первом случае используется один стоп-бит, а во втором — два стоп-бита. Формат байта протокола MODBUS-ASCII показан в табл. 4.1.

Таблица 4.1. *Формат байта протокола MODBUS-ASCII*

С контролем четности	Без контроля четности
1 старт-бит	1 старт-бит
7 бит данных, младшим битом вперед	7 бит данных, младшим битом вперед
1 бит паритета	Нет бита паритета

Таблица 4.1 *(окончание)*

С контролем четности	Без контроля четности
1 стоп-бит	2 стоп-бита
Контрольная сумма (LCR)	Контрольная сумма (LCR)

В ASCII-режиме сообщение начинается с двоеточия (символ ":", код $3A) и заканчивается последовательностью <возврат каретки>, <перевод строки> (CRLF, код $0D, $0A). Так как сообщение состоит из шестнадцатеричных цифр, то допустимым набором символов является 0—9 и A—F.

Принимающее устройство постоянно ожидает символа ":". После того, как он принят, устройство принимает все следующие символы согласно формату сообщений (табл. 4.2). Интервалы между символами сообщения могут быть не более 1 секунды. Интервал более секунды принимающее устройство распознает как ошибку.

Таблица 4.2. *Формат посылки протокола MODBUS-ASCII*

Начало посылки	Адрес абонента	Номер функции	Данные	LCR	Конец посылки
1 символ (знак ":")	2 символа	2 символа	N символов	2 символа	2 символа (CRLF)

4.1.2. Протокол MODBUS-RTU

При использовании RTU-протокола каждый байт сообщения содержит две 4-битовые шестнадцатеричные цифры. Основное преимущество этого протокола в том, что с его помощью можно передавать значительно больший объем информации, чем за то же время с помощью ASCII-протокола. Каждое сообщение передается как непрерывный поток.

Протокол может либо использовать контроль четности (в этом случае используется один стоп-бит), либо работать без контроля четности (в этом случае используются два стоп-бита). Формат байта протокола MODBUS-RTU показан в табл. 4.3.

Таблица 4.3. *Формат байта протокола MODBUS-RTU*

С контролем четности	Без контроля четности
1 старт-бит	1 старт-бит
8 бит данных, младшим битом вперед	8 бит данных, младшим битом вперед
1 бит паритета	Нет бита паритета

Таблица 4.3 *(окончание)*

С контролем четности	Без контроля четности
1 стоп-бит	2 стоп-бита
Контрольная сумма (CRC)	Контрольная сумма (CRC)

В RTU-режиме сообщение начинается с интервала тишины, равного времени передачи 3,5 символов при данной скорости передачи в сети. Это позволяет принимающему устройству абсолютно точно понять, что это не перерыв в передаче предыдущего сообщения. Такая информация необходима приемнику, чтобы сообщение, принятое не с начала (например, при включении приемника), не было интерпретировано как начало посылки. Другими словами, это гарантирует, что первый полученный после тишины символ будет первым символом сообщения, а, кроме того, если получено сообщение оговоренной в протоколе длины, то это достоверно будет полное сообщение.

Первым полем после тишины передается адрес устройства, которому адресуется сообщение, затем следуют номер функции, данные и контрольная сумма. Вслед за последним передаваемым символом также следует интервал тишины продолжительностью не менее 3,5 символов. Новое сообщение может начинаться после этого интервала (табл. 4.4).

Таблица 4.4. *Формат посылки протокола MODBUS RTU*

Начало посылки	Адрес абонента	Номер функции	Данные	LCR	Конец посылки
3,5 символа тишины	8 бит	8 бит	N байт	16 бит	3,5 символа тишины

Фрейм сообщения передается непрерывно. Если интервал тишины продолжительностью 3,5 символа возник во время передачи фрейма, принимающее устройство завершает прием сообщения, сбрасывает принимающий буфер и следующий байт будет воспринят как начало следующего сообщения.

Таким образом, если новое сообщение начнется раньше 3,5-символьного интервала, принимающее устройство воспримет его как продолжение предыдущего сообщения. В этом случае будет установлена ошибка из-за несовпадения контрольных сумм.

4.1.3. Поля MODBUS протокола

Поле "Адрес абонента"

Адресное поле содержит два шестнадцатеричных символа (в ASCII-протоколе) или 8 бит двоичных символов (в RTU-протоколе). Допустимый ад-

рес передачи находится в диапазоне от 0 до 247. Каждому подчиненному устройству присваивается адрес в пределах от 1 до 247.

Адрес 0 используется для широковещательной передачи, его распознает каждое устройство. Когда MODBUS-протокол используется на более высоком уровне сети, широковещательная передача может не поддерживаться или может быть реализована другими методами.

Поле "Код функции"

Поле функции содержит два шестнадцатеричных символа (в ASCII-протоколе) или 8 бит двоичных символов (в RTU-протоколе). Диапазон чисел от 1 до 255. Реализация всех функций не обязательна, контроллер может обрабатывать некоторое подмножество функций. Когда главный передает данные подчиненному, он указывает номер функции, которую должен выполнить подчиненный. В случае нормального ответа подчиненный повторяет оригинальный код функции. Если имеет место ошибка, главному возвращается код функции с установленным в 1 старшим битом, а поле данных содержит описание или причину ошибки.

Например, главный передает подчиненному указание "прочитать группу регистров". Эта функция имеет код <0000 0011>, т. е. $03. Если подчиненный выполнил затребованное действие без ошибки, он возвращает такой же код. Если имеет место ошибка, то он возвращает <1000 0011>, т. е. $83.

Поле "Данные"

Поле данных в сообщении от главного к подчиненному содержит дополнительную информацию, которая необходима подчиненному для выполнения указанной функции. Оно может содержать адреса регистров или выходов, их количество, счетчик передаваемых байтов данных и т. п.

Например, если главный запрашивает у подчиненного прочитать группу регистров (код функции 0x03), поле данных содержит адрес начального регистра и количество регистров. Если главный хочет записать группу регистров (код функции 0x10), поле данных содержит адрес начального регистра, количество регистров, счетчик количества байтов данных и данные для записи в регистры.

Поле данных может отсутствовать (иметь нулевую длину) в определенных типах сообщений.

Поле "Контрольная сумма"

В MODBUS-сетях используются два метода контроля ошибок передачи. Содержание поля контрольной суммы зависит от выбранного способа передачи.

Когда используется ASCII-режим, поле контрольной суммы содержит два ASCII-символа. Контрольная сумма является результатом вычисления LRC

(см. разд. "Контрольная сумма LRC") над содержанием сообщения начиная с ":" и заканчивая CRLF (исключая сами символы начала и конца посылки).

Когда используется RTU-режим, поле контрольной суммы содержит 16-битовую величину. Контрольная сумма является результатом вычисления CRC *(см. разд. "Контрольная сумма CRC 16")* над содержанием сообщения. CRC добавляется к сообщению последним полем младшим байтом вперед.

4.2. Протокол CAN

Интерфейс CAN (Controller Area Network, локальная сеть контроллеров) предназначен для организации высоконадежных и недорогих каналов связи в распределенных системах управления. Он позволяет строить как дешевые мультиплексные каналы, так и высокоскоростные сети.

В настоящее время CAN-интерфейс широко применяется во многих областях, в том числе в промышленной автоматике и в аэрокосмическом приборостроении. CAN-интерфейс (или CAN-Bus) чаще всего используется как связующее звено между главной магистралью управляющего устройства и множеством вспомогательных датчиков, механизмов и т. п., подключение которых к центральной магистрали не всегда целесообразно.

Протокол CAN, разработанный фирмой Bosch, изначально проектировался для нужд автомобильной промышленности: такие компании как Chrysler, Ford, GM, Volvo, Audi, BMW, Renault, SAAB, Volkswagen, Toyota используют CAN для контроля и управления системами автомобилей, в том числе системами безопасности ABS и ASC. Разработанный интерфейс оказался настолько удачным, что к настоящему моменту он применяется повсеместно.

Типичными примерами применения CAN-сети являются системы контроля и управления текстильным оборудованием, производимого компаниями Cezoma, Lindauer Dornier, Rieter, Schlafhorst, Sulzer; полиграфическим оборудованием Heidelberger, Ferag; упаковочными машинами EDF, Island Beverages, Northrup King, Soudronic, TetraPak, Wepamat; деревообрабатывающим оборудованием компании Homag; производством полупроводников — Applied Materials.

Среди других важных областей применения сети нужно отметить промышленную автоматизацию и системы управление роботами. ABB, Bosch, Engel, Kuka и другие компании уже применяют CAN в своем оборудовании. Edeka использует CAN в своем центре логистики. Не обходятся без применения CAN-технологий системы автоматизации и жизнеобеспечения зданий, где они используются в управлении кондиционированием (Colt International), поддержания температуры (Buderus), интегрированного контроля состояния помещений и освещения. Часто CAN применяют в таких системах жизнеобеспечения зданий, как лестничный контроль Saarland Hall in Saarbruecken и Suedwestfunk in Baden-Baden, управление лифтами и подъемниками Kone,

Orenstein & Koppel, Otis, Telelift, пожарный контроль. Основные производители медицинского оборудования Combat Diagnostics, Dr.ger, Fresenius, GE Medical Systems, Philips Medical Systems, Siemens, Storz Endoskope, Toshiba также используют CAN в своих изделиях.

4.2.1. Характеристики протокола CAN

Интерфейс CAN имеет следующие основные характеристики.

- Среда передачи данных в CAN-спецификации не определена. Интерфейс с применением протокола CAN легко адаптируется к физической среде передачи информации. Это может быть дифференциальный сигнал, передаваемый по скрученной паре, оптоволокно, просто открытый коллектор и т. п.
- CAN-контроллер обеспечивает безотказную работу по стандарту ISO 11898 даже в условиях возникновения нижеперечисленных ситуаций:
  - любой из 3 проводов в шине оборван;
  - любой провод замкнут на питание;
  - любой провод замкнут на общий провод.
- Скорость передачи задается программно и может достигать 1 Мбит/с. Пользователь выбирает скорость, исходя из расстояний, числа абонентов и емкости линий передачи. Максимальное число абонентов, подключенных к данному интерфейсу, фактически определяется нагрузочной способностью примененных приемопередатчиков.
- Максимальная скорость передачи: 1 Мбит/с при длине линии до 40 м или 40 Кбит/с при длине линии 1000 м. При этом практически любой CAN-контроллер допускает программирование скорости обмена от 1 Мбит/с до 10 Кбит/с.
- Арбитраж организован так, что он не увеличивает время реакции системы на более приоритетные сообщения.
- Общее количество CAN-узлов не ограничено протоколом.
- Сообщения по CAN-шине могут передаваться одному или одновременно нескольким узлам, настроенным на прием одних и тех же параметров.
- Адресная информация (номер параметра) содержится в сообщении и совмещена с его приоритетом.
- Количество байтов данных настраивается от 0 до 8.
- Если хоть один узел в сети принял сообщение с ошибкой, это сообщение признается ошибочным для всех узлов сети.
- Отказавшие узлы динамически отключаются от шины.
- Подавление синфазных помех осуществляется дифференциальным приемопередатчиком.

4.2.2. Обмен данными в протоколе CAN

CAN — последовательная шина с поддержкой одновременной работы многих мастер-устройств, что означает возможность для всех узлов CAN-сети передавать данные и запрашивать шину нескольким узлам одновременно.

Устройства в CAN-системе соединяются по шине, состоящей из трех проводов (2 сигнальных и один общий).

Сообщения данных, передаваемые из любого узла по CAN-шине, могут содержать от 1 до 8 байт. Каждое сообщение помечено идентификатором, который в сети является уникальным (например, "Нагрев до 240", "Отказ нагрева", "Бункер загружен" и т. д.). При передаче другие узлы сети получают сообщение, и каждый из них проверяет идентификатор. Если сообщение имеет отношение к данному узлу, то оно обрабатывается, в противном случае — игнорируется. CAN-контроллер каждого из устройств может обрабатывать одновременно несколько идентификаторов (например, контроллеры Siemens и Intel могут обрабатывать до 15 идентификаторов). Таким образом, в каждом из устройств можно легко организовать несколько виртуальных каналов обмена информацией с различными устройствами, включая каналы одновременного получения сообщений.

Идентификатор определяет тип и приоритет сообщения. Более низкому числовому значению идентификатора соответствует более высокое значение приоритета. Сообщение, имеющее более высокий приоритет, передается раньше сообщения, имеющего более низкий приоритет. После сообщения с высоким приоритетом передается сообщение с более низким приоритетом, если во время передачи не появится сообщение с более высоким приоритетом; затем передается сообщение с еще более низким приоритетом и т. д.

4.2.3. Обнаружение ошибок в протоколе CAN

CAN содержит 5-ступенчатый механизм обнаружения ошибок.

- Циклический контроль по избыточности (CRC). Каждое переданное сообщение содержит контрольный код CRC, вычисленный передатчиком на основе содержания передаваемого сообщения. Приемные узлы выполняют аналогичную операцию, помечают обнаруженные ошибки и устанавливают соответствующие флаги.
- Контроль передаваемого поля битов. В составе CAN-сообщения передаются предопределенные битовые комбинации, которые контролируются при приеме. Если приемник обнаруживает недопустимый бит в одной из этих комбинаций, то устанавливается флаг "Ошибка формата".
- Контроль сигнала "Подтверждение приема". Каждое переданное сообщение подтверждается приемником, а если этого не произошло, устанавливается флаг "Ошибка подтверждения приема".

- Текущий контроль логического уровня битов. Любой передатчик автоматически контролирует и сравнивает фактический логический уровень битов на шине с уровнем, который он передает. Если уровни не совпадают, помечается ошибка логического уровня битов.
- Контроль заполнения битов. CAN использует методику добавления заполняющего бита для дополнительного контроля передаваемых сообщений. После передачи пяти последовательных битов с одинаковым уровнем передатчик автоматически вводит в разрядный поток бит противоположной полярности. Приемники сообщения автоматически удаляют такие биты перед обработкой сообщения. Если обнаруживается шестой бит одинаковой полярности, то помечается ошибка заполнения битов.

В случае если обнаружена ошибка, узел, обнаруживший ошибку, прерывает передачу посылкой флага ошибки. При этом передатчик автоматически производит повторную передачу сообщения, что предохраняет все узлы от возникновения ошибок и гарантирует непротиворечивость данных в сети.

4.3. Протокол Profibus

Сеть Profibus — это не зависящий от производителя открытый стандарт полевой шины для широкого ранга приложений в производственной автоматизации. Сеть Profibus — это комплексное понятие, она основывается на нескольких стандартах и протоколах. Эта сеть построена в соответствии с многоуровневой моделью ISO 7498-OSI и состоит из трех уровней:

- 1 — физический уровень;
- 2 — канальный уровень;
- 7 — уровень приложений.

Открытость и независимость от производителя гарантируется стандартом EN 50 170, все остальное реализовано в соответствии со стандартом DIN 19245 (техника передачи данных, методы доступа, протоколы передачи, сервисные интерфейсы для уровня приложений, спецификация протоколов, кодирование, коммуникационная модель и т. д.). С помощью Profibus устройства разных производителей могут работать друг с другом без каких-либо специальных интерфейсов. Семейство Profibus состоит из трех совместимых друг с другом версий: это Profibus-PA, Profibus-DP и Profibus-FMS.

Физически Profibus — это электрическая сеть, в основе своей использующая экранированную витую пару, или оптическая сеть на основе оптоволоконного кабеля. Скорость передачи по ней может варьироваться от 9,6 Кбит/с до 12 Мбит/с. Наиболее часто используется витая пара с интерфейсом RS-485.

Для всех версий Profibus существует единый протокол доступа к шине. Этот протокол реализуется на втором уровне модели ISO (который называется FDL). Данный протокол реализует процедуру доступа с помощью маркера (token). Сеть Profibus состоит из ведущих (master) и ведомых (slave) станций. Ведущая станция может контролировать шину, т. е. может передавать сообщения (без удаленных запросов), когда она имеет право на это (т. е. когда у нее есть маркер). Ведомая станция может лишь распознавать полученные сообщения или передавать данные после соответствующего запроса. Маркер вращается в логическом кольце, состоящем из ведущих устройств.

Если сеть состоит только из одного мастера, то маркер не передается (в данном случае в чистом виде будет система "мастер-подчиненный"). Сеть с минимальной конфигурацией может состоять либо из двух мастеров, либо из одного мастера и одного ведомого устройства.

Глава 5

COM-порты и Plug and Play

Windows 98 обнаружила изменение координат положения курсора мыши!
Изменения вступят в силу после перезагрузки системы.

В этой главе мы обсудим вопросы, связанные с автоматическим обнаружением устройств, подключенных к последовательным портам.

5.1. Кратко о Plug and Play

Появление огромного числа моделей периферийных устройств привело к очевидной невозможности их ручного конфигурирования. Выбор портов, номеров аппаратных прерываний, ячеек памяти... А ведь все параметры надо выставить корректно, исключить конфликты оборудования. При этом следует учитывать, что некоторые устройства умеют "подвинуться" на другие номера прерываний, а некоторые требуют фиксированного номера. Иными словами, очевидна необходимость автоматического конфигурирования устройств.

Протокол Plug and Play (дословно, "подключил и играй") позволяет достаточно просто подключать новое оборудование. Перед началом работы система (BIOS при начальной загрузке, Windows при запуске) опрашивает устройства, узнает их требования к системным ресурсам и пытается бесконфликтно разделить ресурсы между устройствами. Если это не удается, конфликтующие устройства будут работать некорректно. В этом случае необходимо вручную внести корректировки в настройки. Многие устройства, например, не поддерживают самостоятельное переключение диапазонов используемых ресурсов, но могут настраиваться с помощью специальных переключателей. Спецификация Plug and Play предусматривает также "горячее" подключение устройств, т. е. подключение во время работы.

Итак, основными функциями системы PnP (это сокращенное обозначение Plug and Play) являются:

- определение подключаемых устройств;
- идентификация устройств и уведомление операционной системы (ОС) об их появлении;

- определение отключения устройства и уведомление об этом ОС;
- автоматическое конфигурирование устройств без вмешательства пользователя.

Разумеется, в нашей книге мы рассмотрим только PnP-протокол для последовательных портов и связанных с ними устройств. Более того, мы будем рассматривать исключительно практический аспект спецификации.

5.2. Запуск процедуры PnP

При старте Windows производит краткий опрос наличия устройств. При необходимости существует возможность запустить этот процесс вручную.

Например, в Windows 98 необходимо нажать кнопку **Пуск**, выбрать пункт **Настройка**, в открывшемся окне выбрать **Панель управления** и активизировать иконку **Установка оборудования**. После этого будет показано диалоговое окно, похожее на рис. 5.1 (вид диалогового окна может немного меняться в зависимости от версии Windows). Нажатие на кнопку **Далее** (точнее говоря, надо два раза нажать **Далее**, ответив утвердительно на предупреждающий диалог) запускает процедуру быстрого поиска новых устройств. Те же действия можно выполнить, щелкнув правой кнопкой мыши по иконке **Мой компьютер**.

Рис. 5.1. Диалоговое окно поиска нового оборудования (Windows 98)

В Windows 2000/XP активизировать процедуру PnP-опроса можно и из окна **Device Manager** (Менеджер устройств), нажав кнопку **Обнаружение изменений аппаратуры** (Scan for hardware changes) (рис. 5.2).

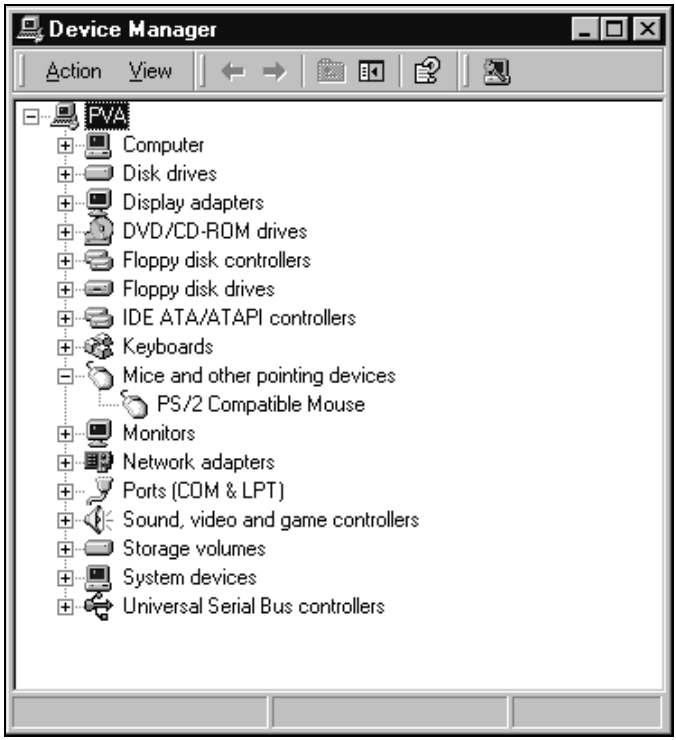

Рис. 5.2. Окно менеджера устройств (Windows 2000)

5.3. Если устройство не находится автоматически...

В Windows 98 SE существует проблема с обнаружением новых устройств, подключенных к COM-порту. Подключенные устройства могут не находиться даже после выполнения полной процедуры поиска.

Для обхода этой проблемы Microsoft рекомендует выключить APM (Advanced Power Management) для последовательных портов. Для этого необходимо внести следующее исправление в реестр Windows: значение ключа `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD\VCOMM` `EnablePowerManagement` заменить с <0100 0000> на <0000 0000>.

Важно!

Некорректное внесение изменений в реестр может повлечь серьезные проблемы, вплоть до полного выхода системы из строя.

В худшем случае, если это не помогает, можно удалить все COM-порты из системы и, перезагрузив компьютер, заставить Windows найти их заново.

5.4. Где хранится информация о найденных устройствах

5.4.1. Структура реестра Windows 98

Последовательные, параллельные порты или модемы могут устанавливаться тремя способами:

- установкой во время инсталляции Windows;
- сервисом Plug and Play;
- установкой вручную с помощью Помощника установки оборудования (Hardware Installation Wizard).

Каждое из установленных устройств имеет два раздельных ключа в реестре: ключ описания аппаратной части и ключ описания программной части (драйвер).

Аппаратный ключ реестра для последовательных портов расположен в ветке `HKEY_LOCAL_MACHIVE\Enum` вместе со всеми другими устройствами, обнаруженными в системе. Для устройств, не поддерживающих стандарт PnP, ключ располагается в ветке `Enum\Root`, а для устройств, совместимых со стандартом PnP, ключ будет располагаться в ветке `Enum\SERENUM` (для полноты описания отметим, что ключей в ветке `Enum` немного — для остальных устройств существуют ключи `Enum\ISA`, `Enum\PCMCIA` и `Enum\PCI`). Примеры ключей приведены в табл. 5.1.

Таблица 5.1. *Аппаратная часть описания устройств в реестре Windows 98*

Ветка реестра	Описание
`Enum\Root\*PNP0500\0000`	Первый COM-порт, найденный системой, обычно COM1. Последующие найденные порты будут нумероваться *PNP0500\0001 и т. д.
`Enum\Root\*PNP0400\0000`	Первый LPT-порт, найденный системой. Обычно LPT1
`Enum\Root\Ports\0000`	Вручную установленный порт. Или COM или LPT
`Enum\SERENUM\PNP0F0A\ ROOT&*PNP0500&0000`	PnP-последовательная Microsoft-мышь, найденная на порту COM1 с помощью SERENUM.VXD

Таблица 5.1 *(окончание)*

Ветка реестра	Описание
`Enum\LPTENUM\ABC1234\ ROOT&*PNP0400&0000`	PnP-параллельный порт с PnP-устройством, имеющим идентификатор ABC1234, найденное с помощью LPTENUM.VXD

Аппаратные ключи реестра имеют несколько веток и значений. Большинство из них создаются системой PnP для внутреннего использования, но они доступны для всех приложений. Последовательные устройства сохраняют два значения.

1. `FriendlyName` — понятное для пользователя системы название устройства. Например: "последовательный порт COM1" или "56K Внешний модем". Ключ создается или согласно содержимому INF-файла или самим VCOMM.
2. `PortName` — короткое имя порта. Например, "COM1" или "LPT1". Имена присваиваются системой. По этим именам производится открытие портов с помощью вызова `CreateFile`. В Windows 2000/XP имена портов можно сменить из окна Менеджера устройств (Device Manager) *(см. разд. 13.4)*.

Программная часть ключа (ключ драйвера устройства) сохраняется в ветке

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Class
```

Для каждого порта системы создается ключ в ветке `\ports`, например, `\ports\0000`. Для каждого модема ключ создается в ветке `\modem`, например, `\modem\0000`. Ссылка на ключ драйвера сохраняется также в аппаратной части ключа реестра, в имени "`Driver`".

В ключе драйвера создаются следующие значения.

- `ConfigDialog` — имя динамической библиотеки с функциями `CommConfigDialog`, `GetDefaultCommConfig` и `SetDefaultCommConfig`. Для COM-портов это обычно "`serialui.dll`", для модемов "`modemui.dll`", для LPT портов этот ключ не создается.
- `Contention` — определяет статическую VxD[1]. Для COM-портов и PCMCIA[2]-модемов это обычно "`*vcd`", а для LPT-портов "`*vpd`". Для нестандартных портов может быть установлено свое имя.
- `DCB` — бинарное содержание структуры DCB (Device Control Block, см. *разд. 20.5*), создаваемой вызовом `SetDefaultCommConfig`. Это значе-

[1] VxD (virtual device driver, виртуальный драйвер устройства).

[2] PCMCIA (Personal Computer Memory Card International Association, Международная ассоциация производителей плат памяти для персональных компьютеров IBM PC).

ние будет установлено при нажатии кнопки **Установить значения по умолчанию** в диалоговом окне `CommConfigDialog`.

- `DevLoader` — всегда "`*vcomm`".
- `DriverDesc` — читаемое название устройства. Например: "Последовательный порт", "Порт принтера".
- `Enumerator` — модуль, отвечающий за обнаружение новых устройств данного типа. Для COM-портов это "`serenum.vxd`", для LPT-портов это "`lptenum.vxd`". Для нестандартных устройств может быть указан свой модуль.
- `EnumPropPages` — имя DLL-модуля и имя функции (точка входа), вызывающая создание страницы свойств, для отображения свойств устройства в Менеджере устройств (Device Manager). Для COM-портов это "`serialui.dll, EnumPropPages`", для модемов это "`modemui.dll, EnumPropPages`", а LPT-порты обычно не имеют страницы свойств.
- `InfPath` — имя соответствующего INF-файла. Для обычных портов "`MSPORTS.INF`".
- `InfSection` — имя секции в файле `InfPath`. Для COM-портов обычно "`ComPort`", а для LPT-портов — "`LptPort`".
- `FriendlyDriver` — имя драйвера, загружаемого при открытии порта по имени, определенному в аппаратной части ключа реестра. Если этот ключ не определен, будет загружен драйвер из ключа `PortDriver`. Для модемов этот ключ всегда "`unimodem.vxd`".
- `PortDriver` — имя драйвера, загружаемого при открытии порта по имени, определенному в аппаратной части ключа реестра. Для COM-портов это "`serial.vxd`", а для LPT-портов это "`lpt.vxd`". Для внутренних и внешних модемов это поле не используется, однако для UART-совместимых PCIMCIA-модемов это значение устанавливается в "`serial.vxd`".
- `PortSubClass` — класс устройства:
  - 00 для параллельных портов;
  - 01 для последовательных портов;
  - 02 для модемов.

5.4.2. Структура реестра Windows 2000

В Windows 2000 структура реестра значительно отличается от Windows 98. Устройство может использовать одну из трех возможных настроек:

- настройки, специфические для устройства;
- если настройки первого типа не существуют, берутся настройки последовательного порта, связанного с устройством;

- если настроек последовательного порта не существует, берутся настройки, "зашитые" в драйвер `serial.sys`.

В ветке реестра

```
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\Serial
```

хранятся общие настройки последовательных портов. Эта ветка содержит следующие ключи.

- `ForceFifoEnable` — если этот ключ имеет ненулевое значение, режим FIFO используется независимо от того, поддерживается ли он устройством. Если этот ключ равен 0, режим FIFO будет включен, только если драйвер обнаружит, что устройство поддерживает этот режим.
- `RxFIFO` — количество байтов в FIFO-буфере приемника, вызывающее прерывание (по умолчанию 8).
- `TxFIFO` — количество байтов в FIFO-буфере передатчика, вызывающее прерывание (по умолчанию 14).
- `PermitShare` — разрешение или запрещение использования прерывания несколькими устройствами одновременно. Если значение ненулевое, может использоваться разделяемое прерывание. По умолчанию 0.

Ветка реестра

```
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\Serial\Enum
```

сохраняет текущее количество найденных портов и содержит следующие ключи.

- `0`, `1`, ... — последовательно нумерованные ключи найденных портов. Значение этих ключей равно идентификатору PnP соответствующего порта, например, ключ `0 = "ACPI\PNP0501\1"`, `1 = "ACPI\PNP0501\2"`.
- `Count` — число найденных портов.

Следует учитывать, что в этой ветке сохраняются только действительные порты, поэтому, например, виртуальный последовательный порт USB не будет отображен в этой ветке реестра.

А вот ветка реестра

```
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\Serenum\Enum
```

сохраняет все последовательные порты: и аппаратные, и виртуальные. Ключи этой ветки такие же, как и у предыдущей, но ссылки на порты могут содержать не только COM-порты, например:

- `0 = ACPI\PNP0501\1`
- `1 = ACPI\PNP0501\2`
- `2 = USB\Vid_067b&Pid_2303\5&38488c91&0&1`
- `Count = 3`

Ветка реестра

```
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\Serial\Parameters
```

сохраняет информацию об аппаратных свойствах порта. Если параметры не отличаются от стандартных, эта ветка будет пуста, в противном случае в ней могут присутствовать следующие параметры (мы приведем наиболее важные параметры, а их полный список можно найти в MSDN).

- `PortName` — имя порта. По умолчанию порты используют имена COMx, однако может быть установлено любое имя.
- `MultiPortDevice` — если не ноль, то это порт на мультипортовой карте.
- `ClockRate` — обозначает частоту кварцевого генератора UART. По умолчанию 1 843 200 Гц.

При отключении устройства из данной конфигурации запись об этом производится в ветку

```
HKEY_CURRENT_CONFIG\System\CurrentControlSet\Enum\ACPI\идент_PnP
```

Например, при выключении порта COM1 будет создан ключ

```
HCC\...\ACPI\PNP0501\1\CSConfigFlags = 1
```

5.5. Алгоритм Plug and Play для COM-портов

В этом разделе мы приведем стандартный алгоритм работы PnP для последовательных портов.

5.5.1. Инициализация порта *(Port initialization)*

- Система пытается открыть порт с помощью системных функций.
- Если порт занят (например, модем или мышь используют этот порт), то порт не будет конфигурироваться.
- Если IRQ (Interrupt ReQuest, запрос прерывания), требуемое порту, уже занято другим драйвером, порт не будет конфигурироваться.

5.5.2. Обнаружение устройств *(Check for device)*

- Выход TxD устанавливается в состояние MARK IDLE.
- Устанавливается DTR = ON, RTS = OFF.

- В течение (200 ± 35) мс ожидается появление сигнала DSR, которое указало бы на наличие устройства, подключенного к порту. В простейшем случае устройство имеет на разъеме перемычку DTR-DSR, обеспечивающую указанный ответ.
- Если пауза ожидания завершена, а DSR = OFF, то система переходит к алгоритму ожидания отключения *(см. разд. 5.5.10)*.
- Если устройство обнаружено, выполняется первая фаза установки устройства *(см. разд. 5.5.3)*.

5.5.3. Установка устройства, фаза 1 *(COM port Setup-1)*

- Порт программируется на режим 1200 бит/с, 7 бит данных, без паритета, 1 стоп-бит.
- Устанавливается DTR = OFF, RTS = OFF.
- Отсчитывается пауза (200 ± 35) мс.
- Устанавливается DTR = ON.
- Отсчитывается пауза (200 ± 35) мс.
- Система переходит на первую фазу алгоритма ожидания ответа устройства *(см. разд. 5.5.4)*.

5.5.4. Ожидание ответа, фаза 1 *(Wait for response-1)*

- Устанавливается RTS = ON.
- В течение (200 ± 35) мс ожидается приход первого символа от устройства. По приходе символа начинается прием идентификатора, т. е. выполнение алгоритма получения идентификатора *(см. разд. 5.5.7)*.
- Если за это время символ не пришел, выполняется вторая попытка опроса — алгоритм установки устройства, фаза 2 *(см. разд. 5.5.5)*.

5.5.5. Установка устройства, фаза 2 *(COM port Setup-2)*

- Устанавливается DTR = OFF и RTS = OFF.
- Выдерживается пауза (200 ± 35) мс.
- Система переходит к алгоритму ожидания ответа, фаза 2 *(см. разд. 5.5.6)*.

5.5.6. Ожидание ответа, фаза 2 *(Wait for response-2)*

- Устанавливаются DTR = ON и RTS = ON.
- В течение (200 ± 35) мс ожидается приход первого символа от устройства, по приходе символа начинается прием идентификатора, т. е. выполнение алгоритма получения идентификатора *(см. разд. 5.5.7)*.
- Если за это время символ не пришел и DSR = OFF, то система переходит к выполнению алгоритма проверки отключения устройства (см. *разд.* 5.5.8). Если же DSR = ON, то выполняется переход в дежурное состояние *(см. разд. 5.5.9)*.

5.5.7. Получение идентификатора *(Collect device ID)*

Посимвольный прием идентификатора устройства имеет ограничения по тайм-ауту в T1 = 200 240 мс на символ, а также общее ограничение времени получения T2 = 2,2 с. Это позволяет принять строку длиной до 256 символов. Это ограничение получается простым вычислением:

(256 символов × 10 бит/символ) / (1200 бит/с) = 2,13 с.

Прием идентификатора выполняется по следующему набору правил.

- Принимаются и складываются в буфер все получаемые символы.
- Если после получения первого символа обнаружена ошибка четности или ошибка кадра, то система переходит в дежурный режим *(см. разд. 5.5.9)*.
- Если хотя бы один символ получен и тайм-аут ожидания символа T1 завершен, выполняется проверка правильности полученного идентификатора. В случае корректного идентификатора система уведомляется о добавлении нового устройства. Иначе система переходит в дежурное состояние *(см. разд. 5.5.9)*.
- Если получен символ конца идентификатора (будет описан далее), выполняется проверка правильности полученного идентификатора. В случае корректного идентификатора система уведомляется о добавлении нового устройства. Иначе система переходит в дежурное состояние *(см. разд. 5.5.9)*.
- Если по завершению ожидания T1 или T2 символ начала идентификатора не получен, система переходит в дежурное состояние *(см. разд. 5.5.9)*.
- Если по завершению ожидания T2 символ завершения идентификатора не получен, система переходит в дежурное состояние *(см. разд. 5.5.9)*.
- Если DSR = OFF, система переходит к алгоритму проверки отключения *(см. разд. 5.5.8)*.

5.5.8. Проверка отключения *(Verify Disconnect)*

На этом шаге проверяется, действительно ли устройство удалено из системы. Используется для модемов и других устройств, которые кратковременно сбрасывают DSR, когда снимается DTR, например, при инициализации или сбросе устройства. Система понимает, что устройство отключено, если DSR = = OFF после некоторого тайм-аута.

Для проверки отключения выполняются следующие действия:

- устанавливаются DTR = ON, RTS = OFF;
- отрабатывается пауза 5 с;
- если после паузы DSR = ON, то система переходит к алгоритму дежурного состояния *(см. разд. 5.5.9)*;
- если же после отработки паузы DSR = OFF, система переходит к алгоритму ожидания отключения *(см. разд. 5.5.10)*.

5.5.9. Дежурное состояние *(Connect Idle)*

- Устанавливаются DTR = ON, RTS = OFF.
- Порт программируется на режим 300 бит/с, 7 бит данных, без паритета, 1 стоп-бит.
- Если в этом состоянии обнаружится DSR = OFF, система уведомляется об отключении устройства и производится переход в режим ожидания отключения *(см. разд. 5.5.10)*.
- Необязательный шаг: система производит мониторинг порта для обнаружения устройств, не поддерживающих спецификацию PnP.

5.5.10. Состояние ожидания отключения *(Disconnect Idle)*

Это состояние не обязательно, если система производит мониторинг порта для обнаружения динамически подключаемых устройств или устройств, не поддерживающих спецификацию PnP.

Система уведомляется об удалении устройств, если до этого они были обнаружены для данного порта.

В этом режиме производятся следующие действия:

- устанавливаются DTR = ON, RTS = OFF;
- порт программируется на 300 бит/с, 7 бит данных, без контроля четности, 1 стоп-бит;

- если в этом состоянии обнаружится DSR = 1, производится переход к алгоритму установки устройства, фаза 1 *(см. разд. 5.5.4)*.
- необязательный шаг: производится обнаружение других устройств, не поддерживающих PnP.

5.6. Формат данных для передачи PnP-идентификатора

Все PnP-совместимые устройства должны передавать свой идентификатор со скоростью 1200 бит/с (это наследство досталось от мыши). Формат данных должен быть совместим с 9-битовым форматом кадра приемника: один стартовый бит, 7 бит данных, 1 стоп-бит. Например, совместимыми являются такие форматы данных:

- 9-битовый кадр: 1 старт-бит, 7 бит данных, нет четности, 1 стоп-бит;
- 10-битовый кадр: 1 старт-бит, 7 бит данных, нет четности, 2 стоп-бита;
- 10-битовый кадр: 1 старт-бит, 7 бит данных, постоянная MARK-четность, 1 стоп-бит;
- 10-битовый кадр: 1 старт-бит, 8 бит данных с установленным в 0 старшим битом, нет контроля четности, 1 стоп-бит.

Эксперименты с совместимостью кадров мы будем проводить в *разд 10.5*.

5.7. Передача PnP-идентификатора

Алгоритм опроса и передачу идентификатора каждое устройство осуществляет самостоятельно, согласно схемам, приведенным выше. Естественно, устройство должно реализовывать "обратную" часть этих алгоритмов.

Так, например, мышь действует согласно следующим правилам:

- передача DTR на DSR осуществляется аппаратно;
- при начале опроса (т. е. DTR = ON и TD = MARK) производится ожидание RTS = ON;
- когда RTS становится ON, мышь передает свой идентификатор;
- после передачи идентификатора мышь переходит в обычное рабочее состояние.

Такой алгоритм может быть реализован и для других устройств, имеющих аппаратную перемычку RTS-DTR.

Алгоритм модемов сложнее:

- при включении питания DSR устанавливается в состояние ON;
- в режиме обычного функционирования модем контролирует сигналы RTS и DTR;
- если модем обнаруживает состояние DTR = OFF, RTS = OFF и IDLE (т. е. отсутствие активных звонков), он переходит в режим ожидания установки DTR;
- если DTR переходит в состояние ON, выполняются следующие шаги:
  - отсчитываются тайм-ауты T1 = 150 мс и T2 = 250 мс, в течение которых контролируется RTS;
  - если DTR становится OFF, модем снова переходит в состояние IDLE;
  - если RTS становится ON раньше, чем закончится тайм-аут T1, модем переходит в состояние IDLE;
  - если RTS = ON, T1 завершился, а T2 еще нет, то модем отправляет компьютеру свой PnP-идентификатор и переходит в режим IDLE;
  - если RTS = OFF и после завершения тайм-аута T2, модем переходит в режим IDLE.

Похожий алгоритм должен быть реализован и для других устройств, не имеющих аппаратной перемычки RTS-DTR.

5.8. Поля PnP-идентификатора

Устройства, поддерживающие стандарт PnP, должны передавать свой идентификатор в специальном формате. Поля передаваемой информации показаны в табл. 5.2. Первое поле зарезервировано для старых устройств (например, мышей), которые самостоятельно генерируют один или несколько символов при включении питания. Следующие 4 поля и последнее являются обязательными, а все оставшиеся — необязательными. Если используются дополнительные поля, контрольная сумма обязательна.

Для совместимости со старыми драйверами мыши все указатели, совместимые с этим манипулятором, должны урезать свои данные до 6-битового набора символов для всех полей, исключая поле `Other ID`. Следовательно, все такие поля должны содержать символы из набора $00$3F или $20$5F. Выбор диапазона производится заданием символа `Begin PnP`. Естественно, символ `End PnP` также должен соответствовать выбранному диапазону.

Таблица 5.2. *Поля PnP-идентификатора*

Название поля	Размер	Необходимость	Описание
Other ID	<17	Нет	Зарезервировано для коротких не-PnP-идентификаторов. Может быть игнорировано системой
Begin PnP	1	Да	Начальный символ. Это либо $28 (символ "("), либо $08
PnP Rev	2	Да	PnP-версия (например, $00 или $01)
EISA ID	3	Да	Уникальный идентификатор производителя
Product ID	4	Да	Уникальный идентификатор продукта
Extend	1	Нет	"\": или $5C или $3C
Serial Number	8	Нет	Необязательный серийный номер устройства
Extend	1	Нет	"\": или $5C или $3C
Class Name	<33	Нет	PnP-идентификатор класса устройства
Extend	1	Нет	"\": или $5C или $3C
Device ID	<41	Нет	Идентификатор совместимости драйверов
Extend	1	Нет	"\": или $5C или $3C
User Name	<41	Нет	Торговая марка продукта
Checksum	2	Да, если есть дополнительные поля	8-битовая арифметическая контрольная сумма всех байт, начиная с Begin PnP до End PnP включительно, исключая байты самой контрольной суммы. Две шестнадцатеричные цифры
End PnP	1	Да	Завершающий символ PnP-идентификатора. Это символ ")": или $29 или $09

Все дополнительные поля должны начинаться с символа "\". Если дополнительное поле отсутствует, но следующее за ним указано, то выводится просто одиночный символ "\". Символ "\" имеет код $3C для 6-битовой кодировки или $5C для 7-битовых кодов ASCII. Для пустых дополнительных полей в конце пакета символы "\" могут не указываться.

Как мы уже считали, длина всего пакета не должна превышать 256 символов.

Естественно, все идентификационные символы между `Begin PnP` и `End PnP` не должны меняться, иначе система будет обнаруживать все новые и новые устройства...

5.8.1. Поле *Other ID*

Первое поле, длина которого не более 16 символов, может содержать необязательный идентификатор. Это поле не обязательно, оно оставлено для совместимости со старыми устройствами. Так, например, мышь генерирует посылку символа "M" (код $4D) при подаче питания. Новые PnP-устройства могут не заполнять это поле. Единственное ограничение: это поле не должно содержать идентификаторов `Begin PnP` или `End PnP` (т. е. ни одного из кодов $08, $28, $09, $29). Поле может включать 7-битовые символы "CR" и "LF" для улучшения читаемости.

5.8.2. Поле *Begin PnP*

Поле `Begin PnP` маркирует начало PnP-идентификатора. Это либо 7-битовый символ $28, либо 6-битовый символ $08. В зависимости от `Begin PnP` обработчик PnP будет ожидать соответствующего завершающего символа `End PnP`.

5.8.3. Поле *PnP Rev*

Два байта (точнее 12 бит) поля `PnP Rev` обозначают номер версии PnP. Двенадцать бит получаются из бит 5—0 первого символа, соединенных с битами 5—0 второго символа. Для получения номера версии это число делится на 100 (см. листинг 17.1). Таким образом, номер версии будет находиться в границах от 0,00 ($00, $00) до 40,95 ($3F, $3F). При этом следует учитывать, что это поле (так же как и все другие) не может содержать символов `Begin PnP` и `End PnP`, поэтому версия 1,05 (код $01, $29) и версия 1,37 (код $02, $09) не могут быть использованы.

5.8.4. Поле *EISA ID*

Каждый производитель оборудования должен иметь уникальный трехбуквенный идентификатор. Если используется 6-битовый код (т. е. `Begin PnP` равно $08), то из кодов ASCII вычитается $20. Для получения таких идентификаторов следует обращаться в специальные сервисные центры.

5.8.5. Поле *Product ID*

Все коммуникационные PnP-устройства должны иметь уникальный идентификатор, определяемый 16-битовым значением. Для 6-битового кода шестнадцатеричные символы смещаются на $20.

5.8.6. Поле *Extend*

Значение поля `Extend` представляется символом обратного слеша "\" (код $5C) для 7-битового кода и символом $3C для 6-битового. Указание такого символа означает, что будет использоваться поле, следующее за этим символом. Например, указание первого символа означает, что будет заполнено поле `Serial Number`. Если поле `Serial Number` не будет использоваться, а будет использовано поле `ClassName`, то необходимо указать два таких символа.

Если хотя бы одно дополнительное поле используется, необходимо указание контрольной суммы.

5.8.7. Поле *Serial Number*

Поле `Serial Number` является необязательным. Устройства могут сообщать свой серийный номер. Этот номер может использоваться драйверами или файлами лицензий. Серийный номер должен состоять из 4 символов.

5.8.8. Поле *Class Name*

Поле `Class Name` — необязательное поле. Оно должно быть не длиннее 32 символов. Передает имя класса устройства. Производители оборудования не могут добавлять свои имена. Возможные значения этого поля приводятся в табл. 5.3. В новых версиях Windows таблица имен расширена.

Таблица 5.3. *Значения поля `Class Name` идентификатора PnP*

Значение	Описание	Версия
`MODEM`	Модем (Data, FAX, Voice и т. д.)	
`MOUSE`	Мышь	
`PRINTER`	Принтер	
`SYSTEM`	Системное устройство. Для Windows 95/98/ME сюда же относятся UPS. В новых версиях Windows выделено отдельное имя	Windows 95/98/ME
`BATTERY`	Устройства питания (например, UPS)	Windows NT/2000/XP

Таблица 5.3 *(окончание)*

Значение	Описание	Версия
`Multifunction`	Комбинированные карты, например, PCMCIA-модемы	
`Multiportserial`	Мультипортовые PnP-карты, но не подключаемые к ним устройства. Простая (старая) мультипортовая карта типа 16550 не включается в этот тип	
`Ports`	COM- и LPT-порты	

5.8.9. Поле *Device ID*

Необязательное поле `Device ID` включает одну или более строк, разделенных символом запятой "," (код $0C или $2C). Идентифицирует список совместимых драйверов для данного устройства. Каждый `Device ID` содержит три символа `EISA ID` и четыре символа `Product ID`.

5.8.10. Поле *User Name*

`User Name` — необязательное поле, содержащее не более 40 символов. Содержит сведения о производителе продукта и о самом продукте. Это поле будет отображаться в списке устройств и в окне **Обнаружено новое устройство**. Строка не должна содержать символов `End PnP`, т. е. $09 и $29.

Для читаемости это поле может содержать символы "CR" и "LF" для 7-битового кода.

5.8.11. Поле *Checksum*

Поле `Checksum` представляет собой контрольную сумму. Если указано хотя бы одно дополнительное поле, указание контрольной суммы обязательно. Контрольная сумма считается как арифметическая сумма всех байт между `Begin PnP` и `End PnP` включительно без символов самой контрольной суммы (см. листинг 17.2).

5.8.12. Поле *End PnP*

Поле `End PnP` — символ завершения PnP-идентификатора. Этот символ должен соответствовать выбранной кодировке. Если `Begin PnP` был $08, то `End PnP` должен быть $09, а для `Begin PnP` $28 должен использоваться код $29.

5.8.13. Пример PnP-идентификатора

Мы приведем несколько примеров PnP-идентификаторов. Следует отметить, что все поля вымышленные и к реальным устройствам отношения не имеют.

В табл. 5.4 показан пример PnP-идентификатора для мыши без посылки дополнительных полей. Использован 6-битовый код.

Таблица 5.4. *Пример идентификатора PnP (мышь)*

Код	Поле	Описание
4D	Other ID	Идентификатор старой мыши
08	Begin PnP	Начало PnP ID
00,01	PnP Rev	Версия PnP равна 0,01
21,2D,23	EISA ID	"AMC" (Компания "Mouse")
11,12,13,14	Product ID	"1234" (случайное число)
09	End PnP	Конец PnP ID

В табл. 5.5 приведен пример PnP-идентификатора некоторого модема, передающего дополнительные поля и использующего 7-битовую кодировку.

Таблица 5.5. *Пример идентификатора PnP (модем)*

Код	Поле	Описание
—	Other ID	Поле пропускается, это не мышь
$28	Begin PnP	Начало PnP ID
$01,$24	PnP Rev	Версия PnP 1,00
$4D,$44,$43	EISA ID	"MDC" (Modem Design Company)
$30,$32,$38,$38	Product ID	"0288" (случайное число)
$5C,$30,$30,$33,$31, $34,$31,$35,$39	Serial #	"\00314159" (случайное число)
$5C,$4D,$4F,$44,$45, $4D	Class Name	"\MODEM" (класс "модем")
$5C,$4D,$44,$43,$30, $31,$34,$34,$2C,$41, $54,$4D,$30,$30,$39, $36	Device ID	"\MDC0144","ATM0096" (список совместимых драйверов)
$5C,$5A,$49,$50,$20, $32,$38,$38	User Name	"\ZIP 288" (имя модели)

Таблица 5.5 *(окончание)*

Код	Поле	Описание
$43,$34	`Checksum`	"C4" (контрольная сумма)
$29	`End PnP`	Конец PnP ID

5.9. INF-файл и его структура

Для установки драйвера устройства Windows ищет специальный файл с расширением inf. Этот файл содержит всю информацию о действиях, которые необходимо произвести для установки драйвера: какие файлы нужно перенести и зарегистрировать в системе, какие ветки реестра необходимо создать и т. д. Кроме того, этот же файл содержит информацию о действиях, которые нужно произвести при удалении устройства из системы, а также дополнительную информацию о производителе устройства.

Структура INF-файла полностью совпадает с обычным INI-файлом: файл состоит из секций и некоторого набора ключей в них.

Конечно же, описывать все поля INF-файла мы не будем. Для этого потребовалась бы книга в несколько раз больше этой. Мы постарались выбрать тот минимум полей и их значений, который будет нам необходим при создании и установке драйвера устройства в *гл. 17*.

5.9.1. Секция *Version*

Секция `Version` содержит основную информацию о INF-файле.

- Ключ `Signature = "signature-name"` описывает версию Windows, для которой применим этот INF-файл. Значение ключа не зависит от регистра, но должно точно соответствовать указанной ниже записи строк (знаки "$" в начале и конце строки обязательны).
  - `"$Windows NT$"` — операционные системы NT.
  - `"$Windows 95$"` — системы Windows 95/98/ME.
  - `"$Chicago$"` — любая версия Window.
- Ключ `Class = class-name` описывает тип устройства. Значение может быть одной из стандартных строк, список которых мы привели в *разд. 5.8.8*. Следует отметить, что если значение `Class` из INF-файла не совпадает со значением, переданным устройством, INF-файл все равно будет считаться правильным, а система установит устройство согласно значению из INF-файла.

Если указывается новый тип устройства, должен быть указан ключ `ClassGuid`, содержащий GUID нового типа устройства. Длина значения ключа `Class` не может превышать 32 символа.

- Ключ `ClassGuid = {nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}` описывает GUID типа устройства. Обязательно указывается для нового типа устройства.
- Ключ `Provider = %INF-creator%` содержит имя компании — разработчика INF-файла.
- Ключ `DriverVer = mm/dd/yyyy[,x.y.v.z]` — это информация о версии драйвера, устанавливаемого INF-файлом. Поле `mm/dd/yyyy` указывает дату драйвера, а необязательные параметры `x`, `y`, `v`, `z` — версию драйвера.

  Если версия драйвера указывается, то каждая цифра должна быть в диапазоне от 0 до 65 535. Следует учитывать, что в версиях Windows ME, 2000 и XP это значение не учитывается при установке драйвера и служит только для отображения версии в Менеджере устройств (Device Manager). А в Windows XP SP1, Windows Server 2003 и более поздних версиях это значение учитывается программой установки драйвера.

  При установке или переустановке драйвера Windows сравнивает дату уже установленного драйвера и всех подходящих INF-файлов и выбирает наиболее новый (кроме Windows 98/ME, которые не учитывают дату при установке драйверов). Если ключ `DriverVer` не указан, система считает дату как `00/00/0000`. Таким образом, файлы без этого ключа автоматически считаются наиболее старыми.

5.9.2. Секция *Manufacturer*

Секция `Manufacturer` описывает производителя одного или более устройств, устанавливаемых INF-файлом.

Секция имеет следующий формат:

```
[Manufacturer]
manufacturer-identifier
[manufacturer-identifier]
[manufacturer-identifier]
... ... ... ... ... ... ...
```

Каждый ключ секции содержит информацию, описывающую одну модель устройства, и должен располагаться на отдельной строке. Допустимо использование одного из следующих форматов:

1. `Manufacturer-name`
2. `%strkey%=models-section-name`
3. `%strkey%=models-section-name[,TargetOSVersion][,TargetOSVersion]`

В случае использования первого формата INF-файл должен содержать секцию с таким же именем *(см. разд. 5.9.4)*. Например:

```
; пример из файла DECPSMW4.INF
[Manufacturer]
"Digital"
[Digital]
"Digital DEClaser 5100/Net"=D5100_MS.SPD,Digital_DEClaser_5100/Net
```

Второй формат аналогичен первому, но позволяет использовать строки для перевода, указываемые в секции `String`. Например:

```
; пример из файла CXPDFPCI.INF
[Manufacturer]
%String1%=DIGI
[DIGI]
%String2%=CxpPCI1ST.Install,MF\DIGIPCI1ST_DEV0
[Strings]
String1="Digi International"
String2="Digi DataFire PCI 1 S/T (CXP)"
```

Естественно, в секции `Strings` должны быть определены все используемые ссылки `%strkey%`.

Третий формат описания доступен, только начиная с версии Windows XP. Он позволяет указывать список версий и типов Windows, для которых предназначен данный INF-файл. Программа установки будет выбирать тот драйвер, который наиболее близко подходит к перечисленным описаниям.

Формат строки `TargetOSVersion` следующий:

```
NT[Architecture][.[OSMajorVersion][.[OSMinorVersion][.[ProductType]
[.SuiteMask]]]]
```

Кратко рассмотрим поля этой строки:

- идентификатор `NT` указывает, что это описание поддерживается только версиями Windows семейства NT;
- поле `Architecture` описывает аппаратную платформу. Если указывается, должен быть или x86 или ia64;
- поле `OSMajorVersion` задает старший номер версии операционной системы. Для Windows XP это номер 5;
- поле `OSMinorVersion` задает младший номер версии операционной системы. Для Windows XP это номер 1;

- поле ProductType может быть одной из констант VER_NT_xxx, определенных в winnt.h, например:
  - 0x0000001 = VER_NT_WORKSTATION;
  - 0x0000002 = VER_NT_DOMAIN_CONTROLLER;
  - 0x0000003 = VER_NT_SERVER;
- поле SuiteMask — это маска из значений констант VER_SUITE_xxxx, определенных в winnt.h, например:
  - 0x00000001 = VER_SUITE_SMALLBUSINESS;
  - 0x00000002 = VER_SUITE_ENTERPRISE;
  - 0x00000004 = VER_SUITE_BACKOFFICE;
  - 0x00000008 = VER_SUITE_COMMUNICATIONS;
  - 0x00000010 = VER_SUITE_TERMINAL;
  - 0x00000020 = VER_SUITE_SMALLBUSINESS_RESTRICTED;
  - 0x00000040 = VER_SUITE_EMBEDDEDNT;
  - 0x00000080 = VER_SUITE_DATACENTER;
  - 0x00000100 = VER_SUITE_SINGLEUSERTS;
  - 0x00000200 = VER_SUITE_PERSONAL;
  - 0x00000400 = VER_SUITE_SERVERAPPLIANCE.

Если ключи секции Manufacturer содержат определения версий ОС, то и платформы, и секции, на которые ссылаются указанные ключи, должны содержать те же определения, например:

```
[Manufacturer]
%MyName% = MyName,NTx86.5.1
... ... ... ...
[MyName]
%MyDev% = InstallA,hwid
... ... ... ...
[MyName.NTx86.5.1]
%MyDev% = InstallB,hwid
... ... ... ...
[InstallA.ntx86]   ; Windows 2000 (NT4-x86 будет также пытаться
                   ; читать эту секцию)
... ... ... ... ...
[InstallA]         ; Win98/WinME (Win95 также будет пытаться
                   ; читать эту секцию)
... ... ... ... ...
[InstallB]         ; Windows XP и позже, и только x86
... ... ... ... ...
```

5.9.3. Секция *DestinationDirs*

Секция `DestinationDirs` указывает одну или несколько директорий для копирования, удаления и переименования файлов.

Эта секция необходима в INF-файле, если он содержит либо ключ `CopyFiles`, либо ссылку на секции `CopyFiles`, `DelFiles` или `RenFiles`.

Каждая директория описывается на отдельной строке секции и имеет формат:

```
[DestinationDirs]
[DefaultDestDir=dirid[,subdir]]
[file-list-section=dirid[,subdir]] ...
```

Ключ *DefaultDescDir*

Ключ `DefaultDescDir` указывает директорию по умолчанию для копирования, удаления и переименования файлов, которые не описаны в списке `file-list-section`.

Ключи *file-list-section*

Ключи `file-list-section` перечисляют имена файлов и их директории, если нужно установить директории, отличные от `DefaultDescDir`.

Ключ *dirid*

Ключ `dirid` указывает директорию, в которой будет находиться указанный файл или файлы. Это может быть или абсолютный путь или номер (идентификатор) директории. Абсолютный путь может ссылаться на секцию `String`.

Идентификатор директории представляет собой целое число. Значения в диапазоне от −1 до 32 767 зарезервированы (табл. 5.6). Кроме того, следует учитывать, что в целях совместимости с Windows 98/ME значение 65 535 приравнивается к −1.

Таблица 5.6. *Таблица значение* `dirid`

Значение	Описание
1	*SourceDrive*:*pathname* (указывает директорию, из которой был установлен INF-файл)
10	Windows-директория, т. е. %windir%.
11	Системная директория, т. е. %windir%\system32 для NT-систем, и %windir%\system для Windows 9x/ME
12	Директория драйверов, т. е. %windir%\system32\drivers для NT-систем и %windir%\system\IoSubsys для Windows 9x/ME

Таблица 5.6 *(окончание)*

Значение	Описание
17	Директория INF-файлов, т. е. %windir%\INF
18	Директория файлов помощи, т. е. Help directory %windir%\HELP
20	Директория шрифтов
24	Системный диск, т. е. если Windows установлена в папку C:\winnt, то это будет "C:\"
30	Корневая директория загрузочного диска (для NT-систем необязательно совпадает с ID24)
50	Системная директория для NT-систем, т. е. %windir%\system (только для NT)
53	Директория пользовательского профиля
54	Директория, где расположены файлы ntldr.exe и osloader.exe (для NT-систем)
−1	Абсолютный путь
16406	All Users\Start Menu
16407	All Users\Start Menu\Programs
16408	All Users\Start Menu\Programs\Startup
16409	All Users\Desktop
16415	All Users\Favorites
16419	All Users\Application Data
16422	Program Files
16427	Program Files\Common
16429	All Users\Templates
16430	All Users\Documents

Ключ *subdir*

Ключ `subdir` обозначает директорию относительно `dirid`, наример,

```
; пример из файла ICW97.INF
[DestinationDirs]
CopyHELP    = 18                    ; LDID_HELP
CopySYS     = 11                    ; LDID_SYS
CopyINF     = 17                    ; LDID_INF
DeleteICW2  = 24,%ProgramFiles%\%OLD_ICWDIR%
```

Обратите внимание, что поддиректория в dirid указывается через запятую, а не через слэш.

5.9.4. Секция описания модели

Секции описания моделей должны соответствовать ссылкам на модели из секции Manufacturer *(см. разд. 5.9.2)*. Формат строк этой секции имеет вид:

```
Device-description=install-section-name,hw-id[,compatible-id...]
```

- Поле device-description — это любая уникальная строка описания или ссылка на строку в секции String.
- Поле install-section-name содержит имя секции, описывающей устройство.
- Поле hw-id содержит строку, соответствующую Hardware ID в идентификаторе PnP и может иметь один из нижеперечисленных форматов.
  - Enumerator\device-id — обычный формат для устройства, устанавливаемого по спецификации PnP. Например, SERENUM\PVA0001.
  - *device-id — указывает, что устройство поддерживается различными сервисами. Например, *PNP0F01 идентифицирует Microsift-мышь, которая имеет совместимый идентификатор SERENUM\PNP0F01.
  - Device-class-id — спецификатор шины, как он описан в аппаратной спецификации.
- В поле compatible-id содержатся один или несколько совместимых с hw-id типов устройств, разделенных запятой.

Рассмотрим пример, который мы будем использовать в практической части книги *(см. гл. 17)*. Секция Manufacturer ссылается на секцию CompanyName, которая содержит описание устройств, подключаемых к последовательному порту (SERENUM) и имеющих идентификаторы PVA0001 и PVA0000. Для таких устройств описание драйвера будет находиться в секции MyDevice_SECTION.

```
[Manufacturer]
%Mfg%=CompanyName
[CompanyName]
%MyDeviceStr%=MyDevice_SECTION, SERENUM\PVA0001, PVA0000
[MyDevice_SECTION]
CopyFiles= MyDevice.CopyFiles
AddReg=MyDevice.AddReg
DelReg=MyDevice.DelReg
LogConfig=MyDevice.Config
```

5.9.5. Секции *xxx.AddReg* и *xxx.DelReg*

Эти секции описывают действия с реестром при установке и удалении устройства. Ссылка на имена секций дается из секции описания модели *(см. разд. 5.9.4)*.

Формат строк этой секции имеет вид:

```
Reg-root,[subkey],[value-entry-name],[flags],[value]
```

- Поле `reg-root` — идентификатор корневой ветки реестра. Может быть одним из обозначений:
  - `HKCR = HKEY_CLASSES_ROOT`;
  - `HKCU = HKEY_CURRENT_USER`;
  - `HKLM = HKEY_LOCAL_MACHINE`;
  - `HKU = HKEY_USERS`.
- Поле `subkey` содержит имя ветки реестра, относительно `reg-root`, или ссылку на строку секции `String`.
- В поле `value-entry-name` содержится имя ключа, создаваемого в реестре в ветке `reg-root\subkey`. Может быть или строкой, заключенной в кавычки, или ссылкой на значение из секции `String`.
- Поле `flags` — необязательное, обозначает тип и свойства добавляемого значения. Значение представляет собой маску из констант `FLG_xxx` (табл. 5.7). Мы приведем самые существенные из этих констант.

Таблица 5.7. *Таблица флагов секции `AddReg (DelReg)`*

Флаг секции `AddReg (DelReg)`	Описание флага
$00000001 `FLG_ADDREG_BINVALUETYPE`	Создать как "Бинарные данные"
$00000010 `FLG_ADDREG_KEYONLY`	Создать `subkey`, но игнорировать `value-entry-name` и его значение
$00000020 `FLG_ADDREG_OVERWRITEONLY`	Переписать значение `value-entry-name`, если оно существует. Если такого значения еще нет, не создавать
$00001000 `FLG_ADDREG_64BITKEY`	Создание 64-битового значения (Windows XP и выше)
$00000000 `FLG_ADDREG_TYPE_SZ`	Создание значения типа `REG_SZ`. Этот флаг является значением по умолчанию, если ключ `flags` не указан

Таблица 5.7 *(окончание)*

Флаг секции AddReg (DelReg)	Описание флага
$00010000 FLG_ADDREG_TYPE_MULTI_SZ	Создание значения типа REG_MULTI_SZ. Для таких значений не требуется код NULL для завершения строки
$00020000 FLG_ADDREG_TYPE_EXPAND_SZ	Добавление значения типа REG_EXPAND_SZ
$00010001 FLG_ADDREG_TYPE_DWORD	Добавление значения типа REG_DWORD
$00020001 FLG_ADDREG_TYPE_NONE	Добавление значения типа REG_NONE (только Windows 2000)

Поле `value` содержит значение ключа реестра `reg-root\subkey\value-enntry-name` и должно соответствовать типу, указанному флагами `flags`.

```
; пример из файла TAPI.INF
[add.reg]
HKU,Software\Microsoft\Windows\CurrentVersion\Telephony,,,
HKU,Software\Microsoft\Windows\CurrentVersion\Telephony\HandoffPriorities,
"RequestMakeCall",2,"DIALER.EXE"
HKLM,Software\Microsoft\Windows\CurrentVersion\Telephony\Locations,
"NextID",3,01,00,00,00
HKLM,Software\Microsoft\Windows\CurrentVersion\RunOnce,TapiSetup2,,
"tapiupr.exe"
```

5.9.6. Секция *xxx.LogConfig*

Секция `xxx.LogConfig` описывает системные ресурсы, требуемые драйвером. Ссылка на имя этой секции дается из секции описания модели *(см. разд. 5.9.4)*.

В этой секции могут указываться ключи (мы снова выбрали наиболее используемые ключи):

- `DMAConfig` — описание ресурсов DMA;
- `IOConfig` — описание ресурсов портов;
- `MemConfig` — описание ресурсов памяти;
- `IRQConfig` — описание ресурсов IRQ.

Пример:

```
[MyDevice1.LogConfig]
DMAConfig=0,1
IOConfig=3bc-3be(3ff::),378-37a(3ff::),278-27a(3ff::)
[MyDevice2.LogConfig]
IOConfig=8@100-3ff%fff8(3ff::)
MemConfig=4000@C8000-EFFFF%FFFFC000
[MyDevice3.LogConfig]
IOConfig=110-11F(3FF::),130-13F(3FF::),150-15F(3FF::)
IRQConfig=3,4
MemConfig=4000@C0000-DFFFF%FFFFC000
```

Подробное описание всех ключей можно найти в MSDN.

5.9.7. Секция *xxx.CopyFiles*

Секция `xxx.CopyFiles` описывает операции над файлами, которые требуются для работы драйвера. Ссылка на имя этой секции дается из секции описания модели *(см. разд. 5.9.4)*.

Секция содержит или имена файлов, или ссылку на секцию с именами файлов.

Каждое имя файла задается строкой вида:

```
Dest-file-name[,source-file-name][,temp-file-name][,flag]
```

- Поле `dest-file-name` задает имя конечного файла. Если секция `source-file-name` не указана, то задает и имя исходного файла.
- Поле `source-file-name` задает имя исходного файла. Если для операции копирования файлов имена исходного и конечного файлов одинаковые, то это поле можно пропустить.
- Поле `temp-file-name` задает имя промежуточного файла, которое будет использовано, если конечный файл занят в данный момент времени. Используется только в Windows 95/98/ME, т. к. Windows NT и выше создают временные файлы автоматически. Операция над файлом будет выполнена при перезагрузке Windows.
- Поле `flag` — необязательное поле, представленное в шестнадцатеричном или десятичном виде. Является маской одного или нескольких системных флагов `COPYFLG_xxx` (табл. 5.8).

Таблица 5.8. *Таблица флагов секции* `CopyFiles`

Флаги секции CopyFiles	Описание флагов
$00000400 COPYFLG_REPLACEONLY	Копировать файл только в том случае, если он уже существует в конечной директории
$00000800 COPYFLG_NODECOMP	Копировать исходный файл без распаковки
$00000008 COPYFLG_FORCE_FILE_IN_USE	Копировать файл в файл temp и использовать его только после рестарта системы (эмуляция поведения занятости файла)
$00000010 COPYFLG_NO_OVERWRITE	Не заменять файл, если он уже существует. Этот флаг не может использоваться с другими флагами
$00001000 COPYFLG_REPLACE_BOOT_FILE	После выполнения операции требовать перезагрузки системы
$00000020 COPYFLG_NO_VERSION_DIALOG	Не переписывать файл в конечной директории, если существующий файл более новый, чем копируемый
$00000004 COPYFLG_NOVERSIONCHECK	Игнорировать версию файла. Переписывать конечный файл независимо от его версии
$00000002 COPYFLG_NOSKIP	Не позволять пользователю отказаться от копирования файла

5.9.8. Секция *Strings*

Секция `Strings` используется для создания INF-файлов на нескольких языках. Другие секции могут ссылаться на ключи этой секции с помощью знаков "%". Строка, заключенная в знаки "%", означает подстановку значения переменной из этой секции.

Значения полей этой секции могут быть обычными строками или строками, заключенными в кавычки, если они содержат существенные пробелы до или после строки.

Для NT-платформы секция `Strings` может содержать индекс языка, для которого указываются строки. Индекс представляет собой объединения константы `LANG_xxx` и `SUBLANG_xxx`. Например,

```
[Strings]        ; Обычный набор строк
DiskName="My Excellent Software"
LocaleSubDir="English"
```

```
[Strings.0407]   ; Набор строк для немецкого языка (0407)
DiskName="Meine ausgezeichnete Software"
LocaleSubDir="German"
[Strings.0419]   ; набор строк для русского языка (0419)
[Strings.0422]   ; набор строк для украинского языка (0422)
```

5.9.9. Связи секций

Собрав все перекрестные ссылки, мы получим картину связей между основными секциями. Итак, секция `Version` является обязательной в INF-файле. Секция `Manufacturer` содержит ссылку на секцию описания модели, которая в свою очередь содержит ссылки на секции для конкретных устройств с заданными серийными номерами. Секция описания устройства содержит ссылки на секции `CopyFiles`, `AddReg`, `DelReg` и `LogConfig`. Кроме того, любая секция может содержать ссылку на строки из секции `Strings` (рис. 5.3).

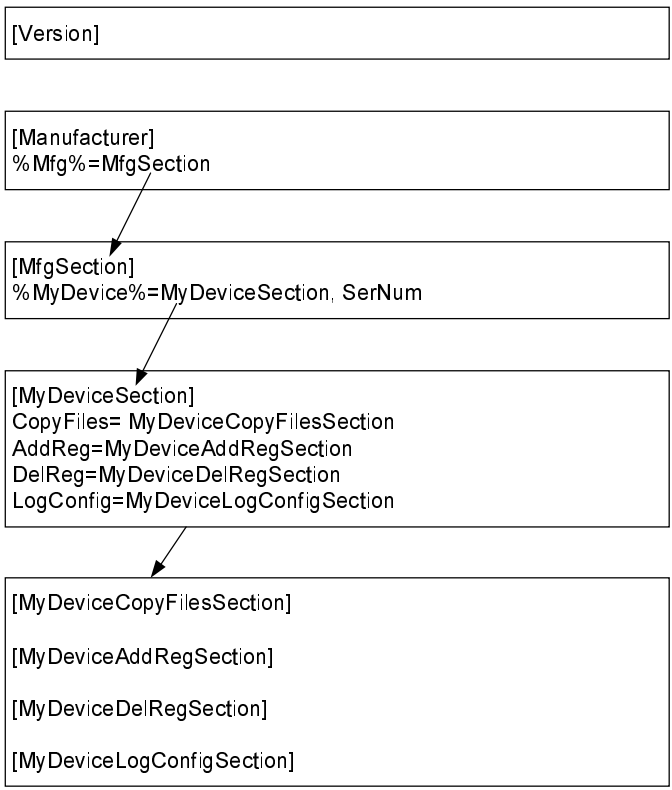

Рис. 5.3. Связи основных секций INF-файла

ЧАСТЬ II

ПРАКТИКА ПРОГРАММИРОВАНИЯ

Глава 6

Использование сервисов BIOS

Быстро и качественно сделаем любую халтуру.

Объявление

В этой главе мы продемонстрируем простейший пример приема и передачи данных. Мы создадим программы приемника и передатчика и научимся передавать данные.

6.1. Подготовка

Последовательная связь подразумевает наличие, как минимум, приемника и передатчика. Конечно, можно использовать два компьютера, но мы поступим проще. Так как Windows 2000[1] не запрещает прямого обращения к портам из DOS-программ, то мы будем использовать две консоли (например, два запущенных Norton Commander или Dos Navigator). В первом окне мы будем работать с программой одного абонента, использующего порт COM1, а во втором — с программой второго абонента, использующего порт COM2. Порт COM1 мы соединим с портом COM2 с помощью нуль-модемного кабеля.

Итак, нам потребуются:

- персональный компьютер с двумя COM-портами;
- операционная система Windows 2000 или Windows 95/98;
- желательно наличие файловой оболочки типа Norton Commander;
- компиляторы Pascal (BPC, версии выше 5 для DOS, и Delphi, версии выше 3 для Windows);
- нуль-модемный кабель для соединения портов (см. рис. 1.3).

Соединим порт COM1 с COM2 и попробуем передать данные с одного порта на другой.

---

[1] Windows 95/98 также разрешает доступ к портам из DOS-программ. Не следует путать консольные программы и DOS-программы.

В Windows 2000 возможна маленькая неприятность: открыв две DOS-консоли, нам придется переключаться между ними, т. к. Windows может останавливать выполнение DOS-программ, если они не активны. Таким образом, отправив байт с одной консоли, нам нужно переключиться на вторую и получить его. Конечно, можно настроить ярлыки для запуска наших программ, но для проведения таких простых тестов это лишняя забота.

6.2. Первая программа последовательной связи

Функции, которыми мы будем пользоваться, подробно описаны в третьей части книги, в *гл. 18*.

Независимо от того, будем мы передавать или принимать данные, необходимо сначала инициализировать порт. Инициализация порта подразумевает задание следующих параметров:

- скорость передачи (точнее делитель);
- число битов данных;
- контроль четности;
- число стоп-битов.

Передачу данных мы будем производить через порт COM1, а принимать данные — с порта COM2. Подробности можно найти в комментариях к кодам листингов 6.1 и 6.2.

Листинг 6.1. Программа передачи данных с помощью INT14H

```
Uses Crt;

Var
 B : Byte;

Begin
 WriteLn('Передача байта через сервис BIOS');
 WriteLn('Порт COM1');

 Asm
  Mov Ah, 0   {код функции — инициализация порта}
  Mov Dx, 0   {номер порта — COM1}
  Mov Al, 227 {11100011b=9600, нет четности, 1 стоп-бит}
              {8 бит данных}
```

```
  INT 14H
 End;

 While not KeyPressed do begin
  Write('Введите байт:'); ReadLn(B);
  Asm
   Mov Ah, 1 {код функции — передача байта}
   Mov Dx, 0 {номер порта — COM1}
   Mov Al, B {передаваемый байт }
   INT 14H
  End;
 End;
End.
```

Листинг 6.2. Программа чтения данных с помощью INT14H

```
Uses Crt;

Var S, B : Byte;

Begin
 WriteLn('Чтение данных с порта COM2');

 Asm
  Mov Ah, 0   {номер функции — инициализация}
  Mov Dx, 1   {номер порта — COM2}
  Mov Al, 227 {11100011b=9600, нет четности, 1 стоп-бит}
              {8 бит данных}
  INT 14H
 End;

 While not KeyPressed do begin
  Asm
   Mov Ah, 3  {номер функции - статус порта }
   Mov Dx, 1  {номер порта - COM2}
   INT 14H
   Mov S, AH
  End;
```

```
  If (S and 1) = 1 then begin {нулевой бит - есть данные }
   Asm
    Mov Ah, 2 {номер функции - чтение данных}
    Mov Dx, 1 {номер порта - COM2}
    INT 14H
    Mov B, Al {принятый байт}
    Mov S, Ah {код ошибки}
   End;

   If S = 0 then begin {нет ошибок}
    WriteLn(B); {отобразим полученный байт}
   End;

  End;

 End;

 While KeyPressed do {очистим клавиатурный буфер}
  ReadKey;
End.
```

Из первой консоли запустим программу чтения данных, а из второй — программу передачи. После ввода байта во второй консоли мы должны увидеть его в первой.

Обратите внимание на задание номера порта. В первой программе в регистр `Dx` записывается 0, означающий выбор COM1, а во второй программе записывается 1, означающая выбор COM2.

Больше ничего сложного в нашей первой программе нет. К сожалению, простота реализации оборачивается другой стороной: обмен данными возможен только на скоростях до 9600 бит/с.

6.3. Другие функции BIOS

Как видно из листинга программ 6.1 и 6.2, мы воспользовались тремя функциями сервиса `INT14H`: функциями `00`, `01` и `02`. Собственно эти три функции и составляют практически весь набор функциональности `INT14H`. Для System/2 существует еще две функции с номерами `04` и `05`. Их описание можно найти в *гл. 18*.

Кроме сервиса INT14H, BIOS предоставляет еще четыре функции для последовательного ввода/вывода. Это функции 03H, 04H, 3FH и 40H прерывания INT21H. Они еще более ограничены, чем INT14H — они работают только с COM1 и только с текущими настройками порта.

Это редко используемый способ работы с портом, и мы не будем рассматривать его в нашей книге. Заметим только, что при необходимости работы с COM2 нужно переставить базовые адреса в области BIOS, например, как показано в листинге 6.3.

Листинг 6.3. Обмен базовых адресов COM1 и COM2

```
;  обмен базовых адресов для COM1 и COM2
   MOV  AX,40H          ;ES указывает на область данных BIOS
   MOV  ES,AX           ;
   MOV  DX,ES:[0]       ;помещаем 1-й базовый адрес в DX
   MOV  AX,ES:[2]       ;помещаем 2-й базовый адрес в AX
   MOV  ES:[0],AX       ;обменниваем адреса
   MOV  ES:[2],DX       ;
```

Описание этих функций можно найти в третьей части книги, в *гл. 18*.

Глава 7

Прямое программирование портов

Вопрос: Как программировать порты на низком уровне?
Ответ: Поставьте компьютер на пол и программируйте.

Из форума по программированию

К сожалению, сервис BIOS весьма ограничен: мы не можем использовать скорость обмена выше 9600 бит/с, а ведь максимальная допустимая скорость самой UART-микросхемы составляет 115 200 бит/с. Избавиться от ограничений позволит прямое программирование портов, без использования функций BIOS. С помощью программ, описанных в этой главе, мы организуем последовательную связь на скорости 28 800 бит/с.

7.1. Коммуникационные порты

Коммуникационные порты занимают в пространстве ввода/вывода по восемь смежных 8-битовых регистров и располагаются по стандартным базовым адресам. *Базовый адрес* — это 2-байтовый адрес порта, который является младшим из группы адресов портов, дающих доступ к UART.

Раньше на всех машинах COM1 имел базовый адрес 3F8H, а COM2 — 2F8H. Новые версии BIOS позволяют задавать этот адрес самостоятельно.

В процессе начального тестирования POST BIOS проверяет наличие последовательных портов по стандартным адресам и помещает базовые адреса портов в ячейки данных BIOS. Нулевое значение адреса является признаком отсутствия порта с данным номером.

Обнаруженные порты инициализируются на скорость обмена 2400 бит/с, 7 бит данных с контролем четности и 1 стоп-бит. Управляющие сигналы интерфейса DTR и RTS переводятся в состояние "выключено" (положительное напряжение). Тайм-аут порта по умолчанию устанавливается в одну секунду.

Полное описание портов можно найти в *гл. 19*.

7.2. Программа определения наличия COM-портов

Как уже говорилось, при инициализации системы BIOS заносит базовые адреса в специальную область данных, находящуюся по адресу 0:40.

Программа, показанная в листинге 7.1, читает таблицу базовых адресов и отображает наличие последовательных портов компьютера.

Листинг 7.1. Отображение наличия COM-портов и их базовых адресов

```
{ Возвращает базовый адрес порта с номером PortIndex }
Function GetBaseAdr(PortIndex : Byte) : Word;
Var LowAdr : Word;
Begin
 { вычисляем младшую часть адреса в таблице }
 LowAdr := (PortIndex-1)*2;
 { получаем базовый адрес порта из таблицы }
 GetBaseAdr:= MemW[$0040:LowAdr];
End;

Var
 PortIndex : Byte;
 BaseAdr   : Word;

Begin
 { Опрос базовых адресов 4 портов }
 For PortIndex:= 1 to 4 do begin
  { Получить базовый адрес}
  BaseAdr:= GetBaseAdr(PortIndex);

  { Анализируем базовый адрес }
  If BaseAdr = 0 then
   WriteLn('COM', PortIndex,' не обнаружен')
  Else
   WriteLn('Базовый адрес COM',PortIndex,' равен ', Long2Hex(BaseAdr));

 End;
End.
```

Важно!

Мы будем часто пользоваться такими функциями, как Long2Hex, Byte2Hex, и др. Для экономии места все эти функции удалены из листингов и приводятся в *прил. 2*.

Результат запуска этой программы может быть, например, такой:

```
E:\SOURCE\4_1>ports.exe
Базовый адрес COM1 равен 03F8
Базовый адрес COM2 равен 02F8
Базовый адрес COM3 равен 03E8
COM4 не обнаружен
```

Новые версии BIOS позволяют изменять базовые адреса портов, поэтому базовые адреса должны считываться каждый раз при запуске программы. Использование констант в качестве базовых адресов может привести к неработоспособности программы, если она будет запущена на компьютере с нестандартно сконфигурированными адресами.

7.3. Обходим ограничения BIOS

Итак, мы умеем определить базовый адрес нужного нам порта. Попробуем создать программу для приема и передачи данных, используя непосредственное обращение к порту.

Будем рассматривать некоторый порт с базовым адресом BaseAdr. Начнем с инициализации порта. Для задания конфигурации порта нам надо:

- установить специальный бит DLAB, открывающий доступ к делителю скорости порта;
- записать младшую часть делителя скорости в порт [BaseAdr+0];
- записать старшую часть делителя скорости в порт [BaseAdr+1];
- сбросить бит DLAB;
- записать конфигурацию в порт [BaseAdr+3].

Установка бита DLAB производится записью числа $80 в порт [BaseAdr+3], т. е. установкой старшего бита этого порта. Заметьте, что запись конфигурации производится в тот же порт, а значит, два последних шага можно объединить, просто записав конфигурацию в порт [BaseAdr+3] (листинг 7.2).

Листинг 7.2. Инициализация порта, имеющего базовый адрес BaseAdr

```
Var { Будет хранить базовый адрес порта }
 BaseAdr : Word;
```

```
{Инициализация порта с номером ComIndex                    }
{ ComIndex — номер порта                                   }
{ Speed    — скорость в бод                                }
{ Params   — конфигурация порта, согласно формату битов LCR }
{Возвращает False, если порт не обнаружен                  }
 Function InitCOM(ComIndex : Byte; Speed : Longint; Params : Byte) :
Boolean;
 var Freq : Longint;
 begin
  InitCOM:= True;

  Freq := 115200 div Speed;

  { Вычисляем базовый адрес порта }
  BaseAdr:= GetBaseAdr(ComIndex);
  If BaseAdr = 0 then begin
   WriteLn('Порт ', ComIndex,' не обнаружен!');
   InitCOM:= False; {вернем ошибку}
   Exit;
  End;
  {Адресуем делитель порта с помощью установки DLAB=1}
  Port[BaseAdr+3]:= $80;
  {Устанавливаем младшую часть делителя}
  Port[BaseAdr+0]:= Freq and $00FF;
  {Устанавливаем старшую часть делителя}
  Port[BaseAdr+1]:= Freq shr 8;
  {Сбрасываем DLAB и прописываем конфигурацию}
  Port[BaseAdr+3]:= Params;
 end;
```

Мы не стали усложнять программу константами настройки порта, а просто передаем параметр `Param`, содержащий нужные нам биты настройки. Значение этого параметра совпадает с форматом порта `LCR`, описанного в *гл. 19*. Например, значение 3 (битовый код <0000 0011>) подразумевает настройку порта на следующие параметры:

- контроль четности отсутствует;
- один стоп-бит;
- 8 бит данных.

Вычисление делителя производится по формуле:

Делитель = 115 200 / *Скорость*

Смысл этой формулы объясняется в справочной части, в *гл. 19*.

Чтение и передача данных очень просты и выполняются чтением и записью порта [BaseAdr+0] при сброшенном бите DLAB, как показано в листинге 7.3.

Листинг 7.3. Чтение и передача данных в порт с базовым адресом BaseAdr

```
Function ReadCOM : Byte; assembler;
 Asm
  Mov Dx, BaseAdr
  In  Al, Dx
 End;

Procedure WriteCOM(B : Byte); assembler;
 Asm
  Mov Dx, BaseAdr
  Mov Al, B
  Out Dx, Al
 End;
```

Недостаток вышеприведенного кода очевиден: мы не обрабатываем никаких ошибок. По результатам вызова ReadCom мы не можем определить, действительно ли мы прочитали байт, и прочитан ли байт без ошибок.

Чтобы исправить этот недостаток, сделаем "оболочки" для функций чтения и записи, добавив в них параметры ожидания и обработку ошибок. Для простоты мы будем задавать время ожидания не в реальных величинах, а в "циклах ожидания". Чаще всего этого достаточно.

Готовность данных для чтения нам дает бит DR, а готовность передатчика определяется по биту THRE порта [BaseAdr+5]. В листинге 7.4 приведен новый код наших функций.

Листинг 7.4. Чтение и передача данных в порт с базовым адресом BaseAdr с определением тайм-аутов и кодов ошибок

```
{ Возвращает бит DR порта LSR }
Function GetDR : Boolean; assembler;
Asm
  Mov Dx, BaseAdr
  Add Dx, 5
```

```
  In  Al, Dx
  And Al, 1
End;

{Возвращает биты ошибок порта LSR          }
{ Бит 4 — обрыв линии                      }
{ Бит 3 — ошибка кадра (неверный стоп-бит) }
{ Бит 2 — ошибка четности                  }
{ Бит 1 — переполнение (потеря символа)    }
Function GetErr : Byte; assembler;
Asm
 Mov Dx, BaseAdr
 Add Dx, 5
 In  Al, Dx
 And Al, 1EH {0001 1110b}
End;

{Возвращает бит THRE порта LSR }
Function GetSR : Boolean; assembler;
Asm
  Mov Dx, BaseAdr
  Add Dx, 5
  In  Al, Dx
  And Al, 20H
End;

{Чтение байта с порта с тайм-аутом                    }
{ B    — прочитанный байт                             }
{ Wait — время ожидания байта                         }
{Возвращает True, если байт реально прочитан          }
Function ReadData(var B : Byte; Wait : LongInt): Boolean;
Var Result : Boolean; w : LongInt;
Begin
 w:= 0; Result:= False;
 For w:= 1 to Wait do begin {ожидание...}
  If GetDR then begin {проверяем доступность данных }
   B:= ReadCOM; {читаем байт с порта}
   Result:= (GetErr = 0); {проверяем код ошибки}
```

```
   Break;
  End;
 End;
 ReadData:= Result;
End;

{Запись байта в порта с тайм-аутом                          }
{ B    — передаваемый байт                                  }
{ Wait — время ожидания готовности передатчика              }
{Возвращает True, если байт передан          }
Function WriteData(B : Byte; Wait : LongInt): Boolean;
Var Result : Boolean; w : LongInt;
Begin
 w:= 0; Result:= False;
 For w:= 1 to Wait do begin {ожидание...}
  If GetSR then begin {передатчик готов?}
   WriteCOM(B); {передаем байт}
   Result:= True;
   Break;
  End;
 End;
 WriteData:= Result;
End;
```

Если прочитанный байт должен иметь строго определенное значение (например, начало или завершение посылки), удобно использовать функцию, возвращающую True только при успешном чтении нужного байта (листинг 7.5).

Листинг 7.5. Чтение байта с проверкой

```
{Вернет True, если байт прочитан успешно и равен ChB}
Function CheckReadData(ChB : Byte; Wait : LongInt) : Boolean;
 Var B : Byte;
 Begin
  CheckReadData:= False;
  If ReadData(B, Wait) then begin
   CheckReadData:= (B = ChB);
  End;
 End;
```

Все эти функции мы соберем в один модуль RS232Dos, которым будем пользоваться в следующем разделе. Исходный код получившегося модуля можно найти на прилагаемом компакт-диске.

7.4. Чтение и запись с помощью модуля *RS232DOS*

Модифицируем наши программы приемника и передатчика таким образом, чтобы вместо сервисов BIOS они использовали наш модуль RS232DOS с прямым программированием портов. Для этого достаточно заменить функции инициализации, чтения и записи данных. Результат показан в листингах 7.6 и 7.7.

Листинг 7.6. Чтение данных с порта COM2 с помощью RS232DOS

```
Uses Crt, RS232Dos;

Var B : Byte;

Begin
 WriteLn('Читаем данные с COM2');
 InitCOM(2, 28800, 3);

 While not KeyPressed do begin
  If ReadData(B, 2000) then begin
   WriteLn(B);
  End;
 End;

 While KeyPressed do
  ReadKey;
End.
```

Листинг 7.7. Передача данных с порта COM1 с помощью RS232DOS

```
Uses Crt, RS232Dos;

Var
 B : Byte;
```

```
Begin
 WriteLn('Запись данных в COM1');
 InitCOM(1, 28800, 3);

 While not KeyPressed do begin
  Write('Введите байт для посылки:'); ReadLn(B);
  WriteData(B, 2000);
 End;

 While KeyPressed do
  ReadKey;
End.
```

Для демонстрации преимуществ прямого программирования перед сервисами BIOS мы установили скорость обмена 28 800 бит/с. В первой консоли мы запустим передатчик и введем значение байта для передачи, а во второй консоли запустим программу чтения. Переданный байт должен отобразиться во второй консоли. Заметим, что скорость обмена можно смело поднять до 115 200 бит/с.

Глава 8

Использование обработки прерываний

Чем дольше вы ждете, тем больше вероятность,
что вы ждете не там.

Прямое обращение к портам имеет лишь один недостаток — для чтения данных необходимо в цикле опрашивать порт, не позволяя программе заниматься другими делами. Для устранения этого недостатка мы научимся обрабатывать прерывания.

8.1. Прерывания

Коммуникационный адаптер может вызывать специальное прерывание при возникновении некоторых условий. Условия вызова прерывания определяются значением порта `IER` (подробное описание порта можно найти в третьей части нашей книги, в *гл. 19*). Возможна настройка следующих причин прерываний:

- прерывание по статусу модема (CTS, DSR, RI, RLSD);
- прерывание по обрыву или ошибке линии;
- прерывание, когда буфер передачи пуст (завершение передачи);
- прерывание при готовности принимаемых данных (по приему символа);
- (в режиме FIFO) прерывание по тайм-ауту.

При возникновении прерывания его причину можно определить по значению регистра `IIR`.

Прерывание, вызываемое коммуникационным портом, является аппаратным. В табл. 8.1 приведены номера аппаратных прерываний (IRQ) и соответствующие им векторы прерываний (INT).

Таблица 8.1. *Таблица прерываний для коммуникационных портов*

Порт	IRQ	Вектор прерывания
COM1	IRQ4 (или 11)	`INT 0CH`
COM2	IRQ3 (или 10)	`INT 0BH`
COM3	IRQ4 (или 11)	`INT 0CH`
COM4	IRQ3 (или 10)	`INT 0BH`

В нашем примере мы будем использовать прерывание по приему символа. Для обработки прерывания следует выполнить следующие действия:

1. Запретить маскируемые прерывания.
2. Запретить аппаратные прерывания нужного IRQ.
3. Заменить нужный вектор прерывания своим обработчиком.
4. Разрешить аппаратные прерывания.
5. Задать конфигурацию порта.
6. разрешить маскируемые прерывания.

Перед завершением программы следует выполнить эти действия в "обратном" порядке:

1. Если производится передача данных, дождаться ее завершения.
2. Запретить прерывания коммуникационного порта.
3. Вернуть старый обработчик прерывания.
4. Запретить маскируемые прерывания.
5. Запретить прерывания нужного IRQ.
6. Разрешить маскируемые прерывания.

Аппаратные прерывания требуют дополнительных действий для их обработки. При возникновении прерывания следует:

1. Убедиться, что прерывание вызвано приходом байта, а не другими причинами.
2. Отдать управление предыдущему обработчику прерываний.
3. Прочитав статус порта, убедиться, что байт был принят без ошибок.
4. При отсутствии ошибок считать байт с порта.
5. Послать уведомление контроллеру прерываний о завершении обработки аппаратного прерывания.
6. Вызвать `callback`-функцию пользователя.

Подробное описание необходимых нам портов контроллера прерываний приводится в *гл. 19*.

8.2. Модуль *RS232Int* для работы с прерываниями

Для работы с прерываниями COM-порта можно использовать модуль, приведенный в листинге 8.1. Основные функции и принцип его работы легко понять из комментариев к коду.

Еще раз подчеркнем необходимость передачи специальных команд контроллеру прерываний, которые сообщают ему о корректном завершении обработки аппаратного прерывания. Если этого не сделать, программа, скорее всего, зависнет после второго или третьего вызова прерывания.

Листинг 8.1. Модуль RS232Int для работы с прерываниями порта

```
Unit RS232Int;

INTERFACE
uses crt;

{Тип для callback-функции, вызываемой при получении нового байта.}
Type
 TCallBackRS = Procedure(Status : Byte; b : Byte);

{              ОСНОВНЫЕ ФУНКЦИИ                  }

{Инициализация порта                             }
{  ComIndex — номер порта                        }
{  Proc     — адрес callback-функции или nil     }
{  Speed    — скорость работы порта              }
{  ComPar   — настройки порта                    }
Procedure InitRS232(ComIndex : Byte; Proc : TCallBackRS;
                    Speed : Longint; COMPar : Byte);
{Закрытие порта, восстановление обработчиков INT}
procedure CloseRS232;
{Передача байта в порт }
procedure OutByteToRS232(b:Byte);
```

```
{              ОТЛАДОЧНЫЕ ФУНКЦИИ                }
{Счетчик вызовов прерывания}
function  GetIntRSCount : Longint;

{           ДОПОЛНИТЕЛЬНЫЕ ФУНКЦИИ              }
{Позволяет задавать callback-адрес вручную}
procedure SetCallBackRS(Proc : TCallBackRS);
{Разрешает IRQ-прерывания}
Procedure EnableIRQ;
{Запрещает IRQ-прерывания}
Procedure DisableIRQ;
{Очистка порта. Просто холостое чтение}
Procedure ClearRSPort;
{Задание параметров порта}
Procedure SetPortState(Speed : Longint; COMPar : Byte);

IMPLEMENTATION

Uses Dos;

Var { старый адрес Int 0Ch/0BH }
 SerOldVec: Pointer;

Const { служебные переменные }
 Save_ds  : Word = 0;
 Int_sts  : Byte = 0;
 Count    : Word = 0;
 Src_ptr  : Word = 0;
 AddrPr   : Word = 0;
 tmp      : Word = 0;

Var
 BaseAdr  : Word;
 EOI      : Byte;
 IRQ      : Byte;
 IRQMask  : Byte;
 INTVect  : Byte;
```

```
Const { число входов в Int }
 CountInt : Longint = 0;

Const { адрес callback-функции }
 CallBackPtr : TCallBackRS = nil;

{ Подпрограмма обработки прерываний от COM-порта }
Procedure Ser_Int; interrupt;
Var B : Byte; COMStatus : Byte;
Begin
  Inc(CountInt); {счетчик вхождений в INT }

  B:= 0;
  Asm
   mov  dx, BaseAdr { регистр идентификации прерываний }
   add  dx, 2       { BaseAdr+2                        }
   in   al,dx
   mov  Int_Sts,al { сохраним его содержимое       }
   test al,1       { есть отложенные прерывания?   }
   jz   @Is_Int    { да                            }
   pushf           { нет, передаем управление      }
   call SerOldVec  { старому обработчику Int       }
@Is_Int:
   mov  al,EOI     { послать EOI для IRQ           }
   out  20h,al     { в 1 контроллер прерываний   }
   test Int_Sts,4  { прерывание по приему?         }
   jnz  @Read_Char { да                            }
   mov  B, 1
@Read_Char:
   inc  B
  End;

  if (B <> 2) then begin

   Asm
    mov  dx, BaseAdr { регистр состояния линии }
    add  dx, 5       { BaseAdr + 5}
    in   al,dx
    and  al, 1EH
```

```
    mov  COMStatus,al { COMStatus<>0, если была ошибка приема }
   End;

   Asm
    mov  dx, BaseAdr { регистр данных }
    in   al, dx      { вводим символ  }
    mov  B , al
   End;

   {Вызов callback, если она задана}
   If @CallBackPtr<>nil then CallBackPtr(COMStatus, B);
  End;
End;

{Запрещение IRQ прерываний}
Procedure DisableIRQ;
Begin
 Asm
  in   al,21h     { IMR 1-го контроллера прерываний  }
  or   al,IRQMask { запретить прерывание IRQ от COM }
  out  21h,al
 End;
End;

{Разрешение IRQ прерываний}
Procedure EnableIRQ;
var NIRQMask : Byte;
Begin
 NIRQMask:= not IRQMask;
 Asm
  in   al,21h    { IMR 1-го контроллера прерываний }
  and  al,NIRQMask
  out  21h,al    { разрешить прерывания от COM }
 End;
End;

{ Возвращает базовый адрес порта с номером PortIndex }
Function GetBaseAdr(PortIndex : Byte) : Word;
Var LowAdr : Word;
```

```
Begin
 { вычисляем младшую часть адреса в таблице }
 LowAdr := (PortIndex-1)*2;
 { получаем базовый адрес порта из таблицы }
 GetBaseAdr:= MemW[$0040:LowAdr];
End;

{Чистка порта. Холостое чтение порта.}
Procedure ClearRSPort;
Begin
 Asm
  mov  dx, BaseAdr { регистр данных         }
  in   al, dx      { сбросить буфер приема  }
 End;
End;

{Установка скорости обмена и характеристик порта }
Procedure SetPortState(Speed : Longint; COMPar : Byte);
Var Freq : Word; FreqH, FreqL : Byte;
Begin
 Freq:= 115200 div Speed;
 FreqH:= Freq shr 8;
 FreqL:= Freq and $00FF;

 Asm
  mov  dx, BaseAdr { регистр управления линией }
  add  dx, 3       { BaseAdr + 3}
  in   al,dx
  or   al,80h      { установить бит DLAB }
  out  dx,al

  mov  dx, BaseAdr
  mov  al, FreqL
  out  dx,al      { младший байт скорости }

  inc  dx          { BaseAdr+1 }
  mov  al,FreqH
```

```
  out  dx,al       { старший байт скорости }
  mov  dx, BaseAdr { регистр управления линией }
  add  dx, 3       { BaseAdr + 3              }
  mov  al, COMPar
  out  dx,al

  mov  dx, BaseAdr { регистр разрешения прерываний  }
  add  dx, 1       { BaseAdr+1                      }
  mov  al, 1       { разрешить прерывания по приему }
  out  dx, al
 End;
End;

{ Инициализация COM-порта }
Procedure InitRS232(ComIndex : Byte; Proc : TCallBackRS;
                    Speed : Longint; COMPar : Byte);
Begin
 CallBackPtr:= Proc; {сохранить адрес callback}

 {Получить базовый адрес порта}
 BaseAdr:= GetBaseAdr(ComIndex);

 {Порт не найден — выход}
 If BaseAdr = 0 then Exit;

 {Инициализация переменных, зависящих от номера COM}
 Case ComIndex of {Номер IRQ порта}
  1: IRQ:= 4; {для COM1/IRQ4}
  2: IRQ:= 3; {для COM2/IRQ3}
 End;
 Case IRQ of {Номер INT вектора прерываний }
  3: INTVect:= $0B; {для IRQ3}
  4: INTVect:= $0C; {для IRQ4}
 End;
 Case IRQ of {Сигнал EOI контроллеру прерываний}
  3: EOI:= $63; {для IRQ3}
  4: EOI:= $64; {для IRQ4}
 End;
```

```
 Case IRQ of {Сигнал EOI контроллеру прерываний}
  3: IRQMask:= $08; {для IRQ3}
  4: IRQMask:= $10; {для IRQ4}
 End;

 Asm
  cli { запретить прерывания }
 End;

 {запретить IRQ прерывания }
 DisableIRQ;

 {Получить текущий обработчик INT}
 GetIntVec(INTVect, SerOldVec);
 {Установить новый обработчик INT}
 SetIntVec(INTVect, Addr(Ser_Int));

 {Разрешить IRQ прерывания}
 EnableIRQ;

 {Задание скорости обмена и характеристик порта}
 SetPortState(Speed, COMPar);

 Asm
  sti { разрешить прерывания }
 End;

 {Чистка порта}
 ClearRSPort;
End;

{ Закрытие порта. Восстановление обработчиков INT }
Procedure CloseRS232;
Begin
 Asm
@Wait_Free:
   mov  dx, BaseAdr { регистр состояния линии }
```

```
   add  dx, 5       { BaseAdr+5              }
   in   al, dx
   nop
   nop
   test al,60h     { передача окончена?          }
   jz   @Wait_Free { ждем, если нет              }
   mov  dx, BaseAdr{ регистр разрешения прерываний }
   add  dx, 1      { BaseAdr+1                   }
   mov  al,0       { запретить прерывания        }
   out  dx,al
   nop
   nop
   nop
   nop
   mov  dx, BaseAdr  { регистр управления модемом }
   add  dx,4         { BaseAdr+4}
   mov  al,00000011b { активизировать DTR и RTS }
   out  dx,al
   nop
   nop
   nop
   nop
  End;

  {Вернуть старый обработчик INT}
  SetIntVec(INTVect, SerOldVec);

  Asm
   cli            { запрет прерываний }
   in   al,21h    { читать маску прерываний }
   nop
   nop
   or   al,IRQMask { запретить IRQ прерывания }
   out  21h,al
   sti             { разрешение прерываний    }
  End;
End;
```

```
{ Подпрограмма вывода символа AL в порт.    }
{ При ошибке возвращает CF=1, иначе CF=0.   }
Procedure OutByteToRS232(b:Byte); assembler;
Asm
@Wait_Line:
  mov  dx, BaseAdr { регистр состояния линии }
  add  dx, 5       { BaseAdr+5               }
  in   al,dx
  test al,20h     { стык готов к передаче? }
  jnz  @Output    { да }
  nop
  loop @Wait_Line { нет, ждем }
  ret

@Output:
  mov  al, B
  mov  dx, BaseAdr  { регистр данных }
  out  dx,al        { вывести символ }
  nop
End;



{Доступ к счетчику вхождений в INT }
function  GetIntRSCount : Longint;
begin
 GetIntRSCount:= CountInt;
end;

{Установка callback вне функции инициализации}
procedure SetCallBackRS(Proc : TCallBackRS);
begin
 CallBackPtr:= Proc;
end;

END.
```

Для инициализации порта используется функция `InitRS232`, параметрами которой являются номер порта, скорость обмена, настройки порта, и новый

параметр — адрес callback-процедуры, которая будет вызвана при возникновении прерывания.

Функция, адрес которой передается в качестве последнего параметра, должна иметь два основных свойства. Во-первых, параметры этой функции должны строго соответствовать параметрам TCallBackRS, а, во-вторых, эта функция должна иметь спецификатор far. Пример работы с портом с помощью этого модуля мы приведем в следующем разделе.

8.3. Последовательный обмен с помощью прерываний

Благодаря модулю RS232Int работа с прерываниями порта очень проста. Достаточно вызвать функцию инициализации, передав указатель на callback-функцию (обратите внимание на спецификатор far) и параметры работы порта. Все остальное будет выполнено автоматически.

Перед завершением программы необходимо вызвать функцию CloseRS232, которая восстановит оригинальный обработчик прерывания.

Листинг 8.2 показывает пример чтения данных с использованием модуля RS232Int.

Листинг 8.2. Чтение данных с помощью обработки прерывания

```
Uses Crt, Dos, Rs232Int;

   { Эта процедура будет вызвана при получении байта }
   { Спецификатор far после процедуры обязателен, так }
   { как эта процедура будет вызываться из прерывания }
   procedure COMCallBack(Status : Byte; b : byte); far;
   begin
    {пока просто печатаем принятый байт}
    {не рекомендуется так делать в реальной программе! }
    WriteLn(b);
   End;

Begin
 {Инициализация порта и регистрация callback-процедуры}
 InitRS232(1, COMCallBack, 9600, 3);
```

```
 {Холостой цикл. Здесь можно делать любые действия!}
 {Благодаря обработке прерываний не требуется постоянного опроса порта.}
 While (True) do begin

  If KeyPressed then { выход по нажатию ESC }
   If ReadKey = #27 then Break;

 End;

{восстановление векторов прерываний, закрытие порта}
 CloseRS232;
End.
```

Несколько замечаний. Во-первых, из кода видно, что в основном цикле не производится никаких действий по обращению к порту. Это и есть основное преимущество прерываний: программа может заниматься своими делами, тогда как процедура обработки прерываний будет вызываться по мере необходимости.

Во-вторых, в нашем примере мы производим печать данных из `callback`-процедуры. В реальной программе так делать не рекомендуется. Дело в том, что аппаратные прерывания (а именно такими являются IRQ3/4) блокируют возникновение других прерываний с более низким приоритетом, а, следовательно, надо стремиться как можно быстрее завершить обработку прерывания. Функции печати слишком медленные для их использования в обработчике прерывания. Оптимально будет либо сразу обработать полученные данные, либо сохранить их в некотором буфере и обработать в основном цикле программы.

Программа передачи данных не требует обработки прерываний, но для полноты мы приведем модуль передачи данных, использующий `RS232Int` (листинг 8.3).

Листинг 8.3. Передача данных с использованием RS232Int

```
Uses Crt, Dos, Rs232Int;

Begin
 {callback нам не нужна, передаем nil вторым параметром}
 InitRS232(2, nil, 9600, 3);

 While (True) do begin
```

```
  If KeyPressed then {если нажата какая-нибудь клавиша...}
   Case ReadKey of  {какая клавиша нажата?}
    '1': OutByteToRS232($31); { послать символ <1>}
    '2': OutByteToRS232($32); { послать символ <2>}
    #27: Break; { выход по ESC }
   End;

 End;

 {восстановление векторов прерываний, закрытие порта}
 CloseRS232;
End.
```

Глава 9

Переход в Windows

Windows — это просто большой эмулятор DOS.

Итак, наш следующий шаг — переход в Windows. Windows 95/98 хотя и не приветствует прямое обращение к портам, все же допускает его. В этой главе мы создадим специальный класс (компонент) для Delphi, позволяющий нам работать с последовательными портами почти так же, как в DOS. Более того, специальный модуль, работающий с драйвером GiveIO, позволит нам осуществить прямое обращение к портам даже в Windows NT/2000. Мы расскажем о принципах работы этого драйвера и расширим его функциональность: сделаем драйвер GiveIOEx, который позволит запустить любую программу с полным доступом к портам.

9.1. Переходим из DOS в Windows

В Delphi, начиная уже со второй версии, нет оператора `port` и работать с портами так же как в DOS не получится. Но минимизировать изменения мы постараемся.

В листинге 9.1 приводится специальный класс `TPort`, который позволяет применять лишь немного отличающийся метод доступа к портам.

Этот класс предоставляет достаточно простой метод переноса программ из DOS в Windows. В *разд. 9.3* мы приведем пример, который специально создан как можно более похожим на наш пример из DOS-части.

Листинг 9.1. Класс `TPort`, обеспечивающий доступ к портам

```
unit Port;

interface

uses
  Windows, SysUtils, Classes;
```

```
type
  TPort = class(TComponent)
  private
   procedure SetByte(Index:Word;Value:Byte);
   procedure SetWord(Index:Word;Value:Word);
   procedure SetLong(Index:Word;Value:LongWord);
   function  GetByte(Index:Word):Byte;
   function  GetWord(Index:Word):Word;
   function  GetLong(Index:Word):LongWord;
  public
   {свойство для обращения к байтовым портам }
   property Byte [Index:Word] : Byte     read GetByte
                                         write SetByte; default;
   {свойство для обращения к двухбайтовым портам }
   property Word [Index:Word] : Word     read GetWord
                                         write SetWord;
   {свойство для обращения к четырехбайтовым портам }
   property DWord[Index:Word] : LongWord read GetLong
                                         write SetLong;
  end;

procedure Register;

implementation

{Регистрация компонента в палитре компонентов}
procedure Register;
begin
  RegisterComponents('System', [TPort]);
end;

{Чтение байта из порта}
function TPort.GetByte(Index: Word): Byte;
begin
 asm
  mov dx,Index
  in al,dx
  mov Result,al
 end;
end;
```

```
{Чтение слова из порта}
function TPort.GetWord(Index: Word): Word;
begin
 asm
  mov dx,Index
  in ax,dx
  mov Result,ax
 end;
end;

{Чтение двойного слова из порта}
function TPort.GetLong(Index: Word): LongWord;
begin
 asm
  mov dx,Index
  in eax,dx
  mov Result,eax
 end;
end;

{Запись байта в порт}
procedure TPort.SetByte(Index: Word; Value: Byte);
begin
 asm
  mov dx,index
  mov al,Value
  out dx,al
 end;
end;

{Запись слова в порт}
procedure TPort.SetWord(Index, Value: Word);
begin
 asm
  mov dx,Index
  mov ax,Value
  out dx,ax
 end;
end;
```

```
{Запись двойного слова в порт}
procedure TPort.SetLong(Index: Word; Value: LongWord);
begin
 asm
  mov dx,Index
  mov eax,Value
  out dx,eax
 end;
end;

end.
```

Для использования этого компонента его необходимо предварительно установить, выбрав в меню **Component** пункт **Install Component** (для Delphi версии 6). После этого надо выбрать компонент `TPort` на вкладке **System** и добавить его на форму. По умолчанию новый компонент получит имя `Port1`. При необходимости вы можете заменить его на более понятное, например, просто `Port`. Обращение к портам с помощью компонента `TPort` выглядит лишь немного сложнее, чем в DOS (листинг 9.2).

Листинг 9.2. Обращение к портам с помощью компонента TPort

```
Port.Byte[$111]:= $80;       {обращение к байтовым портам}
Port.Word[$111]:= $FFFF;     {обращение к двухбайтовым портам}
Port.DWord[$111]:= $1111FFFF; {обращение к четырехбайтовым портам}

{Благодаря спецификатору default к байтам можно обращаться}
{без имени свойства}
Port[$111]:= $80;
A:= Port[$111];
```

Можно использовать этот модуль и без установки компонента, например, как показано в листинге 9.3.

Листинг 9.3. Обращение к портам с помощью динамического создания компонента TPort

```
Uses Port;

procedure TForm1.Button1Click(Sender: TObject);
begin
```

```
 {Оператор with позволит обойтись}
 {без промежуточной переменной}
 With TPort.Create(Self) do
 Try
  Edit1.Text:= IntToStr(Byte[$3F8]);
 Finally
  Free;
 End;
end;
```

9.2. Первая программа для Windows

После того как мы научились обращаться к портам, попробуем создать аналог программ приема и передачи байтов из *гл. 7*, но в среде Windows. Мы постараемся создать как можно более похожую программу, чтобы продемонстрировать простоту перехода из DOS в Windows. А в следующих главах мы займемся использованием "родных" функций Windows.

Запустим Delphi и создадим новый проект. По умолчанию проект будет называться **Project1** и содержать модуль `Unit1.pas`.

Добавим в проект наш модуль `Port` для получения доступа к портам. Единственное существенное отличие от DOS-программы, которое мы позволим себе в Windows, будет заключаться в визуальной настройке параметров порта. Если в DOS мы задавали характеристики порта числом, передаваемым в функцию `Init` (см. примеры в *гл. 7*), то в Windows мы будем визуально выбирать нужные нам характеристики, а это "число" вычислять в программе.

Весь визуальный интерфейс будет заключаться в одной форме, показанной на рис. 9.1.

Основное окно программы разделено на три части. В первой части находятся переключатели номера порта и режимов порта, во второй части — функциональность опроса порта, а в третьей — функциональность передачи данных.

Запустим программу (ее полный код приведен в листинге 9.4) два раза. В первой копии программы выберем порт COM1, а во второй — COM2. Необходимо установить переключатели характеристик таким образом, чтобы настройки обоих портов совпадали. В одном из окон (например, в COM2) нажмем кнопку **Опрос**, а в другом — введем число, например, 44, и нажмем кнопку **Отправить**. В окне порта COM1 мы должны увидеть полученный байт. Если этого не произошло, проверьте правильность распайки кабеля и соединение портов.

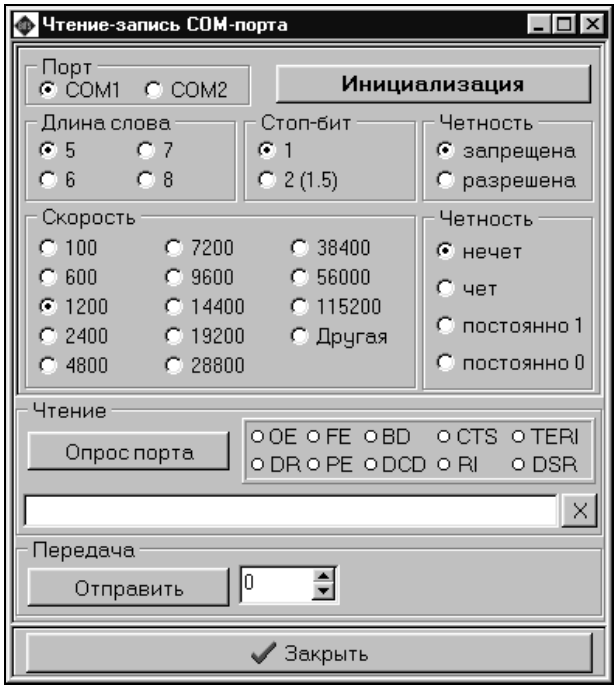

Рис. 9.1. Главное окно нашей первой программы чтения-записи COM-порта для Windows

Еще раз заметим, что приведенный метод работы может использоваться только в Windows 95/98/ME. Для Windows NT/2000/XP должны использоваться либо "родные" функции Windows, либо драйвер прямого доступа, который мы рассмотрим в следующем разделе.

> ***Важно!***
>
> Во всех программах для Delphi мы не приводим код файлов проекта (dpr), т. к. он ничем не отличается от создаваемого по умолчанию. Кроме того, мы не приводим текст dfm-модулей, т. к. сомневаемся, что найдутся желающие набирать его "с листа". Полный, компилируемый код всех примеров есть на прилагаемом компакт-диске.

Листинг 9.4. Прямой доступ к портам из Windows-программы

```
program Comtest;

uses
  Forms,
  Unit1 in 'UNIT1.PAS' {Form1};
```

```
{$R *.RES}

begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.

unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls, Buttons, Spin, Mask, ToolEdit,
  Port;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    btnClose: TBitBtn;
    ReadTimer: TTimer;
    ParamPanel: TPanel;
    rgPortIndex: TRadioGroup;
    btnInit: TButton;
    rgDataLen: TRadioGroup;
    rgStopBits: TRadioGroup;
    rgEnableOdd: TRadioGroup;
    rgBaud: TRadioGroup;
    rgModeOdd: TRadioGroup;
    eOtherBaud: TEdit;
    GroupBox1: TGroupBox;
    bRead: TSpeedButton;
    Panel3: TPanel;
    Bevel2: TBevel;
    DR: TShape;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
```

```
    Label6: TLabel;
    Label7: TLabel;
    Label9: TLabel;
    Label10: TLabel;
    Label8: TLabel;
    Label11: TLabel;
    OE: TShape;
    PE: TShape;
    FE: TShape;
    BD: TShape;
    DCD: TShape;
    RI: TShape;
    CTS: TShape;
    DSR: TShape;
    TERI: TShape;
    Label12: TLabel;
    eReadBuffer: TEdit;
    btnClearReadBuf: TSpeedButton;
    GroupBox2: TGroupBox;
    btnSend: TSpeedButton;
    seSendByte: TSpinEdit;
    procedure FormCreate(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure rgBaudClick(Sender: TObject);
    procedure bReadClick(Sender: TObject);
    procedure ReadTimerTimer(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnSendClick(Sender: TObject);
    procedure btnInitClick(Sender: TObject);
    procedure rgPortIndexClick(Sender: TObject);
    procedure rgPortPropChange(Sender: TObject);
    procedure eOtherBaudKeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure btnClearReadBufClick(Sender: TObject);
    procedure FormActivate(Sender: TObject);
  private
    { Компонент для доступа к порту }
    { Будем создавать его динамически }
    FPort : TPort;
```

```
    { базовый адрес порта }
    BaseCOM : Word;
  public
    procedure ReadCOMParam;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

{ Создание компонента доступа к портам при создании формы }
procedure TForm1.FormCreate(Sender: TObject);
begin
 FPort:= TPort.Create(Self);
end;

{Вызывается при уничтожении формы (выходе из программы) }
procedure TForm1.FormDestroy(Sender: TObject);
begin
 FPort.Free;
end;

{первое чтение}
procedure TForm1.FormActivate(Sender: TObject);
begin
 rgPortIndexClick(nil);
end;

{Инициализация порта}
procedure TForm1.btnInitClick(Sender: TObject);
Var B : Byte; Speed, Freq : Word;
begin
 Try
  {Блокируем повторное нажатие кнопки или}
  {изменение параметров, пока не установим}
  {текущие параметры}
```

```
ParamPanel.Enabled:= False;
ReadTimer.Enabled := False;

{Вычисляем и программируем делитель скорости}
If (rgBaud.ItemIndex = (rgBaud.Items.Count-1)) then
 {скорость обмена не из списка}
 Speed:= StrToIntDef(eOtherBaud.Text, 0)
Else {скорость обмена из списка}
 Speed:= StrToIntDef(rgBaud.Items[rgBaud.ItemIndex], 0);

If Speed = 0 then begin
 MessageDlg('Плохое значение скорости обмена!',
        mtError, [mbOK], 0);
 Exit;
End;

{Устанавливаем бит DLAB для программирования делителя}
FPort[BaseCOM+3]:= FPort[BaseCOM+3] or $80;
{Вычисляем делитель}
Freq:= 115200 div Speed;
{Программируем делитель}
FPort[BaseCOM+0]:= Lo(Freq);
FPort[BaseCOM+1]:= Hi(Freq);
{Сбрасываем бит DLAB}
FPort[BaseCOM+3]:= FPort[BaseCOM+3] and $7F;

B:= 0;
Case rgModeOdd.ItemIndex of
 0: // нечет
  B:= B or $00;
 1: // чет
  B:= B or $10;
 2: // постоянная 1
  B:= B or $20;
 3: // постоянный 0
  B:= B or $30;
End;

Case rgEnableOdd.ItemIndex of
 0: // контроль четности запрещен
```

```
    B:= B or $00;
   1: // контроль четности включен
    B:= B or $08;
  End;

  Case rgStopBits.ItemIndex of
   0: // 1 стоп-бит
    B:= B or $00;
   1: // 2 (1.5) стоп-бита
    B:= B or $04;
  End;

  Case rgDataLen.ItemIndex of
   0: // 5 бит
    B:= B or $00;
   1: // 6 бит
    B:= B or $01;
   2: // 7 бит
    B:= B or $02;
   3: // 8 бит
    B:= B or $03;
  End;

  {установить новые характеристики порта}
  FPort[BaseCOM+3]:= B;
 Finally
  ParamPanel.Enabled:= True;
  ReadTimer.Enabled := bRead.Down;
 End;

 {Перечитать настройки порта}
 ReadCOMParam;
end;

{Перечитать настройки порта при выборе порта}
procedure TForm1.rgPortIndexClick(Sender: TObject);
begin
 {Базовый адрес порта}
 Case rgPortIndex.ItemIndex of
```

```
  0: BaseCOM:= $3F8;
  1: BaseCOM:= $2F8;
 End;
 ReadCOMParam;
end;

{Чтение текущих настроек порта}
procedure TForm1.ReadCOMParam;
Var B : Byte; Speed, Freq : Word; FreqL, FreqH : Byte;
    i : Integer;
begin
 Try
  {Блокируем изменение параметров на время чтения настроек}
  ParamPanel.Enabled:= False;
  ReadTimer.Enabled := False;

  {Устанавливаем бит DLAB для чтения делителя}
  FPort[BaseCOM+3]:= FPort[BaseCOM+3] or $80;
  FreqL:= FPort[BaseCOM+0];
  FreqH:= FPort[BaseCOM+1];
  {Сбрасываем бит DLAB }
  FPort[BaseCOM+3]:= FPort[BaseCOM+3] and $7F;

  {Вычисляем делитель}
  Freq:= FreqH shr 8 + FreqL;

  {Вычисляем скорость обмена}
  If Freq > 0 then Speed:= 115200 div Freq else Speed:= 0;

  {Проверяем наличие такой скорости в таблице}
  i:= rgBaud.Items.IndexOf(IntToStr(Speed));
  If i < 0 then begin {В таблице не нашли — пишем "другая"}
   rgBaud.ItemIndex:= rgBaud.Items.Count-1;
   eOtherBaud.Text := IntToStr(Speed);
   eOtherBaud.Visible:= True;
  End Else begin
   rgBaud.ItemIndex:= i;
   eOtherBaud.Visible:= False;
  End;
```

```
  B:= FPort[BaseCOM+3]; {характеристики порта}

  Case (B and $30) of
   $00: // нечет
    rgModeOdd.ItemIndex:= 0;
   $10: // чет
    rgModeOdd.ItemIndex:= 1;
   $20: // постоянная 1
    rgModeOdd.ItemIndex:= 2;
   $30: // постоянный 0
    rgModeOdd.ItemIndex:= 3;
  End;

  Case (B and $08) of
   $00: // контроль четности запрещен
    rgEnableOdd.ItemIndex:= 0;
   $08: // контроль четности включен
    rgEnableOdd.ItemIndex:= 1;
  End;

  Case (B and $04) of
   $00: // 1 стоп-бит
    rgStopBits.ItemIndex:= 0;
   $04: // 2 (1.5) стоп-бита
    rgStopBits.ItemIndex:= 1;
  End;

  Case (B and $03) of
   $00: // 5 бит
    rgDataLen.ItemIndex:= 0;
   $01: // 6 бит
    rgDataLen.ItemIndex:= 1;
   $02: // 7 бит
    rgDataLen.ItemIndex:= 2;
   $03: // 8 бит
    rgDataLen.ItemIndex:= 3;
  End;
 Finally
  ParamPanel.Enabled:= True;
```

```
  ReadTimer.Enabled := bRead.Down;
 End;
End;

{Кнопка закрытия программы}
procedure TForm1.btnCloseClick(Sender: TObject);
begin
 Close;
end;

procedure TForm1.rgBaudClick(Sender: TObject);
begin
 {Видимость окошка "Другая скорость"}
 eOtherBaud.Visible:=
   (rgBaud.ItemIndex = (rgBaud.Items.Count-1));
 {Изменение характеристик порта}
 rgPortPropChange(nil);
end;

procedure TForm1.eOtherBaudKeyDown(Sender: TObject; var Key: Word; Shift:
TShiftState);
begin
 {Изменение скорости обмена по клавише Enter}
 If Key = 13 then rgPortPropChange(nil);
end;

{Любое изменение характеристик порта}
procedure TForm1.rgPortPropChange(Sender: TObject);
begin
 {Переинициализация}
 btnInitClick(nil);
end;

{Кнопка опроса порта}
procedure TForm1.bReadClick(Sender: TObject);
begin
 ReadTimer.Enabled:= bRead.Down;
end;
```

```
Const {Цвета индикаторов}
 IColor : Array [Boolean] of TColor = (clWhite, clGreen);

procedure TForm1.ReadTimerTimer(Sender: TObject);
Var B, B1, B2 : Byte; S : String;
begin
 {Состояние модема и порта считываем до чтения байта,}
 {т. к. после чтения данных с порта состояние порта изменится.}
 B1:= FPort[BaseCOM+5]; {Байт состояния порта}
 B2:= FPort[BaseCOM+6]; {Байт состояния модема}

 {Проверка наличия данных и их отображение}
 If (B1 and $01) <> 0 then begin {Бит "есть данные для чтения"}
  B:= FPort[BaseCOM+0]; { принятый байт }
  S:= eReadBuffer.Text;
  S:= S + Format('$%X',[B]);
  If Length(S) > 50 then Delete(S, 1, 3);
  eReadBuffer.Text:= S;
 End;

 {Отображение индикаторов состояния порта}
 {Сигнал DR (данные готовы)}
 DR.Brush.Color:= IColor[(B1 and $01) <> 0];
 {Сигнал OE (переполнение)}
 OE.Brush.Color:= IColor[(B1 and $02) <> 0];
 {Ошибка паритета}
 PE.Brush.Color:= IColor[(B1 and $04) <> 0];
 {Ошибка кадра (неверный стоп-бит)}
 FE.Brush.Color:= IColor[(B1 and $08) <> 0];
 {Индикатор обрыва линии}
 BD.Brush.Color:= IColor[(B1 and $10) <> 0];

 {Отображение индикаторов состояния модема}
 {Состояние линии DCD (data carrier detect)}
 DCD.Brush.Color := IColor[(B2 and $80) <> 0];
 {Состояние линии RI (ring indicator)}
 RI .Brush.Color := IColor[(B2 and $40) <> 0];
 {Состояние линии DSR (data set ready)}
 DSR.Brush.Color := IColor[(B2 and $20) <> 0];
```

```
 {Состояние линии CTS (clear to send)}
 CTS.Brush.Color := IColor[(B2 and $10) <> 0];
 {Сигнал "спад огибающей" (окончание звонка)}
 TERI.Brush.Color:= IColor[(B2 and $04) <> 0];
end;

{Очистить буфер чтения}
procedure TForm1.btnClearReadBufClick(Sender: TObject);
begin
 eReadBuffer.Text:= '';
end;

{Отправка байта}
procedure TForm1.btnSendClick(Sender: TObject);
begin
 FPort[BaseCOM]:= seSendByte.Value;
end;

end.
```

Интересно понаблюдать за индикаторами ошибок связи. Для эксперимента можно установить разные характеристики портов приемника и передатчика и пробовать передать байт. Варьируя характеристики, можно добиться индицирования ошибок переполнения или четности.

9.3. Перенос программ из Windows 9*x* в Windows NT/2000

Запустив пример из предыдущего раздела в Windows 98, мы увидим в `Edit1` значение порта $3F8, а запустив его в Windows 2000 — сообщение об ошибке при попытке прямого чтения порта:

```
"Project Project1.exe raised exception class EPrivilege with message
'Privileged instruction'. Process stopped. Use Step or Run to continue."
```

Сообщение об ошибке вызвано тем, что версии Windows NT и выше блокируют прямые обращения к аппаратным ресурсам компьютера. Для использования прямого обращения к портам в Windows NT/2000/XP следует использовать специальный драйвер. Таким образом, Windows отделяет уровень пользовательских программ от аппаратного уровня, что существенно повышает надежность и защищенность операционной системы: обычные про-

граммы обязаны обращаться к драйверу, а драйвер блокирует одновременное использование аппаратных ресурсов, если программа не сообщит ему о возможности их разделения между несколькими программами. Именно так делают стандартные функции Windows для последовательных портов, которые мы рассмотрим в *гл. 10*.

Но что делать, если программа уже написана и работает, а времени на переделку нет? Другими словами, как перенести программу, работающую напрямую с аппаратурой, из Windows 95/98 в Windows NT/2000?

Мы будем рассматривать два варианта перехода из Windows 95/98 в Windows NT/2000/XP. Первый вариант, когда нам доступны исходные тексты программы и есть возможность ее перекомпилировать (т. е. внедрить в программу новый код), и второй, когда исходный текст отсутствует, а заставить программу работать в новых версиях Windows все равно нужно. Вообще говоря, оба этих случая сводятся к одному решению — написанию специального драйвера, разрешающего доступ к портам. Только в первом случае он будет разрешать доступ для текущего процесса, а во втором — для чужого.

За основу мы возьмем драйвер GiveIO, ставший уже классическим. Правда, он рассчитан для работы только в первом случае, когда необходимо разрешить доступ своему процессу, но небольшая его модификация позволит нам решить и вторую проблему.

Драйвер GiveIO создан Дейлом Робертсом (Dale Roberts) специально для обхода ограничения прямого доступа к портам. Исходный код драйвера и некоторые объяснения его работы приведены в *прил. 7*. Конечно, в рамках нашей книги мы не можем приводить подробные объяснения принципов работы и написания драйверов. Мы ограничимся чисто практическим подходом.

9.3.1. Получение доступа к портам в Windows 2000/XP

Читателей, интересующихся принципом работы драйвера, мы отсылаем к *прил. 7*, где описан принцип работы драйвера и приводится его исходный код. Использовать этот драйвера очень просто — достаточно загрузить его в память. После этого его можно даже выгрузить, для работы программы он не требуется.

Для упрощения использования драйвера мы напишем специальный модуль `GiveIO.Pas` (листинг 9.5). Класс, описанный в этом листинге, содержит пять методов:

- `CreateService` — регистрация драйвера в реестре (создание записи о сервисе);
- `StartService` — старт сервиса, соответствующего драйверу;

- `GiveIoForCurrentProcess` — получение доступа для текущего процесса;
- `StopService` — останов сервиса драйвера;
- `RemoveService` — удаление записи о драйвере из реестра.

Мы привели список методов именно в том порядке, в котором их нужно вызвать для получения полного доступа к портам. Причем, как мы уже говорили, совершенно не важно, будет ли драйвер выгружен сразу же после вызова `GiveIoForCurrentProcess` или при завершении программы.

Код программы, показывающей работу драйвера, мы приведем чуть позже, объединив его с примером предоставления доступа без перекомпиляции программы.

Листинг 9.5. Исходный код модуля `GiveIO.Pas`

```
unit GiveIO;

interface

Uses Windows, SysUtils;

type
 TGiveIO = class
  Public
   Function CreateService(SysPath : String) : Boolean;
   Function RemoveService : Boolean;

   Function StartService  : Boolean;
   Function StopService   : Boolean;

   Procedure GiveIoForCurrentProcess;
End;

implementation

Uses WinSvc, Dialogs;

Const
 DriverName : PChar= 'giveio'#0;
 FileDriver : String = 'giveio.sys'#0;
```

```
Function TGiveIO.CreateService(SysPath : String) : Boolean;
var lpServiceArgVectors : PChar;
    hSCMan, hService: SC_HANDLE;
    DriverPath : String;
Begin
 Result:= False;

 {== Создание сервиса ==}
 {Сервис регистрируется в ветке реестра}
 {HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\giveio}
 {Будут созданы ключи:}
 { DisplayName = имя_драйвера (равно giveio)}
 { ImagePath   = путь к файлу sys. При регистрации в локальной директории
}
 {               будет равно \??\E:\Test\giveio.sys}
 hSCMan:= WinSvc.OpenSCManager(
           Nil,         { local }
           Nil,         { SERVICES_ACTIVE_DATABASE }
           SC_MANAGER_ALL_ACCESS
          );
 If hSCMan = 0 then Exit;

 {Получаем полное имя к файлу драйвера }
 DriverPath:= SysPath + FileDriver;

 {Создаем сервис (регистрируем драйвер)}
 hService:= WinSvc.CreateService(hSCMan, Drivername, DriverName,
      SERVICE_ALL_ACCESS, SERVICE_KERNEL_DRIVER,
      SERVICE_DEMAND_START, SERVICE_ERROR_NORMAL,
      PChar(@DriverPath[1]),
      nil,nil,nil,nil,nil);
 {Файл не найден или сервис уже зарегистрирован}
 If hService = 0 then begin
  MessageDlg(IntToStr(GetLastError), mtError, [mbOK], 0);
  CloseServiceHandle(hSCMan);
  Exit;
 End;

 {Регистрация успешна}
 WinSvc.CloseServiceHandle(hService);
```

```
 WinSvc.CloseServiceHandle(hSCMan);
 Result:= True;
End;

Function TGiveIO.RemoveService : Boolean;
var serviceStatus    : TServiceStatus;
    hSCMan, hService : SC_HANDLE;
Begin
 Result:= False;
 {== Удаление сервиса из реестра}
 hSCMan:= WinSvc.OpenSCManager(Nil,Nil,SC_MANAGER_ALL_ACCESS);
 If hSCMan = 0 then Exit;

 hService:= WinSvc.OpenService(hSCMan, DriverName, SERVICE_ALL_ACCESS);
 {Ошибка открытия сервиса}
 If hService=0 then begin
  CloseServiceHandle(hSCMan);
  Exit;
 End;

 WinSvc.DeleteService(hService);
 WinSvc.CloseServiceHandle(hService);
 WinSvc.CloseServiceHandle(hSCMan);
 Result:= True;
End;

Function TGiveIO.StartService  : Boolean;
var lpServiceArgVectors : PChar;
    hSCMan, hService : SC_HANDLE;
    DriverPath : String;
Begin
 Result:= False;

 {== Старт зарегистрированного сервиса}
 hSCMan  := WinSvc.OpenSCManager(Nil, Nil, SC_MANAGER_CONNECT);
 If hSCMan = 0 then Exit;

 hService:= WinSvc.OpenService(hSCMan, DriverName, SERVICE_START);
 If hService = 0 then begin
```

```
  CloseServiceHandle(hSCMan);
  Exit;
 End;

 lpServiceArgVectors:=nil;
 WinSvc.StartService(hService, 0, lpServiceArgVectors);
 WinSvc.CloseServiceHandle(hService);
 WinSvc.CloseServiceHandle(hSCMan);
 Result:= True;
End;

Function TGiveIO.StopService   : Boolean;
var serviceStatus    : TServiceStatus;
    hSCMan, hService : SC_HANDLE;
Begin
 Result:= False;
 {== Остановка сервиса}
 hSCMan:= WinSvc.OpenSCManager(Nil, Nil, SC_MANAGER_CONNECT);
 If hSCMan = 0 then Exit;

 hService:= WinSvc.OpenService(hSCMan, DriverName, SERVICE_STOP);
 If hService = 0 then begin
  WinSvc.CloseServiceHandle(hSCMan);
  Exit;
 End;

 WinSvc.ControlService(hService, SERVICE_CONTROL_STOP, serviceStatus);
 WinSvc.CloseServiceHandle(hService);
 WinSvc.CloseServiceHandle(hSCMan);
 Result:= True;
End;

{== Загрузка драйвера в память ==}
{Дает доступ к портам для текущего процесса}
Procedure TGiveIO.GiveIoForCurrentProcess;
var hDevice : SC_HANDLE;
Begin
 hDevice:= CreateFile('\\.\giveio',
             GENERIC_READ or GENERIC_WRITE,
```

```
            0,nil,
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,
            0
          );
 CloseHandle(hDevice);
End;

End.
```

9.3.2. Расширение возможностей GiveIO

Итак, наша задача: доработать драйвер GiveIO таким образом, чтобы его можно было использовать для получения доступа к портам не только текущего процесса, но и любого процесса, идентификатор которого нам известен.

Прежде чем приступить к работе, следует рассмотреть, как прикладная программа взаимодействует с драйвером. В *гл. 13* мы еще будем обсуждать функцию `DeviceIoControl`, а пока только опишем параметры этой функции:

- `hDevice` — дескриптор драйвера, получаемый с помощью вызова `CreateFile`. В качестве открываемого файла используется имя драйвера;
- код команды, состоящий из идентификатора драйвера, атрибутов команды и номера команды. Идентификатор драйвера гарантирует доставку команды только одному, нужному драйверу, а номер команды передает указание на выполнение конкретных действий;
- входной буфер и его размер — передают данные, необходимые драйверу для выполнения указанной команды;
- выходной буфер и его размер — передают драйверу буфер для размещения ответа;
- код результата — позволяет узнать правильность выполнения команды.

Этих сведений нам пока достаточно. Итак, прикладная программа вызывает `DeviceIoControl` с кодом функции и некоторыми параметрами. Ядро Windows ищет соответствующий драйвер (именно для этой цели, чтобы система могла найти драйвер, мы зарегистрировали его в реестре, вызвав `CreateService`) и передает код функции драйверу. Драйвер выполняет указанные действия и, при необходимости, возвращает некоторые данные. Весь интерес заключается в том, что функции драйвера выполняются на уровне ядра системы и им доступны все ресурсы Windows. Заметим, что "правильный" драйвер устройства должен реализовывать все необходимые функции доступа к аппаратуре внутри себя. Другими словами, надо сделать обработку некоторого набора кодов функций, а программа должна ими

пользоваться (например, коды драйвера последовательного порта описаны в *гл. 22*). У нас же другая цель. Мы хотим создать одну функцию, выполняющуюся на уровне драйвера. Эта функция должна уметь открывать доступ указанному процессу.

Таким образом, нам надо реализовать две части обработки. Первая часть — обработка кода команды в драйвере, вторая — передача данных из программы в драйвер.

Для начала нам нужно придумать номер команды. Вообще говоря, это произвольное положительное число от $800 до $FFF. Самое главное, чтобы и драйвер, и программа использовали одно и то же число. Второе число, необходимое для передачи команды, — код типа драйвера. Коды драйверов от 0 до 32767 зарезервированы, а коды от 32768 до 65535 могут использоваться прикладными программами. Естественно, драйвер нам придется компилировать с помощью Microsoft DDK, т. е. на языке C, а обработку (обращение) к нему мы будем производить, как обычно, на Delphi. В листинге 9.6 приводятся константы, составляющие код функции и макрос для его генерации. А листинг 9.7 содержит тот же код на Delphi.

Листинг 9.6. Файл описания команды для драйвера (GiveIOEx.h)

```
// Номера 32768-65535 зарезервированы для пользователя
#define GIVEIO_TYPE 40000
// Коды функций IOCTL от 0x800 до 0xFFF могут использоваться
#define IOCTL_IOPM_GET_ALL_ACCESS\
    CTL_CODE(GIVEIO_TYPE, 0x900, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

Листинг 9.7. Описание команды драйвера в Delphi

```
Const
 GIVEIO_TYPE     = 40000;
 METHOD_BUFFERED =     0;
 FILE_ANY_ACCESS = $0000;

function Get_Ctl_Code(Nr: Integer): Cardinal;
begin
  Result:=
   (GIVEIO_TYPE shl 16) or
   (FILE_ANY_ACCESS shl 14) or
   (Nr shl 2) or
   METHOD_BUFFERED;
end;
```

```
Var
 IOCTL_IOPM_GET_ALL_ACCESS: Cardinal;
... ... ... ...
IOCTL_IOPM_GET_ALL_ACCESS:= Get_Ctl_Code($900);
```

Теперь остается добавить код обработки новой команды в драйвер. Как мы уже говорили, при выполнении этой команды драйвер должен открывать доступ к процессу, идентификатор которого передается как входной параметр команды. Код получившегося драйвера приведен в листинге 9.8. Для его компиляции нужно использовать утилиту `Build` из поставки Windows DDK, а на прилагаемом компакт-диске можно найти уже готовый SYS-файл.

Листинг 9.8. Код расширенного драйвера GiveIOEx

```
/*
Драйвер предоставления прямого доступа для процесса режима пользователя.
Драйвер основан на коде драйвера GiveIO от Dale Roberts, но в него добавлена возможность предоставления доступа другим процессам с помощью команды 0x900
Компиляция: Используйте средство DDK BUILD
*/

#include <ntddk.h>
#include "GiveIOEx.h"

/* Имя нашего драйвера устройства */
#define DEVICE_NAME_STRING L"giveioex"

/*
Это "структура" IOPM. Это просто массив байт размером 0x2000. Он содержит
8K * 8 бит = 64 Кбит IOPM, которые покрывают все 64-килобайтное адресное
пространство в/в x86 процессора. Каждый нулевой бит предоставляет доступ
к соответствующему порту для user-mode процесса. Каждый единичный бит
запрещает доступ к в/в через соответствующий порт.
*/

#define IOPM_SIZE 0x2000

typedef UCHAR IOPM[IOPM_SIZE];
```

```
/*
Это будет содержать просто массив нулей, который будет копироваться в
настоящую IOPM в TSS через Ke386SetIoAccessMap(). Память выделяется во
время загрузки драйвера.
*/

IOPM * IOPM_local = 0;

/*
Это две недокументированных функции, которые мы используем, чтобы дать
доступ к в/в вызывающему процессу. Ke386IoSetAccessMap() копирует пере-
данную карту в/в в TSS. Ke386IoSetAccessProcess() изменяет указатель сме-
щения IOPM, после чего только что скопированная карта в/в начинает ис-
пользоваться. Иначе смещение IOPM указывает за предел сегмента TSS, что
приведет к генерации исключения при попытке доступа к в/в любым user-mode
процессом.
*/

void Ke386SetIoAccessMap(int, IOPM *);
void Ke386QueryIoAccessMap(int, IOPM *);
void Ke386IoSetAccessProcess(PEPROCESS, int);
NTSTATUS PsLookupProcessByProcessId(IN ULONG ulProcId,
                                    OUT struct _EPROCESS ** pEProcess);
/*
Сервисная функция драйвера.
*/
NTSTATUS GiveioDeviceControl(IN PDEVICE_OBJECT DeviceObject,
                             IN PIRP Irp);

/*
Освободить все выделенные ранее объекты
*/
VOID GiveioUnload(IN PDRIVER_OBJECT DriverObject)
{
 WCHAR DOSNameBuffer[] = L"\\DosDevices\\" DEVICE_NAME_STRING;
 UNICODE_STRING uniDOSString;
 if(IOPM_local)
  MmFreeNonCachedMemory(IOPM_local, sizeof(IOPM));
 RtlInitUnicodeString(&uniDOSString, DOSNameBuffer);
 IoDeleteSymbolicLink (&uniDOSString);
 IoDeleteDevice(DriverObject->DeviceObject);
}
```

```
/*
Устанавливаем IOPM (карту разрешения в/в) процесса ProcessID так, что ему
предоставляется полный доступ к в/в. Наш массив OPM_local[] содержит
только нули, соответственно, IOPM обнулится. Если OnFlag = 1, процессу
предоставляется доступ к в/в. Если он равен 0, доступ запрещается.
*/
VOID SetIOPermissionMap(PEPROCESS ProcessID, int OnFlag)
{
 Ke386IoSetAccessProcess(ProcessID, OnFlag);
 Ke386SetIoAccessMap(1, IOPM_local);
}

/*
Точка входа в драйвер. Выполняется при загрузке. Установка разрешения
доступа ко всем портам для вызывающего процесса.
*/
void GiveIO(void)
{
 SetIOPermissionMap(PsGetCurrentProcess(), 1);
}

/*
Служебный обработчик для user-mode вызова CreateProcess(). Эта функция
введена в таблицу вызовов функций объекта драйвера с помощью DriverEntry().
Когда user-mode приложение вызывает CreateFile(), эта функция получает
управление все еще в контексте вызвавшего приложения, но с CPL (текущий
уровень привилегий процессора), установленным в 0. Это позволяет произво-
дить операции, возможные только в kernel mode. GiveIO() вызывается для
предоставления вызывающему процессу доступа к в/в. Все, что приложение
режима пользователя, которому нужен доступ к в/в, должно сделать, — это
открыть данное устройство, используя CreateFile(). Никаких других дейст-
вий не нужно.
*/
NTSTATUS GiveioCreateDispatch(
 IN PDEVICE_OBJECT DeviceObject,
 IN PIRP Irp
)
{
 GiveIO(); // дать доступ текущему процессу
 Irp->IoStatus.Information = 0;
 Irp->IoStatus.Status = STATUS_SUCCESS;
```

```
 IoCompleteRequest(Irp, IO_NO_INCREMENT);
 return STATUS_SUCCESS;
}

/*
Процедура входа драйвера. Эта процедура вызывается только раз после за-
грузки драйвера в память. Она выделяет необходимые ресурсы для рабо-
ты драйвера. В нашем случае она выделяет память для массива IOPM и созда-
ет устройство, которое может открыть приложение режима пользователя. Она
также создает символическую ссылку на драйвер устройства. Это позволя-
ет user-mode приложению получить доступ к нашему драйверу, используя
\\.\giveioex нотацию.
*/
NTSTATUS DriverEntry(
 IN PDRIVER_OBJECT DriverObject,
 IN PUNICODE_STRING RegistryPath
)
{
 PDEVICE_OBJECT deviceObject;
 NTSTATUS status;

 WCHAR NameBuffer[] = L"\\Device\\" DEVICE_NAME_STRING;
 WCHAR DOSNameBuffer[] = L"\\DosDevices\\" DEVICE_NAME_STRING;
 UNICODE_STRING uniNameString, uniDOSString;

 // Выделим буфер для локальной IOPM и обнулим его.
 IOPM_local = MmAllocateNonCachedMemory(sizeof(IOPM));
 if(IOPM_local == 0)
  return STATUS_INSUFFICIENT_RESOURCES;

 RtlZeroMemory(IOPM_local, sizeof(IOPM));

 // Инициализируем драйвер устройства и объект
 // устройства (device object)

 RtlInitUnicodeString(&uniNameString, NameBuffer);
 RtlInitUnicodeString(&uniDOSString, DOSNameBuffer);

 status = IoCreateDevice(DriverObject, 0, &uniNameString,
               FILE_DEVICE_UNKNOWN, 0, FALSE, &deviceObject);
```

```
 if(!NT_SUCCESS(status))
  return status;

 status = IoCreateSymbolicLink (&uniDOSString, &uniNameString);
 if (!NT_SUCCESS(status))
  return status;

 // Инициализируем точки входа драйвера в объекте
 // драйвера. Все, что нам нужно, — это операции создания
 // (Create) и выгрузки (Unload)
 DriverObject->MajorFunction[IRP_MJ_CREATE]= GiveioCreateDispatch;
 DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
                                              GiveioDeviceControl;
 DriverObject->DriverUnload = GiveioUnload;

 return STATUS_SUCCESS;
}

NTSTATUS
GiveioDeviceControl(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP pIrp
    )

{
 NTSTATUS            ntStatus = STATUS_SUCCESS;
 PIO_STACK_LOCATION  irpSp;
 ULONG               inBufLength;   /* Длина входного буфера  */
 ULONG               outBufLength;  /* Длина выходного буфера */
 PULONG              LongBuffer;    /* Указатель на входной буфер */


 irpSp       = IoGetCurrentIrpStackLocation( pIrp );
 inBufLength = irpSp->Parameters.DeviceIoControl.InputBufferLength;
 LongBuffer  = (PULONG) pIrp->AssociatedIrp.SystemBuffer;

 switch ( irpSp->Parameters.DeviceIoControl.IoControlCode )
 {
  case IOCTL_IOPM_GET_ALL_ACCESS:
```

```
  {
   if (inBufLength >= 4) {
    ULONG ProcessID = LongBuffer[0];
    struct _EPROCESS *Process;

    PsLookupProcessByProcessId(ProcessID, &Process);
    Ke386SetIoAccessMap(1, IOPM_local);
    Ke386IoSetAccessProcess(Process, 1);
    ntStatus = STATUS_SUCCESS;
   } else {
    ntStatus = STATUS_BUFFER_TOO_SMALL;
    pIrp->IoStatus.Information = 0; /* Output Buffer Size */
   }
   break;
  }
 }

 pIrp->IoStatus.Status = ntStatus;
 IoCompleteRequest( pIrp, IO_NO_INCREMENT );
 return ntStatus;
}
```

Обработчик новой команды находится внутри процедуры `GiveioDeviceControl`. При необходимости добавить другие команды, следует выполнить три действия:

1. Добавить новый код команды в заголовочный файл GiveIOEx.h.
2. Добавить обработчик нового кода в оператор `switch` процедуры `GiveioDeviceControl`.
3. Заменить загрузку обычного giveio.sys на расширенный драйвер giveioex.sys.

Со стороны драйвера все готово. Остается только добавить новый метод в класс `TGiveIO`, вызывающий только что созданную функцию $900, как показано в листинге 9.9.

Листинг 9.9. Функция предоставления доступа к портам выбранного процесса

```
unit GiveIO;

interface
```

```
Uses Windows, SysUtils;

type
 TGiveIO = class
  Public
   ... ... ... ... ... ... ... ...
   Procedure GiveIoForProcess(dwProcessId: Cardinal);
 End;

implementation

Uses WinSvc, Dialogs;

   ... ... ... ... ... ... ... ...

Const
 GIVEIO_TYPE       =  40000;
 METHOD_BUFFERED   =  0;
 FILE_ANY_ACCESS   = $0000;

function Get_Ctl_Code(Nr: Integer): Integer;
begin
  Result:=
     (GIVEIO_TYPE shl 16) or
     (FILE_ANY_ACCESS shl 14) or
     (Nr shl 2) or
     METHOD_BUFFERED;
end;

Procedure TGiveIO.GiveIoForProcess(dwProcessId : Cardinal);
var hDevice : SC_HANDLE; Result : Cardinal;
begin
 hDevice:= CreateFile('\\.\giveioex',
             GENERIC_READ or GENERIC_WRITE,
             0,nil,
             OPEN_EXISTING,
             FILE_ATTRIBUTE_NORMAL,
             0
           );
```

```
 DeviceIoControl(hDevice, Get_Ctl_Code($900),
    @dwProcessId, SizeOf(dwProcessId), nil, 0, Result, nil);
 CloseHandle(hDevice);
end;
```

9.3.3. Работа с драйвером GiveIOEx

Теперь все готово к проверке работоспособности нашего драйвера. В качестве тестовой программы мы создадим небольшой проект, выполняющий следующие действия:

- регистрация драйвера;
- удаление драйвера;
- предоставление доступа к портам текущему процессу;
- тестирование прямого доступа к портам;
- запуск нового процесса и предоставление ему доступа к портам.

Вид формы тестового проекта показан на рис. 9.2, а код соответствующего модуля приведен в листинге 9.10.

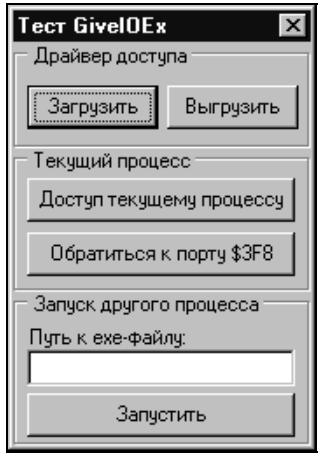

Рис. 9.2. Окно программы тестирования драйвера GiveIOEx

Листинг 9.10. Модуль тестирования драйвера GiveIOEx

```
unit Unit1;

interface
```

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms,  Dialogs, StdCtrls, GiveIO;

type
  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    btnLoad: TButton;
    btnUnload: TButton;
    GroupBox2: TGroupBox;
    btnGiveCurr: TButton;
    btnTest: TButton;
    GroupBox3: TGroupBox;
    ExeName: TEdit;
    btnExec: TButton;
    Label1: TLabel;
    procedure btnTestClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnLoadClick(Sender: TObject);
    procedure btnUnloadClick(Sender: TObject);
    procedure btnGiveCurrClick(Sender: TObject);
    procedure btnExecClick(Sender: TObject);
  private
    FGiveIO : TGiveIO;
  public
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

{Создание компонента TGiveIO}
procedure TForm1.FormCreate(Sender: TObject);
begin
 FGiveIO:= TGiveIO.Create;
end;
```

```
{Уничтожение компонента TGiveIO}
procedure TForm1.FormDestroy(Sender: TObject);
begin
 FGiveIO.Free;
end;

{Тестовая кнопка обращения к базовому COM-порту}
procedure TForm1.btnTestClick(Sender: TObject);
var Result : Byte;
begin
 asm
  mov dx,$3F8
  in al,dx
  mov Result,al
 end;
end;

{Регистрация сервиса в реестре и старт}
procedure TForm1.btnLoadClick(Sender: TObject);
begin
 FGiveIO.CreateService(ExtractFilePath(ParamStr(0)));
 FGiveIO.StartService;
end;

{Останов сервиса и удаление из реестра}
procedure TForm1.btnUnloadClick(Sender: TObject);
begin
 FGiveIO.StopService;
 FGiveIO.RemoveService;
end;

{Получить доступ к портам для текущего процесса}
procedure TForm1.btnGiveCurrClick(Sender: TObject);
begin
 FGiveIO.GiveIoForCurrentProcess;
end;

{Запустить другой процесс (программу) и предоставить}
{ему доступ к портам}
```

```
procedure TForm1.btnExecClick(Sender: TObject);
var si : STARTUPINFO; pi : PROCESS_INFORMATION;
begin
 ZeroMemory(@si, sizeof(si));
 si.cb:= sizeof(si);

 if not CreateProcess(nil, PChar(ExeName.Text), nil, nil, False,
          0, nil, nil, si, pi) then begin
  MessageDlg('Ошибка запуска программы', mtError, [mbOK], 0);
  Exit;
 End;

 FGiveIO.GiveIoForProcess(pi.dwProcessId);
end;

end.
```

Для проверки запустим нашу программу (не забывайте, что мы работаем в Windows 2000 или Windows XP!) и нажмем кнопку **Обратиться к порту $3F8**. Если все правильно, появится окно, сообщающее, что программа попыталась выполнить запрещенную для нее инструкцию (рис. 9.3).

Рис. 9.3. Сообщение о попытке выполнить запрещенную инструкцию

Теперь нажмем кнопку **Загрузить**. Как видно из кода обработчика `btnLoadClick()`, происходит два действия. Первым шагом в реестре регистрируется новый сервис, а вторым — производится его старт. В качестве подтверждения в ветке реестра

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\
```

должен появиться новый раздел `giveioex`, содержащий все параметры нового драйвера (рис. 9.4).

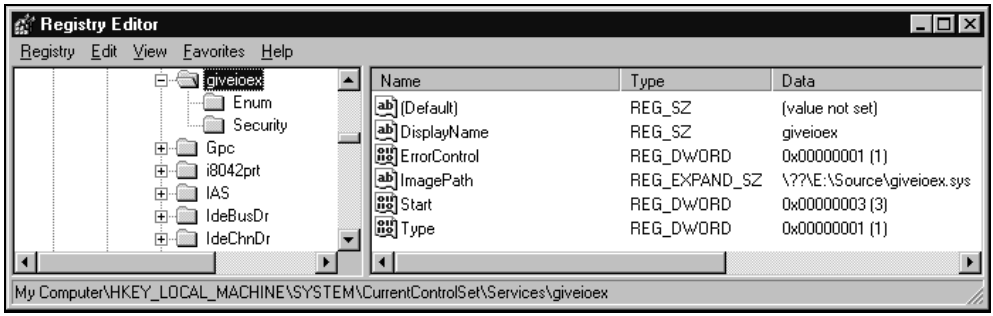

Рис. 9.4. Регистрация драйвера в реестре

Попытавшись теперь обратиться к порту $3F8, мы обнаружим, что доступ открыт и никаких ошибок не возникает.

Остается только убедиться, что мы можем предоставить доступ к портам не только своей программе, но и другой. Введем в поле **Путь к EXE-файлу** название этой же программы, например, `E:\Test\Project1.exe`. Как мы помним, простой ее запуск и попытка обращения к порту приводили к ошибке. Запускаем и пробуем обратиться к порту... Ошибка не возникает! Можете попробовать запустить любую программу, требующую доступа к портам, и она будет работать.

Для того чтобы не таскать файл драйвера вместе с файлом программы, удобнее встроить его внутрь EXE-файла программы. Для этого с помощью любого редактора ресурсов нужно создать файл GiveIOEx.RES, содержащий бинарный код драйвера. Затем необходимо подключить ресурсный файл к программе, и, при старте драйвера, создавать временный файл giveioex.sys, загружать его и удалять. Пример модуля, загружающего образ драйвера из файла ресурса, показан в листинге 9.11 (для экономии места мы снова показали только отличия этого модуля от модуля `GiveIO.Pas`). Для его использования необходимо подключить к основному модулю файл `GiveIO_R` вместо `GiveIO`, а `FGiveIO.CreateService` вызывать без параметров (листинг 9.12).

Листинг 9.11. Модуль `GiveIO_R`, загружающий образ драйвера GiveIO из файла ресурса

```
unitGiveIO_R;

interface

Uses Windows, SysUtils, Classes;
```

```
type
 TGiveIO = class
  Private
   Function CreateFileFromResource(FName : String) : Boolean;
  Public
   Function CreateService : Boolean;
   Function RemoveService : Boolean;

   Function StartService  : Boolean;
   Function StopService   : Boolean;

   Procedure GiveIoForCurrentProcess;
   Procedure GiveIoForProcess(dwProcessId: Cardinal);
 End;

implementation

Uses WinSvc, Dialogs;

Const
 DriverName : PChar= 'giveio'#0;

{ Подключение файла ресурсов }
{$R GiveIO.RES}

function GetSystemDir : String;
var
  Buffer: array[0..1023] of Char;
begin
 SetString(Result, Buffer, GetSystemDirectory(Buffer, SizeOf(Buffer)));
end;

function GetDriverPath : String;
begin
 Result:= GetSystemDir + '\giveioex.sys';
end;

Function TGiveIO.CreateService : Boolean;
var lpServiceArgVectors, P : PChar;
```

```
    hSCMan, hService: SC_HANDLE;
    DriverPath : String;
Begin
 Result:= False;

 DriverPath:= GetDriverPath;
 CreateFileFromResource(DriverPath);
 ... ... ... ... ... ... ... ... ...
End;

Function TGiveIO.RemoveService : Boolean;
var serviceStatus    : TServiceStatus;
    hSCMan, hService : SC_HANDLE;
    DriverPath : String;
Begin
 Result:= False;

 DriverPath:= GetDriverPath;
 Windows.DeleteFile(PChar(DriverPath));

 ... ... ... ... ... ... ... ... ...
End;

... ... ... ... ... ... ... ... ...

Function TGiveIO.CreateFileFromResource(FName : String) : Boolean;
Var S: TFileStream;  Rsrc: HRSRC; Res: THandle;
    Data: Pointer;
Begin
 Result := False;
 Rsrc := FindResource(HInstance, MakeIntResource(102), RT_RCDATA);
 If Rsrc = 0 then Exit;

 Res:= LoadResource(HInstance, Rsrc);
 Try
  Data := LockResource(Res);
  If Data <> nil then
   Try
    S:= TFileStream.Create(FName, fmCreate);
```

```
    Try
     S.WriteBuffer(Data^, SizeOfResource(HInstance, Rsrc));
    Finally
     S.Free;
    End;
    Result:= True;
   Finally
    UnlockResource(Res);
   End;
  Finally
   FreeResource(Res);
  End;
End;

end.
```

Листинг 9.12. Использование GiveIO_R

```
unit Unit1;

interface

uses
 Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls, GiveIO_R;

type
  TForm1 = class(TForm)
   ... ... ... ... ...
  end;
   ... ... ... ... ...
{Регистрация сервиса в реестре и старт}
procedure TForm1.btnLoadClick(Sender: TObject);
begin
 FGiveIO.CreateService;
 FGiveIO.StartService;
end;
   ... ... ... ... ...
end.
```

9.3.4. Еще немного о прямом доступе к портам

Еще одна возможная причина необходимости прямого доступа — время выполнения команд ввода/вывода. Проще говоря, драйвер GiveIO изменяет карту разрешения ввода/вывода (IOPM, I/O Permission Map), "отключая" механизм защиты Windows. На проверку защиты требуется процессорное время, а, отключая защиту, мы экономим на этих проверках. Робертс в работе [12] приводит некоторые оценки. На выполнение команды ввода/вывода тратится (процессор 486):

- в DOS-режиме 16 тактов;
- в режиме ядра — 10 тактов;
- DOS-программа в режиме консоли NT — 29 тактов;
- Windows-программа через драйвер — от 6000 до 12 000 тактов (!);
- Windows-программа в режиме прямого доступа — 30 тактов.

Однако вслед за Робертсом мы должны предупредить обрадовавшихся свободе программистов — не злоупотребляйте прямым доступом к порту! Система защиты придумана не зря и предусматривает жесткую синхронизацию драйверов и устройств. Прямой доступ может привести к сбою синхронизации и, в конечном итоге, краху системы. Кроме того, следует помнить, что драйвер прямого доступа открывает доступ к портам, но не разрешает выполнение всех привилегированных инструкций, таких как `STI` (разрешение прерываний) и `CLI` (запрет прерываний). Такие инструкции могут быть выполнены только драйвером режима ядра.

И еще об одном недостатке драйвера GiveIO. Как видно из исходного кода, этот драйвер использует три недокументированных функции Windows. И, следовательно, гарантировать его работу в последующих версиях Windows нельзя. Еще раз повторим, что драйвер прямого доступа нужен для решения совершенно определенной задачи — быстрого переноса программ из Windows 98 в Windows NT/2000/XP. Но при написании новых программ пользоваться им не следует.

Глава 10

Использование функций Windows

Ищу поклонников Microsoft. Найду — убью.

Объявление в газете

Прямое обращение к портам, конечно, не составляет большого труда и очень напоминает DOS-программирование. Однако это скорее обходной маневр. Более корректно использовать функции Windows, которая предоставляет множество функций для работы с последовательными портами.

10.1. Обзор функций Windows для работы с последовательными портами

Полное описание функций Windows приводится в третьей части книги *(см. гл. 20, 21)*, а здесь мы рассмотрим основные принципы их использования.

Windows работает с портами так же как с файлами. Для открытия порта используется функция `CreateFile`, а для закрытия — `CloseHandle`. Для чтения и передачи данных также используются файловые функции `ReadFile` и `WriteFile` соответственно. В Windows NT/2000/XP можно использовать асинхронные функции `ReadFileEx` и `WriteFileEx` *(см. гл. 13)*. Кроме файловых функций для коммуникационных портов, Windows предоставляет также специальные функции.

Для использования порта необходимо получить его идентификатор — *дескриптор порта*, с помощью которого будут происходить все остальные обращения к порту. После использования надо вызвать функцию `CloseHandle` для освобождения дескриптора и закрытия порта. Windows не поддерживает разделение коммуникационных ресурсов, поэтому с момента вызова `CreateFile` до вызова `CloseHandle` программа получает порт в свое полное владение. Вызов `CreateFile` из другой программы, попытавшейся открыть уже занятый ресурс, вернет ошибку.

В отличие от файловых операций, перед началом работы с портом необходимо настроить такие параметры обмена, как: скорость обмена, параметры

четности, число стоп-бит и т. д. Кроме обычных параметров, уже изученных нами при работе в DOS, существует несколько специфичных для Windows параметров — тайм-ауты. Тайм-ауты позволяют настраивать как интервалы между принимаемыми байтами, так и общее время приема сообщения.

Основные параметры последовательного порта описываются структурой `DCB`, а тайм-ауты — структурой `COMMTIMEOUTS`. Настройка порта сводится к заполнению полей этих структур и вызову функций настройки `SetCommState` и `SetCommTimeouts`. Подробное описание полей всех структур можно найти в *гл. 20*.

Если заполнение полей структуры вызывает затруднение (а как видно из описания структуры, полей в ней очень много), можно воспользоваться функцией `BuildCommDCB`, которая позволяет заполнить поля структуры `DCB` на основе строки, по синтаксису аналогичной строке команды `mode` *(см. прил. 11)*. Кроме того, можно вызвать функцию `GetCommState`, возвращающую текущие параметры порта, изменить нужные из них и вызвать `SetCommState` для задания новых параметров.

С помощью функции `BuildCommDCBAndTimeouts` поля структур `DCB` и `COMMTIMEOUTS` можно заполнить одновременно.

Для настройки порта существует еще одна функция. В специальной динамической библиотеке (DLL) содержится функция `CommConfigDialog`, отображающая стандартное диалоговое окно настройки порта. Функция возвращает структуру `COMMCONFIG`, а непосредственное задание полученных параметров производится с помощью `SetCommConfig`. Так же как и для `DCB`, можно сначала вызвать функцию `GetCommConfig`, поменять нужные поля и вызвать `SetCommConfig` для установки новых параметров.

В диалоговом окне, отображаемом при вызове `CommConfigDialog`, присутствует кнопка **Reset To Default** (в русском варианте ее название переведено не совсем корректно как "Вернуть исходные значения"). При нажатии на нее поля диалогового окна замещаются на принятые в системе параметры по умолчанию. Чтобы считать и изменить параметры по умолчанию, применяются функции `GetDefaultCommConfig` и `SetDefaultCommConfig`. Заметим, что вызов этих функций не меняет настроек порта, а лишь записывает параметры по умолчанию во внутренние области коммуникационного драйвера.

10.2. Специальная настройка порта

В большинстве случаев достаточно тех настроек, которые дают структуры `DCB` и `COMMTIMEOUTS`. Однако при использовании специальных устройств, подключаемых к порту компьютера, может потребоваться более тонкая настройка.

Информацию о возможностях коммуникационного порта и драйвера дает структура COMMPROP. Подробное описание ее полей дается в *гл. 20*.

Получить информацию об устройстве в виде структуры COMMPROP можно с помощью функции GetCommProperties.

Для изменения длины внутренних буферов коммуникационного драйвера применяется функция SetupComm. Все что она делает, это устанавливает размеры в байтах очередей приема и передачи. Следует помнить, что это только рекомендуемые значения. Указанные размеры будут приняты драйвером к сведению, но драйвер может корректировать эти значения при необходимости или совсем отвергнуть их. В последнем случае функция SetupComm завершится с ошибкой. Обычно внешние устройства обмениваются с драйвером достаточно короткими сообщениями, и, следовательно, вызов SetupComm не требуется. Однако если внешнее устройство обменивается пакетами данных длиной несколько тысяч байт, рекомендуется установить соответствующие размеры очередей, чтобы избежать потери данных.

10.3. Получение состояния линий модема

В DOS для получения состояния индикаторов модема мы проверяли состояние порта MCR (адрес BasePort+4). В Windows для этого нам нужно будет вызывать функцию GetCommModemStatus. Подробное описание этой функции можно найти в *гл. 21*.

10.4. Используем функции Windows

Модифицируем нашу программу, заменив прямое обращение к портам вызовами функций Windows.

Внешний вид основной формы изменится не намного. Во-первых, простую кнопку **Инициализация** мы заменим двумя — **Открытие порта** и **Закрытие порта**. Нажатие этих кнопок позволит нам получать коммуникационный порт в полное владение и освободить его после работы. Во-вторых, мы заменим таблицу стандартных скоростей обмена набором констант CBR_xxx. Конечно, можно использовать любые числовые значения, но мы будем пользоваться только описанными в Windows константами скоростей обмена. И последнее изменение: два набора опций, касающихся контроля четности, мы заменим одним общим набором.

В результате наших изменений основная форма должна получиться, как показано на рис. 10.1.

Для доступа к порту мы напишем класс TComPort, скрывающий все тонкости работы с Windows API. Исходный код этого класса и его использование приводятся в листингах 10.1 и 10.2 соответственно. Заметим, что в некото-

рых случаях мы специально упрощали код, делая его, может быть, немного более примитивным, в угоду читаемости и простоты. В следующих главах мы будем усложнять этот класс, добавляя в него новые возможности.

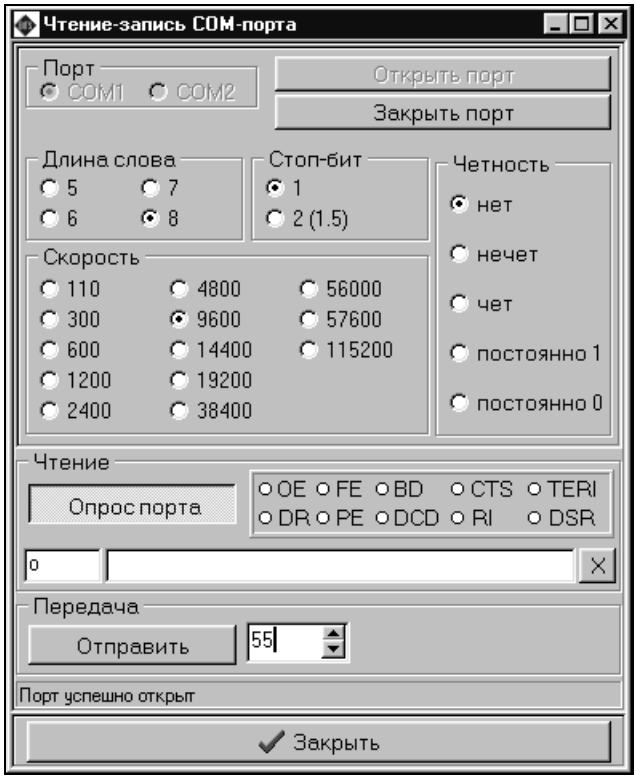

Рис. 10.1. Главное окно первой программы, использующей функции Windows

Листинг 10.1. Класс TComPort, использующий функции Windows

```
unit ComPort;

interface

uses
  Windows, SysUtils, Classes;

Type
 TBaudRate = (
```

```
   br110, br300, br600, br1200, br2400, br4800, br9600,
   br14400, br19200, br38400, br56000, br57600, br115200
);
TByteSize = (
 bs5, bs6, bs7, bs8
);
TParity = (
 ptNONE, ptODD, ptEVEN, ptMARK, ptSPACE
);
TStopbits = (
 sb1BITS, sb1HALFBITS, sb2BITS
);

TComPort = class(TComponent)
 Protected
   FBaseAddress : Word;      {базовый адрес порта}
   FHandle      : THandle;   {дескриптор порта}
   FBaudRate    : TBaudRate; {скорость обмена (бод)}
   FByteSize    : TByteSize; {число бит в байте}
   FParity      : TParity;   {четность}
   FStopbits    : TStopbits; {число стоп-бит}
   Procedure DoOpenPort; {открытие порта}
   Procedure DoClosePort; {закрытие порта}
   Procedure ApplyComSettings; {установка параметров порта}
   Procedure ApplyComTimeouts; {установка тайм-аутов порта}
 Private
   FComNumber: Integer;
   function GetConnected: Boolean;
   procedure SetConnected(const Value: Boolean);
   procedure SetComNumber(const Value: Integer);
   procedure SetBaudRate(const Value: TBaudRate);
   procedure SetByteSize(const Value: TByteSize);
   procedure SetParity(const Value: TParity);
   procedure SetStopbits(const Value: TStopbits);
 Public
   Constructor Create(AOwner : TComponent); override;
   Destructor  Destroy; override;
 Public
   {Открывает/закрывает порт}
```

```
    Procedure Open;
    Procedure Close;
    {Возвращает True, если порт открыт}
    Function  Connect : Boolean;
    {Возвращает структуру состояния порта ComStat, а в    }
    {переменной CodeError возвращается текущий код ошибки }
    Function  GetState(var CodeError : Cardinal) : TComStat;
    {Возвращает состояние модема}
    Function  GetModemState : Cardinal;
    {Читает один байт. Если байт прочитан успешно, возвращает True}
    Function  ReadByte(var B : Byte) : Boolean;
    {Передает один байт. В случае успеха возвращает True}
    Function  WriteByte(const B : Byte) : Boolean;
  Public
    {Номер порта. При изменении порт переоткрывается, если был открыт}
    Property ComNumber  : Integer read FComNumber    write SetComNumber;
    {Возвращает True, если порт открыт}
    Property Connected : Boolean read GetConnected write SetConnected;
    {Скорость обмена}
    Property BaudRate  : TBaudRate read FBaudRate write SetBaudRate;
    {Число бит в байте}
    Property ByteSize  : TByteSize read FByteSize write SetByteSize;
    {Четность}
    Property Parity    : TParity   read FParity   write SetParity;
    {Число стоп-бит}
    Property Stopbits  : TStopbits read FStopbits write SetStopbits;
 End;

implementation

Const
  WindowsBaudRates: array[br110..br115200] of DWORD = (
    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
    {CRB_128000, CBR_256000 — описаны в Windows.pas, но не используются}
  );

Const
  dcb_Binary              = $00000001;
```

```
  dcb_ParityCheck         = $00000002;
  dcb_OutxCtsFlow         = $00000004;
  dcb_OutxDsrFlow         = $00000008;
  dcb_DtrControlMask      = $00000030;
  dcb_DtrControlDisable   = $00000000;
  dcb_DtrControlEnable    = $00000010;
  dcb_DtrControlHandshake = $00000020;
  dcb_DsrSensivity        = $00000040;
  dcb_TXContinueOnXoff    = $00000080;
  dcb_OutX                = $00000100;
  dcb_InX                 = $00000200;
  dcb_ErrorChar           = $00000400;
  dcb_NullStrip           = $00000800;
  dcb_RtsControlMask      = $00003000;
  dcb_RtsControlDisable   = $00000000;
  dcb_RtsControlEnable    = $00001000;
  dcb_RtsControlHandshake = $00002000;
  dcb_RtsControlToggle    = $00003000;
  dcb_AbortOnError        = $00004000;
  dcb_Reserveds           = $FFFF8000;

Constructor TComPort.Create(AOwner : TComponent);
Begin
 Inherited Create(AOwner);
 FHandle:= INVALID_HANDLE_VALUE;
End;

Destructor  TComPort.Destroy;
Begin
 DoClosePort;
 Inherited Destroy;
End;

function TComPort.GetConnected: Boolean;
begin
 Result:= (FHandle <> INVALID_HANDLE_VALUE);
end;

procedure TComPort.SetConnected(const Value: Boolean);
begin
```

```
 If Value then DoOpenPort else DoClosePort;
end;

Function TComPort.Connect : Boolean;
Begin
 DoOpenPort;
 Result:= Connected;
End;

Procedure TComPort.Open;
Begin
 DoOpenPort;
End;

Procedure TComPort.Close;
Begin
 DoClosePort;
End;

procedure TComPort.SetComNumber(const Value: Integer);
var Active : Boolean;
begin
 If FComNumber = Value then Exit;
 Active:= Connected; {сохраним значение активности порта}
 If Active then DoClosePort; {закрыть порт перед изменением индекса}
 FComNumber:= Value; {устанавливаем новое значение номера порта}
 If Active then DoOpenPort; {открыть порт, если он был открыт}
end;

{открытие порта}
Procedure TComPort.DoOpenPort;
Var ComName : String;
Begin
 If Connected then Exit;

 {Для портов 1–9 можно использовать простые имена COM1–COM9, }
 { но для портов 10–256 надо писать полное имя. Для общности }
 { будем всегда использовать полное имя порта                 }
 ComName:= Format('\\.\COM%-d', [FComNumber]);
```

```
 {Открытие последовательного порта }
 FHandle:= CreateFile(
            PChar(ComName), {передаем имя открываемого порта}
            GENERIC_READ or GENERIC_WRITE, {ресурс для чтения и записи}
            0, { неразделяемый ресурс }
            nil, { Нет атрибутов защиты }
            OPEN_EXISTING, {вернуть ошибку, если ресурс не существует}
            FILE_ATTRIBUTE_NORMAL, {синхронный режим доступа}
            0 { Должно быть 0 для COM портов }
           ) ;
 {Если ошибка открытия порта — выход}
 If not Connected then Exit;
 {Инициализация порта}
 ApplyComSettings;
End;

{закрытие порта}
Procedure TComPort.DoClosePort;
Begin
 If not Connected then Exit;
 {Освобождение дескриптора}
 CloseHandle(FHandle);
 {Сброс дескриптора порта}
 FHandle:= INVALID_HANDLE_VALUE;
End;

{установка параметров порта}
Procedure TComPort.ApplyComSettings;
Var dcb: TDCB; {структура описания параметров порта}
Begin
 If not Connected then Exit;

 { Чистка структуры }
 FillChar(dcb, SizeOf(dcb), 0);
 { Пеле DCBLength должно содержать размер структуры }
 dcb.DCBLength:= SizeOf(dcb);
 { Скорость обмена (бод) }
 dcb.BaudRate := WindowsBaudRates[FBaudRate];
```

```
 { Windows не поддерживает небинарный режим работы последовательных
   портов }
 dcb.Flags    := dcb_Binary;
 { Число бит в байте }
 dcb.ByteSize := 5 + Ord(FByteSize);
 { Контроль четности }
 dcb.Parity   := Ord(FParity);
 { Число стоп бит }
 dcb.StopBits := Ord(FStopbits);
 { Установить новые настройки порта }
 SetCommState(FHandle, dcb);
End;

{установка тайм-аутов порта}
Procedure TComPort.ApplyComTimeouts;
Begin
 If not Connected then Exit;
End;

procedure TComPort.SetBaudRate(const Value: TBaudRate);
begin
 If FBaudRate = Value then Exit;
 FBaudRate:= Value;
 ApplyComSettings;
end;

procedure TComPort.SetByteSize(const Value: TByteSize);
begin
 If FByteSize = Value then Exit;
 FByteSize:= Value;
 ApplyComSettings;
end;

procedure TComPort.SetParity(const Value: TParity);
begin
 If FParity = Value then Exit;
 FParity := Value;
 ApplyComSettings;
end;
```

```
procedure TComPort.SetStopbits(const Value: TStopbits);
begin
 If FStopbits = Value then Exit;
 FStopbits := Value;
 ApplyComSettings;
end;

Function TComPort.GetState(var CodeError : Cardinal) : TComStat;
Begin
 {Возвращает структуру состояния порта и код ошибок}
 ClearCommError(FHandle, CodeError, @Result);
End;

Function  TComPort.ReadByte(var B : Byte) : Boolean;
Var RealRead : Cardinal;
Begin
 {Используем синхронное чтение. В случае успешного чтения}
 {функция ReadFile вернет True, а переменная RealRead,   }
 {хранящая число реально прочитанных байт, будет равна 1  }
 Result:= False;
 If ReadFile(FHandle, B, 1, RealRead, nil) then
  Result:= (RealRead = 1);
End;

Function TComPort.WriteByte(const B : Byte) : Boolean;
Var RealWrite : Cardinal;
Begin
 {Используем синхронную запись. В случае успешной передачи}
 {функция WriteFile вернет True, а переменная RealWrite,  }
 {хранящая число реально переданных байт, будет равна 1    }
 Result:= False;
 If WriteFile(FHandle, B, 1, RealWrite, nil) then
  Result:= (RealWrite = 1);
End;

Function TComPort.GetModemState : Cardinal;
Begin
 {Возвращает состояние модема}
```

```
 GetCommModemStatus(FHandle, Result);
End;

end.
```

Листинг 10.2. Использование класса TComPort

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls, Buttons, Spin, Mask, ToolEdit,
  ComPort, ComCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    btnClose: TBitBtn;
    ReadTimer: TTimer;
    ParamPanel: TPanel;
    rgPortIndex: TRadioGroup;
    rgDataLen: TRadioGroup;
    rgStopBits: TRadioGroup;
    rgBaud: TRadioGroup;
    rgParity: TRadioGroup;
    GroupBox1: TGroupBox;
    bRead: TSpeedButton;
    Panel3: TPanel;
    Bevel2: TBevel;
    DR: TShape;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
```

```
    Label9: TLabel;
    Label10: TLabel;
    Label8: TLabel;
    Label11: TLabel;
    OE: TShape;
    PE: TShape;
    FE: TShape;
    BD: TShape;
    DCD: TShape;
    RI: TShape;
    CTS: TShape;
    DSR: TShape;
    TERI: TShape;
    Label12: TLabel;
    eReadBuffer: TEdit;
    btnClearReadBuf: TSpeedButton;
    GroupBox2: TGroupBox;
    btnSend: TSpeedButton;
    seSendByte: TSpinEdit;
    btnOpenPort: TSpeedButton;
    btnClosePort: TSpeedButton;
    StatusBar: TStatusBar;
    eAvaibleBytes: TEdit;
    procedure FormCreate(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure bReadClick(Sender: TObject);
    procedure ReadTimerTimer(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnSendClick(Sender: TObject);
    procedure btnClearReadBufClick(Sender: TObject);
    procedure btnOpenPortClick(Sender: TObject);
    procedure btnClosePortClick(Sender: TObject);
    procedure rgDataLenClick(Sender: TObject);
    procedure rgStopBitsClick(Sender: TObject);
    procedure rgParityClick(Sender: TObject);
    procedure rgBaudClick(Sender: TObject);
  private
    { Компонент для доступа к порту. Будем создавать его }
    { динамически.                                        }
```

```
    FPort : TComPort;
  public
   procedure SetPortProperties;
   procedure SetEnabledControls;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

{ Создание компонента доступа к портам при создании формы }
procedure TForm1.FormCreate(Sender: TObject);
begin
 FPort:= TComPort.Create(Self);
 SetEnabledControls;
end;

{Вызывается при уничтожении формы (выходе из программы) }
procedure TForm1.FormDestroy(Sender: TObject);
begin
 FPort.Free;
end;

{Кнопка открытия порта}
procedure TForm1.btnOpenPortClick(Sender: TObject);
begin
 FPort.Close;
 FPort.ComNumber:= rgPortIndex.ItemIndex+1;
 SetPortProperties;
 FPort.Open;
 If FPort.Connected then
  StatusBar.Panels[0].Text:= 'Порт успешно открыт'
 Else
  StatusBar.Panels[0].Text:= 'Ошибка открытия порта';
 SetEnabledControls;
end;
```

```
procedure TForm1.btnClosePortClick(Sender: TObject);
begin
 FPort.Close;
 StatusBar.Panels[0].Text:= '';
 SetEnabledControls;
end;

procedure TForm1.SetEnabledControls;
Var Active : Boolean;
begin
 Active:= FPort.Connected;
 btnClosePort.Enabled:= Active;
 rgPortIndex.Enabled := not Active;
 btnOpenPort.Enabled := not Active;
 rgDataLen.Enabled   := Active;
 rgStopBits.Enabled  := Active;
 rgParity.Enabled    := Active;
 rgBaud.Enabled      := Active;
end;

{Задание свойств порта}
procedure TForm1.SetPortProperties;
begin
 rgDataLenClick(nil);
 rgStopBitsClick(nil);
 rgParityClick(nil);
 rgBaudClick(nil);
end;



{Кнопка закрытия программы}
procedure TForm1.btnCloseClick(Sender: TObject);
begin
 Close;
end;

{Кнопка опроса порта}
procedure TForm1.bReadClick(Sender: TObject);
```

```
begin
 ReadTimer.Enabled:= bRead.Down;
end;

Const {Цвета индикаторов}
 IColor : Array [Boolean] of TColor = (clWhite, clGreen);

procedure TForm1.ReadTimerTimer(Sender: TObject);
Var B : Byte; S : String; CurrentState : TComStat;
    AvaibleBytes, ErrCode, ModemState : Cardinal;
begin
 {Получает состояние порта }
 CurrentState:= FPort.GetState(ErrCode);
 { Отображаем число полученных, но еще не прочитанных байт}
 AvaibleBytes:= CurrentState.cbInQue;
 eAvaibleBytes.Text:= IntToStr(AvaibleBytes);

 If AvaibleBytes > 0 then begin
  If FPort.ReadByte(B) then begin { получение байта }
   S:= eReadBuffer.Text;
   S:= S + Format('$%X',[B]);
   If Length(S) > 40 then Delete(S, 1, 3);
   eReadBuffer.Text:= S;
  End;
 End;

 {Отображение индикаторов состояния порта}
 { Данные готовы   }
 DR.Brush.Color:= IColor[AvaibleBytes >  0];
 { Переполнение    }
 OE.Brush.Color:= IColor[(ErrCode and CE_OVERRUN ) <> 0];
 { Ошибка паритета }
 PE.Brush.Color:= IColor[(ErrCode and CE_RXPARITY) <> 0];
 { Ошибка кадра    }
 FE.Brush.Color:= IColor[(ErrCode and CE_FRAME   ) <> 0];
 { Обрыв линии     }
 BD.Brush.Color:= IColor[(ErrCode and CE_BREAK   ) <> 0];

 {Отображение индикаторов состояния модема}
 ModemState:= FPort.GetModemState;
```

```
 {Сигнал DCD (data carrier detect)}
 DCD.Brush.Color := IColor[(ModemState and MS_RLSD_ON) <> 0];
 {Состояние линии RI (ring indicator)}
 RI .Brush.Color := IColor[(ModemState and MS_RING_ON) <> 0];
 {Состояние линии DSR (data set ready)}
 DSR.Brush.Color := IColor[(ModemState and MS_DSR_ON ) <> 0];
 {Состояние линии CTS (clear to send)}
 CTS.Brush.Color := IColor[(ModemState and MS_CTS_ON ) <> 0];
 { Сигнал "спад огибающей" (окончание звонка) }
 { Не определяется в Windows }
 {TERI.Brush.Color:= IColor[(CurrentState and $04) <> 0];}
end;

{Очистить буфер чтения}
procedure TForm1.btnClearReadBufClick(Sender: TObject);
begin
 eReadBuffer.Text:= '';
end;

{Отправка байта}
procedure TForm1.btnSendClick(Sender: TObject);
begin
 FPort.WriteByte(seSendByte.Value);
end;

{Длина байта}
procedure TForm1.rgDataLenClick(Sender: TObject);
begin
 FPort.ByteSize:= TByteSize(rgDataLen.ItemIndex);
end;

{Число стоп-бит. Так как Windows оставляет контроль за}
{правильностью настроек на программиста, то запрещаем }
{выставлять 2 стоп-бита для не 5-битовых данных        }
procedure TForm1.rgStopBitsClick(Sender: TObject);
begin
 Case rgStopBits.ItemIndex of
  0: // 1 стоп-бит
   FPort.Stopbits:= sb1BITS;
```

```
  1: // 2 (1.5) стоп-бита
   If rgDataLen.ItemIndex = 0 then FPort.Stopbits:= sb1HALFBITS
    Else FPort.Stopbits:= sb2BITS;
 End;
end;

{Контроль четности}
procedure TForm1.rgParityClick(Sender: TObject);
begin
 FPort.Parity  := TParity(rgParity.ItemIndex);
end;

{Скорость передачи}
procedure TForm1.rgBaudClick(Sender: TObject);
begin
 FPort.BaudRate:= TBaudRate(rgBaud.ItemIndex);
end;

end.
```

Проделаем несколько экспериментов с созданной программой. Запустим, как и в прошлый раз, две копии нашей программы. В первом окне откроем порт COM1, выбрав в параметрах COM1 и нажав кнопку **Открыть порт**. Попробовав открыть тот же порт во втором окне, мы увидим сообщение **Ошибка открытия порта**. Как мы уже говорили, Windows не позволяет двум приложениям использовать один коммуникационный ресурс.

Откроем в первом окне порт COM1, а во втором COM2. Настроив порты на одинаковые характеристики, мы легко передадим байт из одного порта в другой (как и в предыдущей главе, в одном окне нажмем кнопку **Опрос порта**, а во втором — **Отправить**). Настроив порты на разные характеристики, можно увидеть индикацию ошибок четности и переполнения.

Остается отметить, что наша новая программа работает в любой версии Windows.

10.5. Несколько замечаний о контроле четности

Наша программа позволяет сделать несколько интересных наблюдений о принципах контроля четности.

Включим в программе передатчика контроль четности **чет**, а в программе приемника **нечет**. Отправим байт из первой программы во вторую. Видно, что мигают индикаторы **DR** ("Данные готовы") и **PE** ("Ошибка паритета").

Теперь выключим проверку четности (т. е. посылку контрольного бита) в передатчике, а в приемнике оставим контроль **чет**. Видно, что ошибка паритета будет зависеть от значения передаваемого байта. При передаче числа 26 ошибку не видно, а при передаче числа 24 ошибка однозначно возникает.

Контроль четности дополняет байт нужным битом — при контроле **чет** байт дополняется до четного числа бит, при контроле **нечет** — до нечетного числа бит, при контроле **постоянно 0** — всегда дополняется нулевым битом, а при контроле **постоянно 1** — всегда единичным битом.

Таким образом, контроль четности может распознать ошибку только в случае ошибки в одном бите. Если байт будет передан с двумя ошибочными битами, контроль четности не отработает. Для повышения достоверности данных необходимо использовать методы, описанные в *гл. 2*.

Глава 11

Использование потоков

Человеку свойственно ошибаться, но с помощью компьютера
это ему удается намного лучше.

Использование функций Windows для обращения к портам не избавляет нас от необходимости снова опрашивать порт в цикле, как мы делали это в среде DOS без прерываний. Конечно, в Windows мы воспользовались таймером, но у этого метода есть недостаток — сообщения таймера имеют низший приоритет и обрабатываются в последнюю очередь. В этой главе мы воспользуемся другими механизмами Windows и создадим для чтения порта отдельный поток.

11.1. Преимущества потоков

Опрос порта по таймеру имеет существенный недостаток. Сообщениям таймера присваивается низший приоритет, и они обрабатываются в последнюю очередь. В Windows 95/98, например, достаточно удерживать мышкой заголовок окна программы, и обработка таймера будет заморожена.

Избавиться от недостатка таймера позволит использование потоков. Потоки — отдельные процессы Windows. Потокам можно задавать разные приоритеты выполнения, приостанавливать их выполнение или совсем завершать их работу. Подробное описание работы с потоками выходит за рамки этой книги.

В Delphi процедура создания потоков очень проста. Достаточно создать класс от `TThread` и описать (перекрыть) метод `Execute`. При необходимости можно перекрыть конструктор и деструктор.

11.2. Создание потока опроса порта

Добавим в нашу последнюю программу из *гл. 10* создание потока для опроса порта и чтения данных.

Во-первых, удалим таймер из основной формы, он нам больше не потребуется. В класс TComPort добавим свойство ReadActive, управляющее активностью чтения порта. Кроме того, нам нужно будет добавить событие OnReadByte, обработчик которого будет вызываться при получении очередного байта. Разумеется, в реальной программе должно быть событие OnReadData, обработчик которого вызывается при получении очередной порции данных. Сейчас же нас больше интересует сам поток для чтения порта.

Изменения в коде основной формы показаны в листинге 11.1, а изменения в коде компонента показаны в листинге 11.2. Полный код вы можете найти на прилагаемом к книге компакт-диске.

Листинг 11.1. Изменения в коде основного модуля для использования потоков

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls, Buttons, Spin, Mask, ToolEdit,
  ComPort, ComCtrls;

type
  TForm1 = class(TForm)
... ... ... ... ... ... ... ... ... ... ... ... ...
  private
    { Компонент для доступа к порту. Будем создавать его }
    { динамически.                                       }
    FPort : TComPort;
    {Этот обработчик будет вызываться при получении байта}
    procedure OnReadByte(const B : Byte; PortState : TComStat;
                         ErrCode, ModemState : Cardinal);
  public
   procedure SetPortProperties;
   procedure SetEnabledControls;
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}

{ Создание компонента доступа к портам при создании формы }
procedure TForm1.FormCreate(Sender: TObject);
begin
 FPort:= TComPort.Create(Self);
 FPort.OnReadByte:= OnReadByte; {устанавливаем обработчик}
 SetEnabledControls;
end;

... ... ... ... ... ... ... ... ... ... ... ...

{Кнопка опроса порта}
procedure TForm1.bReadClick(Sender: TObject);
begin
 {Включить опрос порта}
 FPort.ReadActive:= bRead.Down;
end;

Const {Цвета индикаторов}
 IColor : Array [Boolean] of TColor = (clWhite, clGreen);

{Будет вызываться по приходу байта. Единственная неприятность -}
{состояние линий и модема будет отражать состояние на момент  }
{получения байта, а не отображаться постоянно. Для постоянного}
{отображения нужно создавать события OnModemStatusChange и    }
{OnPortStateChange. Однако сейчас наша задача проще.          }
procedure TForm1.OnReadByte(const B : Byte; PortState : TComStat;
                            ErrCode, ModemState : Cardinal);
Var S : String;  AvaibleBytes : Cardinal;
begin
 { Отображаем число полученных, но еще не прочитанных байт}
 AvaibleBytes:= PortState.cbInQue;
 eAvaibleBytes.Text:= IntToStr(AvaibleBytes);

 {Теперь нет необходимости проверять наличие байта для чтения,}
 {событие OnReadByte вызывается только после считывания байта }
```

```
 S:= eReadBuffer.Text;
 S:= S + Format('$%X',[B]);
 If Length(S) > 40 then Delete(S, 1, 3);
 eReadBuffer.Text:= S;
... ... ... ... ... ... ... ... ... ... ... ...
end;

... ... ... ... ... ... ... ... ... ... ... ...

end.
```

Листинг 11.2. Добавление потока для чтения данных в компонент TComPort

```
unit ComPort;

interface


uses
  Windows, SysUtils, Classes, Forms;

Type
... ... ... ... ... ... ... ... ... ... ... ...

 {тип события при получении байта}
 TReadByteEvent = procedure (const B : Byte; PortState : TComStat;
                             ErrCode, ModemState : Cardinal) of object;

 TComPort = class; {опережающее описание}

 {читающий поток}
 TReadThread = class(TThread)
   FOwner  : TComPort;
   FByte   : Byte;
   FPState : TComStat;
   FMState : Cardinal;
   FErrCode: Cardinal;
  Protected
```

```
   Procedure Execute; override;
   Procedure DoReadByte;
  Public
   Constructor Create(AOwner : TComPort);
 End;

 TComPort = class(TComponent)
  Protected
... ... ... ... ... ... ... ... ... ... ... ...
    FReadThread  : TReadThread; {читающий поток}
... ... ... ... ... ... ... ... ... ... ... ...
  Private
... ... ... ... ... ... ... ... ... ... ... ...
    procedure SetReadActive(const Value: Boolean);
    function  GetReadActive: Boolean;
... ... ... ... ... ... ... ... ... ... ... ...
   Public
    {Событие, вызываемое при получении байта}
    Property OnReadByte : TReadByteEvent read FOnReadByte write
FOnReadByte;
 End;

implementation

... ... ... ... ... ... ... ... ... ... ... ...

Constructor TComPort.Create(AOwner : TComponent);
Begin
 Inherited Create(AOwner);
 FHandle:= INVALID_HANDLE_VALUE;
 FReadThread:= nil;
End;

... ... ... ... ... ... ... ... ... ... ... ...

{открытие порта}
Procedure TComPort.DoOpenPort;
Var ComName : String;
```

```
Begin
 If Connected then Exit;
 ... ... ... ... ... ... ... ... ... ... ... ...
 {Инициализация порта}
 ApplyComSettings;
 {Создание читающего потока}
 If not Assigned(FReadThread) then
  FReadThread:= TReadThread.Create(Self);
End;

{закрытие порта}
Procedure TComPort.DoClosePort;
Begin
 If not Connected then Exit;
 {Замораживаем поток чтения}
 ReadActive:= False;
 {Уничтожение читающего потока}
 FReadThread.FreeOnTerminate:= True;
 FReadThread.Terminate;
 FReadThread:= nil;
 {Освобождение дескриптора порта}
 CloseHandle(FHandle);
 {Сброс дескриптора порта}
 FHandle:= INVALID_HANDLE_VALUE;
End;

... ... ... ... ... ... ... ... ... ... ... ...

{Активность чтения порта}
procedure TComPort.SetReadActive(const Value: Boolean);
begin
 If not Assigned(FReadThread) then Exit;

 If Value then begin
  If FReadThread.Suspended then FReadThread.Resume;
 End else begin
  If not FReadThread.Suspended then FReadThread.Suspend;
 End;
end;
```

```
function TComPort.GetReadActive: Boolean;
begin
 Result:= False;
 If Assigned(FReadThread) then
  Result:= not FReadThread.Suspended;
end;

{Класс TReadThread }
Constructor TReadThread.Create(AOwner : TComPort);
Begin
 Inherited Create(True);
 FOwner:= AOwner;
End;

Procedure TReadThread.Execute;
Var B : Byte; CurrentState : TComStat;
    AvaibleBytes, ErrCode, ModemState : Cardinal;
Begin
 Try
  With FOwner do begin
   While (not Terminated) and Connected do begin {пока порт открыт}

    {Получаем состояние порта (линий и модема)}
    CurrentState:= GetState(ErrCode);
    ModemState  := GetModemState;

    { Число полученных, но еще не прочитанных байт }
    AvaibleBytes:= CurrentState.cbInQue;
    { Проверка числа доступный байт}
    If AvaibleBytes > 0 then begin
     If ReadByte(B) then begin { получение байта }
      {сохраняем параметры вызова события}
      FByte   := B;
      FPState := CurrentState;
      FMState := ModemState;
      FErrCode:= ErrCode;
      {Вызываем событие OnReadByte. Для синхронизации с VCL}
      {надо вызвать метод Synchronize                      }
```

```
      Synchronize(DoReadByte);
     End;
    End;

    Sleep(1);
   End;
  End;
 Except End;
End;

{Вызывается для передачи события о приходе байта}
{в основной компонент через метод Synchronize   }
Procedure TReadThread.DoReadByte;
Begin
 With FOwner do begin
  If Assigned(FOnReadByte) then
   FOnReadByte(FByte, FPState, FErrCode, FMState);
 End;
End;

end.
```

При открытии порта мы создаем поток `TReadThread`, а при закрытии порта — уничтожаем его. Процедура закрытия порта (`DoClosePort`) немного усложнилась. Сначала мы должны уничтожить читающий поток и только после этого можем закрыть порт и освободить дескриптор.

Второй тонкий момент — вызов события `OnReadByte`. Его вызов вынесен в отдельную процедуру (`DoReadByte`) не случайно. Мы знаем, что в обработчике `OnReadByte`, находящемся в основной форме программы, мы будем использовать VCL-компоненты для отображения состояния порта и полученного байта. Но, с другой стороны, `OnReadByte` вызывается из другого потока, а, следовательно, поток основной формы и читающий поток должны быть синхронизированы. Синхронизация используемых в основной форме VCL-компонентов с читающим потоком достигается вызовом метода `Synchronize`. Этот метод имеет единственный параметр — адрес процедуры, подлежащей синхронизации. В нашем примере мы передаем адрес процедуры `DoReadByte`, в которой и вызываем событие `OnReadByte`.

Еще одно замечание. Поскольку событие `OnReadByte` вызывается только при получении нового байта, то и индикаторы статуса линии и модема будут обновляться только при получении очередного байта. Таким образом, инди-

кация будет отражать состояние порта на момент получения байта. При необходимости корректного отображения состояния порта нужно создавать события `OnPortStatusChange` и `OnModemStatusChange`.

И напоследок маленькая "ложка дегтя". Конечно, использование потоков сильно разгружает основной поток программы и позволяет опрашивать порт с большим приоритетом, но есть маленькая тонкость. Она заключается в вызове `Sleep(1)` в конце процедуры `Execute`. Этот вызов замораживает работу потока на 1 мс и отдает управление другим потокам системы. Однако это одновременно означает, что в течение 1 мс порт опрашиваться не будет! Если же мы уберем эту строку, наш поток будет "съедать" 100% процессорного времени, в чем легко убедиться, запустив системный монитор ресурсов (а Windows 98 просто "зависнет"). Вызов же заморозки процесса на 1 мс уменьшит загрузку процессора до 30—40%, что все равно довольно много. Решением этой проблемы мы займемся в следующей главе.

Глава 12

Функции асинхронного доступа и события

Скорость считается оптимальной, когда встречные
и попутные автомобили проносятся с одинаковой скоростью.

Использование потоков позволило нам разгрузить основную программу от постоянного опроса порта, а также обрабатывать порт независимо от действий в системе. Однако даже чтение порта в отдельном потоке еще не предел совершенства. Асинхронное чтение и применение событий позволит нам использовать ресурсы компьютера по минимуму — читающий поток будет "просыпаться" только в случае появления данных для чтения или других нужных нам событий.

12.1. Асинхронный доступ

При синхронных операциях функции `ReadFile` и `WriteFile` завершаются только после окончания операции или при ошибке. Асинхронный доступ подразумевает возможность начать операцию, а ее завершение обработать в другой момент времени. Таким образом, асинхронный вызов, например `ReadFile`, начнет операцию чтения и сразу же завершит выполнение самой функции `ReadFile`. Результат операции можно узнать, вызвав функцию `GetOverlappedResult`. Более того, асинхронная операция связывается со специальным объектом-событием, который позволяет дожидаться завершения операции, остановив выполнение потока, т. е. практически без затрат процессорного времени. Ожидание производится с помощью обычных функций синхронизации `WaitForSingleObject` и `WaitForMultipleObjects`.

Для использования асинхронных операций порт должен быть открыт с применением флага `FILE_FLAG_OVERLAPPED` при вызове функции `CreateFile`.

Если порт открыт без флага FILE_FLAG_OVERLAPPED, то использование асинхронных операций в Windows 2000 вызовет зависание программы.

Кстати, если порт открыт с флагом FILE_FLAG_OVERLAPPED, а программа использует синхронные операции, то Windows 98 выполнит операции записи, а Windows 2000 просто проигнорирует такой вызов. Синхронный вызов операции чтения может вызвать зависание программы в цикле бесконечного ожидания.

Для использования асинхронных операций необходимо внести изменения в код нашей программы, приведенной в предыдущей главе. Измененный код показан в листинге 12.1.

Листинг 12.1. Изменения в процедуре открытия порта для асинхронного доступа

```
{открытие порта}
Procedure TComPort.DoOpenPort;
Var ComName : String;
Begin
 If Connected then Exit;

 {Для портов 1–9 можно использовать простые имена  }
 {COM1–COM9, но для портов 10–256 надо писать      }
 {полное имя. Для общности будем всегда использовать}
 {полное имя порта.                                 }
 ComName:= Format('\\.\COM%-d', [FComNumber]);

 {Открытие последовательного порта }
 FHandle:= CreateFile(
   PChar(ComName), {передаем имя открываемого порта}
   GENERIC_READ or GENERIC_WRITE, {ресурс для чтения и записи}
   0, { не разделяемый ресурс }
   nil, { Нет атрибутов защиты }
   OPEN_EXISTING, {вернуть ошибку, если ресурс не существует}
  {асинхронный режим доступа}
  FILE_ATTRIBUTE_NORMAL or FILE_FLAG_OVERLAPPED,
  0 { Должно быть 0 для COM-портов }
 ) ;
 {Если ошибка открытия порта — выход}
 If not Connected then Exit;
```

```
 {Инициализация порта}
 ApplyComSettings;
 {Создание читающего потока}
 If not Assigned(FReadThread) then
  FReadThread:= TReadThread.Create(Self);
End;
```

Кроме открытия порта, нам нужно будет избавиться от синхронных операций чтения и записи данных. Код новой функции записи данных приводится в листинге 12.2, а операцией чтения мы займемся чуть позже.

Листинг 12.2. Асинхронная запись данных

```
Function TComPort.WriteByte(const B : Byte) : Boolean;
Var Signaled, RealWrite, BytesTrans : Cardinal;
    {структура для асинхронной записи}
    WriteOL : TOverLapped;
Begin
 Result:= False;

 {создание события для асинхронной записи}
 FillChar(WriteOL, SizeOf(WriteOL), 0);
 WriteOL.hEvent:= CreateEvent(nil, True, True, nil);

 Try
  {начало асинхронной записи}
  WriteFile(FHandle, B, 1, RealWrite, @WriteOL);
  {ожидание завершения асинхронной операции}
  Signaled:= WaitForSingleObject(WriteOL.hEvent, INFINITE);
  {получение результата асинхронной операции}
  Result  :=
   (Signaled = WAIT_OBJECT_0) and
   (GetOverlappedResult(FHandle, WriteOL, BytesTrans, False));
 Finally
  {освобождение дескриптора события}
  CloseHandle(WriteOL.hEvent);
 End;
End;
```

12.2. События последовательного порта

Опрос состояния порта в цикле — не самый оптимальный способ определения наличия данных для чтения. В DOS мы научились использовать прерывания, позволяющие вызывать нужное нам событие только при получении нового байта. Мы могли управлять причинами возникновения прерываний, настроив вызов прерываний, например, по изменению состояния линий модема.

В Windows также есть механизм, похожий на систему прерываний DOS. Функция `SetCommMask` позволяет настроить обработку следующих событий:

- обрыв линии;
- изменение состояния линии CTS;
- изменение состояния линии DSR;
- одна из ошибок: `CE_FRAME`, `CE_OVERRUN` или `CE_RXPARITY`;
- сигнал RI;
- изменение состояния RLSD;
- принят символ;
- получен специальный символ, описанный в блоке `DCB`;
- буфер посылки опустошен.

Отличий этого набора условий от набора, предоставляемого DOS, немного, но они достаточно существенны. Во-первых, Windows позволяет настраивать события от CTS, DSR, RI и RLSD раздельно. Во-вторых, допускается раздельная обработка ошибок обрыва линии и формата данных. И последнее новшество, присутствующее исключительно в Windows, — в блоке `DCB` можно задать специальный символ, по приходу которого будет генерироваться событие. Это удобно, например, при обработке символа начала посылки.

Для ожидания наступления события используется функция `WaitCommEvent`, подробное описание которой приводится в *гл. 21.*

Опережая события, отметим, что с помощью вызова `DeviceIoControl` можно установить еще одно условие: приемный буфер заполнен на 80% (флаг `SERIAL_EV_RX80FULL`, см. *разд. 22.16*).

12.3. Использование событий

Итак, возьмем наш пример из предыдущей главы и изменим функцию `Execute` читающего потока таким образом, чтобы читающий поток "просыпался" только при получении символа, не занимая процессорное время без надобности.

Все изменения, которые нам потребуется сделать, касаются только класса `TComPort` и читающего потока:

- в функцию открытия порта следует добавить флаг `FILE_FLAG_OVERLAPPED` (мы уже сделали это для использования асинхронной записи);
- создать сигнальный объект для асинхронных операций (для чтения нужно создать свой объект-событие);
- установить маску наступления событий с помощью `SetCommMask`;
- добавить код ожидания наступления события;
- переделать чтение байта в асинхронный вариант;
- добавить обработку асинхронного чтения.

Несмотря на внушительный список, изменений не так уж и много. Исходный код новой функции `Execute` приводится в листинге 12.3, а полный код программы можно найти на прилагаемом к книге компакт-диске.

Листинг 12.3. Асинхронная запись данных

```
{Основная функция потока}
Procedure TReadThread.Execute;
Var B : Byte; CurrentState : TComStat;
    AvaibleBytes, ErrCode, ModemState, RealRead : Cardinal;
    ReadOL : TOverLapped; {структура для асинхронного чтения}
    Signaled, Mask : DWORD;
    BytesTrans : DWORD; {не используется для WaitCommEvent}
Begin
 With FOwner do begin
  Try
   {создание события для асинхронного чтения}
   FillChar(ReadOL, SizeOf(ReadOL), 0);
   ReadOL.hEvent:= CreateEvent(nil, True, True, nil);

   {Маска событий, которые будет отслеживать читающий поток }
   {Пока это только получение символа                        }
   SetCommMask(FHandle, EV_RXCHAR);

   While (not Terminated) and Connected do begin {пока порт открыт}
    { Ждем одного из событий }
    WaitCommEvent(FHandle, Mask, @ReadOL);
```

```
  Signaled:= WaitForSingleObject(ReadOL.hEvent, INFINITE);

  If (Signaled  = WAIT_OBJECT_0) then begin
   If GetOverlappedResult(FHandle, ReadOL,
                          BytesTrans, False) then begin

     {после GetOverlappedResult в переменной mask,   }
     {которая передавалась в WaitCommEvent, появятся }
     {флаги произошедших событий, либо 0 в случае    }
     {ошибки.                                        }
     If (Mask and EV_RXCHAR) <> 0 then begin
      {Получаем состояние порта (линий и модема)}
      CurrentState:= GetState(ErrCode);
      ModemState  := GetModemState;

      { Число полученных, но еще не прочитанных байт}
      AvaibleBytes:= CurrentState.cbInQue;
      { Проверка числа доступных байт}
      If AvaibleBytes > 0 then begin
       If ReadFile(FHandle, B, 1, RealRead, @ReadOL) then begin
        {сохраняем параметры вызова события}
        FByte   := B;
        FPState := CurrentState;
        FMState := ModemState;
        FErrCode:= ErrCode;
        {Вызываем событие OnReadByte. Для синхронизации с VCL}
        {надо вызвать метод Synchronize.                     }
        Synchronize(DoReadByte);
       End;
      End;

     End;
    End;
   End;
  End;

Finally
 {закрытие дескриптора сигнального объекта}
```

```
    CloseHandle(ReadOL.hEvent);
    {Сброс события и маски ожидания}
    SetCommMask(FHandle, 0);
   End;
  End;
End;
```

Результат наших изменений легко увидеть, запустив программы приема и передачи данных и программу Менеджера ресурсов. Загрузка процессора не превышает 1% независимо от наличия данных!

Глава 13

Специальные коммуникационные функции

Программист — это человек, который сначала долго думает, чтобы потом ничего не делать.

Существует несколько специальных коммуникационных функций, используемых при реализации не-RS-232-протоколов. Эти функции пригодятся нам при работе с протоколом RS-485. Кроме того, Windows NT/2000 предоставляет еще несколько функций для асинхронной работы с портами. В этой главе мы рассмотрим их использование.

13.1. Выполнение дополнительных операций

Функция `EscapeCommFunction` позволяет выполнить дополнительные операции. Эта функция посылает код напрямую драйверу. Например, приложение может останавливать передачу с помощью посылки кода `SETBREAK` и возобновлять ее с помощью кода `CLRBREAK`. Те же операции могут быть выполнены с помощью функций `SetCommBreak` и `ClearCommBreak`.

Функция `EscapeCommFunction` может использоваться для реализации ручного управления сигналами модема. Например, с помощью кодов `CLRDTR` и `SETDTR` можно вручную управлять линией DTR (листинг 13.1). Заметим, что для ручного управления этими линиями необходима установка соответствующего режима в полях `fDtrControl` и `fRtsControl` блока `DCB` *(см. гл. 20)*.

Листинг 13.1. Управление линиями DTR и RTS

```
TComPort = class(TComponent)
... ... ... ... ... ... ... ... ... ... ... ... ...
 Public
  {Управление линией DTR}
  Procedure ToggleDTR(State : Boolean);
```

```
  {Управление линией RTS}
  Procedure ToggleRTS(State : Boolean);
End;

{Управление линией DTR}
Procedure TComPort.ToggleDTR(State : Boolean);
const Funcs: Array[Boolean] of Cardinal = (CLRDTR, SETDTR);
Begin
 If Connected then
  EscapeCommFunction(FHandle, Funcs[State]);
End;

{Управление линией RTS}
Procedure TComPort.ToggleRTS(State : Boolean);
const Funcs: Array[Boolean] of Cardinal = (CLRRTS, SETRTS);
Begin
 If Connected then
  EscapeCommFunction(FHandle, Funcs[State]);
End;
```

Кроме описанных в *гл. 21* кодов функций, в Windows 95/98 присутствует еще одна полезная, но недокументированная в MSDN функция с кодом 10. Ее вызов возвращает базовый адрес порта. Пример использования этой функции приведен в листинге 13.2.

Листинг 13.2. Недокументированный вызов EscapeCommFunction — базовый адрес порта (только в Windows 95/98)

```
TComPort = class(TComponent)
... ... ... ... ... ... ... ... ... ... ... ...
 Public
  Function GetBaseAddress : Word;
End;

{отображение базового адреса}
procedure TForm1.btnBaseAdrClick(Sender: TObject);
var S : String;
begin
 S:= IntToHex(FPort.GetBaseAddress, 3);
 MessageDlg('Базовый адрес порта '+S, mtInformation, [mbOK], 0);
end;
```

```
{получение базового адреса}
Function  TComPort.GetBaseAddress : Word;
Begin
 { недокументированный код функции — получение базового адреса в dx}
 { работает только для Windows 95/98 }
 EscapeCommFunction(FHandle, 10);
 Asm
  mov  Result, dx
 End;
End;
```

13.2. Прямое управление драйвером

Функция `DeviceIoControl` позволяет посылать расширенные команды управления напрямую драйверу ресурса, указывая ему выполнить определенные действия. Эта функция позволяет специфически конфигурировать драйвер, устанавливая характеристики, нужные устройству.

Мы уже использовали эту функцию в *гл. 9* для обращения к расширенному драйверу GiveIO. В этой главе мы рассмотрим использование данной функции для обращения к стандартному драйверу последовательного порта.

> ***Важно!***
>
> Возможность обращения к драйверу последовательного порта появилась только в Windows 2000.

Как мы видели, при вызове функции `DeviceIoConrol` в нее передаются код выполняемой операции, буфер входных параметров и буфер для выходных параметров. Коды операций и формат буферов для драйвера последовательного порта описаны в справочной части книги, в *гл. 22*.

Сначала мы сравним прямое обращение к драйверу и обычный метод доступа, а затем приступим к написанию настоящей программы. Для этого добавим в наш класс `TComPort` два метода, позволяющих считать и установить скорость обмена порта, которые будут использовать `DeviceIoControl`, как показано в листинге 13.3.

Листинг 13.3. Использование `DeviceIoControl` (Windows 2000)

```
unit ComPort;

interface

uses
  Windows, SysUtils, Classes, Forms;
```

```
... ... ... ... ... ... ... ... ...
TComPort = class(TComponent)
   ... ... ... ... ... ... ... ... ...
   Public {Дополнительные функции}
    ... ... ... ... ... ... ... ... ...
    Procedure Direct_SetBaudRate(Baud : Word);
    Function  Direct_GetBaudRate : Cardinal;
End;

... ... ... ... ... ... ... ... ...

{Константы для формирования функций}
{коммуникационного драйвера        }
Const
 FILE_DEVICE_SERIAL_PORT = $0000001b;

{Константы METHOD_xxx. Для COM-портов}
{необходимо использовать METHOD_BUFFERED}
Const
 METHOD_BUFFERED   = 0;
 METHOD_IN_DIRECT  = 1;
 METHOD_OUT_DIRECT = 2;
 METHOD_NEITHER    = 3;

{Метод доступа. Для COM-портов нужно}
{использовать FILE_ANY_ACCESS }
Const
 FILE_ANY_ACCESS   = $0000;
 FILE_READ_ACCESS  = $0001;
 FILE_WRITE_ACCESS = $0002;

{Формирование кода функции DeviceIoControl}
{для обращения к коммуникационному порту  }
function Get_Ctl_Code(Nr: Integer): Integer;
begin
  Result:=
    (FILE_DEVICE_SERIAL_PORT shl 16) or
    (FILE_ANY_ACCESS shl 14) or
    (Nr shl 2) or
```

```
    METHOD_BUFFERED;
end;

{Функция получения скорости обмена}
{с помощью прямого обращения к драйверу}
Function TComPort.Direct_GetBaudRate : Cardinal;
Var IOCTL_SERIAL_GET_BAUD_RATE, Skip : Cardinal;
Begin
 Result:= 0;

 {Код функции IOCTL_SERIAL_GET_BAUD_RATE = 20}
 IOCTL_SERIAL_GET_BAUD_RATE:= Get_Ctl_Code(20);

 {Обращаемся к драйверу. Если успешно, результат}
 {будет в переменной Result}
 If not DeviceIoControl(FHandle,
                IOCTL_SERIAL_GET_BAUD_RATE, nil, 0,
                @Result, SizeOf(Result), Skip, nil) then
 Begin
  MessageDlg('Ошибка '+IntToStr(GetLastError), mtError, [mbOK], 0);
 End;
End;

{Функция установки скорости обмена}
{с помощью прямого обращения к драйверу}
Procedure TComPort.Direct_SetBaudRate(Baud : Word);
Var IOCTL_SERIAL_SET_BAUD_RATE, Skip : Cardinal;
Begin
 {Код функции IOCTL_SERIAL_SET_BAUD_RATE = 01}
 IOCTL_SERIAL_SET_BAUD_RATE:= Get_Ctl_Code(01);

 {Обращаемся к драйверу}
 DeviceIoControl(FHandle,
       IOCTL_SERIAL_SET_BAUD_RATE, @Baud, SizeOf(Baud),
       nil, 0, Skip, nil);
End;
```

Добавив к нашей программе еще одну кнопку, например, **Получить скорость обмена**, мы вызовем нашу новую функцию (листинг 13.4).

Листинг 13.4. Вызов Direct_GetBaudRate (Windows 2000)

```
procedure TForm1.btnTest1Click(Sender: TObject);
begin
 MessageDlg(
  Format('Скорость обмена=%d',[ FPort.Direct_GetBaudRate],
  mtConfirmation, [mbOK], 0
 );
end;
```

Если сообщение отобразит правильную скорость обмена, значит, все прошло без ошибок.

Теперь посмотрим, какие преимущества дает прямое обращение к драйверу устройства. Очевидно, если мы напрямую обращаемся к драйверу, то экономим процессорное время. Убедиться в этом нам поможет небольшой тест, приведенный в листинге 13.5.

Листинг 13.5. Сравнение GetCommState и DiviceIoControl (Windows 2000)

```
Procedure TComPort.Test;
Var dcb: TDCB; i : LongInt; T1, T2 : Cardinal;
    IOCTL_SERIAL_GET_BAUD_RATE, Skip, Rate : Cardinal;
Begin
 {Инициализация dcb}
 FillChar(dcb, SizeOf(dcb), 0);
 dcb.DCBLength:= SizeOf(dcb);

 {Вызываем GetCommState 5000 раз}
 T1:= GetTickCount;
 For i:= 1 to 5000 do  {~156}
  GetCommState(FHandle, dcb);
 T2:= GetTickCount;
 {Печатаем затраченное время}
 MessageDlg(IntToStr(T2-T1), mtWarning, [mbOK], 0);

 {Получаем код функции}
 IOCTL_SERIAL_GET_BAUD_RATE:= Get_Ctl_Code(20);

 {Вызываем DeviceIoControl 5000 раз}
 T1:= GetTickCount;   {~30}
```

```
 For i:= 1 to 5000 do
   DeviceIoControl(FHandle, IOCTL_SERIAL_GET_BAUD_RATE, nil, 0,
       @Rate, SizeOf(Rate), Skip, nil);
 T2:= GetTickCount;
 {Печатаем затраченное время}
 MessageDlg(IntToStr(T2-T1), mtWarning, [mbOK], 0);
End;
```

В этом примере мы выполняем одну и ту же функцию — определение скорости обмена — двумя способами. Первый способ — обычный вызов `GetCommState`, а второй — прямое обращение к порту. Запустив эту программу на компьютере с процессором P4-2500 и Windows 2000, мы получили весьма впечатляющий результат. Затраты на первый цикл составляют примерно 140—160 мс, а на второй 20—35 мс. Прямое обращение быстрее обычного как минимум в 4 раза!

Теперь мы готовы к практическому использованию функции `DeviceIoControl`. Модуль, показанный в листинге 13.6, работает с портом в режиме on-line, позволяя менять настройки и управлять линиями порта. В отличие от примеров предыдущей главы, этот модуль не хранит текущее состояние настроек порта.

Листинг 13.6. Модуль ComPortOnline (Windows 2000)

```
unit ComPortOnline;

interface


uses
  Windows, SysUtils, Classes, Forms;

Type
 TByteSize = (
  bs5, bs6, bs7, bs8
 );
 TParity = (
  ptNONE, ptODD, ptEVEN, ptMARK, ptSPACE
 );

 TStopbits = (
```

```
 sb1BITS, sb1HALFBITS, sb2BITS
);

TComPortOnline = class(TComponent)
 Protected
   FBaseAddress : Word;      {базовый адрес порта}
   FHandle      : THandle;   {дескриптор порта}
   FByteSize    : TByteSize; {число бит в байте}
   FStopbits    : TStopbits; {число стоп-бит}
   Procedure DoOpenPort; {открытие порта}
   Procedure DoClosePort; {закрытие порта}
 Private
   FComNumber: Integer;
   function  GetConnected: Boolean;
   procedure SetComNumber(const Value: Integer);
   procedure SetBaudRate(const Value: Cardinal);
   function  GetBaudRate : Cardinal;
   function  GetLineControl(const Index: Integer): Byte;
   procedure SetLineControl(const Index: Integer; const Value: Byte);
   function GetByteSize: TByteSize;
   function GetParity: TParity;
   function GetStopbits: TStopbits;
   procedure SetByteSize(const Value: TByteSize);
   procedure SetParity(const Value: TParity);
   procedure SetStopbits(const Value: TStopbits);
   function GetDTRRTS(const Index: Integer): Boolean;
   procedure SetDTRRTS(const Index: Integer;
                       const Value: Boolean);
   function GetModemState(const Index: Integer): Boolean;
   procedure SetModemState(const Index: Integer;
                       const Value: Boolean);
 Public
   Constructor Create(AOwner : TComponent); override;
   Destructor  Destroy; override;
 Public
   {Открывает/закрывает порт}
   Procedure Open;
   Procedure Close;
```

```
Function GetTest : Cardinal;
  Public
    Property Connected : Boolean read GetConnected;
    {Номер порта. При изменении порт переоткрывается,}
    {если был открыт}
    Property ComNumber  : Integer  read  FComNumber
                                   write SetComNumber
                                   stored False;
    {Скорость обмена}
    Property BaudRate  : Cardinal  read  GetBaudRate
                                   write SetBaudRate
                                   stored False;

    {Число бит в байте}
    Property ByteSize  : TByteSize read  GetByteSize
                                   write SetByteSize
                                   stored False;
    {Четность}
    Property Parity    : TParity   read  GetParity
                                   write SetParity
                                   stored False;
    {Число стоп-бит}
    Property Stopbits  : TStopbits read  GetStopbits
                                   write SetStopbits
                                   stored False;
    {Управление линией DTR}
    Property DTR       : Boolean   index 0 read  GetDTRRTS
                                           write SetDTRRTS
                                           stored False;

    {Управление линией DTR}
    Property RTS       : Boolean   index 1 read  GetDTRRTS
                                           write SetDTRRTS
                                           stored False;
    {Управление линией OUT1}
    Property OUT1      : Boolean   index 0 read  GetModemState
                                           write SetModemState
                                           stored False;
```

```
    {Управление линией OUT2}
    Property OUT2      : Boolean   index 1 read  GetModemState
                                             write SetModemState
                                             stored False;
 End;

implementation

uses Dialogs;

Type
 TSERIAL_LINE_CONTROL = packed record
  StopBits   : Byte;
  Parity     : Byte;
  WordLength : Byte;
 End;

{Константы для GetCtrlCode}
Const
 FILE_DEVICE_SERIAL_PORT = $0000001b;
 METHOD_BUFFERED         = 0;
 FILE_ANY_ACCESS         = $0000;

function GetCtrlCode(NFunction : Integer): Integer;
begin
  Result:=
     (FILE_DEVICE_SERIAL_PORT shl 16) or
     (FILE_ANY_ACCESS shl 14) or
     (NFunction shl 2) or
     METHOD_BUFFERED;
end;

Function SetByteMask(Value : Cardinal; Mask : Byte; State : Boolean) :
Cardinal;
Begin
 If State then begin
  Result:= Value or Mask;
 End else begin
```

```
  Result:= Value and ($FF-Mask);
 End;
End;

{Конструктор}
Constructor TComPortOnline.Create(AOwner : TComponent);
Begin
 Inherited Create(AOwner);
 FHandle:= INVALID_HANDLE_VALUE;
 FComNumber:= -1;
End;

{Деструктор}
Destructor  TComPortOnline.Destroy;
Begin
 DoClosePort;
 Inherited Destroy;
End;

{Возвращает состояние соединения}
function TComPortOnline.GetConnected: Boolean;
begin
 Result:= (FHandle <> INVALID_HANDLE_VALUE);
end;

{Открытие порта}
Procedure TComPortOnline.Open;
Begin
 DoOpenPort;
End;

{Закрытие порта}
Procedure TComPortOnline.Close;
Begin
 DoClosePort;
End;
```

```
{Установка номера порта}
procedure TComPortOnline.SetComNumber(const Value: Integer);
var Active : Boolean;
begin
 Active:= Connected; {сохраним значение активности порта}
 If Active then DoClosePort; {закрыть порт перед изменением индекса}
 FComNumber:= Value; {устанавливаем новое значение номера порта}
 If Active then DoOpenPort; {открыть порт, если он был закрыт}
end;

{открытие порта}
Procedure TComPortOnline.DoOpenPort;
Var ComName : String;
Begin
 If Connected then Exit;

 {Для портов 1–9 можно использовать простые имена COM1–COM9,  }
 {но для портов 10-256 надо писать полное имя. Для общности   }
 {будем всегда использовать полное имя порта                  }
 ComName:= Format('\\.\COM%-d', [FComNumber]);

 {Открытие последовательного порта }
 FHandle:= CreateFile(
            PChar(ComName), {передаем имя открываемого порта}
            GENERIC_READ or GENERIC_WRITE, {ресурс для чтения и записи}
            0, { неразделяемый ресурс }
            nil, { Нет атрибутов защиты }
            OPEN_EXISTING, {вернуть ошибку, если ресурс не существует}
            FILE_ATTRIBUTE_NORMAL or FILE_FLAG_OVERLAPPED, {асинхронный
режим доступа}
            0 { Должно быть 0 для COM-портов }
           ) ;
End;

{закрытие порта}
Procedure TComPortOnline.DoClosePort;
Begin
 If not Connected then Exit;
```

```
 {Освобождение дескриптора порта}
 CloseHandle(FHandle);
 {Сброс дескриптора порта}
 FHandle:= INVALID_HANDLE_VALUE;
End;

{== Get/Set BaudRate}
function TComPortOnline.GetBaudRate : Cardinal;
Var Skip : Cardinal;
begin
 Result:= 0;
 DeviceIoControl(FHandle, GetCtrlCode(20), nil, 0,
                 @Result, SizeOf(Result), Skip, nil);
end;

procedure TComPortOnline.SetBaudRate(const Value: Cardinal);
Var Skip : Cardinal;
begin
 DeviceIoControl(FHandle, GetCtrlCode(1), @Value, SizeOf(Value),
                nil, 0, Skip, nil);
end;

function TComPortOnline.GetLineControl(const Index: Integer): Byte;
var L : TSERIAL_LINE_CONTROL; Skip : Cardinal;
begin
 DeviceIoControl(FHandle, GetCtrlCode(21), nil, 0,
                 @L, SizeOf(L), Skip, nil);
 Case Index of
  0: Result:= L.WordLength;
  1: Result:= L.Parity;
  2: Result:= L.StopBits;
 End;
end;

procedure TComPortOnline.SetLineControl(const Index: Integer;
  const Value: Byte);
var L : TSERIAL_LINE_CONTROL; Skip : Cardinal;
```

```
begin
 DeviceIoControl(FHandle, GetCtrlCode(21), nil, 0,
                  @L, SizeOf(L), Skip, nil);
 Case Index of
  0: L.WordLength:= Value;
  1: L.Parity    := Value;
  2: L.StopBits  := Value;
 End;
 DeviceIoControl(FHandle, GetCtrlCode(3), @L, SizeOf(L),
                 nil, 0, Skip, nil);
end;


function TComPortOnline.GetByteSize: TByteSize;
begin
 Result:= TByteSize(GetLineControl(0)– 5);
end;

function TComPortOnline.GetParity: TParity;
begin
 Result:= TParity(GetLineControl(1));
end;

function TComPortOnline.GetStopbits: TStopbits;
begin
 Result:= TStopbits(GetLineControl(2));
end;

procedure TComPortOnline.SetByteSize(const Value: TByteSize);
begin
 SetLineControl(0, Byte(Value)+5);
end;

procedure TComPortOnline.SetParity(const Value: TParity);
begin
 SetLineControl(1, Byte(Value));
end;

procedure TComPortOnline.SetStopbits(const Value: TStopbits);
begin
```

```
 SetLineControl(2, Byte(Value));
end;

function TComPortOnline.GetDTRRTS(const Index: Integer): Boolean;
var _DTRRST, Skip : Cardinal;
begin
 DeviceIoControl(FHandle, GetCtrlCode(30), nil, 0,
                  @_DTRRST, SizeOf(_DTRRST), Skip, nil);
 Case Index of
  0: Result:= (_DTRRST and 1) <> 0; // DTR
  1: Result:= (_DTRRST and 2) <> 0; // RTS
 End;
end;

procedure TComPortOnline.SetDTRRTS(const Index: Integer;
  const Value: Boolean);
var _DTRRST, Skip : Cardinal; Code : Integer;
begin
 If Value then begin
  Case Index of
   0: Code:=  9; // IOCTL_SERIAL_SET_DTR
   1: Code:= 12; // IOCTL_SERIAL_SET_RTS
  End;
 End else begin
  Case Index of
   0: Code:= 10; // IOCTL_SERIAL_CLR_DTR
   1: Code:= 13; // IOCTL_SERIAL_CLR_RTS
  End;
 End;
 DeviceIoControl(FHandle, GetCtrlCode(Code), nil, 0,
                     nil, 0, Skip, nil);
end;

Function TComPortOnline.GetTest : Cardinal;
var Skip : Cardinal;
begin
 DeviceIoControl(FHandle, GetCtrlCode(37), nil, 0,
                  @Result, SizeOf(Result), Skip, nil);
end;
```

```
function TComPortOnline.GetModemState(const Index: Integer): Boolean;
var _MState, Skip : Cardinal;
begin
 DeviceIoControl(FHandle, GetCtrlCode(37), nil, 0,
                  @_MState, SizeOf(_MState), Skip, nil);
 Case Index of
  0: Result:= (_MState and 4) <> 0; // OUT1
  1: Result:= (_MState and 8) <> 0; // OUT2
 End;
end;

procedure TComPortOnline.SetModemState(const Index: Integer;
  const Value: Boolean);
var _MState, Skip : Cardinal;
begin
 DeviceIoControl(FHandle, GetCtrlCode(37), nil, 0,
                  @_MState, SizeOf(_MState), Skip, nil);
 Case Index of
  0: SetByteMask(_MState, 4, Value); // OUT1
  1: SetByteMask(_MState, 8, Value); // OUT2
 End;
 DeviceIoControl(FHandle, GetCtrlCode(38),@_MState, SizeOf(_MState), nil,
0, Skip, nil);
end;

end.
```

Как видно, обращение к свойствам порта не представляет большой сложности и сводится к простой установке свойств. Нам кажется, особых комментариев здесь не требуется. Отметим только, что чтение и установка свойств порта производятся напрямую, без сохранения текущего состояния, и, по этой причине, для всех свойств нашего нового класса выставлены спецификаторы `Stored:= False`, означающие отключение сохранения значения этих свойств в DFM-файле.

Для тестирования нового модуля мы создадим небольшую программу, исходный код которой показан в листинге 13.7, а внешний вид формы — на рис. 13.1.

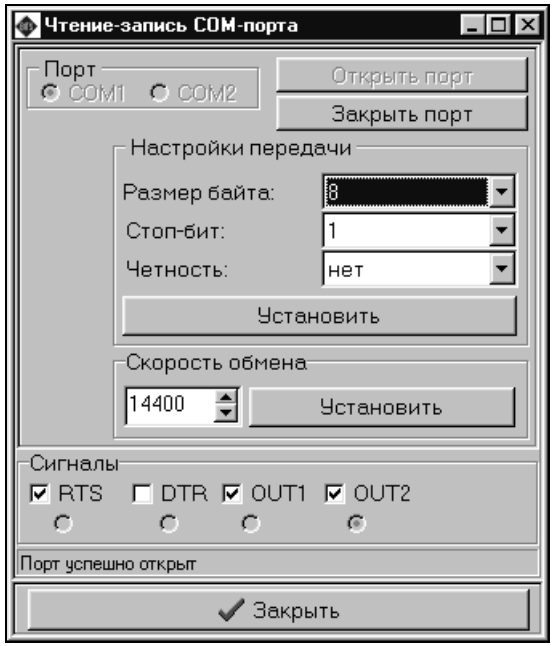

Рис. 13.1. Окно программы прямого обращения к драйверу порта

Листинг 13.7. Модуль `ComPortOnline` (Windows 2000)

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
Controls,Forms, Dialogs, ExtCtrls, StdCtrls, Buttons, Spin, Mask,
ComCtrls,Comportonline;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    btnClose: TBitBtn;
    ParamPanel: TPanel;
    rgPortIndex: TRadioGroup;
```

```
    btnOpenPort: TSpeedButton;
    btnClosePort: TSpeedButton;
    StatusBar: TStatusBar;
    GroupBox3: TGroupBox;
    BaudRate: TSpinEdit;
    btnBaudRate: TButton;
    GroupBox4: TGroupBox;
    Label1: TLabel;
    cbDataLen: TComboBox;
    cbStopBits: TComboBox;
    Label2: TLabel;
    cbParity: TComboBox;
    Label13: TLabel;
    btnLineProperties: TButton;
    GroupBox1: TGroupBox;
    cbRTS: TCheckBox;
    cbDTR: TCheckBox;
    Timer1: TTimer;
    _DTR: TRadioButton;
    _RTS: TRadioButton;
    cbOUT1: TCheckBox;
    cbOUT2: TCheckBox;
    _OUT1: TRadioButton;
    _OUT2: TRadioButton;
    procedure FormCreate(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnOpenPortClick(Sender: TObject);
    procedure btnClosePortClick(Sender: TObject);
    procedure btnLinePropertiesClick(Sender: TObject);
    procedure btnBaudRateClick(Sender: TObject);
    procedure cbRTSClick(Sender: TObject);
    procedure cbDTRClick(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure cbOUT1Click(Sender: TObject);
  private
    { Компонент для доступа к порту. Будем создавать его}
    { динамически.                                       }
    FPort : TComPortOnline;
```

```
  public
   procedure SetEnabledControls;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

{ Создание компонента доступа к портам при создании формы }
procedure TForm1.FormCreate(Sender: TObject);
begin
 FPort:= TComPortOnline.Create(Self);
 SetEnabledControls;
end;

{Вызывается при уничтожении формы (выходе из программы)}
procedure TForm1.FormDestroy(Sender: TObject);
begin
 FPort.Free;
end;

{Кнопка открытия порта}
procedure TForm1.btnOpenPortClick(Sender: TObject);
begin
 FPort.ComNumber:= rgPortIndex.ItemIndex+1;
 FPort.Open;
 If FPort.Connected then
  StatusBar.Panels[0].Text:= 'Порт успешно открыт'
 Else
  StatusBar.Panels[0].Text:= 'Ошибка открытия порта';
 SetEnabledControls;
 {Отображение текущего состояния порта}
 If FPort.Connected then begin
  cbDTR.Checked := FPort.DTR;
  cbRTS.Checked := FPort.RTS;
  cbOUT1.Checked:= FPort.OUT1;
```

```
  cbOUT2.Checked:= FPort.OUT2;
  BaudRate.Value:= FPort.BaudRate;
  cbDataLen.ItemIndex := Byte(FPort.ByteSize);
  cbStopBits.ItemIndex:= Byte(FPort.Stopbits);
  cbParity.ItemIndex  := Byte(FPort.Parity  );
 End;
end;

procedure TForm1.btnClosePortClick(Sender: TObject);
begin
 FPort.Close;
 StatusBar.Panels[0].Text:= '';
 SetEnabledControls;
end;

procedure TForm1.SetEnabledControls;
Var Active : Boolean;
begin
 Active:= FPort.Connected;
 btnClosePort.Enabled:= Active;
 rgPortIndex.Enabled := not Active;
 btnOpenPort.Enabled := not Active;
 cbDataLen.Enabled   := Active;
 cbStopBits.Enabled  := Active;
 cbParity.Enabled    := Active;
 BaudRate.Enabled    := Active;
 btnBaudRate.Enabled := Active;
end;

{Кнопка закрытия программы}
procedure TForm1.btnCloseClick(Sender: TObject);
begin
 Close;
end;

{Установка базового адреса}
procedure TForm1.btnBaudRateClick(Sender: TObject);
begin
```

```
 FPort.BaudRate:= BaudRate.Value;
end;

{Установка характеристик обмена}
procedure TForm1.btnLinePropertiesClick(Sender: TObject);
begin
 FPort.Parity:= TParity(cbParity.ItemIndex);
 FPort.ByteSize:= TByteSize(cbDataLen.ItemIndex);
 Case cbStopBits.ItemIndex of
  0: // 1 стоп-бит
   FPort.Stopbits:= sb1BITS;
  1: // 2 (1.5) стоп-бита
   If cbDataLen.ItemIndex = 0 then FPort.Stopbits:= sb1HALFBITS
    Else FPort.Stopbits:= sb2BITS;
 End;
end;

{Управление RTS}
procedure TForm1.cbRTSClick(Sender: TObject);
begin
 FPort.RTS:= cbRTS.Checked;
end;

{Управление DTR}
procedure TForm1.cbDTRClick(Sender: TObject);
begin
 FPort.DTR:= cbDTR.Checked;
end;

procedure TForm1.cbOUT1Click(Sender: TObject);
begin
 FPort.OUT1:= cbOUT1.Checked;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
 If not FPort.Connected then Exit;
 _RTS.Checked := FPort.RTS;
 _DTR.Checked := FPort.DTR;
```

```
 _OUT1.Checked:= FPort.OUT1;
 _OUT2.Checked:= FPort.OUT2;
end;

end.
```

13.3. Обнаружение всех портов системы

Очень часто необходимо отобразить список всех портов, доступных в системе. Например, для выбора порта при подключении устройства. Конечно, можно в цикле перебрать порты от COM1 до COM256 и пробовать их открывать, но, во-первых, это просто неудобно, а во-вторых, порт может быть занят и не откроется, а программа воспримет это как отсутствие порта.

Для перечисления всех доступных портов используется функция `EnumPorts`. Хотя в документации она описывается как функция для работы с портами принтера, она возвращает все порты, доступные в системе. Заметим только, что в Delphi для ее использования требуется подключить модуль `WinSpool`.

Подробное описание параметров функции `EnumPorts` приведено в *гл. 20*, а в этой главе мы покажем пример ее использования.

Добавим к нашему классу `TComPort` еще два метода. Метод `EnumComPorts` будет возвращать нам список всех коммуникационных портов в системе, а `EnumComPortsEx` не только имена портов, но и их описания.

Код функций и пример их использования приведены в листинге 13.8. Нам потребуется также добавить на основную форму кнопку вызова функции и `TListBox` для отображения списка портов.

Листинг 13.8. Перечисление всех COM-портов, доступных в системе

```
unit Unit1;

interface
... ... ... ... ... ... ... ... ... ... ... ...
{Перечислить доступные порты}
procedure TForm1.btnEnumPortsClick(Sender: TObject);
begin
 {! Добавим на форму TListBox для отображения списка}
 TComPort.EnumComPortsEx(ListBox1.Items);
end;
```

```
unit ComPort;

interface

uses
  Windows, SysUtils, Classes, Forms;

TComPort = class(TComponent)
   Public {Дополнительные функции}
    ... ... ... ... ... ... ... ... ... ... ... ...
    {Перечисление всех COM-портов}
    Class procedure EnumComPorts(Ports: TStrings);
    Class procedure EnumComPortsEx(Ports: TStrings);
 End;

implementation

uses
  Dialogs, WinSpool;

... ... ... ... ... ... ... ... ... ... ... ...

{ перечисление имен всех доступных коммуникационных портов}
class procedure TComPort.EnumComPorts(Ports: TStrings);
var
  BytesNeeded, Returned, I: DWORD;
  Success: Boolean;
  PortsPtr: Pointer;
  InfoPtr: PPortInfo1;
  TempStr: string;
begin
 {Запрос размера нужного буфера}
 Success := EnumPorts(nil, 1, nil, 0, BytesNeeded, Returned);

 If (not Success) and (GetLastError = ERROR_INSUFFICIENT_BUFFER) then
begin
  {Отводим нужный блок памяти}
  GetMem(PortsPtr, BytesNeeded);
  Try
```

```
   {Получаем список имен портов}
   Success := EnumPorts(nil, 1, PortsPtr, BytesNeeded, BytesNeeded,
Returned);

   {Переписываем имена в StringList, отсеивая не-COM-порты}
   For I := 0 to Returned – 1 do begin
    InfoPtr := PPortInfo1(DWORD(PortsPtr) + I * SizeOf(TPortInfo1));
    TempStr := InfoPtr^.pName;
    If Copy(TempStr, 1, 3) = 'COM' then
     Ports.Add(TempStr);
   End;
  Finally
   {Освобождаем буфер}
   FreeMem(PortsPtr);
  End;
 End;
end;

{ перечисление всех доступных коммуникационных портов и их описаний}
class procedure TComPort.EnumComPortsEx(Ports: TStrings);
var
  BytesNeeded, Returned, I: DWORD;
  Success: Boolean;
  PortsPtr: Pointer;
  InfoPtr: PPortInfo2;
  TempStr: string;
  Description : string;
begin
 {Запрос размера нужного буфера}
 Success := EnumPorts(nil, 2, nil, 0, BytesNeeded, Returned);

 If (not Success) and (GetLastError = ERROR_INSUFFICIENT_BUFFER) then
begin
  {Отводим нужный блок памяти}
  GetMem(PortsPtr, BytesNeeded);
  Try
   {Получаем список имен портов и их описания}
   Success := EnumPorts(nil, 2, PortsPtr, BytesNeeded, BytesNeeded,
Returned);
```

```
    {Переписываем имена в StringList, отсеивая не-COM-порты}
    For I := 0 to Returned - 1 do begin
     InfoPtr := PPortInfo2(DWORD(PortsPtr) + I * SizeOf(TPortInfo2));
     TempStr := InfoPtr^.pPortName;

     {Добавляем описание порта, если оно есть}
     Description:= '';
     If InfoPtr^.pDescription <> nil then
      Description:= ' ' + InfoPtr^.pDescription;

     {Переписываем имена в StringList, отсеивая не-COM-порты}
     If Copy(TempStr, 1, 3) = 'COM' then begin
      TempStr:= TempStr + Description;
      Ports.Add(TempStr);
     End;
    End;
   Finally
    {Освобождаем буфер}
    FreeMem(PortsPtr);
   End;
  end;
end;
```

Запустив программу и нажав кнопку отображения списка портов, мы получим, например, такой результат:

```
COM1: Последовательный порт
COM2: Последовательный порт
COM3: Genius GM56PCI-L Modem Card
```

Прежде чем перейти к следующему разделу, еще одно маленькое замечание относительно реализации новых функций. Мы описали эти функции как `class procedure`. Это позволяет вызывать их, не создавая экземпляр конкретного класса, т. е. просто `TComPort.EnumComPortsEx()`.

13.4. Имена портов

В Windows NT/2000/XP устройства, такие, как диски, а также последовательные и параллельные порты, имеют внутренние имена и, необязательно одно или несколько внешних *DOS-имен*. Ядро и драйверы используют только внутренние имена, а прикладные программы Win32 используют DOS-имена.

Внутренние имена имеют формат поддиректории в папке Device, например, `\Device\CDRom0` или `\Device\Seral0`. Однако прикладные программы не могут использовать внутренние имена и должны использовать внешние, такие как "A:", "C:", "COM1", "LPT1". Несколько DOS-имен могут указывать на одно и то же устройство, однако обратное утверждение не верно. Одно имя может ссылаться только на одно устройство. Так, например, последовательный порт `\Device\Serial0` может иметь имена COM1 и MyCOM.

Хотя прикладные программы не могут использовать внутренние имена, они могут получать, добавлять и удалять DOS-имена для внутренних имен устройств. Эти операции выполняются с помощью функций `QueryDosDevice` и `DefileDosDevice`, которые описаны в *гл. 21*.

Код программы, демонстрирующей использование этих функций, приведен в листинге 13.9.

Листинг 13.9. Работа с DOS-именами устройств

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, ComCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    StatusBar: TStatusBar;
    lbNameList: TListBox;
    Panel1: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    NTDeviceName: TEdit;
    DOSDeviceName: TEdit;
    btnGetName: TButton;
    btnAddName: TButton;
    btnDelName: TButton;
    btnGetList: TButton;
    procedure btnGetNameClick(Sender: TObject);
    procedure btnAddNameClick(Sender: TObject);
```

```
    procedure btnDelNameClick(Sender: TObject);
    procedure btnGetListClick(Sender: TObject);
    procedure lbNameListDblClick(Sender: TObject);
  private
  public
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

{Получение NT-имени по DOS-имени}
procedure TForm1.btnGetNameClick(Sender: TObject);
Var Result : Array [1..MAX_PATH] of Char;
begin
 If (QueryDosDevice(PChar(DOSDeviceName.Text),
                    @Result, MAX_PATH) <> 0) then
  NtDeviceName.Text:= StrPas(@Result)
 Else
  NtDeviceName.Text:= 'Устройство не существует';
end;

{Добавить DOS-имя для NT-имени}
procedure TForm1.btnAddNameClick(Sender: TObject);
begin
 If not DefineDosDevice(
   DDD_RAW_TARGET_PATH,
   PChar(DosDeviceName.Text),
   PChar(NtDeviceName.Text))
 then
  StatusBar.Panels[0].Text:= 'Ошибка добавления имени';
end;

{Удалить DOS-имя}
procedure TForm1.btnDelNameClick(Sender: TObject);
Var Result : Array [1..MAX_PATH] of Char;
```

```
begin
 {Ищем NT-имя для удаляемого DOS-имени}
 If not(QueryDosDevice(PChar(DOSDeviceName.Text),
                       @Result, MAX_PATH) <> 0) then begin
  StatusBar.Panels[0].Text:= 'DOS-имя не определено';
  Exit;
 End;

 {Удаляем DOS-имя}
 If not DefineDosDevice(
       DDD_RAW_TARGET_PATH or
       DDD_REMOVE_DEFINITION or
       DDD_EXACT_MATCH_ON_REMOVE,
   PChar(DosDeviceName.Text),
   PChar(NtDeviceName.Text)
 ) then
  StatusBar.Panels[0].Text:= 'Ошибка удаления имени';
end;

{Получение всех имен устройства}
procedure TForm1.btnGetListClick(Sender: TObject);
var BufSize : Cardinal; P, PName : Pointer; SName : String;
begin
 {Очищаем предыдущий список}
 lbNameList.Items.Clear;

 {Размер буфера}
 BufSize:= 10240;
 {Распределяем память для буфера}
 GetMem(P, BufSize);
 {Запрашиваем список имен}
 If QueryDosDevice(nil, P, BufSize) <> 0 then begin
  {Цикл по всем именам...}
  PName:= P;
  While (True) do begin
   SName:= StrPas(PName);
   If SName = '' then Break;
   {Добавляем в список}
```

```
    lbNameList.Items.Add(SName);
    {Переход к следующему устройству}
    {Сдвигаем указатель на следующую строку}
    PName:= Pointer(LongInt(PName) + Length(SName)+1);
   End;
  End;
  {Освобождаем буфер}
  FreeMem(P);
end;

{Сортировка списка по двойному щелчку}
procedure TForm1.lbNameListDblClick(Sender: TObject);
begin
  lbNameList.Sorted:= True;
  lbNameList.Sorted:= False;
end;

end.
```

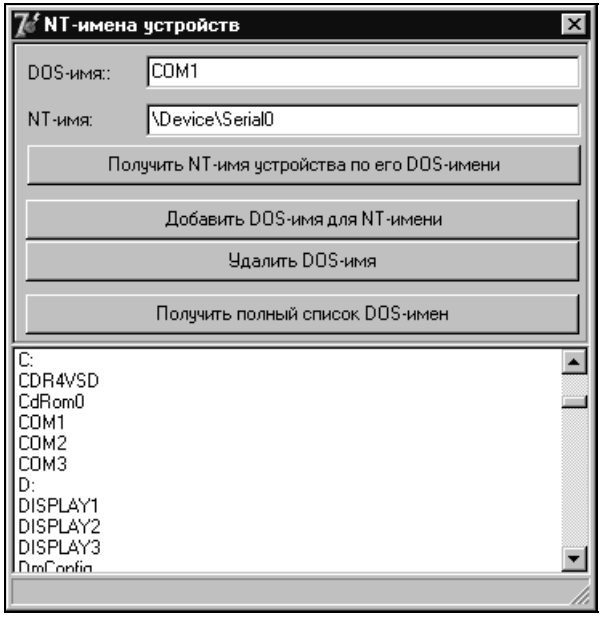

Рис. 13.2. Окно программы работы с именами устройств

Основное диалоговое окно нашей тестовой программы показано на рис. 13.2. Введем в верхнее поле строку **COM1** и нажмем кнопку **Получить NT-имя устройства по его DOS-имени**. В поле **NT-имя** должна отобразиться строка, похожая на "`\Device\Serial0`".

Теперь введем в поле **DOS-имя** какое-нибудь новое имя, например, MyDevice и нажмем кнопку **Добавить DOS-имя для NT-имени**. Результат мы можем увидеть, нажав кнопку **Получить полный список DOS-имен** или **Получить NT-имя устройства по его DOS-имени**.

Следует отметить, что присвоение одному коммуникационному порту двух или более имен не дает возможности открывать этот порт из нескольких программ одновременно. Открыв порт COM1 в одной программе, мы не сможем открыть порт MyDevice из другой, если они ссылаются на одно физическое устройство `\Device\Serial0`.

В Windows 2000 и выше имеется возможность напрямую задавать имя коммуникационного порта, точнее говоря, выбирать его номер. Это можно сделать в диалоговом окне **Advanced Settings for COM*xx*** (Дополнительные установки для порта COM*xx*) окна Менеджера устройств (Device Manager). Вид диалогового окна показан на рис. 13.3.

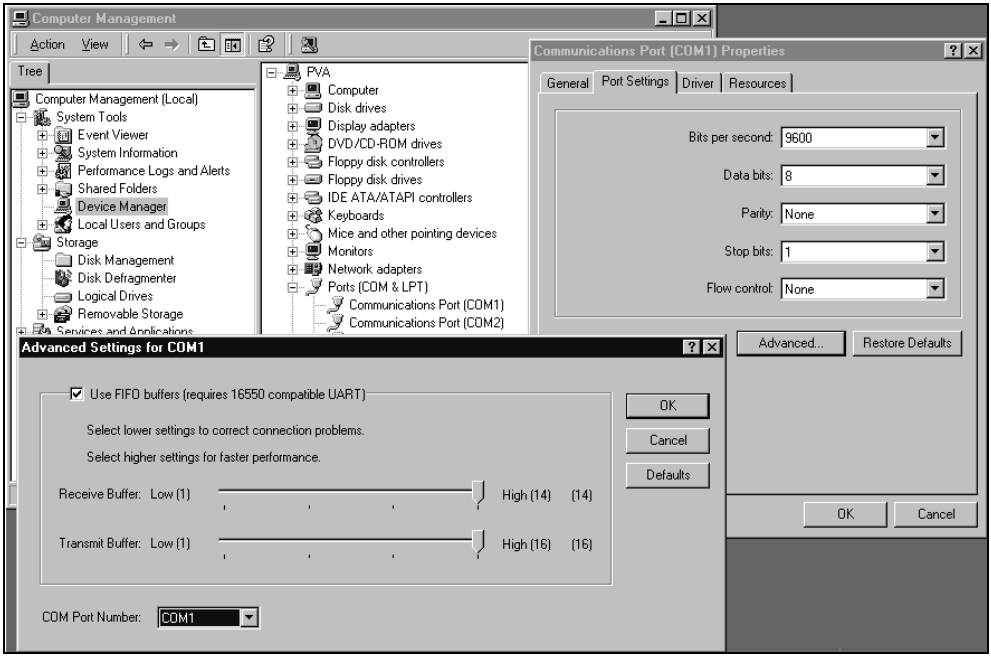

Рис. 13.3. Выбор имени порта в Windows 2000

13.5. APC-функции Windows 2000/XP

Механизм *асинхронного вызова процедур* (APC, Asynchronous Procedure Calls) позволяет задать адрес процедуры, которая будет вызываться системой при завершении асинхронной операции.

Подробное рассмотрение механизма и функций APC выходит за рамки этой книги. Рассмотрим только использование трех "основных" функций — `ReadFileEx`, `WriteFileEx` и `SleepEx`. Все APC-функции могут использоваться только в Windows NT/2000/XP (они присутствуют и в Windows 98, но не могут использоваться для коммуникационных портов).

По принципу действия эти функции похожи на прерывания в DOS. Вызов этих функций стартует асинхронную операцию, а при завершении этой операции вызывается определенная функция, адрес которой задается как параметр. Для того чтобы функция завершения была вызвана, необходимо, чтобы поток находился в так называемом "тревожном состоянии" (alertable wait state). Перевод потока в такое состояние производится с помощью вызова "расширенной" функции ожидания `SleepEx`, а также функциями `WaitForSingleObjectEx` и `WaitForMultipleObjectsEx`.

Разумеется, для использования этих функций порт должен быть открыт с помощью флага `FILE_FLAG_OVERLAPPED`.

В листинге 13.10 приводится код функций приема и передачи байта, использующих APC-вызовы вместо обычных асинхронных функций. Полный код программ можно найти на прилагаемом компакт-диске.

Листинг 13.10. Использование APC-функций Windows 2000

```
unit ComPort;

interface


uses
  Windows, SysUtils, Classes, Forms;

... ... ... ... ... ... ... ... ... ... ... ... ...

Function TComPort.WriteByte(const B : Byte) : Boolean;
Var WriteOL : TOverLapped; {структура для асинхронной записи}

    {Callback-процедура, вызываемая после завершения передачи}
```

```
    Procedure OnCompletionWrite(dwErrorCode,
                                dwNumberOfBytesTransfered : Cardinal;
                                var lpOverlapped : TOverlapped
                               ); far; stdcall;
    begin
     MessageBeep(0);
    end;

Begin
 Result:= False;
 {создание события для асинхронной записи}
 FillChar(WriteOL, SizeOf(WriteOL), 0);
 WriteOL.hEvent:= CreateEvent(nil, True, True, nil);
 {асинхронная отправка байта}
 WriteFileEx(FHandle, @B, 1, WriteOL, @OnCompletionWrite);
 SleepEx(INFINITE, True);
 {освобождение дескриптора события}
 CloseHandle(WriteOL.hEvent);
End;
... ... ... ... ... ... ... ... ... ... ... ...

{Основная функция потока}
Procedure TReadThread.Execute;
Var ReadOL : TOverLapped; {структура для асинхронного чтения}

    {Callback-процедура, вызывающаяся при получении байта}
    Procedure OnCompletionRead(dwErrorCode, dwNumberOfBytesTransfered :
Cardinal; var lpOverlapped : TOverlapped); far; stdcall;
    begin
     {К сожалению, не удалось вызвать Synchronize}
     {из callback-функции, т. к. она выполняется}
     {в рамках одного потока}
    end;

Begin
 With FOwner do
  While (not Terminated) and Connected do begin {пока порт открыт}
   {Запуск операции асинхронного чтения}
   ReadFileEx(FHandle, @FByte, 1, @ReadOL, @OnCompletionRead);
```

```
    {Ожидание завершения операции}
    SleepEx(INFINITE, True);
    {Сюда мы попадем, когда байт будет принят}
    Synchronize(DoReadByte);
   End;
End;

{Вызывается для передачи события о приходе байта}
{в основной компонент через метод Synchronize   }
Procedure TReadThread.DoReadByte;
Begin
 With FOwner do begin
  If Assigned(FOnReadByte) then
   FPState:= FOwner.GetState(FErrCode);
   FMState:= FOwner.GetModemState;
   FOnReadByte(FByte, FPState, FErrCode, FMState);
 End;
End;
```

Функции обратного вызова должны иметь спецификатор `stdcall`. Заметим, что передачу байта основному потоку мы производим не в функции обратного вызова (хотя она и вызывается при получении байта), а в теле потока, т. к. вызвать `Synchronize` и обращение к объектам другого потока из функции `OnCompletionRead` невозможно. И еще одно замечание. Система не использует параметр `hEvent` в структуре `Overlapped`, передаваемой в функции `WriteFileEx` и `ReadFileEx`. Это поле можно использовать под свои нужды.

Глава 14

Реализация протоколов обмена

Люди делятся на 10 категорий:
тех, кто понимает двоичное счисление,
и тех, кто не понимает.

До сих пор мы принимали и передавали всего один байт. В реальной программе необходимо принимать и передавать значительно больше информации. В этой главе мы займемся реализацией различных протоколов обмена.

14.1. Прием и передача простого пакета

Как уже отмечалось, во всех предыдущих примерах мы принимали и передавали только один байт. В этом разделе мы научимся производить обмен целым буфером, а для проверки работоспособности нашей новой программы мы добавим методы приема и передачи строк.

Простой пакет данных — это последовательность байтов, заканчивающаяся нулем. Для передачи пакета данных этого достаточно. А вот с приемом будет сложнее. Одним из параметров функции `ReadFile` является число байтов, которые мы хотим получить. В следующих главах такой величиной станет длина пакета данных, но пока мы просто добавим в класс `TComPort`, который мы использовали в *гл. 12*, свойство `InBufSize`. Это свойство будет задавать *размер входной очереди*.

Для передачи последовательности байтов мы заменим метод `WriteByte` на `WriteBuffer`, с помощью которого создадим дополнительную функцию `WriteString`, удобную для проверки работоспособности программы.

Кроме того, нам потребуется добавить специальное свойство `WaitForFullBuffer`, управляющее моментом чтения данных с порта. Если `WaitForFullBuffer` будет равно `True`, мы будем читать данные с порта только при заполнении всего буфера (т. е. когда в буфере накопится не менее `InBufSize` байт), а если `False`, то чтение данных мы будем производить при получении хоть какого-нибудь количества байтов.

Для проверки приема и передачи буферов мы будем использовать прием и передачу строк. Для этого потребуется внести некоторые изменения в основную форму, а именно:

- удалим индикаторы состояния линии и модема (здесь они нам не потребуются);
- в окне передачи данных заменим компонент `TSpinEdit` на `TEdit`, а в окне приема — компонент `TEdit` заменим на `TListBox`. Это позволит нам передавать вместо одного байта целую строку и отображать полученные строки;
- кнопка **X** будет очищать весь буфер приема;
- кнопка **Отправить** будет передавать целую строку;
- событие `OnReadByte` необходимо заменить на `OnReadPacket`, что позволит нам добавлять полученные строки в `TListBox`, который будет служить буфером приема.

Изменения, которые потребуется внести в класс `TComPort`, приведены в листинге 14.1. Полный код программы можно найти на прилагаемом компакт-диске.

Листинг 14.1. Прием и передача строки символов. Модуль TComPort

```
unit ComPort;

interface


uses
  Windows, SysUtils, Classes, Forms;

  ... ... ... ... ... ... ... ... ... ... ... ...

 {тип события при получении байта}
 TReadPacketEvent = procedure (const Packet : Pointer;
        const Size : Integer; const ErrCode: Cardinal) of object;

 TComPort = class; {опережающее описание класса}

 {читающий поток}
 TReadThread = class(TThread)
   FOwner    : TComPort;
```

```
   FInBuffer : Pointer; {входной буфер}
   FErrCode  : Cardinal;{сохраняет код ошибок}
   FCount    : Integer; {реально прочитанное число байт}
  Protected
   Procedure Execute; override;
   Procedure DoReadPacket;
  Public
   Constructor Create(AOwner : TComPort);
   Destructor  Destroy; override;
 End;

 TComPort = class(TComponent)
  Protected
  ... ... ... ... ... ... ... ... ... ... ... ...
    FInBufSize   : Cardinal; {размер входной очереди}
    FWaitFullBuffer : Boolean; {ожидание наполнения буфера}
  Public
    {Размер входного буфера}
    Property InBufSize  : Cardinal read  FInBufSize
                                  write SetInBufSize;
    {Ожидание наполнения буфера}
    Property WaitFullBuffer : Boolean read FWaitFullBuffer
                                     write FWaitFullBuffer;
  Public
    {Событие, вызываемое при получении байта}
    Property OnReadPacket : TReadPacketEvent read  FOnReadPacket
                                             write FOnReadPacket;
    ... ... ... ... ... ... ... ... ... ... ... ...
  End;

... ... ... ... ... ... ... ... ... ... ... ...

Constructor TComPort.Create(AOwner : TComponent);
Begin
 Inherited Create(AOwner);
 FHandle:= INVALID_HANDLE_VALUE;
 FReadThread:= nil;
 FInBufSize := 10;
```

```
 FWaitFullBuffer:= False;
End;

Destructor  TComPort.Destroy;
Begin
 {Закрываем порт и соединение}
 DoClosePort;
 Inherited Destroy;
End;

{Изменение размера входного буфера}
procedure TComPort.SetInBufSize(const Value: Cardinal);
begin
 {Разрешаем менять только при выключенном соединении}
 If Connected then Exit;
 {Нельзя задать нулевое или неверное значение размера}
 If Value <= 0 then Exit;
 {Запоминаем новый размер буфера}
 FInBufSize:= Value;
end;

{Передача строки}
Function  TComPort.WriteString(const S : String) : Boolean;
Begin
 WriteBuffer(PChar(S), Length(S));
End;

{Передача буфера}
Function  TComPort.WriteBuffer(const P : PChar; const Size : Integer) :
Boolean;
Var Signaled, RealWrite, BytesTrans : Cardinal;
    WriteOL : TOverLapped; {структура для асинхронной записи}
Begin
 Result:= False;
 If P = nil then Exit;

 {создание события для асинхронной записи}
 FillChar(WriteOL, SizeOf(WriteOL), 0);
 WriteOL.hEvent:= CreateEvent(nil, True, True, nil);
```

```
 Try
   {начало асинхронной записи}
   WriteFile(FHandle, P^, Size, RealWrite, @WriteOL);
   {ожидания завершения асинхронной операции}
   Signaled:= WaitForSingleObject(WriteOL.hEvent, INFINITE);
   {получение результата асинхронной операции}
   Result  :=
      (Signaled = WAIT_OBJECT_0) and
      (GetOverlappedResult(FHandle, WriteOL, BytesTrans, False));
 Finally
  {освобождение дескриптора события}
  CloseHandle(WriteOL.hEvent);
 End;
End;

... ... ... ... ... ... ... ... ... ... ... ...

{Класс TReadThread }
Constructor TReadThread.Create(AOwner : TComPort);
Begin
 Inherited Create(True);
 FOwner:= AOwner;
 {Создаем новый буфер}
 GetMem(FInBuffer, FOwner.FInBufSize);
End;

Destructor  TReadThread.Destroy;
Begin
 {Освобождаем буфер}
 FreeMem(FInBuffer);
 Inherited Destroy;
End;

{Основная функция потока}
Procedure TReadThread.Execute;
Var B : Byte; CurrentState : TComStat;
    AvaibleBytes, ErrCode, ModemState, RealRead : Cardinal;
    ReadOL : TOverLapped; {структура для асинхронного чтения}
    Signaled, Mask : DWORD;
```

```
    BytesTrans : DWORD; {не используется для WaitCommEvent}
    bReadable : Boolean; {готовность к чтению данных}
Begin
 With FOwner do begin
  Try
    {создание события для асинхронного чтения}
    FillChar(ReadOL, SizeOf(ReadOL), 0);
    ReadOL.hEvent:= CreateEvent(nil, True, True, nil);

    {Маска событий, которые будет отслеживать читающий поток }
    {Пока это только получение символа                        }
    SetCommMask(FHandle, EV_RXCHAR);

    {пока порт открыт}
    While (not Terminated) and Connected do begin
     { Ждем одного из событий }
     WaitCommEvent(FHandle, Mask, @ReadOL);

     Signaled:= WaitForSingleObject(ReadOL.hEvent, INFINITE);

     If (Signaled  = WAIT_OBJECT_0) then begin
      If GetOverlappedResult(FHandle, ReadOL,
                             BytesTrans, False) then begin

       {после GetOverlappedResult в переменной mask,}
       {которая передавалась в WaitCommEvent, появятся флаги}
       {произошедших событий, либо 0 в случае ошибки.}
       If (Mask and EV_RXCHAR) <> 0 then begin
        {Получаем состояние порта (линий и модема)}
        CurrentState:= GetState(ErrCode);
        ModemState  := GetModemState;

        { Число полученных, но еще не прочитанных байт}
        AvaibleBytes:= CurrentState.cbInQue;
        { Проверка числа доступных байт}
        If FWaitFullBuffer then begin
         {ждать только полного буфера}
         bReadable:= AvaibleBytes >= FInBufSize;
        End else begin
```

```
        {ждать любого числа байт}
         bReadable:= AvaibleBytes > 0;
        End;

        If bReadable then begin
         {Чистка буфера}
         ZeroMemory(FInBuffer, FInBufSize);
         If ReadFile(FHandle, FInBuffer^,
                     Min(FInBufSize, AvaibleBytes),
                     RealRead, @ReadOL) then begin
          {сохраняем параметры вызова события}
          FErrCode:= ErrCode;
          FCount  := RealRead;
          {Вызываем событие OnReadByte. Для синхронизации с VCL}
          {надо вызвать метод Synchronize                      }
          Synchronize(DoReadPacket);
         End;
        End;

       End;
      End;
     End;
    End;

  Finally
   {закрытие дескриптора сигнального объекта}
   CloseHandle(ReadOL.hEvent);
   {Сброс события и маски ожидания}
   SetCommMask(FHandle, 0);
  End;
 End;

End;

{Вызывается для передачи события о приходе байта}
{в основной компонент через метод Synchronize   }
Procedure TReadThread.DoReadPacket;
Begin
 With FOwner do begin
```

```
  If Assigned(OnReadPacket) then
   OnReadPacket(FInBuffer, FCount, FErrCode);
 End;
End;

end.
```

Перед созданием теста программы для нового модуля еще немного модифицируем сам модуль. Нам будет удобнее, если вместо модуля и динамического создания компонента доступа к порту мы будем использовать обычные компоненты Delphi.

14.2. Создание компонента

Прежде чем заняться созданием более сложных протоколов обмена, немного модифицируем наши программу и класс. В предыдущих главах мы строили примеры, исходя из аналогичных DOS-программ. В этой главе мы предлагаем окончательно избавиться от всего лишнего и значительно упростить наш класс.

Во-первых, для редактирования свойств порта в Windows уже есть специальная функция `CommConfigDialog`, поэтому нет необходимости использовать свои диалоговые окна (эта функция подробно описана в *гл. 21*).

Во-вторых, до сих пор мы использовали динамическое создание нашего класса, в то время как Delphi предоставляет более удобный механизм — компоненты.

Создадим компонент, основанный на `TComPort`. Для этого нам потребуется:

- добавить класс `TComProp`, который будет хранить настройки нашего порта;
- добавить метод, вызывающий диалоговое окно редактирования свойств порта;
- добавить специальный редактор свойств порта из Инспектора объектов;
- создать ресурс с иконкой нашего компонента для палитры компонентов;
- зарегистрировать наш класс в палитре компонентов.

Подробное описание процесса создания компонентов выходит за рамки этой книги. Всех интересующихся мы отсылаем к бестселлеру Рея Конопки [7]. Изменения в коде класса `TComPort` показаны в листинге 14.2, а полный код и файл ресурса ComPort.dcr можно найти на прилагаемом компакт-диске.

Листинг 14.2. Компонент `TComPort`

```
unit ComPort;

interface
```

```
uses
  Windows, SysUtils, Classes, Forms;

 ... ... ... ... ... ... ... ... ... ... ... ... ...

 {Класс свойств порта }
 TComProp = class(TPersistent)
  Private
   FBaudRate    : TBaudRate; {скорость обмена (бод)}
   FByteSize    : TByteSize; {число бит в байте}
   FParity      : TParity;   {четность}
   FStopbits    : TStopbits; {число стоп-бит}
   {возвращает структуру DCB из свойств порта}
   function  GetDCB: TDCB;
   {устанавливает свойства порта из структуры DCB}
   procedure SetDCB(const Value: TDCB);
  Public
   Property DCB : TDCB read GetDCB write SetDCB;
  Published
   {Скорость обмена}
   Property BaudRate  : TBaudRate read FBaudRate write FBaudRate;
   {Число бит в байте}
   Property ByteSize  : TByteSize read FByteSize write FByteSize;
   {Четность}
   Property Parity    : TParity   read FParity   write FParity;
   {Число стоп-бит}
   Property Stopbits  : TStopbits read FStopbits write FStopbits;
 End;

 TComPort = class(TComponent)
  Protected
    ... ... ... ... ... ... ... ... ... ... ... ... ...
    FComProp     : TComProp; {свойства порта}
    ... ... ... ... ... ... ... ... ... ... ... ... ...
  Public
    ... ... ... ... ... ... ... ... ... ... ... ... ...
    {Диалог настройки параметров порта}
    Function  CommDialog : Boolean;
```

```
  Published
    {Номер порта}
    Property ComNumber  : Integer read FComNumber
                                  write SetComNumber;
    {Параметры порта}
    Property ComProp : TComProp read FComProp write FComProp;
    {Размер входного буфера}
    Property InBufSize  : Cardinal read FInBufSize
                                   write SetInBufSize;
    {Ожидание наполнения буфера}
    Property WaitFullBuffer : Boolean read FWaitFullBuffer
                                      write FWaitFullBuffer;
   Published
    {Событие, вызываемое при получении байта}
    Property OnReadPacket : TReadPacketEvent read  FOnReadPacket
                                             write FOnReadPacket;
End;

implementation

uses
  Dialogs, WinSpool, Math;

{Добавление ресурса с иконкой для палитры компонент}
{$R *.dcr}

... ... ... ... ... ... ... ... ... ... ... ...

Constructor TComPort.Create(AOwner : TComponent);
Begin
 Inherited Create(AOwner);
 FComNumber:= 1;
 {создание объекта, сохраняющего свойства порта}
 FComProp  := TComProp.Create; {свойства порта}
 FHandle:= INVALID_HANDLE_VALUE;
 FReadThread:= nil;
 FInBufSize := 10;
 FWaitFullBuffer:= False;
End;
```

```
Destructor  TComPort.Destroy;
Begin
 {Закрываем порт и соединение}
 DoClosePort;
 FComProp.Free;
 Inherited Destroy;
End;

{Установка свойств из FComProp}
Procedure TComPort.ApplyComSettings;
Begin
 If not Connected then Exit;
 { Установить настройки порта согласно DCB }
 SetCommState(FHandle, FComProp.DCB);
End;

{Диалог настройки параметров порта}
Function TComPort.CommDialog : Boolean;
Var ComCfg : TCommConfig;
Begin
 ZeroMemory(@ComCfg, SizeOf(TCommConfig));
 ComCfg.dwSize:= SizeOf(TCommConfig);
 ComCfg.dcb   := FComProp.DCB;
 Result:= CommConfigDialog(PChar(GetComName(False)), 0, ComCfg);
 If Result then begin
  FComProp.DCB:= ComCfg.dcb;
  ApplyComSettings;
 End;
End;


{ Класс TComProp }

function TComProp.GetDCB: TDCB;
begin
 { Чистка структуры }
 ZeroMemory(@Result, SizeOf(TDCB));
 { Пеле DCBLength должно содержать размер структуры }
 Result.DCBLength:= SizeOf(TDCB);
```

```
 { Скорость обмена (бод) }
 Result.BaudRate := WindowsBaudRates[FBaudRate];
 { Windows не поддерживает небинарный режим работы }
 { последовательных портов }
 Result.Flags    := dcb_Binary;
 { Число бит в байте }
 Result.ByteSize := 5 + Ord(FByteSize);
 { Контроль четности }
 Result.Parity   := Ord(FParity);
 { Число стоп-бит }
 Result.StopBits := Ord(FStopbits);
end;

procedure TComProp.SetDCB(const Value: TDCB);
var i : TBaudRate;
begin
 {скорость обмена (бод)}
 FBaudRate:= br110;
 For i:= Low(WindowsBaudRates) to High(WindowsBaudRates) do begin
  If Value.BaudRate = WindowsBaudRates[i] then begin
   FBaudRate:= i;
   Break;
  End;
 End;
 {число бит в байте}
 FByteSize:= TByteSize(Value.ByteSize-5);
 {четность}
 FParity  := TParity(Value.Parity);
 {число стоп-бит}
 FStopbits:= TStopBits(Value.StopBits);
end;
end.
```

Для того чтобы свойства стали видны в Инспекторе объектов, нам потребовалось перенести их в секцию `Published`. Однако этого недостаточно. Инспектор объектов Delphi не знает, как редактировать свойство `ComProp`. Для добавления возможности редактирования нестандартного свойства следует добавить и зарегистрировать специальный класс-редактор. Код такого редактора показан в листинге 14.3.

Листинг 14.3. Редактор свойства ComProp для компонента TComPort

```
Unit ComPortEditor;

interface

{$IFDEF VER140}
uses Windows, Classes, Graphics, Forms, Controls, Buttons, DesignIntf,
  DesignWindows, StdCtrls, ComCtrls, DesignEditors, TypInfo, SysUtils;
{$ELSE}

{$IFDEF VER150}
uses Windows, Classes, Graphics, Forms, Controls, Buttons,
  StdCtrls, ComCtrls, TypInfo, SysUtils,
  DesignEditors, DesignIntf;
{$ELSE}
uses Windows, Classes, Graphics, Forms, Controls, Buttons,
  StdCtrls, ComCtrls, TypInfo, SysUtils, dsgnintf;
{$ENDIF}
{$ENDIF}

type
  TDCBPropertyEditor = class(TClassProperty)
  public
   procedure Edit; override;
   function  GetAttributes: TPropertyAttributes; override;
   function  GetValue: string; override;
   procedure SetValue(const Value: string); override;
  end;

procedure Register;

implementation

Uses Dialogs, ComPort;

{ Отображение редактора }
procedure TDCBPropertyEditor.Edit;
Var Comp     : TPersistent;
```

```
begin
 { Получить редактируемый компонент }
 Comp:= GetComponent(0);
 { Проверим тип компонента для исключения ошибок }
 If not (Comp is TComPort) then Exit;

 If TComPort(Comp).CommDialog then
  If Designer <> nil then {передаем уведомление об изменениях}
   Designer.Modified;     {дизайнеру форм}
end;

{ Параметры редактора свойств:}
{  paDialog   — для редактирования будет отображаться диалог}
{  paReadOnly — нельзя редактировать свойство напрямую}
function TDCBPropertyEditor.GetAttributes: TPropertyAttributes;
begin
 Result := [paDialog, paSubProperties];
end;

{Строка, отображаемая в Инспекторе объектов}
function TDCBPropertyEditor.GetValue: string;
begin
 Result := '(COM Properties)';
end;

{ Установка значения}
procedure TDCBPropertyEditor.SetValue(const Value: string);
begin
  {нельзя устанавливать значение}
end;

{ Регистрация редактора свойств }
procedure Register;
begin
 RegisterComponents('Comm', [TComPort]);
 RegisterPropertyEditor(TypeInfo(TComProp), TComPort,
                        'ComProp', TDCBPropertyEditor);
end;
end.
```

Немного сложная конструкция условной компиляции нужна для того, чтобы мы могли использовать наш компонент в любой версии Delphi (3—6). Для создания редактора нам потребовалось перекрыть всего 4 метода класса `TClassProperty`. Их работу легко понять из комментариев к коду или открыв справку по **Open Tools API**.

Теперь все готово. Зарегистрировать новый компонент можно либо выбрав в меню **Component** пункт **Install Component**, либо открыв файл пакета, приведенный в листинге 14.4. Отметим, что при создании нового пакета в Delphi 6 необходимо вручную добавить модуль `designide` в секцию `requires`.

Листинг 14.4. Пакет `ComPortPacket`, регистрирующий компонент `TComPort` и его редакторы свойств (Delphi 6)

```
package ComPortPacket;

{$R *.res}
{$ALIGN 8}
{$ASSERTIONS ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS ON}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO ON}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST OFF}
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
{$DESCRIPTION 'Communication Components'}
{$IMPLICITBUILD OFF}
```

```
requires
  rtl,
  designide, {добавить после создания пакета для Delphi 6!}
  vcl;

contains
  ComPort in 'ComPort.pas',
  ComPortEditor in 'ComPortEditor.pas';

end.
```

После установки значок `TComPort` должен появиться в палитре компонентов на вкладке **Comm**. Теперь нам остается заменить динамическое создание нашего класса на созданный компонент. Кроме того, как мы уже отмечали, мы удалим все лишние диалоговые окна настройки свойств порта, редактирование которых мы будем производить с помощью системной функции Windows.

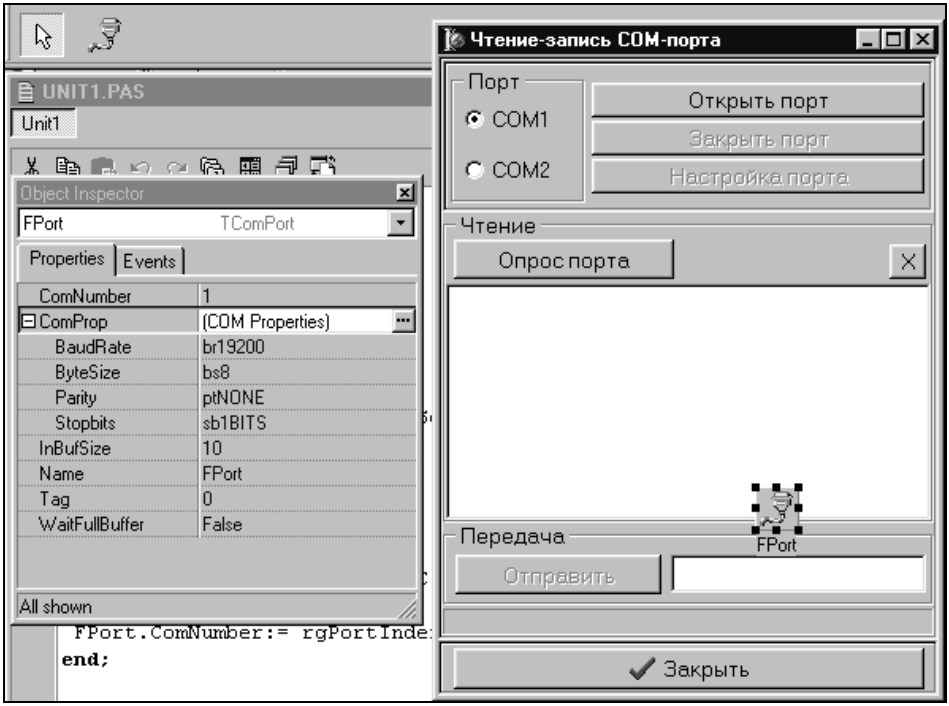

Рис. 14.1. Форма, использующая компонент `TComPort`

На рис. 14.1 показан новый вид формы и Инспектор объектов, отображающий свойства нашего компонента, а листинг 14.5 показывает исходный текст программы.

Листинг 14.5. Использование компонента TComPort

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls,Forms, Dialogs, ExtCtrls, StdCtrls, Buttons, Spin, Mask,
  ComPort, ComCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    btnClose: TBitBtn;
    ParamPanel: TPanel;
    rgPortIndex: TRadioGroup;
    GroupBox1: TGroupBox;
    bRead: TSpeedButton;
    btnClearReadBuf: TSpeedButton;
    GroupBox2: TGroupBox;
    btnSend: TSpeedButton;
    btnOpenPort: TSpeedButton;
    btnClosePort: TSpeedButton;
    StatusBar: TStatusBar;
    ESendStr: TEdit;
    lbBufRead: TListBox;
    FPort: TComPort;
    btnComProperty: TSpeedButton;
    procedure FormCreate(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure bReadClick(Sender: TObject);
    procedure btnSendClick(Sender: TObject);
```

```
    procedure btnClearReadBufClick(Sender: TObject);
    procedure btnOpenPortClick(Sender: TObject);
    procedure btnClosePortClick(Sender: TObject);
    procedure FPortReadPacket(const Packet: Pointer; const Size: Integer;
      const ErrCode: Cardinal);
    procedure btnComPropertyClick(Sender: TObject);
    procedure rgPortIndexClick(Sender: TObject);
  private
  public
   procedure SetEnabledControls;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

{ Создание компонента доступа к портам при создании формы }
procedure TForm1.FormCreate(Sender: TObject);
begin
 SetEnabledControls;
end;

{Изменение номера порта}
procedure TForm1.rgPortIndexClick(Sender: TObject);
begin
 FPort.ComNumber:= rgPortIndex.ItemIndex+1;
end;

{Кнопка открытия порта}
procedure TForm1.btnOpenPortClick(Sender: TObject);
begin
 FPort.Close;
 FPort.ComNumber:= rgPortIndex.ItemIndex+1;
 FPort.Open;
 If FPort.Connected then
```

```
  StatusBar.Panels[0].Text:= 'Порт успешно открыт'
 Else
  StatusBar.Panels[0].Text:= 'Ошибка открытия порта';
 SetEnabledControls;
end;

procedure TForm1.btnClosePortClick(Sender: TObject);
begin
 FPort.Close;
 StatusBar.Panels[0].Text:= '';
 SetEnabledControls;
 bRead.Down:= False;
end;

procedure TForm1.SetEnabledControls;
Var Active : Boolean;
begin
 Active:= FPort.Connected;
 btnClosePort.Enabled:= Active;
 rgPortIndex.Enabled := not Active;
 btnOpenPort.Enabled := not Active;
 btnComProperty.Enabled:= Active;
 btnSend.Enabled:= Active;
end;


{Кнопка закрытия программы}
procedure TForm1.btnCloseClick(Sender: TObject);
begin
 Close;
end;

{Кнопка опроса порта}
procedure TForm1.bReadClick(Sender: TObject);
begin
 FPort.ReadActive:= bRead.Down;
end;
```

```
procedure TForm1.FPortReadPacket(const Packet: Pointer;
  const Size: Integer; const ErrCode: Cardinal);
Var S : String; i : Integer;
begin
 S:= IntToStr(Size) + '>';

 For i:= 0 to Size-1 do begin
  S:= S + Char(Pointer(LongInt(Packet)+i)^);
 End;

 If ErrCode <> 0 then S:= '[!]' + S;
 lbBufRead.Items.Add(S);
end;

{Очистить буфер чтения}
procedure TForm1.btnClearReadBufClick(Sender: TObject);
begin
 lbBufRead.Items.Clear;
end;

{Отправка байта}
procedure TForm1.btnSendClick(Sender: TObject);
begin
 FPort.WriteString(ESendStr.Text);
end;

{Настройка порта}
procedure TForm1.btnComPropertyClick(Sender: TObject);
begin
 FPort.CommDialog;
end;



end.
```

Как всегда, работоспособность получившейся программы можно проверить, запустив две копии программы для двух портов (рис. 14.2).

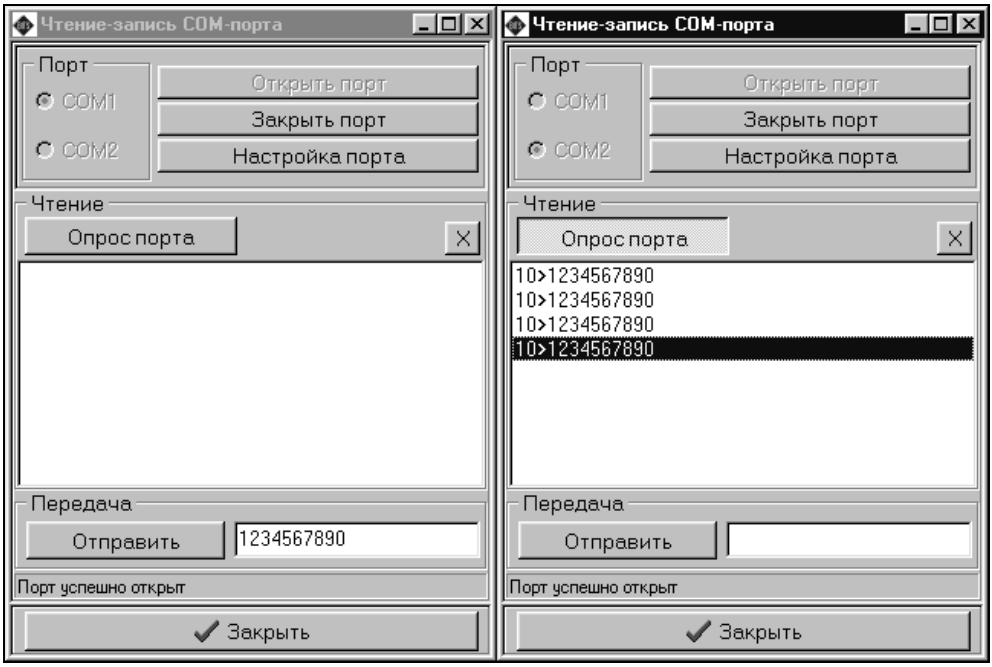

Рис. 14.2. Прием и передача строк

14.3. Реализация ASCII-протокола

Практически каждое устройство, подключаемое к коммуникационному порту, имеет свой уникальный протокол обмена. Изменять код классов `TComPort` и `TReadThread` для реализации различных протоколов обмена, конечно же, нерационально.

В этой главе мы создадим еще один компонент `TComAscii`, который будет "подключаться" к компоненту `TComPort` и, пользуясь его свойствами, принимать и передавать данные, согласно некоторому ASCII-протоколу (что такое ASCII-протокол, можно прочитать в *гл. 2*).

Основная идея `TComAscii` проста: программе не надо знать, какие реальные пакеты данных были получены, не надо знать и про пакеты, полученные с ошибками. Достаточно, если основная программа будет получать только проверенное содержимое этих пакетов. Более того, во многих случаях не нужно даже накапливать полученные пакеты — основной программе достаточно информации, содержащейся в последнем пакете (например, если передается текущее значение температуры или давления).

Исходный код компонента TComAscii приведен в листинге 14.6, а файл ресурса ComAscii.dcr, содержащий иконку для палитры компонентов, можно найти на прилагаемом компакт-диске.

Листинг 14.6. Компонент TComAscii

```
unit ComAscii;

interface

uses
  Windows, SysUtils, Classes, Forms, ComPort;

Type
 {Событие при получении новой строки}
 TNewStrRead = procedure(const NewStr : String) of object;

 {Реализация ASCII-протокола обмена}
 TComAscii = class(TComponent)
   FComPort : TComPort;
  private
    FStartStr: String;
    FEndStr  : String;
    FMaxLen  : Word;
    FOnNewStrRead : TNewStrRead;
    FInBuf   : String;
    FStart   : Boolean;
    procedure SetComPort(const Value: TComPort);
    procedure OnPortReadPacket(const Packet: Pointer;
         const Size: Integer; const ErrCode: Cardinal);
    function  CheckStartStr : Boolean;
    function  CheckEndStr(var S : String) : Boolean;
    procedure CheckMaxLen;
  Public
   Constructor Create(AOwner : TComponent); override;
   Destructor  Destroy; override;
   Procedure   Notification(AComponent: TComponent; Operation:
   TOperation); override;
  Published
```

```
   {Привязка к компоненту TComPort}
   Property ComPort : TComPort read FComPort write SetComPort;
   {Стартовая строка}
   Property StartStr: String read FStartStr write FStartStr;
   {Завершающая строка}
   Property EndStr  : String read FEndStr write FEndStr;
   {Максимальная длина принимаемой строки}
   Property MaxLen  : Word read FMaxLen write FMaxLen;
  Published
   {Событие при получении новой строки}
   Property OnNewStrRead : TNewStrRead read FOnNewStrRead write
FOnNewStrRead;
  Public
   {Передача строки согласно протоколу }
   Procedure SendString(const S : String);
 End;

Procedure Register;

implementation

{Добавление ресурса с иконкой для палитры компонентов}
{$R *.dcr}

Procedure Register;
Begin
 RegisterComponents('Comm', [TComAscii]);
End;

Constructor TComAscii.Create(AOwner : TComponent);
Begin
 Inherited Create(AOwner);
 FInBuf := '';
 FStart := False;
 FMaxLen:= 10;
End;

Destructor  TComAscii.Destroy;
Begin
```

```
 Inherited Destroy;
End;

{Вызывается при операциях в редакторе форм Delphi}
procedure TComAscii.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited;
  {Если компонент TComPort удалили с формы...}
  if (Operation = opRemove) and (FComPort <> nil) then begin
    if AComponent = FComPort then FComPort:= nil;
  end;
end;

{Передача строки согласно протоколу}
procedure TComAscii.SendString(const S: String);
begin
 If Assigned(FComPort) then
  If FComPort.Connected then
   FComPort.WriteString(FStartStr + S + FEndStr);
end;

{Привязка к компоненту TComPort}
procedure TComAscii.SetComPort(const Value: TComPort);
begin
 FComPort:= nil;
 If Value <> nil then begin
  FComPort:= Value;
  {Заставляем компонент TComPort}
  {пересылать уведомления редактора}
  {форм этому компоненту}
  FComPort.FreeNotification(Self);
  {Перехватываем обработчик получения новых пакетов}
  FComPort.OnReadPacket:= OnPortReadPacket;
 End;
end;

{Обработка получения новой строки}
procedure TComAscii.OnPortReadPacket(const Packet: Pointer;
        const Size: Integer; const ErrCode: Cardinal);
```

```
Var S : String; i : Integer;
begin
 {Получаем из буфера строку}
 S:= '';
 For i:= 0 to Size-1 do begin
  S:= S + Char(Pointer(LongInt(Packet)+i)^);
 End;
 {Сохраняем строку во входном буфере}
 FInBuf:= FInBuf + S;

 {Если начало посылки еще не получено...}
 If not FStart then begin
  {Проверяем на наличие символов начала посылки}
  If not CheckStartStr then begin
   CheckMaxLen;
   Exit;
  End;
  {начало посылки найдено и выкинуто...}
  FStart:= True;
 End;

 {Если начало посылки уже получено...}
 If FStart then begin
  {Проверяем наличие символов конца пакета}
  S:= '';
  If not CheckEndStr(S) then begin
   CheckMaxLen;
   Exit;
  End;
  {конец посылки найден и обработан...}
  If Assigned(FOnNewStrRead) then
   FOnNewStrRead(S);
  FStart:= False;
 End;
end;

{Проверка на присутствие символов начала посылки }
{Все символы до символов начала посылки удаляются}
{Если начало посылки найдено, возвращает True    }
```

```
function TComAscii.CheckStartStr : Boolean;
Var i : Integer;
begin
 Result:= False;
 i:= Pos(FStartStr, FInBuf);
 If i = 0 then Exit; {в строке нет начала посылки}
 {удаляем все символы вместе с началом посылки}
 Delete(FInBuf, 1, i + Length(FStartStr)-1);
 Result:= True;
end;

{Ограничение входного буфера}
procedure TComAscii.CheckMaxLen;
var i : Integer;
begin
 i:= Length(FInBuf);
 If i > FMaxLen*2 then
  Delete(FInBuf, 1, i - FMaxLen*2);
end;

{Проверка на присутствие символов конца посылки }
{Если конец посылки найден, возвращает True     }
function TComAscii.CheckEndStr(var S : String) : Boolean;
Var i : Integer;
begin
 Result:= False;
 i:= Pos(FEndStr, FInBuf);
 If i = 0 then Exit; {в строке нет конца посылки}
 {получение "чистой" строки}
 S:= Copy(FInBuf, 1, i-1);
 {удаляем все символы вместе с концом посылки}
 Delete(FInBuf, 1, i + Length(FStartStr));
 Result:= True;
end;

end.
```

После инсталляции в палитре компонентов должен появиться второй компонент.

Протокол передачи, который мы реализовали, заключается в добавлении к каждому передаваемому блоку данных префикса (задается свойством `StartStr`) и постфикса (задается свойством `EndStr`). Естественно, символы, содержащиеся в добавочных строках, не должны встречаться внутри самого сообщения.

Как видно из кода, компонент перехватывает обработку `OnReadPacket` и обрабатывает его самостоятельно. Для привязки компонента `TComAscii` к компоненту работы с портом необходимо в редакторе форм установить свойство `ComPort` в `FPort`, как показано на рис. 14.3. Основной программе достаточно обработать событие `OnNewStrRead`, вызывающееся при получении полного пакета данных, и метод `SendString` для передачи данных. Всю остальную работу берет на себя компонент.

Пример использования компонента `TComAscii` показан в листинге 14.7, а полный исходный код содержится на прилагаемом компакт-диске.

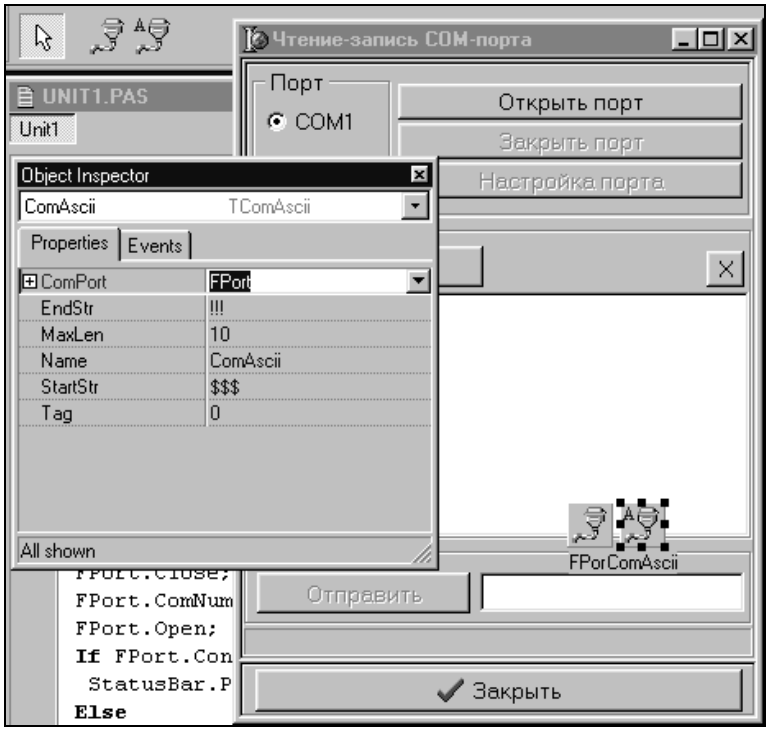

Рис. 14.3. Привязка компонента `TComAscii` к компоненту `TComPort`

Листинг 14.7. Использование компонента TComAscii

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
Forms, Dialogs, ExtCtrls, StdCtrls, Buttons, Spin, Mask,
ComPort, ComCtrls, ComAscii;

type
  TForm1 = class(TForm)
    ... ... ... ... ... ... ... ... ... ... ... ...
    FPort: TComPort;
    ComAscii: TComAscii;
    ... ... ... ... ... ... ... ... ... ... ... ...
    procedure ComAsciiNewStrRead(const NewStr: String);
    ... ... ... ... ... ... ... ... ... ... ... ...
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

{инициализация состояния кнопок}
procedure TForm1.FormCreate(Sender: TObject);
begin
 SetEnabledControls;
end;

... ... ... ... ... ... ... ... ... ... ... ...

{Отправка байта}
procedure TForm1.btnSendClick(Sender: TObject);
```

```
begin
 ComAscii.SendString(ESendStr.Text);
end;

{Получение новой строки}
procedure TForm1.ComAsciiNewStrRead(const NewStr: String);
begin
 lbBufRead.Items.Add(NewStr);
end;

... ... ... ... ... ... ... ... ... ... ... ...

end.
```

Теперь мы можем создать несколько компонентов для работы с разными протоколами. Подключая эти компоненты к `TComPort`, можно использовать разные протоколы, изменяя их одним щелчком мыши! Такая система напоминает связку `DataSource-DataSet` для баз данных.

Реализация передачи других данных также существенно упрощается. Например, листинг 14.8 показывает передачу числа с плавающей точкой.

Листинг 14.8. Передача числа с плавающей точкой по протоколу ASCII

```
{Передача числа с плавающей точкой}
{PointPos – заданная точность     }
Procedure TComAscii.SendSingle(const S : Single; const PointPos :
Integer);
Begin
 SendString(Format('%.*f', [PointPos, S]));
End;
```

14.4. Реализация бинарного протокола

Как мы уже отмечали, строковый протокол обмена имеет несколько недостатков. Самый существенный из них — большой объем данных, необходимых для передачи чисел, особенно чисел с плавающей точкой. Действительно, в примере из предыдущего раздела для передачи числа с двумя знаками до точки и четырьмя после точки потребуется (3 + 2 + 1 + 4 + 3) = 13 знаков. Таким образом, для передачи 4 байт информации требуется 13 байт. А при добавлении контрольной суммы потребуется еще 3 дополнительных байта.

В этой главе мы реализуем простейший бинарный протокол обмена, состоящий из 7 байт:

- 0-й байт — стартовый байт;
- 1—4-й байты — число с плавающей точкой;
- 5-й байт — завершающий байт;
- 6-й байт — контрольная сумма.

Легко посчитать, что для передачи SINGLE-числа в бинарном виде потребуется всего (1 + 4 + 1) = 6 байт, или 7 — при передаче с контрольной суммой. Таким образом, мы выигрываем в объеме данных почти в два раза (причем независимо от числа знаков после точки)!

Так же как и в случае ASCII-протокола, мы создадим специальный компонент `TComBin`, который будет привязываться к транспортному компоненту `TComPort`. Исходный код этого компонента показан в листинге 14.9, а файл и ресурс с иконкой для палитры компонентов можно найти на прилагаемом компакт-диске.

Листинг 14.9. Компонент бинарного протокола обмена TComBin

```
unit ComBin;

interface

uses
  Windows, SysUtils, Classes, Forms, ComPort;

Type
 {Событие при получении нового значения}
 TNewValueRead = procedure(const Value : Single) of object;

 SingleArray = Array [1..4] of Byte;

 {Реализация простого бинарного протокола обмена}
 TComBin = class(TComponent)
  private
    FComPort : TComPort; {транспортный компонент}
    FStartByte : Byte;   {байт начала посылки   }
    FEndByte   : Byte;   {байт конца посылки    }
    FOnNewValueRead : TNewValueRead; {событие}
    FByteIndex : Integer; {счетчик байт посылки}
```

```
    FInBuffer  : SingleArray; {буфер для числа}
    FUseCS     : Boolean; {проверка контрольной суммы}
    procedure SetComPort(const Value: TComPort);
    procedure OnPortReadPacket(const Packet: Pointer;
         const Size: Integer; const ErrCode: Cardinal);
    function  GetPacketSize : Integer;
   {Вычисление контрольной суммы для числа Single}
    function  GetCSSingle(AValue : Single) : Byte;
   {Вычисление контрольной суммы для SingleArray}
    function  GetCSBuf(Buf : SingleArray) : Byte;
  Public
   Constructor Create(AOwner : TComponent); override;
   Destructor  Destroy; override;
   Procedure   Notification(AComponent: TComponent;
                        Operation: TOperation); override;
  Published
   {Привязка к компоненту TComPort}
   Property ComPort : TComPort read FComPort write SetComPort;
   {Стартовый байт}
   Property StartByte: Byte read FStartByte write FStartByte;
   {Завершающий байт}
   Property EndByte  : Byte read FEndByte write FEndByte;
   {Проверка контрольной суммы}
   Property UseCS : Boolean read FUseCS write FUseCS;
  Published
   {Событие при получении новой строки}
   Property OnNewValueRead : TNewValueRead read FOnNewValueRead
                                        write FOnNewValueRead;
  Public
   {Передача числа согласно протоколу }
   Procedure SendValue(const AValue : Single);
 End;

Procedure Register;

implementation

{Добавление ресурса с иконкой для палитры компонентов}
{$R *.dcr}
```

```
Procedure Register;
Begin
 RegisterComponents('Comm', [TComBin]);
End;

Constructor TComBin.Create(AOwner : TComponent);
Begin
 Inherited Create(AOwner);
 FByteIndex:= 0;
End;

Destructor  TComBin.Destroy;
Begin
 Inherited Destroy;
End;

{Вызывается при операциях в редакторе форм Delphi}
procedure TComBin.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited;
  {Если компонент TComPort удалили с формы...}
  if (Operation = opRemove) and (FComPort <> nil) then begin
    if AComponent = FComPort then FComPort:= nil;
  end;
end;

{Вычисление контрольной суммы для числа Single}
function  TComBin.GetCSSingle(AValue : Single) : Byte;
Var Buf : SingleArray;
begin
 Move(AValue, Buf, SizeOf(Buf));
 Result:= GetCSBuf(Buf);
end;

{Вычисление контрольной суммы для SingleArray}
function  TComBin.GetCSBuf(Buf : SingleArray) : Byte;
var i : Integer;
```

```
begin
 Result:= 0;
 {$R-}
 For i:= Low(Buf) to High(Buf) do begin
  Result:= Result + Buf[i];
 End;
 {$R+}
end;

{Передача числа согласно протоколу}
procedure TComBin.SendValue(const AValue : Single);
Var CS : Byte;
begin
 If Assigned(FComPort) then
  If FComPort.Connected then begin
   {Послать символ начала посылки}
   FComPort.WriteBuffer(@FStartByte, SizeOf(FStartByte));
   {Послать число}
   FComPort.WriteBuffer(@AValue, SizeOf(AValue    ));
   {Послать символ завершения посылки}
   FComPort.WriteBuffer(@FEndByte, SizeOf(FEndByte  ));
   {контрольная сумма, если включено}
   If FUseCS then begin
    CS:= GetCSSingle(AValue);
    FComPort.WriteBuffer(@CS, SizeOf(CS));
   End;
  End;
end;

{Привязка к компоненту TComPort}
procedure TComBin.SetComPort(const Value: TComPort);
begin
 FComPort:= nil;
 If Value <> nil then begin
  FComPort:= Value;
  {Заставляем компонент TComPort}
  {пересылать уведомления редактора}
  {форм этому компоненту}
  FComPort.FreeNotification(Self);
```

```
  {Перехватываем обработчик получения новых пакетов}
  FComPort.OnReadPacket:= OnPortReadPacket;
 End;
end;

{Вычисление размера передаваемого пакета}
function TComBin.GetPacketSize : Integer;
begin
 Result:= SizeOf(FStartByte) + SizeOf(Single) + SizeOf(FEndByte);
 If FUseCS then {+контрольная сумма}
  Inc(Result);
end;

{Обработка получения нового значения}
procedure TComBin.OnPortReadPacket(const Packet: Pointer;
        const Size: Integer; const ErrCode: Cardinal);
Var B : Byte; i : Integer; bNextByte : Boolean; AValue : Single;
begin
 For i:= 0 to Size-1 do begin
  B:= Byte(Pointer(LongInt(Packet)+i)^);
  bNextByte:= False;
  Case FByteIndex of
   0: begin     // ожидается символ начала посылки
     bNextByte:= (B = FStartByte);
   End;{0}
   1..4: begin // получение числа
    FInBuffer[FByteIndex]:= B;
    bNextByte:= True;
   End; {1..4}
   5: begin   // ожидается символ конца посылки
     bNextByte:= (B = FEndByte);
   End; {5}
   6: begin
     bNextByte:= (B= GetCSBuf(FInBuffer));
   End; {6}
  End;

  If bNextByte then begin
   {переходим к ожиданию следующего}
   {байта посылки}
```

```
   Inc(FByteIndex);
  End else begin
   {неверный формат посылки}
   {начинаем ожидание сначала}
   FByteIndex:= 0;
  End;

  {посылка закончена}
  If FByteIndex = GetPacketSize then begin
   Try
    Try
     {Переписываем буфер в число Single}
     Move(FInBuffer, AValue, SizeOf(AValue));
    Except End;
    {Вызываем обработчик}
    If Assigned(FOnNewValueRead) then
     FOnNewValueRead(AValue);
   Finally
    {Независимо от ошибок начинаем отсчет снова}
    FByteIndex:= 0;
   End;
  End;
 End;

end;

end.
```

Основная работа производится в обработчике `OnPortReadPacket`. Рассмотрим его более подробно.

При запуске программы счетчик `FByteIndex` равен 0, и, следовательно, компонент будет ожидать получения стартового байта. Счетчик байтов увеличится только в случае совпадения очередного байта со значением стартового байта. Затем, пока счетчик будет равен значению от 1 до 4, мы будем просто сохранять байты в буфер. Значение счетчика 5 будет означать, что компонент ожидает получения завершающего байта. Если хотя бы один байт не будет совпадать с протоколом обмена, переменная `bNextByte` примет значение `False`, и счетчик байтов сбросится в ноль. Завершение обработки пакета

произойдет в случае достижения счетчиком величины длины пакета. В этом случае мы скопируем полученные байты из временного буфера в переменную типа SINGLE и вызовем обработчик события OnNewValueRead.

Отдельно стоит обратить внимание на свойство компонента UseCS. Если UseCS установлено в True, обмен будет производиться с добавлением контрольной суммы. Контрольная сумма вычисляется в функции GetCSSingle и представляет собой байтовую сумму с переполнением всех байтов числа. Естественно, при добавлении контрольной суммы мы увеличиваем значение длины протокола, возвращаемое функцией GetPacketSize, на 1.

Теперь нужно либо добавить этот компонент в пакет ComPortPacket.dpk, либо создать другой пакет, как показано в листинге 14.10. После инсталляции пакета в палитре компонентов появится значок компонента TComBin.

Листинг 14.10. Пакет для установки компонента TComBin

```
package ComPortPacket;

{$R *.res}
{$ALIGN 8}
{$ASSERTIONS ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS ON}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO ON}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST OFF}
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
```

```
{$DESCRIPTION 'Communication Components'}
{$IMPLICITBUILD OFF}

requires
  rtl,
  designide,
  vcl;

contains
  ComPort in 'ComPort.pas',
  ComPortEditor in 'ComPortEditor.pas',
  ComBin in 'ComBin.pas';
end.
```

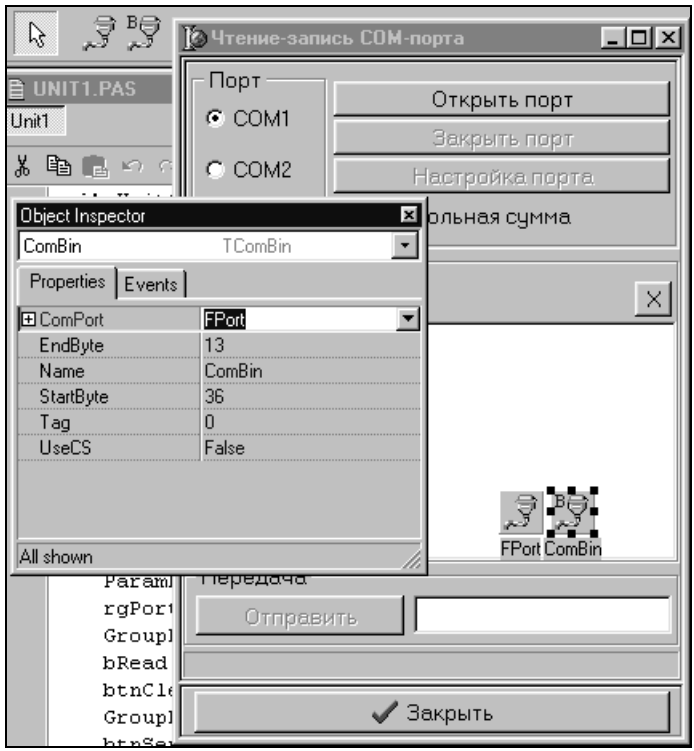

Рис. 14.4. Форма с компонентом TComBin

Основная форма потребует мелких изменений. Во-первых, компонент TComAscii нужно заменить на TComBin. Во-вторых, нужно создать новый

обработчик события, а метод передачи строки заменить передачей числа. Кроме того, мы добавили переключатель использования контрольной суммы при обмене. Измененная форма показана на рис. 14.4, а изменения в коде приведены в листинге 14.11. Полный исходный код можно найти на прилагаемом компакт-диске.

Листинг 14.11. Изменения в основной форме для использования TComBin

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
Forms, Dialogs, ExtCtrls, StdCtrls, Buttons, Spin, Mask,
ComPort, ComCtrls, ComBin;

type
  TForm1 = class(TForm)
    ... ... ... ... ... ... ... ... ... ... ... ...
    FPort: TComPort;
    ComBin: TComBin;
    cbUseCS: TCheckBox;
    ... ... ... ... ... ... ... ... ... ... ... ...
    procedure cbUseCSClick(Sender: TObject);
    procedure ComBinNewValueRead(const Value: Single);
    ... ... ... ... ... ... ... ... ... ... ... ...
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

... ... ... ... ... ... ... ... ... ... ... ...

{Передача числа с плавающей точкой}
procedure TForm1.btnSendClick(Sender: TObject);
```

```
Var V : Single;
begin
 Try
  V:= StrToFloat(ESendValue.Text);
 Except
  MessageDlg('Неверный формат числа', mtError, [mbOK], 0);
  Exit;
 End;
 ComBin.SendValue(V);
end;

{Получение нового числа}
procedure TForm1.ComBinNewStrRead(const Value: Single);
begin
 lbBufRead.Items.Add(FloatToStr(Value));
end;

{Включить использование контрольной суммы}
procedure TForm1.cbUseCSClick(Sender: TObject);
begin
 ComBin.UseCS:= cbUseCS.Checked;
end;

end.
```

Открыв одну копию программы для передачи по COM1, а вторую — для приема по COM2, можно поэкспериментировать с контрольной суммой. Естественно, что передача с контрольной суммой будет правильно приниматься даже при выключенной опции при приеме. Действительно, дополнительный байт будет считаться ошибочной посылкой и игнорироваться. А вот установить опции наоборот — не получится: программа приема будет ожидать контрольной суммы, и если она не будет передана, то ни один пакет не будет получен.

Подводя итог, подчеркнем главный результат — основная программа получает только правильные, освобожденные от "неинформационных" байтов, пакеты.

Глава 15

Вычисление контрольных сумм

Проверка — высшая форма доверия.

В *гл. 2* мы уже обсуждали вопросы, связанные с применением контрольных сумм для повышения достоверности передаваемой информации. В этой главе мы рассмотрим различные способы вычисления контрольной суммы, применяемые в наиболее распространенных протоколах обмена.

15.1. Вычисление простой контрольной суммы

Самая простая контрольная сумма может быть вычислена при помощи суммирования с переполнением всех информационных байтов. Язык C не контролирует выход за пределы типа, поэтому реализация вычисления такой суммы сводится к обыкновенному суммированию всех байтов данных.

Реализация подсчета контрольной суммы в Pascal и Delphi показана в листинге 15.1. Естественно, в начале вычисления переменная `CS` должна быть инициализирована нулем.

Листинг 15.1. Вычисление простейшей контрольной суммы

```
byte { контрольная сумма }
 CS : Byte;


{В Pascal для вычисления контрольной суммы }
{нужно отключать контроль переполнения     }
procedure AddCS(b : Byte);
begin
  {$R-}
  CS:= CS + b;
```

```
  {$R+}
end;

{В Delphi тоже можно отключать переполнение,}
{а можно использовать механизм исключений   }
procedure AddCS(b : Byte);
begin
 Try
  CS:= CS + b;
 Except End;
end;
```

С помощью дополнительной проверки можно обойтись без обработки исключений (листинг 15.2). Такая конструкция удобна при отладке программы, т. к. даже при включенной опции **Останавливать при появлении исключений** (Stop On Delphi Exception) программа не будет останавливаться на каждом байте, добавляемом к контрольной сумме.

Листинг 15.2. Вычисление простой контрольной суммы без вызова исключения

```
{А с помощью if можно обойтись без обработки}
{исключений                                  }
procedure AddCS(b : byte);
begin
 if (CS + b) > 256 then
  CS:= CS + b - 256
 else
  CS:= CS + b;
end;
```

15.2. Вычисление LCR

Собственно сам алгоритм вычисления LCR уже описан в *гл. 2*, в *разд. "Контрольная сумма LCR"*, и нам остается только строго следовать его шагам. В листинге 15.3 приведен код вычисления LCR для буфера, задаваемого указателем P.

Листинг 15.3. Вычисление контрольной суммы LCR

```
Function CalculateLRC(P : PChar; Len : Word) : Byte;
Var i : Word;
```

```
Begin
 {$R-}
 Result:= 0;
 For i:= 0 to Len-1 do
   Result:= Result + Byte(Pointer(LongInt(P)+i)^);
 Result:= $FF - Result;
 Inc(Result);
 {$R+}
End;
```

Следует отметить, что шаги 2 и 3 алгоритма (вычитание $FF и прибавление 1) сводятся к операции `Result:= -Result`, поэтому вычисление можно сократить (листинг 15.4).

Листинг 15.4. Вычисление контрольной суммы LCR

```
Function CalculateLRC(P : PChar; Len : Word) : Byte;
Var i : Word;
Begin
 {$R-}
 Result:= 0;
 For i:= 0 to Len-1 do
   Result:= Result + Byte(Pointer(LongInt(P)+i)^);
 Result:= -Result;
 {$R+}
End;
```

15.3. Вычисление CRC16

Алгоритм вычисления CRC16 также уже был описан в *гл. 2*, в *разд. "Контрольная сумма LCR"*. Реализация этого алгоритма приведена в листинге 15.5, где показан код вычисления CRC16 для буфера, задаваемого указателем `P`.

Листинг 15.5. Вычисление контрольной суммы CRC16

```
Function CalculateCRC16(P : PChar; Len : Word) : Word;
Var iByte, i : Word; B : Byte;
Begin
{$R-}
 {инициализация}
```

```
 Result:= $FFFF;
 {цикл по всем байтам буфера}
 For iByte:= 0 to Len-1 do begin
  {очередной байт буфера}
  B:= Byte(Pointer(LongInt(P)+iByte)^);
  {вычисление очередного CRC}
  Result:= (Result and $FF00) + (B xor Lo(Result));
  For i:= 1 to 8 do begin
   If ((Result and $0001) <> 0) then
    Result:= (Result shr 1) xor $A001
   Else
    Result:= (Result shr 1);
  End;
 End;
 {$R+}
End;
```

Часто для увеличения скорости расчета создают таблицу всех возможных комбинаций CRC для старшего и младшего регистров, обеспечивая быстрое вычисление новой величины CRC для каждого байта буфера. Пример такого кода показан в листинге 15.6. Именно такой способ вычисления применяется в протоколе MODBUS.

Листинг 15.6. Быстрое вычисление контрольной суммы CRC16

```
Const
 CRCLoTable : Array [0..255] of Byte = (
  $00,$C1,$81,$40,$01,$C0,$80,$41,$01,$C0,$80,$41,$00,
  $C1,$81,$40,$01,$C0,$80,$41,$00,$C1,$81,$40,$00,$C1,
  $81,$40,$01,$C0,$80,$41,$01,$C0,$80,$41,$00,$C1,$81,
  $40,$00,$C1,$81,$40,$01,$C0,$80,$41,$00,$C1,$81,$40,
  $01,$C0,$80,$41,$01,$C0,$80,$41,$00,$C1,$81,$40,$01,
  $C0,$80,$41,$00,$C1,$81,$40,$00,$C1,$81,$40,$01,$C0,
  $80,$41,$00,$C1,$81,$40,$01,$C0,$80,$41,$01,$C0,$80,
  $22,$00,$C1,$81,$40,$00,$C1,$81,$40,$01,$C0,$80,$41,
  $01,$C0,$80,$41,$00,$C1,$81,$40,$01,$C0,$80,$41,$00,
  $C1,$81,$40,$00,$C1,$81,$40,$01,$C0,$80,$41,$01,$C0,
  $80,$41,$00,$C1,$81,$40,$00,$C1,$81,$40,$01,$C0,$80,
  $41,$00,$C1,$81,$40,$01,$C0,$80,$41,$01,$C0,$80,$41,
```

```
  $00,$C1,$81,$40,$00,$C1,$81,$40,$01,$C0,$80,$41,$01,
  $C0,$80,$41,$00,$C1,$81,$40,$01,$C0,$80,$41,$00,$C1,
  $81,$40,$00,$C1,$81,$40,$01,$C0,$80,$41,$00,$C1,$81,
  $40,$01,$C0,$80,$41,$00,$C0,$80,$41,$00,$C1,$81,$40,
  $01,$C0,$80,$41,$00,$C1,$81,$40,$00,$C1,$81,$40,$01,
  $C0,$80,$41,$01,$C0,$80,$41,$00,$C1,$81,$40,$00,$C1,
  $81,$40,$01,$C0,$80,$41,$00,$C1,$81,$40,$01,$C0,$80,
  $41,$01,$C0,$80,$41,$00,$C1,$81,$40
 );

 CRCHiTable : Array [0..255] of Byte = (
  $00,$C0,$C1,$01,$C3,$03,$02,$C2,$C6,$06,$07,$C7,$05,
  $C5,$C4,$04,$CC,$0C,$0D,$CD,$0F,$CF,$CE,$0E,$0A,$CA,
  $CB,$0B,$C9,$09,$08,$C8,$D8,$18,$19,$D9,$1B,$DB,$DA,
  $1A,$1E,$DE,$DF,$1F,$DD,$1D,$1C,$DC,$14,$D4,$D5,$15,
  $D7,$17,$16,$D6,$D2,$12,$13,$D3,$11,$D1,$D0,$10,$F0,
  $30,$31,$F1,$33,$F3,$F2,$32,$36,$F6,$F7,$37,$F5,$35,
  $34,$F4,$3C,$FC,$FD,$3D,$FF,$3F,$3E,$FE,$FA,$3A,$3B,
  $FB,$39,$F9,$F8,$38,$28,$E8,$E9,$29,$EB,$2B,$2A,$EA,
  $EE,$2E,$2F,$EF,$2D,$ED,$EC,$2C,$E4,$24,$25,$E5,$27,
  $E7,$E6,$26,$22,$E2,$E3,$23,$E1,$21,$20,$E0,$A0,$60,
  $61,$A1,$63,$A3,$A2,$62,$66,$A6,$A7,$67,$A5,$65,$64,
  $A4,$6C,$AC,$AD,$6D,$AF,$6F,$6E,$AE,$AA,$6A,$6B,$AB,
  $69,$A9,$A8,$68,$78,$B8,$B9,$79,$BB,$7B,$7A,$BA,$BE,
  $7E,$7F,$BF,$7D,$BD,$BC,$7C,$B4,$74,$75,$B5,$77,$B7,
  $B6,$76,$72,$B2,$B3,$73,$B1,$71,$70,$B0,$50,$90,$91,
  $51,$93,$53,$52,$92,$96,$56,$57,$97,$55,$95,$94,$54,
  $9C,$5C,$5D,$9D,$5F,$9F,$9E,$5E,$5A,$9A,$9B,$5B,$99,
  $59,$58,$98,$88,$48,$49,$89,$4B,$8B,$8A,$4A,$4E,$8E,
  $8F,$4F,$8D,$4D,$4C,$8C,$44,$84,$85,$45,$87,$47,$46,
  $86,$82,$42,$43,$83,$41,$81,$80,$40
 );

Function CalculateQuickCRC(P : PChar; Len : Word) : Word;
Var iByte : Word; CRCHi, CRCLo, Index : Byte; B : Byte;
Begin
 {$R-}
 CRCHi:= $FF;
 CRCLo:= $FF;
```

```
 For iByte:= 0 to Len-1 do begin
  Index:= CRCLo xor Byte(Pointer(LongInt(P)+iByte)^);
  CRCLo:= CRCHi xor CRCLoTable[Index];
  CRCHi:= CRCHiTable[Index];
 End;
 Result:= (CRCHi shl 8) or CRCLo;
 {$R+}
End;
```

Еще один способ вычисления CRC16 заключается в создании единой таблицы для старшего и младшего байта. В этом случае расчет CRC сводится к выполнению одного цикла (листинг 15.7). Такой алгоритм применяется, например, в протоколе XModem.

Листинг 15.7. Вычисление контрольной суммы CRC16 с помощью таблицы CRC

```
Const
  CrcTable: array[0..255] of Word = (
    $0000, $1021, $2042, $3063, $4084, $50a5, $60c6, $70e7,
    $8108, $9129, $a14a, $b16b, $c18c, $d1ad, $e1ce, $f1ef,
    $1231, $0210, $3273, $2252, $52b5, $4294, $72f7, $62d6,
    $9339, $8318, $b37b, $a35a, $d3bd, $c39c, $f3ff, $e3de,
    $2462, $3443, $0420, $1401, $64e6, $74c7, $44a4, $5485,
    $a56a, $b54b, $8528, $9509, $e5ee, $f5cf, $c5ac, $d58d,
    $3653, $2672, $1611, $0630, $76d7, $66f6, $5695, $46b4,
    $b75b, $a77a, $9719, $8738, $f7df, $e7fe, $d79d, $c7bc,
    $48c4, $58e5, $6886, $78a7, $0840, $1861, $2802, $3823,
    $c9cc, $d9ed, $e98e, $f9af, $8948, $9969, $a90a, $b92b,
    $5af5, $4ad4, $7ab7, $6a96, $1a71, $0a50, $3a33, $2a12,
    $dbfd, $cbdc, $fbbf, $eb9e, $9b79, $8b58, $bb3b, $ab1a,
    $6ca6, $7c87, $4ce4, $5cc5, $2c22, $3c03, $0c60, $1c41,
    $edae, $fd8f, $cdec, $ddcd, $ad2a, $bd0b, $8d68, $9d49,
    $7e97, $6eb6, $5ed5, $4ef4, $3e13, $2e32, $1e51, $0e70,
    $ff9f, $efbe, $dfdd, $cffc, $bf1b, $af3a, $9f59, $8f78,
    $9188, $81a9, $b1ca, $a1eb, $d10c, $c12d, $f14e, $e16f,
    $1080, $00a1, $30c2, $20e3, $5004, $4025, $7046, $6067,
    $83b9, $9398, $a3fb, $b3da, $c33d, $d31c, $e37f, $f35e,
    $02b1, $1290, $22f3, $32d2, $4235, $5214, $6277, $7256,
    $b5ea, $a5cb, $95a8, $8589, $f56e, $e54f, $d52c, $c50d,
```

```
    $34e2, $24c3, $14a0, $0481, $7466, $6447, $5424, $4405,
    $a7db, $b7fa, $8799, $97b8, $e75f, $f77e, $c71d, $d73c,
    $26d3, $36f2, $0691, $16b0, $6657, $7676, $4615, $5634,
    $d94c, $c96d, $f90e, $e92f, $99c8, $89e9, $b98a, $a9ab,
    $5844, $4865, $7806, $6827, $18c0, $08e1, $3882, $28a3,
    $cb7d, $db5c, $eb3f, $fb1e, $8bf9, $9bd8, $abbb, $bb9a,
    $4a75, $5a54, $6a37, $7a16, $0af1, $1ad0, $2ab3, $3a92,
    $fd2e, $ed0f, $dd6c, $cd4d, $bdaa, $ad8b, $9de8, $8dc9,
    $7c26, $6c07, $5c64, $4c45, $3ca2, $2c83, $1ce0, $0cc1,
    $ef1f, $ff3e, $cf5d, $df7c, $af9b, $bfba, $8fd9, $9ff8,
    $6e17, $7e36, $4e55, $5e74, $2e93, $3eb2, $0ed1, $1ef0
  );

Function CalculateQuickCRC1(P : PChar; Len : Word) : Word;
Var iByte : Word;
Begin
 {$R-}
 Result:= 0;
 For iByte:= 0 to Len-1 do begin
  Result:= CrcTable[Result shr 8] xor (Result shl 8)
           xor Byte(Pointer(LongInt(P)+iByte)^);
 End;
 {$R+}
End;
```

15.4. Вычисление CRC32

Для увеличения достоверности данных можно увеличить число контрольных битов. В таких случаях применяют 4-байтовую контрольную сумму, называемую CRC32.

Алгоритм вычисления CRC32 практически не отличается от алгоритма вычисления CRC16, за исключением того, что все операции производятся над словами (т. е. по 2 байта). Пример реализации вычисления CRC32 показан в листинге 15.8.

Листинг 15.8. Вычисление контрольной суммы CRC32

```
Const
 Crc32Table : Array[0..255] of Cardinal = (
 $00000000, $77073096, $ee0e612c, $990951ba,
```

```
$076dc419, $706af48f, $e963a535, $9e6495a3,
$0edb8832, $79dcb8a4, $e0d5e91e, $97d2d988,
$09b64c2b, $7eb17cbd, $e7b82d07, $90bf1d91,
$1db71064, $6ab020f2, $f3b97148, $84be41de,
$1adad47d, $6ddde4eb, $f4d4b551, $83d385c7,
$136c9856, $646ba8c0, $fd62f97a, $8a65c9ec,
$14015c4f, $63066cd9, $fa0f3d63, $8d080df5,
$3b6e20c8, $4c69105e, $d56041e4, $a2677172,
$3c03e4d1, $4b04d447, $d20d85fd, $a50ab56b,
$35b5a8fa, $42b2986c, $dbbbc9d6, $acbcf940,
$32d86ce3, $45df5c75, $dcd60dcf, $abd13d59,
$26d930ac, $51de003a, $c8d75180, $bfd06116,
$21b4f4b5, $56b3c423, $cfba9599, $b8bda50f,
$2802b89e, $5f058808, $c60cd9b2, $b10be924,
$2f6f7c87, $58684c11, $c1611dab, $b6662d3d,
$76dc4190, $01db7106, $98d220bc, $efd5102a,
$71b18589, $06b6b51f, $9fbfe4a5, $e8b8d433,
$7807c9a2, $0f00f934, $9609a88e, $e10e9818,
$7f6a0dbb, $086d3d2d, $91646c97, $e6635c01,
$6b6b51f4, $1c6c6162, $856530d8, $f262004e,
$6c0695ed, $1b01a57b, $8208f4c1, $f50fc457,
$65b0d9c6, $12b7e950, $8bbeb8ea, $fcb9887c,
$62dd1ddf, $15da2d49, $8cd37cf3, $fbd44c65,
$4db26158, $3ab551ce, $a3bc0074, $d4bb30e2,
$4adfa541, $3dd895d7, $a4d1c46d, $d3d6f4fb,
$4369e96a, $346ed9fc, $ad678846, $da60b8d0,
$44042d73, $33031de5, $aa0a4c5f, $dd0d7cc9,
$5005713c, $270241aa, $be0b1010, $c90c2086,
$5768b525, $206f85b3, $b966d409, $ce61e49f,
$5edef90e, $29d9c998, $b0d09822, $c7d7a8b4,
$59b33d17, $2eb40d81, $b7bd5c3b, $c0ba6cad,
$edb88320, $9abfb3b6, $03b6e20c, $74b1d29a,
$ead54739, $9dd277af, $04db2615, $73dc1683,
$e3630b12, $94643b84, $0d6d6a3e, $7a6a5aa8,
$e40ecf0b, $9309ff9d, $0a00ae27, $7d079eb1,
$f00f9344, $8708a3d2, $1e01f268, $6906c2fe,
$f762575d, $806567cb, $196c3671, $6e6b06e7,
$fed41b76, $89d32be0, $10da7a5a, $67dd4acc,
$f9b9df6f, $8ebeeff9, $17b7be43, $60b08ed5,
```

```
 $d6d6a3e8, $a1d1937e, $38d8c2c4, $4fdff252,
 $d1bb67f1, $a6bc5767, $3fb506dd, $48b2364b,
 $d80d2bda, $af0a1b4c, $36034af6, $41047a60,
 $df60efc3, $a867df55, $316e8eef, $4669be79,
 $cb61b38c, $bc66831a, $256fd2a0, $5268e236,
 $cc0c7795, $bb0b4703, $220216b9, $5505262f,
 $c5ba3bbe, $b2bd0b28, $2bb45a92, $5cb36a04,
 $c2d7ffa7, $b5d0cf31, $2cd99e8b, $5bdeae1d,
 $9b64c2b0, $ec63f226, $756aa39c, $026d930a,
 $9c0906a9, $eb0e363f, $72076785, $05005713,
 $95bf4a82, $e2b87a14, $7bb12bae, $0cb61b38,
 $92d28e9b, $e5d5be0d, $7cdcefb7, $0bdbdf21,
 $86d3d2d4, $f1d4e242, $68ddb3f8, $1fda836e,
 $81be16cd, $f6b9265b, $6fb077e1, $18b74777,
 $88085ae6, $ff0f6a70, $66063bca, $11010b5c,
 $8f659eff, $f862ae69, $616bffd3, $166ccf45,
 $a00ae278, $d70dd2ee, $4e048354, $3903b3c2,
 $a7672661, $d06016f7, $4969474d, $3e6e77db,
 $aed16a4a, $d9d65adc, $40df0b66, $37d83bf0,
 $a9bcae53, $debb9ec5, $47b2cf7f, $30b5ffe9,
 $bdbdf21c, $cabac28a, $53b39330, $24b4a3a6,
 $bad03605, $cdd70693, $54de5729, $23d967bf,
 $b3667a2e, $c4614ab8, $5d681b02, $2a6f2b94,
 $b40bbe37, $c30c8ea1, $5a05df1b, $2d02ef8d
);

Function CalculateCRC32(P : PChar; Len : Word) : Cardinal;
Var iByte : Word; B : Byte;
Begin
 {$R-}
 Result:= $FFFFFFFF;
 For iByte:= 0 to Len-1 do begin
  B:= Byte(Pointer(LongInt(P)+iByte)^);
  Result:= Crc32Table[Byte(Result xor Cardinal(B))]
                     xor ((Result shr 8) and $00FFFFFF);
 End;
 Result:= Result xor $FFFFFFFF;
 {$R+}
End;
```

Глава 16

Интерфейс RS-485

Ток по проводам шел с переменным успехом...

Теперь у нас есть все функции для работы с интерфейсом RS-232. Небольшие изменения кода позволят нам использовать интерфейс RS-485, который является основой многих промышленных стандартов. Хотя RS-485 является аппаратной "разновидностью" RS-232, он требует дополнительных программных решений и, по этой причине, рассматривается в отдельной главе.

16.1. Стандарт RS-485

Несмотря на распространенность, протокол RS-232 имеет несколько существенных недостатков:

- необходимость трехпроводной линии;
- длина соединяющих проводов не может превышать 10—15 метров;
- к одному порту компьютера может быть подключен только один абонент.

Избавиться от этих недостатков помогает специальный интерфейс RS-485, для реализации которого достаточно двухпроводной линии, длина кабеля может достигать 1 км (для низких скоростей передачи до 3 км), а количество подключаемых к одному порту устройств ограничено 32 абонентами.

Обычная система, основанная на стандарте RS-485, содержит несколько приемников, несколько формирователей и согласующих резисторов. Физически такая сеть представляет собой приемопередатчики, соединенные при помощи витой пары — двух скрученных проводов. Сигнал передается с помощью разности потенциалов между двумя проводами: по одному проводу идет оригинальный сигнал, а по второму — его инверсная копия (этот принцип называется *дифференциальной (балансной) передачей* данных). Другими словами, если на одном проводе 1, то на другом 0, и наоборот. Таким образом, между двумя проводами витой пары всегда есть разность потенциалов: при 1 она положительна, при 0 — отрицательна. Такой способ передачи обеспечивает высокую помехоустойчивость, т. к. помеха действует

практически одинаково на оба провода, оставляя разницу потенциалов неизменной.

Аппаратная часть этой системы представляет собой микросхемы приемопередатчиков с дифференциальными входами/выходами (к линии) и цифровыми портами (к портам UART-контроллера).

Интерфейс RS-485 представляет собой полудуплексный интерфейс. Прием и передача ведутся по одной паре проводов с разделением по времени. Во время приема отключается передатчик, а во время передачи — приемник. Для переключения приема и передачи используется один из управляющих сигналов порта. Этот сигнал подается на вход "разрешение приемника", а его инвертированное значение — на вход "разрешение передатчика".

Все устройства подключаются к витой паре одинаково: прямые выходы (называемые A) к одному проводу, а инверсные (называемые B) — к другому (рис. 16.1).

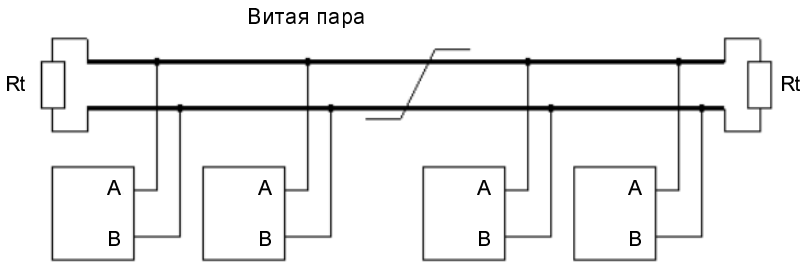

Рис. 16.1. Схема подключения устройств в RS-485

Количество подключаемых устройств ограничивается входным сопротивлением со стороны линии. Обычно эта величина составляет 12 кОм и, с учетом согласующих резисторов, ограничивает сеть 32 абонентами.

Более подробное обсуждение физической реализации сети выходит за рамки этой книги. Объяснения принципов построения сетей можно найти, например, в [2] и [4]. В [9] можно также найти хорошие рекомендации по правильной организации сетей.

16.2. Как работает COM-порт IBM PC

Прежде чем приступить непосредственно к программированию протокола, необходимо отметить некоторую специфику реализации последовательного интерфейса в IBM PC.

Последовательный интерфейс в IBM PC сделан буферным. При передаче символ закладывается в буфер, а затем уже оттуда передается в линию. Эта

тонкость никак не влияет на реализацию RS-232, однако оказывается существенной при реализации RS-485, т. к. абонентам сети необходимо переключать линию после получения идентификатора конца посылки. В IBM PC сделать это средствами, предоставляемыми микросхемой UART, невозможно.

Для демонстрации этого не вполне очевидного момента можно использовать простую программу, написанную на языке Pascal (DOS). Переключателем линии выберем вывод DTR, который будем сбрасывать в 0 при передаче данных от компьютера (листинг 16.1).

Листинг 16.1. Пробная программа для тестирования сигналов порта

```
Uses Crt, RS232int;

Var
 DELAY1, DELAY2, DELAY3 : Word;

(*  == Процедура передачи байта из модуля RS232Int ==
Procedure OutByteToRS232(b:Byte); assembler;
Asm
@Wait_Line:
  mov  dx,3FDh    { регистр состояния линии }
  in   al,dx
  test al,20h     { стык готов к передаче? }
  jnz  @Output    { да }
  nop
  loop @Wait_Line { нет, ждем }
  ret

@Output:
  mov  al, B
  mov  dx,3F8h    { регистр данных }
  out  dx,al      { вывести символ }
  nop
End;
*)

Procedure SendOn;
Begin
 Port[$3FC]:= 0; {включение передачи: DTR=0}
End;
```

```
Procedure SendOff;
Begin
 Port[$3FC]:= 1; {выключение передачи: DTR=1}
End;

Begin
 DELAY1:= 10; DELAY2:= 10; DELAY3:= 10;

 InitRS232(1, nil, 9600, 3); {инициализируем порт}
 SendOff;

 While (True) do begin
  SendOn;
  Delay(DELAY1);
  OutByteToRS232($55);
  OutByteToRS232($55);
  OutByteToRS232($55);
  OutByteToRS232($55);
  Delay(DELAY2);
  SendOff;
  Delay(DELAY3);

  If (KeyPressed) then begin
   Case ReadKey of
    {Клавиша ESC — выход}
    #27: Break;
    {изменение величины DELAY1}
    '1': DELAY1:= DELAY1 + 10;
    '2': DELAY1:= DELAY1 - 10;
    {изменение величины DELAY2}
    '3': DELAY2:= DELAY1 + 10;
    '4': DELAY2:= DELAY1 - 10;
    {изменение величины DELAY3}
    '5': DELAY1:= DELAY1 + 10;
    '6': DELAY1:= DELAY1 - 10;
   End;
   WriteLn('DELAY1=', DELAY1);
   WriteLn('DELAY2=', DELAY2);
```

```
   WriteLn('DELAY3=', DELAY3);
  End;

 End;

 While KeyPressed do ReadKey;

 CloseRS232;
End.
```

Подключив двухлучевой осциллограф к выводам передатчика и DTR, мы увидим картинку, показанную на рис. 16.2.

Пауза `DELAY3` задает интервал между посылками, пауза `DELAY1` — интервал между переключением линии и началом посылки.

Самая интересная величина — пауза `DELAY2`, задающая интервал между окончанием посылки и переключением линии. Установим `DELAY2 = 0` и заметим, что, несмотря на ожидание флага "передатчик готов" в процедуре `OutByteToRS232` (мы специально привели ее полный код из модуля `RS232Int`), линия DTR переключается примерно в начале отправки последнего байта.

Строго говоря, флаг "передатчик готов" вовсе не означает "байт передан".

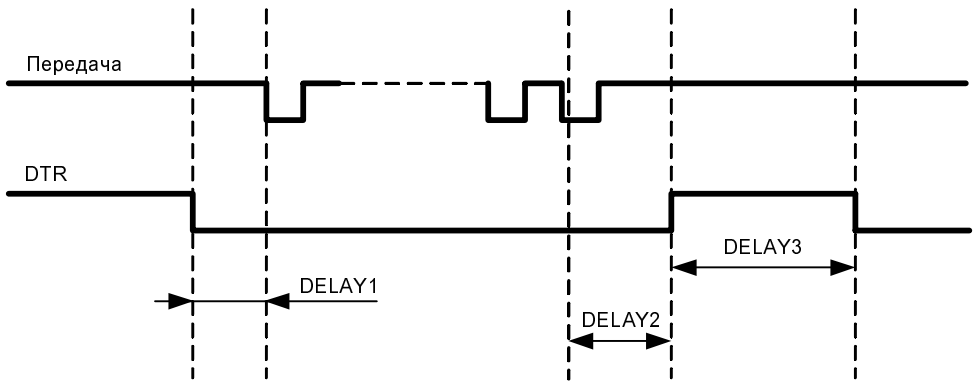

Рис. 16.2. Зависимость переключения линий от величин пауз DELAY тестовой программы для RS-485

Подведем итог эксперимента. Для правильной реализации протокола RS-485 необходимо задать величину `DELAY2` таким образом, чтобы переключение

линии произошло *заведомо после* передачи последнего байта. Вычислить эту величину можно только приблизительно. Обычно это делается так, как показано в листинге 16.2.

Листинг 16.2. Вычисление паузы отправки полного пакета данных

```
{вычислим число бит, необходимых для передачи одного символа:}
{     число бит в байте + 1 старт-бит + 1 стоп-бит           }
BitsPerChar := DCB.ByteSize + 2;

{если есть контроль четности, то еще один бит }
if (Parity <> 0 ) then inc(BitsPerChar);

{если используется 2 стоп-бита, то плюс еще один бит }
if (StopBits = 2) then inc(BitsPerChar);

{Baud задается в бит/сек — вычисляем кол-во необходимых}
{микросекунд на один бит}
MicroSecsPerBit := 10000000 div Baud;

{Микросекунд на буфер. 18 получено из 16 символов буфера}
{TX FIFO, 1 символ в TX Output и 1 один символ, если он }
{уже начал передаваться}
MicroSecs := MicroSecsPerBit * BitsPerChar * 18;

{Ограничение снизу}
if (MicroSecs < 10000) then MicroSecs := MicroSecs + MicroSecs;

{Микросекунды переводим в миллисекунды}
MilliSecs := Microsecs div 10000;

{Округляем вверх}
if( (Microsecs mod 10000) <> 0 ) then inc(MilliSecs);

{Пауза на полученное число миллисекунд}
Sleep(MilliSecs);
```

16.3. Реализация RS-485

Теперь мы готовы к правильной реализации протокола RS-485. На основе нашего компонента `TComBin` мы создадим компонент `TCom485`, реализующий двухпроводной интерфейс.

Хотя мы говорили, что для реализации различных протоколов нам не потребуется вносить изменения в код транспортного компонента, мы добавим в него пару свойств. Они никак не повлияют на наши предыдущие программы, но позволят включать ручное управление линиями RTS и DTR. Код, который нужно добавить в компонент `TComPort`, приведен в листинге 16.3.

Листинг 16.3. Свойства `TComPort` для управления линиями RTS и DTR

```
unit ComPort;

interface

uses
  Windows, SysUtils, Classes, Forms;
  ... ... ... ... ... ... ... ... ... ... ... ...

Type
 TComProp = class(TPersistent)
  Private
   ... ... ... ... ... ... ... ... ... ... ... ...
   FEnableRTS   : Boolean;   {разрешить управлять RTS}
   FEnableDTR   : Boolean;   {разрешить управлять DTR}
 Public
   ... ... ... ... ... ... ... ... ... ... ... ...
   {разрешить управлять RTS}
   Property EnableRTS : Boolean read FEnableRTS write FEnableRTS;
   {разрешить управлять DTR}
   Property EnableDTR : Boolean read FEnableDTR write FEnableDTR;
 End;
 ... ... ... ... ... ... ... ... ... ... ... ...
function TComProp.GetDCB: TDCB;
begin
 { Чистка структуры }
 ZeroMemory(@Result, SizeOf(TDCB));
```

```
 { Пеле DCBLength должно содержать размер структуры }
 Result.DCBLength:= SizeOf(TDCB);
 { Скорость обмена (бод) }
 Result.BaudRate := WindowsBaudRates[FBaudRate];
 { Windows не поддерживает небинарный режим работы }
 { последовательных портов }
 Result.Flags    := dcb_Binary;
 { Число бит в байте }
 Result.ByteSize := 5 + Ord(FByteSize);
 { Контроль четности }
 Result.Parity   := Ord(FParity);
 { Число стоп-бит }
 Result.StopBits := Ord(FStopbits);
 If FEnableRTS then {разрешить управлять RTS}
  Result.Flags:= Result.Flags or dcb_DtrControlEnable;
 If FEnableDTR then {разрешить управлять DTR}
  Result.Flags:= Result.Flags or dcb_RtsControlEnable;
end;
  ... ... ... ... ... ... ... ... ... ... ... ...
End.
```

Код компонента `TCom485` почти ничем не отличается от кода компонента `TComBin`, за исключением управления линией DTR, как показано в листинге 16.4.

Листинг 16.4. Управление линией DTR в протоколе RS-485

```
{Передача числа согласно протоколу}
procedure TCom485.SendValue(const AValue : Single);
Var CS : Byte;
begin
 If Assigned(FComPort) then
  If FComPort.Connected then begin
   {Включаем DTR - передача}
   FComPort.ToggleDTR(True);
   {Послать символ начала посылки}
   FComPort.WriteBuffer(@FStartByte, SizeOf(FStartByte));
   {Послать число}
   FComPort.WriteBuffer(@AValue, SizeOf(AValue    ));
```

```
    {Послать символ завершения посылки}
    FComPort.WriteBuffer(@FEndByte  , SizeOf(FEndByte  ));
    {контрольная сумма, если включено}
    If FUseCS then begin
     CS:= GetCSSingle(AValue);
     FComPort.WriteBuffer(@CS, SizeOf(CS));
    End;
    {Пауза для полной отправки посылки}
    Sleep(FComPort.ComProp.GetWaitTxDelay);
    {Выключаем DTR - конец передачи}
    FComPort.ToggleDTR(True);
  End;
end;
```

Глава 17

Работа Plug and Play

Программист — это не тот, кто пишет программы,
а тот, чьи программы работают.

В *гл. 5* мы описали, как работает Plug and Play для последовательных портов. В этой главе мы проведем несколько экспериментов и создадим программу, эмулирующую с помощью порта COM1 некоторое устройство, подключенное к COM2.

17.1. Эмулятор устройства

Наша задача — заставить Windows "поверить", что к порту COM2 подключено некое устройство, поддерживающее стандарт PnP. Так как мы договорились, что порты нашего компьютера соединены нуль-модемным кабелем, то для решения этой задачи нужно написать программу, отвечающую на запросы, поступающие к порту COM2 через порт COM1 (рис. 17.1).

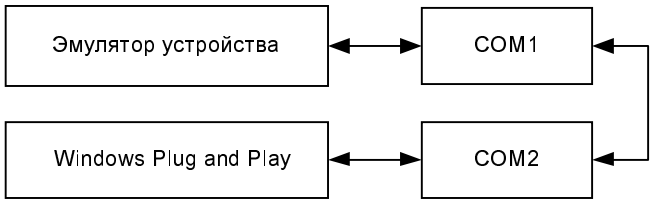

Рис. 17.1. Структурная схема эмулятора Plug and Play

Для реализации такой системы мы воспользуемся компонентом `TComPort`. На главную форму программы мы вынесем составляющие PnP-идентификатора и окно сообщений, в которое будем выводить производимые действия. Поскольку PnP работает на фиксированной скорости обмена, то кнопка **Настройка порта** нам не потребуется. Также мы удалим кнопку

Опрос порта. В результате у нас должна получиться форма, показанная на рис. 17.2.

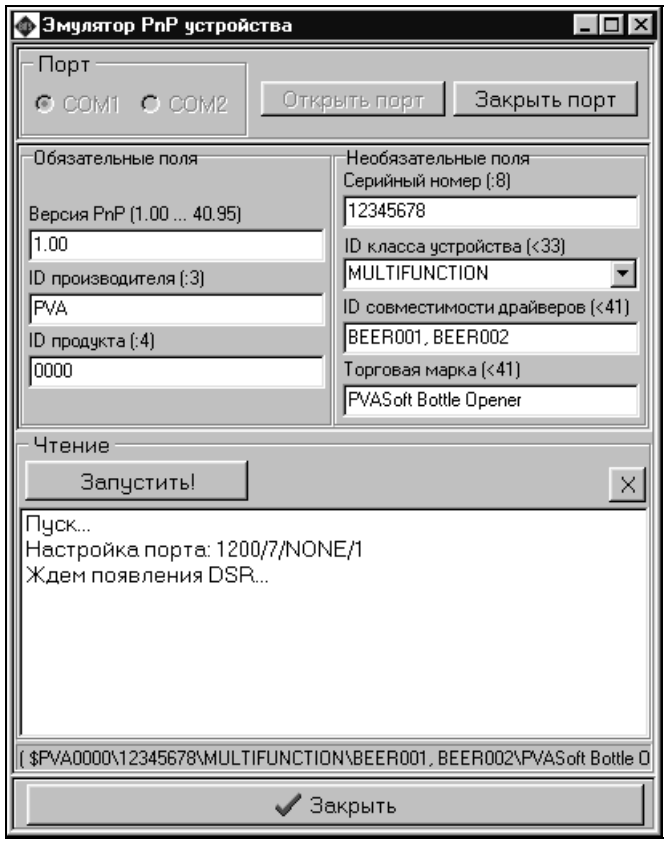

Рис. 17.2. Вид программы эмулятора PnP-устройства

Все поля PnP-идентификатора нам уже известны из *гл. 5*. Единственная тонкость — создание бинарного значения поля `PnP Rev` из его обычного значения. Для этого нам нужно провести операции, обратные описанным: умножить номер версии на 100 и выделить первые 6 бит и следующие 6 бит. Как это сделать, показано в листинге 17.1.

Листинг 17.1. Формирование поля `PnP Rev`

```
Function TForm1.GetPnPRevison(RevN : String) : String;
Var iRev : Integer; iRevH, iRevL : Byte;
```

```
Begin
 { RevN = строковое значение, например, 1.0 }
 { Получаем iRev — цифровое значение }
 iRev:= Trunc(StrToFloatDef(RevN, 0)*100);
 { Выделяем старшие 6 бит и младшие 6 бит}
 iRevH:= (iRev shr 6) and $3F;
 iRevL:= iRev and $3F;
 { Создаем строку из двух символов}
 Result:= Char(iRevH)+Char(iRevL);
End;
```

При формировании PnP-идентификатора не следует забывать, что контрольная сумма считается, включая значения полей `Begin ID` и `End ID`, но не включая значения самой контрольной суммы. Именно поэтому мы сначала вычислим "внутренности" PnP-идентификатора, потом добавим скобки (мы выбрали 7-битовый набор символов), затем подсчитаем контрольную сумму, и только после этого сформируем полный PnP-идентификатор.

Собственно сам алгоритм, реализующий ответы на запросы PnP-протокола, достаточно прост и полностью совпадает с алгоритмом протокола, описанным в *гл. 5*. Он запускается кнопкой **Запустить!** и работает до тех пор, пока открыт выбранный порт. Полный код получившейся программы показан в листинге 17.2.

Листинг 17.2. Код программы PnP-эмулятора

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls, Buttons, Spin, Mask,
  ComPort, ComCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    btnClose: TBitBtn;
    ParamPanel: TPanel;
```

```
    rgPortIndex: TRadioGroup;
    GroupBox1: TGroupBox;
    btnOpenPort: TSpeedButton;
    btnClosePort: TSpeedButton;
    StatusBar: TStatusBar;
    lbTrace: TListBox;
    FPort: TComPort;
    Panel4: TPanel;
    bRead: TSpeedButton;
    btnClearLog: TSpeedButton;
    Panel3: TPanel;
    GroupBox2: TGroupBox;
    PnP_Rev: TLabeledEdit;
    EISA_ID: TLabeledEdit;
    Product_ID: TLabeledEdit;
    GroupBox3: TGroupBox;
    Serial_Number: TLabeledEdit;
    Driver_ID: TLabeledEdit;
    User_Name: TLabeledEdit;
    Class_Name: TComboBox;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure bReadClick(Sender: TObject);
    procedure btnOpenPortClick(Sender: TObject);
    procedure btnClosePortClick(Sender: TObject);
    procedure rgPortIndexClick(Sender: TObject);
    procedure btnClearLogClick(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
  private
    Function GetPnPRevison(RevN : String) : String;
    Function GetPnPCS(SID : String) : Byte;
  public
   function  MakePnP_ID : String;
   procedure SetEnabledControls;
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}

{ Создание компонента доступа к портам при создании формы }
procedure TForm1.FormCreate(Sender: TObject);
begin
 DecimalSeparator:= '.';
 SetEnabledControls;
end;

{Изменение номера порта}
procedure TForm1.rgPortIndexClick(Sender: TObject);
begin
 FPort.ComNumber:= rgPortIndex.ItemIndex+1;
end;

{Кнопка открытия порта}
procedure TForm1.btnOpenPortClick(Sender: TObject);
begin
 FPort.Close;
 FPort.ComNumber:= rgPortIndex.ItemIndex+1;
 FPort.Open;
 If FPort.Connected then
  StatusBar.Panels[0].Text:= 'Порт успешно открыт'
 Else
  StatusBar.Panels[0].Text:= 'Ошибка открытия порта';
 SetEnabledControls;
 If FPort.Connected then begin
  FPort.ToggleRTS(False);
  FPort.ToggleDTR(False);
 End;
end;

{Кнопка закрытия порта}
procedure TForm1.btnClosePortClick(Sender: TObject);
begin
 FPort.Close;
 StatusBar.Panels[0].Text:= '';
```

```
 SetEnabledControls;
 bRead.Down:= False;
end;

{Установка активности элементов формы}
procedure TForm1.SetEnabledControls;
Var Active : Boolean;
begin
 Active:= FPort.Connected;
 btnClosePort.Enabled:= Active;
 rgPortIndex.Enabled := not Active;
 btnOpenPort.Enabled := not Active;
 bRead.Enabled       := Active;
end;

{Кнопка закрытия программы}
procedure TForm1.btnCloseClick(Sender: TObject);
begin
 Close;
end;

{ Получение контрольной суммы строки PnP-идентификатора }
Function TForm1.GetPnPCS(SID : String) : Byte;
Var i : Integer;
Begin
 Result:= 0;
 SID:= '(' + SID + ')';
 {суммируем}
 For i:= 1 to Length(SID) do begin
  {$R-}
  Result:= Result + Byte(SID[i]);
  {$R+}
 End;
End;

{ Формирование поля PnP Rev }
Function TForm1.GetPnPRevison(RevN : String) : String;
Var iRev : Integer; iRevH, iRevL : Byte;
```

```
Begin
 { RevN = строковое значение, например, 1.0 }
 { Получаем iRev - цифровое значение }
 iRev:= Trunc(StrToFloatDef(RevN, 0)*100);
 { Выделяем старшие 6 бит и младшие 6 бит}
 iRevH:= (iRev shr 6) and $3F;
 iRevL:= iRev and $3F;
 { Создаем строку из двух символов}
 Result:= Char(iRevH)+Char(iRevL);
End;

{ Формирование PnP-идентификатора}
Function TForm1.MakePnP_ID : String;
Begin
 {Внутреннее содержание PnP ID}
 Result:=
  Format('%s%3s%4s\%s\%s\%s\%s',[
    GetPnPRevison(PnP_Rev.Text),
    EISA_ID.Text,
    Product_ID.Text,
    Serial_Number.Text,
    Class_Name.Text,
    Driver_ID.Text,
    User_Name.Text
  ]);
{ Должно получиться что-то вроде строки: }
{ "#$01#$24'PVA0000\12345678\MULTIFUNCTION\BEER001, BEER002\Bottle
Opener" }

 { Добавить BeginID, EndID и контрольную сумму }
 Result:= Format('(%s%s)',[Result, IntToHex(GetPnPCS(Result),2)]);
End;

{Запустить!}
procedure TForm1.bReadClick(Sender: TObject);
begin
 StatusBar.Panels[0].Text:=  MakePnP_ID;

 lbTrace.Items.Add('Пуск...');
 lbTrace.Items.Add('Настройка порта: 1200/7/NONE/1');
```

```
 FPort.ComProp.BaudRate:= br1200;
 FPort.ComProp.ByteSize:= bs7;
 FPort.ComProp.Parity  := ptNONE;
 FPort.ComProp.Stopbits:= sb1BITS;
 While FPort.Connected do begin
  lbTrace.Items.Add('Ждем появления DSR...');
  While (FPort.GetModemState and MS_DSR_ON) = 0 do
   Application.ProcessMessages;
  lbTrace.Items.Add('Устанавливаем в ответ DTR');
  FPort.ToggleDTR(True);
  lbTrace.Items.Add('Ждем установки CTS...');
  While (FPort.GetModemState and MS_CTS_ON) = 0 do
   Application.ProcessMessages;
  lbTrace.Items.Add('Передаем идентификатор');
  FPort.WriteString(MakePnP_ID);
 End;
 lbTrace.Items.Add('END');
end;

{Кнопка очистки лог-файла}
procedure TForm1.btnClearLogClick(Sender: TObject);
begin
 lbTrace.Items.Clear;
end;

{Закрытие порта перед закрытием формы,}
{позволяющее завершиться циклу PnP-эмуляции}
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
 FPort.Close;
end;

end.
```

Мы надеемся, что такое полезное устройство, как Bottle Opener (открывалка бутылок) обязательно пригодится вам в вашей работе!

Итак, все готово. Запускаем программу. Открыв порт COM1, нажимаем кнопку **Запустить!**. Теперь надо активизировать алгоритм обнаружения новых устройств. Сделать это можно разными путями, которые описаны в *гл. 5.*

В Windows 98 возможны проблемы, и, может быть, придется сначала удалить COM-порты из системы, а затем найти их заново.

После нескольких секунд работы PnP-сканер обратится к порту COM2 и обнаружит, что появилось новое устройство (рис. 17.3)!

Рис. 17.3. Windows 2000 обнаружила новое устройство

После обнаружения нового устройства Windows предложит поискать и установить драйвер для него (рис. 17.4). Этим мы займемся в следующем разделе.

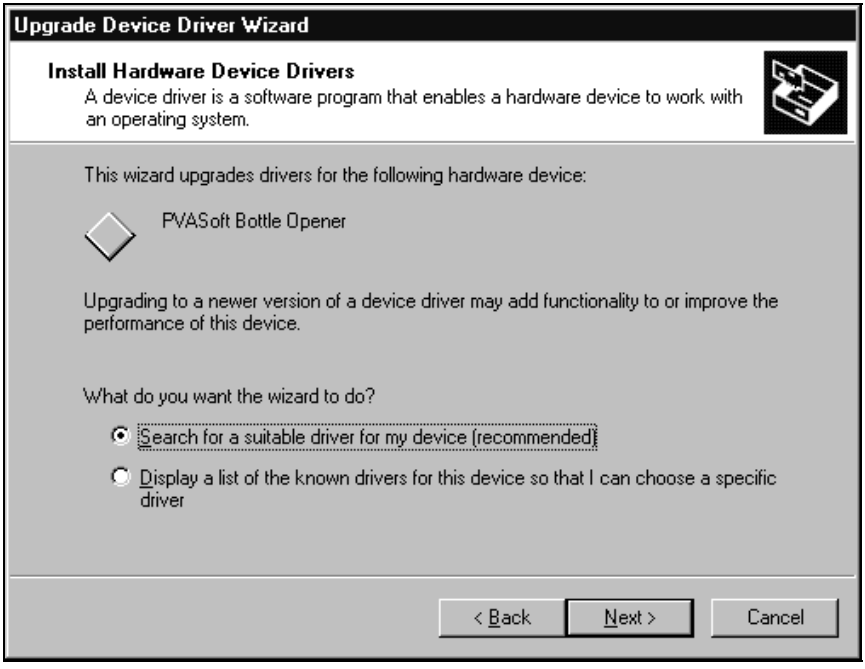

Рис. 17.4. Windows 2000 предлагает установить драйвер

17.2. Установка драйвера устройства

Как мы уже описывали в *гл. 5*, для установки драйвера устройства Windows ищет специальный файл с расширением inf. Этот файл содержит всю информацию о действиях, которые необходимо произвести для установки драйвера: какие файлы нужно перенести и зарегистрировать в системе, какие ветки реестра необходимо создать и т. д. Кроме того, этот же файл содержит информацию о действиях, которые нужно произвести при удалении устройства из системы, а также дополнительную информацию о производителе устройства.

"За кадром" остался довольно интересный вопрос: какой именно INF-файл ищет Windows при установке нового устройства? Очевидно, что совершенно произвольный файл не подходит.

Для ответа на этот вопрос приведем листинг "правильного" INF-файла для нашего устройства (листинг 17.3).

Листинг 17.3. INF-файл для нового устройства

```
; INF-файл для устройства Bottle Opener

[Version]
; Сигнатура "для всех версий Windows"
signature="$CHICAGO$"
; Класс устройства
Class=MODEM
; Ссылка на строку из раздела Strings
Provider=%Mfg%

; Производитель — для каждого производителя
; указывается ссылка на соответствующую
; секцию описания
[Manufacturer]
%Mfg%=CompanyName

; Директория
[DestinationDirs]
DefaultDestDir=30,MyFolder

[CompanyName]
; Ссылка на раздел MyDevice_SECTION для устройств,
```

```
; подключенных к последовательному порту с серийным
; номером PVA0000
%MyDeviceStr%=MyDevice_SECTION, SERENUM\PVA0000

; Свойства устройства — изменения реестра,
; копируемые файлы
[MyDevice_SECTION]
CopyFiles=COPY_SECTION
AddReg=MyDevice.AddReg
DelReg=MyDevice.DelReg
LogConfig=MyDevice.Config

[MyDevice.AddReg]
; ничего не записываем в реестр при install

[MyDevice.DelReg]
; ничего не будем удалять при uninstall

[MyDevice.Config]
; ничего не конфигурируем

[COPY_SECTION]
; драйвер устройства — это notepad.exe
notepad.exe ;=
win.ini ;=

; строки. Эта секция может меняться для разных языков
[Strings]
Mfg="PVASoft"
MyDeviceStr="PVASoft Bottle Opener"
```

Изменяя значения полей, легко выяснить, что контрольными полями при установке драйвера являются:

- поле `Signature` секции `Version` должно содержать правильную версию Windows или строку `"$CHICAGO$"`, говорящую, что драйвер подходит для всех версий Windows;
- поле `Class` секции `Version` должно содержать правильный идентификатор класса, который может не совпадать с идентификатором, переданным устройством, но должен быть зарезервированным словом из табл. 5.3;

- секция CompanyName (правильнее сказать — та секция, на которую делается ссылка из Manufacturer для выбранного производителя устройства) должна содержать поле, в левой части которого указан тип подключения (в нашем случае это SERENUM — устройство, подключаемое к последовательному порту), а в правой — идентификатор устройства, совпадающий со строками "ID производителя" и "ID продукта" (в нашем случае это PVA0000), разделенные знаком "\".

Секция Company Name может содержать описание нескольких устройств с разными серийными номерами. Для каждого серийного номера указывается своя секция установок (в нашем файле это секция MyDevice_SECTION). Секция установок содержит ссылки на секции CopyFiles (копируемые файлы), AddReg (регистрация в реестре при установке устройства), DelReg (удаляемые из реестра ветки при удалении устройства) и LogConfig (конфигурирование устройства).

Для описания нескольких устройств с разными серийными номерами можно использовать разделитель ",":

```
%MyDeviceStr% = MyDevice_SECTION, SERENUM\PVA0001, PVA0000
```

Рис. 17.5. Установка драйвера устройства

Если файл сформирован правильно, Windows установит новое устройство. В качестве файла драйвера мы указали стандартный Блокнот (Notepad) и

файл win.ini. При установке Windows спросит о его местонахождении. После установки новое устройство будет добавлено в систему (рис. 17.5).

Естественно, Windows будет сообщать об ошибках в драйвере, т. к. собственно драйвер мы и не установили, а Блокнот (Notepad) не знает, как отвечать на запросы Windows. Тем не менее, новое устройство будет успешно добавлено в систему (рис. 17.6).

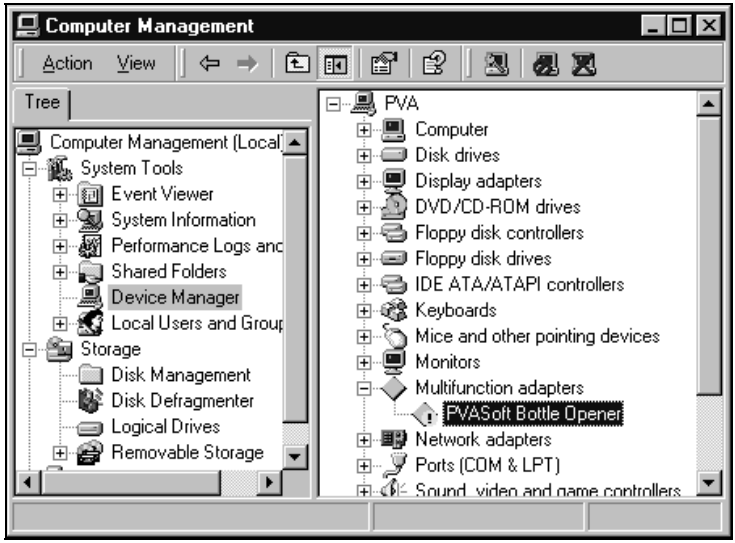

Рис. 17.6. Новое устройство установлено

Заметьте, что на системном диске создалась директория MyFolder, в которую скопированы файлы notepad.exe и win.ini. Эту директорию мы указали в секции `DestinationDirs`, а список файлов в секции `COPY_SECTION`.

17.3. Обнаружение изменений

Еще один вопрос, часто возникающий при программировании: как в другой программе обнаружить изменения аппаратной конфигурации?

Сделать это очень просто. При изменениях в аппаратной конфигурации Windows рассылает сообщение `WM_DEVICECHANGE`, передающее структуру `TWMDeviceChange`. Эта структура описывается следующим образом:

```
type
 TWMDeviceChange = record
   Msg:    Cardinal;
   Event:  UINT;      { код события  }
```

```
   dwData: Pointer;  { данные       }
   Result: LongInt;
  end;
```

Код события — одна из констант DBT_XXX. Описания этих констант нет в Delphi (по крайней мере, в Delphi версий 3—7), поэтому нам придется позаимствовать их из файла dbt.h набора заголовков Visual Studio. Полный файл dbt.pas можно найти на прилагаемом компакт-диске, а нас, прежде всего, интересуют константы:

```
DBT_DEVICEARRIVAL        = $8000
DBT_DEVICEREMOVECOMPLETE = $8004
```

Первый код события означает обнаружение нового устройства, а второй код — отключение устройства. Следует отметить, что сообщение о подключении устройства будет послано только в случае успешной установки устройства, т. е. в нашем случае оно послано не будет: мы не установили реальный драйвер устройства, и Windows законно считает, что устройство установлено с ошибкой. А вот подключение или отключение флеш-диска будет зарегистрировано.

В Windows 2000 добавлена еще одна константа

```
DBT_DEVNODES_CHANGED = $0007
```

Это событие сообщает об изменении в списке узлов дерева устройств в Менеджере устройств (Device Manager) (рис. 17.6). В нашем случае был добавлен узел Multifunction Adapter, поэтому мы получим это сообщение, несмотря на то, что наше устройство подключено некорректно. Вообще говоря, изменение в списке самих устройств вызывает изменение узла дерева устройств, поэтому это сообщение мы будем получать достаточно часто.

Итак, для примера напишем небольшую программу, которая производит обработку сообщения WM_DEVICECHANGE и отображает результат на экране. Собственно, весь код этой программы будет сводиться к обработчику сообщения (листинг 17.4).

Листинг 17.4. Обработка сообщения WM_DEVICECHANGE

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, DBT;
```

```
type
  TForm1 = class(TForm)
    lbDevice: TListBox;
  private
    {обработчик сообщения WM_DEVICECHANGE}
    procedure WMDEVICECHANGE(var Msg : TWMDeviceChange); message
WM_DEVICECHANGE;
  public
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

{обработчик сообщения WM_DEVICECHANGE}
procedure TForm1.WMDEVICECHANGE(var Msg: TWMDeviceChange);
var
  lpdb  : PDevBroadcastHdr;
  lpdbv : PDevBroadcastVolume;
  lpdbpr: PDevBroadCastPort;

begin
 {Заголовок сообщения}
 lpdb := PDevBroadcastHdr(Msg.dwData);

 {Отображаем код события}
 lbDevice.Items.Add('Обнаружено событие. Код:'+ IntToHex(Msg.Event, 4));

 Case Msg.Event of
  DBT_DEVICEARRIVAL: begin {Добавление}
   lbDevice.Items.Add('>Добавлено устройство. Код:' +
                          IntToHex(lpdb^.dbch_devicetype, 4));
```

```
   { Новое устройство — порт (последовательный или параллельный) }
   If lpdb^.dbch_devicetype = DBT_DEVTYP_PORT then begin
    lpdbpr:= PDevBroadCastPort(Msg.dwData);
    lbDevice.Items.Add('>>Добавлен порт. Имя:'+lpdbpr.dbcp_name);
   End;

   { Новое устройство — логический диск }
   If lpdb^.dbch_devicetype = DBT_DEVTYP_VOLUME then begin
    lpdbv := PDevBroadcastVolume(Msg.dwData);
    lbDevice.Items.Add('>>Добавлен логический диск');
   End;
  End;

  DBT_DEVICEREMOVECOMPLETE: begin {Удаление}
   lbDevice.Items.Add('>Удалено устройство.
Код:'+IntToHex(lpdb^.dbch_devicetype, 4));

   { Удаленное устройство — порт }
   {(последовательный или параллельный) }
   If lpdb^.dbch_devicetype = DBT_DEVTYP_PORT then begin
    lpdbpr:= PDevBroadCastPort(Msg.dwData);
    lbDevice.Items.Add('>>Удален порт. Имя:'+lpdbpr.dbcp_name);
   End;

   { Удаленное устройство — логический диск }
   If lpdb^.dbch_devicetype = DBT_DEVTYP_VOLUME then begin
    lpdbv := PDevBroadcastVolume(Msg.dwData);
    lbDevice.Items.Add('>>Удален логический диск');
   End;
  End;
 End;

end;

end.
```

Для демонстрации мы будем обрабатывать два события: добавление или удаление порта (последовательного или параллельного) и добавление или уда-

ление логического диска. Работу первого сообщения можно проверить, установив опцию **Выключено** (Disable) для порта COM1 и затем, включив его снова (рис. 17.7).

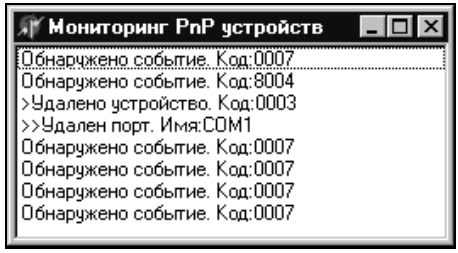

Рис. 17.7. Обнаружение изменений в аппаратной конфигурации

ЧАСТЬ III

СПРАВОЧНИК

Глава 18

Сервис BIOS INT14H/INT21H

18.1. Сервис BIOS INT14H

Порты персонального компьютера поддерживаются сервисом BIOS INT14H, который имеет следующие функции:

- 00 — инициализация порта;
- 01 — передача байта;
- 02 — прием байта;
- 03 — получение состояния порта;
- 04 — расширенная инициализация;
- 05 — расширенное управление портом.

Две последние функции применяются только в System/2.

Для вызова конкретной функции ее номер задается в регистре AH (листинг 18.1). Номер порта задается в регистре DX и кодируется следующим образом: значение 0 задает порт COM1, значение 1 задает COM2 и т. д. Точнее говоря, AH задает номер канала в соответствии с базовым адресом портов в таблице по адресу 40:0.

В зависимости от функции входные параметры могут указываться в регистрах BH, AL, CH, CL. При выходе все регистры, кроме регистра AX, сохраняются (это правило не распространяется на использование расширенных функций).

Листинг 18.1. Вызов INT14H

```
MOV AH, номер_функции
MOV DX, номер_порта ; (0 — COM1, 1 — COM2,...)
MOV AL, параметр-1  ; если надо
MOV BH, параметр-2  ; если надо
MOV CH, параметр-3  ; если надо
MOV CL, параметр-4  ; если надо
INT 14H
; выход — регистры AH и (или) AL
```

Ниже мы перечислим все функции сервиса INT14H. При описании будем использовать несколько сокращений. Если регистр должен иметь фиксированное значение, мы будем писать знак равенства "=", например `AH = 3`. Биты регистра будем указывать в квадратных скобках, например `AH[4:3]` будет означать биты 4 и 3 регистра `AH`. Если значение какого-либо бита не существенно, мы будем обозначать его символом "x". Например, запись "x1" означает, что первый бит не важен, а второй должен равняться 1.

18.1.1. Функция 00: инициализация порта

Функция 00 инициализирует порт. Ее входными регистрами являются:

- `AH = 0;`
- `DX` — номер порта;
- `AL` — флаги инициализации порта:
  - биты [7:5] задают скорость обмена:
    - ◊ 000 — 110 бит/с;
    - ◊ 001 — 150 бит/с;
    - ◊ 010 — 300 бит/с;
    - ◊ 011 — 600 бит/с;
    - ◊ 100 — 1200 бит/с;
    - ◊ 101 — 2400 бит/с;
    - ◊ 110 — 4800 бит/с;
    - ◊ 111 — 9600 бит/с;
  - биты [4:3] задают код контроля четности:
    - ◊ x0 — нет;
    - ◊ 01 — нечет;
    - ◊ 11 — чет;
  - бит 2 задает число стоп-бит:
    - ◊ 0 — 1 стоп-бит;
    - ◊ 1 — 2 стоп-бита;
  - биты [1:0] задают длину слова:
    - ◊ 00 — 5 бит;
    - ◊ 01 — 6 бит;
    - ◊ 10 — 7 бит;
    - ◊ 11 — 8 бит.

На выходе функция возвращает статус линии в регистре AH, как описано в *разд. 18.1.4*.

По умолчанию (при запуске DOS) порт COM1 инициализируется так:

- 2400 бит/с;
- без проверки на четность;
- 1 стоп-бит;
- 8-битовые слова.

Команда MODE используется для установки иных характеристик.

18.1.2. Функция 01: посылка символа в порт

Функция 01 передает один символ в указанный порт. Входными регистрами являются:

- AH = 1;
- DX — номер порта;
- AL — посылаемый символ.

На выходе регистр AL будет сохранен. Если в регистре AH установлен бит 7, это свидетельствует об ошибке, и биты AH[6:0] хранят статус линии связи *(см. разд. 18.1.4)*.

18.1.3. Функция 02: получение символа

Функция 02 принимает один символ с выбранного коммуникационного порта. Входными параметрами функции является содержимое двух регистров:

- AH = 2;
- DX — номер порта.

На выходе в регистре AL содержится полученный символ. Если произошла ошибка, регистр AH будет содержать ненулевое значение.

18.1.4. Функция 03: получить статус порта

Функция 03 возвращает статус порта. Эту функцию следует вызывать, чтобы узнать готовность порта к приему или передаче. Входными параметрами функции 03 является содержимое двух регистров:

- AH = 3;
- DX — номер порта.

На выходе регистр AH содержит статус линии:

- бит 7 — тайм-аут;
- бит 6 — регистр сдвига передатчика пуст (пауза передачи);
- бит 5 — промежуточный регистр передатчика пуст (готов принять символ для передачи);
- бит 4 — обнаружен обрыв линии;
- бит 3 — ошибка кадра (отсутствие стоп-бита);
- бит 2 — ошибка паритета принятого символа;
- бит 1 — переполнение (потеря символа);
- бит 0 — регистр данных содержит принятый символ.

Регистр AL содержит статус состояния модема:

- бит 7 — состояние линии DCD;
- бит 6 — состояние линии RI;
- бит 5 — состояние линии DSR;
- бит 4 — состояние линии CTS;
- бит 3 — изменение состояния DCD;
- бит 2 — изменение огибающей RI;
- бит 1 — изменение состояния DSR;
- бит 0 — изменение состояния CTS.

18.1.5. Функция 04: расширенная инициализация (System/2)

Функция 04 осуществляет расширенную инициализацию порта и присутствует только в System/2. Входные параметры функции содержатся в следующих регистрах:

- AH = 4;
- DX — номер порта (0—3) в соответствии с базовым адресом портов в таблице по адресу 40:0;
- AL — обработка сигнала Break:
  - 0 — без обработки Break;
  - 1 — с обработкой Break;
- BH — контроль паритета:
  - 0 — нет контроля;
  - 1 — контроль по нечетности;

- 2 — контроль по четности;
- 3 — постоянный нечет;
- 4 — постоянный чет;

❒ BL — число стоп-битов:

- 0 — 1 стоп-бит;
- 1 — 2 стоп-бита для 6—8-битового кода, 1,5 стоп-бита для 5-битового кода;

❒ CH — число битов в байте:

- 0 — 5-битовый код;
- 1 — 6-битовый код;
- 2 — 7-битовый код;
- 3 — 8-битовый код;

❒ CL — скорость обмена:

- 0 — 110 бит/с;
- 1 — 150 бит/с;
- 2 — 300 бит/с;
- 3 — 600 бит/с;
- 4 — 1200 бит/с;
- 5 — 2400 бит/с;
- 6 — 4800 бит/с;
- 7 — 9600 бит/с;
- 8 — 19 200 бит/с.

На выходе регистр AH будет содержать состояние линии управления, а регистр AL — состояние модема.

18.1.6. Функция 05: расширенное управление портом (System/2)

Функция 05 обеспечивает расширенное управление портом и присутствует только в System/2. Функция выполняет две подфункции в зависимости от значения регистра AL.

Входные параметры первой подфункции:

❒ AX = 0500H;

❒ DX — номер порта (0—3) в соответствии с таблицей базовых адресов по адресу 40:0.

Выходом является состояние регистра управления модемом в регистре BL:

- биты [7:5] — нули;
- бит 4 — цикл (возможность диагностирования);
- бит 3 — управление выходом OUT2;
- бит 2 — управление выходом OUT1;
- бит 1 — запрос на передачу;
- бит 0 — терминал данных готов (DTR).

Входные регистры второй подфункции:

- AX = 0501H;
- DX — номер порта (0—3) в соответствии с таблицей базовых адресов по адресу 40:0;
- BL — состояние регистра управления модемом (как в функции 0500H).

На выходе в регистре AH содержится состояние линии управления, а в регистре AL — состояние модема (как в функции 03, см. *разд. 18.1.4*).

18.2. Сервис BIOS INT21H

Функции 03H, 04H прерывания INT 21H позволяют работать с коммуникационным портом COM1 (который также называется *стандартным вспомогательным устройством*, AUX (AUXiliary)). Для работы с COM2 необходимо поменять местами базовые адреса портов в области 40:0 (см. листинг 6.3).

Для DOS коммуникационный порт является последовательным файлом, доступным для чтения и записи. Для работы с файлами предусмотрены функции 3FH и 40H прерывания INT21H. Эти функции работают с файлами через специальный идентификатор файла — "описатель" (прототип дескриптора файла в Windows). Для работы со стандартными устройствами зарезервированы несколько первых номеров описателей:

- 0 — стандартное устройство ввода (обычно клавиатура);
- 1 — стандартное устройство вывода (обычно экран);
- 2 — стандартное устройство ошибок (всегда CON-экран);
- 3 — стандартное устройство AUX (первый порт COM1);
- 4 — стандартный принтер (LPT1).

Таким образом, для коммуникационного порта зарезервирован описатель 3.

18.2.1. Функция 03H: вспомогательный ввод

Эта функция считывает (ожидает) символ со стандартного вспомогательного устройства COM1 и возвращает этот символ в AL. Ввод не буферизируется и должен опрашиваться (не управляется прерываниями).

На входе задается только номер функции в регистре AH (AH = 3), на выходе в AL записан символ, полученный от AUX.

18.2.2. Функция 04H: вспомогательный вывод

Функция 04H посылает символ в DL на стандартное вспомогательное устройство. Входными регистрами являются:

- AH = 4;
- DL — символ, выводимый на AUX/COM1.

Выходных регистров нет.

18.2.3. Функция 3FH: чтение данных через описатель

При вызове этой функции количество байтов, равное CX, считывается из файла или устройства с описателем, указанным в BX. Данные читаются с текущей позиции указателя чтения/записи файла и помещаются в буфер вызывающей программы, адресуемый через DS:DX. Итак, входными регистрами являются:

- AH = 3FH;
- BX — описатель файла;
- DS:DX — адрес буфера для чтения данных;
- CX — число считываемых байтов (обычно это размер буфера DS:DX).

После вызова необходимо сравнивать возвращаемое значение AX (число прочитанных байтов) с CX (запрошенное число байт):

- если AX = CX и CF сброшен — чтение без ошибок;
- если AX = 0, достигнут конец файла (EOF);
- если AX < CX (но AX ненулевой):
  - при чтении с устройства — входная строка имеет длину, равную содержимому AX в байтах;
  - при чтении из файла — в процессе чтения достигнут EOF.

При чтении с устройства в AX содержится длина считанной строки с учетом завершающего возврата каретки CR (ASCII-код 0DH).

18.2.4. Функция 40H: запись данных через описатель

При вызове этой функции CX байт данных записывается в файл или на устройство с описателем, заданным в BX. Данные считываются из буфера, адресуемого через DS:DX, и записываются, начиная с текущей позиции указателя чтения/записи файла. Входными регистрами являются:

- AH = 40H;
- BX — описатель файла (как в функции 3FH, см. *разд. 18.2.3*);
- DS:DX — адрес буфера, содержащего записываемые данные;
- CX — число записываемых байтов.

После вызова функции необходимо сравнивать возвращаемое значение AX (число записанных байтов) с CX (запрошенное число байтов для записи):

- если AX = CX, запись была успешной;
- если AX < CX, встретилась ошибка (чаще всего переполнение).

Глава 19

Порты IBM PC

Компьютер может иметь до четырех последовательных портов COM1—COM4 (в большинстве случаев имеются два порта) с поддержкой на уровне BIOS.

Коммуникационные порты занимают в пространстве ввода/вывода по 8 смежных 8-битовых регистров и могут располагаться по стандартным базовым адресам. Порты могут генерировать аппаратные прерывания.

Конфигурирование портов на плате расширения производится перемычками, а порты на системной плате конфигурируются через BIOS Setup.

В табл. 19.1 для четырех коммуникационных портов представлены базовые адреса, номер IRQ, номер вектора прерывания, базовый адрес порта и адрес тайм-аута порта.

Таблица 19.1. *Ресурсы BIOS коммуникационных портов*

Порт	Базовый адрес	IRQ	Вектор прерывания	Базовый адрес порта	Адрес тайм-аута порта (с)
COM1	3F8H	IRQ4 (или 11)	INT 0CH	WORD 0:0400	BYTE 0:047C
COM2	2F8H	IRQ3 (или 10)	INT 0BH	WORD 0:0402	BYTE 0:047D
COM3	3E8H			WORD 0:0404	BYTE 0:047E
COM4	2E8H			WORD 0:0406	BYTE 0:047F

Примечание

Для портов COM3 и COM4 возможны альтернативные адреса 3E0H, 338H и 2E0H, 228H соответственно. Для PS/2 стандартными для портов COM3—COM8 являются адреса 3320H, 3228H, 4220H, 4228H, 5220H, 5228H соответственно. В PS/2 все порты вызывают прерывание IRQ3.

Важно

Для простоты мы будем указывать все порты относительно базового адреса 3F8H, однако все описанное будет справедливо и для 2FxH.

19.1. Порт 3F8H (BasePort+0): данные и делитель

Этот порт работает по-разному в зависимости от бита DLAB в байте порта 3FBH:

- если `DLAB = 0`, производятся следующие операции:
  - запись в регистр `TSR` 8 битов передаваемого байта;
  - чтение из регистра `RBR` 8 битов принимаемого байта;
- если `DLAB = 1`, то в этот порт пишется младший байт делителя часов, который вместе со старшим байтом (порт 3F9H) составляет 16-битовое значение, задающее скорость передачи (табл. 19.2).

19.2. Порт 3F9H (BasePort+1): разрешение прерываний и делитель

Работа этого порта, также как и порта 3F8H, зависит от бита `DLAB`:

- если `DLAB = 0`, то в регистре разрешения прерываний `IER` (Interrupt Enable Register):
  - биты [7:6] не используются;
  - бит 5 — включение режима низкого потребления (Low Power Mode, *только микросхема 16750*);
  - бит 4 — включение "спящего" режима (Sleep Mode, *только микросхема 16750*);
  - бит 3 — разрешение прерывания по статусу модема (CTS, DSR, RI, RLSD);
  - бит 2 — разрешение прерывания по обрыву или ошибке линии;
  - бит 1 — разрешение прерывания, когда буфер передачи пуст (завершение передачи);
  - бит 0 — прерывание при готовности принимаемых данных (по приему символа). В режиме FIFO — прерывание по тайм-ауту;
- если `DLAB = 1`, то в порт 3F8H пишется старший байт делителя.

19.3. Порт 3FAH (BasePort+2): идентификация прерываний

Если происходит прерывание, то для того, чтобы выяснить причину, следует прочесть регистр идентификации прерываний `IIR` (Interrupt Identification Register). Для упрощения анализа UART выстраивает внутренние запросы прерывания по системе приоритетов. Порядок приоритетов (по убыванию) следующий:

- ❒ состояние линии;
- ❒ прием символа;
- ❒ освобождение регистра передатчика;
- ❒ состояние модема.

При возникновении условий прерывания UART указывает на источник с высшим приоритетом до тех пор, пока он не будет сброшен. Только после этого будет выставлен запрос с указанием следующего источника.

Кроме идентификации прерываний, регистр `IIR` содержит биты режима FIFO:

- ❒ биты [7:6] — признак режима FIFO:
  - 11 — режим FIFO 16550A;
  - 10 — FIFO включен, но не используется;
  - 00 — обычный режим;
- ❒ бит 5 — включение 64-байтового буфера FIFO *(только для 16750)*;
- ❒ бит 4 — не используется;
- ❒ бит 3 — прерывание по тайм-ауту приема в режиме FIFO (в буфере есть символы для считывания); для микросхем 8250 и 16450 не используется;
- ❒ биты [2:1] — причина прерывания с наивысшим приоритетом (в обычном, не-FIFO-режиме):
  - 11 — ошибка/обрыв линии, сброс выполняется чтением регистра состояния линии;
  - 10 — принят символ, сброс выполняется чтением данных;
  - 01 — передан символ (регистр `THR` пуст), сброс выполняется записью данных;
  - 00 — изменение состояния модема, сброс выполняется чтением регистра состояния модема;

- бит 0 может использоваться при программном опросе, и указывает на ожидание прерывания. Если этот бит равен 0, прерывание ожидает, а содержимое IIR может использоваться в качестве указателя типа прерывания для программы обслуживания прерывания. Когда он равен 1, ожидающего прерывания нет, и опрос продолжается (если он используется).

В режиме FIFO причину прерывания идентифицируют биты [3:1].

- 011 — ошибка/обрыв линии. Сброс выполняется чтением регистра состояния линии.
- 010 — принят символ. Сброс выполняется чтением регистра данных приемника.
- 110 — индикатор тайм-аута (за интервал времени, равный времени передачи четырех символов, не передано и не принято ни одного символа, хотя в буфере имеется, по крайней мере, один). Сброс выполняется чтением регистра данных приемника.
- 001 — регистр THR пуст. Сброс выполняется записью данных.
- 000 — изменение состояния модема (CTS, DSR, RI или DCD). Сброс выполняется чтением регистра MSR.

19.4. Порт 3FBH (BasePort+3): управление линией

Регистр управления линией LCR (Line Control Register) имеет следующие биты.

- Бит 7, DLAB — управление доступом к делителю частоты. После выполнения команды OUT 3FBH, 80H можно записать в 3F8H младшую часть, а в 3F9H — старшую часть кода делителя частоты *(см. разд. 19.1)*.
- Бит 6, BRCON (Break Control) — формирование обрыва линии (посылка нулей) при BRCON = 1.
- Бит 5, STICPAR (Sticky Parity) — принудительное формирование бита паритета:
  - 0 — контрольный бит генерируется в соответствии с паритетом выводимого символа;
  - 1 — постоянное значение контрольного бита: при EVENPAR = 1 — нулевое, при EVENPAR = 0 — единичное.
- Бит 4, EVENPAR (Even Parity Select) — выбор типа контроля:
  - 0 — нечетность;
  - 1 — четность.

- Бит 3, PAREN (Parity Enable) — разрешение контрольного бита:
  - 1 — контрольный бит (паритет или постоянный) разрешен;
  - 0 — контрольный бит запрещен.
- Бит 2, STOPB (Stop Bits) — количество стоп-бит:
  - 0 — 1 стоп-бит;
  - 1 — 2 стоп-бита (для 5-битового кода длина будет 1,5).
- Биты [1:0], SERIALDB (Serial Data Bits) — количество бит данных:
  - 00 — 5 бит;
  - 01 — 6 бит;
  - 10 — 7 бит;
  - 11 — 8 бит.

19.5. Порт 3FCH (BasePort+4): управление модемом

Регистр управления модемом MCR (Modem Control Register).

- Биты [7:5] зарезервированы и должны быть равны 0.
- Бит 4, LME (Loop Back Mode Enable) — разрешение режима диагностики:
  - 0 — нормальный режим;
  - 1 — режим диагностики.
- Бит 3, IE (Interrupt Enable) — разрешение прерываний с помощью внешнего выхода OUT2; в режиме диагностики поступает на вход MSR.7:
  - 0 — прерывания запрещены;
  - 1 — прерывания разрешены.
- Бит 2, OUT1C (OUT1 Bit Control) — управление выходным сигналом 1 (не используется); в режиме диагностики поступает на вход MSR.6.
- Бит 1, RTSC (Request To Send Control) — управление выходом RTS; в режиме диагностики поступает на вход MSR.4:
  - 0 — активен (−V);
  - 1 — пассивен (+V).

- Бит 0, DTRC (Data Terminal Ready Control) — управление выходом DTR; в режиме диагностики поступает на вход MSR.5:
  - 0 — активен (−V);
  - 1 — пассивен (+V).

19.6. Порт 3FDH (BasePort+5): статус линии

Регистр статуса линии LSR (точнее статус состояния приемопередатчика) содержит следующие биты.

- Бит 7, FIFOE (FIFO Error Status) — ошибка принятых данных в режиме FIFO (буфер содержит хотя бы один символ, принятый с ошибкой формата, паритета или обрывом). В не-FIFO-режиме всегда 0.
- Бит 6, TEMPT (Transmitter Empty Status) — регистр передатчика пуст (нет данных для передачи ни в сдвиговом регистре, ни в буферных регистрах THR или FIFO). Если установлен в 1, то данных для обработки нет.
- Бит 5, THRE (Transmitter Holding Register Empty) — регистр передатчика готов принять байт для передачи. В режиме FIFO указывает на отсутствие символов в FIFO-буфере передачи. Может являться источником прерывания. В обычном режиме если 1, то можно начинать передачу.
- Бит 4, BD (Break Detected) — индикатор обрыва линии (вход приемника находится в состоянии 0 не менее чем время посылки символа).
- Бит 3, FE (Framing Error) — ошибка кадра (неверный стоп-бит).
- Бит 2, PE (Parity Error) — ошибка контрольного бита (паритета или фиксированного).
- Бит 1, OE (Overrun Error) — переполнение (потеря символа). Если прием очередного символа начинается до того, как предыдущий выгружен из сдвигающего регистра в буферный регистр или в регистр FIFO, предыдущий символ в сдвигающем регистре теряется и этот бит выставляется в 1.
- Бит 0, DR (Receiver Data Ready) — принятые данные готовы (в DHR или FIFO-буфере). Сброс — чтением приемника.

Индикаторы ошибок (биты [4:1]) сбрасываются после чтения регистра LSR. В режиме FIFO признаки ошибок хранятся в FIFO-буфере вместе с каждым символом. В регистре они устанавливаются (и вызывают прерывание) в тот момент, когда символ, принятый с ошибкой, находится на вершине FIFO (первый в очереди на считывание). В случае обрыва линии в FIFO заносится только один "обрывной" символ, а UART ждет восстановления и последующего старт-бита.

19.7. Порт 3FEH (BasePort+6): статус модема

Регистр статуса модема MSR (Modem Status Register).

- Бит 7, DCD (Data Carrier Detect) — состояние линии DCD.
- Бит 6, RI (Ring Indicator) — состояние линии RI.
- Бит 5, DSR (Data Set Ready) — состояние линии DSR.
- Бит 4, CTS (Clear To Send) — состояние линии CTS.
- Бит 3, DDCD (Delta Data Carrier Detect) — изменилось состояние линии DCD.
- Бит 2, TERI (Trailing Edge Of Ring Indicator) — спад огибающей (окончание звонка).
- Бит 1, DDSR (Delta Data Set Ready) — изменение состояния DSR.
- Бит 0, DCTS (Delta Clear To Send) — изменение состояния CTS.

Биты [7:4] принимают значение 0 при активном состоянии линии (−V) и значение 1 при пассивном состоянии (+V). Биты изменения [3:0] сбрасываются при чтении регистра.

19.8. Порт 3FFH (BasePort+7): заглушка

Порт 3FFH не используется и может отсутствовать в некоторых реализациях.

19.9. Скорость передачи

Как описано выше, скорость передачи задается в портах 3F8H и 3F9H при выставленном бите DLAB.

Вычисление делителя производится по формуле:

Делитель = (1/16) × *Частота_синхронизации* / *Скорость*_бит/с.

Для IBM PC частота синхронизации составляет 1,8432 МГц, и, следовательно, формула имеет вид:

Делитель = 115 200 / *Скорость*_бит/с.

Стандартные скорости и их делители приведены в табл. 19.2.

Таблица 19.2. *Стандартные скорости и значения делителя*

Бит/с	Делитель	3F8H	3F9H	Бит/с	Делитель	3F8H	3F9H
110	1047	04	17	9600	12	00	0C
150	768	03	00	14 400	8	00	08
300	384	01	80	19 200	6	00	06
600	192	00	C0	28 800	4	00	04
1200	96	00	60	38 400	3	00	03
2400	48	00	30	57 600	2	00	02
4800	24	00	18	115 200	1	00	01

19.10. Порты контроллера прерываний

Каждый коммуникационный порт может генерировать аппаратные прерывания. Так, порт COM1 генерирует прерывания IRQ4, а COM2 — прерывания IRQ3.

Для каждого аппаратного прерывания предусмотрен вектор прерывания. Для IRQ4 это вектор 0CH, а для IRQ3 — вектор 0BH.

Для того чтобы программа могла установить конкретный вектор прерывания, необходимо сначала запретить вызов соответствующих аппаратных прерываний. Для этого требуется задать маску вызова аппаратных прерываний с помощью записи в порт 21H. Формат порта разрешения аппаратных прерываний показан ниже.

Порт 0021H (чтение/запись). *Регистр разрешения аппаратных прерываний*:

- Бит 7: 0 задает разрешение прерываний принтера.
- Бит 6: 0 задает разрешение прерываний дисковода.
- Бит 5: 0 задает разрешение прерываний жесткого диска.
- Бит 4: 0 задает разрешение прерываний COM1.
- Бит 3: 0 задает разрешение прерываний COM2.
- Бит 2: 0 задает разрешение прерываний видео.
- Бит 1: 0 задает разрешение прерываний клавиатуры, мыши и таймера реального времени (RTC, Real Time Clock).
- Бит 0: 0 задает разрешение прерываний таймера.

Таким образом, для разрешения или запрещения прерываний от COM1 используется маска 10H, а для COM2 — маска 08H.

После самостоятельной обработки аппаратного прерывания необходимо послать контроллеру прерываний сигнал "прерывание обработано" — `EOI` (End Of Interrupt). Для этого в порт 20H необходимо послать байт следующего формата.

- Биты [7:5] равны 011, обозначая сигнал `EOI`.
- Биты [4:3] зарезервированы и равны 0.
- Биты [2:0] задают номер IRQ от 0 до 7.

Номера, векторы прерываний и их адреса показаны в табл. 19.3.

Таблица 19.3. *Векторы прерываний, их адреса и IRQ*

Номер	Вектор	Описание	IRQ
00020H	Int 08H	Таймер, канал 0	IRQ0
00024H	Int 09H	Клавиатура	IRQ1
00028H	Int 0AH	Каскад со вторым ПКП (AT)	IRQ2
0002CH	Int 0BH	RS-232C, COM2	IRQ3
00030H	Int 0CH	RS-232C, COM1	IRQ4
00034H	Int 0DH	Жесткий диск PC/XT	IRQ5
00038H	Int 0EH	Контроллер НГМД	IRQ6
0003CH	Int 0FH	Параллельный принтер	IRQ7

Глава 20

Структуры Windows для работы с COM-портами

20.1. Структура настроек порта *COMMCONFIG*[1]

Структура `COMMCONFIG` описывает настройки коммуникационного порта. Описание структуры на языке C имеет следующий вид:

```
typedef struct _COMM_CONFIG {
 DWORD   dwSize;
 WORD    wVersion;
 WORD    wReserved;
 DCB     dcb;
 DWORD   dwProviderSubType;
 DWORD   dwProviderOffset;
 DWORD   dwProviderSize;
 WCHAR   wcProviderData[1];
} COMMCONFIG, *LPCOMMCONFIG;
```

Описание структуры на языке Delphi имеет вид:

```
PCommConfig = ^TCommConfig;
_COMMCONFIG = record
  dwSize: DWORD;
  wVersion: Word;
  wReserved: Word;
  dcb: TDCB;
  dwProviderSubType: DWORD;
```

[1] Многие приведенные здесь структуры могут использоваться также для конфигурирования параллельных портов и модемов, но мы будем рассматривать их использование только для COM-портов.

```
  dwProviderOffset: DWORD;
  dwProviderSize: DWORD;
  wcProviderData: array[0..0] of WCHAR;
end;
TCommConfig = _COMMCONFIG;
```

Структура COMMCONFIG содержит следующие поля.

- dwSize — должно быть равно размеру структуры.
- wVersion — должно равняться 1.
- wReserved — не используется.
- dcb — это поле задает описание параметров COM-порта в структуре DCB *(см. разд. 20.5)*.
- dwProviderSubType — должно быть равно PST_RS232 (значение 1). Полный список возможных значений приводится в табл. 20.2.
- dwProviderOffser — смещение в байтах от начала структуры до устройство-зависимого блока информации wcProviderData.
- dwProviderSize — размер блока wcProviderData в байтах.
- wcProviderData — устройство-зависимый блок информации. Это поле может быть любого размера или вообще отсутствовать. Поскольку структура COMMCONFIG может быть в дальнейшем расширена, для определения положения данного поля следует использовать dwProviderOffset. Если dwProviderSubType равно PST_RS232 или PST_PARALLELPORT, то данное поле отсутствует. Если dwProviderSubType = PST_MODEM, то данное поле содержит структуру MODEMSETTINGS.

Для установки параметров порта используется функция SetCommConfig. Поскольку ручное заполнение полей структуры может быть затруднительно, рекомендуется предварительно считать текущее значение полей с помощью вызова GetCommConfig, затем изменить значение нужных полей и вызвать SetCommConfig.

Кроме того, с помощью функции CommConfigDialog можно отобразить стандартный диалог Windows для изменения характеристик порта. Эта функция не изменяет характеристики порта, а просто возвращает структуру COMMCONFIG (листинг 20.1).

Листинг 20.1. Пример использования структуры COMMCONFIG

```
Function TComPort.CommDialog : Boolean;
Var ComCfg : TCommConfig;
```

```
Begin
 ZeroMemory(@ComCfg, SizeOf(TCommConfig));
 ComCfg.dwSize:= SizeOf(TCommConfig);
 ComCfg.dcb   := FComProp.DCB;
 Result:= CommConfigDialog(PChar(GetComName(False)), 0, ComCfg);
 If Result then begin
  FComProp.DCB:= ComCfg.dcb;
  ApplyComSettings;
 End;
End;
```

20.2. Структура *COMMPROP*

Структура `COMMPROP` используется функцией `GetCommProperties` для получения информации о коммуникационном драйвере. Описание структуры на языке C имеет следующий вид:

```
typedef struct {
  WORD  wPacketLength;
  WORD  wPacketVersion;
  DWORD dwServiceMask;
  DWORD dwReserved1;
  DWORD dwMaxTxQueue;
  DWORD dwMaxRxQueue;
  DWORD dwMaxBaud;
  DWORD dwProvSubType;
  DWORD dwProvCapabilities;
  DWORD dwSettableParams;
  DWORD dwSettableBaud;
  WORD  wSettableData;
  WORD  wSettableStopParity;
  DWORD dwCurrentTxQueue;
  DWORD dwCurrentRxQueue;
  DWORD dwProvSpec1;
  DWORD dwProvSpec2;
  WCHAR wcProvChar[1];
} COMMPROP;
```

Описание структуры на языке Delphi имеет вид:

```
PCommProp = ^TCommProp;
_COMMPROP = record
```

```
  wPacketLength: Word;
  wPacketVersion: Word;
  dwServiceMask: DWORD;
  dwReserved1: DWORD;
  dwMaxTxQueue: DWORD;
  dwMaxRxQueue: DWORD;
  dwMaxBaud: DWORD;
  dwProvSubType: DWORD;
  dwProvCapabilities: DWORD;
  dwSettableParams: DWORD;
  dwSettableBaud: DWORD;
  wSettableData: Word;
  wSettableStopParity: Word;
  dwCurrentTxQueue: DWORD;
  dwCurrentRxQueue: DWORD;
  dwProvSpec1: DWORD;
  dwProvSpec2: DWORD;
  wcProvChar: array[0..0] of WCHAR;
end;
TCommProp = _COMMPROP;
```

Структура `COMMPROP` содержит следующие поля.

- `wPacketLength` — размер пакета (при чтении COM-порта обычно 64).
- `wPacketVersion` — версия структуры (для COM-порта обычно 2).
- `dwServiceMask` равно `SP_SERIALCOMM` (значение 1) для COM-портов.
- `dwReserved1` — зарезервировано и не используется.
- `dwMaxTxQueue` — максимальный размер внутреннего выходного буфера в байтах. Если 0, то ограничение не используется. Для обычного режима COM-портов вернет 0.
- `dwMaxRxQueue` — максимальный размер внутреннего входного буфера в байтах. Если 0, то ограничение не используется. Для обычного режима COM-порта вернет 0.
- `dwMaxBaud` — максимально доступное значение скорости порта в бит/с. Это поле может принимать одно из значений, приведенных в табл. 20.1. Для COM-порта в обычном режиме вернет `BAUD_USER`.
- `dwProvSubType` — режим использования порта. Для COM-портов чаще всего равно `PST_RS232`. В общем случае может вернуть одно из значений, приведенных в табл. 20.2.

- `dwProvCapabilities` — определяет параметры совместимости порта. Возможные значения этого поля приведены в табл. 20.3.
- `dwSettableParams` — определяет параметры, которые могут быть изменены. Возможные значения этого поля приведены в табл. 20.4. Обычный COM-порт возвращает маску 127 ($7F).
- `dwSettableBaud` — определяет маску допустимых скоростей обмена. Маска состоит из тех же констант, которые использовались в `dwMaxBaud`. Обычно COM-порт возвращает маску 0x1006FFFF.
- `wSettableData` — определяет маску допустимых форматов байта, приведенных в табл. 20.5. Обычный COM-порт возвращает значение маски 0x0F.
- `wSettableStopParity` — определяет допустимые значения для числа стоп-бит и четности согласно битовой маске, представленной в табл. 20.6. Обычный COM-порт возвращает значение маски $1F07.
- `dwCurrentTxQueue` — размер внутреннего выходного буфера. Значение 0 свидетельствует о том, что эта величина недоступна.
- `dwCurrentRxQueue` — размер внутреннего входного буфера. Значение 0 говорит о том, что эта величина недоступна.
- `dwProvSpec1`, `dwProvSpec2`, `wcProvChar` — величины, зависящие от драйвера. Программы должны игнорировать эти поля.

Таблица 20.1. *Константы допустимых скоростей в* `COMMPROP`

Значение	Скорость, бит/с	Величина
BAUD_075	75	$00000001
BAUD_110	110	$00000002
BAUD_134_5	134.5	$00000004
BAUD_150	150	$00000008
BAUD_300	300	$00000010
BAUD_600	600	$00000020
BAUD_1200	1200	$00000040
BAUD_1800	1800	$00000080
BAUD_2400	2400	$00000100
BAUD_4800	4800	$00000200
BAUD_7200	7200	$00000400
BAUD_9600	9600	$00000800
BAUD_14400	14 400	$00001000

Таблица 20.1 *(окончание)*

Значение	Скорость, бит/с	Величина
BAUD_19200	19 200	$00002000
BAUD_38400	38 400	$00004000
BAUD_56K	56 000	$00008000
BAUD_57600	57 600	$00040000
BAUD_115200	115 200	$00020000
BAUD_128K	128 000	$00010000
BAUD_USER	Устанавливается программно	$10000000

Таблица 20.2. *Типы режимов работы порта*

Значение	Режим	Величина
PST_FAX	Факс	$021
PST_LAT	Протокол LAT	$101
PST_MODEM	Модем	$006
PST_NETWORK_BRIDGE	Неопределенное сетевое устройство	$100
PST_PARALLELPORT	Параллельный порт	$002
PST_RS232	Порт RS-232	$001
PST_RS422	Порт RS-422	$003
PST_RS423	Порт RS-423	$004
PST_RS449	Порт RS-449	$005
PST_SCANNER	Сканер	$022
PST_TCPIP_TELNET	Протоколы TCP/IP и Telnet	$102
PST_UNSPECIFIED	Не определено	$000
PST_X25	Стандарт X.25	$103

Таблица 20.3. *Параметры совместимости порта*

Значение	Тип	Маска
PCF_16BITMODE	Поддерживается специальный 16-битовый режим	$200

Таблица 20.3 *(окончание)*

Значение	Тип	Маска
PCF_DTRDSR	Поддерживается DTR /DSR	$001
PCF_INTTIMEOUTS	Поддерживается внутренний тайм-аут	$080
PCF_PARITY_CHECK	Поддерживается контроль четности	$008
PCF_RLSD	Поддерживается RLSD	$004
PCF_RTSCTS	Поддерживается RTS/CTS	$002
PCF_SETXCHAR	Поддерживается управление XON/XOFF	$020
PCF_SPECIALCHARS	Поддержка специального символа	$100
PCF_TOTALTIMEOUTS	Поддерживается полный тайм-аут	$040
PCF_XONXOFF	Поддерживается контроль XON/XOFF	$010

Таблица 20.4. *Параметры, доступные для изменения*

Значение	Параметр	Маска
SP_BAUD	Скорость передачи	$02
SP_DATABITS	Число бит данных	$04
SP_HANDSHAKING	Контроль XON/XOFF	$10
SP_PARITY	Четность	$01
SP_PARITY_CHECK	Контроль четности	$20
SP_RLSD	Сигнал RLSD	$40
SP_STOPBITS	Число стоп-бит	$08

Таблица 20.5. *Допустимые форматы данных*

Значение	Бит в байте	Маска
DATABITS_5	5 бит данных	$01
DATABITS_6	6 бит данных	$02
DATABITS_7	7 бит данных	$04

Таблица 20.5 *(окончание)*

Значение	Бит в байте	Маска
DATABITS_8	8 бит данных	$08
DATABITS_16	16 бит данных	$10
DATABITS_16X	Специальное аппаратное расширение	$20

Таблица 20.6. *Допустимые значения числа стоп-бит и четности*

Значение	Параметр	Маска
STOPBITS_10	1 стоп-бит	$0001
STOPBITS_15	1,5 стоп-бита	$0002
STOPBITS_20	2 стоп-бита	$004
PARITY_NONE	Нет четности	$0100
PARITY_ODD	Нечетность	$0200
PARITY_EVEN	Четность	$0400
PARITY_MARK	Постоянная 1	$0800
PARITY_SPACE	Постоянный 0	$1000

Поле `wcProvChar` имеет переменную длину и может содержать или не содержать данные, поэтому для выделения нужного блока памяти необходимо выполнить следующие шаги (листинг 20.2).

1. Выделить память под структуру `COMMPROP`.
2. Запросить информацию у системы, вызвав функцию `GetCommProperties`.
3. Если поле `wPacketLength` содержит значение, большее, чем `sizeof(COMMPROP)`, то имеется дополнительная информация. Для ее получения измените размер ранее выделенного блока памяти. Новый размер должен быть равен значению, занесенному системой в поле `wPacketLength`. Установите в поле `wProvSpec1` значение `COMMPROP_INITIALIZED`. Это будет означать, что выделен достаточный блок памяти для получения дополнительной информации. Повторно вызовите функцию `GetCommProperties`.

Впрочем, для обычного коммуникационного порта эта процедура не требуется, т. к. дополнительная информация для него отсутствует. Чаще всего дополнительная информация представлена в виде структуры `MODEMDEVCAPS`, которая размещается на месте поля `wcProvChar`, если поле `dwProvSubType` содержит значение `PST_MODEM`.

Листинг 20.2. Пример использования структуры COMMPROP

```
{Получение максимально доступной скорости обмена}
function TCommPort.GetMaxBaud: TBaudRate;
var Prop : TCommProp;
begin
 Result:= br110;
 {Если нет соединения - ошибка }
 if IsConnected then begin
  GetCommProperties(Handle, Prop);
  case Prop.dwMaxBaud of
   BAUD_300   : Result:= br300;
   BAUD_600   : Result:= br600;
   BAUD_1200,
   BAUD_1800  : Result:= br1200;
   BAUD_2400  : Result:= br2400;
   BAUD_4800,
   BAUD_7200  : Result:= br4800;
   BAUD_9600  : Result:= br9600;
   BAUD_14400 : Result:= br14400;
   BAUD_19200 : Result:= br19200;
   BAUD_38400 : Result:= br38400;
   BAUD_56K   : Result:= br56000;
   BAUD_57600 : Result:= br57600;
   BAUD_115200: Result:= br115200;
   BAUD_128K  : Result:= br128000;
   BAUD_USER  : Result:= brCustom;
  end;
 end else begin
  { Error! }
 end;
end;
```

20.3. Структура тайм-аутов *COMMTIMEOUTS*

Структура `COMMTIMEOUTS` используется функциями `SetCommTimeouts` и `GetCommTimeouts` для установки и получения значений тайм-аутов коммуникационного порта. Значения этих тайм-аутов определяют поведение функ-

ций ReadFile, WriteFile, ReadFileEx и WriteFileEx при операциях с коммуникационными устройствами. Описание структуры на языке C имеет следующий вид:

```
typedef struct COMMTIMEOUTS {
  DWORD ReadIntervalTimeout;
  DWORD ReadTotalTimeoutMultiplier;
  DWORD ReadTotalTimeoutConstant;
  DWORD WriteTotalTimeoutMultiplier;
  DWORD WriteTotalTimeoutConstant;
} COMMTIMEOUTS, *LPCOMMTIMEOUTS;
```

Описание структуры на языке Delphi имеет вид:

```
PCommTimeouts = ^TCommTimeouts;
_COMMTIMEOUTS = record
  ReadIntervalTimeout: DWORD;
  ReadTotalTimeoutMultiplier: DWORD;
  ReadTotalTimeoutConstant: DWORD;
  WriteTotalTimeoutMultiplier: DWORD;
  WriteTotalTimeoutConstant: DWORD;
end;
TCommTimeouts = _COMMTIMEOUTS;
```

Структура COMMTIMEOUTS имеет следующие поля.

- ReadIntervalTimeout — задает допустимый интервал ожидания в миллисекундах между получением двух символов. Если время ожидания первого символа или время между получением двух символов превышено, функция ReadFile завершает работу. Значение 0 указывает на отсутствие тайм-аута. Значение MAXDWORD, вместе с нулевыми значениями полей ReadTotalTimeoutConstant и ReadTotalTimeoutMultiplier, означает немедленный возврат из операции чтения с передачей уже принятого символа, даже если ни одного символа не было получено.
- ReadTotalTimeoutMultiplier — задает множитель в миллисекундах, используемый для вычисления полного времени ожидания операции чтения. Для каждой операции чтения это значение умножается на количество запрошенных байтов.
- ReadTotalTimeoutConstant — задает константу, добавляемую к произведению, полученному при вычислении времени из ReadTotalTimeoutMultiplier.
- Поля WriteTotalTimeoutMultiplier и WriteTotalTimeoutConstant аналогичны полям ReadTotalTimeoutMultiplier и ReadTotalTimeoutConstant

соответственно, но служат для передачи. Задание нулевого значения всех констант `ReadXXX` указывает на отсутствие тайм-аутов чтения, а всех `WriteXXX` — отключение тайм-аутов записи.

Пусть, например, заданы такие тайм-ауты:

- `ReadTotalTimeoutMultiplier = 2`;
- `ReadTotalTimeoutConstant = 1`;
- `ReadIntervalTimeout = 1`.

Рассмотрим случай считывания 250 символов. Если операция чтения завершится за $250 \times 2 + 1 = 501$ мс, то будет считано все сообщение. Если операция чтения не завершится за 501 мс, то она все равно будет завершена. При этом будут возвращены символы, прием которых завершился до истечения тайм-аута операции. Остальные символы могут быть получены следующей операцией чтения. Операция чтения будет завершена также в случае, если между началом двух последовательных символов пройдет более 1 мс.

Если поля `ReadIntervalTimeout` и `ReadTotalTimeoutMultiplier` установлены в `MAXDWORD`, а `ReadTotalTimeoutConstant` больше нуля и меньше `MAXDWORD`, то выполнение операции чтения подчиняется следующим правилам:

- если в буфере есть символы, то чтение немедленно завершается и возвращается символ из буфера;
- если в буфере нет символов, то операция чтения будет ожидать появления любого символа, после чего она немедленно завершится;
- если в течение времени, заданного полем `ReadTotalTimeoutConstant`, не будет принято ни одного символа, операция чтения завершится по тайм-ауту.

Для получения текущих значений тайм-аутов используется функция `GetCommTimeouts`, а для установки — функция `SetCommTimeouts` (листинг 20.3). С помощью вызова `BuildCommDCBAndTimeouts` можно заполнить структуру `COMMTIMEOUTS` из строки, формат которой совпадает с форматом команды `Mode`.

Листинг 20.3. Пример использования структуры `COMMTIMEOUTS`

```
var
  CommTimeOut : TCommTimeouts;
FillChar(CommTimeOut, SizeOf(TCommTimeOuts), 0);
CommTimeOut.ReadIntervalTimeout := MAXDWORD;
SetCommTimeOuts(SaveHandle, CommTimeOut);
```

20.4. Структура статуса порта *COMSTAT*

Структура `COMSTAT` содержит информацию о коммуникационном устройстве. Эта структура заполняется с помощью вызова `ClearCommError`. Описание структуры на языке C имеет следующий вид:

```
typedef struct _COMSTAT {
  DWORD fCtsHold  : 1;
  DWORD fDsrHold  : 1;
  DWORD fRlsdHold : 1;
  DWORD fXoffHold : 1;
  DWORD fXoffSent : 1;
  DWORD fEof      : 1;
  DWORD fTxim     : 1;
  DWORD fReserved : 25;
  DWORD cbInQue;
  DWORD cbOutQue;
} COMSTAT, *LPCOMSTAT;
```

Описание структуры на языке Delphi имеет вид:

```
_COMSTAT = record
  Flags: TComStateFlags;
  Reserved: array[0..2] of Byte;
  cbInQue: DWORD;
  cbOutQue: DWORD;
end;
TComStat = _COMSTAT;
```

Структура `COMSTAT` имеет следующие поля.

- `fCtsHold` — показывает, что производится ожидание сигнала CTS для начала передачи. Если это поле `True`, передатчик находится в режиме ожидания.
- `fDsrHold` — показывает, что передатчик находится в режиме ожидания DSR.
- `fRlsdHold` — показывает, что передатчик находится в режиме ожидания RLSD.
- `fXoffHold` — показывает, что передатчик находится в режиме ожидания, т. к. был получен символ `XOFF`.
- `fXoffSend` — показывает, что передатчик находится в режиме ожидания, т. к. символ `XOFF` был передан. Передатчик ждет посылки символа `XON` и игнорирует другие символы.
- `fEof` — показывает, что был получен символ `EOF`.

- `fTxim` — показывает, что передатчик находится в режиме передачи символа функцией `TransmitCommChar`.
- `fReserved` — не используется.
- `cbInQue` — число байт, полученных коммуникационным драйвером, но еще не прочитанных с помощью `ReadFile`.
- `cbOutQue` — число байт, переданных функциями передачи, но еще не отправленных. Это поле будет нулевое для синхронного режима передачи.

Чаще всего используют поле `cbInQue`. Это поле показывает число байт в приемном буфере. Структура `COMSTAT` может быть считана вызовом функции `ClearCommError`, которая сбрасывает флаги ошибки, если таковые были, но записывает текущее состояние порта в структуру `COMSTAT` (листинг 20.4).

Листинг 20.4. Пример использования структуры COMSTAT

```
// Возвращает число байт в приемном буфере
function TCommPort.CountRX: DWORD;
var Stat : TCOMSTAT; Errs: DWORD;
begin
 Result:= 65535; // ошибка
 if not Connected then Exit; // порт не открыт
 // Получение числа байт в буфере
 ClearCommError(FHandle, Errs, @Stat );
 Result:= Stat.cbInQue;
end;
```

20.5. Структура *DCB*

Структура `DCB` (Device Control Block, блок управления данными) содержит управляющие настройки коммуникационного устройства. Описание структуры на языке C имеет следующий вид:

```
typedef struct _DCB {
  DWORD DCBlength;
  DWORD BaudRate;
  DWORD fBinary: 1;
  DWORD fParity: 1;
  DWORD fOutxCtsFlow:1;
  DWORD fOutxDsrFlow:1;
  DWORD fDtrControl:2;
  DWORD fDsrSensitivity:1;
```

```
  DWORD fTXContinueOnXoff:1;
  DWORD fOutX: 1;
  DWORD fInX: 1;
  DWORD fErrorChar: 1;
  DWORD fNull: 1;
  DWORD fRtsControl:2;
  DWORD fAbortOnError:1;
  DWORD fDummy2:17;
  WORD  wReserved;
  WORD  XonLim;
  WORD  XoffLim;
  BYTE  ByteSize;
  BYTE  Parity;
  BYTE  StopBits;
  char  XonChar;
  char  XoffChar;
  char  ErrorChar;
  char  EofChar;
  char  EvtChar;
  WORD  wReserved1;
} DCB;
```

Описание структуры на языке Delphi имеет вид:

```
_DCB = packed record
  DCBlength : DWORD;
  BaudRate  : DWORD;
  Flags     : Longint;
  WReserved : Word;
  XonLim    : Word;
  XoffLim   : Word;
  ByteSize  : Byte;
  Parity    : Byte;
  StopBits  : Byte;
  XonChar   : CHAR;
  XoffChar  : CHAR;
  ErrorChar : CHAR;
  EofChar   : CHAR;
  EvtChar   : CHAR;
```

```
  wReserved1: Word;
end;
TDCB = _DCB;
```

Структура DCB имеет следующие поля.

- DCBlength — содержит длину структуры. См. пример в *разд. 21.6.*
- BaudRate — задает скорость обмена. Может задаваться либо числом (в бит/с), либо одной из констант табл. 20.7.
- fBinary — должно быть True, т. к. Windows поддерживает только бинарный метод доступа.
- fParity — включает режим контроля четности. Если поле True, выполняется контроль четности.
- fOutxCtsFlow — включает управление выводом с помощью сигнала CTS. Если это поле True и CTS сброшен, то вывод приостанавливается до установки сигнала CTS.
- fOutxDsrFlow — включает управление выводом с помощью сигнала DSR. Если это поле True и DSR сброшен, то вывод приостанавливается до установки сигнала DSR.
- fDtrControl — может принимать одно из значений.
  - DTR_CONTROL_DISABLE — запрещает использование сигнала DTR.
  - DTR_CONTROL_ENABLE — разрешает использование сигнала DTR. Конкретное значение сигнала можно задавать через вызов EscapeCommFunction.
  - DTR_CONTROL_HANDSHAKE — автоматическое управление DTR. Вызов функции EscapeCommFunction вызовет ошибку. Этот режим используется, например, модемами при восстановлении потерянной связи.
- fDsrSensitivity — включает восприятие DSR сигнала. Если это поле True, то порт игнорирует все принимаемые данные, пока не будет установлен сигнал DSR.
- fTXContinueOnXoff — задает, прекращается ли передача при переполнении приемного буфера и передаче драйвером символа XoffChar.
  - Если это поле равно True, то передача продолжается, несмотря на то, что приемный буфер содержит более XoffLim символов и близок к переполнению, а драйвер передал символ XoffChar для приостановления потока принимаемых данных.
  - Если поле равно False, то передача не будет продолжена до тех пор, пока в приемном буфере не останется меньше XonLim символов и драйвер не передаст символ XonChar для возобновления потока принимае-

мых данных. Таким образом, это поле вводит некую зависимость между управлением входным и выходным потоками информации.

- `fOutX` — включает режим `XON/XOFF`-передачи. Если это поле `True`, передача останавливается по получении символа `XoffChar` и возобновляется при получении символа `XonChar`.
- `fInX` — включает режим `XON/XOFF`-приема. Если это поле `True`, символ `XoffChar` посылается при заполнении входного буфера до предела `XoffLim` байт, а символ `XonChar` посылается, когда буфер опустошен до размера `XonLim`.
- `fErrorChar` — разрешает замещение ошибочных символов. Если это поле `TRUE` и поле `fParity = True`, то ошибочные символы будут заменены символом `ErrorChar`.
- `fNull` — включает игнорирование нулевых байтов при приеме. Если это поле `True`, нулевые байты будут игнорироваться при получении.
- `fRtsControl` — задает режим управления линией RTS. Может принимать одно из значений.
  - `RTS_CONTROL_DISABLE` — запрещает использование линии RTS.
  - `RTS_CONTROL_ENABLE` — разрешает использование линии RTS. Конкретное значение RTS может быть задано с помощью вызова функции `EscapeCommFunction`.
  - `RTS_CONTROL_HANDSHAKE` — включает автоматическое управление линией RTS. Драйвер устанавливает RTS в 1, когда входной буфер менее чем на половину пуст, и в 0, когда буфер заполнен более чем на три четверти. Вызов `EscapeCommFunction` вызовет ошибку.
  - `RTS_CONTROL_TOGGLE` (только для Windows NT/2000/XP) — задает режим управления, когда RTS будет в 1, если есть байты для передачи. RTS сбросится в 0 после передачи всех байтов буфера. Вызов функции `EscapeCommFunction` приведет к ошибке. Этот режим не работает в Windows 95/98/ME.
- `fAbortOnError` — прекращает операции чтения/записи при возникновении ошибок. Драйвер порта не будет воспринимать любые операции чтения/записи до тех пор, пока не будет вызвана функция `ClearCommError`.
- `fDummy2` и `wReserved` — не используются. `wReserved` должно быть установлено в 0.
- `XonLim` задает — минимальное число символов в приемном буфере перед посылкой символа `XON`.
- `XoffLim` — определяет максимальное количество байтов в приемном буфере перед посылкой символа `XOFF`. Максимально допустимое количест-

во байтов в буфере вычисляется вычитанием данного значения из размера приемного буфера в байтах.

- `ByteSize` — задает число бит в байте. Принимает значения от 5 до 8.
- `Parity` — задает проверку паритета:
  - `NOPARITY` — нет проверки, бит четности отсутствует;
  - `ODDPARITY` — проверка нечетности, бит дополняет до нечетности;
  - `EVENPARITY` — проверка четности, бит дополняет до четности;
  - `MARKPARITY` — фиксировано 1, бит четности всегда 1;
  - `SPACEPARITY` — фиксировано 0, бит четности всегда 0.
- `StopBits` — задает число стоп-бит:
  - `ONESTOPBIT` — один стоп-бит;
  - `ONE5STOPBITS` — 1,5 стоп-бита (только для 5- и 6-битовых данных);
  - `TWOSTOPBITS` — 2 стоп-бита.
- `XonChar`, `XoffChar` — задают начальный и конечный символы для режимов `fTXContinueOnXoff = True`, или `fInX = True`, или `fOutX = True`.
- `ErrorChar` — задает символ-замену при условии `fErrorChar = True`.
- `EofChar` — задает символ конца данных.
- `EvtChar` — задает символ для вызова события.
- `wReserved1` — не используется.

Таблица 20.7. *Скорости обмена*

Константа	Значение	Величина
CBR_110	110	110
CBR_300	300	300
CBR_600	600	600
CBR_1200	1200	1200
CBR_2400	2400	2400
CBR_4800	4800	4800
CBR_9600	9600	9600
CBR_14400	14 400	14 400
CBR_19200	19 200	19 200
CBR_38400	38 400	38 400
CBR_5600	56 000	56 000

Таблица 20.7 *(окончание)*

Константа	Значение	Величина
CBR_57600	57 600	57 600
CBR_115200	115 200	$1C200
CBR_128000	128 000	$1F400
CBR_256000	256 000	$3E800

Важно отметить, что программист должен сам следить за совместимостью параметров блока `DCB`. Например:

- `ByteSize` может задаваться только от 5 до 8;
- комбинация `ByteSize = 5` и 2 стоп-бита не допустима, так же, как 1,5 стоп-бита для 6-, 7- и 8-битовых данных.

Для записи структуры `DCB` применяется функция `SetCommState`, а для чтения — функция `GetCommState`.

Если заполнение многочисленных полей этой структуры вызывает затруднение, можно воспользоваться функцией `BuildCommDCB` (или функцией `BuildCommDCBAndTimeouts`), заполняющей ее поля, согласно строке настроек. Формат строки совпадает с форматом команды `Mode`. Также можно сначала прочитать текущее состояние с помощью `GetCommState`, изменить нужные поля структуры, а затем выполнить запись с помощью `SetCommState` (листинг 20.5).

Листинг 20.5. Пример использования структуры DCB

```
var
 DCB: TDCB;
GetCommState(FComPort.FCommThread.ComHandle, DCB);
DCB.Flags:= FFlags;
SetCommState(FComPort.FCommThread.ComHandle, DCB);
```

В Delphi, как видно из описания структуры, битовые поля заменены на одно поле `Flags`. Для задания конкретных бит флагов используются константы, описанные в листинге 20.6.

Листинг 20.6. Константы для поля `Flags` структуры `DCB` (Delphi)

```
Const
  dcb_Binary              = $00000001;
```

```
  dcb_ParityCheck         = $00000002;
  dcb_OutxCtsFlow         = $00000004;
  dcb_OutxDsrFlow         = $00000008;
  dcb_DtrControlMask      = $00000030;
  dcb_DtrControlDisable   = $00000000;
  dcb_DtrControlEnable    = $00000010;
  dcb_DtrControlHandshake = $00000020;
  dcb_DsrSensivity        = $00000040;
  dcb_TXContinueOnXoff    = $00000080;
  dcb_OutX                = $00000100;
  dcb_InX                 = $00000200;
  dcb_ErrorChar           = $00000400;
  dcb_NullStrip           = $00000800;
  dcb_RtsControlMask      = $00003000;
  dcb_RtsControlDisable   = $00000000;
  dcb_RtsControlEnable    = $00001000;
  dcb_RtsControlHandshake = $00002000;
  dcb_RtsControlToggle    = $00003000;
  dcb_AbortOnError        = $00004000;
  dcb_Reserveds           = $FFFF8000;
```

Глава 21

Функции Windows для работы с COM-портами

Важно

В Windows для работы с коммуникационными портами используются те же функции, что и для работы с файлами. В нашей книге для таких функций мы опишем параметры, относящиеся только к работе с портами. Описание остальных параметров вы можете найти, например, в MSDN.

21.1. Функции *CreateFile* и *CloseHandle*: открытие и закрытие порта

Функция `CreateFile` открывает порт, а функция `CloseHandle` — закрывает. Если процесс попытается открыть ресурс, уже открытый другим процессом, функция `CreateFile` вернет ошибку. Дескриптор, полученный после вызова `CreateFile`, должен быть закрыт после использования вызовом `CloseHandle`.

Формат заголовков `CreateFile` и `CloseHandle` на языке C имеет следующий вид:

```
HANDLE CreateFile(
 LPCTSTR lpFileName,             // имя файла (порта)
 DWORD   dwDesiredAccess,        // способ доступа к файлу
 DWORD   dwShareMode,            // тип совместного доступа
 LP_SECURITY_ATTRIBUTES lpSA,    // атрибуты защиты
 DWORD   dwCreationDisposition,  // параметры создания файла
 DWORD   dwFlagsAndAttributes,   // атрибуты файла
 HANDLE   hTemplateFile          // дескриптор template-файла
);

BOOL CloseHandle(
 HANDLE  hObject               // дескриптор порта
);
```

Формат заголовков CreateFile и CloseHandle на языке Delphi имеет следующий вид:

```
function CreateFile(
 lpFileName: PChar;           // имя файла (порта)
 dwDesiredAccess,             // способ доступа к файлу
 dwShareMode: DWORD;          // тип совместного доступа
 lpSA : PSecurityAttributes;  // атрибуты защиты
 dwCreationDisposition,       // параметры создания файла
 dwFlagsAndAttributes: DWORD; // атрибуты файла
 hTemplateFile: Thandle       // дескриптор template-файла
): THandle;
function CloseHandle(
 hObject: THandle             // дескриптор порта
): BOOL;
```

Для использования функции CreateFile в приложении к коммуникационному порту должны быть выполнены несколько условий.

- В качестве имени порта должна передаваться строка вида \\.\COM1, \\.\COM2 и т. д. Для портов с номерами COM1—COM9 могут использоваться простые имена COM1—COM9 без префиксов, а для COM10 и далее необходимо указание префикса.
- Способ доступа должен быть задан явно и установлен в значение GENERIC_READ (порт используется только для чтения данных), или GENERIC_WRITE (порт используется только для записи данных), или GENERIC_READ or GENERIC_WRITE (порт используется и для чтения, и для записи данных).
- Параметр dwShareMode устанавливается в значение 0, означающее "общий доступ к ресурсу запрещен", т. к. коммуникационные порты нельзя делать разделяемыми ресурсами.
- Атрибуты защиты не используются и устанавливаются в nil.
- Параметр создания файла dwCreationDisposition устанавливается в значение OPEN_EXISTING, указывая открыть ресурс, если он существует, или вернуть ошибку, если не существует. Другие значения этого параметра не допускаются.
- Атрибуты для порта должны быть установлены в значение FILE_ATTRIBUTE_NORMAL для синхронного и в значение FILE_ATTRIBUTE_NORMAL or FILE_FLAG_OVERLAPPED для асинхронного доступа к порту;
- Последний параметр обязательно должен иметь значение nil.

21.1.1. Дополнительные сведения

Для нотации языка C строка, передаваемая в функцию `CreateFile`, будет выглядеть как `\\\\.\\COM1`.

Функция `CreateFile` может применяться для открытия файла драйвера, например, мы использовали ее для загрузки драйвера прямого доступа к портам `giveIO` *(см. разд. 9.3.1)*.

В Windows NT и выше в качестве имени порта можно использовать любое DOS-имя, связанное с устройством *(см. разд. 13.4)*.

21.1.2. Возвращаемое значение

Если функция `CreateFile` выполнена успешно, возвращается дескриптор открытого ресурса. Этот дескриптор используется для доступа к открытому ресурсу. Если при открытии ресурса произошла ошибка, функция возвращает значение `INVALID_HANDLE_VALUE`, а подробности можно узнать, вызвав функцию `GetLastError`.

После использования дескриптор должен быть освобожден вызовом функции `CloseHandle`.

Функция `CloseHandle` возвращает ненулевое значение, если закрытие порта выполнено, и возвращает 0, если произошла ошибка. Расширенную информацию об ошибке можно получить с помощью вызова `GetLastError`.

21.1.3. Пример вызова

Листинг 21.1 показывает структуру программы, использующей функции `CreateFile`/`CloseHandle` для доступа к коммуникационному порту COM1.

Листинг 21.1. Использование функций `CreateFile` и `CloseHandle`

```
{Переменная для хранения дескриптора порта}
var
 ComHandle : THandle;

{Открыть порт}
ComHandle:= CreateFile('\\.\COM1',
             GENERIC_READ or GENERIC_WRITE,
             0,
             nil,
             OPEN_EXISTING,
```

```
            FILE_ATTRIBUTE_NORMAL or FILE_FLAG_OVERLAPPED,
            0
    );

{Проверить результат}
if ComHandle = INVALID_HANDLE_VALUE then begin
  {Ошибка открытия порта, функция GetLastError вернет код ошибки}
  Exit;
end;
{... порт открыт успешно ...}
{... использование порта через дескриптор ComHandle  ...}
{Закрытие порта}
CloseHandle(ComHandle);
```

21.2. Функция *ReadFile*: чтение данных из порта

Функция `ReadFile` производит синхронное или асинхронное чтение данных из файла (порта).

Формат заголовка `ReadFile` на языке C имеет следующий вид:

```
BOOL ReadFile(
  HANDLE         hHandle,      // дескриптор, полученный от CreateFile
  LPVOID         lpBuffer,     // буфер для чтения
  DWORD          nNBTR,        // число байт для чтения
  LPDWORD        nNBR,         // реально прочитанное число байт
  LPOVERLAPPED   lpOverlapped  // параметры асинхронного чтения
);
```

Формат заголовка `ReadFile` на языке Delphi имеет следующий вид:

```
function ReadFile(
  hFile: THandle;                 // дескриптор, полученный от CreateFile
  var Buffer;                     // буфер для чтения
  nNumberOfBytesToRead: DWORD;    // число байт для чтения
  var lpNumberOfBytesRead: DWORD;// реально прочитанное число байт
  lpOverlapped: Poverlapped       // параметры асинхронного чтения
): BOOL;
```

Первый параметр передает дескриптор порта, полученный с помощью функции `CreateFile`. Указатель на буфер для чтения данных задается с по-

мощью второго параметра, а размер этого буфера в байтах — с помощью третьего.

Параметр nNBR передает указатель на переменную типа DWORD, в которую возвращено реально прочитанное число байт. В Windows NT/2000/XP этот параметр не может быть NULL, если lpOverlapped равно NULL и может быть NULL, если параметр lpOverlapped ненулевой. В Windows 95/98/ME этот параметр не может быть нулевым. Для получения количества байт, прочитанных в асинхронном режиме, может использоваться функция GetOverlappedResult.

Последний параметр передает настройки для асинхронного чтения данных. Если порт был открыт с параметром FILE_FLAG_OVERLAPPED, этот параметр обязательно должен указывать на правильную структуру типа OVERLAPPED, если же порт был открыт без использования FILE_FLAG_OVERLAPPED, то этот указатель обязательно должен быть NULL.

21.2.1. Дополнительные сведения

В Windows NT/2000/XP можно использовать функцию ReadFileEx для асинхронного чтения данных *(см. разд. 21.4)*.

Так как ReadFile требует распределения буфера перед чтением данных, то можно воспользоваться функцией ClearCommError для получения числа доступных для чтения байтов.

21.2.2. Возвращаемое значение

Функция завершается, если прочитано необходимое количество байтов или произошла ошибка. Если чтение прошло успешно, возвращается ненулевое значение.

В случае ошибки возвращается 0, а код ошибки можно получить с помощью вызова GetLastError.

Перед чтением необходимо установить тайм-ауты ожидания с помощью вызова SetCommTimeouts.

21.2.3. Пример вызова

Листинг 21.2 показывает пример использования функции ReadFile для синхронного чтения данных. Перед чтением данных вызывается функция ClearCommError для получения числа символов, доступных для чтения.

Листинг 21.2. Пример использования функции ReadFile

```
var
 ComHandle : THandle;
```

```
 CurrentState : TComStat;
 CodeError : Cardinal;
 PData : Pointer;
 AvaibleBytes, RealRead : Cardinal;
Begin
 ComHandle:= CreateFile(...);
... ... ... ... ... ... ...
 {Возвращает структуру состояния порта и код ошибок}
 ClearCommError(ComHandle, CodeError, @CurrentState);
 { Число полученных, но еще не прочитанных байт }
 AvaibleBytes:= CurrentState.cbInQue;
 { Проверка числа доступных байт }
 If AvaibleBytes > 0 then begin
  GetMem(PData, AvaibleBytes);
  If ReadFile(ComHandle, PData^, AvaibleBytes, RealRead, nil) then begin
   {Реально прочитано RealRead байт}
  End;
  FreeMem(PData);
 End;
... ... ... ... ... ... ...
 CloseHandle(ComHandle);
end;
```

21.3. Функция *WriteFile*: передача данных

Функция `WriteFile` производит синхронную или асинхронную запись данных в файл (порт).

Формат заголовка `WriteFile` на языке C имеет следующий вид:

```
BOOL WriteFile(
  HANDLE       hHandle,     // дескриптор, полученный от CreateFile
  LPCVOID      lpBuffer,    // буфер данных
  DWORD        nNBTW,       // длина буфера
  LPDWORD      nNBW,        // реально отправленное число байт
  LPOVERLAPPED lpOverlapped // параметры асинхронной записи
);
```

Формат заголовка WriteFile на языке Delphi имеет следующий вид:

```
function WriteFile(
  hFile       : THandle;    // дескриптор, полученный от CreateFile
  const Buffer;             // буфер данных
  nNBTW       : DWORD;      // длина буфера
  var lpNBW   : DWORD;      // реально отправленное число байт
  lpOverlapped: POverlapped // параметры асинхронной записи
): BOOL;
```

Первый параметр передает дескриптор порта, полученный с помощью функции CreateFile. Указатель на буфер данных для записи задается с помощью второго параметра, а размер этого буфера в байтах — с помощью третьего.

Параметр nNBW задает указатель на переменную типа DWORD, в которую будет записано реально переданное число байтов. В Windows 95/98/ME этот параметр не может быть нулевым. В Windows NT/2000/XP этот параметр может быть нулевым, если задан указатель на параметры асинхронной записи lpOverlapped, и не может быть нулевым, если задается синхронная запись, т. е. указатель lpOverlapped нулевой.

21.3.1. Дополнительные сведения

В Windows NT/2000/XP можно использовать функцию WriteFileEx для асинхронной записи данных *(см. разд. 21.5)*.

21.3.2. Возвращаемое значение

Если выполнение успешно, функция WriteFile возвращает ненулевое значение. Если функция завершилась с ошибкой, она возвращает нулевое значение, а код ошибки можно узнать с помощью вызова GetLastError.

21.3.3. Пример вызова

Листинг 21.3 показывает пример использования функции WriteFile для синхронной записи данных в коммуникационный порт, а листинг 21.4 — пример асинхронной записи.

Листинг 21.3. Пример использования функции WriteFile (синхронная запись)

```
var
  FComPortHandle   : THANDLE;
  DataPtr  : Pointer;
  nToWrite, nWrite : Integer;
```

```
FComPortHandle:= CreateFile(...);
nToWrite:= количество передаваемых байт
GetMem(DataPtr, nToWrite);
... заполняем буфер данными ...
WriteFile(FComPortHandle, DataPtr^, nToWrite, nWrite, nil);
FreeMem(DataPtr);
CloseHandle(FComPortHandle);
```

Листинг 21.4. Пример использования функции `WriteFile` (асинхронная запись)

```
var
  FComHandle : THandle;
  AsyncPtr   : PAsync;
  BytesTrans : DWORD;

FComHandle:= CreateFile(...,
               FILE_ATTRIBUTE_NORMAL or FILE_FLAG_OVERLAPPED,
               0
             );
{Создание асинхронных параметров}
New(AsyncPtr);
With AsyncPtr^ do begin
   Kind := 0; { 0 – write, 1 – read }
   GetMem(Data, Count);
   Move(Buffer, Data^, Count);
   Size := Count;
end;
{Передача данных}
WriteFile(FComHandle, Buffer, Count, BytesTrans, @AsyncPtr^.Overlapped);
{Освобождение памяти}
Dispose(AsyncPtr);
CloseHandle(FComHandle);
```

21.4. Функция *ReadFileEx*: APC-чтение данных

Функция `ReadFileEx` осуществляет асинхронное чтение данных. Работа этой функции похожа на вызов обычной функции `ReadFile` в режиме асинхрон-

ного чтения, но ReadFileEx позволяет программе выполнять другие действия во время чтения данных. При завершении чтения будет вызвана специальная callback-процедура.

В Windows 95/98/ME эта функция не может быть использована для чтения данных из COM-порта.

Формат заголовка ReadFileEx на языке C имеет следующий вид:

```
BOOL ReadFileEx(
  HANDLE    hFile,           // дескриптор порта
  LPVOID    lpBuffer,        // буфер для данных
  DWORD     dwBufLen,        // число байт для чтения
  LPOVERLAPPED lpOverlapped, // параметры асинх. чтения
  // callback-процедура, выполняемая по завершении чтения
  LP_OVERLAPPED_COMPLETION_ROUTINE lpComletionRoutine
);
```

Формат заголовка ReadFileEx на языке Delphi имеет следующий вид:

```
function ReadFileEx(
  hFile               : THandle;     // дескриптор порта
  lpBuffer            : Pointer;     // буфер для данных
  nNumberOfBytesToRead: DWORD;       // число байт для чтения
  lpOverlapped        : POverlapped; // параметры асинх. чтения
  // callback-процедура, выполняемая по завершении чтения
  lpCompletionRoutine : TPROverlappedCompletionRoutine
): BOOL;
```

Все параметры, кроме последнего, совпадают с параметрами функции ReadFile. Последний параметр задает адрес процедуры, которая будет выполнена по завершении чтения данных. Эта процедура должна иметь тип TPROverlappedCompletionRoutine, описываемый следующим образом:

```
VOID CALLBACK FileIOCompletionRoutine(
 DWORD        dwErrorCode,                // код ошибки
 DWORD        dwNumberOfBytesTransfered,  // число прочитанных байт
 LPOVERLAPPED lpOverlapped                // асинхронная структура
);
```

В Delphi тип этой процедуры описан как обычный указатель, без спецификации параметров:

```
type
 TPROverlappedCompletionRoutine = TFarProc;
```

На самом деле формат заголовка этой процедуры в Delphi должен быть такой, как показан в листинге 21.5.

Параметры `callback`-процедуры имеют следующий смысл:

- параметр `dwErrorCode` принимает значение 0, если операция успешна;
- параметр `dwNumberOfBytesTransfered` равен числу прочитанных байтов или 0, если функция завершена с ошибкой;
- параметр `lpOverlapped` передает структуру асинхронного чтения.

21.4.1. Возвращаемое значение

При успешном завершении функция `ReadFileEx` возвращает ненулевое значение, а при ошибочном — ноль, при этом код ошибки можно получить с помощью вызова `GetLastError`.

21.4.2. Дополнительные сведения

Для завершения всех асинхронных операций может использоваться функция `CancelIo`.

Функция `ReadFileEx` игнорирует параметр `hEvent` в структуре `lpOverlapped`, и он может использоваться программой.

21.4.3. Пример вызова

Листинг 21.5 показывает пример использования `ReadFileEx`. Обратите внимание, что процедура, вызываемая при завершении операции, должна иметь спецификатор `stdcall`.

Листинг 21.5. Пример использования функции ReadFileEx

```
Procedure TReadThread.Execute;
Var ReadOL : TOverLapped; {структура для асинхронного чтения}

  {Callback-процедура, вызывающаяся при получении байта}
  Procedure OnCompletionRead(
   dwErrorCode, dwNumberOfBytesTransfered :Cardinal;
   var lpOverlapped : TOverlapped
  ); stdcall;
  begin
  end;
Begin
 With FOwner do
```

```
  While (not Terminated) and Connected do begin {пока порт открыт}
   {Запуск операции асинхронного чтения}
   ReadFileEx(FHandle, @FByte, 1, @ReadOL, @OnCompletionRead);
   {Ожидание завершения операции}
   SleepEx(INFINITE, True);
   {Сюда мы попадем, только когда байт будет принят}
   Synchronize(DoReadByte);
  End;
End;
```

21.5. Функция *WriteFileEx*: APC-передача данных

Функция `WriteFileEx` производит асинхронную запись данных. Работа этой функции похожа на вызов функции `WriteFile` в режиме асинхронной записи, но `WriteFileEx` позволяет программе выполнять другие действия во время записи (передачи) данных. При завершении записи будет вызвана специальная `callback`-процедура.

В Windows 95/98/ME эта функция не может быть использована для чтения данных из COM-порта.

Формат заголовка `WriteFileEx` на языке C имеет следующий вид:

```
BOOL WriteFileEx(
  HANDLE    hFile,           // дескриптор порта
  LPCVOID   lpBuffer,        // буфер для данных
  DWORD     dwBufLen,        // число байт для записи
  LPOVERLAPPED lpOverlapped, // параметры асинх. записи
  // callback-процедура, выполняемая по завершении записи
  LP_OVERLAPPED_COMPLETION_ROUTINE lpComletionRoutine
);
```

Формат заголовка `WriteFileEx` на языке Delphi имеет следующий вид:

```
function WriteFileEx(
  hFile    : THandle;              // дескриптор порта
  lpBuffer : Pointer;              // буфер для данных
  nNumberOfBytesToWrite: DWORD;    // число байт для записи
  const lpOverlapped : TOverlapped; // параметры асинх. записи
  lpCompletionRoutine: FARPROC
): BOOL;
```

Все параметры этой функции совпадают с параметрами функций WriteFile и ReadFileEx.

21.5.1. Возвращаемое значение

При успешном завершении функция WriteFileEx возвращает ненулевое значение, а при ошибочном — ноль. Код ошибки можно получить с помощью вызова GetLastError.

21.5.2. Пример вызова

Листинг 21.6 показывает пример использования WriteFileEx. Обратите внимание на спецификатор stdcall функции обратного вызова OnCompletionWrite.

Листинг 21.6. Пример использования функции WriteFileEx

```
Function TComPort.WriteByte(const B : Byte) : Boolean;
Var WriteOL : TOverLapped; {структура для асинхронной записи}
  {Callback-процедура, вызываемая после завершения передачи}
  Procedure OnCompletionWrite(
   dwErrorCode, dwNumberOfBytesTransfered : Cardinal;
   var lpOverlapped : TOverlapped
  ); stdcall;
  begin
   MessageBeep(0);
  end;

Begin
 Result:= False;
 {создание события для асинхронной записи}
 FillChar(WriteOL, SizeOf(WriteOL), 0);
 WriteOL.hEvent:= CreateEvent(nil, True, True, nil);
 {асинхронная отправка байта}
 WriteFileEx(FHandle, @B, 1, WriteOL, @OnCompletionWrite);
 SleepEx(INFINITE, True);
 {освобождение дескриптора события}
 CloseHandle(WriteOL.hEvent);
End;
```

21.6. Функция *BuildCommDCB*: создание структуры *DCB* из строки

Функция BuildCommDCB помогает заполнить многочисленные поля структуры DCB согласно строке, имеющей формат команды mode.

Формат заголовка BuildCommDCB на языке C имеет следующий вид:

```
BOOL BuildCommDCB(
  LPCTSTR  lpDef,  // строка
  LPDCB    lpDCB   // указатель на блок параметров DCB
);
```

Формат заголовка BuildCommDCB на языке Delphi имеет следующий вид:

```
function BuildCommDCB(
  lpDef: PChar;     // строка
  var lpDCB: TDCB   // указатель на блок параметров DCB
): BOOL;
```

Структура DCB заполняется согласно параметрам, описанным в строке описания lpDef. Строка описания должна иметь формат как для команды Mode, например,

```
baud=1200 parity=N data=8 stop=1
COM1: baud=1200 parity=N data=8 stop=1
```

Обычно функция BuldCommDCB изменяет только явно перечисленные в строке lpDef поля. Однако существуют два исключения из этого правила.

1. При задании скорости обмена 110 бит в секунду автоматически устанавливается формат обмена с двумя стоп-битами. Это сделано для совместимости с командой mode из MS-DOS или Windows NT.
2. По умолчанию запрещается программное (XON/XOFF) и аппаратное управление потоком. Необходимо вручную заполнить требуемые поля DCB, если требуется управление потоком.

Функция BuilCommDCB поддерживает как новый, так и старый форматы командной строки mode. Однако нельзя смешивать эти форматы в одной строке. Новый формат строки позволяет явно задавать значения для полей DCB, отвечающих за управление потоком. При использовании старого формата необходимо учитывать следующие соглашения:

- для строк вида 9600,n,8,1 (не заканчивающихся символами "x" или "p"):
  - fInX, fOutX, fOutXDsrFlow, fOutXCtsFlow устанавливаются в False;
  - fDtrControl устанавливается в DTR_CONTROL_ENABLE;
  - fRtsControl устанавливается в RTS_CONTROL_ENABLE;

- для строк вида `9600,n,8,1,x` (заканчивающихся символом "`x`"):
  - `fInX`, `fOutX` устанавливаются в `True`;
  - `fOutXDsrFlow`, `fOutXCtsFlow` устанавливаются в `False`;
  - `fDtrControl` устанавливается в `DTR_CONTROL_ENABLE`;
  - `fRtsControl` устанавливается в `RTS_CONTROL_ENABLE`;
- для строк вида `9600,n,8,1,p` (заканчивающихся символом "`p`"):
  - `fInX`, `fOutX` устанавливаются в `False`;
  - `fOutXDsrFlow`, `fOutXCtsFlow` устанавливаются `True`;
  - `fDtrControl` устанавливается в `DTR_CONTROL_HANDSHAKE`;
  - `fRtsControl` устанавливается в `RTS_CONTROL_HANDSHAKE`.

Важно отметить, что функция `BuildCommDCB` только заполняет поля `DCB` указанными значениями. Это подготовительный шаг к конфигурированию порта, но не само конфигурирование, которое выполняется функцией `SetCommState`. Поэтому вы можете вызвать `BuildCommDCB` для общего заполнения структуры `DCB`, затем изменить значения нужных полей и после этого вызывать функцию конфигурирования порта.

21.6.1. Дополнительные сведения

В Windows 95/98 из-за ошибки в реализации невозможно передать в качестве первого параметра константную строку — строку необходимо скопировать во временный буфер. При использовании константы возникает ошибка `Access Violation`.

Для заполнения структуры `DCB` можно использовать также функцию `BuildCommDCBAndTimeouts`. Еще одним простым способом заполнения `DCB` является вызов `GetCommState`, возвращающий текущее значение полей `DCB`, изменение нужных полей и вызов `SetCommState` для установки новых значений.

21.6.2. Возвращаемое значение

Если функция `BuildCommDCB` выполнена успешно, возвращается ненулевое значение, иначе — нулевое. Код ошибки можно получить с помощью вызова `GetLastError`.

21.6.3. Пример вызова

Листинг 21.7 показывает один из способов простого заполнения полей структуры `DCB` с помощью вызова `BuildCommDCB`.

Листинг 21.7. Пример использования функции BuildCommDCB

```
Var
  DCB : TDCB;
begin
 FillMemory(@dcb, SizeOf(dcb), 0);
 dcb.DCBlength:= SizeOf(dcb);
 if not BuildCommDCB('9600,n,8,1', dcb) then begin
  // DCB не создана.
  Exit;
 end;
 // структура DCB готова к использованию
end;
```

21.7. Функция *BuildCommDCBAndTimeouts*: создание структуры *DCB* и тайм-аутов из строки

Функция BuildCommDCBAndTimeouts позволяет заполнять структуры DCB и COMMTIMEOUTS одновременно.

Формат заголовка BuildCommDCBAndTimeouts на языке C имеет следующий вид:

```
BOOL BuildCommDCBAndTimeouts(
  LPCTSTR        lpDef,          // строка
  LPDCB          lpDCB,          // указатель на блок параметров
  LPCOMMTIMEOUTS lpCommTimeOuts  // структура тайм-аутов порта
);
```

Формат заголовка BuildCommDCBAndTimeouts на языке Delphi имеет следующий вид:

```
function BuildCommDCBAndTimeouts(
  lpDef: PChar;                      // строка
  var lpDCB: TDCB;                   // указатель на блок параметров
  var lpCommTimeouts: TCommTimeouts  // структура тайм-аутов порта
): BOOL;
```

Функция аналогична BuidCommDCB, но, кроме заполнения DCB, позволяет заполнять поля структуры COMMTIMEOUTS из подстроки TO = xxx:

- при TO = ON функция включает использование тайм-аутов чтения и записи;

- при TO = OFF функция выключает использование тайм-аутов;
- если параметр TO = xxx не указан, функция не изменяет значений в структуре COMMTIMEOUTS.

21.8. Функции *SetCommBreak* и *ClearCommBreak*: управление выводом данных

Функция SetCommBreak замораживает передачу данных для указанного порта до тех пор, пока не будет вызвана функция ClearCommBreak.

Формат заголовков этих функций на языке C имеет следующий вид:

```
BOOL SetCommBreak(
 HANDLE hFile // дескриптор порта, полученный из CreateFile
);
BOOL ClearCommBreak(
 HANDLE hFile // дескриптор порта, полученный из CreateFile
);
```

Формат заголовков на языке Delphi имеет вид:

```
function SetCommBreak(hFile: THandle): BOOL;
function ClearCommBreak(hFile: THandle): BOOL;
```

Последовательный канал передачи данных можно перевести в специальное состояние, называемое *разрывом связи*. При этом передача данных прекращается, а выходная линия переводится в состояние 0. Приемник, обнаружив, что за время, необходимое для передачи стартового бита, битов данных, бита четности и стоповых битов, приемная линия ни разу не перешла в состояние 1, так же фиксирует у себя состояние разрыва.

Следует заметить, что состояние разрыва линии устанавливается аппаратно. Поэтому нет другого способа возобновить прерванную с помощью SetCommBreak передачу данных, кроме вызова ClearCommBreak.

Единственный параметр этих функций задает дескриптор порта, полученный при вызове функции CreateFile.

21.8.1. Возвращаемое значение

Если выполнение прошло успешно, функция SetCommBreak возвращает ненулевое значение.

21.9. Функция *ClearCommError*: получение и сброс ошибок порта

Функция `ClearCommError` получает информацию об ошибках порта и сбрасывает флаги ошибок.

Формат заголовка `ClearCommError` на языке C имеет вид:

```
BOOL ClearCommError(
  HANDLE    hFile,   // дескриптор порта
  LPDWORD   lpError, // код ошибки
  LPCOMSTAT lpStat   // состояние порта
);
```

Формат заголовка на языке Delphi имеет вид:

```
function ClearCommError(
  hFile        : THandle;  // дескриптор порта
  var  lpErrors: DWORD;    // код ошибки
  lpStat       : PComStat  // состояние порта
): BOOL;
```

Входным параметром является дескриптор порта, полученный с помощью `CreateFile`. Параметр `lpError` возвращает маску, состоящую из кодов ошибок, приведенных в табл. 21.1 (для полноты приведены все коды ошибок).

Таблица 21.1. *Коды ошибок для функции* `ClearCommError`

Маска	Значение	Величина
CE_BREAK	Обрыв линии	$010
CE_DNS	(Windows 95/98/ME) Порт не выбран	$800
CE_FRAME	Ошибка кадра	$008
CE_IOE	Обнаружена ошибка ввода/вывода	$400
CE_MODE	Либо требуемый режим работы не поддерживается, либо параметр hFile задан неверно	$8000
CE_OOP	(Windows 95/98/ME, для параллельного порта). Закончилась бумага	$1000
CE_OVERRUN	Буфер полон. Следующий символ будет потерян	$002
CE_PTO	(Windows 95/98/ME, параллельный порт). Тайм-аут параллельного порта	$200

Таблица 21.1 *(окончание)*

Маска	Значение	Величина
CE_RXOVER	Входной буфер переполнен или символ был принят после получения символа конца EOF	$001
CE_RXPARITY	Ошибка паритета	$004
CE_TXFULL	Приложение пытается послать символ, но передатчик полон	$100

Поле `cbInQue` в структуре `PComStat` может использоваться для получения числа байтов во входной очереди порта. Эта величина необходима перед вызовом функции `ReadFile`, т. к. перед чтением данных необходимо распределение памяти для буфера (листинг 21.2).

21.9.1. Возвращаемое значение

Если выполнение успешно, функция `ClearCommError` возвращает ненулевое значение.

21.10. Функция *Cancello*: прерывание асинхронных операций

Функция `CancelIo` прерывает все асинхронные операции чтения и записи для данного потока. Функция не влияет на операции в других потоках.

Формат заголовка на языке C имеет вид:

```
BOOL CancelIo(
  HANDLE hFile // дескриптор порта
);
```

Формат заголовка на языке Delphi имеет вид:

```
function CancelIo(hFile: THandle): BOOL;
```

Параметр `hFile` задает дескриптор порта, полученный при открытии в функции `CreateFile`.

21.10.1. Возвращаемое значение

Ненулевое значение, если выполнение успешно.

21.11. Функция *EscapeCommFunction*: управление портом

Функция `EscapeCommFunction` позволяет передавать некоторые команды напрямую драйверу.

Формат заголовка на языке C имеет вид:

```
BOOL EscapeCommFunction(
  HANDLE  hFile,  // дескриптор порта
  DWORD   dwFunc  // номер функции
);
```

Формат заголовка на языке Delphi имеет вид:

```
function EscapeCommFunction(hFile: THandle; dwFunc: DWORD): BOOL;
```

Функция, указанная в параметре `dwFunc`, выполняется для порта с дескриптором `hFile`. Указание на выполнение функции передается напрямую драйверу порта. Допустимые значения параметра приведены в табл. 21.2.

Таблица 21.2. *Коды функции* `EscapeCommFunction`

Функция	Описание	Значение
CLRDTR	Сбрасывает сигнал DTR	6
CLRRTS	Сбрасывает сигнал RTS	4
SETDTR	Устанавливает сигнал DTR	5
SETRTS	Устанавливает сигнал RTS	3
SETXOFF	Указывает передатчику выполнить операцию, имитирующую прием символа XOFF	1
SETXON	Указывает передатчику выполнить операцию, имитирующую прием символа XON	2
SETBREAK	Замораживает передачу до выполнения функции `ClearCommBreak` или функции `EscapeCommFunction` с параметром CLRBREAK. Эта функция не сбрасывает данные, которые еще не были переданы	8
CLRBREAK	Восстанавливает передачу данных, выполняя действия, аналогичные вызову `ClearCommBreak`	9
RESETDEV	Сброс устройства, если это возможно	7
GETCOMBASE	Получить базовый адрес порта (недокументированно). Только в Windows 95/98/ME	10

21.11.1. Возвращаемое значение

При успешном выполнении функция EscapeCommFunction возвращает ненулевое значение.

21.12. Функции *GetCommMask* и *SetCommMask*: маска вызова событий

Функция GetCommMask позволяет получить флаги событий, а функция SetCommMask — задать эти флаги. Флаги задают типы событий, которые будут отслеживаться драйвером порта. Ожидание выбранных событий производится с помощью функции WaitCommEvent.

Формат заголовков этих функций на языке C имеет вид:

```
BOOL GetCommMask(
  HANDLE  hFile,    // дескриптор порта
  LPDWORD lpEvtMask // маска событий
);
BOOL SetCommMask(
  HANDLE  hFile,    // дескриптор порта
  DWORD   lpEvtMask // маска событий
);
```

Формат заголовков на языке Delphi имеет вид:

```
function GetCommMask(hFile: THandle; var lpEvtMask: DWORD): BOOL;
function SetCommMask(hFile: THandle; dwEvtMask: DWORD): BOOL;
```

Маска событий состоит из флагов, приведенных в табл. 21.3.

Таблица 21.3. *Маски событий для GetCommMask/SetCommMask*

Значение	Описание
EV_BREAK	Прерывание ввода
EV_CTS	Изменение состояния линии CTS
EV_DSR	Изменение состояния линии DSR
EV_ERR	Одна из ошибок: CE_FRAME, CE_OVERRUN или CE_RXPARITY
EV_RING	Сигнал RI
EV_RLSD	Изменение состояния RLSD
EV_RXCHAR	Принят символ

Таблица 21.3 *(окончание)*

Значение	Описание
EV_RXFLAG	Получен специальный символ, описанный в блоке DCB
EV_TXEMPTY	Буфер посылки опустошен

21.12.1. Возвращаемое значение

Не ноль, если выполнение успешно.

21.12.2. Пример вызова

См. пример в *разд. 21.14*.

21.13. Функция *CreateEvent*: создание объекта ожидания

Функция `CreateEvent` создает или открывает именованный или безымянный объект-событие.

Формат заголовка на языке C имеет вид:

```
HANDLE CreateEvent(
 LP_SECURITY_ATTRIBUTES lpEventSA,  // атрибуты защиты
 BOOL  bManualReset,   // тип управления
 BOOL  bInitialState,  // начальное состояние
 LPCTSTR lpName        // имя объекта
);
```

Формат заголовка на языке Delphi имеет вид:

```
function CreateEvent(
  lpEventAttributes: PSecurityAttributes; // атрибуты защиты
  bManualReset,           // тип управления
  bInitialState: BOOL;    // начальное состояние
  lpName: PChar           // имя объекта
): THandle;
```

Параметр `lpEventSA` задает атрибуты защиты. Если этот параметр `NULL`, берутся параметры родительского процесса.

Параметр bManualReset задает режим управления событием. Если этот параметр True, то программист должен вызвать функцию ResetEvent для переключения события в несигнальное состояние. Если False, то объект автоматически переключается в несигнальное состояние после завершения ожидания.

Параметр bInitialState задает начальное состояние объекта. Если True, то объект создается в сигнальном состоянии, иначе — в несигнальном.

Параметр lpName задает имя объекта. Имя ограничено длиной MAX_PATH символов. Имя объекта регистрозависимо. Если вместо имени указано NULL, то создается безымянный объект. Имя может содержать любые символы, кроме символа "\".

21.13.1. Возвращаемое значение

Если выполнение успешно, функция CreateEvent возвращает дескриптор созданного объекта. Если именованный объект уже существует, возвращается его дескриптор, а функция GetLastError возвращает ERROR_ALREADY_EXISTS. В случае ошибки возвращается NULL.

21.13.2. Пример вызова

См. пример в *разд. 21.14*.

21.14. Функция *WaitCommEvent*: ожидание события COM-порта

Функция WaitCommEvent ожидает наступления события, заданного маской функции SetCommMask.

Формат заголовка на языке C имеет вид:

```
BOOL WaitCommEvent(
  HANDLE        hFile,        // дескриптор порта
  LPDWORD       lpEvtMask,    // маска событий
  LPOVERLAPPED  lpOverlapped  // параметры
);
```

Формат заголовка на языке Delphi имеет вид:

```
function WaitCommEvent(
  hFile         : THandle;     // дескриптор порта
  var lpEvtMask : DWORD;       // маска событий
  lpOverlapped  : POverlapped  // параметры
): BOOL;
```

Если в процессе выполнения WaitCommEvent маска событий будет изменена с помощью SetCommMask, функция незамедлительно завершится, а указатель lpEvtMask будет сброшен в 0.

Функция WaitCommEvent обнаруживает изменение состояния линий, но не сообщает об их текущем состоянии. Для получения текущего состояния линий CTS, DSR, RLSD и индикатора вызова можно использовать функцию GetCommModemStatus.

Параметр hFile задает дескриптор порта, возвращаемый функцией CreateFile. Маска событий состоит из тех же флагов, что и для SetCommMask.

Указатель на OVERLAPPED требуется, если порт был открыт с флагом FILE_FLAG_OVERLAPPED. Если порт открыт с флагом FILE_FLAG_OVERLAPPED, а указатель lpOverlapped равен NULL, функция некорректно завершит работу. В случае, когда порт открыт с помощью FILE_FLAG_OVERLAPPED, структура параметров должна содержать дескрипторы созданных с помощью CreateEvent объектов-событий.

21.14.1. Возвращаемое значение

Если выполнение успешно, функция WaitCommEvent возвращает ненулевое значение.

21.14.2. Дополнительные сведения

Версии Windows, основанные на Win32, не поддерживают событие RING для последовательных портов, задаваемое флагом EV_RING функции SetCommMask. Для обхода этой ситуации предлагается создать поток, обрабатывающий результат функции GetCommModemStatus:

```
While (true) do begin
  GetCommModemStatus(hCommPort, @dwModemStatus);
  if ((MS_RING_ON and dwModemStatus) <> 0) then
   // обнаружен сигнал RING
End;
```

21.14.3. Пример вызова

Листинг 21.8 показывает использование функции WaitCommEvent для организации потока чтения данных TReadThread. При инициализации устанавливается маска ожидания наличия данных EV_RXCHAR. Вызов WaitCommEvent стартует ожидание этого события, а функция WaitForSingleObject позволяет "разбудить" поток в нужный момент. Функция GetOverlappedResult возвращает результат ожидания — маску произошедших событий.

Перед чтением данных производится вызов функции ClearCommError, позволяющей узнать об ошибках порта, и самое главное — получить число символов, доступных для чтения. Это позволяет корректно распределить память для буфера данных функции ReadFile.

Листинг 21.8. Пример использования WaitCommEvent

```
Procedure TReadThread.Execute;
Var CurrentState : TComStat;
    AvaibleBytes, ErrCode, RealRead : Cardinal;
    ReadOL : TOverLapped; {структура для асинхронного чтения}
    Signaled, Mask : DWORD;
    BytesTrans : DWORD; {не используется для WaitCommEvent}
    bReadable : Boolean; {готовность к чтению данных}
Begin
 With FOwner do begin
  Try
    {создание события для асинхронного чтения}
    FillChar(ReadOL, SizeOf(ReadOL), 0);
    ReadOL.hEvent:= CreateEvent(nil, True, True, nil);

    {Маска событий, которые будет отслеживать читающий поток }
    {Пока это только получение символа                        }
    SetCommMask(FHandle, EV_RXCHAR);

    While (not Terminated) and Connected do begin {пока порт открыт}
     { Ждем одного из событий }
     WaitCommEvent(FHandle, Mask, @ReadOL);

     Signaled:= WaitForSingleObject(ReadOL.hEvent, INFINITE);

     If (Signaled  = WAIT_OBJECT_0) then begin
      If GetOverlappedResult(FHandle, ReadOL, BytesTrans,
                                              False) then begin

       {после GetOverlappedResult в переменной mask, которая}
       {передавалась в WaitCommEvent, появятся флаги произошедших}
       {событий, либо 0 в случае ошибки}
       If (Mask and EV_RXCHAR) <> 0 then begin
```

```
        {Получаем состояние порта (линий и модема)}
        ClearCommError(FHandle, ErrCode, @ CurrentState);
        { Число полученных, но еще не прочитанных байт}
        AvaibleBytes:= CurrentState.cbInQue;
        { Проверка числа доступных байт }
        If FWaitFullBuffer then begin
         {ждать только полного буфера }
         bReadable:= AvaibleBytes >= FInBufSize;
        End else begin
         {ждать любого числа байт}
         bReadable:= AvaibleBytes > 0;
        End;

        If bReadable then begin
         {Чистка буфера}
         ZeroMemory(FInBuffer, FInBufSize);
         If ReadFile(FHandle, FInBuffer^,
           Min(FInBufSize, AvaibleBytes), RealRead, @ReadOL) then begin
          {сохраняем параметры вызова события}
          FErrCode:= ErrCode;
          FCount  := RealRead;
          {Вызываем событие OnReadByte. Для синхронизации с VCL}
          {надо вызвать метод Synchronize}
          Synchronize(DoReadPacket);
         End;
        End;
       End;
      End;
     End;
    End;

  Finally
   {закрытие дескриптора сигнального объекта}
   CloseHandle(ReadOL.hEvent);
   {Сброс события и маски ожидания}
   SetCommMask(FHandle, 0);
  End;
 End;
End;
```

21.15. Функции *GetCommConfig* и *SetCommConfig*: конфигурирование параметров порта

Функция `GetCommConfig` возвращает, а функция `SetCommConfig` — устанавливает текущую конфигурацию коммуникационного устройства. Конфигурация передается с помощью структуры `COMMCONFIG` *(см. разд. 20.1)*.

Формат заголовков этих функций на языке C имеет вид:

```
BOOL GetCommConfig(
  HANDLE       hCommDev,  // дескриптор порта
  LPCOMMCONFIG lpCC,      // указатель на COMMCONFIG
  LPDWORD      lpdwSize   // возвращает размер lpCC
);
BOOL SetCommConfig(
  HANDLE       hCommDev, // дескриптор порта
  LPCOMMCONFIG lpCC,     // указатель на COMMCONFIG
  DWORD        dwSize    // передает размер lpCC
);
```

Формат заголовков на языке Delphi имеет вид:

```
function GetCommConfig(
  hCommDev    : THandle;     // дескриптор порта
  var lpCC    : TCommConfig; // указатель на COMMCONFIG
  var lpdwSize: DWORD        // возвращает размер lpCC
): BOOL;
function SetCommConfig(
  hCommDev    : THandle;     // дескриптор порта
  const lpCC  : TCommConfig; // указатель на COMMCONFIG
  dwSize      : DWORD        // возвращает размер lpCC
): BOOL;
```

Первый параметр передает дескриптор порта, полученный с помощью функции `CreateFile`. Второй параметр передает указатель на структуру параметров порта `COMMCONFIG`. Третий параметр либо возвращает размер структуры, либо передает его.

21.15.1. Возвращаемое значение

Возвращает ненулевое значение, если вызов успешен.

21.15.2. Пример вызова

Структура COMMCONGIG имеет переменную длину, поэтому для правильного распределения памяти надо вызвать ее дважды. Первый раз — для определения необходимого размера памяти, а второй — для получения данных (листинг 21.9).

Листинг 21.9. Пример вызова GetCommConfig и SetCommConfig

```
Var
 CommHandle : THandle;
 Buffer     : PCommConfig;
 Size       : DWORD;
begin
 {Открываем порт}
 ComHandle:= CreateFile(...);
 {Создаем временный буфер}
 GetMem(Buffer, SizeOf(TCommConfig));
 {Получаем необходимый размер буфера}
 Size:= 0;
 GetCommConfig(CommHandle, Buffer^, Size);
 {Освободить временный буфер}
 FreeMem(Buffer, SizeOf(TCommConfig));
 {Создаем нужный буфер}
 GetMem(Buffer, Size);
 {Получаем данные}
 GetCommConfig(CommHandle, Buffer^, Size);
 {Устанавливаем скорость обмена}
 Buffer^.dcb.BaudRate:= 1200;
 {Задаем новую конфигурацию}
 SetCommConfig(CommHandle, Buffer^, Size);
 {Освобождаем буфер}
 FreeMem(Buffer, Size);
 {Закрываем порт}
 CloseHandle(CommHandle);
end;
```

21.16. Функция *CommConfigDialog*: диалог конфигурирования порта

Функция `CommConfigDialog` отображает диалог конфигурирования параметров порта (рис. 21.1).

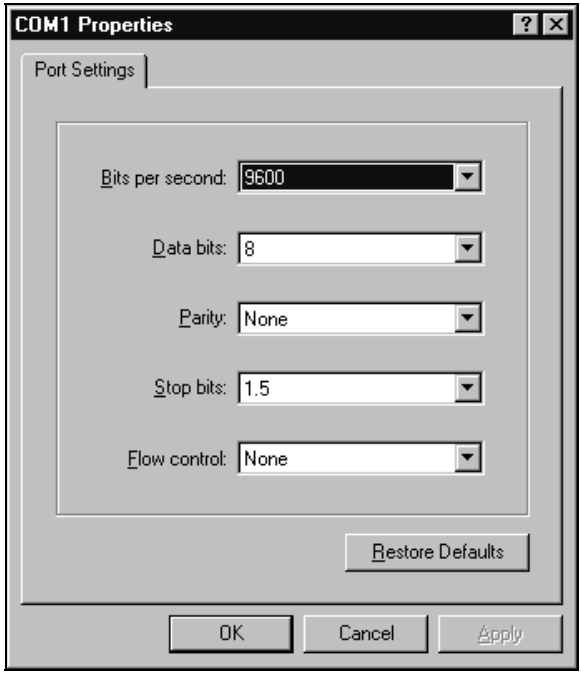

Рис. 21.1. Диалог редактирования настроек порта (`CommConfigDialog`)

Функция `CommConfigDialog` не выполняет настройки порта. Она всего лишь позволяет пользователю изменить некоторые поля в блоке `DCB`, содержащемся в структуре `COMMCONFIG`. Кнопка **Restore Default** (Восстановить исходные значения) устанавливает настройки порта, заданные с помощью функции `SetDefaultCommConfig`. Эти настройки можно прочитать с помощью вызова `GetDefaultCommConfig`.

Формат заголовка на языке C имеет вид:

```
BOOL CommConfigDialog(
  LPCTSTR lpszName, // название порта
  HWND    hWnd,     // дескриптор окна, которому
                    // будет принадлежать диалог
```

```
 LPCOMMCONGIF lpCC  // указатель на структуру COMMCONFIG
);
```

Формат заголовка на языке Delphi имеет вид:

```
function CommConfigDialog(
  lpszName  : PChar;
  hWnd      : HWND;
  var lpCC  : TCommConfig
): BOOL;
```

Первый параметр задает имя порта (или имя устройства). В Windows 95/98/ME имя должно быть указано в короткой форме, т. е. COM1, но не `\\.\COM1`. Второй параметр задает дескриптор окна, которому будет принадлежать диалог (может быть указан 0). Последний параметр передает указатель на структуру `COMMCONFIG`.

21.16.1. Возвращаемое значение

Возвращает ненулевое значение при успешном выполнении.

21.16.2. Дополнительные сведения

Для работы функции `CommConfigDialog` требуется специальный DLL-модуль, предоставляемый разработчиком коммуникационного устройства. Для COM-портов это модуль serialui.dll, для модемов — модуль modemui.dll. Вид диалога зависит от операционной системы и динамической библиотеки, предоставляемой производителем устройства.

21.16.3. Пример вызова

```
Var
  comm : TCOMMCONFIG;
comm.dwSize:= sizeof(comm);
CommConfigDialog('COM1', 0, comm);
```

21.17. Функция *GetCommProperties*: прочитать свойства порта

Функция `GetCommProperties` возвращает структуру `COMMPROP` свойств порта.

Формат заголовка на языке C имеет вид:

```
BOOL GetCommProperties(
  HANDLE hFile,         // дескриптор порта
```

```
  LPCOMMPROP lpCommProp // указатель на структуру COMMPROP
);
```

Формат заголовка на языке Delphi имеет вид:

```
function GetCommProperties(
  hFile          : THandle;
  var lpCommProp : TCommProp
): BOOL;
```

Первый параметр передает дескриптор порта, полученный с помощью `CreateFile`. Второй параметр — указатель на структуру `COMMPROP`.

21.17.1. Возвращаемое значение

Ненулевое значение, если функция выполнена успешно.

21.17.2. Пример вызова

См. листинг 20.2.

21.18. Функции *GetCommState* и *SetCommState*: состояние порта

Функция `GetCommState` читает, а функция `SetCommState` устанавливает текущие настройки порта согласно данным, записанным в структуре `DCB`.

Формат заголовков этих функций на языке C имеет вид:

```
BOOL GetCommState(
  HANDLE hFile, // дескриптор порта
  LPDCB  lpDCB  // указатель на структуру DCB
);
BOOL SetCommState(
  HANDLE hFile, // дескриптор порта
  LPDCB  lpDCB  // указатель на структуру DCB
);
```

Формат заголовков на языке Delphi имеет вид:

```
function GetCommState(hFile: THandle; var   lpDCB: TDCB): BOOL;
function SetCommState(hFile: THandle; const lpDCB: TDCB): BOOL;
```

Первый параметр передает дескриптор порта, полученный с помощью `CreateFile`. Второй параметр передает указатель на структуру `DCB`.

21.18.1. Возвращаемое значение

Возвращает ненулевое значение при успешном выполнении.

21.18.2. Пример вызова

Структура DCB имеет множество полей, заполнение которых вручную — достаточно трудоемкая задача. Значительно проще считать текущее значение DCB с помощью вызова GetCommState, изменить нужные поля структуры и вызвать SetCommState для установки новых параметров порта (листинг 21.10).

Листинг 21.10. Пример вызова GetCommState и SetCommState

```
procedure TComPort.SetRxBuffer(Value: cardinal);
var DCB: TDCB;
begin
 {Сохранить как новое значение свойства RxBuffer}
 FRxBuffer  := Value;
 {Лимит XOFF равен 3/4}
 FXOffLimit := Value div 4 * 3;
 {Лимит XON  равен 1/2}
 FXOnLimit  := Value div 2;
 {Получить текущее значение полей DCB}
 GetCommState(FComHandle, DCB);
 {Установить новые значения}
 DCB.XoffLim:= FXOffLimit;
 DCB.XonLim := FXOnLimit;
 {Отправить установки драйверу}
 SetCommState(FComHandle, DCB);
end;
```

21.19. Функции *GetCommTimeouts* и *SetCommTimeouts*: тайм-ауты порта

Функция GetCommTimeouts читает, а функция SetCommTimeouts устанавливает текущие настройки тайм-аутов согласно данным, записанным в структуре COMMTIMEOUTS.

Формат заголовков этих функций на языке C имеет вид:

```
BOOL SetCommTimeouts(
  HANDLE         hFile,// дескриптор порта
```

```
  LPCOMMTIMEOUTS lpCT  // указатель на COMMTIMEOUTS
);
BOOL GetCommTimeouts(
  HANDLE hFile,// дескриптор порта
  LPCOMMTIMEOUTS lpCT  // указатель на COMMTIMEOUTS
);
```

Формат заголовка на языке Delphi имеет вид:

```
function SetCommTimeouts(
  hFile: THandle;
  const lpCommTimeouts: TcommTimeouts
): BOOL;
```

Первый параметр передает дескриптор порта, полученный с помощью `CreateFile`. Второй параметр передает указатель на структуру `COMMTIMEOUTS`.

21.19.1. Возвращаемое значение

Возвращает ненулевое значение при успешном выполнении.

21.19.2. Пример вызова

См. листинг 20.3.

21.20. Функция *PurgeComm*: сброс буферов порта

Вызов функции `PurgeComm` удаляет все символы из входного и (или) выходного буферов. Также может прерывать незаконченные операции с портом.

Формат заголовка на языке C имеет вид:

```
BOOL PurgeComm(
  HANDLE hFile, // дескриптор порта
  DWORD  dwFlag // флаги сбрасываемых буферов
);
```

Формат заголовка на языке Delphi имеет вид:

```
function PurgeComm(hFile: THandle; dwFlags: DWORD): BOOL;
```

Параметр `hFile` передает дескриптор порта, полученный с помощью `CreateFile`. Параметр `dwFlag` определяет маску операций согласно табл. 21.4.

Таблица 21.4. *Маски операций для* `PurgeComm`

Значение	Операция
PURGE_TXABORT	Завершает все асинхронные операции записи и завершается немедленно, даже если операции еще не завершились
PURGE_RXABORT	Завершает все асинхронные операции чтения и завершается немедленно, даже если операции еще не завершились
PURGE_TXCLEAR	Очищает выходной буфер (если ресурс имеет такой буфер)
PURGE_RXCLEAR	Очищает входной буфер (если ресурс имеет такой буфер)

21.20.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

21.20.2. Пример вызова

Пример, показанный в листинге 21.11, может применяться для сброса входных и выходных буферов, например, перед изменением скорости обмена порта. Сброс буферов перед изменениями конфигурации гарантирует, что в выходной поток не попадут некорректные символы (и, аналогично, не будут приняты из входного буфера).

Листинг 21.11. Пример использования функции PurgeComm

```
function TCommPort.FlushBuffers(bInBuf, bOutBuf: Boolean): Boolean;
var dwAction: DWORD;
begin
 Result:= False;
 // нет соединения — ошибка
 if not Connected then Exit;
 // маска операции
 dwAction:= 0;
 // Сброс выходного буфера
 if bOutBuf then
  dwAction:= dwAction or PURGE_TXABORT or PURGE_TXCLEAR;
 // Сброс входного буфера
 if bInBuf then
```

```
  dwAction:= dwAction or PURGE_RXABORT or PURGE_RXCLEAR;
 // Выполнить сброс
 Result:= PurgeComm(FHandle, dwAction);
end;
```

21.21. Функция *SetupComm*: конфигурирование размеров буферов

Функция SetupComm задает рекомендуемые размеры внутренних буферов — размеры в байтах очередей приема и передачи. Важно отметить, что это только рекомендуемые размеры. В общем случае система сама в состоянии определить требуемый размер очередей. Внутренние очереди драйвера позволяют избежать потери данных, если программа не успевает их считывать, и пауз в работе программы, если программа передает данные слишком быстро. Размер очереди выбирается немного бóльшим максимальной длины сообщения. Указанные размеры очередей будут приняты драйвером к сведению. Но он оставляет за собой право внести коррективы или вообще отвергнуть устанавливаемое значение. В последнем случае функция завершится с ошибкой.

Формат заголовка на языке C имеет вид:

```
BOOL SetupComm(
  // дескриптор порта
  HANDLE hFile,
  // рекомендуемый размер внутреннего входного буфера в байтах
  DWORD  dwInQueue,
  // рекомендуемый размер внутреннего выходного буфера в байтах
  DWORD  dwOutQueue
);
```

Формат заголовка на языке Delphi имеет вид:

```
function SetupComm(
  // дескриптор порта
  hFile: THandle;
  // рекомендуемый размер внутреннего входного буфера в байтах
  dwInQueue,
  // рекомендуемый размер внутреннего выходного буфера в байтах
  dwOutQueue: DWORD
): BOOL;;
```

Параметр `hFile` передает дескриптор порта, полученный с помощью `CreateFile`, `dwInQueue` передает рекомендуемый размер входного буфера, а `dwOutQueue` — выходного буфера.

В Windows NT/2000/XP размеры буферов должны быть четными числами. При попытке задать нечетные числа функция вернет `ERROR_INVALID_DATA`.

21.21.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

21.22. Функции *GetDefaultCommConfig* и *SetDefaultCommConfig*: настройки порта по умолчанию

Функция `GetDefaultCommConfig` читает, а `SetDefaultCommConfig` устанавливает настройки порта по умолчанию. Эти настройки будут приняты при нажатии кнопки **Restore Defaults** (Восстановить исходные значения) в диалоговом окне, вызываемом функцией `CommConfigDialog` *(см. разд. 21.16)*.

Формат заголовков этих функций на языке C имеет вид:

```
BOOL SetDefaultCommConfig(
  LPCTSTR      lpszName, // строковое имя порта
  LPCOMMCONFIG lpCC,     // указатель на COMMCONFIG
  DWORD        dwSize    // размер структуры lpCC
);
BOOL GetDefaultCommConfig(
  LPCTSTR      lpszName,  // строковое имя порта
  LPCOMMCONFIG lpCC,      // указатель на COMMCONFIG
  LPDWORD      lpdwSize   // размер структуры lpCC
);
```

Формат заголовков на языке Delphi имеет вид:

```
function SetDefaultCommConfig(
  lpszName: PChar;       // строковое имя порта
  lpCC    : PCommConfig; // указатель на COMMCONFIG
  dwSize  : DWORD        // размер структуры lpCC
): BOOL;
function GetDefaultCommConfig(
  lpszName    : PChar;       // строковое имя порта
```

```
  var lpCC    : TCommConfig; // указатель на COMMCONFIG
  var lpdwSize: DWORD        // размер структуры lpCC
): BOOL;
```

Первый параметр передает имя порта в строковом виде, как для `CreateFile`. Второй параметр передает указатель на структуру параметров `COMMCONFIG`, а третий — передает (или принимает) размер структуры `COMMCONFIG`.

21.22.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

21.23. Функция *TransmitCommChar*: передача специальных символов

Иногда требуется срочная передача символа, имеющего специальное значение, а в очереди передатчика еще есть данные. Функция `TransmitCommChar` служит для передачи внеочередных символов вне зависимости от наличия данных в очереди передатчика.

При обнаружении символа, переданного этой функцией, выставляется флаг `fTxim` в структуре `COMSTAT`.

Эта функция удобна для посылки символа прерывания обмена (например, <Ctrl>+<C>).

Специальный символ может быть передан только один раз — если символ, переданный с помощью этой функции, еще не отправлен в линию, то повторный вызов функции вернет ошибку.

Формат заголовка на языке C имеет вид:

```
BOOL TransmitCommChar(
  HANDLE  hFile, // дескриптор порта
  char    cChar  // передаваемый символ
);
```

Формат заголовка на языке Delphi имеет вид:

```
function TransmitCommChar(hFile: THandle; cChar: CHAR): BOOL;
```

Первый параметр передает дескриптор порта, полученный с помощью `CreateFile`. Второй параметр передает символ.

21.23.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

21.24. Функция *GetCommModemStatus*: статус модема

Функция `GetCommModemStatus` возвращает состояние сигналов модема.

Формат заголовка на языке C имеет вид:

```
BOOL GetCommModemStatus(
  HANDLE   hFile,        // дескриптор порта
  LPDWORD  lpModemStatus // статус модема
);
```

Формат заголовка на языке Delphi имеет вид:

```
function GetCommModemStatus(
  hFile: THandle;         // дескриптор порта
  var lpModemStat: DWORD  // статус модема
): BOOL;
```

Первый параметр передает дескриптор порта, полученный с помощью `CreateFile`. Второй параметр возвращает статус модема, состоящий из масок, приведенных в табл. 21.5.

Таблица 21.5. *Маски состояния модема для* `GetCommModemStatus`

Маска	Описание	Значение
`MS_CTS_ON`	Сигнал CTS установлен	$0010
`MS_DSR_ON`	Сигнал DSR установлен	$0020
`MS_RING_ON`	Сигнал вызова установлен	$0040
`MS_RLSD_ON`	Сигнал RLSD установлен	$0080

21.24.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

21.24.2. Пример вызова

Код, показанный в листинге 21.12, представляет собой функцию, возвращающую состояние всей линий модема. Вместо маски возвращается обычный для Delphi набор состояний.

Листинг 21.12. Пример вызова GetCommModemStatus

```
type
  {Описание состояний линии}
  TLineStatus = (lsCTS, lsDSR, lsRING, lsCD);
  {Набор состояний линии}
  TLineStatusSet = set of TLineStatus;

{Получить состояние линии}
function TCommPort.GetLineStatus: TLineStatusSet;
var Status : DWORD;
begin
 Result:= [];
 {нет соединения}
 if not Connected then exit;
 { получаем состояние линий модема }
 if not GetCommModemStatus(FHandle, Status) then exit;
 {разбираем полученную маску состояний}
 if Status and MS_CTS_ON  <> 0 then Result:= Result + [lsCTS ];
 if Status and MS_DSR_ON  <> 0 then Result:= Result + [lsDSR ];
 if Status and MS_RING_ON <> 0 then Result:= Result + [lsRING];
 if Status and MS_RLSD_ON <> 0 then Result:= Result + [lsCD  ];
end;
```

21.25. Функция *WaitForSingleObject*: ожидание сигнального состояния объекта

Функция `WaitForSingleObject` ожидает сигнального состояния объекта синхронизации. Функция завершается в двух случаях:

- объект перешел в сигнальное состояние;
- по завершении тайм-аута ожидания.

Формат заголовка на языке C имеет вид:

```
DWORD WaitForSingleObject(
  HANDLE   hHandle,       // дескриптор объекта
  DWORD    dwMilliseconds // пауза ожидания сигнального состояния
);
```

Формат заголовка на языке Delphi имеет вид:

```
function WaitForSingleObject(
  hHandle       : THandle; // дескриптор объекта
  dwMilliseconds: DWORD    // пауза ожидания сигнального состояния
): DWORD;
```

Первый параметр передает дескриптор объекта (порта). Пауза ожидания `dwMilliseconds` задается в миллисекундах. Специальная константа `INFINITE` задает неограниченное время ожидания.

При необходимости ожидания сигнального состояния одного из нескольких объектов или всех объектов одновременно следует воспользоваться функцией `WaitForMultipleObjects`, а не объединять `WaitForSingleObject` с помощью `OR` или `AND`.

21.25.1. Возвращаемое значение

При успешном выполнении функция `WaitForSingleObject` возвращает значение, указывающее на состояние объекта. При ошибке возвращает `WAIT_FAILED`. В табл. 21.6 приводятся возможные результаты функции.

Таблица 21.6. *Результаты функции* `WaitForSingleObject`

Код	Описание
`WAIT_ABANDONED`	Используется для мьютексов. Возвращается в случае, когда объект не был освобожден, хотя поток, его создавший, уже завершен. В нашей книге мы не используем мьютексы
`WAIT_OBJECT_0`	Объект перешел в сигнальное состояние
`WAIT_TIMEOUT`	Тайм-аут завершен, а сигнальное состояние не достигнуто. Естественно, в случае `INFINITE` такой результат невозможен
`WAIT_FAILED`	Ошибка вызова `WaitForSingleObject`

21.26. Функция *WaitForMultipleObjects*: ожидание сигнального состояния объектов

При необходимости ожидания сигнального состояния одного из нескольких объектов одновременно, следует воспользоваться функцией `WaitForMultipleObjects`, а не объединять `WaitForSingleObject` с помощью

оператора OR. Эта функция ожидает сигнального состояния одного или всех вместе объектов синхронизации. Функция завершается в двух случаях:

- один или все объекты перешли в сигнальное состояние;
- по завершении тайм-аута ожидания.

Формат заголовка на языке C имеет вид:

```
DWORD WaitForMultipleObjects(
  DWORD  nCount,             // число ожидаемых объектов
  const  HANDLE * lpHandles, // массив дескрипторов объектов
  BOOL   bWaitAll            // флаг "ожидать все объекты"
  DWORD  dwMilliseconds      // пауза ожидания сигнального состояния
);
```

Формат заголовка на языке Delphi имеет вид:

```
function WaitForMultipleObjects(
  nCount        : DWORD;         // число ожидаемых объектов
  lpHandles     : PWOHandleArray;// массив дескрипторов объектов
  bWaitAll      : BOOL;          // флаг "ожидать все объекты"
  dwMilliseconds: DWORD          // пауза ожидания сигнального состояния
): DWORD;
```

Первый параметр передает число объектов синхронизации, дескрипторы которых передаются во втором параметре. Если третий параметр True, функция будет ожидать сигнального состояния всех объектов, иначе — хотя бы одного из них. Четвертый параметр задает паузу ожидания так же, как в функции WaitForSingleObject. Пауза ожидания задается в миллисекундах. Специальная константа INFINITE задает неограниченное время ожидания.

21.26.1. Возвращаемое значение

При успешном выполнении функция WaitForMultipleObjects возвращает значение, указывающее на состояние объектов. При ошибке возвращает WAIT_FAILED. В табл. 21.7 приводятся возможные результаты функции.

Таблица 21.7. *Результаты функции* `WaitForMultipleObjects`

Код	Описание
От WAIT_ABANDONED_0 до (WAIT_ABANDONED_0+nCount-1)	Если bWaitAll = True, возвращаемое значение показывает, что один из объектов — неосвобожденный мьютекс. Если bWaitAll = False, то результат функции минус константа WAIT_ABANDONED_0 будет равен индексу неосвобожденного мьютекса в массиве lpHandles

Таблица 21.7 *(окончание)*

Код	Описание
От WAIT_OBJECT_0 до (WAIT_OBJECT_0+nCount-1)	Если bWaitAll = True, то все объекты перешли в сигнальное состояние. Если False, то результат функции минус константа WAIT_OBJECT_0 будет равен индексу объекта в массиве lpHandles, первым перешедшего в сигнальное состояние
WAIT_TIMEOUT	Тайм-аут завершен, а сигнальное состояние ни одного объекта не достигнуто. Естественно, в случае INFINITE такой результат невозможен
WAIT_FAILED	Ошибка вызова WaitForSingleObject

21.27. Функция *GetOverlappedResult*: результат асинхронной операции

Функция GetOverlappedResult возвращает результат асинхронной операции с файлом или коммуникационным устройством.

Формат заголовка на языке C имеет вид:

```
BOOL GetOverlappedResult(
  HANDLE       hHandle,     // дескрипторов объекта (файла или порта)
  LPOVERLAPPED lpOverlapped,// структура асинхронного вызова
  LPDWORD      lpNBT,       // число переданных или прочитанных байт
  BOOL         bWait        // флаг ожидания
);
```

Формат заголовка на языке Delphi имеет вид:

```
function GetOverlappedResult(
  hFile             : THandle; // дескрипторов объекта (файла или порта)
  const lpOverlapped: TOverlapped; // структура асинхронного вызова
  var   lpNBT       : DWORD;  // число переданных или прочитанных байт
  bWait             : BOOL    // флаг ожидания
): BOOL;
```

Первый параметр передает дескриптор файла или коммуникационного устройства. Перед вызовом этот дескриптор должен быть связан с асинхронной операцией, используя функции ReadFile, WriteFile, DeviceIoControl или WaitCommEvent.

Второй параметр передает структуру OVERLAPPED, которая использовалась при начале асинхронной операции.

Третий параметр возвращает число байтов, реально прочитанных или записанных в результате операции. Этот параметр не используется для коммуникационных портов.

Если параметр bWait равен True, то функция не завершится, пока операция не будет завершена. Если False и операция продолжается, то функция вернет False, а вызов GetLastError вернет ERROR_IO_INCOMPETE.

В Windows 95/98/ME, если параметр bWait равен True, то поле hEvent в структуре OVERLAPPED не может быть NULL.

21.27.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

21.28. Функция *DeviceIoControl*: прямое управление драйвером

Функция DeviceIoControl посылает команду, задаваемую кодом dwIoCode, напрямую драйверу устройства с дескриптором hDevice, указывая ему выполнить определенные действия.

Для коммуникационных устройств дескриптор можно получить с помощью вызова CreateFile с обязательным использованием флага асинхронных операций FILE_FLAG_OVERLAPPED.

Формат заголовка на языке C имеет вид:

```
BOOL DeviceIoControl(
  HANDLE       hDevice,     // дескриптор устройства
  DWORD        dwIoCode,    // код выполняемой функции
  LPVOID       lpInBuffer,  // указатель на входные данные
  DWORD        dwInBufSize, // размер входного буфера
  LPVOID       lpOutBuffer, // указатель на выходной буфер
  DWORD        nOutBufSize, // размер выходного буфера
  LPDWORD      lpBytesRetn, // требуемый размер выходного буфера
  LPOVERLAPPED lpOverlapped // структура асинхронного вызова
);
```

Формат заголовка на языке Delphi имеет вид:

```
function DeviceIoControl(
  hDevice         : THandle; // дескриптор устройства
```

```
  dwIoControlCode : DWORD;   // код выполняемой функции
  lpInBuffer      : Pointer; // указатель на входные данные
  nInBufferSize   : DWORD;   // размер входного буфера
  lpOutBuffer     : Pointer; // указатель на выходной буфер
  nOutBufferSize  : DWORD;   // размер выходного буфера
  var lpBytesRetn : DWORD;   // требуемый размер выходного буфера
  lpOverlapped: Poverlapped  // структура асинхронного вызова
): BOOL;
```

Параметр `lpInBuffer` передает входные данные, если они необходимы для выполнения операции. Может быть передано `NULL`, если для выполнения операции не требуется дополнительная информация. Размер передаваемых входных данных задается параметром `dwInBufSize`.

Если функция, задаваемая кодом `dwIoCode`, должна возвращать данные, то для этих данных передается указатель на буфер `lpOutBuffer` и его размер `nOutBufSize`. Если функция не возвращает данные, `lpOutBuffer` должно быть установлено в `NULL`, а `nOutBufSize` в 0.

Последний параметр возвращает реальный размер данных, сохраненных в `lpOutBufSize`. Если буфер слишком маленький, функция завершится с ошибкой, а вызов `GetLastError` вернет `ERROR_INSUFFICIENT_BUFFER`. Значение `lpBytesRetn` в этом случае будет равно 0. Некоторые драйверы, если буфер мал для принятия полного пакета данных, возвращают только часть данных. В этом случае `GetLastError` возвращает значение `ERROR_MORE_DATA`, а `lpBytesRetn` равно числу полученных байтов. Приложение должно снова вызвать `DeviceIoControl` с теми же параметрами, указав новый стартовый адрес.

Если `lpOverlapped` равно `NULL`, то `lpByteRetn` не должно быть `NULL`. Даже если функция не возвращает данные и `lpOutBuffer` равно `NULL`, вызов `DeviceIoControl` будет использовать `lpBytesRetn`. Однако после таких операций значение `lpBytesRetn` бессмысленно.

Если `lpOverlapped` не равно `NULL`, то `lpByteRetn` может быть `NULL`. Если этот параметр не `NULL` и операция возвращает данные, то `lpBytesRetn` не имеет смысла, пока вызванная асинхронная операция не завершится. Для получения результата и числа возвращенных байтов используется вызов `GetOverlappedResult`. Если `hDevice` связан с портом ввода/вывода, то получить число возвращенных байтов можно с помощью функции `GetQueuedCompletionStatus`.

Если `lpOverlapped` не `NULL`, то она должна содержать указатель на структуру `OVERLAPPED`. Если `hDevice` был открыт без использования флага `FILE_FLAG_OVERLAPPED`, то этот параметр будет игнорироваться. Иначе операция будет расцениваться как асинхронная. Структура `lpOverlapped` в этом

случае должна содержать правильный дескриптор объекта события, иначе функция завершится с непредсказуемой ошибкой.

Для асинхронных операций функция `DeviceIoControl` завершается сразу же, а сигнальный объект-событие будет сигнализировать о завершении операции. В синхронном режиме функция не завершится, пока не завершится операция, или по ошибке.

Важно отметить, что в Windows 95/98/ME эту функцию можно применять только к дескрипторам виртуальных драйверов. Например, для открытия системного VxD-драйвера надо передавать в функцию `CreateFile` имя `\\.\vwin32`.

Коды функций зависят от конкретного устройства. Для последовательных портов эта функция может применяться, только начиная с Windows 2000. Коды функций для последовательных портов приводятся в *гл. 22*.

21.28.1. Возвращаемое значение

При успешном выполнении функция `DeviceIoControl` возвращает ненулевое значение. В случае ошибки вернет 0, а вызов `GetLastError` вернет код ошибки.

21.29. Функция *EnumPorts*: перечисление портов

Функция `EnumPorts` производит перебор всех портов системы. Если `pName` равно `NULL`, производится перебор локальных портов, иначе — портов указанного сервера (для портов принтера).

Формат заголовка на языке C имеет вид:

```
BOOL EnumPorts(
  LPTSTR  pName,      // имя сервера (для портов принтера)
  DWORD   Level,      // тип получаемой информации
  LPBYTES pPorts,     // буфер информации о порте
  DWORD   cbBuf,      // размер информации в pPorts
  LPDWORD pcbNeeded, // число записанных или требуемых байт
  LPDWORD pcReturned // число перечисленных портов
);
```

Формат заголовка на языке Delphi имеет вид:

```
function EnumPorts(
  pName : PChar;      // имя сервера (для портов принтера)
  Level : DWORD;      // тип получаемой информации
```

```
  pPorts: Pointer;   // буфер информации о порте
  cbBuf : DWORD;     // размер информации в pPorts
  var pcbNeeded,     // число записанных или требуемых байт
  pcReturned : DWORD // число перечисленных портов
): BOOL;
```

Параметр `Level` может принимать два значения: 1 или 2. Если он равен 1, то функция возвращает в буфер `pPorts` набор структур `PORT_INFO_1`, иначе — `PORT_INFO_2`. Эти структуры описываются следующим образом:

```
PORT_INFO_1 = record
 pName       : PWideChar;
end;
PORT_INFO_2 = record
 pPortName   : PWideChar;
 pMonitorName: PWideChar;
 pDescription: PWideChar;
 fPortType   : DWORD;
 Reserved    : DWORD;
end;
```

Параметр `cbBuf` описывает размер блока памяти `pPorts`. Как принято в Windows API, параметр `pcbNeeded` возвращает размер буфера, необходимого для выполнения функции. Если `cbBuf` (и, соответственно, сам буфер `pPorts`) слишком маленький, функция завершится с ошибкой, а `GetLastError` вернет код `ERROR_INSUFFICIENT_BUFFER`. В этом случае `pcbNeeded` будет содержать нужный размер буфера. Если же `cbBuf` будет больше или равен необходимому размеру, то `pcbNeeded` будет содержать число реально записанных в буфере данных.

Параметр `pcReturned` возвращает число структур, записанных в буфер `pPorts`.

Для невиртуальных портов их имена возвращаются со знаком ":", например, "COM1:", а для виртуальных — без него, например, "FILE".

21.29.1. Дополнительные сведения

Для использования функции `EnumPorts` в Delphi, необходимо подключить модуль `WinSpool`.

21.29.2. Возвращаемое значение

При успешном выполнении функция `EnumPorts` возвращает ненулевое значение. В случае ошибки вернет 0, а вызов `GetLastError` вернет код ошибки.

21.29.3. Пример вызова

Хотя в MSDN функция EnumPorts описывается для портов принтера, она также возвращает и коммуникационные порты. Отличить одни порты от других можно, сравнив начало имени порта со строкой COM (листинг 21.13).

Листинг 21.13. Пример вызова функции EnumComPorts

```
{ перечисление имен всех доступных коммуникационных портов}
class procedure TComPort.EnumComPorts(Ports: TStrings);
var
  BytesNeeded, Returned, I: DWORD;
  Success: Boolean;
  PortsPtr: Pointer;
  InfoPtr: PPortInfo1;
  TempStr: string;
begin
 {Запрос размера нужного буфера}
 Success := EnumPorts(nil, 1, nil, 0, BytesNeeded, Returned);

 If (not Success) and (GetLastError = ERROR_INSUFFICIENT_BUFFER) then
begin
  {Отводим нужный блок памяти}
  GetMem(PortsPtr, BytesNeeded);
  Try
   {Получаем список имен портов}
   Success := EnumPorts(nil, 1, PortsPtr, BytesNeeded, BytesNeeded,
Returned);

   {Переписываем имена в StringList, отсеивая не-COM-порты}
   For I := 0 to Returned - 1 do begin
    InfoPtr := PPortInfo1(DWORD(PortsPtr) + I * SizeOf(TPortInfo1));
    TempStr := InfoPtr^.pName;
    If Copy(TempStr, 1, 3) = 'COM' then
     Ports.Add(TempStr);
   End;
  Finally
   {Освобождаем буфер}
```

```
   FreeMem(PortsPtr);
  End;
 End;
end;
```

21.30. Функция *QueryDosDevice*: получение имени устройства по его DOS-имени

Функция QueryDosDevice возвращает информацию о DOS-имени устройства. Функция может получить текущее значение NT-имени для данного DOS-имени или вернуть список всех DOS-имен системы.

Формат заголовка на языке C имеет вид:

```
DWORD QueryDosDevice(
  LPCTSTR pName,     // DOS-имя устройства
  LPTSTR  pResult,   // буфер для результата
  DWORD   dwMax,     // размер буфера pResult
);
```

Формат заголовка на языке Delphi имеет вид:

```
function QueryDosDevice(
  lpDeviceName : PChar; // DOS-имя устройства
  lpTargetPath : PChar; // буфер для результата
  ucchMax      : DWORD  // размер буфера pResult
): DWORD;
```

Первый параметр передает DOS-имя устройства, например, "COM1" или "C:". В этом случае в буфер pResult будет записано внутреннее имя устройства. Если же pName нулевой, то в буфер будут записаны все DOS-имена, зарегистрированные в системе. Каждое имя — строка, завершающаяся нуль-символом. Признак завершения таблицы имен — двойной нуль-символ (т. е. пустая строка).

Второй параметр передает буфер для сохранения результата. Размер буфера передается в параметре dwMax. Если размер буфера мал, то результат выполнения зависит от версии операционной системы (см. ниже).

21.30.1. Возвращаемое значение

При успешном выполнении функция QueryDosDevice возвращает число символов, сохраненных в буфере. В случае ошибки вернет 0, а вызов GetLastError вернет код ошибки.

Если буфер имеет недостаточный размер, функция вернет 0, а вызов `GetLastError` вернет код `ERROR_INSUFFICIENT_BUFFER`. В Windows NT/2000, если буфер мал, функция запишет в буфер столько данных, на сколько хватит буфера.

21.30.2. Пример вызова

Листинг 21.14 показывает получение списка всех имен в системе с помощью вызова `QueryDosDevice`.

Листинг 21.14. Пример вызова функции QueryDosDevice

```
{Получение всех имен устройства}
procedure TForm1.btnGetListClick(Sender: TObject);
var BufSize : Cardinal; P, PName : Pointer; SName : String;
begin
 {Очищаем предыдущий список}
 lbNameList.Items.Clear;

 {Размер буфера}
 BufSize:= 10240;
 {Распределяем память для буфера}
 GetMem(P, BufSize);
 {Запрашиваем список имен}
 If QueryDosDevice(nil, P, BufSize) <> 0 then begin
  {Цикл по всем именам...}
  PName:= P;
  While (True) do begin
   SName:= StrPas(PName);
   If SName = '' then Break;
   {Добавляем в список}
   lbNameList.Items.Add(SName);
   {Переход к следующему устройству}
   {Сдвигаем указатель на следующую строку}
   PName:= Pointer(LongInt(PName) + Length(SName)+1);
  End;
 End;
 {Освобождаем буфер}
 FreeMem(P);
end;
```

21.31. Функция *DefineDosDevice*: операции с DOS-именем устройства

Функция `DefineDosDevice` определяет, переопределяет или удаляет DOS-имя из системы. Выполняемая операция зависит от параметра `dwFlags`. Его возможные значения приведены в табл. 21.8.

Формат заголовка на языке C имеет вид:

```
BOOL DefineDosDevice(
  DWORD   dwFlags,    // код операции
  LPCTSTR pName,      // DOS-имя устройства или его префикс
  LPCTSTR pDevice,    // NT-имя устройства
);
```

Формат заголовка на языке Delphi имеет вид:

```
function DefineDosDevice(
  dwFlags      : DWORD;  // код операции
  lpDeviceName,          // DOS-имя устройства или его префикс
  lpTargetPath : PChar   // NT-имя устройства
): BOOL;
```

Первый параметр задает код операции (табл. 21.8). Второй параметр передает DOS-имя устройства, а третий — внутреннее имя устройства. Для выполнения этой функции требуются права администратора системы.

21.31.1. Возвращаемое значение

При успешном выполнении функция `DefineDosDevice` возвращает ненулевое значение. В случае ошибки функция вернет 0, а вызов `GetLastError` вернет код ошибки.

21.31.2. Пример вызова

Листинг 21.15 показывает добавление имени в систему.

Листинг 21.15. Пример вызова DefineDosDevice

```
{Добавить DOS-имя для NT-имени}
procedure TForm1.btnAddNameClick(Sender: TObject);
begin
 If not DefineDosDevice(
   DDD_RAW_TARGET_PATH,
```

```
   PChar(DosDeviceName.Text),
   PChar(NtDeviceName.Text))
 then
  StatusBar.Panels[0].Text:= 'Ошибка добавления имени';
end;
```

Таблица 21.8. *Коды операций функции* `DefineDosDevice`

Значение	Описание
`DDD_RAW_TARGET_PATH`	При указании этого параметра функция не конвертирует значение `pDevice` в имя устройства, а берет указанное значение имени "как есть"
`DDD_REMOVE_DEFINITION`	Удаление указанного DOS-имени для выбранного устройства. Функция просматривает все имена, зарегистрированные для устройства, и удаляет те, префиксы которых совпадают с указанной в `pName` строкой. Если существует несколько таких имен для устройства, функция удалит только первое из них. Если параметр `pDevice` нулевой или указывает на строку нулевой длины, то функция удалит первое подходящее имя из списка имен системы
`DDD_EXACT_MATCH_ON_REMOVE`	Указывается вместе с `DDD_REMOVE_DEFINITION` для гарантирования, что функция не удалит лишнего. Функция будет удалять имя только при полном совпадении имени из списка имен и указанного в `pName`

Глава 22

Функции *DiviceIoControl* для последовательных портов

Описание и параметры функции `DeviceIoControl` приведены в *гл. 21*, а в этой главе мы опишем функции коммуникационного драйвера, которые можно вызывать с помощью `DeviceIoControl`.

Еще раз отметим, что функции `DeviceIoControl` для последовательных портов можно использовать только начиная с Windows 2000.

22.1 Функции драйвера последовательного порта

Все приведенные функции описываются в контексте параметров `DeviceIoControl`:

- для вызова конкретной функции необходимо сформировать ее код с помощью функции `Get_Ctl_Code(числовой_код_функции)` из *разд. 13.2*;
- указание относительно входных и выходных параметров функции нужно расценивать как указание на содержимое буферов `lpInBuffer` и `lpOutBuffer` функции `DeviceIoControl` *(см. разд. 21.28)*.

Последнее означает, что указание "входные параметры отсутствуют" свидетельствует о том, что `lpInBuffer = nil`, `dwInBufSize = 0`, но при наличии выходных параметров функции передается значение `dwOutBufSize`.

Далее перечислим функции `DeviceIoControl` для последовательных портов.

- Функция `IOCTL_SERIAL_SET_BAUD_RATE` устанавливает скорость обмена последовательного порта. Входной буфер должен содержать указатель на переменную типа `Cardinal`, равную скорости обмена. Выходных параметров нет.

Листинг 22.1. Пример вызова DeviceIoControl(1)

```
Baud : Cardinal;
Baud:= 19200;
DeviceIoControl(
       FHandle, Get_Ctl_Code(1),
       @Baud, SizeOf(Baud),
       nil, 0,
       Skip, nil
);
```

В MSDN почему-то указывается, что скорость обмена задается константами `SERIAL_BAUD_xxx`, однако простая проверка показывает, что скорость обмена задается константами `CBR_xxx`, которые используются в структуре `DCB`.

Аналог в Windows API — функция `SetCommState`.

- Функция `IOCTL_SERIAL_SET_QUEUE_SIZE` устанавливает размер внутреннего приемного буфера. Если требуемый размер буфера больше, чем текущий размер, то будет создан новый буфер, иначе размер буфера не изменится. Входной параметр — структура `TSERIAL_QUEUE_SIZE`, выходных параметров нет. Аналог в Windows API — функция `SetupComm`.
- Функция `IOCTL_SERIAL_SET_LINE_CONTROL` устанавливает значение порта LCR (Line Control Register), который содержит характеристики обмена: число стоп-битов, число битов в байте и настройки контроля четности. Входной параметр — структура `TSERIAL_LINE_CONTROL`, выходных параметров нет.

  Поля этой структуры могут принимать значения:

  - Значения поля `WordLength`:
    - `SERIAL_5_DATA = 5`;
    - `SERIAL_6_DATA = 6`;
    - `SERIAL_7_DATA = 7`;
    - `SERIAL_8_DATA = 8`.
  - Значения поля `StopBits`:
    - `STOP_BIT_1 = 0`;
    - `STOP_BITS_1_5 = 1` (только для 5-битовых данных);
    - `STOP_BITS_2 = 2` (не для 5-битовых данных).

- Значения поля Parity:
  - ◊ NO_PARITY = 0;
  - ◊ ODD_PARITY = 1;
  - ◊ EVEN_PARITY = 2;
  - ◊ MARK_PARITY = 3;
  - ◊ SPACE_PARITY = 4.

Аналог в Windows API — функция SetCommState.

- ❐ Функция IOCTL_SERIAL_SET_BREAK_ON устанавливает сигнал прерывания так же, как SetCommBreak. Входных и выходных параметров нет. Ту же операцию можно выполнить вызовом EscapeCommFunction(SETBREAK).
- ❐ Функция IOCTL_SERIAL_SET_BREAK_OFF сбрасывает сигнал прерывания так же, как ClearCommBreak. Входных и выходных параметров нет. Ту же операцию можно выполнить вызовом EscapeCommFunction(CLRBREAK).
- ❐ Функция IOCTL_SERIAL_IMMEDIATE_CHAR требует передачу специального символа вне очереди, аналогично вызову TransmitCommChar. Входной параметр — указатель на переменную типа BYTE, содержащую код передаваемого символа. Выходных параметров нет.
- ❐ Функция IOCTL_SERIAL_SET_TIMEOUTS устанавливает значение тайм-аутов порта. Входной параметр — указатель на структуру TSERIAL_TIMEOUTS. Выходных параметров нет. Аналог в Windows API — функция SetCommTimeouts.
- ❐ Функция IOCTL_SERIAL_GET_TIMEOUTS возвращает информацию о тайм-аутах порта. Входных параметров нет, выходной — указатель на структуру TSERIAL_TIMEOUTS. Аналог в Windows API — функция GetCommTimeouts.
- ❐ Функция IOCTL_SERIAL_SET_DTR устанавливает сигнал DTR. Если драйвер сконфигурирован на автоматическое управление этим сигналом, функция не выполнится. Входных и выходных параметров нет. Аналогичные действия можно выполнить вызовом EscapeCommFunction(SETDTR).
- ❐ Функция IOCTL_SERIAL_CLR_DTR сбрасывает сигнал DTR. Если драйвер сконфигурирован на автоматическое управление этим сигналом, функция не выполнится. Входных и выходных параметров нет. Аналогичные действия можно выполнить вызовом EscapeCommFunction(CLRDTR).
- ❐ Функция IOCTL_SERIAL_RESET_DEVICE производит рестарт порта. Входных и выходных параметров нет. Аналогичные действия можно выполнить вызовом EscapeCommFunction(RESETDEV).
- ❐ Функция IOCTL_SERIAL_SET_RTS устанавливает сигнал RTS. Если драйвер сконфигурирован на автоматическое управление этим сигналом, функция

не выполнится. Входных и выходных параметров нет. Аналогичные действия можно выполнить вызовом `EscapeCommFunction(SETRTS)`.

- Функция `IOCTL_SERIAL_CLR_RTS` сбрасывает сигнал RTS. Если драйвер сконфигурирован на автоматическое управление этим сигналом, функция не выполнится. Входных и выходных параметров нет. Аналогичные действия можно выполнить вызовом `EscapeCommFunction(CLRRTS)`.
- Функция `IOCTL_SERIAL_SET_XOFF` эмулирует получение символа `Xoff`. Если приложение не использует автоматическое управление обменом с помощью `Xon/Xoff`, клиент должен сам подать сигнал `Xon`. Входных и выходных параметров нет. Аналогичные действия можно выполнить вызовом `EscapeCommFunction(SETXOFF)`.
- Функция `IOCTL_SERIAL_SET_XON` эмулирует получение символа `Xon`. Если приложение не использует автоматическое управление обменом с помощью `Xon/Xoff`, клиент должен сам подать сигнал `Xoff`. Аналогичные действия можно выполнить вызовом `EscapeCommFunction(SETXON)`.
- Функция `IOCTL_SERIAL_GET_WAIT_MA` возвращает маску ожидания событий. Аналог функции `GetCommMask`. Входных параметров нет, выходной — указатель на переменную типа `Cardial`, содержащую маску флагов событий (табл. 22.1).

Таблица 22.1. *Маски ожидания для функции `IOCTL_SERIAL_GET_WAIT_MASK`*

Флаг	Значение	Описание
SERIAL_EV_RXCHAR	$0001	Получен символ
SERIAL_EV_RXFLAG	$0002	Получен специальный символ, описанный в блоке DCB
SERIAL_EV_TXEMPTY	$0004	Буфер посылки опустошен
SERIAL_EV_CTS	$0008	Сигнал CTS изменил состояние
SERIAL_EV_DSR	$0010	Сигнал DSR изменил состояние
SERIAL_EV_RLSD	$0020	Сигнал RLSD изменил состояние
SERIAL_EV_BREAK	$0040	Получен сигнал BREAK
SERIAL_EV_ERR	$0080	Произошла ошибка приема
SERIAL_EV_RING	$0100	Обнаружен сигнал вызова
SERIAL_EV_PERR	$0200	Ошибка принтера
SERIAL_EV_RX80FULL	$0400	Приемный буфер заполнен более чем на 80 процентов
SERIAL_EV_EVENT1	$0800	Событие, определяемое драйвером
SERIAL_EV_EVENT2	$1000	Событие, определяемое драйвером

Следует отметить, что, хотя аналогом этой функции является функция GetCommMask, аналогов флагов SERIAL_EV_PERR, SERIAL_EV_RX80FULL, SERIAL_EV_EVENT1 и SERIAL_EV_EVENT2 для GetCommMask не существует.

- Функция IOCTL_SERIAL_SET_WAIT_MASK устанавливает маску ожидания событий. Входной параметр — указатель на переменную типа Cardinal, содержащую маску событий согласно табл. 22.1. Аналог в Windows API — функция SetCommMask, однако аналогов некоторым флагам прямого вызова не существует.
- Функция IOCTL_SERIAL_WAIT_ON_MASK осуществляет ожидание события, установленного функцией IOCTL_SERIAL_SET_WAIT_MASK. Входных параметров нет, а выходной — указатель на переменную типа Cardinal, содержащую маску произошедших событий. Аналог в Windows API — WaitCommEvent.
- Функция IOCTL_SERIAL_PURGE очищает выбранные буферы согласно маске, передаваемой в качестве входного параметра — указателя на переменную типа Cardinal. Маска может состоять из следующих констант:
  - SERIAL_PURGE_RXABORT ($01) — все буфера;
  - SERIAL_PURGE_RXCLEAR ($02) — входной буфер;
  - SERIAL_PURGE_TXABORT ($04) — все запросы на передачу данных;
  - SERIAL_PURGE_TXCLEAR ($08) — выходной буфер.

  Аналог в Windows API — функция PurgeComm.
- Функция IOCTL_SERIAL_GET_BAUD_RATE возвращает значение текущей скорости обмена. Входных параметров нет, выходной — указатель на переменную типа Cardinal, содержащую результат. Аналог в Windows API — функция GetCommState.
- Функция IOCTL_SERIAL_GET_LINE_CONTROL возвращает характеристики обмена: число стоп-битов, число битов в байте и контроле четности. Входных параметров нет, выходной — указатель на структуру TSERIAL_LINE_CONTROL. Аналог в Windows API — функция GetCommState.
- Функция IOCTL_SERIAL_GET_CHARS возвращает (устанавливает) значения специальных символов, применяющихся для управления обменом: XON, XOFF и т. д. Входных параметров нет, выходной — указатель на структуру TSERIAL_CHARS. Аналог в Windows API — функция GetCommState.
- Функция IOCTL_SERIAL_SET_CHARS устанавливает значения специальных символов, применяющихся для управления обменом: XON, XOFF и т. д. Входной параметр — указатель на структуру TSERIAL_CHARS, выходных параметров нет. Аналог в Windows API — функция SetCommState.
- Функция IOCTL_SERIAL_GET_HANDFLOW возвращает режим управления линиями DTR и RTS. Входных параметров нет, выходной — указатель на

структуру TSERIAL_HANDFLOW. Аналог в Windows API — функция GetCommState.

- Функция IOCTL_SERIAL_SET_HANDFLOW устанавливает режим управления линиями DTR и RTS. Входной параметр — указатель на структуру TSERIAL_HANDFLOW, выходных параметров нет. Аналог в Windows API — функция SetCommState.
- Функция IOCTL_SERIAL_GET_MODEMSTATUS возвращает состояние линий модема. Входных параметров нет, выходной — указатель на переменную типа Cardinal, содержащую маску состояния линий модема (табл. 22.2).

Таблица 22.2. *Маски состояния модема для* `IOCTL_SERIAL_GET_MODEMSTATUS`

Маска	Значение	Описание
SERIAL_IOC_MCR_DTR	$01	Сигнал DTR активен
SERIAL_IOC_MCR_RTS	$02	Сигнал RTS активен
SERIAL_IOC_MCR_OUT1	$04	Сигнал OUT1 активен
SERIAL_IOC_MCR_OUT2	$08	Сигнал OUT2 активен
SERIAL_IOC_MCR_LOOP	$10	Состояние тестирования

Аналог этой команды — функция GetCommModemStatus, однако, она не позволяет получить состояние линий OUT1 и OUT2.

- Функция IOCTL_SERIAL_GET_COMMSTATUS возвращает статус последовательного порта. Входных параметров нет, выходной — указатель на структуру TSERIAL_STATUS. Аналог в Windows API — функция GetCommState.
- Функция IOCTL_SERIAL_XOFF_COUNTER устанавливает счетчик символов XOFF. Входной параметр — указатель на структуру TSERIAL_STATUS.
- Функция IOCTL_SERIAL_GET_PROPERTIES возвращает информацию о возможных режимах порта. Входных параметров нет, выходной — указатель на структуру TSERIAL_COMMPROP. Аналог в Windows API — GetCommProp.
- Функция IOCTL_SERIAL_GET_DTRRTS возвращает информацию о сигналах DTR и RTS. Входных параметров нет, выходной — указатель на переменную типа Cardinal, возвращающую маску флагов SERIAL_xxx:
  - SERIAL_DTR_STATE = $01;
  - SERIAL_RTS_STATE = $02;
  - SERIAL_CTS_STATE = $10;
  - SERIAL_DSR_STATE = $20;

- SERIAL_RI_STATE = $40;
- SERIAL_DCD_STATE = $80.

- Функция IOCTL_SERIAL_LSRMST_INSERT включает или выключает вставку информации о статусе линии и статусе модема в принимаемый поток данных.
- Функция IOCTL_SERIAL_CONFIG_SIZE возвращает размер конфигурационной структуры. Для последовательных портов всегда возвращает 0. Не поддерживается в Windows 2000 и выше.
- Функция IOCTL_SERIAL_GET_COMMCONF возвращает конфигурацию последовательного порта.
- Функция IOCTL_SERIAL_SET_COMMCONFIG устанавливает конфигурацию последовательного порта.
- Функция IOCTL_SERIAL_GET_STATS возвращает статистику работы COM-порта. Статистика включает в себя число переданных байтов, число принятых байтов и статистику ошибок. Драйвер самостоятельно увеличивает эти значения при обмене. Входных параметров нет, выходной — указатель на структуру TSERIALPERF_STATS.
- Функция IOCTL_SERIAL_CLEAR_STATS очищает статистику обмена по COM-порту. Для получения статистики используется функция с кодом IOCTL_SERIAL_GET_STATS. Функция не имеет ни входных, ни выходных параметров.
- Функция IOCTL_SERIAL_GET_MODEM_CONTROL возвращает состояние регистров модема. Входных параметров нет, выходной — указатель на переменную типа Cardinal, содержащую маску флагов SERIAL_xxx:
  - SERIAL_IOC_MCR_DTR = $01;
  - SERIAL_IOC_MCR_RTS = $02;
  - SERIAL_IOC_MCR_OUT1 = $04;
  - SERIAL_IOC_MCR_OUT2 = $08;
  - SERIAL_IOC_MCR_LOOP = $10.
- Функция IOCTL_SERIAL_SET_MODEM_CONTROL устанавливает состояние регистров модема. Входной параметр — указатель на переменную типа Cardinal, содержащую маску флагов (как в функции IOCTL_SERIAL_GET_MODEM_CONTROL, см. *разд. 22.37*), выходных параметров нет.
- Функция IOCTL_SERIAL_SET_FIFO_CONTROL записывает значение регистра FCR (FIFO Control Register). Драйвер не проверяет устанавливаемое значение. Входной параметр — указатель на переменную типа Cardinal, содержащую значение регистра, выходных параметров нет.

22.2. Структуры драйвера последовательного порта

Ниже, в листингах 22.2—22.10, приводятся описания структур, используемых для обращения к коммуникационному порту с помощью функции `DeviceIoControl`.

Листинг 22.2. Структура TSERIAL_LINE_CONTROL

```
TSERIAL_LINE_CONTROL = packed record
  StopBits   : Byte;
  Parity     : Byte;
  WordLength : Byte;
End;
```

Листинг 22.3. Структура TSERIAL_QUEUE_SIZE

```
TSERIAL_QUEUE_SIZE = packed record
  InSize  : Cardinal;
  OutSize : Cardinal;
End;
```

Листинг 22.4. Структура TSERIAL_TIMEOUTS

```
TSERIAL_TIMEOUTS = packed record
  ReadIntervalTimeout         : Cardinal;
  ReadTotalTimeoutMultiplier  : Cardinal;
  ReadTotalTimeoutConstant    : Cardinal;
  WriteTotalTimeoutMultiplier : Cardinal;
  WriteTotalTimeoutConstant   : Cardinal;
End;
```

Листинг 22.5. Структура TSERIAL_STATUS

```
TSERIAL_STATUS = packed record
  Errors           : Cardinal;
  HoldReasons      : Cardinal;
```

```
  AmountInInQueue  : Cardinal;
  AmountInOutQueue : Cardinal;
  EofReceived      : BOOL;
  WaitForImmediate : BOOL;
End;
```

Листинг 22.6. Структура TSERIAL_HANDFLOW

```
TSERIAL_HANDFLOW = packed record
  ControlHandShake : Cardinal;
  FlowReplace      : Cardinal;
  XonLimit         : Cardinal;
  XoffLimit        : Cardinal;
End;
```

Листинг 22.7. Структура TSERIAL_CHARS

```
TSERIAL_CHARS = packed record
  EofChar   : Byte;
  ErrorChar : Byte;
  BreakChar : Byte;
  EventChar : Byte;
  XonChar   : Byte;
  XoffChar  : Byte;
End;
```

Листинг 22.8. Структура TSERIAL_XOFF_COUNTER

```
TSERIAL_XOFF_COUNTER = packed record
  Timeout  : Cardinal; // задается в мс
  Counter  : Longint;  // должно быть больше или равно нуля
  XoffChar : Byte;
End;
```

Листинг 22.9. Структура TSERIAL_COMMPROP

```
TSERIAL_COMMPROP = packed record
  PacketLength       : WORD;
```

```
  PacketVersion      : WORD;
  ServiceMask        : Cardinal;
  Reserved1          : Cardinal;
  MaxTxQueue         : Cardinal;
  MaxRxQueue         : Cardinal;
  MaxBaud            : Cardinal;
  ProvSubType        : Cardinal;
  ProvCapabilities   : Cardinal;
  SettableParams     : Cardinal;
  SettableBaud       : Cardinal;
  SettableData       : WORD;
  SettableStopParity : WORD;
  CurrentTxQueue     : Cardinal;
  CurrentRxQueue     : Cardinal
  ProvSpec1          : Cardinal;
  ProvSpec2          : Cardinal;
  ProvChar           : Pointer;
End;
```

Листинг 22.10. Структура TSERIALPERF_STATS

```
TSERIALPERF_STATS = packed record
  ReceivedCount    : Cardinal;
  TransmittedCount : Cardinal;
  FrameErrorCount  : Cardinal;
  SerialOverrunErrorCount : Cardinal;
  BufferOverrunErrorCount : Cardinal;
  ParityErrorCount : Cardinal;
End;
```

22.3. Соответствие функций *DeviceIoControl* и Windows API

В табл. 22.3 приведено соответствие основных кодов функций прямого обращения к драйверу и их аналогов в Windows API.

Таблица 22.3. *Таблица соответствия кодов* `DeviceIoControl` *и Windows API*

Код	Функция `DeviceIoControl`	Аналог Windows API
1	IOCTL_SERIAL_SET_BAUD_RATE	SetCommState
2	IOCTL_SERIAL_SET_QUEUE_SIZE	SetupComm
3	IOCTL_SERIAL_SET_LINE_CONTROL	SetCommState
4	IOCTL_SERIAL_SET_BREAK_ON	SetCommBreak
5	IOCTL_SERIAL_SET_BREAK_OFF	ClearCommBreak
6	IOCTL_SERIAL_IMMEDIATE_CHAR	TransmitCommChar
7	IOCTL_SERIAL_SET_TIMEOUTS	SetCommTimeouts
8	IOCTL_SERIAL_GET_TIMEOUTS	GetCommTimeouts
9	IOCTL_SERIAL_SET_DTR	EscapeCommFunction
10	IOCTL_SERIAL_CLR_DTR	EscapeCommFunction
11	IOCTL_SERIAL_RESET_DEVICE	EscapeCommFunction
12	IOCTL_SERIAL_SET_RTS	EscapeCommFunction
13	IOCTL_SERIAL_CLR_RTS	EscapeCommFunction
14	IOCTL_SERIAL_SET_XOFF	EscapeCommFunction
15	IOCTL_SERIAL_SET_XON	EscapeCommFunction
16	IOCTL_SERIAL_GET_WAIT_MASK	GetCommMask
17	IOCTL_SERIAL_SET_WAIT_MASK	SetCommMask*
18	IOCTL_SERIAL_WAIT_ON_MASK	WaitCommEvent*
19	IOCTL_SERIAL_PURGE	PurgeComm
20	IOCTL_SERIAL_GET_BAUD_RATE	GetCommState
21	IOCTL_SERIAL_GET_LINE_CONTROL	GetCommState
22	IOCTL_SERIAL_GET_CHARS	GetCommState
23	IOCTL_SERIAL_SET_CHARS	SetCommState
24	IOCTL_SERIAL_GET_HANDFLOW	GetCommState
25	IOCTL_SERIAL_SET_HANDFLOW	SetCommState
26	IOCTL_SERIAL_GET_MODEMSTATUS	GetCommModemStatus*
27	IOCTL_SERIAL_GET_COMMSTATUS	GetCommState
28	IOCTL_SERIAL_XOFF_COUNTER	
29	IOCTL_SERIAL_GET_PROPERTIES	GetCommProp
30	IOCTL_SERIAL_GET_DTRRTS	

Таблица 22.3 *(окончание)*

Код	Функция DeviceIoControl	Аналог Windows API
31	IOCTL_SERIAL_LSRMST_INSERT	
32	IOCTL_SERIAL_CONFIG_SIZE	
33	IOCTL_SERIAL_GET_COMMCONFIG	GetCommConfig
34	IOCTL_SERIAL_SET_COMMCONFIG	SetCommConfig
35	IOCTL_SERIAL_GET_STATS	
36	IOCTL_SERIAL_CLEAR_STATS	
37	IOCTL_SERIAL_GET_MODEM_CONTROL	GetCommModemStatus*
38	IOCTL_SERIAL_SET_MODEM_CONTROL	
39	IOCTL_SERIAL_SET_FIFO_CONTROL	

* Звездочкой отмечены коды, функциональность которых шире, чем соответствующих функций Windows API.

Глава 23

AT-команды и регистры модемов

В этой главе мы опишем основные команды и регистры модемов. Эта информация будет полезна при программировании или управлении модемами из терминальной программы.

23.1. Основные AT-команды модемов

- Команда `A/` — выполнить повторно последнюю команду — это единственная, не считая Escape-последовательности, команда, перед которой не требуются символы AT. Эта команда выполняет повтор предыдущей команды.
- Команда `ATA` инициирует режим ответа. Модем выходит из режима ожидания и переходит в режим передачи данных (DATA MODE). Через 2 секунды (задержка подтверждения) модем выдаст тональный сигнал ответа. Если после завершения паузы, задаваемой регистром `S7`, не будет обнаружена несущая, то модем перейдет в состояние ожидания и вернется в холостой режим (IDLE MODE).
- Команда `ATBn`: выбор BELL/ITU-T. Эта команда выбирает тип протокола:
  - `ATB0`: выбирает протокол ITU-T V.22;
  - `ATB1`: протокол Bell 212A.

  Эти команды допустимы только для скорости передачи 1200 бит/с. По умолчанию выбран протокол Bell 212A (`n = 1`).
- Команда `ATCn` управляет передачей сигнала несущей. Значение `n = 0` запрещает передачу несущей, а значение `n = 1` разрешает.
- Команда `ATD`: набор номера.

  Набираемые цифры: `0 1 2 3 4 5 6 7 8 9 # * A B C D`

  Разделители: `" "` (пробел)

  Модификаторы: `T P W @ ! R S / ; ,`

Команда ATD начинает последовательность набора номера. Когда обнаружена команда D, модем ждет 2 секунды и затем начинает набор номера. Модем набирает 12 стандартных тоновых цифр. Эти цифры могут быть разделены символами-разделителями, указанными выше. Модификаторы определяют способ, которым будет осуществляться набор:

- T: тоновый набор (т. е. ATDT 3451798);
- P: импульсный набор (т. е. ATDP 2627506).

Все следующие модификаторы используются внутри строки набираемых цифр (например, ATDT 34517W9821 или ATDP 262@7506).

- Модификатор "W": ждать наборного тона. Этот модификатор вызовет задержку набора до появления тонального сигнала набора. Если по тайм-ауту, задаваемому регистром S7, не будет обнаружен тональный сигнал набора, вызов будет прерван.
- Модификатор "@": ждать ответной паузы. Если в строке набора будет обнаружен этот символ, модем выдерживает 5-секундную паузу. Если паузы не будет в течение 30 секунд, то вызов будет прерван, и модем вернет код результата NO ANSWER — нет ответа.
- Модификатор "!": ожидание. Этот символ переводит модем в состояние ожидания на 1/2 секунды.
- Модификатор "R": возобновить режим ответа. Если обнаружен этот символ, модем ждет 2 секунды и затем переходит в режим ответа, что эквивалентно команде ATA.
- Модификатор "S": набрать хранимый номер. Команда ATDS вызовет набор номера, сохраняемого в энергонезависимой памяти.
- Модификатор "/": ждать 1/8 секунды. Эта пауза идентична паузе, вызванной модификатором "запятая", с той лишь разницей, что длится 1/8 секунды.
- Модификатор ";": вернуться в командный режим. Этот символ заставляет модем вернуться в холостой режим (IDLE MODE). Модем остается активизированным.
- Модификатор ",": ждать S8 секунд. Этот символ заставляет модем ждать в течение периода, который определяется содержимым регистра S8. Затем модем продолжит набор номера.

- Команда ATEn: команда эхо. Эта команда управляет режимом возвращения (ATE1) или невозвращения (ATE0) командных символов. При включении питания все символы, посылаемые в модем, возвращаются. По умолчанию устанавливается значение n = 1.
- Команда ATFn: установка режима дуплекса. При n = 0 устанавливается полудуплексный режим, а при n = 1 — полнодуплексный режим.

- Команда ATHn: команда подключения к линии. Параметр n может быть равным 0, 1 или 2. По умолчанию устанавливается 0 (состояние ожидания). При n = 1 (команда ATH1) выполняется команда "поднять трубку", а при n = 0 — команда "положить трубку". Команда ATH2 включена только в целях совместимости и не вызывает никаких действий.
- Команда ATIn: команда идентификации. Команда вызывает выдачу идентификационного кода модема. Различные модемы поддерживают различные наборы команд n.
- Команда ATLn управляет громкостью звука:
  - ATL0: слабая громкость;
  - ATL1: низкая громкость;
  - ATL2: средняя громкость (по умолчанию);
  - ATL3: высокая громкость.
- Команда ATMn управляет работой громкоговорителя:
  - ATM0: громкоговоритель выключен всегда;
  - ATM1: громкоговоритель включен в процессе вызова и ответа, т. е. до появления несущей (по умолчанию);
  - ATM2: громкоговоритель включен всегда;
  - ATM3: то же, что и ATM1, но не слышен тон DTMF.
- Команда ATN задает режим согласования скоростей.
  - ATN0: регистр S37 определяет скорость, на которой модем осуществляет соединение с удаленным модемом. Если S37 = 0, то скорость соединения совпадает со скоростью передачи данных от компьютера к модему. Если скорость выбирается более, чем из одного стандарта (таких как BELL 212A или CCITT V.22 для скорости 1200 бит/с), то это достигается варьированием команды ATB(0,1).
  - ATN1: соединение может происходить на любой скорости, которую поддерживают и локальный, и удаленный модемы. При выборе работы по стандарту CCITT команда B игнорируется.
- Команда ATOn управляет режимом работы модема (по умолчанию n = 0).
  - ATO0. Работа этой команды зависит от текущего режима модема. Если модем находится в командном режиме (COMMAND MODE), команда ATO0 переведет модем в режим данных (DATA MODE) и инициализирует установление связи на скорости автоматически установленной командой AT. Этой команде должна предшествовать команда ATH1 для выведения модема из режима ожидания. Если модем находится в интерактивном режиме (ON-LINE INTERACTIVE MODE), команда ATO0 вернет модем в режим DATA MODE.

- `ATO1`. Если модем находится в интерактивном режиме (ON-LINE INTERACTIVE MODE), команда `ATO1` запросит настроечную последовательность от другого модема. Эта команда может быть использована, только если модем соединен в режиме V.22bis (2400 бит/с).

- Команда `ATQn`: выдача сообщений модемом. Эта команда управляет посылкой ответных сообщений к DTE. Пользователь должен обеспечить время, необходимое для завершения команды. Значение `n = 0` разрешает ответные сообщения, а значение `n = 1` — запрещает. По умолчанию `n = 0`. Команда `ATZ` сбрасывает режим в режим по умолчанию.

- Команда `ATSr?`: запрос `S`-регистра. Эта команда возвращает десятичную величину, хранящуюся в выбранном `S`-регистре. Например, для получения содержимого регистра 7 выполните команду `ATS7?`.

- Команда `ATSr=n`: установка регистра. Эта команда используется для загрузки значения в `S`-регистр. Например, для установки 3 в регистр 0 выполните `ATS0 = 3`.

- Команда `ATVn`: выбор кодов для сообщений модема. Эта команда управляет форматом кода ответа. `ATV0` устанавливает режим "сжатые цифровые ответы", а `ATV1` — "развернутые словесные ответы". По умолчанию `n = 1`.

- Команда `ATWn` устанавливает режим сообщения о состоянии соединения:

  - `ATW0`: не сообщает скорость соединения и используемый протокол коррекции ошибок;

  - `ATW1`: сообщает скорость соединения и используемый протокол коррекции ошибок;

  - `ATW2`: не сообщает скорость соединения и используемый протокол коррекции ошибок. При установке соединения на экране отображается строка CONNECT и скорость COM-порта.

- Команда `ATXn` управляет разрешением определенных кодов ответа. Подробности можно найти в документации модема.

- Команда `ATYn`: длинный разрыв. Эта команда позволяет пользователю разрешить или запретить (по умолчанию) дополнительную возможность разрыва связи по состоянию "длинная пауза или длинный пробел" (Long Space).

  - `ATY0`: запретить длинный разрыв.

  - `ATY1`: разрешить длинный разрыв. Если модем получает непрерывный сигнал разрыва от удаленного модема в течение более чем 1,6 секунд, модем "зависает". Если затем появляется команда `ATH0` или DTR прерывает связь (только при задействованной опции &D2), модем разрывает соединение и в течение 4 секунд передает сигнал разрыва, а затем переходит в состояние ожидания.

- Команда ATZ: сброс модема. Эта команда восстанавливает исходное состояние модема в соответствии с параметрами, хранящимися в энергонезависимой памяти. Если модем был в интерактивном режиме (ON_LINE INTERACTIVE MODE), эта команда прерывает вызов и переводит модем в состояние ожидания.
- Команда AT&Cn: управление несущей. Если n = 0, сигнал DCD (контакт 8 интерфейса RS-232C) задействован всегда. При n = 1 сигнал DCD включается при обнаружении несущей.
- Команда AT&Dn: управление сигналом DTR (готовность передачи данных):
  - AT&D0: сигнал DTR всегда задействован;
  - AT&D1: модем переходит в командный режим при отсутствии сигнала DTR;
  - AT&D2: модем переходит в состояние ожидания и возвращается в командный режим при прекращении сигнала DTR;
  - AT&D3: модем инициализируется при прекращении сигнала DTR.
- Команда AT&F: загрузка S-регистров из постоянной памяти (заводская установка).
- Команда AT&Gn: управление Guard-тоном (сигналом предупреждения):
  - AT&G0: нет сигнала предупреждения;
  - AT&G1: Guard-тон 550 Гц;
  - AT&G2: Guard-тон 1800 Гц.
- Команда AT&Mn выбирает один из режимов работы.
  - AT&M0: асинхронный режим.
  - AT&M1: синхронный режим 1 — асинхронный вызов и последующий переход в синхронный режим.
  - AT&M2 : синхронный режим 2 — набор хранимого в памяти номера.
  - AT&M3: синхронный режим 3 — контроль с помощью сигнала DTR режима "данные/разговор" (Data/Talk Mode). Этот режим позволяет позвонить с параллельного телефонного аппарата при выключенном сигнале DTR, поговорить, после чего перевести модем в режим обмена данных путем включения сигнала DTR.
- Команда AT&Qn управляет режимом связи.
  - AT&Q0: асинхронная связь в прямом режиме. Скорость по COM-порту должна совпадать со скоростью связи по телефонной линии.
  - AT&Q1: синхронный режим 1 для терминалов, работающих в синхронном и асинхронном режимах. Модем получает команды набора номера в асинхронном виде, а затем переключается в синхронный режим.

При пропадании сигнала DTR или при потере несущей на время, превышающее установленное, модем разрывает соединение и возвращается в асинхронный режим.

- `AT&Q2`: синхронный режим 2 для синхронных терминалов и хранения номеров абонентов. Асинхронный терминал сохраняет или меняет номера телефонов, вводимые по команде `&Z0` в конфигурацию 0 ПАЗУ. Модем набирает номер при появлении сигнала DTR. Модем разрывает соединение и возвращается в асинхронный режим при пропадании сигнала DTR или при потере несущей на время, превышающее установленное в регистре `S10` (обычно 1,4 с).
- `AT&Q3`: синхронный режим 3, использует сигнал DTR в режиме переключения "голос/данные". Оператор вручную набирает номер при выключенном сигнале DTR, затем включает сигнал DTR, переводя модем в режим данных. Модем разрывает соединение и возвращается в асинхронный режим при пропадании сигнала DTR или при потере несущей на время, превышающее установленное в регистре `S10` (обычно 1,4 с).
- `AT&Q4`: команда зарезервирована.
- `AT&Q5`: режим коррекции ошибок. Модем устанавливает соединение по протоколу коррекции ошибок. Если соединение по протоколу коррекции ошибок не произошло, то разрыв модемом соединения или установление соединения в нормальном режиме определяется регистром `S36`.
- `AT&Q6`: асинхронная связь в нормальном режиме. Активизирован буфер модема, позволяющий установить скорость по последовательному порту бóльшую, нежели физическая скорость передачи данных по линии связи.
- `AT&Q7`: команда зарезервирована.
- `AT&Q8`: модем переходит в MNP-режим работы. Если регистр `S36 = 1`, то модем, находясь в состоянии передачи данных, устанавливает соединение в непосредственном режиме. При ином значении `S36` модем устанавливает соединение в MNP-режиме. Если MNP-режим не может быть установлен, то модем переходит в нормальный режим.
- `AT&Q9`: установление соединения по протоколу V.42bis. Если соединение не может быть установлено, модем устанавливает связь по протоколу V.42 (LAP-M). Если и это невозможно, модем переходит в нормальный режим.

❒ Команда `AT&Rn` управляет сигналами RTS/CTS в синхронном режиме работы:

- `AT&R0`: CTS следует за RTS (по умолчанию);
- `AT&R1`: CTS всегда включен.

- Команда AT&S: управление сигналом DSR:
  - AT&S0: сигнал DSR всегда включен (по умолчанию);
  - AT&S1: сигнал DSR используется как обычно.
- Команда AT&Tn управляет режимом тестирования:
  - AT&T0: завершить выполняемый тест;
  - AT&T1: запускает локальный аналоговый тест;
  - AT&T2: запускает локальный цифровой тест;
  - AT&T3: подготавливает модем к выполнению удаленного цифрового теста по запросу другого модема;
  - AT&T4: запрещает удаленный цифровой тест;
  - AT&T5: разрешает удаленный цифровой тест;
  - AT&T6: инициализирует удаленный цифровой тест;
  - AT&T7: инициализирует удаленный цифровой тест с самопроверкой.
- Команда AT&V: просмотр текущей конфигурации и пользовательских профайлов модема. По этой команде на экране отобразятся активная конфигурация, пользовательские профайлы, а также хранимые в памяти модема телефонные номера.
- Команда AT&W сохраняет текущие значения S-регистров в энергонезависимой памяти, откуда эти установки вызываются при включении питания или командой ATZ.
- Команда AT&Xn: передача тактовых сигналов. Эта команда используется для выбора источника тактовых сигналов (для штырька 15 интерфейса RS-232C) при синхронной связи:
  - AT&X0: тактовые сигналы модема (по умолчанию);
  - AT&X1: тактовые сигналы от DTE;
  - AT&X2: модем создает тактовые сигналы передачи из несущей получаемого сигнала.
- Команда AT&Yn: выбор активной конфигурации из существующих пользовательских профайлов:
  - AT&Y0: загрузка пользовательского профайла 0 в активную конфигурацию;
  - AT&Y1: загрузка пользовательского профайла 1 в активную конфигурацию.
- Команда AT&Z=n: сохранить телефонный номер в энергонезависимой памяти. Параметр n — любая строка цифр или набираемых символов не длиннее 32 знаков. Сохраняемый номер может быть набран с использованием команды ATDS. Можно сохранить только один номер.

23.2. Основные регистры модемов

- Регистр S0: количество звонков до ответа (хранится в энергонезависимой памяти).

  По умолчанию: 0.

  Диапазон: 0...255.

  Устанавливает количество звонков, которые модем должен получить перед тем, как начнет процедуру автоответа. Величина 0 выключит автоответ.

- Регистр S1: счетчик звонков.

  По умолчанию: 0.

  Диапазон: 0...255.

  S1 — это регистр только для чтения, его значение увеличивается на единицу каждый раз при получении сигнала звонка. Он автоматически очищается, если в течение 8 секунд не поступит ни одного сигнала звонка.

- Регистр S2: символ кода ESC.

  По умолчанию: 43, десятичное значение ASCII-символа "+".

  Диапазон: 0...255.

  S2 содержит десятичную ASCII-величину символа ESC. Значение больше, чем 127 запретит обнаружение ESC-кода. Этот код позволяет выйти из режима данных (DATA MODE) и перейти в режим интерактивного обмена (ON LINE INTRACTIVE MODE).

- Регистр S3: символ возврата каретки.

  По умолчанию: 13, десятичное значение ASCII-символа "CR".

  Диапазон: 0...255.

  S3 содержит десятичную величину символа возврата каретки. Этот символ используется как окончание командной строки, а также как результирующий код.

- Регистр S4: символ перевода строки.

  По умолчанию: 10, десятичное значение ASCII-символа "LF".

  Диапазон: 0...255.

  S4 содержит десятичную величину символа перевода строки. Этот символ выдается после символа возврата каретки, если включен режим полного результата командой ATV1.

- Регистр S5: символ возврата на шаг.

  По умолчанию: 8, десятичное значение ASCII-символа "BS".

  Диапазон: 0...255.

S5 содержит десятичную величину символа возврата на шаг. Во время ввода команды при получении символа "BS" имеет место следующая последовательность:

- символ "BS" возвращается;
- на DTE выдается символ пробела (20H);
- другой символ "BS" выдается на DTE.

Если пользователь попытается вернуть на шаг первый командный символ, следующий за AT, то ничего не будет возвращено. Так как каждый символ BS может вызвать посылку на DTE до трех символов, следует принять меры для недопущения передачи символов "BS" один за другим.

- Регистр S6: ожидание тона набора.

  По умолчанию: 2.

  Диапазон: 0...255.

  S6 содержит время в секундах, которое модем будет ждать перед набором телефонного номера. Эта пауза введена для задержки тона набора после выхода из состояния ожидания. Значение S6 игнорируется, если разрешены ATX3 или ATX4. Если в S6 установлено значение меньше, чем 2 секунды, модем игнорирует эту величину и ожидает как минимум 2 секунды.

- Регистр S7: ожидание несущей после набора.

  По умолчанию: 30.

  Диапазон: 0...255.

  S7 содержит время в секундах, которое модем будет ждать появления несущей после завершения набора номера. В режиме ответа в S7 содержится время, в течение которого модем ждет появления несущей после посылки тона ответа.

- Регистр S8: время паузы набора номера.

  По умолчанию: 2.

  Диапазон: 0...255.

  S8 содержит время в секундах, в течение которого модем будет ждать при получении запятой в строке набора номера. Эта возможность используется для введения паузы в процессе набора номера.

- Регистр S9: время ответа на обнаружение несущей.

  По умолчанию: 6 (600 мс).

  Диапазон: 0...255 (в 1/10 с).

  S9 содержит время в 1/10 с, в течение которого модем будет выдавать тон ответа перед продолжением процедуры соединения. Это позволяет увели-

чить время обнаружения несущей в зашумленных линиях и при нестандартных тоновых сигналах.

- Регистр `S10`: время ответа на потерю несущей.

  По умолчанию: 14 (1,4 с).

  Диапазон: 0...255 (в 1/10 с).

  `S10` устанавливает задержку между моментами потери несущей и отключением модема. При увеличении этого времени модем становится менее чувствительным к пропаданию сигналов в линии. При установке в регистр `S10` значения 255 модем игнорирует статус несущей и функционирует таким образом, как будто несущая присутствует всегда.

- Регистр `S11`: длительность тона нажатия.

  По умолчанию: 95 (95 мс).

  Диапазон: 0...255 миллисекунд.

  `S11` содержит длительность в миллисекундах тона нажатия и интервала. Значение по умолчанию вызовет посылку в течение 95 мс тона нажатия и 95-миллисекундный интервал между тонами.

- Регистр `S12`: время ограничения ESC-кода.

  По умолчанию: 50 (1 с).

  Диапазон: 0...255 (1/50 с).

  `S12` содержит величину в 20-миллисекундных единицах времени разграничения ESC-кода. Это время, в течение которого модем задерживает расшифровку ESC-последовательности при получении ESC-кода, поэтому задержки прихода следующих символов ESC-последовательности не должны превышать этого времени.

ЧАСТЬ IV

ПРИЛОЖЕНИЯ

Приложение 1

Микросхемы UART

Общее описание

WD8250 представляет собой программируемый элемент асинхронной связи (ACE) в 40-контактном корпусе. Устройство изготавливается по технологии кремниевых затворов N/MOS. ACE является программируемым устройством, которое использует двунаправленную 8-битовую шину данных с тремя состояниями.

ACE используется для преобразования параллельных данных в последовательный формат со стороны передачи и преобразования последовательных данных в параллельные со стороны приемника. Последовательный формат представляет собой стартовый бит для передачи и приема, следующие от 5 до 8 бит данных, бит паритета (если запрограммирован) и один, полтора (только 5-битовый формат) или два стоповых бита. Максимальная рекомендуемая скорость передачи данных 56 Кбит/с. Внутренние регистры дают возможность пользователю программировать различные типы прерываний, сигналов управления модема и знаковые форматы. Пользователь может считывать состояние ACE в любое время, читая регистры состояния, прерывания и модема.

Дополнительной характерной особенностью ACE является программируемый генератор скорости, который может делить или внутренний синхронизирующий сигнал от кварца, или внешнюю частоту уровня TTL на число от 1 до $(2^{16} - 1)$.

Характерные особенности

- ❒ WD8250 спроектирована для наиболее простого соединения с широко используемыми микропроцессорами (Z-80, 8080A, 6800 и др.).
- ❒ Полная двойная буферизация.
- ❒ Независимое управление передачей, приемом, линиями состояния, прерыванием.

- Программируемый генератор скорости передачи позволяет делить любые входные синхроимпульсы на число от 1 до $(2^{16} - 1)$ и производит внутреннюю шестнадцатикратную синхронизацию.
- Независимый ввод синхронизирующих импульсов приемника.
- Полностью программируемый последовательный интерфейс.
- 5-, 6-, 7- или 8-битовые данные.
- Формирование бита контроля четности или нечетности и его обнаружение.
- Формирование 1, 1,5 или 2 стоповых битов.
- Формирование скорости в бит/с (канал связи со скоростью до 56 К бит/с).
- Обнаружение ложного стартового бита.
- Полные характеристики сообщений о состоянии.
- TTL-драйвер с тремя состояниями для двунаправленной шины данных.
- Генерация и обнаружение прерывания передачи.
- Характеристики внутреннего диагностирования.
- Контроль линии связи для обнаружения неисправной изоляции.
- Моделирование ошибок прерывания передачи, паритета, переполнения, кадрирования.
- Полностью управляемая приоритетная система прерывания.
- Единственный источник питания +5 В.

Описание контактов

В табл. П1.1 приводится краткое описание контактов UART-микросхемы.

Таблица П1.1. *Описание контактов UART-микросхемы*

Номер контакта	Обозначение	Сигнал	Функция
От 1 до 8	D0—D7	Шина данных	Линии ввода/вывода с тремя состояниями. Двунаправленные линии связи межу WD8250 и шиной данных. Все скомпонованные данные TX и RX, управляющие слова и информация о состоянии передаются через шину данных D0—D7
9	RCLK	Вход синхронизации приемника	Этот вход является входом задающей частоты для приемника на ИМС (может быть подсоединен к пятнадцатому контакту BAUDOUT) (*частота = скорость в бит/с* × 16)

Таблица П1.1 *(продолжение)*

Номер контакта	Обозначение	Сигнал	Функция
10	SIN	Вход последовательных данных	Ввод получаемых данных из канала связи (от периферийного устройства, модема или устройства сопряжения)
11	SOUT	Выход последовательных данных	Вывод передаваемых данных, предоставляемых в последовательном коде в канал связи. Сигнал SOUT устанавливается в определенное состояние (лог. 1) при общем сбросе
12	CS0	Выбор ИМС0	Когда уровень сигналов CS0 и CS1 высокий, а CS2 — низкий, выбирается ИМС. Выбор выполняется, когда строб адреса ADS фиксирует выбранные сигналы ИМС
13	CS1	Выбор ИМС1	
14	CS2	Выбор ИМС2	
15	BAUDOUT	Выход синхронизации передатчика	16-кратный синхросигнал передатчика ИМС 8250. Частота синхросигнала равна частоте генератора, разделенной на число, загруженное в регистр делителя. Сигнал BAUDOUT может быть использован для синхронизации приемника при подключении его к RCLR (контакт 09)
16	XTAL1	Вход внешнего синхросигнала	К этим контактам подключают кварц или внешний задающий генератор для установки требуемой скорости передачи
17	XTAL2	Выход внешнего синхросигнала	
18	DOSTR	Вход строба вывода данных	Когда ИМС выбрана, низкий или высокий уровни сигнала DOSTR позволяют запомнить данные в выбранном регистре WD8250 (записывает ЦПУ). Только одна из этих линий должна использоваться, а вторая должна быть установлена в неактивное состояние
19	DOSTR		
20	Vss	Заземление	Заземление

Таблица П1.1 *(продолжение)*

Номер контакта	Обозначение	Сигнал	Функция
21	DISTR	Вход строба ввода данных	Когда ИМС выбрана, низкий или высокий уровень сигнала DISTR позволяет считывать с выбранного регистра WD8250 (читает ЦПУ). Только одна из этих линий может быть использована, а вторая должна быть установлена в неактивное состояние
22	DISTR		
23	DDIS	Выход блокировки буфера данных	Выход имеет низкий уровень сигнала всякий раз, когда данные считываются с WD8250. Может быть использован для изменения направления данных от внешнего приемопередатчика
24	CSOUT	Выход выбора ИМС	Выход имеет высокий уровень сигнала всякий раз, когда ИМС выбрана. Передача данных в линию не может быть инициирована до тех пор, пока уровень сигнала CSOUT высокий
25	ADS	Вход строба адреса	При низком уровне сигнала обеспечивается фиксирование сигналов для выбора регистра (A0, A1, A2) и выбора ИМС (CS0, CS1,CS2). Примечание: положительный фронт сигнала ADS необходим, когда сигналы выбора регистра (A0, A1, A2) и выбора ИМС (CS0, CS1,CS2) нестабильны во время операции считывания или записи. Если это не требуется, уровень сигнала входа ADS может быть постоянно низким
26	A2	Выбор регистра A2	
27	A1	Выбор регистра A1	Эти 3 входа используются для выбора внутреннего регистра WD8250 во время считывания и записи
28	A0	Выбор регистра A0	
29	NC	Нет подсоединения	Не используется

Таблица П1.1 *(окончание)*

Номер контакта	Обозначение	Сигнал	Функция
30	INTRPT	Выход прерывания	Уровень сигнала выхода высокий всякий раз, когда присутствует разрешенное прерывание
31	OUT2	Выход 2	Выход, предназначенный для пользователя, который может быть запрограммирован третьим битом регистра управления модема. Бит, равный 1, формирует низкий уровень сигнала OUT2
32	RTS	Выход запроса передачи	Когда уровень сигнала выхода низкий, он сообщает модему или устройству сопряжения, что WD8250 готов передать данные. *См. регистр управления модема*
33	DTR	Выход готовности терминала данных	Если выход имеет низкий уровень сигнала, он сообщает модему или устройству сопряжения, что WD8250 готов получить данные
34	OUT1	Выход 1	Выход, предназначенный для пользователя, может быть запрограммирован вторым битом регистра управления модема. Бит, равный 1, формирует низкий уровень сигнала OUT1
35	MR	Вход общего сброса	Высокий уровень сигнала сбрасывает регистры в начальные состояния
36	CTS	Вход готовности для передачи	Сигнал с АПД, указывающий, что устройство готово к передаче данных. *См. регистр состояния модема*
37	DSR	Вход готовности устройства сопряжения	Сигнал с АПД, используемый для указания состояния готовности устройства сопряжения. *См. регистр состояния модема*
38	RSLD	Вход детектора принимаемого линейного сигнала канала данных	Сигнал с АПД, указывающий, что идет получение сигнала, который удовлетворяет условиям качества сигнала. *См. регистр состояния модема*
39	RI	Вход индикатора вызова	Вход имеет низкий уровень сигнала, если сигнал вызова получен модемом или устройством сопряжения. *См. регистр состояния модема*
40	Vcc	+5 В	Питание +5 В

Типы микросхем UART

В персональных компьютерах используется несколько типов микросхем, совместимых с 8250:

- 8250 — первая версия UART;
- 16450 — новая версия, но по существу та же 8250;
- 16550 — имеет внутренний FIFO-буфер для приемника и передатчика, но реализованный с ошибками;
- 16550A — то же, что 16550, но ошибки в FIFO исправлены.

Windows определяет тип микросхемы автоматически, а в DOS-режиме можно использовать такой алгоритм:

- проверка микросхемы 8250:
  - сохранить значение регистра расширения (адрес `BasePort+7`);
  - записать случайное число (не равное считанному значению) в регистр расширения;
  - считать регистр расширения;
  - если число не сохранилось, это микросхема 8250;
  - восстановить значение регистра расширения;
- проверка микросхемы 16550:
  - сохранить значение регистра определения прерывания `IER` (адрес `BasePort+2`);
  - включить режим FIFO (записать в `IER` значение 1);
  - прочитать значение регистра `IER`;
  - если бит `IER[6]` равен 1, то это микросхема 16550A;
  - если бит `IER[7]` равен 1, то это микросхема 16550;
  - если предыдущие тесты не прошли, это микросхема 16450.

Windows NT/2000 автоматически включает использование FIFO, если обнаруживает микросхему, которая корректно поддерживает работу в этом режиме. Для ручного управления режимом FIFO следует отключить опцию **Use FIFO buffers** (Использовать FIFO) в окне настроек порта (см. рис. 13.3) или воспользоваться редактором реестра *(см. разд. 5.4)*.

Приложение 2

Библиотека функций для Pascal

В нашей книге мы использовали дополнительные функции, такие как преобразование числа в шестнадцатеричное представление, преобразование строки в число и т. д. В Delphi для таких преобразований уже существуют функции `StrToInt`, `StrToIntDef`, `IntToHex`, `IntToStr` и т. д., а для языка Pascal аналоги таких функций приведены в листинге П2.1.

Листинг П2.1. Функции преобразования для Pascal (DOS)

```
{ Преобразование байта в шестнадцатеричную строку }
Function Byte2Hex(B: Byte):String;
Const hStr : String ='0123456789ABCDEF';
Begin
 Byte2Hex:= hStr[(B div 16)+1]+ hStr[(B mod 16)+1];
End;
{ Преобразование длинного целого в шестнадцатеричную строку }
Function Long2Hex(B: Longint):String;
Begin
 Long2Hex:= Byte2Hex(B div $100)+ Byte2Hex(B and $0FF);
End;
{ Преобразование строки в целое число. Если преобразование }
{ невозможно, возвращается DefValue                          }
Function StrToIntDef(S : String; DefValue : Long):Long;
Var ErrCode : Int; Value :Long;
Begin
 Val(S, Value, ErrCode);
 If ErrCode <> 0 then Value:= DefValue;
```

```
 StrToIntDef:= Value;
End;
{ Преобразование строки в Single-число. Если преобразование }
{ невозможно, возвращается DefValue                          }
Function StrToFloatDef(S : String; DefValue : Single):Single;
Var ErrCode : Int; Value : Single;
Begin
 Val(S, Value, ErrCode);
 If ErrCode <> 0 then Value:= DefValue;
 StrToFloatDef:= Value;
End;
```

Приложение 3

Формат *SINGLE* чисел с плавающей точкой

В отличие от байтов, слов и строк, формат чисел с плавающей точкой может иметь различные реализации. Так, в Borland Pascal имеется формат `SINGLE`, занимающий 4 байта; формат `Real`, занимающий 6 байт и формат `Extended`, занимающий 10 байт. В языке C также имеется несколько форматов представления чисел с плавающей точкой. Со стороны устройств, подключаемых к компьютеру, формат дробных чисел зависит от процессора и реализации его библиотек (стандартных или написанных самостоятельно).

Чаще всего микропроцессорные устройства используют 4-байтовое представление чисел с плавающей точкой — `SINGLE` (IEEE-стандарт чисел с плавающей точкой одинарной точности).

Битовое представление чисел типа `SINGLE` приведено в табл. П3.1.

Таблица П3.1. *Битовое представление чисел типа* `SINGLE`

Номер бита	Описание
31	Бит знака (s)
30–23	Порядок (p)
22–0	Мантисса (m)

Значение числа вычисляется по формуле:

$$(-1) \times 2^{p-127} \times m.$$

Нулевое значение представлено четырьмя нулевыми байтами. Диапазон, задаваемый этим типом, приблизительно от 1E−38 до 1E+38.

Для портов существует только функция передачи байта, поэтому для передачи числа типа `SINGLE` необходимо реализовать функцию, последовательно передающую байты числа. Очевидно, что можно реализовать эту функцию двумя способами: либо посылая байты в порядке 0, 1, 2, 3, либо в порядке 3, 2, 1, 0. Ясно, что эти способы эквивалентны, важно лишь, чтобы приемник и передатчик работали по одинаковому протоколу *(см. прил. 8)*.

Приложение 4

Описание функций Windows для использования в Visual Basic

При необходимости применения описанных в данной книге функций на языке Visual Basic следует использовать их описания, приведенные в листинге П4.1.

Листинг П4.1. Описание функций коммуникационного порта для Visual Basic

```
Declare Function ReadFileEx Lib "kernel32" Alias "ReadFileEx" (ByVal
hFile As Long, lpBuffer As Any, ByVal nNumberOfBytesToRead As Long,
lpOverlapped As OVERLAPPED, ByVal lpCompletionRoutine As Long) As Long


Declare Function WriteFileEx Lib "kernel32" Alias "WriteFileEx" (ByVal
hFile As Long, lpBuffer As Any, ByVal nNumberOfBytesToWrite As Long,
lpOverlapped As OVERLAPPED, ByVal lpCompletionRoutine As Long) As Long


Declare Function BuildCommDCB Lib "kernel32" Alias "BuildCommDCBA" (ByVal
lpDef As String, lpDCB As DCB) As Long


Declare Function BuildCommDCBAndTimeouts Lib "kernel32" Alias
"BuildCommDCBAndTimeoutsA" (ByVal lpDef As String, lpDCB As DCB,
lpCommTimeouts As COMMTIMEOUTS) As Long


Declare Function SetCommBreak Lib "kernel32" Alias "SetCommBreak" (ByVal
nCid As Long) As Long


Declare Function ClearCommBreak Lib "kernel32" Alias "ClearCommBreak"
(ByVal nCid As Long) As Long


Declare Function ClearCommError Lib "kernel32" Alias "ClearCommError"
(ByVal hFile As Long, lpErrors As Long, lpStat As COMSTAT) As Long
```

```
Declare Function EscapeCommFunction Lib "kernel32" Alias
"EscapeCommFunction" (ByVal nCid As Long, ByVal nFunc As Long) As Long


Declare Function GetCommMask Lib "kernel32" Alias "GetCommMask" (ByVal
hFile As Long, lpEvtMask As Long) As Long


Declare Function CreateEvent Lib "kernel32" Alias "CreateEventA"
(lpEventAttributes As SECURITY_ATTRIBUTES, ByVal bManualReset As Long,
ByVal bInitialState As Long, ByVal lpName As String) As Long


Declare Function WaitCommEvent Lib "kernel32" Alias "WaitCommEvent"
(ByVal hFile As Long, lpEvtMask As Long, lpOverlapped As OVERLAPPED) As
Long


Declare Function GetCommConfig Lib "kernel32" Alias "GetCommConfig"
(ByVal hCommDev As Long, lpCC As COMMCONFIG, lpdwSize As Long) As Long


Declare Function SetCommConfig Lib "kernel32" Alias "SetCommConfig"
(ByVal hCommDev As Long, lpCC As COMMCONFIG, ByVal dwSize As Long) As
Long


Declare Function CommConfigDialog Lib "kernel32" Alias "CommConfigDia-
logA" (ByVal lpszName As String, ByVal hWnd As Long, lpCC As COMMCONFIG)
As Long


Declare Function GetCommProperties Lib "kernel32" Alias
"GetCommProperties" (ByVal hFile As Long, lpCommProp As COMMPROP) As Long


Declare Function GetCommState Lib "kernel32" Alias "GetCommState" (ByVal
nCid As Long, lpDCB As DCB) As Long


Declare Function SetCommState Lib "kernel32" Alias "SetCommState" (ByVal
hCommDev As Long, lpDCB As DCB) As Long


Declare Function GetCommTimeouts Lib "kernel32" Alias "GetCommTimeouts"
(ByVal hFile As Long, lpCommTimeouts As COMMTIMEOUTS) As Long


Declare Function SetCommTimeouts Lib "kernel32" Alias "SetCommTimeouts"
(ByVal hFile As Long, lpCommTimeouts As COMMTIMEOUTS) As Long


Declare Function PurgeComm Lib "kernel32" Alias "PurgeComm" (ByVal hFile
As Long, ByVal dwFlags As Long) As Long
```

```
Declare Function SetupComm Lib "kernel32" Alias "SetupComm" (ByVal hFile
As Long, ByVal dwInQueue As Long, ByVal dwOutQueue As Long) As Long


Declare Function GetDefaultCommConfig Lib "kernel32" Alias
"GetDefaultCommConfigA" (ByVal lpszName As String, lpCC As COMMCONFIG,
lpdwSize As Long) As Long


Declare Function SetDefaultCommConfig Lib "kernel32" Alias
"SetDefaultCommConfigA" (ByVal lpszName As String, lpCC As COMMCONFIG,
ByVal dwSize As Long) As Long


Declare Function TransmitCommChar Lib "kernel32" Alias "TransmitCommChar"
(ByVal nCid As Long, ByVal cChar As Byte) As Long
```

Приложение 5

Использование ActiveX-компонента *MSComm32*

В Windows включен специальный компонент `MSComm32.OCX`, позволяющий работать с коммуникационным портом из любого приложения, поддерживающего ActiveX.

> ***Важно***
>
> В Windows 95/98 вызов `MSComm.PortOpen` приводит к утечке памяти в размере примерно 4 Кбайт на 100 вызовов. Эта ошибка исправлена в Visual Studio 97 SP2.

В листинге П5.1 показана работа с этим компонентом на языке Visual Basic.

Листинг П5.1. Реализация передатчика с помощью MSComm32

```
' РЕАЛИЗАЦИЯ ПЕРЕДАТЧИКА (MSDN, Q158008)

' Основные описания

      Dim Offset As Long
      Dim FileData As Byte
      Dim FileName As String

' Для работы этой программы свойства компонента MSComm1
' должны быть установлены в следующие значения:
'      RThreshold = 1
'      SThreshold = 3

     ' обработчик Load для Form
     ' 28800 baud, no parity, 8 data, 1 stop bit.
     ' Для RS-232 максимальная скорость 28 800
```

```
    Private Sub Form_Load()

        ' Открыть порт
        MSComm1.Settings = "28800,N,8,1"
        MSComm1.InputLen = 1
        MSComm1.CommPort = 2
        MSComm1.PortOpen = True

        ' Имя файла для передачи
        FileName = "d:\sample.hlp"

        ' Открыть файл

        Offset = 1
        Open FileName For Binary Access Read As #1
    End Sub

    ' Обработчик события ONCOMM для Comm-компонента
    ' Свойство CommEvent принимает значение 2 при готовности
    ' передатчика к передаче следующего символа

    Private Sub MSComm1_OnComm()
        If MSComm1.CommEvent = 2 Then
            temp = MSComm1.Input
            If Offset <= FileLen(FileName) Then
                Get #1, Offset, FileData
                q = FileData
                MSComm1.Output = Format(q, "000")
                Offset = Offset + 1
            End If
        End If
    End Sub

    ' Обработчик UnLoad для Form — закрыть файл
    Private Sub Form_Unload(Cancel As Integer)
        Close #1
    End Sub

' РЕАЛИЗАЦИЯ ПРИЕМНИКА

    ' Основные описания
```

```
    Dim ByteCount As Long
    Dim FileData As Byte

' Для работы программы свойства MSComm1 должны быть изменены так:
' RThreshold = 3

    ' Обработчик Load для Form
    ' 28800 baud, no parity, 8 data, 1 stop bit.

    Private Sub Form_Load()
        MSComm1.Settings = "28800,N,8,1"
        MSComm1.InputLen = 3
        MSComm1.CommPort = 1
        MSComm1.PortOpen = True

        ' Открытие файла для сохранения полученных данных
        Open "c:\sample.txt" For Binary Access Write As #1
    End Sub

    ' Обработчик OnComm
    Private Sub MSComm1_OnComm()
        If MSComm1.CommEvent = 2 And MSComm1.InBufferCount > 0 Then
            FileData = CInt(MSComm1.Input)
            ByteCount = ByteCount + 1
            Put #1, ByteCount, FileData
            MSComm1.Output = Chr$(26)
        End If
    End Sub

   ' Событие CLICK по кнопке
    Private Sub Command1_Click()
        MSComm1.Output = Chr$(26)
    End Sub

    ' Обработчик UnLoad для Form
    Private Sub Form_Unload(Cancel As Integer)
        Close #1
    End Sub
```

Приложение 6

Сервис BIOS — обслуживание INT14H

В *гл. 6* и *18* мы использовали функции сервиса BIOS, а в листинге П6.1 мы приводим пример реализации самого сервиса.

Листинг П6.1. Реализация INT14H

```
;--INT 14-------------------------------------------------------------
; RS232_IO
;       ЭТА ПРОГРАММА ОБСЛУЖИВАЕТ RS232
;   00H инициализировать коммуникационный порт
;   01H послать символ через выбранный порт RS-232
;   02H получить символ через выбранный порт RS-232
;   03H дать статус порта связи
;----------------------------------------------------------------------
; COM1 и COM2 инициализирует BIOS.
; при инициализации заполняются следующие ячейки
;
; адрес регистра данных передатчика COM1
; 0:400  RS232_BASE[0]     DW  3F8H
; адрес регистра данных передатчика COM2
; 0:402  RS232_BASE[2]     DW  2F8H
; Время ожидания прихода символа для COM1
; 0:47C  RS232_TIM_OUT[0]  DB  01
; 0:47D  RS232_TIM_OUT[1]  DB  01
; 0:47E  RS232_TIM_OUT[2]  DB  01
; 0:47F  RS232_TIM_OUT[3]  DB  01
; Для инициализации COM3 и COM4 необходимо записать адрес в
;  0:404  RS232_BASE[4]  DW  ?
```

```
;  0:406  RS232_BASE[6]  DW  ?
; и использовать функцию  AH = 0
;--------------------------------------------------------------
  ASSUME  CS:CODE,DS:DATA
  ORG     0E729H
A1      LABEL   WORD      ; ТАБЛИЦА ЗНАЧЕНИЙ ДЛЯ ИНИЦИАЛИЗАЦИИ
  DW      1047            ; 110 БОД
  DW      768             ; 150
  DW      384             ; 300
  DW      192             ; 600
  DW      96              ; 1200
  DW      48              ; 2400
  DW      24              ; 4800
  DW      12              ; 9600

RS232_IO        PROC    FAR

;--  ВХОД В ПРЕРЫВАНИЕ
  STI
  PUSH    DS
  PUSH    DX
  PUSH    SI
  PUSH    DI
  PUSH    CX
  PUSH    BX
  MOV     SI,DX  ; В SI НОМЕР COM-ПОРТА ИСПОЛЬЗУЕТСЯ
                 ; ДЛЯ АДРЕСАЦИИ К ЯЧЕЙКАМ RS232_BASE
  MOV     DI,DX  ; DI ИСПОЛЬЗУЕТСЯ ДЛЯ АДРЕСАЦИИ К
                 ; RS232_TIM_OUT
  SHL     SI,1
  CALL    DDS
  MOV     DX,RS232_BASE[SI]       ; ПОЛУЧЕНИЕ БАЗОВОГО АДРЕСА
  OR      DX,DX                   ; ТЕСТ НЕ ИНИЦИАЛИЗИРОВАННОГО ПОРТА
  JZ      A3                      ; ВОЗВРАТ
  OR      AH,AH                   ; ТЕСТ ДЛЯ (AH)= 0
  JZ      A4                      ; ИНИЦИАЛИЗАЦИЯ
  DEC     AH                      ; ТЕСТ ДЛЯ (AH)= 1
  JZ      A5                      ; ПЕРЕСЫЛКА AL
```

```
  DEC     AH                    ; ТЕСТ ДЛЯ (AH)=2
  JZ      A12                   ; ЧТЕНИЕ В AL
A2:
  DEC     AH                    ; ТЕСТ ДЛЯ (AH)=3
  JNZ     A3
  JMP     A18                   ; ПОЛУЧЕНИЕ СТАТУСА
A3:                             ; ВЫХОД
  POP     BX
  POP     CX
  POP     DI
  POP     SI
  POP     DX
  POP     DS
  IRET

;-----  ИНИЦИАЛИЗАЦИЯ
A4:
  MOV     AH,AL       ; ПАРАМЕТРЫ ИНИЦИАЛИЗАЦИИ В AH
  ADD     DX,3        ; В DX - 3FBH (2FBH) АДРЕС РЕГИСТРА
                      ; УПРАВЛЕНИЯ ЛИНИЕЙ LCR
  MOV     AL,80H
  OUT     DX,AL       ; УСТАНОВКА DLAB=1 ДЛЯ ПОЛУЧЕНИЯ
                      ; ДОСТУПА К ФИКСАТОРАМ ДЕЛИТЕЛЯ

;-----  ОПРЕДЕЛЕНИЕ СКОРОСТИ ПЕРЕДАЧИ
  MOV     DL,AH       ; ВЫДЕЛЕНИЕ ИЗ БАЙТА ИНИЦИАЛИЗАЦИИ
  MOV     CL,4        ; СКОРОСТИ ПЕРЕДАЧИ.
  ROL     DL,CL                 ;
  AND     DX,0EH                ;
  MOV     DI,OFFSET A1          ;  ПОЛУЧЕНИЕ СМЕЩЕНИЯ ТАБЛИЦЫ ДЕЛИТЕЛЯ
  ADD     DI,DX                 ;
  MOV     DX,RS232_BASE[SI]     ;  ПОЛУЧЕНИЕ БАЗОВОГО АДРЕСА
  INC     DX                    ;  АДРЕС СТАРШЕГО БАЙТА РЕГИСТРА ДЕЛИТЕЛЯ
  MOV     AL,CS:[DI]+1          ;  ПОЛУЧЕНИЕ ВЕЛИЧИНЫ ДЕЛИТЕЛЯ
  OUT     DX,AL                 ;  ЗАПИСЬ СТАРШЕГО БАЙТА
  DEC     DX
  MOV     AL,CS:[DI]            ;
  OUT     DX,AL                 ;  ЗАПИСЬ МЛАДШЕГО БАЙТА
  ADD     DX,3                  ;  РЕГИСТР УПРАВЛЕНИЯ МОДЕМОМ
```

```
  MOV     AL,AH                   ;  ПАРАМЕТРЫ
  AND     AL,01FH                 ;  ОБНУЛЕНИЕ 7,6 И 5 БИТ
  OUT     DX,AL                   ;  ЗАПИСЬ ПАРАМЕТРОВ
  DEC     DX
  DEC     DX                      ; РЕГИСТР ИДЕНТИФИКАЦИИ ПРЕРЫВАНИЙ
  MOV     AL,0
  OUT     DX,AL                   ; ВСЕ ПРЕРЫВАНИЯ ВЫКЛЮЧЕНЫ
  JMP     SHORT A18               ; COM_STATUS

;-----  ПЕРЕСЫЛКА СИМВОЛА AH = 1
A5:
  PUSH    AX
  ADD     DX,4                    ; РЕГИСТР УПРАВЛЕНИЯ МОДЕМОМ
  MOV     AL,3                    ; ВЫХОД DTR И RTC В АКТИВНОМ СОСТОЯНИИ
  OUT     DX,AL
  INC     DX                      ; РЕГИСТР СОСТОЯНИЯ МОДЕМА
  INC     DX
  MOV     BH,30H                  ; ОЧИСТКА ДЛЯ ПЕРЕДАЧИ И ГОТОВНОСТЬ
                                  ; ПЕРЕДАЧИ
  CALL    WAIT_FOR_STATUS         ; ОЖИДАНИЕ
  JE      A9                      ; ДА, ПЕРЕДАЧА СИМВОЛА
A7:
  POP     CX
  MOV     AL,CL                   ; ПЕРЕЗАГРУЗКА БАЙТА ДАННЫХ
A8:
  OR      AH,80H                  ; ИНДИКАЦИЯ TIME OUT
  JMP     A3                      ; ВЫХОД
A9:                               ; CLEAR_TO_SEND
  DEC     DX                      ; РЕГИСТР СОСТОЯНИЯ ЛИНИИ
A10:                              ; WAIT_SEND
  MOV     BH,20H                  ; РЕГИСТР ДАННЫХ ПЕРЕДАТЧИКА ПУСТ
  CALL    WAIT_FOR_STATUS
  JNZ     A7                      ; ВОЗВРАТ С УСТАНОВЛЕННЫМ TIME OUT
A11:                              ; OUT_CHAR
  SUB     DX,5                    ; ПОРТ ДАННЫХ
  POP     CX                      ; ВОЗВРАЩЕНИЕ ДАННЫХ
  MOV     AL,CL                   ;
  OUT     DX,AL                   ; ВЫВОД СИМВОЛА
  JMP     A3                      ; ВЫХОД
```

```
;-----  ЧТЕНИЕ AH = 2
A12:
  ADD     DX,4              ; РЕГИСТР УПРАВЛЕНИЯ МОДЕМОМ
  MOV     AL,1              ; ПЕРЕВОД DTR В АКТИВНОЕ СОСТОЯНИЕ
  OUT     DX,AL
  INC     DX                ; РЕГИСТР СОСТОЯНИЯ МОДЕМА
  INC     DX
A13:                        ; WAIT_DSR
  MOV     BH,20H            ; ГОТОВНОСТЬ К ПЕРЕДАЧЕ ДАННЫХ
  CALL    WAIT_FOR_STATUS   ;
  JNZ     A8                ; ВОЗВРАТ С ОШИБКОЙ
A15:                        ; WAIT_DSR_END
  DEC     DX                ; РЕГИСТР СОСТОЯНИЯ ЛИНИИ
A16:                        ; WAIT_RECV
  MOV     BH,1              ; ИНДИКАТОР ГОТОВНОСТИ ДАННЫХ В ПРИЕМНИКЕ
  CALL    WAIT_FOR_STATUS
  JNZ     A8                ; TIME OUT ОШИБКА
A17:                        ; GET_CHAR
  AND     AL,00011110B       ; ТЕСТ ОШИБОЧНОГО СОСТОЯНИЯ
  MOV     DX,RS232_BASE[SI]  ; ПОРТ ДАННЫХ
  IN      AL,DX              ; ЧТЕНИЕ СИМВОЛА
  JMP     A3                 ; ВЫХОД

;-----  СТАТУС AH = 3
A18:
  MOV     DX,RS232_BASE[SI]
  ADD     DX,5                    ; РЕГИСТР СОСТОЯНИЯ ЛИНИИ
  IN      AL,DX
  MOV     AH,AL                   ; ПЕРЕВОД В AH ДЛЯ ВОЗВРАТА
  INC     DX                      ; РЕГИСТР СОСТОЯНИЯ МОДЕМА
  IN      AL,DX                   ;
  JMP     A3                      ; ВОЗВРАТ
;----------------------------------------
; ОЖИДАНИЕ ДЛЯ ПОЛУЧЕНИЯ СОСТОЯНИЯ      :
; ВХОД :                               :
;       BH=МАСКА                       :
;       DX=АДРЕС РЕГИСТРА СТАТУСА      :
; ВЫХОД:                               :
;       ZF УСТАНОВЛЕН = ВРЕМЯ ВЫШЛО    :
```

```
;       AH=СТАТУС                       :
;---------------------------------------
WAIT_FOR_STATUS PROC    NEAR
  MOV     BL,RS232_TIM_OUT[DI]  ; ЧТЕНИЕ СЧЕТЧИКА
WFS0:
  SUB     CX,CX
WFS1:
  IN      AL,DX         ; ПОЛУЧЕНИЕ СТАУСА
  MOV     AH,AL         ; СОХРАНЕНИЕ В AH
  AND     AL,BH         ; ВЫДИЛЕНИЕ БИТОВ ДЛЯ ТЕСТА
  CMP     AL,BH         ; СРАВНЕНИЕ С МАСКОЙ
  JE      WFS_END       ; ВЫХОД С СБРОШЕННЫМ ZF
  LOOP    WFS1          ; ПОВТОРИТЬ СНОВА
  DEC     BL
  JNZ     WFS0

  OR      BH,BH         ; УСТАНОВИТЬ ZF
WFS_END:
  RET
WAIT_FOR_STATUS ENDP
RS232_IO        ENDP
```

Приложение 7

Как работает GiveIO

Для объяснения принципа работы драйвера GiveIO нам придется сказать несколько слов о реализации защиты ввода/вывода (в/в) в Windows NT/2000/XP.

Основа защиты в/в — четырехуровневая система привилегий процессора 80x86. Наиболее привилегированный уровень — уровень 0, наименее привилегированный — уровень 3. Операционная система Windows использует только два крайних уровня:

- уровень 0 для режима ядра с полным доступом (full access kernel mode);
- уровень 3 для ограниченного режима пользователя (user mode).

Текущий уровень привилегий (CPL, Current Privilege Level) хранится в двух младших битах регистра `CS`.

Защита в/в имеет два уровня проверок.

В битах 12 и 13 регистра `EFLAGS` процессор хранит уровень привилегий, для которого разрешен прямой доступ к портам в/в, называемый IOPL (I/O Privilege Level). ОС Windows всегда устанавливает IOPL равным нулю, поэтому программы, выполняющиеся на уровне ядра (уровень 0), всегда имеют прямой доступ к портам в/в, а программы, выполняющиеся в режиме пользователя (уровень 3), проходят вторую стадию.

Вторая стадия проверки защиты реализуется с помощью *карты разрешения в/в* (IOPM, I/O Permission Map), представляющей собой битовый массив, каждый бит которого соответствует порту в/в. Если бит установлен в 1, доступ запрещен, и при доступе к соответствующему порту произойдет исключение. Если бит установлен в 0, то к данному порту предоставлен прямой и беспрепятственный доступ. Таблица IOPM хранится в структуре "сегмент состояния задачи" (TSS, Task State Segment) в основной памяти, в сегменте, на который ссылается селектор сегмента в регистре задачи (TR, Task Register) процессора. Смещение IOPM внутри TSS хранится в 2-байтовом числе по смещению 0x66 в TSS. Следует отметить, что хотя адресное пространство в/в процессоров семейства 80x86 включает 65 536 8-битовых

портов, таблица IOPM не обязательно должна содержать 8192 байта: массив всегда начинается с адреса 0, но есть возможность не предоставлять битовую маску для старших адресов. Любая, непредоставленная часть битового массива, считается равной 1, следовательно, доступ к этим портам будет запрещен.

В библиотеке функций, осуществляющих поддержку драйверов режима ядра (`NTOSKRNL`), содержатся три недокументированные функции.

- `Ke386SetIoAccessMap()` принимает 2 аргумента: параметр типа `Integer`, который нужно установить в 1, чтобы функция работала, и указатель на буфер. Функция копирует переданную карту доступа к в/в длиной 0x2000 байт из буфера в TSS по смещению 0x88.
- `Ke386QueryIoAccessMap()` принимает такие же аргументы, но делает противоположное, копируя текущую таблицу IOPM из TSS в буфер длиной 0x2000 байт.
- `Ke386IoSetAccessProcess()` принимает 2 аргумента: указатель на структуру процесса, полученный вызовом `PsGetCurrentProcess()`, и целое число, которое должно быть установлено в 1 для разрешения доступа к в/в, или в 0 для его запрещения. Когда целочисленный аргумент равен 0, функция запрещает доступ к в/в установкой смещения IOPM переданного процесса за границу сегмента TSS. Когда целочисленный аргумент равен 1, функция разрешает доступ к в/в, устанавливая смещение IOPM переданного процесса на начало IOPM по смещению 0x88 в TSS.

Используя эти функции, можно читать, изменять и записывать обратно таблицу IOPM, предоставляя доступ к нужным портам установкой соответствующих битов в 0.

Драйвер Робертса GiveIO, код которого приведен в листинге П7.1, устанавливает все биты таблицы IOPM в нули, открывая полный доступ к портам для режима пользователя. Следующий листинг (П7.1) — пробное приложение, называющееся TestIO. В нем используется прямой доступ к системному динамику для демонстрации прямого обращения к портам в/в. Следует отметить, что основной задачей драйвера является изменение таблицы IOPM, поэтому сам драйвер для работы не требуется. В тестовом примере драйвер выгружается сразу же после открытия.

Листинг П7.1. Драйвер GiveIO (Dale Roberts)

```
/*
GiveIO — создан Dale Roberts
Компиляция: Используйте средство DDK BUILD
Назначение: Предоставить доступ к прямому в/в для процесса режима
пользователя
*/
```

```
#include <ntddk.h>
#include <mondebug.h>

/* Имя нашего драйвера устройства */
#define DEVICE_NAME_STRING L"giveio"

/*
Это "структура" IOPM. Это просто массив байтов размером 0x2000. Он содержит 8К * 8 бит = 64 Кбит IOPM, которые покрывают все 64-килобайтное адресное пространство в/в x86 процессора. Каждый нулевой бит предоставляет доступ к соответствующему порту для user-mode-процесса. Каждый единичный бит запрещает доступ к в/в через соответствующий порт.
*/
#define IOPM_SIZE 0x2000

typedef UCHAR IOPM[IOPM_SIZE];

/*
Это будет содержать просто массив нулей, который будет копироваться в настоящую IOPM в TSS через Ke386SetIoAccessMap(). Память выделяется во время загрузки драйвера.
*/

IOPM *IOPM_local = 0;

/*
Это две недокументированные функции, которые мы используем, чтобы дать вызывающему процессу доступ к в/в вызывающему процессу. Ke386IoSetAccessMap() копирует переданную карту в/в в TSS. Ke386IoSetAccessProcess() изменяет указатель смещения IOPM, после чего только что скопированная карта в/в начинает использоваться. Иначе смещение IOPM указывает за предел сегмента TSS, что приведет к генерации исключения при попытке доступа к в/в любым user-mode-процессом.
*/

void Ke386SetIoAccessMap(int, IOPM *);
void Ke386QueryIoAccessMap(int, IOPM *);
void Ke386IoSetAccessProcess(PEPROCESS, int);

/*
Освободить все выделенные ранее объекты
*/
```

```
VOID GiveioUnload(IN PDRIVER_OBJECT DriverObject)
{
 WCHAR DOSNameBuffer[] = L"\\DosDevices\\" DEVICE_NAME_STRING;
 UNICODE_STRING uniDOSString;
 if(IOPM_local)
  MmFreeNonCachedMemory(IOPM_local, sizeof(IOPM));
 RtlInitUnicodeString(&uniDOSString, DOSNameBuffer);
 IoDeleteSymbolicLink (&uniDOSString);
 IoDeleteDevice(DriverObject->DeviceObject);
}

/*
Устанавливаем IOPM (карту разрешения в/в) вызывающего процесса так, что
ему предоставляется полный доступ к в/в. Наш массив OPM_local[] содержит
только нули, соответственно IOPM обнулится. Если OnFlag == 1, процессу
предоставляется доступ к в/в. Если он равен 0, доступ запрещается.
*/

VOID SetIOPermissionMap(int OnFlag)
{
 Ke386IoSetAccessProcess(PsGetCurrentProcess(), OnFlag);
 Ke386SetIoAccessMap(1, IOPM_local);
}

void GiveIO(void)
{
 SetIOPermissionMap(1);
}

/*
Служебный обработчик для user-mode-вызова CreateProcess(). Эта функция
введена в таблицу вызовов функций объекта драйвера с помощью
DriverEntry(). Когда user-mode-приложение вызывает CreateFile(), эта
функция получает управление все еще в контексте вызвавшего приложения, но
с CPL (текущий уровень привилегий процессора), установленным в 0. Это
позволяет производить операции, возможные только в kernel mode. GiveIO()
вызывается для предоставления вызывающему процессу доступа к в/в. Все что
приложение режима пользователя, которому нужен доступ к в/в, должно сде-
лать — это открыть данное устройство, используя CreateFile(). Никаких
других действий не нужно.
*/
```

```
NTSTATUS GiveioCreateDispatch(
 IN PDEVICE_OBJECT DeviceObject,
 IN PIRP Irp
)
{
 GiveIO(); // give the calling process I/O access
 Irp->IoStatus.Information = 0;
 Irp->IoStatus.Status = STATUS_SUCCESS;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);
 return STATUS_SUCCESS;
}

/*
Процедура входа драйвера. Эта процедура вызывается только раз после загрузки драйвера в память. Она выделяет необходимые ресурсы для работы драйвера. В нашем случае она выделяет память для массива IOPM и создает устройство, которое может открыть приложение режима пользователя. Она также создает символическую ссылку на драйвер устройства. Это позволяет user-mode-приложению получить доступ к нашему драйверу, используя \\.\giveio нотацию.
*/

NTSTATUS DriverEntry(
 IN PDRIVER_OBJECT DriverObject,
 IN PUNICODE_STRING RegistryPath
)
{
 PDEVICE_OBJECT deviceObject;
 NTSTATUS status;

 WCHAR NameBuffer[] = L"\\Device\\" DEVICE_NAME_STRING;
 WCHAR DOSNameBuffer[] = L"\\DosDevices\\" DEVICE_NAME_STRING;
 UNICODE_STRING uniNameString, uniDOSString;

 // Выделим буфер для локальной IOPM и обнулим его.
 IOPM_local = MmAllocateNonCachedMemory(sizeof(IOPM));
 if(IOPM_local == 0)
  return STATUS_INSUFFICIENT_RESOURCES;

 RtlZeroMemory(IOPM_local, sizeof(IOPM));
```

```
 // Инициализируем драйвер устройства и объект
 // устройства (device object)

 RtlInitUnicodeString(&uniNameString, NameBuffer);
 RtlInitUnicodeString(&uniDOSString, DOSNameBuffer);

 status = IoCreateDevice(DriverObject, 0, &uniNameString,
                  FILE_DEVICE_UNKNOWN, 0, FALSE, &deviceObject);
 if(!NT_SUCCESS(status))
  return status;

 status = IoCreateSymbolicLink (&uniDOSString, &uniNameString);
 if (!NT_SUCCESS(status))
  return status;

 // Инициализируем точки входа драйвера в объекте
 // драйвера. Все что нам нужно — это операции создания
 // (Create) и выгрузки (Unload)
 DriverObject->MajorFunction[IRP_MJ_CREATE] = GiveioCreateDispatch;
 DriverObject->DriverUnload = GiveioUnload;
 return STATUS_SUCCESS;
}
```

Листинг П7.2. Тест драйвера GiveIO (Dale Roberts)

```
/*
TSTIO.EXE — создан Dale Roberts
Компиляция: cl -DWIN32 tstio.c
Назначение: Для тестирования драйвера GiveIO производим какой-нибудь в/в.
Мы обращаемся к внутреннему динамику PC.
*/
#include <stdio.h>
#include <windows.h>
#include <math.h>
#include <conio.h>

typedef struct {
 short int pitch;
```

```
 short int duration;
} NOTE;

/* Таблица нот.*/
NOTE notes[] = {
  {14, 500}, {16, 500}, {12, 500},
  {0, 500}, {7, 1000}
};

/*
Установка частоты динамика PC в герцах. Динамик управляется таймером
Intel 8253/8254 с портами в/в 0x40-0x43.
*/

void setfreq(int hz)
{
 hz = 1193180 / hz; // базовая частота таймера 1,19 MHz
 _outp(0x43, 0xb6); // Выбор таймера 2, операция записи, режим 3.
 _outp(0x42, hz); // устанавливаем делитель частоты
 _outp(0x42, hz >> 8); // старший байт делителя
}

/*
Спикер управляется через порт 0x61. Установка двух младших битов разреша-
ет канал 2 таймера 8253/8254 и включает динамик.
*/
void playnote(NOTE note)
{
 _outp(0x61, _inp(0x61) | 0x03); // включаем динамик
 setfreq((int)(400 * pow(2, note.pitch / 12.0)));
 Sleep(note.duration);
 _outp(0x61, _inp(0x61) & ~0x03); // выключаем
}

/*
Открытие и закрытие устройства GiveIO. Это должно дать нам прямой доступ
к в/в. Потом пытаемся проиграть нашу мелодию.
*/
int main()
```

```
{
 int i;
 HANDLE h;

 h = CreateFile("\\\\.\\giveio", GENERIC_READ, 0, NULL,
      OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
 if(h == INVALID_HANDLE_VALUE) {
  printf("Couldn't access Giveio device\n");
  return(-1);
 }
 CloseHandle(h);

 for(i=0; i < sizeof(notes)/sizeof(int); ++i)
  playnote(notes[i]);
 return 0;
}
```

Приложение 8

Реализация процедуры преобразования SINGLE-чисел в строку

Здесь мы приведем пример кода, реализующий ASCII-протокол для чисел с плавающей точкой типа `SINGLE`. Код написан на языке C, без применения системных библиотек, и может использоваться для микропроцессоров 8051/8052. Хотя, конечно, можно воспользоваться функцией `sprintf`, но ее выполнение потребует подключение библиотеки, добавляющей 3—4 Кбайт кода. Для микропроцессоров это совершенно неприемлемый вариант.

ASCII-протокол, реализованный в листинге П8.1, имеет формат:

- ❒ знак числа (символ минус или пробел, если знака нет);
- ❒ 6 цифр числа, включая символ десятичного разделителя;
- ❒ байты завершения посылки (коды 13, 10).

Для сравнения во второй половине кода приводится реализация того же протокола в бинарном виде. Переключение протоколов производится с помощью переменной `bProtocolMode`. Бинарный протокол обмена имеет вид:

- ❒ байт начала посылки (код 36);
- ❒ четыре байта числа с плавающей точкой (от младшего к старшему);
- ❒ байт конца посылки (код 13);
- ❒ контрольная сумма.

Еще раз заметим, что бинарный протокол требует значительно меньше байтов для передачи той же информации: если строковый протокол требует (1 + 6 + 2) = 9 байт, то бинарный всего (1 + 4 + 1 + 1) = 7 байт. Кроме того, бинарный протокол использует контрольную сумму для повышения достоверности данных. Посылка контрольной суммы для строкового протокола потребовала бы еще 3 байта.

Листинг П8.1. Передача числа с плавающей точкой в ASCII-протоколе

```
/*
Программа передачи числа с плавающей точкой строкой или байтами
(c) Агуров П.В.
*/

/* тип для преобразования SINGLE в байты */
typedef union{
 byte   b[4];
 float  f;
} UFLOAT;



/* Режимы посылки байта                                    */
/*   RS_START — обнулить контрольную сумму                 */
/*   RS_WORK  — засчитать в контрольную сумму              */
/*   RS_END   — ничего не делать с контрольной суммой      */
#define RS_START 0
#define RS_WORK  1
#define RS_END   2

/* ============================== */
/* Посылка одного байта           */
/* Параметры:                     */
/*   mode — режим посылки байта   */
/*   b    — байт для посылки      */
/* ============================== */
void SendByte(byte mode, byte b){
  data uint i;

  if (mode == RS_START) {
   RsCS = 0;
  } else {
   if (mode == RS_WORK) {
    /* Подсчет контрольной суммы — байтовая сумма */
    /* с переполнением                            */
    RsCS += b;
   }
  }
```

```
  TI = 0;
  i = 0; do { i++; if (i>350) return; } while (TI);
  SBUF = b;
  i = 0; do { i++; if (i>350) return; } while (!TI);
}

data single Ves; /* значение веса */
data byte   iPoint; /* точность чисел — позиция десятичной точки */
data bool   bProtocolMode; /* тип протокола */

/* Реализация протокола */
void SendRS232(){
 data byte i;
 data long l;
 data byte RSBuf[6];

[1]
 f_tmp.f = Ves; /* значение веса */

 if (bProtocolMode){
/* ========================== */
/* ASCII-протокол             */
/*    знак или пробел         */
/*    6 цифр числа            */
/*    конец посылки 0D, 0A    */
/* ========================== */
[2]
  SendByte(RS_WORK, (f_tmp.f < 0)?0x2D:0x20); /* знак "-" или пробел */
[3]
  if (f_tmp.f < 0) /* модуль float */
   f_tmp.f = -f_tmp.f;
[4]
  for (i=0; i< iPoint; i++) /* точность */
   f_tmp.f = f_tmp.f * 10.0;
[5]
  for (i= 0; i < 7; i++) /* чистка буфера пробелами */
   RSBuf[i] = 0x20;
```

```
[6]
  for (i= 0; i < 7; i++){
   if ((iPoint != i) || (iPoint == 0)) {
    l = (unsigned long)f_tmp.f/10;
    RSBuf[i] = (byte)((unsigned long)f_tmp.f - (unsigned long)l * 10.0);
    RSBuf[i]+= 0x30; /* переводим из байта в десятичную цифру */
    f_tmp.f = l;
   } else {
    RSBuf[i] = 0x2E; /* точка */
   }
  }
[7]
  for (i= 0; i < 7; i++) /* посылаем число */
   SendByte(RS_WORK, RSBuf[6 - i]);
[8]
  /* символы конца посылки 0D, 0A*/
  SendByte(RS_WORK, 13);
  SendByte(RS_WORK, 10);
 } else {
/* ========================== */
/* Бинарный протокол работы : */
/*   начало посылки '$'       */
/*   число single             */
/*   конец посылки LR         */
/*   контрольная сумма        */
/* ========================== */
    SendByte(RS_START, 36        );
    SendByte(RS_WORK , f_tmp.b[0]);
    SendByte(RS_WORK , f_tmp.b[1]);
    SendByte(RS_WORK , f_tmp.b[2]);
    SendByte(RS_WORK , f_tmp.b[3]);
    SendByte(RS_END  , 13        );
    SendByte(RS_END  , RsCS      );
   }
  }
}
```

Как видно из кода, перевод числа с плавающей точкой, записанного в переменную Ves, производится с помощью следующей последовательности

действий (номера пунктов списка отвечают номерам блоков в листинге П8.1):

1. Переписываем число в буфер `ftmp`. Специально описанный тип `UFLOAT` позволяет нам получить доступ как к самому числу `ftmp.f`, так и к байтам `ftmp.b[0]`, `ftmp.b[1]`, `ftmp.b[2]`, `ftmp.b[3]`.
2. В зависимости от результата выражения (`Ves < 0`) посылается либо символ минус (код 0x2D), либо символ пробел (0x20).
3. Удаляем знак числа (стараемся обойтись без системной функции `abs()`, требующей подключения библиотеки).
4. Делаем предварительные операции, необходимые для получения строки из числа; т. к. нам нужно строго ограниченное число знаков после точки, все лишние знаки мы "отрежем".
5. Чистим буфер передачи пробелами.
6. Самый сложный цикл, который, собственно, и производит преобразование числа в строку (разбирается ниже).
7. Передаем байты полученной строки.
8. Передаем байты завершения посылки.

Разберем цикл преобразования более подробно. В блоке [4] программы, приведенной в листинге П8.1, мы умножили передаваемое число на 10 ровно столько раз, сколько знаков нам нужно получить после запятой. В результате мы нашли число, округлив которое, мы получим исходное число с нужным нам числом знаков точности, но без точки:

3,1415926 *с точностью 3 даст нам целое число* $3{,}141 \times 1000 = 3141$.

Теперь наша задача последовательно "отрезать" цифры числа. Получить самую последнюю цифру легко — достаточно нацело поделить число на 10 и взять остаток от деления. Остаток мы найдем, умножив результат на 10 и вычитая это произведение из исходного числа:

$3141 / 10 = 314$, *остаток* $3141 - 314 \times 10 = 3141 - 3140 = 1$.

Теперь в качестве исходного числа мы возьмем 314 и снова произведем те же действия:

$314 / 10 = 31$, *остаток* $314 - 31 \times 10 = 314 - 310 = 4$;

$31 / 10 = 3$, *остаток* $31 - 3 \times 10 = 1$;

$3 / 10$ *дает остаток* 3.

Ясно, что проделать эту операцию нам надо столько раз, сколько цифр нам надо получить.

Собирая вычисленные остатки в буфер, мы получаем строковое представление числа. Проверка (`iPoint<>i`) позволяет нам поставить десятичную точку на свое законное место.

Приложение 9

Индикаторы состояния модема

В табл. П9.1 приводится описание индикаторов модема. Эта информация окажется полезной при написании программы, работающей с модемом или при отладке программ управления линиями модема.

Таблица П9.1. *Индикаторы состояния модема*

Линия	Описание
AA	Индикатор AA (Auto Answer) загорается, когда модем установлен в режим автоматического ответа на входящий звонок
TR	Индикатор TR (Data Terminal Ready) загорается, когда модем получает сигнал DTR. Драйвер модема устанавливает этот сигнал в состояние ON, когда компьютер готов к приему или передаче данных. Данный индикатор не светится при работе с компьютером Macintosh, который использует сигнал DTR для другой цели
OH	Индикатор OH (Off Hook) светится, когда модем "поднимает трубку" т. е. подключается к телефонной линии. (Он может мигать во время импульсного набора номера.)
CD	Индикатор CD (Carrier Detect) светится, когда модем обнаруживает несущую удаленного модема
HS	Индикатор HS (High Speed) загорается при работе модема с "высокой скоростью". В моделях IDC-1414 и IDC-1914 это означает любую скорость, превышающую 1200 бод
TD(SD)	Индикатор TD (Transmit Data) мигает всякий раз, когда происходит передача данных от терминала к модему. При непрерывной передаче данных на высоких скоростях этот индикатор может светиться почти постоянно
RD	Индикатор RD (Receive Data) мигает, когда модем передает данные терминалу. При непрерывной передаче данных на высоких скоростях этот индикатор может светиться почти постоянно
MR	Индикатор MR (Modem Ready) светится постоянно, если модем исправен и включен в сеть. Индикатор может мигать, когда модем выполняет самотестирование

Приложение 10

Инструменты

При написании книги мы попробовали множество программ для работы с последовательными портами, часть из которых представляем вашему вниманию.

Порт-монитор

Лучшей программой мониторинга обращений к порту мы считаем программу Марка Руссановича (Mark Russinovich) Portmon (рис. П10.1). На его сайте **http://www.sysinternals.com** свободно доступны две версии программы: для Windows 98 и для Windows NT/2000/XP.

Программа Portmon позволяет отслеживать обращения к последовательному порту как локальной, так и удаленной машины, отображать исполняемые команды и время их выполнения. Это действительно ценный инструмент!

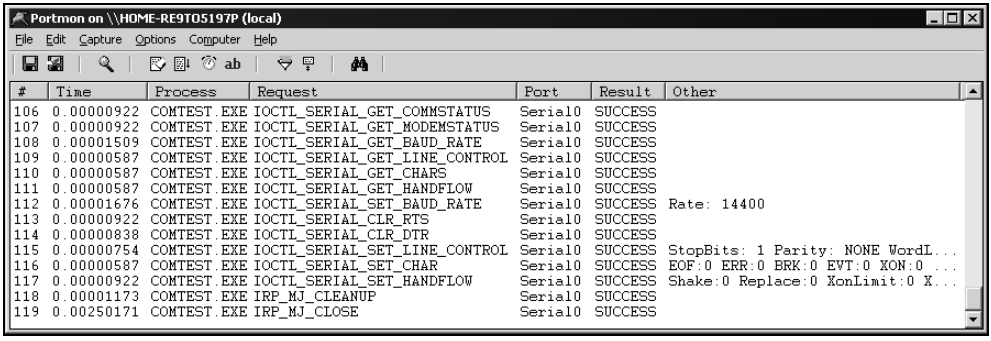

Рис. П10.1. Программа Portmon Марка Руссановича

Разработка драйверов Driver Wizard

Довольно интересная программа (рис. П10.2). Ее отличие от остальных заключается в возможности генерации кода для Delphi. Естественно, Delphi-код работает с помощью специального VxD-драйвера. Сайт разработчиков — **http://www.jungo.com**.

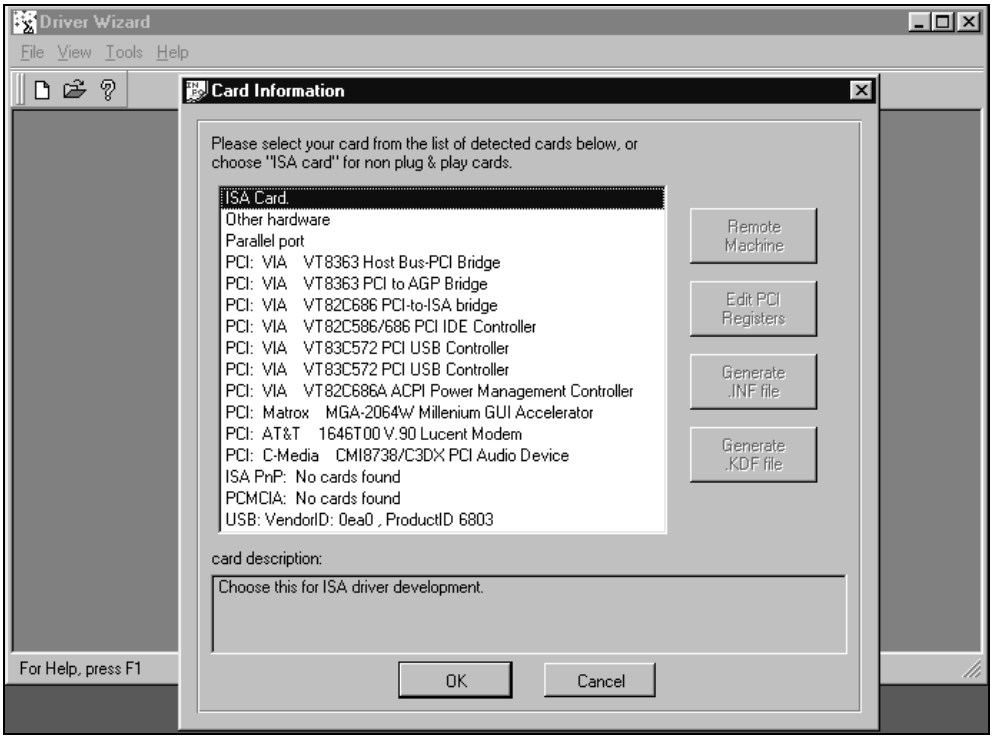

Рис. П10.2. Программа Driver Wizard

Разработка драйверов NuMega Driver Studio

Специальный модуль, подключающийся к MS Visual Studio. Позволяет создать драйвер, последовательно отвечая на вопросы помощника. Пожалуй, лучшая программа для разработки драйверов. Сайт — **http://www.numega.com**.

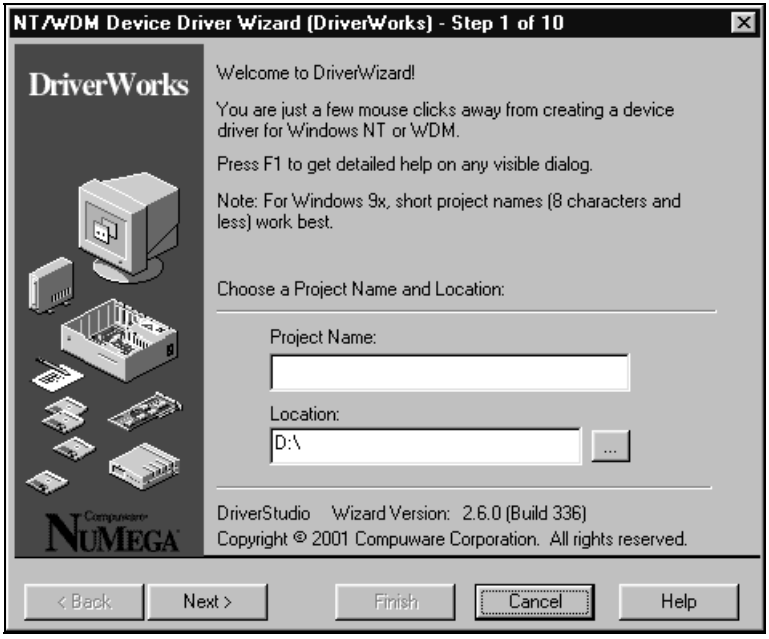

Рис. П10.3. Программа NuMega Driver Studio

Отладчик NuMega SoftICE

Мы думаем, эта программа не нуждается в дополнительном представлении. Можно сказать, что этот отладчик драйверов, и вообще любых программ, вне конкуренции.

Эмулятор компьютера Connectix Virtual PC

Полноценный эмулятор персонального компьютера (рис. П10.4). Позволяет тонко настроить ресурсы эмулируемого компьютера, например, COM-порт можно настроить на ввод/вывод из файла. Более того, можно связать две системы, запущенные в эмуляторе по коммуникационному порту.

Терминал HyperTerminal

Небольшой терминал, встроенный в Windows (рис. П10.5). Довольно неудобно производится конфигурирование настроек соединения, не запоминает имена передаваемых файлов, но пользоваться можно.

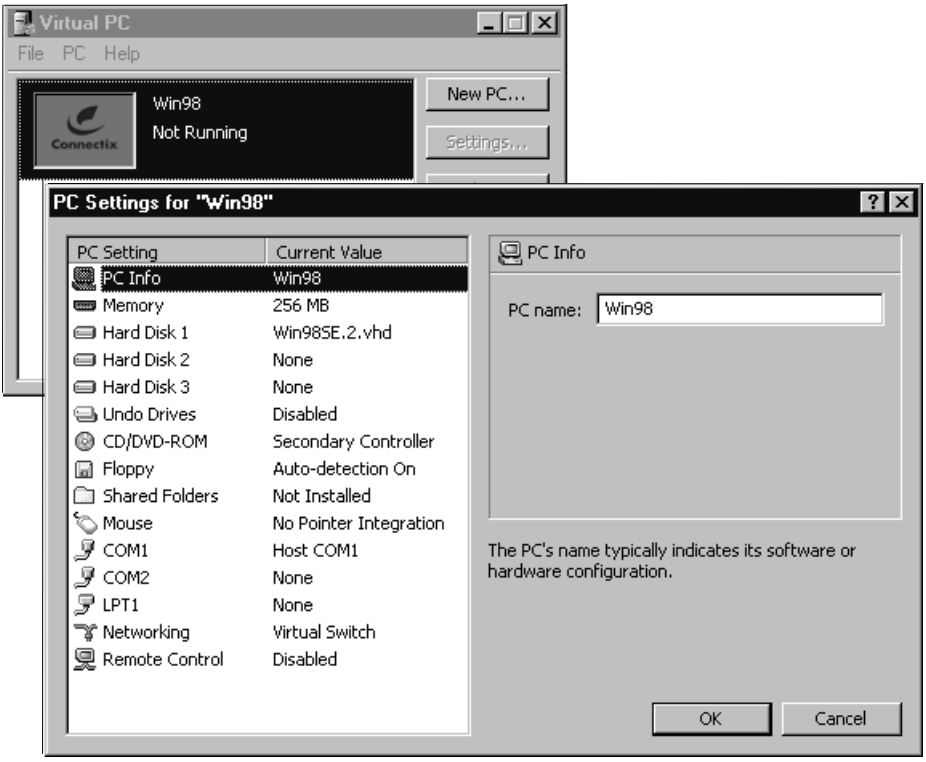

Рис. П10.4. Программа Virtual PC

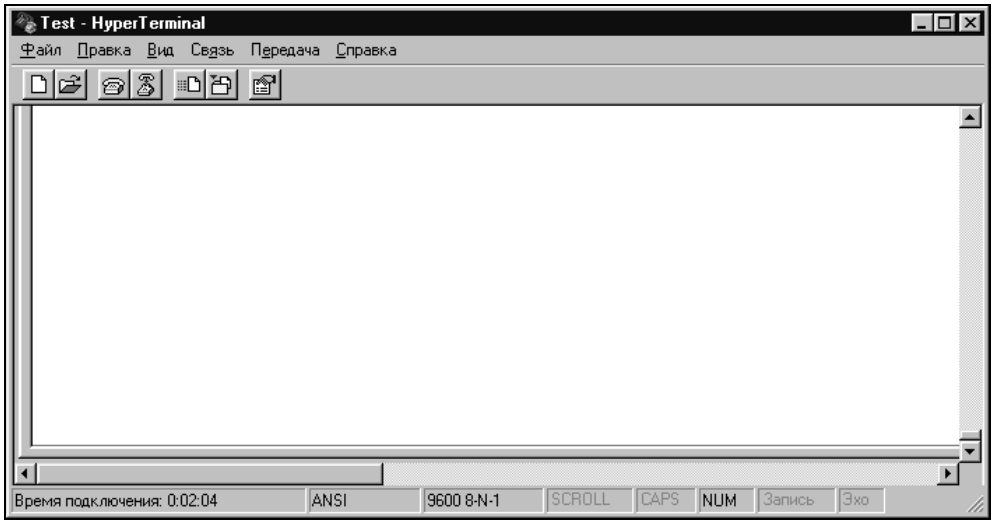

Рис. П10.5. Программа HyperTerminal

Приложение 11

Формат команды *Mode*

Для установки параметров коммуникационного порта, например COM1 или COM2, используется команда следующего формата:

```
mode comm[:] [baud=b] [parity=p] [data=d] [stop=s] [to=on|off]
[xon=on|off] [odsr=on|off] [octs=on|off] [dtr=on|off|hs]
[rts=on|off|hs|tg] [idsr=on|off]
```

Параметры этой команды следующие.

- `COMm` задает номер порта, например COM1 или COM2.
- `baud = b`. Может устанавливаться скорость 110, 150, 300, 600, 1200, 2400, 4800, 9600 или 19200 бит/с. Указывать нужно только 2 цифры, т. е. `baud=19` будет означать выбор скорости 19200.
- `parity = p`. Задает режим контроля четности. Параметр `p` может принимать следующие значения:
  - `n` (нет контроля четности);
  - `o` (нечетный);
  - `e` (четный, по умолчанию);
  - `m` (фиксированная четность MARK);
  - `s` (фиксированная четность SPACE).
- `data = d`. Указывает число битов в байте. Параметр `d` может принимать значения от 5 до 8 (по умолчанию 7). Не все компьютеры поддерживают значения 5 и 6.
- `stop = s`. Определяет число стоп-битов. Параметр `s` может принимать значения:
  - 1 (по умолчанию);
  - 1,5;
  - 2 (по умолчанию при скорости 110).
- `to = on|off`. Устанавливает бесконечный тайм-аут (если `on`).
- `xon = on|off`. Включает использование протокола `XON/XOFF`.

- `odsr = on|off`. Включает режим DSR-квитирования (handshaking DSR).
- `octs = on|off`. Включает режим CTS-квитирования (handshaking CTS).
- `dtr = on|off|hs`. Управляет линией DTR, включая ее (`on`), выключая (`off`) или устанавливая в режим квитирования (`hs`).
- `rts = on|off|hs`. Управляет линией RTS, включая ее (`on`), выключая (`off`) или устанавливая в режим квитирования (`hs`).
- `idsr = on|off`. Управляет линией DSR.

Настройки порта сохраняются только в текущей сессии Windows, т. е. после перезагрузки будут снова установлены параметры по умолчанию:

- скорость 1200;
- контроль четности выключен;
- число битов данных 7;
- число стоп-битов 1.

То же самое происходит при любых изменениях в Менеджере устройств (Device Manager). Для исправления первой ситуации рекомендуется создать простой ярлык с параметрами **Минимизировать при запуске** (Minimized) и **Закрывать после выполнения** (Close window on exit), содержащий строку изменения параметров, например:

```
C:\winnt\system32\mode.com com1: 9600,n,8,1
```

Настройки порта принято записывать в сокращенной форме, например:

- `8N1` означает: 8 битов данных, нет четности, 1 стоп-бит;
- `7E1` означает: 7 битов данных, четная (Even) четность, 1 стоп-бит.

Приложение 12

Часто задаваемые вопросы (FAQ)

FAQ: Общие вопросы

1. Что такое COM и RS и чем они отличаются?

Ответ: COM — сокращение от слова Communication — последовательный. COM-порт означает последовательный порт. RS — название стандарта последовательного обмена. Настоящее название стандарта "EIA/TIA-232-E", а аббревиатура "RS" расшифровывается как "Recommended Standard". Подробности описаны в *гл. 1.*

2. Что такое порт?

Ответ: слово "порт" применяется в разных смыслах. Во-первых, это может быть просто разъем компьютера, например — "подключить к COM-порту нуль-модемный кабель". Во-вторых, это может быть некий виртуальный адрес, записывая или считывая значение которого, можно получать определенную информацию. Эти порты адресуются с помощью операторов `Port` и `PortW`. Например, "записать в порт $3FB число $80".

3. Чем "протокол RS-232" отличается от "интерфейса RS-232"?

Ответ: так же как с понятием "порт", тут происходит некоторая путаница. Для связи двух устройств с помощью последовательной передачи данных используются две составляющие: аппаратная и программная. Аппаратная составляющая описывается стандартом RS-232 (или родственными стандартами типа RS-485, RS-422): уровни напряжений, разъемы, назначение сигналов и т. п. Программная реализация зависит от конкретного устройства, но условно ее тоже можно поделить на две части: системную (установка параметров связи, инициализация и т. д.) и пользовательскую (последовательность передачи данных). Часто системную часть называют протоколом RS-232.

4. Что такое контрольная сумма и как ее считать?

Ответ: контрольная сумма — дополнительные байты, добавляемые к пакету данных, позволяющие клиенту получить информацию о достоверности данных. Чаще всего это просто сумма всех байтов данных. Подробности см. в *разд. 2.4*, *гл. 14* и *15*.

5. Откуда взялось ограничение числа абонентов в протоколе RS-485?

Ответ: "Допустимая нагрузка драйвера RS-485/RS-422 количественно определяется в терминах единичной нагрузки, которая, в свою очередь, определяется как входной импеданс одного стандартного приемника RS-485 (12 кОм). Таким образом, стандартный драйвер RS-485 может управлять 32 единичными нагрузками (32 параллельных 12-килоомных нагрузки). Однако для некоторых приемников RS-485 входное сопротивление является более высоким — 48 кОм (1/4 единичной нагрузки) или даже 96 кОм (1/8 единичной нагрузки) — и, соответственно, к одной шине могут быть подключены сразу 128 или 256 таких приемников. Вы можете подключить любую комбинацию типов приемников, если их параллельный импеданс не превышает 32 единичных нагрузки (т. е. суммарное сопротивление не меньше 375 Ом)" [3].

6. Почему нужно подключать COM-устройства при выключенном питании?

Ответ: естественно, подключение устройств при выключенном питании касается только устройств с автономным питанием. Для подключения устройств типа мыши выключение питания не требуется (другое дело, что Windows или драйвер DOS может ее опознать только после перезагрузки). А вот устройства с независимым питанием могут создать разность потенциалов, которая в момент подключения может оказаться приложенной к выходным или, еще хуже, входным цепям интерфейса и вывести из строя микросхемы UART. На современных материнских платах микросхемы устанавливаются без панелек, и для замены их нужно выпаивать из платы. Такая операция возможна далеко не всегда.

7. Откуда берется питание для мыши?

Закономерный вопрос. Ответ: стандарт последовательной связи никак не определяет питание устройств. Для мыши вопрос решен просто: питание берется от неиспользуемых сигнальных цепей, установленных в состояние 1.

Некоторые мыши переключают режим работы в зависимости от установленных питающих сигналов.

8. Что означает режим *HANDSHAKING*?

Ответ: `HANDSHAKING` (квитирование установления связи) — это метод, с помощью которого компьютеры и другие устройства обмениваются статусной информацией. В стандарте RS-232, сигналы квитирования (handshaking signals) используются одним из связанных устройств для того, чтобы сообщить другому о своей готовности к приему данных. Аппаратное квитирование — когда эти сигналы передаются по специальным каналам (handshaking wires). Программное квитирование — когда статусная информация передается через специальные информационные байты в общем потоке данных.

9. Что такое дуплексная и полудуплексная передача?

Ответ: полный дуплекс (Full Duplex) подразумевает, что компьютер может одновременно передавать и принимать данные — существует два раздельных канала данных (один для входящих данных и один для исходящих).

Полудуплекс (Half Duplex) означает, что компьютер не может одновременно передавать и принимать данные. Обычно это подразумевает, что существует только один канал передачи данных. Однако это не значит, что любой из сигналов RS-232 не используется. Скорее всего, коммуникационная линия использует какой-нибудь другой стандарт, отличный от RS-232, который не поддерживает полнодуплексную работу.

FAQ: Программирование в DOS

1. Что такое базовый адрес порта?

Ответ: базовый адрес порта — 2-байтовое число, относительно которого считаются все остальные порты последовательной связи. Например, базовый адрес COM1 равен $3FB, соответственно адрес порта IER будет $3FB + 1 = $3FC.

2. Написал программу обработки прерываний порта, а она периодически "виснет". Для других прерываний все нормально работает

Ответ: скорее всего, вы забыли послать сигнал `EOI` контроллеру прерываний. Прерывания последовательного порта представляют собой аппаратные

прерывания (IRQ). Для их корректной обработки применяются специальные методы, описанные в *гл. 8.*

3. При выводе на экран данных, полученных в обработчике прерывания, выводится "мусор"

Ответ: вывод на экран использует прерывание INT21H для вывода символа, позиционирования курсора и т. д. Использование прерываний внутри аппаратного прерывания не очень корректно. Кроме того, прерывания последовательного порта являются аппаратными прерываниями и имеют наивысший приоритет. Длительная задержка внутри обработчика прерывания может привести к сбою системы. Мы рекомендуем убрать отображение данных из обработчика прерываний, а в обработчике только лишь проверять их достоверность и складывать в буфер. Печать и использование данных лучше делать в основном цикле программы.

4. Прерывание и основная программа используют один буфер. Как их синхронизировать в DOS?

Ответ: конечно, в DOS нет методов синхронизации, принятых в Windows. Поэтому перед выборкой данных из буфера нужно запретить прерывания, быстро скопировать данные в локальный буфер и снова разрешить прерывания. Это позволит избежать конфликтов, если основная программа начала забирать данные из буфера, забрала их до середины, а в это время обработчик прерывания их поменял. При этом следует запретить не только маскируемые, но и аппаратные прерывания соответствующего IRQ. *См. разд. 2.4 и гл. 8.*

5. Как бы перехватить вывод программы в COM-порт под DOS?

Ответ: если программа использует INT14H или INT21H, то достаточно просто перехватить обработчик прерывания. Но, вообще говоря, на это лучше не рассчитывать. Ограниченность сервисов BIOS приводит к необходимости программирования портов напрямую, без использования системных функций. В этом случае проще всего воспользоваться эмулятором PC, например VmWare. Хороший эмулятор умеет перенаправлять вывод в файл или пайп. Найти VmWare можно на сайте **http://www.vmware.com**.

FAQ: Программирование в Windows

1. Почему в Delphi пропал оператор *Port* и как тогда обращаться к портам?

Ответ: в Delphi 1 этот оператор еще присутствовал, а начиная с версии 3 — удален. Дело в том, что Windows не позволяет прямого обращения к портам. Заменой может служить класс `TPort`, описанный в *гл. 9*.

2. Можно ли использовать прямое обращение к портам в Windows NT/2000?

Ответ: нет. Для доступа к портам нужен специальный VxD-драйвер. Для Delphi существует модуль `TGiveIO`, подключение которого делает возможным прямое обращение к портам. Этот модуль содержит в себе бинарный образ VxD-драйвера и загружает его в память при запуске программы. *См. разд. 9.3*.

3. Программа асинхронно работает с портом. В Windows 98 все работало, а в Windows 2000 или "виснет", или не работает

Ответ: проверьте, что порт открыт с помощью флага `FILE_FLAG_OVERLAPPED`. Windows 98 терпимо относится к смешиванию синхронного открытия и асинхронных операций, и наоборот. А вот Windows 2000 такого произвола не допускает. Убедитесь, что дескриптор асинхронной структуры `OVERLAPPED` создан и используется корректно.

4. Зачем надо использовать *WaitForSingleObject*, если уже есть вызов *GetOverlappedResult*?

Ответ: вызов `GetOverlappedResult` действительно дождется завершения асинхронной операции, но прервать это ожидание или задать время ожидания невозможно. Вызов `WaitSingleObject` позволяет более гибко управлять ожиданием завершения. А использование `WaitForMuitipleObjects` позволяет вводить дополнительные сигналы завершения ожидания.

5. Как найти все доступные COM-порты?

Ответ: во многих источниках приводится код перебора имен портов от 1 до 32 и попытки их открытия. Так делать не рекомендуется. Во-первых, если

порт занят, он не будет распознан, а во-вторых, максимальное число портов зависит от версии Windows. Для перебора портов используйте системную функцию `EnumPorts`, применение которой показано в *гл. 13*.

6. Как бы перехватить вывод программы в COM-порт под Windows?

Ответ: проще всего воспользоваться эмулятором PC — см. ответ № 5 из FAQ для DOS. Иначе придется написать свой драйвер COM-порта, например, с помощью Numega Driver Studio или Jungo Windriver.

Если нужно перехватить не вывод программы, а команды, передаваемые программой (т. е. открытие/закрытие порта, программирование режимов порта и т. д.), то можно воспользоваться утилитой PortMon, созданной Марком Руссоновичем (Mark Russinovich) с его сайта **http://www.sysinternals.com**.

7. Как выключить Plug and Play для COM-порта при загрузке Windows?

Ответ: если COM-портов на вашем компьютере нет, или к ним подключено нестандартное оборудование, стоит отключить автоопределение устройств, чтобы "не смущать" операционную систему.

В Windows 2000/XP можно указать параметр

```
/fastdetect = COM1,2,3
```

в файле загрузки Boot.ini, указывающий Windows пропустить поиск устройств на заданных портах. Если номера не указаны, поиск отключается на всех COM-портах.

В Windows NT этот параметр назывался `/noserialmice`.

FAQ: Связь, модемы, терминалы

1. Что такое FIFO и FOSSIL?

Ответ: аппаратно реализованная очередь (FIFO, First In First Out) в аппаратной части последовательного порта. Нормально реализованным FIFO считается UART 16550A, 16550AF или 16550AFN. В некоторых моделях внутренних (internal) модемов аппаратно установлена микросхема UART 16550AFN. А FOSSIL — это драйвер COM-порта, ориентированный на работу с модемом, подключенным к COM-порту. Существует много разнообразных версий и типов FOSSIL-драйверов. Наибольшее распространение получили BNU и X00.

Драйвер FOSSIL (сокращение от Fido/Opus/Seadog/Standart/Layer) занимается в основном буферизацией потоков из (от) модема. Он перехватывает аппаратное прерывание от COM-порта, которое возникает по приходу (уходу) каждого байта или возникновению ошибок. На большой скорости обмена байты приходят быстро, и, соответственно, прерывание вызывается часто и времени на считывание пришедшего байта из регистров UART остается мало. Драйвер может и не успеть считать байт, и он "потеряется", а это, в свою очередь, приведет к ошибке контрольной суммы (CRC Error) в целом блоке данных.

FIFO позволяет организовать все это несколько иначе — приходящие байты будут копиться во внутреннем буфере (обычно до 14—16 байт). При заполнении буфера генерируется прерывание, и драйвер FOSSIL считывает весь буфер целиком за один раз. При этом время реакции драйвера FOSSIL на прерывание уже не так критично — приходящие байты будут складироваться в тот же внутренний буфер.

Приложение 13

Описание компакт-диска

Папки	Описание	Разделы
\GLAVA06\2	Программа приема и передачи данных с использованием сервиса BIOS INT14H	6.2
\GLAVA07\2	Определение наличия COM-портов и их базовых адресов в DOS	7.2
\GLAVA07\3-4	Прямое программирование коммуникационных портов в DOS	7.3, 7.4
\GLAVA08\2-3	Прием и передача данных с помощью прерываний коммуникационного порта в DOS	8.2, 8.3
\GLAVA09\1	Класс `TPort` для обращения к портам из Windows	9.1
\GLAVA09\2	Пример использования класса `TPort` для обращения к портам из Windows	9.2
\GLAVA09\3	Драйвер GiveIOEx для прямого обращения к портам из Windows NT/2000/XP и пример его использования. Для компиляции драйвера необходимо наличие Windows 2000 DDK	9.3
\GLAVA10\4	Пример последовательной связи с использованием функций Windows	10.4
\GLAVA11\2	Пример использования потоков для последовательной связи в Windows	11.2
\GLAVA12\3	Пример последовательной связи из Windows с помощью асинхронных функций	12.3
\GLAVA13\1	Дополнительные коммуникационные функции: управление линиями DTR/RTS и определение базового адреса порта (Windows 98)	13.1
\GLAVA13\2	Прямое управление драйвером коммуникационного порта с помощью функции `DeviceIoControl`	13.2
\GLAVA13\3	Обнаружение всех последовательных портов в системе	13.3

(окончание)

Папки	Описание	Разделы
\GLAVA13\4	Работа с именами портов (Windows NT/2000/XP)	13.4
\GLAVA13\5	Использование APC-функций Windows 2000	13.5
\GLAVA14\1-2	Прием и передача простого пакета данных (строка). Реализация компонента `TComPort` и редактора свойств	14.1, 14.2
\GLAVA14\3	Реализация ASCII-протокола	14.3
\GLAVA14\4	Реализация бинарного протокола	14.4
\GLAVA16\2	Тестирование свойств COM-порта	16.2
\GLAVA16\3	Реализация протокола RS-485	16.3
\GLAVA17\1	Эмулятор PnP-устройства, INF-файл для установки драйвера	17.1
\GLAVA17\3	Обнаружение изменений в аппаратной конфигурации системы	17.3

Литература и интернет-ресурсы

1. Perrin B. The Art and Science of RS-485. — **http://www.gdscorp.com/pdf/rs485.pdf**.
2. Maxim's Application Note #083, 119, 373, 723, 367. — **http://www.maxim−ic.com**.
3. Microsoft Development Network. — **http://msdn.microsoft.com/**.
4. Бень Е. А. RS-485 для чайников. — **http://kilm.by.ru/im/inter/rs485/chainik.shtml**.
5. Гук М. Аппаратные средства IBM PC. Энциклопедия. — СПб.: Питер, 2002.
6. Гук М. Аппаратные интерфейсы IBM PC. Энциклопедия. — СПб.: Питер, 2002.
7. Конопка Р. Создание оригинальных компонент в среде Delphi. — Киев: DiaSoft Ltd., 1996.
8. Лагутенко О.И. Модемы. Справочник пользователя. — СПб.: Лань, 1997.
9. Локотков А. Интерфейсы последовательной передачи данных. Стандарты RS-422/RS-485. В записную книжку инженера//СТА. — 1997 — № 3. — С. 110—119.
10. Лукач Ю. С. Программно-технические средства персональных ЭВМ семейства IBM PC. — Свердловск: Инженерно-техническое бюро, 1990.
11. Рихтер Д. Windows для профессионалов. — Microsoft Press, 1999.
12. Робертс Д. Прямой INPUT/OUTPUT в среде WINNT. — **http://www.void.ru/content/701/**.

Предметный указатель

A

APC 231
ARQ 41

C

COM-порт 11

D

DCE 12
DTE 12

I

INF-файл:
  ключевые поля 301
  поиск при установке 300
INF-файл:
  структура 84
INT14H 100, 311
INT21H 316
IOPM 165

P

Plug and Play:
  алгоритм для COM-портов 73
  вычисление поля PnP Rev 292
  запуск процедуры 67
  идентификатор 78
  обнаружение изменений 303
  реестр Windows 2000 71
  формирование идентификатора 293
  функции 66
  эмулятор 293

U

UATR 11

А

Активная пауза 45

Б

Базовый адрес:
  значение 104
  определение в DOS 105
  определение в Windows 98 202
  таблица 319
Бит:
  BD 324
  BRCON 322
  CTS 325
  DCD 325
  DCTS 325
  DDCD 325
  DDSR 325
  DLAB 106, 320, 322
  DR 324
  DSR 325
  DTRC 324
  EVENPAR 322
  FE 324
  FIFOE 324
  IE 323
  LME 323
  Low Power Mode 320

OE 324
OUT1C 323
PAREN 323
PE 324
RI 325
RTSC 323
SERIALDB 323
Sleep Mode 320
STIPAR 322
STOPB 323
TEMPT 324
TERI 325
THRE 324
Битстаффингование 44

Д

Дескриптор порта 166
Драйвер GiveIo:
использование 144
недостатки 165
описание 143
расширение возможностей 148

И

Инициализация порта:
INT14H 100, 312
прямое программирование 106

К

Класс:
TCom485 288
TComAscii 254
TComBin 263
TComPort 168
TComPort асинхронный 195
TComPort компонент 241
TComPort с потоками 186
TComPortOnline 207
TComProp 241
TPort 130
TReadThread 192
Константа:
CBR_xxx 168
CLRBREAK 201
CLRDTR 201
FILE_FLAG_OVERLAPPED 194, 231
SERIAL_xxx 402, 403
SETBREAK 201
SETDTR 201
Контроль четности 41
недостатки 184
Контрольная сумма:
CRC16 (алгоритм) 43
CRC16 (вычисление) 275
CRC32 (вычисление) 279
LCR (вычисление) 274
LRC (алгоритм) 42
виды 42
простая (алгоритм) 42
простая (вычисление) 269, 273

М

Модем:
AT-команды 49, 409
Escape-последовательность 50
регистры 50
функции 49
Модуль:
RS232DOS 111
RS232Int 115

Н

Нуль-модемный кабель 12

О

Обратная связь 40

П

Порт:
DOS-имя 225
FCR 403
IER 113, 320
IIR 321
LCR 322, 398
(окончание рубрики см. на с. 474)

Порт *(окончание)*:
LSR 324
MCR 168, 323
MSR 325
RBR 320
TSR 320
завершения аппаратных прерываний 327
обнаружение 222
обращение из Windows 127
получение доступа из Windows 143
разрешение аппаратных прерываний 326
режимов FIFO 321
таблица IRQ 319
Последовательный интерфейс:
принцип работы 19
разъемы 20
события в Windows 197
Потоки:
использование для чтения порта 185
преимущества 185
синхронизация 192
Прерывания:
обработка 114
типы 113
Протокол:
3D Serial Mouse 47
ASCII 33, 254
Bimodem 55
CAN 61
HDLC 44
Kermit 55
Microsoft Mouse 46
Microsoft Plus 47
MODBUS 45, 56
MODBUS-ASCII 57
MODBUS-RTU 58
PC Mouse 48
Profibus 64
RS-485 288
SLIP 45
XModem 51
XModem-1K 52
XModem-CRC 52
YModem 53
YModem-G 54
ZModem 54
бинарный 35, 263
модем ASCII 50
потеря данных 35
составляющие 12, 32
типы 33
Прямое управление драйвером 203

P

Разрыв связи 362

C

Сигнал:
OUT1 323, 402
OUT2 402
готовность DCE (CC/DSR) 25, 325
готовность DTE (CD/DTR) 25, 201, 399, 401, 402
готовность к передаче (CB/CTS) 24, 325
готовность к передаче по дополнительному каналу (SCB/SCTS) 29
готовность к приему (CJ) 26
детектор качества сигнала (CG/SQ) 26
запрос передачи (CA/RTS) 23, 399, 400, 401, 402
запрос передачи по дополнительному каналу (SCA/SRTS) 29
защитная "земля" (AA) 22
индикатор вызова (CE/RI) 25, 325
индикатор тестирования (TM) 27
местный шлейф (LL) 27
обнаружение несущей (CF/DCD) 25, 325
обнаружение несущей дополнительного канала (SCF/SDCD) 29
передаваемые данные (BA/TxD/TD) 23
передаваемые данные дополнительного канала (SBA/STD) 28

переключатель скорости передачи данных от DCE (CI) 26
переключатель скорости передачи данных от DTE (CH) 26
принимаемые данные (BB/RxD/RD) 23
принимаемые данные дополнительного канала (SBB/SRD) 28
сигнальная "земля" (AB/SG) 22
синхронизация передачи от DCE (DB/TC) 28
синхронизация передачи от DTE (DA) 28
синхронизация приема от DCE (DD/RC) 28
удаленный шлейф (RL) 27
Синхронизация:
в DOS 38
в Windows 38, 192
Скорость обмена:
вычисление 108, 325
задание в DOS 106, 320
задание в Windows 342
максимально допустимая 332
Стандарт:
RS-232 14
RS-232 недостатки 282
RS-422A 15
RS-423A 15
RS-449 16
RS-485 15, 282, 288
RS-562 16
V.24 17
V.28 17
V.35 17
X.21bis 18
двойные названия 12
X.21 18
Стартовый байт 44
Структура:
COMMCONFIG 167, 328, 329, 372, 382
COMMPROP 168, 330
COMMTIMEOUTS 167, 336, 361, 377
COMSTAT 339, 364, 382
DCB 167, 201, 340, 359, 361, 376
MODEMDEVCAPS 335
OVERLAPPED 351
TSERIAL_CHARS 401, 405
TSERIAL_COMMPROP 402, 405
TSERIAL_HANDFLOW 402, 405
TSERIAL_LINE_CONTROL 398, 401, 404
TSERIAL_QUEUE_SIZE 404
TSERIAL_STATUS 402, 404
TSERIAL_TIMEOUTS 399, 404
TSERIAL_XOFF_COUNTER 405
TSERIALPERF_STATS 403, 406
TWMDeviceChange 303

Ф

Функция:
BuildCommDCB 167, 359, 361
BuildCommDCBAndTimeouts 167, 361
CancelIo 364
ClearCommBreak 201, 362, 399
ClearCommError 340, 351, 363, 370
CloseHandle 166, 347
CommConfigDialog 167, 241, 329, 374, 381
CreateEvent 367, 369
CreateFile 166, 347
DefineDosDevice 226, 395
DeviceIoControl 148, 203, 388, 397
EnumPorts 222, 390
EscapeCommFunction 201, 365, 399, 400
GetCommConfig 167, 329, 372
GetCommMask 366, 400
GetCommModemStatus 168, 369, 383, 402
GetCommProp 402
GetCommProperties 168, 330, 375
GetCommState 360, 376, 401, 402
GetCommTimeouts 336, 377, 399
GetDefaultCommConfig 167, 374, 381
GetOverlappedResult 194, 351, 369, 387
(окончание рубрики см. на с. 476)

Функция *(окончание)*:
PurgeComm 378, 401
QueryDosDevice 226, 393
ReadFile 166, 194, 350, 370
ReadFileEx 166, 231, 351, 354
SetCommBreak 201, 362, 399
SetCommConfig 167, 329, 372
SetCommMask 198, 366, 368, 401
SetCommState 167, 360, 376, 398, 399, 401, 402
SetCommTimeouts 167, 336, 351, 377, 399
SetDefaultCommConfig 167, 374, 381
SetupComm 168, 380, 398
SleepEx 231
Synchronize 192
TransmitCommChar 382, 399
WaitCommEvent 198, 366, 368, 401
WaitForMultipleObjects 194, 385
WaitForMultipleObjectsEx 231
WaitForSingleObject 194, 198, 369, 384
WaitForSingleObjectEx 231
Windows для работы с портами 166
WriteFile 166, 194, 352
WriteFileEx 166, 231, 353, 357