ПАВЕЛ АГУРОВ

ПРАКТИКА ПРОГРАММИРОВАНИЯ USB

USB 1.1/2.0

КЛАССЫ HID, CDC
И ПРИМЕРЫ ИХ РЕАЛИЗАЦИИ

ФУНКЦИИ РАБОТЫ С USB
ДЛЯ WINDOWS 98/NT/2000/XP

РАЗРАБОТКА USB-ДРАЙВЕРОВ
ДЛЯ WINDOWS 2000/XP

ПРИМЕРЫ ПРОГРАММ
НА ЯЗЫКАХ DELPHI, C, C#

СПЕЦИАЛЬНЫЕ ФУНКЦИИ
RAW INPUT, DIRECT INPUT,
SETUP API

PRO
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

Павел Агуров

ПРАКТИКА ПРОГРАММИРОВАНИЯ USB

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.06
ББК 32.973.26-018.1
А27

Агуров П. В.

А27 Практика программирования USB. — СПб.: БХВ-Петербург, 2006. — 624 с.: ил.

ISBN 978-5-94157-851-1

В книге собрана информация, необходимая для создания USB-устройств и драйверов для операционной системы Microsoft Windows 2000/XP. Рассмотрен процесс создания USB-устройства: от написания программы микроконтроллера (примеры реализованы для микропроцессора AT89C5131) до разработки собственного WDM-драйвера. Содержится описание специальных классов устройств: HID-класс, позволяющий обойтись без разработки драйвера, и класс CDC, позволяющий работать с USB как с обычным COM-портом. Рассмотрено использование функций Raw Input, Direct Input и Setup API, содержится большое количество практических советов и примеров программ на языках Delphi, C и C#. Для удобства читателей все исходные коды приводятся на прилагаемом компакт-диске.

Для программистов и разработчиков аппаратуры

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Константин Костенко*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.03.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 50,31.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.02.953.Д.006421.11.04
от 11.11.2004 г. выдано Федеральной службой по надзору
в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-94157-851-1

© Агуров П. В., 2006
© Оформление, издательство "БХВ-Петербург", 2006

Оглавление

Введение **1**

Для кого эта книга 1
Что вы найдете в книге 2
Программные требования 2
Аппаратные требования 2
О программном коде 3
Краткое описание глав 3
Обозначения 5
Благодарности 6
Обратная связь 6

Часть I. Общие сведения о USB **7**

Глава 1. Спецификация USB **9**

1.1. Что такое USB и зачем это надо 9
1.1.1. Общая архитектура USB 11
1.1.2. Физическая и логическая архитектура USB 11
1.1.3. Составляющие USB 13
1.1.4. Свойства USB-устройств 14
1.1.5. Принципы передачи данных 15
1.1.6. Механизм прерываний 15
1.1.7. Режимы передачи данных 15
1.1.8. Логические уровни обмена данными 16
1.1.8.1. Уровень клиентского ПО 17
1.1.8.2. Уровень системного драйвера USB 17
1.1.8.3. Уровень хост-контроллера интерфейса 19
1.1.8.4. Уровень шины периферийного USB-устройства 19
1.1.8.5. Уровень логического USB-устройства 19
1.1.8.6. Функциональный уровень USB-устройства 19
1.1.9. Передача данных по уровням 20
1.1.10. Типы передач данных 21
1.1.11. Кадры 24
1.1.12. Конечные точки 25
1.1.13. Каналы 26
1.1.14. Пакеты 27
1.1.14.1. Формат маркер-пакетов IN, OUT, SETUP и PING 29

1.1.14.2. Формат пакета SOF 30
1.1.14.3. Формат пакета данных 30
1.1.14.4. Формат пакета подтверждения 31
1.1.14.5. Формат пакета SPLIT 31
1.1.15. Контрольная сумма 32
1.1.15.1. Алгоритм вычисления CRC 32
1.1.15.2. Программное вычисление CRC 34
1.1.16. Транзакции 37
1.1.16.1. Типы транзакций 37
1.1.16.2. Подтверждение транзакций и управление потоком 38
1.1.16.3. Протоколы транзакций 40
1.2. Запросы к USB-устройствам 42
1.2.1. Конфигурационный пакет 42
1.2.2. Стандартные запросы к USB-устройствам 46
1.2.2.1. Получение состояния *GET_STATUS* 46
1.2.2.2. Сброс свойства *CLEAR_FEATURE* 47
1.2.2.3. Разрешение свойства *SET_FEATURE* 47
1.2.2.4. Задание адреса на шине *SET_ADDRESS* 48
1.2.2.5. Получение дескриптора *GET_DESCRIPTOR* 48
1.2.2.6. Передача дескриптора *SET_DESCRIPTOR* 49
1.2.2.7. Получение кода конфигурации *GET_CONFIGURATION* 49
1.2.2.8. Задание кода конфигурации *SET_CONFIGURATION* 50
1.2.2.9. Получение кода настройки интерфейса *GET_INTERFACE* 50
1.2.2.10. Задание кода настройки интерфейса *SET_INTERFACE* 50
1.2.2.11. Задание номера кадра синхронизации *SYNC_FRAME* 51
1.2.2.12. Обработка стандартных запросов 51
1.2.3. Дескриптор устройства 52
1.2.3.1. Дескриптор устройства 52
1.2.3.2. Уточняющий дескриптор устройства 55
1.2.3.3. Дескриптор конфигурации 56
1.2.3.4. Дескриптор интерфейса 59
1.2.3.5. Дескриптор конечной точки 60
1.2.3.6 Дескриптор строки 63
1.2.3.7. Специфические дескрипторы 64
1.2.3.8. Порядок получения дескрипторов 65
1.3. Система Plug and Play (PnP) 69
1.3.1. Конфигурирование USB-устройств 70
1.3.2. Нумерация USB-устройств 70
1.3.3. PnP-идентификаторы USB-устройств 72
1.3.4. Символьные имена устройств 73
1.4. Модель WDM 74

Глава 2. Программирование на языке С для микроконтроллера 78

2.1. Общие сведения о языке С для микроконтроллеров 78
2.2. Использование стандартных библиотек 80

2.3. Программирование для AT89C5131 ... 80
2.3.1. Файл инициализации ... 81
2.3.2. Структуры дескрипторов ... 83
2.3.3. Структура проекта ... 85

Глава 3. Инструменты ... 87

3.1. Программаторы ... 87
3.1.1. Программатор Flip ... 87
3.1.2. Программатор ER-Tronik ... 91
3.2. Инструменты создания драйверов ... 94
3.2.1. NuMega Driver Studio ... 94
3.2.2. Jungo WinDriver ... 94
3.2.3. Jungo KernelDriver ... 94
3.3. Средства Microsoft Visual Studio ... 94
3.3.1. Depends (Dependency Walker) ... 95
3.3.2. Error Lookup ... 95
3.3.3. GuidGen ... 95
3.4. Средства Microsoft DDK ... 96
3.4.1. DeviceTree ... 97
3.4.2. DevCon ... 97
3.4.2.1. Ключ *classes* ... 98
3.4.2.2. Ключ *driverfiles* ... 98
3.4.2.3. Ключ *hwids* ... 100
3.4.2.4. Ключ *rescan* ... 101
3.4.2.5. Ключ *stack* ... 101
3.4.2.6. Ключ *status* ... 102
3.4.3. ChkInf и GenInf ... 103
3.5. Средства CompuWare Corporation ... 103
3.5.1. Monitor ... 103
3.5.2. SymLink ... 104
3.5.3. EzDriverInstaller ... 104
3.5.4. WdmSniff ... 105
3.6. Средства SysInternals ... 106
3.6.1. WinObj ... 106
3.7. Средства USB Forum ... 106
3.7.1. HID Descriptor Tool ... 106
3.8. USB Command Verifier ... 109
3.9. Средства HDD Software ... 111
3.10. Средства Sourceforge ... 111
3.11. Программа мониторинга Bus Hound ... 112

Глава 4. Принципы использования функций Win32 в .NET ... 114

4.1. Общие сведения ... 114
4.2. Импорт функций Win32 ... 115

4.3. Структуры ... 116
4.3.1. Атрибут *StructLayout* ... 117
4.3.2. Атрибут *MarshalAs* ... 118
4.4. Прямой доступ к данным ... 120
4.5. Обработка сообщений Windows ... 121
4.6. Общие сведения о WMI ... 122
4.7. Интернет-ресурсы к этой главе ... 124

ЧАСТЬ II. КЛАССЫ USB ... 125

Глава 5. Класс CDC ... 127

5.1. Методы преобразования интерфейсов USB/RS-232 ... 127
5.2. Общие сведения об интерфейсе RS-232 ... 128
5.2.1. Линии обмена ... 129
5.2.1.1. Передаваемые данные (BA/TxD/TD) ... 130
5.2.1.2. Принимаемые данные (BB/RxD/RD) ... 130
5.2.1.3. Запрос передачи (CA/RTS) ... 130
5.2.1.4. Готовность к передаче (CB/CTS) ... 131
5.2.1.5. Готовность DCE (CC/DSR) ... 132
5.2.1.6. Готовность DTE (CD/DTR) ... 132
5.2.1.7. Индикатор вызова (CE/RI) ... 132
5.2.1.8. Обнаружение несущей (CF/DCD) ... 132
5.2.1.9. Готовность к приему (CJ) ... 133
5.3. Спецификация CDC ... 133
5.3.1. Стандартные дескрипторы ... 133
5.3.2. Функциональные дескрипторы ... 135
5.3.2.1. Заголовочный функциональный дескриптор ... 136
5.3.2.2. Дескриптор режима команд ... 137
5.3.2.3. Дескриптор абстрактного устройства ... 138
5.3.2.4. Дескриптор группирования ... 138
5.3.3. Специальные запросы ... 139
5.3.3.1. Запрос *SET_LINE_CODING* ... 139
5.3.3.2. Запрос *GET_LINE_CODING* ... 140
5.3.3.3. Запрос *SET_CONTROL_LINE_STATE* ... 140
5.3.3.4. Запрос *SEND_BREAK* ... 141
5.3.4. Нотификации ... 141
5.3.4.1. Нотификация *RING_DETECT* ... 141
5.3.4.2. Нотификация *SERIAL_STATE* ... 141
5.4. Поддержка CDC в Windows ... 142
5.4.1. Обзор функций Windows для работы с последовательными портами ... 142
5.4.1.1. Основные операции с портом ... 142
5.4.1.2. Функции настройки порта ... 143
5.4.1.3. Специальная настройка порта ... 143

5.4.1.4. Получение состояния линий модема ... 144
5.4.1.5. Работа с CDC на платформе .NET ... 144
5.4.2. Соответствие функций Windows и USB-запросов ... 144

Глава 6. Класс HID ... 146

6.1. Спецификация HID-устройств ... 146
6.2. Порядок обмена данными с HID-устройством ... 148
6.3. Установка драйвера HID-устройства ... 149
6.4. Идентификация HID-устройства ... 149
6.4.1. Идентификация загрузочных устройств ... 150
6.4.2. Дескриптор конфигурации HID-устройства ... 150
6.4.3. HID-дескриптор ... 151
6.4.4. Дескриптор репорта ... 153
6.5. Структура дескриптора репорта ... 153
6.5.1. Элементы репорта ... 154
6.5.1.1. Элементы короткого типа ... 154
6.5.1.2. Элементы длинного типа ... 154
6.5.2. Типы элементов репорта ... 155
6.5.2.1. Основные элементы ... 155
6.5.2.2. Глобальные элементы ... 158
6.5.2.3. Локальные элементы ... 161
6.5.3. Примеры дескрипторов ... 162
6.6. Запросы к HID-устройству ... 165
6.6.1. Запрос *GET_REPORT* ... 166
6.6.2. Запрос *SET_REPORT* ... 167
6.6.3. Запрос *GET_IDLE* ... 167
6.6.4. Запрос *SET_IDLE* ... 168
6.6.5. Запрос *GET_PROTOCOL* ... 168
6.6.6. Запрос *SET_PROTOCOL* ... 169
6.7. Инструменты ... 169
6.8. Драйверы для HID-устройств в Windows ... 169

Глава 7. Другие классы USB ... 179

ЧАСТЬ III. ПРАКТИКА ПРОГРАММИРОВАНИЯ USB ... 181

Глава 8. Создание USB-устройства на основе AT89C5131 ... 183

8.1. Общая информация об AT89C5131 ... 183
8.2. Структурная схема AT89C5131 ... 185
8.3. USB-регистры AT89C5131 ... 187
8.3.1. Регистр *USBCON* ... 187
8.3.2. Регистр *USBADDR* ... 189
8.3.3. Регистр *USBINT* ... 190
8.3.4. Регистр *USBIEN* ... 191

8.3.5. Регистр *UEPNUM* 192
8.3.6. Регистр *UEPCONX* 193
8.3.7. Регистр *UEPSTAX* 195
8.3.8. Регистр *UEPRST* 197
8.3.9. Регистр *UEPINT* 198
8.3.10. Регистр *UEPIEN* 199
8.3.11. Регистр *UEPDATX* 200
8.3.12. Регистр *UBYCTLX* 201
8.3.13. Регистр *UFNUML* 201
8.3.14. Регистр *UFNUMH* 201
8.4. Схемотехника AT89C5131 202
8.5. Базовый проект для AT89C5131 204
8.5.1. Первая версия программы для AT89C5131 204
8.5.2. Добавляем строковые дескрипторы 226
8.5.3. Добавление конечных точек 231
8.6. Загрузка программы 235

Глава 9. Реализация класса CDC 236

9.1. Реализация CDC 236
9.2. Дескрипторы устройства 237
9.2.1. Инициализация конечных точек 240
9.2.2. Обработка CDC-запросов 241
9.2.3. Конфигурирование RS-порта и CDC-линии 243
9.2.4. Прием и передача данных 244
9.3. Установка драйвера 247
9.4. Программирование обмена данными с CDC-устройством на языке Delphi 249
9.5. Программирование обмена с CDC-устройством на языке C# 279
9.5.1. Использование компонента MSCOMM 279
9.5.2. Использование функций Win32 283
9.6. Проблемы CDC 289

Глава 10. Реализация класса HID 290

10.1. Реализация HID на AT89C5131 290
10.2. Передача нескольких байтов 296
10.3. Feature-репорты 298
10.4. Передача данных от хоста (*SET_REPORT*) 300
10.5. Установка HID-устройства 301
10.6. Обмен данными с HID-устройством 301
10.6.1. Получение имени HID-устройства 302
10.6.2. Получение атрибутов устройства и чтение репортов 305
10.6.3. Передача данных от хоста к HID-устройству 311
10.7. Примеры HID-устройств 312
10.7.1. Реализация устройства "мышь" 312
10.7.2. Реализация устройства "клавиатура" 313

10.8. Использование HID-протокола ... 316
10.8.1. Интерпретация данных ... 318
10.8.2. Коллекции ... 320
10.8.3. Массивы и кнопки ... 326
10.9. HID-устройство с несколькими репортами ... 329

Глава 11. Специальные функции Windows ... 332

11.1. Функции Setup API ... 332
11.1.1. Перечисление USB-устройств ... 332
11.1.2. Получение состояния USB-устройства ... 342
11.2. Перечисление USB-устройств с помощью WMI ... 345
11.3. Специальные функции Windows XP ... 347
11.3.1. *HidD_GetInputReport* — чтение HID-репортов ... 347
11.3.2. Получение данных Raw Input ... 348
11.4. Функции DirectX ... 357
11.5. Диалог добавления нового оборудования ... 363
11.6. Работа с символьными именами устройств ... 364
11.7. Безопасное извлечение флэш-дисков ... 368
11.8. Обнаружение добавления и удаления устройств ... 374
11.9. Интернет-ресурсы ... 378

Глава 12. Разработка драйвера ... 379

12.1. Основные процедуры драйвера WDM ... 379
12.1.1. Процедура *DriverEntry* ... 379
12.1.2. Процедура *AddDevice* ... 382
12.1.3. Процедура *Unload* ... 384
12.1.4. Рабочие процедуры драйвера ... 386
12.1.4.1. Заголовок пакета ... 386
12.1.4.2. Ячейки стека ввода/вывода ... 387
12.4.1.3. Рабочие процедуры драйвера ... 388
12.1.5. Обслуживание запросов IOCTL ... 393
12.2. Загрузка драйвера и обращение к процедурам драйвера ... 399
12.2.1. Процедура работы с драйвером ... 399
12.2.2. Регистрация драйвера ... 401
12.2.2.1. Регистрация с помощью SCM-менеджера ... 401
12.2.2.2. Параметры драйвера в реестре ... 406
12.2.3. Обращение к рабочим процедурам ... 408
12.2.4. Хранение драйвера внутри исполняемого файла ... 409
12.3. Создание драйвера с помощью Driver Studio ... 411
12.3.1. Несколько слов о библиотеке Driver Studio ... 413
12.3.1.1. Класс *KDriver* ... 413
12.3.1.2. Класс *KDevice* ... 413
12.3.1.3. Класс *KIrp* ... 414
12.3.1.4. Класс *KRegistryKey* ... 414
12.3.1.5. Класс *KLowerDevice* ... 414
12.3.1.6. Классы USB ... 416

12.3.2. Другие классы Driver Studio 417
12.3.3. Создание шаблона драйвера с помощью Driver Studio 417
12.3.3.1. Шаг 1. Задание имени и пути проекта 418
12.3.3.2. Шаг 2. Выбор архитектуры драйвера 418
12.3.3.3. Шаг 3. Выбор шины 418
12.3.3.4. Шаг 4. Задание набора конечных точек 420
12.3.3.5. Шаг 5. Задание имени класса и файла 421
12.3.3.6. Шаг 6. Выбор функций драйвера 421
12.3.3.7. Шаг 7. Выбор способа обработки запросов 423
12.3.3.8. Шаг 8. Создание сохраняемых параметров драйвера 424
12.3.3.9. Шаг 9. Свойства драйвера 425
12.3.3.10. Шаг 10. Задание кодов IOCTL 427
12.3.3.11. Шаг 11. Дополнительные настройки 427
12.3.4. Доработка шаблона драйвера 429
12.3.5. Базовые методы класса устройства 429
12.3.6. Реализация чтения данных 433
12.3.7. Установка драйвера 434
12.3.8. Программа чтения данных 435
12.3.9. Чтение данных с конечных точек других типов 445
12.3.10. "Чистый" драйвер USB-устройства 446

ЧАСТЬ IV. СПРАВОЧНИК 465

Глава 13. Формат INF-файла 467

13.1. Структура INF-файла 467
13.1.1. Секция *Version* 468
13.1.2. Секция *Manufacturer* 470
13.1.3. Секция *DestinationDirs* 472
13.1.3.1. Ключ *DefaultDescDir* 472
13.1.3.2. Ключи *file-list-section* 472
13.1.3.3. Ключ *dirid* 473
13.1.3.4. Ключ *subdir* 474
13.1.4. Секция описания модели 474
13.1.5. Секция *xxx.AddReg* и *xxx.DelReg* 475
13.1.6. Секция *xxx.LogConfig* 477
13.1.7. Секция *xxx.CopyFiles* 477
13.1.8. Секция *Strings* 479
13.1.9. Связи секций 479
13.2. Создание и тестирование INF-файлов 480
13.3. Установка устройств с помощью INF-файла 482
13.4. Ветки реестра для USB 482

Глава 14. Базовые функции Windows 484

14.1. Функции *CreateFile* и *CloseHandle*: открытие и закрытие объекта 484
14.1.1. Дополнительные сведения 485

14.1.2. Возвращаемое значение ... 486
14.1.3. Пример вызова ... 486
14.2. Функция *ReadFile*: чтение данных ... 488
14.2.1. Дополнительные сведения ... 489
14.2.2. Возвращаемое значение ... 490
14.2.3. Пример вызова ... 490
14.3. Функция *WriteFile*: передача данных ... 491
14.3.1. Дополнительные сведения ... 492
14.3.2. Возвращаемое значение ... 492
14.3.3. Пример вызова ... 493
14.4. Функция *ReadFileEx*: APC-чтение данных ... 494
14.4.1. Возвращаемое значение ... 495
14.4.2. Дополнительные сведения ... 495
14.4.3. Пример вызова ... 496
14.5. Функция *WriteFileEx*: APC-передача данных ... 496
14.5.1. Возвращаемое значение ... 497
14.5.2. Пример вызова ... 497
14.6. Функция *WaitForSingleObject*: ожидание сигнального состояния объекта ... 498
14.6.1. Возвращаемое значение ... 499
14.7. Функция *WaitForMultipleObjects*: ожидание сигнального состояния объектов ... 499
14.7.1. Возвращаемое значение ... 500
14.8. Функция *GetOverlappedResult*: результат асинхронной операции ... 501
14.8.1. Возвращаемое значение ... 502
14.9. Функция *DeviceIoControl*: прямое управление драйвером ... 502
14.9.1. Возвращаемое значение ... 504
14.10. Функция *CancelIo*: прерывание операции ... 504
14.10.1. Возвращаемое значение ... 504
14.11. Функция *QueryDosDevice*: получение имени устройства по его DOS-имени ... 505
14.11.1. Возвращаемое значение ... 505
14.11.2. Пример вызова ... 506
14.12. Функция *DefineDosDevice*: операции с DOS-именем устройства ... 507
14.12.1. Возвращаемое значение ... 507
14.12.2. Пример вызова ... 507

Глава 15. Структуры и функции Windows для последовательных портов ... 509

15.1. Структура настроек порта *COMMCONFIG* ... 509
15.2. Структура свойств порта *COMMPROP* ... 511
15.3. Структура тайм-аутов *COMMTIMEOUTS* ... 518
15.4. Структура статуса порта *COMSTAT* ... 520
15.5. Структура *DCB* ... 522
15.6. Функция *BuildCommDCB*: создание структуры *DCB* из строки ... 528
15.6.1. Дополнительные сведения ... 530

15.6.2. Возвращаемое значение 530
15.6.3. Пример вызова 530
15.7. Функция *BuildCommDCBAndTimeouts*: создание структуры *DCB* и тайм-аутов из строки 531
15.8. Функции *SetCommBreak* и *ClearCommBreak*: управление выводом данных 531
15.8.1. Возвращаемое значение 532
15.9. Функция *ClearCommError*: получение и сброс ошибок порта 532
15.9.1. Возвращаемое значение 533
15.10. Функция *EscapeCommFunction*: управление портом 534
15.10.1. Возвращаемое значение 534
15.11. Функции *GetCommMask* и *SetCommMask*: маска вызова событий 535
15.11.1. Возвращаемое значение 535
15.12. Функция *WaitCommEvent*: ожидание события COM-порта 536
15.12.1. Возвращаемое значение 537
15.12.2. Дополнительные сведения 537
15.12.3. Пример вызова 537
15.13. Функции *GetCommConfig* и *SetCommConfig*: конфигурирование параметров порта 540
15.13.1. Возвращаемое значение 541
15.13.2. Пример вызова 541
15.14. Функция *CommConfigDialog*: диалог конфигурирования порта 542
15.14.1. Возвращаемое значение 543
15.14.2. Дополнительные сведения 543
15.14.3. Пример вызова 543
15.15. Функция *GetCommProperties*: прочитать свойства порта 544
15.15.1. Возвращаемое значение 544
15.15.2. Пример вызова 544
15.16. Функции *GetCommState* и *SetCommState*: состояние порта 544
15.16.1. Возвращаемое значение 545
15.16.2. Пример вызова 545
15.17. Функции *GetCommTimeouts* и *SetCommTimeouts*: тайм-ауты порта 546
15.17.1. Возвращаемое значение 547
15.17.2. Пример вызова 547
15.18. Функция *PurgeComm*: сброс буферов порта 547
15.18.1. Возвращаемое значение 548
15.18.2. Пример вызова 548
15.19. Функция *SetupComm*: конфигурирование размеров буферов 548
15.19.1. Возвращаемое значение 550
15.20. Функции *GetDefaultCommConfig* и *SetDefaultCommConfig*: настройки порта по умолчанию 550
15.20.1. Возвращаемое значение 551
15.21. Функция *TransmitCommChar*: передача специальных символов 551
15.21.1. Возвращаемое значение 551

15.22. Функция *GetCommModemStatus*: статус модема 552
15.22.1. Возвращаемое значение 552
15.22.2. Пример вызова 553
15.23. Функция *EnumPorts*: перечисление портов 553
15.23.1. Дополнительные сведения 555
15.23.2. Возвращаемое значение 555
15.23.3. Пример вызова 555

Глава 16. Структуры и функции Windows Setup API 557

16.1. Функция *SetupDiGetClassDevs*: перечисление устройств 557
16.1.1. Возвращаемое значение 558
16.2. Функция *SetupDiDestroyDeviceInfoList*: освобождение блока описания устройства 558
16.2.1. Возвращаемое значение 559
16.3. Функция *SetupDiEnumDeviceInterfaces*: информация об устройстве 559
16.3.1. Возвращаемое значение 560
16.4. Функция *SetupDiGetDeviceInterfaceDetail*: детальная информация об устройстве 561
16.5. Функция *SetupDiEnumDeviceInfo*: информация об устройстве 562
16.6. Функция *SetupDiGetDeviceRegistryProperty*: получение Plug and Play свойств устройства 564
16.7. Функция *CM_Get_DevNode_Status*: статус устройства 565
16.8. Функция *CM_Request_Device_Eject*: безопасное извлечение устройства 566

Глава 17. Структуры и функции Windows HID API 568

17.1. Функция *HidD_Hello*: проверка библиотеки 568
17.2. Функция *HidD_GetHidGuid*: получение GUID 569
17.3. Функция *HidD_GetPreparsedData*: создание описателя устройства 570
17.4. Функция *HidD_FreePreparsedData*: освобождение описателя устройства 571
17.5. Функция *HidD_GetFeature*: получение Feature-репорта 571
17.6. Функция *HidD_SetFeature*: передача Feature-репорта 573
17.7. Функция *HidD_GetNumInputBuffers*: получение числа буферов 573
17.8. Функция *HidD_SetNumInputBuffers*: установка числа буферов 574
17.9. Функция *HidD_GetAttributes*: получение атрибутов устройства 575
17.10. Функция *HidD_GetManufacturerString*: получение строки производителя 576
17.11. Функция *HidD_GetProductString*: получение строки продукта 578
17.12. Функция *HidD_GetSerialNumberString*: получение строки серийного номера 578
17.13. Функция *HidD_GetIndexedString*: получение строки по индексу 579
17.14. Функция *HidD_GetInputReport*: получение Input-репорта 580
17.15. Функция *HidD_SetOutputReport*: передача Output-репорта 581

17.16. Функция *HidP_GetCaps*: получение свойств устройства 581
17.17. Функция *HidP_MaxDataListLength*: получение размеров репортов 584
17.18. Функция *HidD_FlushQueue*: сброс буферов 585
17.19. Функция *HidP_GetLinkCollectionNodes*: дерево коллекций 585
17.20. Функции *HidP_GetScaledUsageValue и HidP_SetScaledUsageValue*: получение и задание преобразованных значений 586
17.21. Функция *HidP_MaxUsageListLength: размер буфера для кодов клавиш* 587
17.22. Функция *HidP_UsageListDifference*: различие между массивами 587

ПРИЛОЖЕНИЯ 589

Приложение 1. Дополнительные функции 591

Приложение 2. Компиляция примеров в других версиях Delphi 592

Приложение 3. Таблица идентификаторов языков (LangID) 593

Приложение 4. Таблица кодов производителей (Vendor ID, Device ID) 596

Приложение 5. Как создать ярлык Device Manager 598

Приложение 6. Часто задаваемые вопросы 599

Приложение 7. Описание компакт-диска 600

Литература 602

Предметный указатель 603

Введение

Со дня выпуска книги "Интерфейс USB. Практика использования и программирования" прошел почти год. Судя по количеству писем, она оказалась востребована, и появилась идея второй книги, основанной на вопросах читателей и являющейся логическим продолжением первой.

В процессе дискуссий с читателями выяснились несколько основных направлений, вызывающих наибольший интерес: USB-устройства, не требующие написания драйверов (HID, CDC), подключение USB-устройств, как виртуальных последовательных портов, а также реализация всех алгоритмов на современном языке C#. В этом издании мы постараемся дать информацию об этих и многих других вопросах.

Кроме того, мы рассмотрим некоторые функции, добавленные в операционную систему Windows XP, значительно облегчающие работу с USB-интерфейсом. Так, например, функции Raw Input позволяют получать данные HID-устройств с помощью одной-двух строк кода (без использования сложных функций поиска и открытия устройства). Мы будем также обсуждать функции DirectX для чтения данных HID-устройства — класс функций `DirectInput`.

Для кого эта книга

Данная книга предназначена для программистов-практиков. Она не содержит сведений об аппаратной части USB, а полностью посвящена вопросам программирования.

Эта книга для вас, если:

- вы хотите научиться использовать современный USB-интерфейс;
- вы хотите разработать USB-устройство, работающее без установки драйверов;
- вы хотите работать с USB-устройством как с обычным последовательным портом;
- вы хотите разработать свой драйвер для USB-устройства.

Что вы найдете в книге

Эта книга посвящена USB-интерфейсу, но вы не найдете в ней сведений об электрических характеристиках и аппаратной части, зато найдете множество исходных кодов, примеров программ и просто полезных советов. Книга содержит сведения, необходимые для работы с USB-интерфейсом и написания драйверов. На прилагаемом компакт-диске содержатся полные исходные тексты и скомпилированные модули программ.

Программные требования

На компьютере должна быть установлена либо Windows 2000, либо Windows XP. Желательно при этом наличие всех доступных пакетов обновлений (Service Pack). Возможно использование Windows 98, но многие примеры, связанные с написанием драйверов, в ней работать не будут.

Для написания программ мы будем использовать пакеты Borland Delphi 6, Visual Studio 6 и Microsoft C# (.NET 1.1). Так как мы не будем использовать никаких специфических функций, присущих именно этим средствам разработки, то все примеры могут быть скомпилированы в других версиях этих пакетов, практически без модификации (*см. приложение 2*). Для компиляции драйверов, приведенных в книге, потребуется Windows 2000 DDK или Windows XP DDK, в соответствии с вашей версией Windows.

Как дополнительный источник информации мы рекомендуем установить MSDN[1].

Аппаратные требования

Достаточно обычного домашнего компьютера, на котором компиляция программы в Borland Delphi или Visual Studio занимает приемлемое для вас время. Установка Borland Delphi 6 потребует примерно 300 Мбайт на жестком диске, установка Visual Studio — 240 Мбайт, MSDN — 1,5 Гбайт, Windows DDK — 700 Мбайт, C# — 300 Мбайт, .NET 1.1 — 70 Мбайт. Для тестирования программ необходимо иметь одно или несколько USB-устройств. Для создания своих устройств будет необходим соответствующий инструментарий.

[1] MSDN (Microsoft Developer Network) — собрание документов компании Microsoft, содержащее сведения обо всех ее разработках.

О программном коде

Вопрос о языках программирования, используемых в книге, сложен. Мы признаемся, что так и не смогли выбрать между Delphi, C и C#. Книги [3] и [4] были написаны для языка Delphi, однако, количество писем читателей, желающих получить код для языка C, примерно одинаково с количеством писем читателей, использующих Delphi. А актуальность и нарастающая популярность C# не дают возможности не приводить примеры для платформы .NET. Идя на компромисс, мы приведем многие примеры на Delphi, примеры, специфические для .NET, будут использовать язык C#. Для языка C мы приведем заголовки основных структур и функций.

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

В программах на Delphi не приводится код самого проекта (DPR-файл) и код формы (DFM-файлы) — их можно найти на компакт-диске. Следует отдельно заметить (этот вопрос очень часто возникал у читателей первой книги), что для компиляции Delphi-программ, требуется либо вручную подключать необходимые функции (см. листинг 7.6), либо использовать модули из библиотеки JEDI (**http://delphi-jedi.org**). Мы специально не включали библиотеку на компакт-диск, т. к. читатели всегда могут получить последнюю версию с официального сайта. *Подробную информацию о компиляции примеров в различных версиях Delphi см. в приложении 2.*

Краткое описание глав

Главы 1—4 составляют первую часть книги. В ней приводятся общие сведения и спецификации USB, объясняются общие принципы программирования на языке C для микроконтроллера и принципы программирования аппаратуры на платформе .NET. Кроме того, первая часть содержит главу об инструментах, делающих работу с USB более простой и комфортной.

- *Глава 1* ("Спецификация USB") содержит описание USB-интерфейса: архитектуру, механизмы передачи данных, конечные точки, форматы пакетов, запросы. Кроме того, в этой главе содержатся общие сведения о системе Plug and Play и драйверной модели WDM (Windows Driver Model).
- *Глава 2* ("Программирование на языке C для микроконтроллера") содержит информацию об общих принципах программирования на языке C для микроконтроллера на примере AT89C5131. Здесь мы приводим базовый проект, которым будем пользоваться при построении приме-

ров. Программисты, знакомые с языком C, могут смело пропустить эту главу.

- *Глава 3* ("Инструменты") содержит описание программ, которые могут оказаться полезными при работе с USB-интерфейсом.
- *Глава 4* ("Принципы использования функций Win32 в .NET") содержит информацию, необходимую для программирования аппаратуры на платформе .NET.

Главы 5—7 составляют вторую часть книги, посвященную классам USB-устройств, использование которых избавляет от создания собственных драйверов.

- *Глава 5* ("Класс CDC") содержит описание класса коммуникационных устройств. Устройства, которые мы рассмотрим, "видны" операционной системе как обычный последовательный порт. В этой же главе приведено описание функций Windows для работы с такими устройствами (как для Win32, так и для .NET).
- *Глава 6* ("Класс HID") содержит описание класса устройств связи с пользователем, который позволяет создавать широкий спектр USB-устройств, не требующих написания дополнительных драйверов.
- *Глава 7* ("Другие USB-классы") содержит информацию о некоторых других USB-классах.

Главы 8—12 составляют практическую часть книги. В этой части мы создадим USB-устройства классов CDC и HID и научимся работать с ними с помощью функций Windows. Кроме того, эта часть содержит всю информацию, необходимую для создания своего драйвера.

- *Глава 8* ("Создание USB-устройства на основе AT89C5131") содержит описание процесса разработки USB-устройства. Этот проект будет базой для создания других USB-устройств.
- *Глава 9* ("Реализация класса CDC") содержит описание процесса разработки CDC-устройства и программы для обмена данными с ним.
- *Глава 10* ("Реализация класса HID") содержит описание процесса разработки HID-устройства и программы для обмена данными с ним.
- *Глава 11* ("Специальные функции Windows") содержит примеры программ, использующих функции Setup API, WMI, DirectX и множество других полезных модулей. Кроме того, в этой главе приводятся листинги программ получения Raw Input-данных.
- *Глава 12* ("Разработка USB-драйвера") описывает процесс разработки собственного USB-драйвера.

Главы 13—17 содержат справочные материалы.

Обозначения

При описании некоторых данных мы будем пользоваться битовым представлением, заключая число разрядов каждого поля в квадратные скобки. Например:

- [5] поле `A`;
- [2] поле `Б`.

Такое описание означает, что поле `A` содержит 5 битов, а поле `Б` — 2 бита. Еще одно представление битовых полей — указание конкретных диапазонов битов с помощью знака ":", например:

- [16:5] зарезервированы;
- [4:0] индекс.

Такое описание означает, что биты с 16 по 5 включительно зарезервированы, а биты с 4 по 0 включительно представляют собой индекс. Отличить первое описание от второго обычно легко по контексту изложения.

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 3FH;
- битовые последовательности заключены в угловые скобки, например, <0010>, либо, при написании двоичного числа, обозначены символом "b" в конце, например, 10101111b.

Для описания версий протоколов, структур и т. д. будет использоваться специальный тип чисел BCD (Binary Coded Decimal, двоично-десятичное число). Такие числа записываются в шестнадцатеричном виде 0xJJMN для обозначения версии JJ.M.N, т. е. JJ обозначает старший номер версии, M — младший номер версии и N — номер подверсии. Например, версия 2.1.3 будет представлена числом 0x213, а версия 2.0 будет записана числом 0x0200 (*см. приложение 1*).

При описании регистров мы будем пользоваться следующими обозначениями режимов доступа:

- RO — (read only) регистр только для чтения, запись в него не возможна;
- WO — (write only) регистр только для записи, чтение значения не возможно;
- R/W — (read/write) возможно и чтение и запись значения;
- R/W2 — (read/write word) возможно чтение и запись слова;

- R/WC — (read/write clear) разрешены как чтение, так и запись значения, однако при записи 1 в некоторый разряд регистра приводит к его сбросу в ноль.

При необходимости указания версии Windows мы будем использовать следующие сокращения:

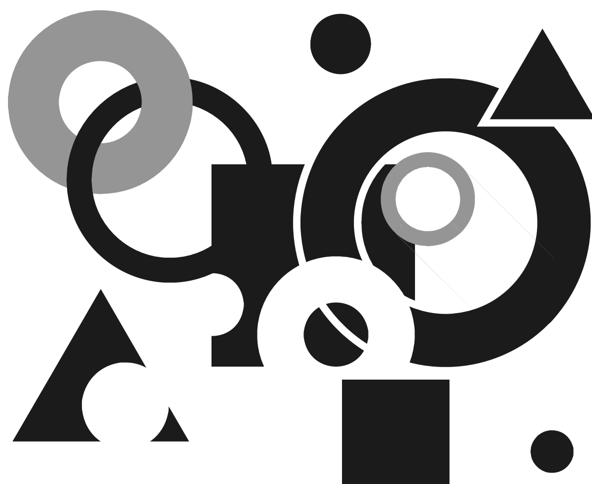
- Windows 9*x* будет означать семейство Windows 95/98/ME;
- Windows NT в общем случае будет обозначать семейство Windows NT/2000/XP;
- При необходимости указания конкретной версии мы будем писать номер этой версии без сокращений, например, Windows NT4 или Windows 98.

Благодарности

Вторая книга, посвященная программированию USB, не увидела бы свет, если бы не труд многих людей, прямо или косвенно принимавших участие в ее разработке. Автор выражает благодарность Сергею Малову, как обычно, помогавшему в разработке аппаратной части. Отдельное спасибо заместителю главного редактора издательства "БХВ-Петербург" Евгению Рыбакову, без участия и советов которого книга не была бы такой интересной. Спасибо всем читателям, присылавшим свои отзывы, критику и пожелания, а также коллективу издательства.

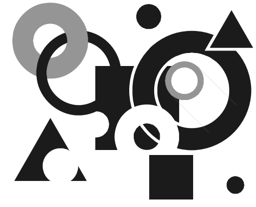
Обратная связь

Обо всех ошибках или пожеланиях сообщайте по адресу **books@pvasoft.com**. На сайте **http://book.pvasoft.com** можно найти текущий список опечаток и дополнений.



ЧАСТЬ I

Общие сведения о USB



Глава 1

Спецификация USB

Подробное описание USB [4] мы рассматривать не будем, а приведем лишь минимум сведений, необходимых для практической работы.

1.1. Что такое USB и зачем это надо

Шина USB (Universal Serial Bus, универсальная последовательная шина) появилась в начале 1996 года как попытка решения проблемы множественности интерфейсов. К тому времени персональные компьютеры (ПК) были оснащены большим количеством разнообразных внешних интерфейсов, полезных и необходимых, но обладающих одним недостатком: все они требовали своего специального разъема и, чаще всего, выделенного аппаратного прерывания (IRQ, Interrupt ReQuest).

Первая спецификация (версия 1.0) USB была опубликована в начале 1996 года, а осенью 1998 года появилась спецификация 1.1, исправляющая проблемы, обнаруженные в первой редакции. Весной 2000 года была опубликована версия 2.0, в которой предусматривалось 40-кратное повышение пропускной способности шины. Так, спецификация 1.0 и 1.1 обеспечивает работу на скоростях 12 Мбит/с и 1,5 Мбит/с, а спецификация 2.0 — на скорости 480 Мбит/с. При этом предусматривается обратная совместимость USB 2.0 с USB 1.x, т. е. "старые" USB 1.x устройства будут работать с USB 2.0 контроллерами, правда, на скорости 12 Мбит/с.

Разработчики шины ориентировались на создание интерфейса, обладающего следующими свойствами:

- легкореализуемое расширение периферии ПК;
- дешевое решение, позволяющее передавать данные со скоростью до 12 Мбит/с (480 Мбит/с для USB 2.0);
- полная поддержка в реальном времени голосовых, аудио- и видеопотоков;

- гибкость протокола смешанной передачи изохронных данных и асинхронных сообщений;
- интеграция с выпускаемыми устройствами;
- охват всевозможных конфигураций и конструкций ПК;
- обеспечение стандартного интерфейса, способного быстро завоевать рынок;
- создание новых классов устройств, расширяющих ПК.

Спецификация USB определяет следующие функциональные возможности интерфейса:

- простота использования для конечного пользователя:
  - простота кабельной системы и подключений;
  - скрытие подробностей электрического подключения от конечного пользователя;
  - самоидентифицирующиеся устройства с автоматическим конфигурированием;
  - динамическое подключение и переконфигурирование периферийных устройств;
- широкие возможности работы:
  - пропускная способность от нескольких Кбит/с до нескольких Мбит/с;
  - поддержка одновременно как изохронной, так и асинхронной передачи данных;
  - поддержка одновременных операций со многими устройствами (multiple connections);
  - поддержка до 127 устройств на шине;
  - передача разнообразных потоков данных и сообщений;
  - поддержка составных устройств (периферийных устройств, выполняющих несколько функций);
  - низкие накладные расходы передачи данных;
- равномерная пропускная способность:
  - гарантированная пропускная способность и низкие задержки голосовых и аудиоданных;
  - возможность использования всей полосы пропускания;
- гибкость:
  - поддержка разных размеров пакетов, которые позволяют настраивать функции буферизации устройств;
  - настраиваемое соотношение размера пакета и задержки данных;
  - управление потоком (flow control) данных на уровне протокола;

- надежность:
  - контроль ошибок и восстановление на уровне протокола;
  - динамическое добавление и удаление устройств прозрачно для конечного пользователя;
  - поддержка идентификации неисправных устройств;
  - исключение неправильного соединения устройств;
- выгода для разработчиков:
  - простота реализации и внедрения;
  - объединение с архитектурой Plug and Play;
- дешевая реализация:
  - дешевые каналы со скоростью работы до 1,5 Мбит/с;
  - оптимизация для интеграции с периферией;
  - применимость для реализации дешевой периферии;
  - дешевые кабели и разъемы;
  - использование выгодных товарных технологий;
- возможность простого обновления.

Практически все поставленные задачи были решены, и весной 1997 года стали появляться компьютеры, оборудованные разъемами для подключения USB-устройств.

1.1.1. Общая архитектура USB

Обычная архитектура USB подразумевает подключение одного или нескольких *USB-устройств* к компьютеру (рис. 1.1), который в такой конфигурации является главным управляющим устройством и называется *хостом*. Подключение USB-устройств к хосту производится с помощью *кабелей*. Для соединения компьютера и USB-устройства используется *хаб*. Компьютер имеет встроенный хаб, называемый *корневым хабом*.

1.1.2. Физическая и логическая архитектура USB

Физическая архитектура USB определяется следующими правилами (рис. 1.2):

- устройства подключаются к хосту;
- физическое соединение устройств между собой осуществляется по топологии многоярусной звезды, вершиной которой является корневой хаб;
- центром каждой звезды является хаб;

- каждый кабельный сегмент соединяет между собой две точки: хост с хабом или функцией (*см. далее*), хаб с функцией или другим хабом;
- к каждому порту хаба может подключаться периферийное USB-устройство или другой хаб, при этом допускается до 5 уровней каскадирования хабов, не считая корневого.

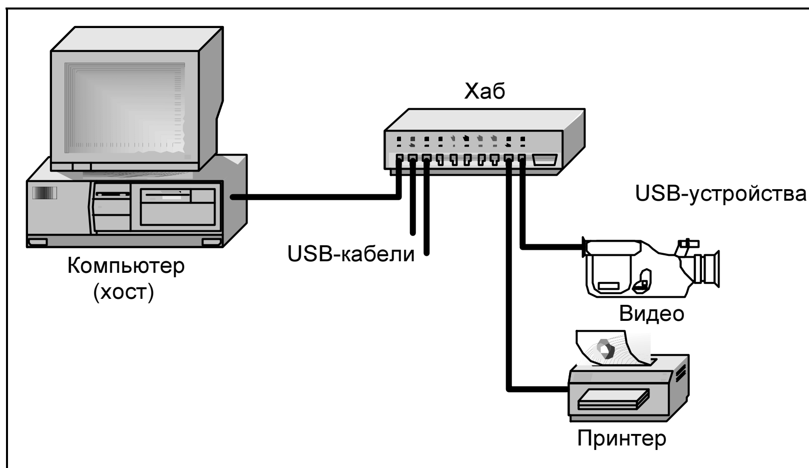


Рис. 1.1. Обычная архитектура USB

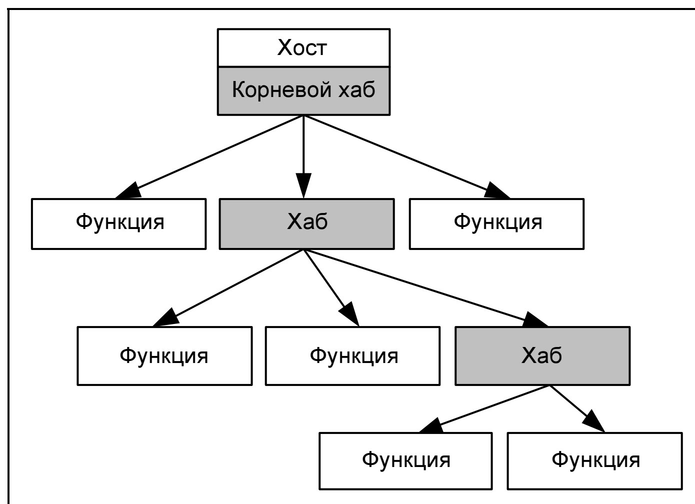


Рис. 1.2. Физическая архитектура USB

Детали физической архитектуры скрыты от прикладных программ в системном программном обеспечении (ПО), поэтому *логическая архитектура* выглядит как обычная звезда, центром которой является прикладное ПО, а вершинами — набор *конечных точек* (рис. 1.3).

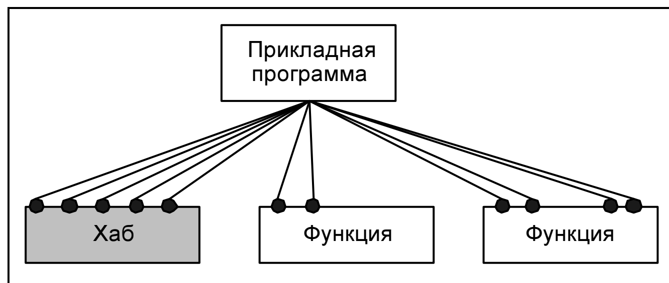


Рис. 1.3. Логическая архитектура USB

Прикладная программа ведет обмен информацией с каждой конечной точкой.

1.1.3. Составляющие USB

Шина USB состоит из следующих элементов:

- *хост-контроллер* (host controller) — это главный контроллер, который входит в состав системного блока компьютера и управляет работой всех устройств на шине USB. Для краткости мы будем писать просто хост. На шине USB допускается наличие только одного хоста. Системный блок персонального компьютера содержит один или несколько хостов, каждый из которых управляет отдельной шиной USB;
- *устройство* (device) может представлять собой хаб, функцию или их комбинацию (compound device);
- *порт* (port) — точка подключения;
- *хаб* (hub, другое название — *концентратор*) — устройство, которое обеспечивает дополнительные порты на шине USB. Другими словами, хаб преобразует один порт (*восходящий порт*, upstream port) во множество портов (*нисходящие порты*, downstream ports). Архитектура допускает соединение нескольких хабов (не более 5). Хаб распознает подключение и отключение устройств к портам и может управлять подачей питания на порты. Каждый из портов может быть разрешен или запрещен и сконфигурирован на полную или ограниченную скорость обмена. Хаб обеспечивает изоляцию сегментов с низкой скоростью от высокоскоростных. Хаб может ограничивать ток, потребляемый каждым портом;

- *корневой хаб* (root hub) — это хаб, входящий в состав хоста;
- *функция* (function) — это периферийное USB-устройство или его отдельный блок, способный передавать и принимать информацию по шине USB. Каждая функция предоставляет конфигурационную информацию, описывающую возможности периферийного USB-устройства и требования к ресурсам. Перед использованием функция должна быть сконфигурирована хостом — ей должна быть выделена полоса в канале и выбраны опции конфигурации;
- *логическое USB-устройство* (logical device) представляет собой набор конечных точек.

1.1.4. Свойства USB-устройств

Спецификация USB достаточно жестко определяет набор свойств, которые должно поддерживать любое USB-устройство:

- адресация — устройство должно отзываться на назначенный ему уникальный адрес и только на него;
- конфигурирование — после включения или сброса устройство должно предоставлять нулевой адрес для возможности конфигурирования его портов;
- передача данных — устройство имеет набор конечных точек для обмена данными с хостом. Для конечных точек, допускающих разные типы передач, после конфигурирования доступен только один из них;
- управление энергопотреблением — любое устройство при подключении не должно потреблять от шины ток, превышающий 100 мА. При конфигурировании устройство заявляет свои потребности тока, но не более 500 мА. Если хаб не может обеспечить устройству заявленный ток, устройство не будет использоваться;
- приостановка — USB-устройство должно поддерживать приостановку (suspended mode), при которой его потребляемый ток не превышает 500 мкА. USB-устройство должно автоматически приостанавливаться при прекращении активности шины;
- удаленное пробуждение — возможность удаленного пробуждения (remote wakeup) позволяет приостановленному USB-устройству подать сигнал хосту, который тоже может находиться в приостановленном состоянии. Возможность удаленного пробуждения описывается в конфигурации USB-устройства. При конфигурировании эта функция может быть запрещена.

1.1.5. Принципы передачи данных

Механизм передачи данных является асинхронным и блочным. Блок передаваемых данных называется *USB-фреймом* или *USB-кадром* (*см. разд. 1.1.11*) и передается за фиксированный временной интервал. Оперирование командами и блоками данных реализуется при помощи логической абстракции, называемой *каналом* (*см. разд. 1.1.13*). Внешнее устройство также делится на логические абстракции, называемые *конечными точками* (*см. разд. 1.1.12*). Таким образом, канал является логической связкой между хостом и конечной точкой внешнего устройства. Канал можно сравнить с открытым файлом.

Для передачи команд (и данных, входящих в состав команд) используется канал по умолчанию, а для передачи данных открываются либо *потоковые каналы*, либо *каналы сообщений* (*см. разд. 1.1.13*).

Все операции по передаче данных по шине USB инициируются хостом. Периферийные USB-устройства сами начать обмен данными не могут. Они могут только реагировать на команды хоста.

1.1.6. Механизм прерываний

Для шины USB настоящего механизма прерываний (как, например, для последовательного порта) не существует. Вместо этого хост опрашивает подключенные устройства на предмет наличия данных о прерывании. Опрос происходит в фиксированные интервалы времени, обычно каждые 1—32 мс. Устройству разрешается посылать до 64 байт данных.

С точки зрения драйвера, возможности работы с прерываниями фактически определяются хостом, который и обеспечивает поддержку физической реализации USB-интерфейса.

1.1.7. Режимы передачи данных

Пропускная способность шины USB, соответствующей спецификации 1.1, составляет 12 Мбит/с (т. е. 1,5 Мбайт/с). Спецификация 2.0 определяет шину с пропускной способностью 400 Мбайт/с. Полоса пропускания делится между всеми устройствами, подключенными к шине.

Шина USB имеет три режима передачи данных:

- низкоскоростной (LS, Low-speed);
- полноскоростной (FS, Full-speed);
- высокоскоростной (HS, High-speed, только для USB 2.0).

1.1.8. Логические уровни обмена данными

Спецификация USB определяет три *логических уровня* с определенными правилами взаимодействия. USB-устройство содержит интерфейсную, логическую и функциональную части. Хост тоже делится на три части — интерфейсную, системную и ПО. Каждая часть отвечает только за определенный круг задач. Логическое и реальное взаимодействие между ними показано на рис. 1.4.

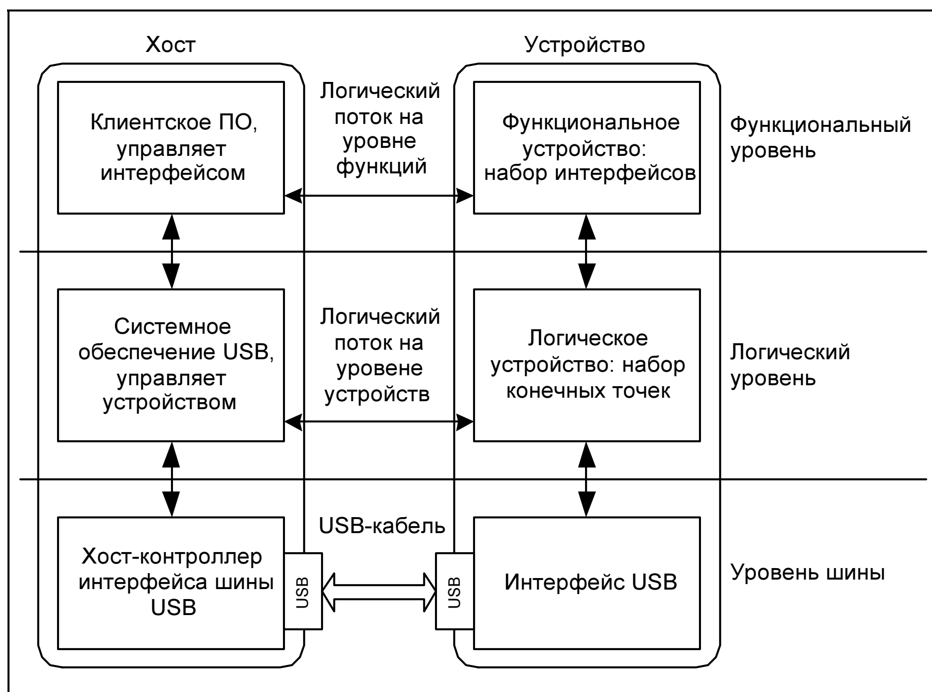


Рис. 1.4. Взаимодействие компонентов USB

Таким образом, операция обмена данными между прикладной программой и шиной USB выполняется путем передачи буферов памяти через следующие уровни:

- уровень клиентского ПО в хосте:
  - обычно представляется драйвером USB-устройства;
  - обеспечивает взаимодействие пользователя с операционной системой с одной стороны и системным драйвером с другой;

- уровень системного драйвера USB в хосте (USBD, Universal Serial Bus Driver):
  - управляет нумерацией устройств на шине;
  - управляет распределением пропускной способности шины и мощности питания;
  - обрабатывает запросы пользовательских драйверов;
- уровень хост-контроллера интерфейса шины USB (HCD, Host Controller Driver):
  - преобразует запросы ввода/вывода в структуры данных, по которым выполняются физические транзакции;
  - работает с регистрами хоста.

Рассмотрим каждый уровень более подробно.

1.1.8.1. Уровень клиентского ПО

На уровне клиентского программного обеспечения определяется тип передачи данных, необходимый для выполнения затребованной прикладной программой операции. После определения типа передачи данных на системный уровень передается следующее:

- буфер памяти, называемый клиентским буфером;
- *пакет запроса на ввод/вывод* (IRP, Input/output Request Packet), указывающий тип необходимой операции.

IRP содержит только сведения о запросе (адрес и длина буфера в оперативной памяти). Непосредственно обработкой запроса занимается системный драйвер USB. *Подробную информацию о запросах клиентского ПО см. в разд. 1.2, о формировании пакетов IRP — в гл. 12.*

1.1.8.2. Уровень системного драйвера USB

Уровень системного драйвера USB необходим для управления ресурсами USB. Он отвечает за выполнение следующих действий:

- распределение полосы пропускания шины USB;
- назначение логических адресов каждому физическому USB-устройству;
- планирование транзакций.

Распределение полосы пропускания

До установления каналов передач из хоста в конечную точку какого-либо USB-устройства системный драйвер должен сначала определить, может ли шина обеспечить требуемую полосу пропускания для данной точки. В каждом USB-устройстве есть специальная таблица, содержащая дескрипторы

конечных точек, в которых хранится значение минимально допустимой полосы пропускания для нее.

В фазе начальной инициализации системный драйвер читает эти дескрипторы и определяет необходимую суммарную полосу пропускания для USB-устройства. Хранящееся в дескрипторе значение определяет, какая доля пропускной способности шины необходима для работы конечной точки. При этом, однако, не учитываются никакие накладные расходы. Определяя общую потребность для поддержки канала к каждой конечной точке, системный драйвер USB учитывает следующее:

- число байтов данных;
- тип передачи данных;
- время восстановления хоста;
- время заполнения битами;
- уровень вложенности топологии.

Назначение логических адресов

Логическое USB-устройство представляет собой набор независимых конечных точек (*см. разд. 1.1.12*), с которыми клиентское ПО обменивается информацией. Каждому логическому USB-устройству (как функции, так и хабу) назначается свой адрес (1—127), уникальный на данной шине USB. Каждая конечная точка логического USB-устройства идентифицируется своим номером (0—15) и направлением передачи (IN — передача к хосту, OUT — от хоста).

Планирование транзакций

Транзакция на шине USB — это последовательность обмена пакетами между хостом и ПУ, в ходе которой может быть передан или принят один пакет данных. Когда клиентское ПО передает IRP системному драйверу USB, то он преобразует их в одну или несколько транзакций шины и затем передает получившийся перечень транзакций драйверу контроллера хоста.

Архитектура системного драйвера USB

Системный драйвер USB состоит из *драйвера USB* и *драйвера контроллера хоста*. Драйвер контроллера хоста принимает от системного драйвера перечень транзакций и выполняет следующие действия:

- планирует исполнение полученных транзакций, добавляя их к списку транзакций;
- извлекает из списка очередную транзакцию и передает ее на уровень хост-контроллера интерфейса шины USB;
- отслеживает состояние каждой транзакции вплоть до ее завершения.

При выполнении всех связанных с командным пакетом транзакций системный уровень уведомляет об этом клиентский уровень.

1.1.8.3. Уровень хост-контроллера интерфейса

Уровень хост-контроллера интерфейса шины USB получает отдельные транзакции от драйвера контроллера хоста (в составе уровня системного обеспечения USB) и преобразует их в соответствующую последовательность операций шины. В результате этого USB-пакеты передаются вдоль всей физической иерархии хабов (на рис. 1.4 мы изобразили последовательность хабов как одну логическую линию, но физически это может быть как один USB-кабель, так и последовательность хабов) до периферийного USB-устройства (правая часть рис. 1.4).

1.1.8.4. Уровень шины периферийного USB-устройства

Нижний уровень периферийного USB-устройства называется уровнем интерфейса шины USB. Он взаимодействует с интерфейсным уровнем шины USB на стороне хоста и передает пакеты данных от хоста в формате, определяемом спецификацией USB. Затем он передает пакеты вверх — уровню логического USB-устройства.

1.1.8.5. Уровень логического USB-устройства

Средний уровень периферийного USB-устройства называется уровнем логического USB-устройства. Каждое логическое USB-устройство представляется набором своих конечных точек, с которыми может взаимодействовать системный уровень хоста. Эти точки являются источниками и приемниками всех коммуникационных потоков между хостом и периферийными USB-устройствами.

1.1.8.6. Функциональный уровень USB-устройства

Самый верхний уровень периферийного USB-устройства называется функциональным уровнем. Этот уровень соответствует уровню клиентского обеспечения хоста. С точки зрения клиентского уровня, нижележащие уровни нужны для организации между ним и конечными точками прямых *каналов данных* (*см. разд. 1.1.13*), которые идут вплоть до функционального уровня периферийного USB-устройства. А с точки зрения нашей схемы функциональный уровень выполняет следующие действия:

- получает данные, посылаемые клиентским уровнем хоста из конечных точек каналов данных нижележащего уровня логического USB-устройства;
- посылает данные клиентскому уровню хоста, направляя их в конечные точки каналов данных нижележащего уровня логического USB-устройства.

1.1.9. Передача данных по уровням

Логически передача данных между конечной точкой и ПО производится с помощью выделения канала и обмена данными по этому каналу, а с точки зрения представленных уровней передача данных выглядит следующим образом (рис. 1.5).

1. Клиентское ПО посылает IPR-запросы на уровень системного драйвера USB.
2. Системный драйвер USB разбивает запросы на транзакции по следующим правилам:
   - выполнение запроса считается законченным, когда успешно завершены все транзакции, его составляющие;
   - все подробности отработки транзакций (такие как ожидание готовности, повтор транзакции при ошибке, неготовность приемника и т. д.) до клиентского ПО не доводятся;
   - ПО может только запустить запрос и ожидать или выполнения запроса, или выхода по тайм-ауту;
   - устройство может сигнализировать о серьезных ошибках (*см. разд. 1.1.14*), что приводит к аварийному завершению запроса, о чем уведомляется источник запроса.

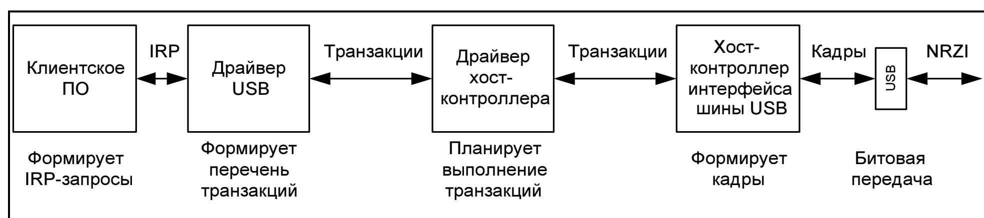


Рис. 1.5. Уровни передачи данных

3. Драйвер контроллера хоста принимает от системного драйвера USB перечень транзакций и выполняет следующие действия:
   - планирует исполнение полученных транзакций, добавляя их к списку транзакций;
   - извлекает из списка очередную транзакцию и передает ее уровню хост-контроллера интерфейса шины USB;
   - отслеживает состояние каждой транзакции вплоть до ее завершения.
4. Хост-контроллер интерфейса шины USB формирует кадры.

5. Кадры передаются последовательной передачей бит по методу, называемому *NRZI with bit stuffing* (Non Return to Zero Invert, метод возврата к нулю с инвертированием единиц).

Таким образом, можно сформировать следующую *упрощенную* схему (рис. 1.6):

- каждый кадр состоит из наиболее приоритетных посылок, состав которых формирует драйвер контроллера хоста;
- каждая передача состоит из одной или нескольких транзакций (*см. разд. 1.1.16*);
- каждая транзакция состоит из пакетов;
- каждый пакет состоит из идентификатора пакета, данных (если они есть) и контрольной суммы.

В следующих разделах мы рассмотрим все составляющие передачи более подробно.

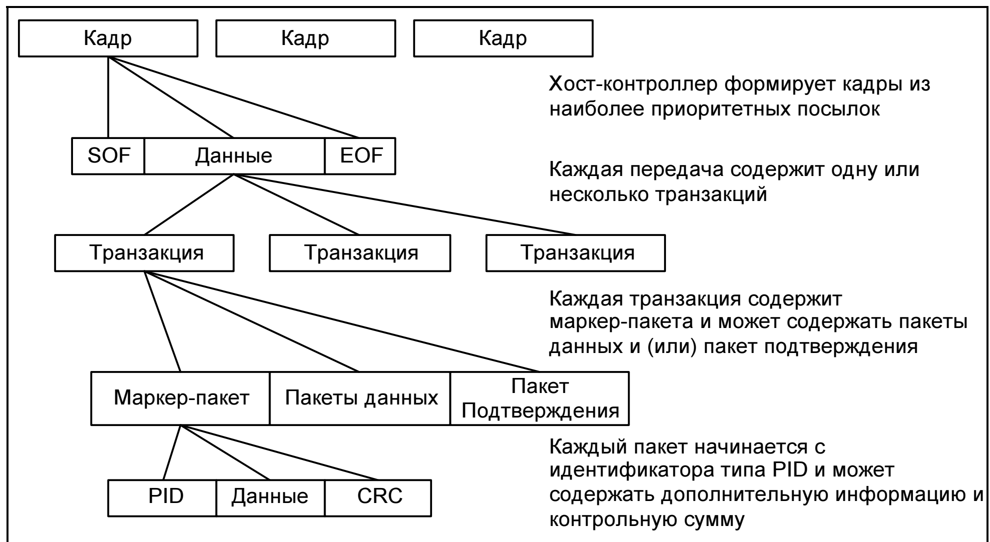


Рис. 1.6. Общая схема составляющих USB-протокола

1.1.10. Типы передач данных

Спецификация шины определяет четыре различных типа передачи (transfer type) данных для конечных точек (табл. 1.1):

- *управляющие передачи* (control transfers) — используются хостом для конфигурирования устройства во время подключения, для управления уст-

ройством и получения статусной информации в процессе работы. Протокол обеспечивает гарантированную доставку таких посылок. Длина поля данных управляющей посылки не может превышать 64 байта на полной скорости и 8 байтов на низкой. Для таких посылок хост гарантированно выделяет 10% полосы пропускания;

- *передачи массивов данных* (bulk data transfers) — применяются при необходимости обеспечения гарантированной доставки данных от хоста к функции или от функции к хосту, но время доставки не ограничено. Такая передача занимает всю доступную полосу пропускания шины. Пакеты имеют поле данных размером 8, 16, 32 или 64 байт. Приоритет у таких передач самый низкий, они могут приостанавливаться при большой загрузке шины. Допускаются только на полной скорости передачи. Такие посылки используются, например, принтерами или сканерами;
- *передачи по прерываниям* (interrupt transfers) — используются в том случае, когда требуется передавать одиночные пакеты данных небольшого размера. Каждый пакет требуется передать за ограниченное время. Операции передачи носят спонтанный характер и должны обслуживаться не медленнее, чем того требует устройство. Поле данных может содержать до 64 байтов при передаче на полной скорости и до 8 байтов на низкой. Предел времени обслуживания устанавливается в диапазоне 1—255 мс для полной скорости и 10—255 мс — для низкой. Такие передачи используются в устройствах ввода, таких как мышь и клавиатура;
- *изохронные передачи* (isochronous transfers) — применяются для обмена данными в "реальном времени", когда на каждом временном интервале требуется передавать строго определенное количество данных, но доставка информации не гарантирована (передача данных ведется без повторения при сбоях, допускается потеря пакетов). Такие передачи занимают предварительно согласованную часть пропускной способности шины и имеют заданную задержку доставки. Изохронные передачи обычно используются в мультимедийных устройствах для передачи аудио- и видеоданных, например, цифровая передача голоса. Изохронные передачи разделяются по способу синхронизации конечных точек — источников или получателей данных — с системой: различают асинхронный, синхронный и адаптивный классы устройств, каждому из которых соответствует свой тип канала USB.

Таблица 1.1. Типы передач по шине USB

Тип передачи	Направление	Частота запуска	Гарантия доставки
Управляющие посылки	Двунаправленные	Не гарантируется	Есть
Изохронные (только для высокоскоростных устройств)	Однонаправленные	Каждые 1 мс	Нет

Таблица 1.1 *(окончание)*

Тип передачи	Направление	Частота запуска	Гарантия доставки
Передача по прерываниям	Однонаправленные	Определяется частотой опроса	Есть
Передача массивов данных	Двунаправленные	Не гарантируется	Есть

Все операции по передаче данных инициируются только хостом независимо от того, принимает ли он данные или пересылает в периферийное USB-устройство. Все невыполненные операции хранятся в виде четырех списков по типам передач. Списки постоянно обновляются новыми запросами. Планирование операций по передаче информации в соответствии с упорядоченными в виде списков запросами выполняется хостом с интервалом один кадр. Обслуживание запросов выполняется по следующим правилам:

- наивысший приоритет имеют изохронные передачи;
- после отработки всех изохронных передач система переходит к обслуживанию передач прерываний;
- в последнюю очередь обслуживаются запросы на передачу массивов данных;
- по истечении 90% указанного интервала хост автоматически переходит к обслуживанию запросов на передачу управляющих команд независимо от того, успел ли он полностью обслужить другие три списка или нет.

Выполнение этих правил гарантирует, что управляющим передачам всегда будет выделено не менее 10% пропускной способности шины USB. Если передача всех управляющих пакетов будет завершена до истечения выделенной для них доли интервала планирования, то оставшееся время будет использовано хостом для передач массивов данных. Таким образом:

- изохронные передачи гарантированно получают 90% пропускной способности шины;
- передачи прерываний занимают оставшуюся часть этой доли;
- под передачу данных большого объема выделяется все время, оставшееся после изохронных передач и передач прерываний (по-прежнему в рамках 90% пропускной способности);
- управляющим передачам гарантируется 10% пропускной способности шины;
- если передача всех управляющих пакетов будет завершена до завершения выделенного для них 10-процентного интервала, то оставшееся время будет использовано для передач данных большого объема.

1.1.11. Кадры

Любой обмен по шине USB инициируется хостом. Он организует обмены с устройствами согласно своему плану распределения ресурсов.

Контроллер циклически (с периодом 1,0±0,0005 мс) формирует *кадры* (frames), в которые укладываются все запланированные передачи (рис. 1.7). Каждый кадр начинается с посылки маркер-пакета SOF (Start Of Frame, начало кадра, *см. разд. 1.1.14*), который является синхронизирующим сигналом для всех устройств, включая хабы. В конце каждого кадра выделяется интервал времени EOF (End Of Frame, конец кадра), на время которого хабы запрещают передачу по направлению к контроллеру. Если хаб обнаружит, что с какого-то порта в это время ведется передача данных, то он отключит этот порт.

В режиме высокоскоростной передачи (*см. разд. 1.1.10*) пакеты SOF передаются в начале каждого *микрокадра* (период 125±0,0625 мкс). Хост планирует загрузку кадров так, чтобы в них всегда находилось место для наиболее приоритетных передач, а свободное место кадров заполняется низкоприоритетными передачами больших объемов данных. Спецификация USB позволяет занимать под периодические транзакции (изохронные и прерывания) до 90% пропускной способности шины.

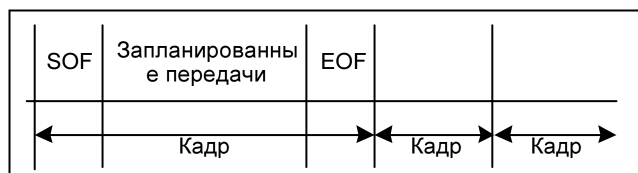


Рис. 1.7. Поток кадров USB

Каждый кадр имеет свой номер. Хост оперирует 32-битным счетчиком, но в маркере SOF передает только младшие 11 бит (*см. разд. 1.1.14*). Номер кадра циклически увеличивается во время EOF.

Для изохронной передачи (*см. разд. 1.1.10*) важна синхронизация устройств и контроллера. Есть три варианта синхронизации:

- синхронизация внутреннего генератора устройства с маркерами SOF;
- подстройка частоты кадров под частоту устройства;
- согласование скорости передачи (приема) устройства с частотой кадров.

В каждом кадре может быть выполнено несколько транзакций, их допустимое число зависит от скорости, длины поля данных каждой из них, а также от задержек, вносимых кабелями, хабами и USB-устройствами. Все транзак-

ции кадров должны быть завершены до момента времени EOF. Частота генерации кадров может немного варьироваться с помощью специального регистра хоста, что позволяет подстраивать частоту для изохронных передач. Подстройка частоты кадров контроллера возможна под частоту внутренней синхронизации только одного USB-устройства.

1.1.12. Конечные точки

Конечная точка (endpoint) — это часть USB-устройства, которая имеет уникальный идентификатор и является получателем или отправителем информации, передаваемой по шине USB. Проще говоря, это буфер, сохраняющий несколько байт. Обычно это блок данных в памяти или регистр микроконтроллера. Данные, хранящиеся в конечной точке, могут быть либо принятыми данными, либо данными, ожидающими передачу. В хосте также присутствует буфер для приема и передачи данных, но отсутствуют конечные точки. Конечная точка имеет следующие основные параметры:

- частота доступа к шине;
- допустимая величина задержки обслуживания;
- требуемая ширина полосы пропускания канала;
- номер;
- способ обработки ошибок;
- максимальный размер пакета, который может быть принят или отправлен;
- используемый тип посылок;
- направление передачи данных.

Любое USB-устройство имеет *конечную точку с нулевым номером* или *нулевую точку* (endpoint zero). Эта точка позволяет хосту опрашивать USB-устройство с целью определения его типа и параметров, а также выполнять его инициализацию и конфигурирование.

Кроме нулевой точки, USB-устройства обычно имеют дополнительные конечные точки, которые используются для обмена данными с хостом. Дополнительные точки могут работать либо только на прием данных от хоста (*входные точки*, IN), либо только на передачу данных хосту (*выходные точки*, OUT). Число дополнительных конечных точек определяется режимом передачи (*см. разд. 1.1.7*). Для низкоскоростных USB-устройств допускается наличие одной или двух дополнительных конечных точек, а для высокоскоростных — до 15 входных и 15 выходных дополнительных точек.

Нулевая точка становится доступна после того, как USB-устройство подключено к шине, включено и получило сигнал сброса по шине (bus reset). Все остальные конечные точки после включения питания или сброса нахо-

дятся в неопределенном состоянии и недоступны для работы до тех пор, пока хост не выполнит процедуру конфигурирования.

1.1.13. Каналы

Канал (pipe[1]) — это логическое соединение между конечной точкой USB-устройства и ПО хоста (рис. 1.8).

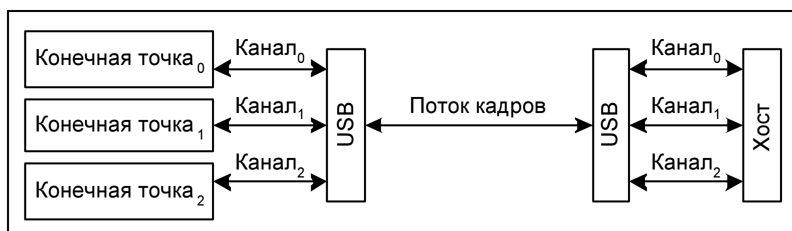


Рис. 1.8. Каналы USB

Существуют две модели каналов:

- *потоковый канал* (или просто *поток*, streaming pipe) — это канал для передачи данных, структура которых определяется клиентским ПО. Потоки используются для передачи массивов данных, передачи данных по прерываниям и изохронной передачи данных. Поток всегда однонаправленный. Один и тот же номер конечной точки может использоваться для двух разных потоковых каналов — ввода и вывода. Передачи данных в потоковых каналах подчиняются следующим *правилам синхронизации*:
  - запросы клиентских драйверов для разных каналов, поставленные в определенном порядке друг относительно друга, могут выполняться в другом порядке;
  - запросы для одного канала будут исполняться строго в порядке их поступления;
  - если во время выполнения какого-либо запроса происходит серьезная ошибка (STALL), поток останавливается;
- *канал сообщений* (message pipe или control pipe) — это канал для передачи данных, структура которых определяется спецификацией USB. Каналы этого типа двунаправленные и применяются для передачи управляющих посылок. Каналы сообщений строго синхронизированы — спецификация USB запрещает одновременную обработку нескольких запросов: нельзя

---

[1] Слово "pipe" дословно означает "труба", но мы будем пользоваться более подходящим и благозвучным переводом "канал".

начинать передачу нового сообщения, пока не завершена обработка предыдущего. В случае возникновения ошибки передача сообщения может быть прервана хостом, после чего хост может начать передачу нового сообщения.

Основными характеристиками каналов являются:

- полоса пропускания канала;
- используемый каналом тип передачи данных;
- характеристики, соответствующие конечной точке: направление передачи данных и максимальный размер пакета.

Полоса пропускания шины делится между всеми установленными каналами. Выделенная полоса закрепляется за каналом, и если установление нового канала требует такой полосы, которая не вписывается в уже существующее распределение, запрос на выделение канала отвергается. Архитектура USB предусматривает внутреннюю буферизацию всех USB-устройств, причем, чем большая полоса пропускания требуется, тем больше должен быть буфер. Шина должна обеспечивать обмен с такой скоростью, чтобы задержка данных в устройстве, вызванная буферизацией, не превышала нескольких миллисекунд.

Канал сообщений, связанный с нулевой конечной точкой, называется *основным каналом сообщений* (default control pipe или control pipe 0). Владельцем этого канала является USBD, и он используется для конфигурирования USB-устройства. Основной канал сообщений поддерживает только управляющие передачи. Остальные каналы (они называются *клиентскими каналами*, client pipe) создаются в процессе конфигурирования. Их владельцами являются драйверы USB-устройств. По клиентским каналам могут передаваться как потоки, так и сообщения с помощью любых типов передач.

Набор клиентских каналов, с которыми работает драйвер USB-устройства, называется *интерфейсом устройства* или *связкой клиентских каналов* (pipe's bundle).

1.1.14. Пакеты

Информация по каналу передается в виде *пакетов* (packet, рис. 1.9). Каждый пакет начинается с *поля синхронизации* `SYNC` (SYNChronization), за которым следует *идентификатор пакета* `PID` (Packet IDentifier), значения которого приведены в табл. 1.2. Поле `Check` представляет собой побитовую инверсию `PID`.

SYNC	PID	Check	Данные пакета	EOP

Рис. 1.9. Структура пакета

Таблица 1.2. *Список кодов PID*

Обозначение	Код PID	Источник	Описание
Идентификаторы маркер-пакетов (Token Packet)			
OUT	0001b	Хост	Маркер транзакции вывода, передает адрес и номер конечной точки при передаче от хоста к функции
IN	1001b		Маркер транзакции ввода, передает адрес и номер конечной точки при передаче от функции к хосту
SOF	0101b		Маркер начала кадра, содержит номер кадра
SETUP	1101b		Маркер транзакции управления: передает адрес и номер конечной точки при передаче команды от хоста к функции
Идентификаторы пакетов данных (Data Packet)			
Data0	0011b	Хост, USB-устройство	Пакеты данных с четным и нечетным PID, чередуются для точной идентификации подтверждений
Data1	1011b		
Data2	0111b		Дополнительные типы пакетов данных, используемые в транзакциях с широкополосными изохронными точками (в USB 2.0 для HS)
MData	1111b		
Идентификаторы пакетов подтверждений (Handshake)			
ACK	0010b	Хост, USB-устройство	Подтверждение безошибочного приема пакета
NAK	1010b	USB-устройство	Приемник не сумел принять или передатчик не сумел передать данные. Может использоваться для управления потоком данных ("ответ на запрос не готов"). В транзакциях прерываний является признаком отсутствия необслуживаемых прерываний
STALL	1110b	USB-устройство	Произошел сбой в конечной точке или запрос не поддерживается, требуется вмешательство хоста
NYET	0110b	USB-устройство	Подтверждение безошибочного приема, но указание на отсутствие места для приема следующего пакета максимального размера (USB 2.0)

Таблица 1.2 *(окончание)*

Обозна-чение	Код PID	Источник	Описание
Идентификаторы специальных пакетов (Special Packet)			
PRE	1100b	Хост	Специальный маркер, сообщающий, что следующий пакет будет передаваться в режиме LS (разрешает трансляцию данных на низкоскоростной порт хаба)
ERR		USB-устройство, хаб	Сигнализация ошибки в расщепленной транзакции (USB 2.0)
SPLIT (SS/CS)	1000b	Хост	Маркер расщепленной транзакции (USB 2.0). В зависимости от назначения обозначается как SS (маркер запуска) и CS (маркер завершения), назначение определяется битом SC в теле маркера
PING	0100b	Хост	Пробный маркер высокоскоростного управления потоком (USB 2.0)
RESERV	0000b		Зарезервированный PID

Из табл. 1.2 видно, что два младших бита идентификатора определяют группу, к которой он принадлежит:

- 00 — специальный пакет;
- 01 — маркер-пакет;
- 10 — пакет подтверждение;
- 11 — пакет данных.

Структура данных пакета зависит от группы, к которой он относится.

1.1.14.1. Формат маркер-пакетов IN, OUT, SETUP и PING

Поле данных пакетов типа IN, OUT, SETUP и PING содержит следующие поля (полный формат пакета показан на рис. 1.10):

- [7] адрес функции;
- [4] адрес конечной точки;
- [5] циклический контрольный код.

Маркер транзакции отмечает начало очередной транзакции на шине USB и позволяет адресовать до 127 функций USB (нулевой адрес используется для конфигурирования) и по 16 конечных точек в каждой функции. Поле `CRC` вычисляется по полям `Func` и `EndP`.

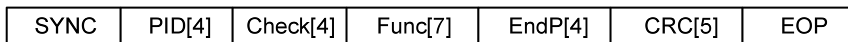


Рис. 1.10. Формат пакета типов IN, OUT, SETUP и PING

1.1.14.2. Формат пакета SOF

Поле данных пакета SOF содержит (полный формат пакета показан на рис. 1.11):

- [11] номер кадра;
- [5] циклический контрольный код.

Пакет SOF используется для отметки начала кадра (*см. разд. 1.1.11*). Хотя хост оперирует 32-битным счетчиком кадров, в маркере SOF передаются только младшие 11 бит. Контрольная сумма вычисляется по 11 битам поля `Frame`.

Рис. 1.11. Формат пакета типа SOF

1.1.14.3. Формат пакета данных

В поле данных пакетов типа Data0, Data1, Data2 и MData может содержаться от 0 до 1023 байт данных, за которыми следует 16-разрядный циклический контрольный код (*см. разд. 1.1.15*), вычисленный по полю `Data`. Пакет данных всегда должен посылать целое число байт. Для LS режима максимальный размер пакета равен 8 байтам, для FS — 1023 байта, а для HS — 1024 байта. Полный формат пакета данных показан на рис. 1.12.

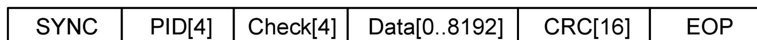


Рис. 1.12. Формат пакетов данных

1.1.14.4. Формат пакета подтверждения

Поле данных пакетов подтверждения пустое. Полный формат пакетов подтверждения показан на рис. 1.13.

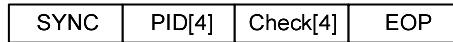


Рис. 1.13. Формат пакета подтверждения

1.1.14.5. Формат пакета SPLIT

Поле данных пакета SPLIT содержит (полный формат пакета показан на рис. 1.14):

- [7] адрес хаба;
- [1] флаг `SC`;
- [7] адрес порта;
- [1] поле `S`;
- [1] поле `E`;
- [1] тип конечной точки `ET`;
- [5] циклический контрольный код.

Флаг `SC` (Start/Complete) принимает значение 0, если это *маркер запуска* (SS, Start Split) расщепленной транзакции и 1, если это *маркер завершения* (CS, Complete Split).

В зависимости от типа транзакции, поля `S` и `E` трактуются по-разному. Для управляющих транзакций и прерываний поле `S` определяет скорость (0 — FS, 1 — LS). Для остальных транзакций, кроме OUT, поля `S` и `E` содержат нули. Для транзакции OUT значение полей `[S:E]` трактуется следующим образом:

- `10` — стартовый пакет;
- `01` — последний пакет;
- `00` — промежуточный пакет;
- `11` — в пакете все данные транзакции.

Поле `ET` описывает тип целевой конечной точки, с которой будет производиться транзакция:

- 00 — управление;
- 01 — изохронная транзакция;

- 10 — передача массива данных;
- 11 — прерывание.

Контрольная сумма вычисляется по полям от `HubAddr` до `ET` включительно.

SYNC	PID[4]	Check[4]	HubAddr[7]	CS[1]	Port[7]	S[1]	E[1]	ET[2]	CRC[5]	EOP

Рис. 1.14. Формат пакета SPLIT

1.1.15. Контрольная сумма

Протокол USB использует *циклический избыточный код* (CRC, Cyclic Redundancy Checksums) для защиты полей пакета. CRC-контроль является более мощным методом обнаружения ошибок и используется для обнаружения ошибок на уровне блоков данных. Он основан на делении и умножении многочленов. В определенном смысле CRC-контроль является *алгоритмом хэширования*, который отображает (хэширует) элементы большого набора на элементы меньшего набора. Процесс хэширования приводит к потере информации. Хотя каждый отдельный элемент набора данных отображается на один и только один элемент хэш-набора — обратное не верно. При таком способе большой набор всех возможных двоичных чисел отображается на меньший набор всех возможных CRC.

1.1.15.1. Алгоритм вычисления CRC

Вычисление и использование CRC-кода производится в соответствии со следующей последовательностью действий. К содержимому кадра, описываемого полиномом $F(x)$, добавляется набор единиц, количество которых равно длине поля `CRC`:

$$L(x) = \sum_{n=0}^{15} x^n = 1111111111111111.$$

Образованное таким образом число $x^{16} \times F(x) + x^k \times L(x)$, где k — степень $F(x)$, делится на производящий полином $g(x)$. Остаток $O(x)$ от такого деления, определяется из соотношения:

$$Q(x)g(x) = x^{16} \times F(x) + x^k \times L(x) + O(x),$$

где $Q(x)$ — частное от деления $x^{16} \times F(x) + x^k \times L(x)$ на $g(x)$, в инвертированном виде помещается в контрольное поле кадра. На приемной стороне выполняется деление содержимого кадра с полем `CRC` $x^{16} \times F'(x) + x^k \times L(x) + O(x)$ на полином $g(x)$, где $F'(x)=x^{16} \times F(x) + L(x) + O(x)$ — передаваемая кодовая комбинация.

Результат такого деления можно привести к виду:

$$\frac{x^{16}[x^{16} \times F(x) + x^k \times L(x) + O(x)]}{g(x)} + \frac{x^{16} \times L(x)}{g(x)} = \frac{x^{16}[Q(x) \times g(x)]}{g(x)} + \frac{x^{16} \times L(x)}{g(x)}.$$

Числитель первого слагаемого делится на $g(x)$, поэтому в приемнике, если при передаче не было ошибок, остаток получается равным остатку от деления постоянного числителя второго слагаемого ($x^{16} \times L(x) / g(x)$) и имеет вид:

$$x^{12} + x^{11} + x^{10} + x^8 + x^3 + x^2 + x + 1 = 1110100001111.$$

Таким образом, если результат вычислений на приемной стороне равен некоторому определенному числу (в некоторых системах нулю, либо другому числу, не совпадающему с приведенным выше), то считается, что передача выполнена без ошибок.

При выборе порождающего полинома руководствуются желаемой разрядностью остатка и его способностью выявлять ошибки. Ряд порождающих полиномов принят международными организациями. В протоколе USB используются два порождающих полинома — один для пакетов маркеров и второй для пакетов данных. Для маркеров используется полином $x^5 + x^2 + x^0$, а для данных — $x^{16} + x^{15} + x^2 + x^0$. Соответственно, получаемый контрольный код имеет размерность 5 битов и 16 битов.

Алгоритм вычисления 16-разрядной контрольной суммы (ее более привычное название — CRC16) выглядит следующим образом. В 16-битовый регистр 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 данная операция не производится.

Этот процесс повторяется, пока не будет сделано 8 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.

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 1 бит, старший бит заполняется нолями.
4. Если младший бит равен нулю, повторяется шаг 3 (сдвиг).
5. Если младший бит равен единице, производится операция исключающего ИЛИ над регистром CRC и полиномиальным числом `$A001`.
6. Шаги 3 и 4 повторяются восемь раз.
7. Повторяются шаги со 2 по 5 для следующего сообщения. Это повторяется до тех пор, пока все байты сообщения не будут обработаны.

Финальное содержание регистра CRC и есть контрольная сумма.

1.1.15.2. Программное вычисление CRC

При вычислении следует учитывать порядок передачи битов в протоколе USB. Листинг 1.1 показывает пример вычисления CRC5 и CRC16 для битовых последовательностей, а рис. 1.15 — результат выполнения этой программы.

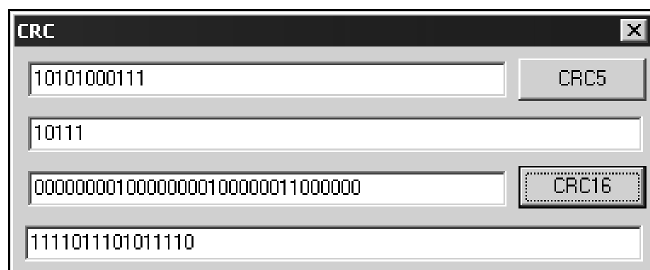


Рис. 1.15. Вычисление CRC5 и CRC16

Листинг 1.1. Вычисление CRC5 и CRC16 для битовых последовательностей

```
unit Unit1;

interface

uses
 Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
 Forms, Dialogs, StdCtrls;
```

```
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    btnCRC5: TButton;
    E_CRC5: TEdit;
    Edit2: TEdit;
    btnCRC16: TButton;
    E_CRC16: TEdit;
    procedure btnCRC5Click(Sender: TObject);
    procedure btnCRC16Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

// Сдвиг битовой строки влево:
//  результат - самый левый символ
//  удаляет первый символ
//  добавляет в конец 0
function Shift(var S : String) : Char;
begin
 Result:= S[1];
 Delete(S, 1, 1);
 S:= S + '0';
end;

// Операция XOR для битовых строк
procedure XorStr(var S : String; XStr : String);
```

```
var i : Byte;
begin
 for i:= 1 to Length(XStr) do begin
  if S[i] = XStr[i] then S[i]:= '0' else S[i]:= '1';
 End;
end;

// Вычисление CRC5
procedure TForm1.btnCRC5Click(Sender: TObject);
var S : String; i : Byte;
    Result : String;
begin
 Result:= '11111';

 S:= Edit1.Text;
 For i:= 1 to Length(S) do
  If S[i] <> Shift(Result) then
   XorStr(Result, '00101');
 XorStr(Result, '11111');

 E_CRC5.Text:= Result;
end;

// Вычисление CRC16
procedure TForm1.btnCRC16Click(Sender: TObject);
var S : String; i : Byte;
    Result : String;
begin
 Result:= '1111111111111111';

 S:= Edit2.Text;
 For i:= 1 to Length(S) do
  If S[i] <> Shift(Result) then
   XorStr(Result, '1000000000000101');
 XorStr(Result, '1111111111111111');
```

```
 E_CRC16.Text:= Result;
end;


end.
```

1.1.16. Транзакции

Все обмены (*транзакции*) по USB состоят из трех пакетов (рис. 1.16). Каждая транзакция планируется и начинается по инициативе хоста, который посылает маркер-пакет, в котором описываются тип и направление передачи, адрес USB-устройства и номер конечной точки. В каждой транзакции возможен обмен только между USB-устройством (его конечной точкой) и хостом.

Адресуемое маркером USB-устройство распознает свой адрес и готовится к обмену. Источник данных, определенный маркером, передает пакет данных или уведомление об отсутствии данных, предназначенных для передачи. После успешного приема пакета приемник данных посылает пакет-подтверждение.

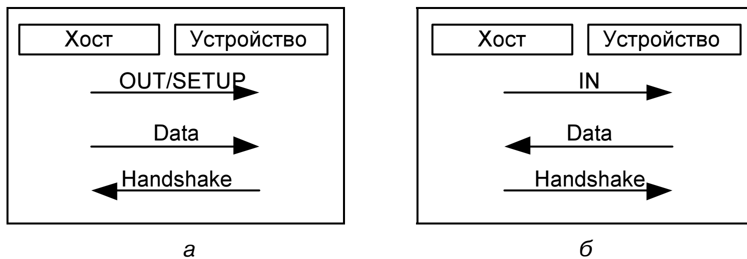


Рис. 1.16. *а* — передача данных от хоста; *б* — передача данных к хосту

Периферийное USB-устройство не может выдавать на шину какую-либо информацию по собственной инициативе и не может самостоятельно посылать запросы прерываний.

1.1.16.1. Типы транзакций

Спецификация USB определяет следующие типы транзакций:

- ❒ передача команды:
  - хост посылает маркер SETUP, содержащий номер функции и номер конечной точки, для которой предназначена команда;

- хост посылает выбранной конечной точке пакет данных со сброшенным битом синхронизации (т. е. пакет типа Data0), содержащий 8-байтный код команды;
- функция посылает хосту пакет подтверждения;

- ❒ изохронная передача данных:
  - хост посылает маркер OUT, содержащий номер функции и номер конечной точки, для которой предназначены данные;
  - хост посылает выбранной конечной точке пакет данных со сброшенным битом синхронизации (т. е. пакет типа Data0);
- ❒ передача данных с подтверждением:
  - хост посылает маркер OUT, содержащий номер функции и номер конечной точки, для которой предназначены данные;
  - хост посылает выбранной конечной точке пакет данных;
  - функция посылает хосту пакет подтверждения;
- ❒ изохронный прием данных:
  - хост посылает маркер IN, содержащий номер функции и номер конечной точки, от которой запрашиваются данные;
  - выбранная конечная точка передает хосту пакет данных со сброшенным битом синхронизации (т. е. пакет типа Data0);
- ❒ прием данных с подтверждением:
  - хост посылает маркер IN, содержащий номер функции и номер конечной точки, от которой запрашиваются данные;
  - выбранная конечная точка передает хосту пакет данных или пакет подтверждения (NAK — данные не готовы, STALL — сбой);
  - если хост получил пакет данных, он посылает пакет подтверждения (ACK).

1.1.16.2. Подтверждение транзакций и управление потоком

При выполнении транзакций используются три типа пакетов подтверждения, или по другому — *пакетов квитирования*:

- ❒ ACK — информация принята получателем без ошибок, операция успешно завершена;
- ❒ NAK — функция занята (не готова к приему или передаче данных);
- ❒ STALL — произошел сбой при выполнении операции, функция не может принять или передать данные.

Если транзакция выполнена успешно — передается пакет ACK, а в случае ошибки возможно несколько вариантов:

- при неготовности USB-устройства к выполнению операции (нет данных для передачи хосту, отсутствует место в буфере для приема, не завершена операция управления и т. д.) передается пакет NAK, заставляющий хост повторить транзакцию позже. Такая ситуация, в общем-то, является вполне нормальной и клиентское ПО не уведомляется об ошибке;
- если при выполнении транзакции передачи данных с подтверждением, в пакете данных обнаружена ошибка по контрольному коду, получатель пакета не высылает пакет подтверждения. Отправитель при отсутствии подтверждения от получателя должен зафиксировать ошибку передачи данных и повторить транзакцию;
- при серьезной ошибке передается пакет STALL, обозначающий, что без вмешательства хоста работа с конечной точкой невозможна. В отличие от NAK, ответ STALL доводится до сведения драйвера USBD, который отменяет все дальнейшие транзакции с этой точкой, и до клиентского драйвера.

Ответ NAK позволяет только уведомить о невозможности приема данных, в то время как сам пакет данных уже отправлен. Такое поведение попусту растрачивает пропускную способность шины. В спецификации USB 2.0 добавлен еще один пакет, позволяющий более гибко управлять транзакциями передачи больших объемов данных — пакет PING. С помощью этого пакета хост может предварительно опросить готовность устройства к приему пакета максимального размера. В ответ на этот запрос USB-устройство должно либо ответить пакетом ACK, подтвердив готовность к приему пакета данных, либо пакетом NAK, если оно не готово. Отрицательный ответ заставит хост повторить пробу позже, а положительный разрешает ему выполнить транзакцию вывода данных. На транзакцию вывода данных после положительного ответа на пробу USB-устройство может ответить следующими пакетами:

- ACK — прием пакета успешен и устройство готово к приему следующего полноразмерного пакета;
- NYET — успешный прием, но неготовность к следующему пакету;
- NAK — неготовность USB-устройства (может случиться, что за время от пакета пробы до посылки пакета USB-устройство перестало быть готово к приему).

Высокоскоростное устройство в дескрипторах конечных точек сообщает о возможной интенсивности посылок NAK: поле `bInterval` для конечных точек передачи данных и управления указывает число микрокадров, приходящееся на один ответ NAK (0 означает, что устройство никогда не ответит посылкой пакета NAK на транзакцию вывода).

1.1.16.3. Протоколы транзакций

В зависимости от типа передачи данных каждая посылка состоит из одной или нескольких транзакций.

Управляющие посылки

Существуют три типа управляющих посылок (рис. 1.17):

- *посылка записи данных* (control write) состоит из следующих транзакций:
  - передача команды;
  - передача с подтверждением одного или нескольких пакетов данных;
  - прием с подтверждением пустого пакета данных, подтверждающего успешное завершение операции;
- *посылка чтения данных* (control read) состоит из следующих транзакций:
  - передача команды;
  - прием с подтверждением одного или нескольких пакетов данных;
  - передача с подтверждением пустого пакета данных, подтверждающего успешное завершение операции;
- *посылка без данных* (no-data control) состоит из следующих транзакций:
  - передача команды;
  - прием с подтверждением пустого пакета данных, подтверждающего успешное завершение операции.

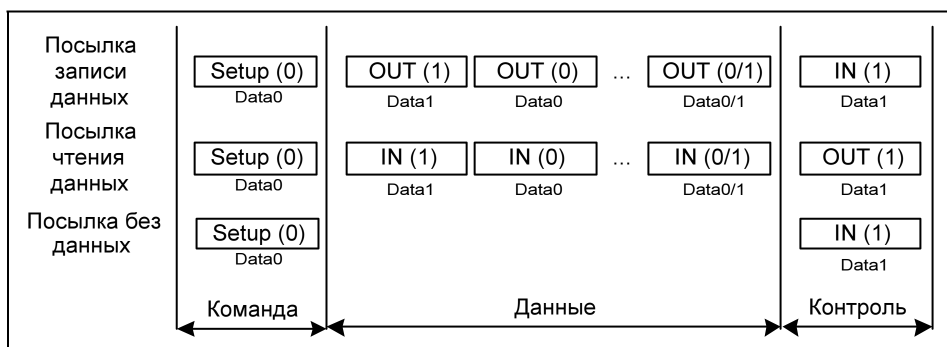


Рис. 1.17. Формат управляющих посылок

При выполнении транзакции передачи данных признак синхронизации данных должен быть сброшен в ноль (блок данных, содержащий код команды, имеет тип Data0). Если команда предполагает прием или передачу дан-

ных, то после каждой транзакции признак синхронизации данных инвертируется: первый блок имеет тип Data1, второй — Data0, третий — Data1 и т. д. Пустой пакет данных, подтверждающий завершение управляющей посылки, должен иметь тип Data1.

При передаче управляющей посылки максимальный размер пакета для полноскоростного устройства может составлять 8, 16, 32 или 64 байта, а для низкоскоростного всегда равен 8 байтам. Для передачи сообщений по Основному каналу сообщений всегда используется максимальный размер пакета, равный 8 байтам.

Передачи массивов данных

Существуют два типа передачи массивов (рис. 1.18):

- ❒ передача массива данных от хоста к конечной точке (bulk write);
- ❒ прием хостом массива данных от конечной точки (bulk read).

Передача данных от хоста к конечной точке состоит из следующих друг за другом транзакций передачи данных с подтверждением, а передача данных от конечной точки к хосту — из следующих друг за другом транзакций приема с подтверждением. И в том и в другом случае перед началом передачи массива триггер синхронизации данных должен быть сброшен в 0: при выполнении первой транзакции блок данных имеет тип Data0, второй — Data1, третий — Data0 и т. д.

Прием и передачу массивов данных могут выполнять только полноскоростные устройства. Максимальный размер пакета при передаче массива может быть равен 8, 16, 32 или 64 байтам.

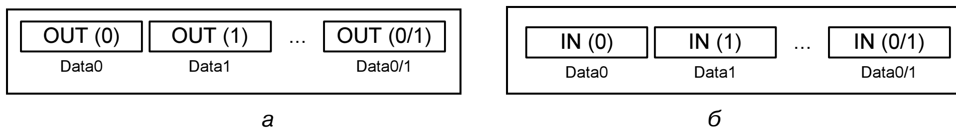


Рис. 1.18. Формат посылок передачи данных:
а — передача массива; *б* — прием массива

Передачи по прерываниям

Существуют два типа передачи по прерываниям:

- ❒ передача массива данных от хоста к конечной точке по прерыванию;
- ❒ прием хостом массива данных от конечной точки по прерыванию.

Передача данных по прерыванию заключается в выполнении транзакции передачи пакета данных с подтверждением от хоста к конечной точке. При-

ем заключается в выполнении транзакции приема пакета данных с подтверждением от конечной точки. При приеме или передаче каждого блока данных происходит переключение триггера данных. Первый передаваемый (или принимаемый) блок должен иметь тип Data0, следующий — Data1 и т. д.

Максимальный размер пакета при передаче по прерыванию для низкоскоростных устройств не может быть более 8 байт, а для высокоскоростных — более 64 байт.

Изохронные передачи

Существуют два типа изохронной передачи:

- изохронная передача данных от хоста к конечной точке;
- изохронный прием данных хостом от конечной точки.

Изохронная передача данных заключается в выполнении транзакции передачи пакета данных без подтверждения от хоста к конечной точке. Изохронный прием заключается в выполнении транзакции приема пакета данных без подтверждения от конечной точки.

Состояние триггера данных при изохронной передаче игнорируется, но рекомендуется сбросить его в ноль перед началом передачи. Изохронную передачу могут выполнять только полноскоростные устройства. Максимальный размер пакета данных при изохронной передаче — 1023 байта.

1.2. Запросы к USB-устройствам

Все USB-устройства принимают запросы от хоста и отвечают на них через основной канал сообщений (*см. разд. 1.1.13*). Запросы выполняются при помощи управляющих посылок (*см. разд. 1.1.16.3*).

1.2.1. Конфигурационный пакет

Запрос и его параметры передаются устройству в *конфигурационном пакете* (Setup Packet). Конфигурационный пакет имеет размер 8 байт (табл. 1.3). Структура конфигурационного пакета на языке Pascal показана в листинге 1.2.

Таблица 1.3. *Конфигурационный пакет*

Смещение	Поле	Размер	Описание
0	`bmRequestType`	BYTE	Тип запроса
1	`bRequest`	BYTE	Код запроса

Таблица 1.3 *(окончание)*

Смещение	Поле	Размер	Описание
2	wValue	WORD	Параметр запроса
4	wIndex	WORD	Индекс или смещение
6	wLength	WORD	Число байт для передачи

Листинг 1.2. Конфигурационный пакет

```
TSetupPacket = packed record
   bmRequest : UCHAR;
   bRequest  : UCHAR;
   wValue    : Array [1..2] of UCHAR;
   wIndex    : Array [1..2] of UCHAR;
   wLength   : Array [1..2] of UCHAR;
 End;
```

Тип запроса `bmRequestType` имеет размер 1 байт и состоит из следующих битов:

- [7] направление передачи:
  - 0 — от хоста к USB-устройству;
  - 1 — от USB-устройства к хосту;
- [6:5] код типа запроса:
  - 0 — стандартный запрос;
  - 1 — специфический запрос для данного класса;
  - 2 — специфический запрос изготовителя;
  - 3 — зарезервирован;
- [4:0] код получателя:
  - 0 — USB-устройство;
  - 1 — интерфейс;
  - 2 — другой получатель;
  - 4—31 зарезервированы.

Поле кода запроса определяет операцию, выполняемую запросом. В спецификации USB определены только коды стандартных запросов к устройству (листинг 1.3).

Листинг 1.3. Коды стандартных запросов

```
// GET_STATUS (определение состояния устройства)
#define USB_REQUEST_GET_STATUS                  0x00
// CLEAR_FEATURE (сброс устройства)
#define USB_REQUEST_CLEAR_FEATURE               0x01
// код 2 зарезервирован
// SET_FEATURE (установить свойство)
#define USB_REQUEST_SET_FEATURE                 0x03
// код 4 зарезервирован
// SET_ADDRESS (установить адрес)
#define USB_REQUEST_SET_ADDRESS                 0x05
// GET_DESCRPTOR (получить дескриптор)
#define USB_REQUEST_GET_DESCRIPTOR              0x06
// SET_DESCRIPTOR (загрузить дескриптор)
#define USB_REQUEST_SET_DESCRIPTOR              0x07
// GET_CONFIGURATION (получить код текущей конфигурации)
#define USB_REQUEST_GET_CONFIGURATION           0x08
// SET_CONFIGURATION (установить конфигурацию)
#define USB_REQUEST_SET_CONFIGURATION           0x09
// GET_INTERFACE (получить код интерфейса)
#define USB_REQUEST_GET_INTERFACE               0x0A
// SET_INTERFACE (установить интерфейс)
#define USB_REQUEST_SET_INTERFACE               0x0B
// SYNC_FRAME (кадр синхронизации)
#define USB_REQUEST_SYNC_FRAME                  0x0C
```

Значение параметров `Value` и `Index` зависят от типа запроса. В запросах на прием или передачу дескрипторов параметр `Value` содержит тип дескриптора (листинг 1.4) в старшем байте и индекс дескриптора — в младшем.

Листинг 1.4. Типы дескрипторов

```
// Стандартный дескриптор устройства
#define USB_DEVICE_DESCRIPTOR_TYPE              0x01
// Дескриптор конфигурации
#define USB_CONFIGURATION_DESCRIPTOR_TYPE       0x02
```

```
// Дескриптор строки
#define USB_STRING_DESCRIPTOR_TYPE                0x03
// Дескриптор интерфейса
#define USB_INTERFACE_DESCRIPTOR_TYPE             0x04
// Дескриптор конечной точки
#define USB_ENDPOINT_DESCRIPTOR_TYPE              0x05
// Уточняющий дескриптор устройства
#define DEVICE_QUALIFIER                          0x06
// Дескриптор дополнительной конфигурации
#define OTHER_SPEED_CONFIGURATION                 0x07
// Дескриптор управления питанием интерфейса
#define INTERFACE_POWER                           0x08
// Дескриптор OTG
#define OTG                                       0x09
// Отладочный дескриптор
#define DEBUG                                     0x0A
// Дополнительный дескриптор интерфейса
#define INTERFACE_ASSOCIATION                     0x0B
```

Поле `Index` обычно используется для задания номера интерфейса или конечной точки. Если поле `Index` задает конечную точку, то оно имеет следующий формат:

- [15:8] зарезервированы и должны содержать нули;
- [7] направление передачи конечной точки:
  - 0 — выход (OUT, от хоста);
  - 1 — вход (IN, к хосту);
- [6:4] зарезервированы и должны содержать нули;
- [3:0] номер конечной точки.

Если поле `Index` задает номер интерфейса, то оно имеет следующий формат:

- [15:8] зарезервированы и должны содержать нули;
- [7:0] номер интерфейса.

Если USB-устройство получает некорректный или неподдерживаемый запрос, оно должно ответить пакетом типа STALL (*см. разд. 1.1.14*).

1.2.2. Стандартные запросы к USB-устройствам

1.2.2.1. Получение состояния *GET_STATUS*

Запрос `GET_STATUS` позволяет определить состояние USB-устройства, интерфейса или конечной точки. Поля запроса содержат следующие значения:

- `bmRequestType` — тип запроса:
  - `10000000b` — получить состояние USB-устройства;
  - `10000001b` — получить состояние интерфейса;
  - `10000010b` — получить состояние конечной точки;
- `bRequest` — `0x00`;
- `wValue` — `0`;
- `wIndex` — адресант запроса:
  - `0`, если запрос обращен к USB-устройству;
  - номер интерфейса, если запрос к интерфейсу;
  - номер конечной точки, если запрос к конечной точке;
- `wLength` — `2`.

По запросу `GET_STATUS` устройство возвращает 16-разрядное слово состояния, описывающее текущее состояние USB-устройства, интерфейса или конечной точки.

Слово состояния устройства имеет следующие биты:

- [15:2] зарезервированы и должны содержать нули;
- [1] — реакция на сигнал пробуждения от шины USB (remote wakeup):
  - 0 — USB-устройство игнорирует сигнал пробуждения;
  - 1 — USB-устройство реагирует на сигнал пробуждения;
- [0] — режим питания:
  - 0 — USB-устройство получает питание от шины USB;
  - 1 — USB-устройство получает питание от собственного источника.

Слово состояния интерфейса зарезервировано и содержит нули во всех разрядах.

Разряды *слова состояния конечной точки* имеют следующие значения:

- [15:1] зарезервированы и должны содержать нули;
- [0] — признак *блокировки* (halt) конечной точки:
  - 0 — конечная точка функционирует нормально;
  - 1 — передача данных заблокирована.

1.2.2.2. Сброс свойства *CLEAR_FEATURE*

Запрос `CLEAR_FEATURE` используется для запрета свойства или состояния, указываемого значением поля `wValue`. Поля запроса содержат следующие значения:

- `bmRequestType` — тип запроса:
  - `00000000b` — запретить свойство устройства;
  - `00000001b` — запретить свойство интерфейса;
  - `00000010b` — запретить свойство конечной точки;
- `bRequest` — `1`;
- `wValue` — код свойства;
- `wIndex` — адресант запроса:
  - `0`, если запрос обращен к USB-устройству;
  - номер интерфейса, если запрос к интерфейсу;
  - номер конечной точки, если запрос к конечной точке;
- `wLength` — `0`.

Поле `wValue` может принимать одно из трех значений:

- `0` — блокировка конечной точки (`ENDPOINT_HALT`, получатель — конечная точка);
- `1` — разрешить выполнение сигнала пробуждения (`DEVICE_REMOTE_WAKEUP`, получатель — устройство);
- `2` — тестовый режим (`TEST_MODE`, получатель — USB-устройство).

Передача данных по запросу `CLEAR_FEATURE` не производится. Сброс состояния `ENDPOINT_HALT` разблокирует конечную точку; сброс состояния `DEVICE_REMOTE_WAKEUP` лишает устройство способности реагировать на сигнал пробуждения.

1.2.2.3. Разрешение свойства *SET_FEATURE*

Запрос `SET_FEATURE` используется для разрешения свойства или состояния, указываемого значением поля `wValue`. Поля запроса содержат следующие значения:

- `bmRequestType` — тип запроса:
  - `00000000b` — разрешить свойство USB-устройства;
  - `00000001b` — разрешить свойство интерфейса;
  - `00000010b` — разрешить свойство конечной точки;

- bRequest — 0x03;
- wValue — код свойства;
- wIndex — адресант запроса:
  - 0, если запрос обращен к USB-устройству;
  - номер интерфейса, если запрос к интерфейсу;
  - номер конечной точки, если запрос к конечной точке;
- wLength — 0.

Передача данных по запросу SET_FEATURE не производится. Установка состояния ENDPOINT_HALT блокирует конечную точку; установка состояния DEVICE_REMOTE_WAKEUP позволяет устройству реагировать на сигнал пробуждения.

1.2.2.4. Задание адреса на шине *SET_ADDRESS*

Запрос SET_ADDRESS позволяет присвоить USB-устройству значение адреса на шине. Поля запроса содержат следующие значения:

- bmRequestType — 00000000b;
- bRequest — 0x05;
- wValue — адрес устройства;
- wIndex — 0;
- wLength — 0.

Передача данных при выполнении запроса SET_ADDRESS не производится. Следует отметить, что поле wValue имеет размер два байта, однако, адрес USB-устройства не может быть больше чем 127. Если адрес имеет значение больше 127 или поля wIndex или wLength не равны нулю, поведение USB-устройства не определено.

1.2.2.5. Получение дескриптора *GET_DESCRIPTOR*

Запрос GET_DESCRIPTOR позволяет получить дескриптор устройства, дескриптор конфигурации или дескриптор строки. Поля запроса содержат следующие значения:

- bmRequestType — 10000000b, 10000001b или 10000010b;
- bRequest — 0x06;
- wValue — тип дескриптора в старшем байте (см. листинг 1.4) и индекс дескриптора в младшем байте (при запросе дескриптора устройства индекс имеет значение 0);

- wIndex — 0 для дескриптора устройства или конфигурации, или идентификатор языка для дескриптора строки;
- wLength — размер дескриптора в байтах.

Подробную информацию о получении формата дескриптора см. в разд. 1.2.3.

Значение поля bmRequestType зависит от типа запрашиваемого дескриптора:

- 10000000b — если дескриптор относится к USB-устройству;
- 10000001b — если дескриптор относится к интерфейсу;
- 10000010b — если дескриптор относится к конечной точке.

Подробную информацию см. в разд. 5.3.2 и 6.4.3.

1.2.2.6. Передача дескриптора *SET_DESCRIPTOR*

Запрос SET_DESCRIPTOR позволяет дополнить существующий или добавить новый дескриптор USB-устройства, конфигурации или строки. Поля запроса содержат следующие значения:

- bmRequestType — 00000000b, 00000001b или 00000010b;
- bRequest — 0x07;
- wValue, wIndex, wLength — те же значения, что и для GET_DESCRIPTOR.

В процессе выполнения запроса SET_DESCRIPTOR хост передает USB-устройству дескриптор, тип которого определяется параметрами запроса.

1.2.2.7. Получение кода конфигурации *GET_CONFIGURATION*

По запросу GET_CONFIGURATION USB-устройство выдает код своей текущей конфигурации. Поля запроса содержат следующие значения:

- bmRequestType — 10000000b;
- bRequest — 0x08;
- wValue — 0;
- wIndex — 0;
- wLength — 1.

При выполнении запроса GET_CONFIGURATION от USB-устройства к хосту передается один байт данных, содержащий код конфигурации, которую требуется установить.

1.2.2.8. Задание кода конфигурации *SET_CONFIGURATION*

Запрос SET_CONFIGURATION позволяет задать USB-устройству новую конфигурацию. Поля запроса содержат следующие значения:

- bmRequestType — 00000000b;
- bRequest — 0x09;
- wValue — код конфигурации в младшем байте (старший байт зарезервирован);
- wIndex — 0;
- wLength — 0.

При выполнении запроса SET_CONFIGURATION данные не передаются.

1.2.2.9. Получение кода настройки интерфейса *GET_INTERFACE*

Запрос GET_INTERFACE позволяет получить код текущей настройки для указанного интерфейса. Поля запроса содержат следующие значения:

- bmRequestType — 10000001b;
- bRequest — 0x0A;
- wValue — 0;
- wIndex — номер интерфейса;
- wLength — 1.

При выполнении запроса GET_INTERFACE от USB-устройства к хосту передается один байт данных, содержащий код текущего варианта настройки интерфейса.

1.2.2.10. Задание кода настройки интерфейса *SET_INTERFACE*

Запрос SET_INTERFACE позволяет задать новый вариант настройки для указанного интерфейса. Поля запроса содержат следующие значения:

- bmRequestType — 00000001b;
- bRequest — 0x0B;
- wValue — код варианта настройки интерфейса;
- wIndex — номер интерфейса;
- wLength — 0.

При выполнении запроса SET_INTERFACE передача данных не производится.

1.2.2.11. Задание номера кадра синхронизации *SYNC_FRAME*

Запрос SYNC_FRAME используется для задания номера кадра синхронизации. Поля запроса содержат следующие значения:

- bmRequestType — 10000010b;
- bRequest — 0x0C;
- wValue — 0;
- wIndex — номер конечной точки;
- wLength — 2.

С помощью запроса SYNC_FRAME хост передает заданной конечной точке, работающей в изохронном режиме, шестнадцатиразрядное слово данных, содержащее номер кадра, который конечная точка должна использовать для синхронизации передачи.

1.2.2.12. Обработка стандартных запросов

Листинг 1.5 содержит константы стандартных запросов. Каждый идентификатор является объединением полей bmRequestType и bRequest. Эти константы позволяют легко распознавать запросы.

Листинг 1.5. Константы описания стандартных запросов и их применение

```
#define GET_STATUS_DEVICE      0x8000
#define GET_STATUS_INTERF      0x8100
#define GET_STATUS_ENDPNT      0x8200
#define CLEAR_FEATURE_DEVICE   0x0001
#define CLEAR_FEATURE_INTERF   0x0101
#define CLEAR_FEATURE_ENDPNT   0x0201
#define SET_FEATURE_DEVICE     0x0003
#define SET_FEATURE_INTERF     0x0103
#define SET_FEATURE_ENDPNT     0x0203
#define SET_ADDRESS            0x0005
#define GET_DESCRIPTOR_DEVICE  0x8006
#define GET_DESCRIPTOR_INTERF  0x8106
#define GET_DESCRIPTOR_ENDPNT  0x8206
#define SET_DESCRIPTOR         0x0007
#define GET_CONFIGURATION      0x8008
#define SET_CONFIGURATION      0x0009
```

```
#define GET_INTERFACE          0x810A
#define SET_INTERFACE          0x010B
#define SYNCH_FRAME            0x820C

  switch (wRequest)
  {
    case GET_STATUS_DEVICE:
     break;
    case GET_STATUS_INTERF:
     break;
     ... ... ... ... ... ... ... ...
    default:
      break;
  }
```

1.2.3. Дескриптор устройства

Дескриптор устройства (device descriptor) — это структура данных, или форматированный блок информации, который позволяет хосту получить описание USB-устройства. Каждый дескриптор содержит информацию либо об USB-устройстве в целом, либо о его части.

Все USB-устройства должны передавать хабу свои дескрипторы в ответ на стандартный запрос. Это означает, что любое периферийное USB-устройство должно делать две вещи: во-первых, хранить информацию о своих дескрипторах, и, во-вторых, пересылать эту информацию в ответ на запрос хаба в определенном формате.

Типы дескрипторов были описаны в предыдущем разделе (см. листинг 1.3). Спецификация USB определяет специальную группу дескрипторов, которая должна выдаваться USB-устройством в ответ на стандартные запросы. Такие дескрипторы называются *стандартными дескрипторами* (standard descriptors).

1.2.3.1. Дескриптор устройства

Стандартный дескриптор устройства (standard device descriptor) содержит основную информацию об USB-устройстве в целом и обо всех существующих конфигурациях. USB-устройство может иметь только один такой дескриптор. HS-устройство, имеющее различную информацию для HS- и FS-режимов, должно иметь также *уточняющий дескриптор устройства* (*см. ниже*). Структура стандартного дескриптора устройства показана в табл. 1.4, а соответствующее описание на языках C и Pascal показано в листинге 1.6.

Таблица 1.4. *Структура стандартного дескриптора устройства*

Смещение	Поле	Размер	Описание
0	`bLength`	1	Размер дескриптора в байтах
1	`bDescriptorType`	1	Тип дескриптора (`USB_DEVICE_DESCRIPTOR_TYPE`)
2	`bcdUSB`	2	Номер версии спецификации USB в формате BCD
4	`bDeviceClass`	1	Код класса USB
5	`bDeviceSubClass`	1	Код подкласса USB-устройства
6	`bDeviceProtocol`	1	Код протокола USB
7	`bMaxPacketSize0`	1	Максимальный размер пакета для нулевой конечной точки
8	`idVendor`	2	Идентификатор изготовителя
10	`idProduct`	2	Идентификатор продукта
12	`bcdDevice`	2	Номер версии устройства в формате BCD
14	`iManufacturer`	1	Индекс дескриптора строки, описывающей изготовителя
15	`iProduct`	1	Индекс дескриптора строки, описывающей продукт
16	`iSerialNumber`	1	Индекс дескриптора строки, содержащей серийный номер USB-устройства
17	`bNumConfigurations`	1	Количество возможных конфигураций USB-устройства

Листинг 1.6. Стандартный дескриптор устройства

```
// описание на языке C
typedef struct _USB_DEVICE_DESCRIPTOR {
 UCHAR  bLength;
 UCHAR  bDescriptorType;
 USHORT bcdUSB;
 UCHAR  bDeviceClass;
 UCHAR  bDeviceSubClass;
 UCHAR  bDeviceProtocol;
 UCHAR  bMaxPacketSize0;
```

```
 USHORT idVendor;
 USHORT idProduct;
 USHORT bcdDevice;
 UCHAR  iManufacturer;
 UCHAR  iProduct;
 UCHAR  iSerialNumber;
 UCHAR  bNumConfigurations;
} USB_DEVICE_DESCRIPTOR;
// описание на языке Pascal
TUsbDeviceDescriptor = packed record
  bLength            : BYTE;
  bDescriptorType    : BYTE;
  bcdUSB             : WORD;
  bDeviceClass       : BYTE;
  bDeviceSubClass    : BYTE;
  bDeviceProtocol    : BYTE;
  bMaxPacketSize0    : BYTE;
  idVendor           : WORD;
  idProduct          : WORD;
  bcdDevice          : WORD;
  iManufacturer      : BYTE;
  iProduct           : BYTE;
  iSerialNumber      : BYTE;
  bNumConfigurations : BYTE;
 End;
```

Поля стандартного дескриптора конфигурации подчиняются следующим правилам:

- размер дескриптора (поле `bLength`) всегда составляет 18 байтов;
- код типа дескриптора (поле `bDescriptorType`) имеет значение `1`;
- номер версии (поле `bcdUSB`) представляется в формате BCD и может принимать следующие значения:
  - `0100H` — версия 1.0;
  - `0110H` — версия 1.1;
  - `0200H` — версия 2.0;
  - HS-устройства должны возвращать значение версии 2.0;

- поле кода класса (поле `bDeviceClass`) может принимать следующие значения:
  - значение `00H` обозначает, что интерфейсы функционируют независимо друг от друга, и каждый из них имеет собственный код класса;
  - значение между `1` и `FEH` обозначает, что устройство поддерживает различные спецификации для интерфейсов, и интерфейсы не могут функционировать независимо;
  - значение `FFH` обозначает, что класс устройства определяется изготовителем;
- код подкласса (поле `bDeviceSubClass`) имеет значение `0`;
- код протокола (поле `bDeviceProtocol`) имеет значение `0`;
- максимальный размер пакета для нулевой конечной точки (поле `bMaxPacketSize0`) составляет 64 байта для HS и 8 байт для других режимов (хотя в общем случае могут использоваться значения 8, 16, 32 и 64);
- число возможных конфигураций (поле `bNumConfiguration`) описывает число конфигураций только для текущей скорости работы, но не для обеих скоростей.

Идентификатор изготовителя устройства, идентификатор продукта и номер версии используются для подбора драйвера (*см. разд. 13.1*).

Индексы дескрипторов строк используются для получения информации об устройстве в текстовом формате: при передаче запроса на получение дескриптора строки индекс дескриптора передается в младшем байте параметра `wValue`.

1.2.3.2. Уточняющий дескриптор устройства

Уточняющий дескриптор устройства (device qualifier descriptor) содержит дополнительную информацию о HS-устройстве при его работе на другой скорости. Например, если устройство работает в FS-режиме, то уточняющий дескриптор вернет информацию об HS-режиме работы и наоборот. Структура уточняющего дескриптора показана в табл. 1.5.

Таблица 1.5. *Структура уточняющего дескриптора устройства*

Смещение	Поле	Размер	Описание
0	`bLength`	1	Размер дескриптора в байтах
1	`bDescriptorType`	1	Тип дескриптора (`DEVICE_QUALIFIER`)

Таблица 1.5 *(окончание)*

Смещение	Поле	Размер	Описание
2	`bcdUSB`	2	Номер версии спецификации USB в формате BCD (равно 0200H)
4	`bDeviceClass`	1	Код класса USB
5	`bDeviceSubClass`	1	Код подкласса USB-устройства
6	`bDeviceProtocol`	1	Код протокола USB
7	`bMaxPacketSize0`	1	Максимальный размер пакета для нулевой конечной точки
8	`bNumConfigurations`	1	Количество дополнительных конфигураций устройства
9	`bReserved`	1	Зарезервировано, должно быть равно нулю

Поля `idVendor`, `idProduct`, `bcdDevice`, `iManufacturer`, `iProduct`, `iSerialNumber`, присутствующие в стандартном дескрипторе, в дополнительном дескрипторе отсутствуют, т. к. эта информация одинакова для всех скоростей работы.

1.2.3.3. Дескриптор конфигурации

Стандартный дескриптор конфигурации (standard configuration descriptor) содержит информацию об одной из возможных конфигураций USB-устройства. Структура дескриптора конфигурации показана в табл. 1.6, а соответствующее описание на языках C и Pascal показано в листинге 1.7.

Таблица 1.6. *Структура стандартного дескриптора конфигурации*

Смещение	Поле	Размер	Описание
0	`bLength`	1	Размер дескриптора в байтах
1	`bDescriptorType`	1	Тип дескриптора (`USB_CONFIGURATION_DESCRIPTOR_TYPE`)
2	`wTotalLength`	2	Общий объем данных (в байтах), возвращаемый для данной конфигурации

Таблица 1.6 (окончание)

Смещение	Поле	Размер	Описание
4	bNumInterfaces	1	Количество интерфейсов, поддерживаемых данной конфигурацией
5	bConfigurationValue	1	Идентификатор конфигурации, используемый при вызове SET_CONFIGURATION для установки данной конфигурации
6	iConfiguration	1	Индекс дескриптора строки, описывающей данную конфигурацию
7	bmAttributes	1	Характеристики конфигурации
8	MaxPower	1	Код мощности, потребляемой USB-устройством от шины

Листинг 1.7. Дескриптор конфигурации

```
// Описание на языке C
typedef struct _USB_CONFIGURATION_DESCRIPTOR {
 UCHAR  bLength;
 UCHAR  bDescriptorType;
 USHORT wTotalLength;
 UCHAR  bNumInterfaces;
 UCHAR  bConfigurationValue;
 UCHAR  iConfiguration;
 UCHAR  bmAttributes;
 UCHAR  MaxPower;
} USB_CONFIGURATION_DESCRIPTOR;
// Описание на языке Pascal
TUsbConfigurationDescriptor = packed record
 bLength             : BYTE;
 bDescriptorType     : BYTE;
 wTotalLength        : WORD;
 bNumInterfaces      : BYTE;
 bConfigurationValue : BYTE;
```

```
 iConfiguration      : BYTE;
 bmAttributes        : BYTE;
 MaxPower            : BYTE;
End;
```

Поле `bmAttributes` представляет собой битовую маску, имеющую следующие значения:

- [7] зарезервирован и должен равняться нулю;
- [6] признак наличия у USB-устройства собственного источника питания:
  - 0 — получает питание по шине;
  - 1 — имеет собственный источник питания;
- [5] признак возможности пробуждения USB-устройства по внешнему сигналу:
  - 0 — не имеет такой возможности;
  - 1 — имеет возможность пробуждения;
- [4:0] зарезервированы и должны содержать нули.

Значение поля `MaxPower` равно максимальному току в миллиамперах, потребляемому USB-устройством от шины, деленному на 2.

USB-устройство может иметь один или несколько дескрипторов конфигурации в соответствии с количеством возможных конфигураций, указанных в стандартном дескрипторе. Каждая конфигурация имеет один или несколько интерфейсов. Каждый интерфейс имеет ноль или несколько конечных точек.

Каждая конфигурация описывается одним стандартным дескриптором, размер которого составляет 9 байт. Поле `bConfigurationValue` является идентификатором конфигурации, описываемой данным дескриптором, и используется при установке конфигурации.

Каждая конфигурация может иметь один или несколько интерфейсов. Количество доступных интерфейсов указывается в поле `bNumInterfaces`. Например, ISDN-устройство может иметь конфигурацию с двумя интерфейсами, каждый из которых предоставляет канал по 64 Кбит/с, либо конфигурацию с одним интерфейсов но имеющую канал 128 Кбит/с.

USB-устройство может вернуть дескриптор конфигурации, с полем `bDescriptionType`, равным `OTHER_SPEED_CONFIGURATION`. Такой дескриптор описывает конфигурацию для HS-режима, если данное устройство поддерживает этот режим.

1.2.3.4. Дескриптор интерфейса

Стандартный дескриптор интерфейса (standard interface descriptor) содержит информацию об одном из интерфейсов, доступных при определенной конфигурации USB-устройства. Структура дескриптора интерфейса показана в табл. 1.7, а соответствующее описание на языках С и Pascal показано в листинге 1.8.

Таблица 1.7. *Структура стандартного дескриптора интерфейса*

Смещение	Поле	Размер	Описание
0	`bLength`	1	Размер дескриптора в байтах
1	`bDescriptorType`	1	Тип дескриптора (`USB_INTERFACE_DESCRIPTOR_TYPE`)
2	`bInterfaceNumber`	1	Номер данного интерфейса (нумеруются с 0) в наборе интерфейсов, поддерживаемых в данной конфигурации
3	`bAlternateSetting`	1	Альтернативный номер интерфейса
4	`bNumEndpoints`	1	Число конечных точек для этого интерфейса без учета нулевой конечной точки
5	`bInterfaceClass`	1	Код класса интерфейса
6	`bInterfaceSubClass`	1	Код подкласса интерфейса
7	`bInterfaceProtocol`	1	Код протокола
8	`iInterface`	1	Индекс дескриптора строки, описывающей интерфейс

Листинг 1.8. Дескриптор интерфейса

```
// описание на языке С
typedef struct _USB_INTERFACE_DESCRIPTOR {
 UCHAR bLength;
 UCHAR bDescriptorType;
 UCHAR bInterfaceNumber;
 UCHAR bAlternateSetting;
 UCHAR bNumEndpoints;
 UCHAR bInterfaceClass;
 UCHAR bInterfaceSubClass;
```

```
 UCHAR bInterfaceProtocol;
 UCHAR iInterface;
} USB_INTERFACE_DESCRIPTOR;
// описание на языке Pascal
TUsbInterfaceDescriptor = packed record
  bLength            : BYTE;
  bDescriptorType    : BYTE;
  bInterfaceNumber   : BYTE;
  bAlternateSetting  : BYTE;
  bNumEndpoints      : BYTE;
  bInterfaceClass    : BYTE;
  bInterfaceSubClass : BYTE;
  bInterfaceProtocol : BYTE;
  iInterface         : BYTE;
 End;
```

Размер дескриптора интерфейса всегда составляет 9 байт. При идентификации и нумерации USB-устройств на шине дескриптор интерфейса может использоваться для определения типа устройства по кодам класса, подкласса и протокола.

Дескриптор интерфейса возвращается USB-устройством при выполнении запроса `GET_DESCRIPTOR` и не может быть запрошен или установлен напрямую вызовами `GET_DESCRIPTOR` или `SET_DESCRIPTOR`.

USB-устройство может иметь *альтернативный набор установок* (alternate settings), что позволяет изменять настройки после конфигурирования. По умолчанию всегда устанавливаются обычные настройки интерфейса, а с помощью запроса `SET_INTERFACE` могут быть установлены альтернативные настройки или возвращены настройки по умолчанию.

Если интерфейс использует только нулевую конечную точку, то поле `bNumEndpoints` должно быть равно нулю.

1.2.3.5. Дескриптор конечной точки

Стандартный дескриптор конечной точки (standard endpoint descriptor) содержит информацию об одной из конечных точек, доступных при использовании определенного интерфейса. Структура дескриптора конечной точки показана в табл. 1.8, а соответствующее описание на языках C и Pascal показано в листинге 1.9.

Таблица 1.8. *Структура стандартного дескриптора конечной точки*

Смещение	Поле	Размер	Описание
0	`bLength`	1	Размер дескриптора в байтах
1	`bDescriptorType`	1	Тип дескриптора (`USB_ENDPOINT_DESCRIPTOR_TYPE`)
2	`bEndpointAddress`	1	Код адреса конечной точки
3	`bmAttributes`	1	Атрибуты конечной точки
4	`wMaxPacketSize`	2	Максимальный размер пакета для конечной точки
6	`bInterval`	1	Интервал опроса конечной точки при передаче данных (задается в миллисекундах)

Листинг 1.9. Дескриптор конечной точки

```
// описание на языке C
typedef struct _USB_ENDPOINT_DESCRIPTOR {
 UCHAR  bLength;
 UCHAR  bDescriptorType;
 UCHAR  bEndpointAddress;
 UCHAR  bmAttributes;
 USHORT wMaxPacketSize;
 UCHAR  bInterval;
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR;
// описание на языке Pascal
TUsbEndpointDescriptor = packed record
 bLength          : BYTE;
 bDescriptorType  : BYTE;
 bEndpointAddress : BYTE;
 bmAttributes     : BYTE;
 wMaxPacketSize   : WORD;
 bInterval        : BYTE;
End;
```

Размер дескриптора конечной точки составляет 7 байт.

Код адреса `bEndpointAddress` и байт атрибутов `bmAttributes` для многих классов периферийных USB-устройств позволяет однозначно определить функциональное назначение конечной точки. Код адреса `bEndpointAddress` содержит следующие биты:

- [7] направление передачи (игнорируется для каналов сообщений):
  - 0 — OUT (от хоста);
  - 1 — IN (к хосту);
- [6:4] зарезервированы и должны содержать нули;
- [3:0] номер конечной точки.

Байт атрибутов `bmAttributes` содержит следующие биты:

- [7:6] зарезервированы и должны быть равны нулю;
- [5:4] тип использования конечной точки:
  - 00 — конечная точка (данные);
  - 01 — конечная точка для явной обратной связи;
  - 10 — конечная точка неявной обратной связи;
  - 11 — зарезервировано;
- [3:2] тип синхронизации (для изохронных каналов, *см. разд. 1.1.13*):
  - 00 — нет синхронизации;
  - 01 — асинхронная;
  - 10 — адаптивная;
  - 11 — синхронная;
- [1:0] тип конечной точки:
  - 00 — канал сообщений;
  - 01 — изохронный канал;
  - 10 — канал передачи данных;
  - 11 — канал прерываний.

Интервал опроса конечной точки (поле `bInterval`) имеет значение только в том случае, если точка используется для передачи данных по прерываниям. Для изохронных конечных точек это поле всегда равно 1. Для остальных типов конечных точек значение этого поля игнорируется.

В HS-режиме для изохронных передач и прерываний биты [12:11] поля `wMaxPacketSize` определяют число транзакций внутри фрейма, определяемого значением поля `bInterval` (табл. 1.9).

Таблица 1.9. Соответствие числа транзакций и размера пакетов

Биты [12:11] поля wMaxPacketSize	Максимально допустимый размер пакета (биты [10:0] поля wMaxPacketSize)
00	1–1024
01	513–1024
10	683–1024
11	Не используется, зарезервировано

1.2.3.6. Дескриптор строки

Дескриптор строки (Unicode string descriptor) содержит текст в формате Unicode. Строка не ограничивается нулем, а длина строки вычисляется вычитанием 2 из размера дескриптора. Структура дескриптора строки показана в табл. 1.10, а соответствующее описание на языках С и Pascal показано в листинге 1.10.

Таблица 1.10. Структура дескриптора строки

Смещение	Поле	Размер	Описание
0	bLength	1	Размер дескриптора в байтах (N+2)
1	bDescriptorType	1	Тип дескриптора (USB_STRING_DESCRIPTOR_TYPE)
2	bString	N	Строка символов Unicode

Листинг 1.10. Дескриптор строки

```
// описание на языке С
typedef struct _USB_STRING_DESCRIPTOR {
 UCHAR bLength;
 UCHAR bDescriptorType;
 WCHAR bString[1];
} USB_STRING_DESCRIPTOR;
// описание на языке Pascal
TUsbStringDescriptor = packed record
```

```
 bLength         : BYTE;
 bDescriptorType : BYTE;
 bString         : Array of Byte;
End;
```

Дескриптор строки является не обязательным. Если USB-устройство не поддерживает дескрипторы строк, все ссылки на такие дескрипторы из дескрипторов устройства, конфигурации или интерфейса должны иметь нулевое значение.

USB-устройство может поддерживать несколько различных языков, поэтому при запросе дескриптора строки нужно задавать идентификатор языка (`LANGID`). Строковый индекс 0 соответствует дескриптору строки, содержащей массив шестнадцатиразрядных идентификаторов всех поддерживаемых языков. Массив идентификаторов не ограничен нулем, а размер массива в байтах вычисляется вычитанием 2 из размера дескриптора (табл. 1.11).

Таблица 1.11. *Структура дескриптора идентификатора языков*

Смещение	Поле	Размер	Описание
0	`bLength`	1	Размер дескриптора в байтах (N+2)
1	`bDescriptorType`	1	Тип дескриптора (`USB_STRING_DESCRIPTOR_TYPE`)
2	`wLANGID[0]`	2	Идентификатор языка
...	... ...	... ...	... ...
N	`wLANGID[x]`	2	Идентификатор языка

1.2.3.7. Специфические дескрипторы

Специфические дескрипторы (class-specific descriptor) могут использоваться в устройствах определенных классов. Например, мы будем описывать дескрипторы, специфические для коммуникационных USB-устройств (*см. разд. 5.3.2*), и дескрипторы, специфические для HID-устройств[2] (*см. разд. 6.4.3*).

[2] HID-устройство (Human Interface Device) — это устройство связи с пользователем.

1.2.3.8 Порядок получения дескрипторов

В процессе нумерации (*см. разд. 1.3.2*) хост с помощью управляющих посылок (*см. разд. 1.1.10*) запрашивает дескрипторы от USB-устройства в такой последовательности:

- дескриптор устройства;
- дескрипторы конфигураций;
- дескрипторы интерфейсов для конфигурации;
- дескрипторы интерфейсов всех конечных точек.

Реально, при запросе дескриптора конфигурации, USB-устройство сразу возвращает последовательность "вложенных" дескрипторов:

- дескриптор конфигурации-1;
  - дескриптор интерфейса-1;
    - дескриптор конечной точки-1 для интерфейса-1;
    - дескриптор конечной точки-2 для интерфейса-1;
    - ....
  - дескриптор интерфейса-2;
    - дескриптор конечной точки-1 для интерфейса-2;
  - ....
- дескриптор конфигурации-2.

Общая длина возвращаемых данных заранее не известна. Для определения общей длины возвращаемого списка дескрипторов служит поле `wTotalLength`. Чтобы получить весь список дескрипторов, нужно запросить первые 8 байт дескриптора конфигурации, запомнить значение поля `wTotalLength`, а затем использовать это значение в качестве параметра при повторной подаче запроса. Важно понимать, что такой алгоритм действий выполняется внутри драйвера, а со стороны пользовательской программы получение списка дескрипторов выглядит несколько проще (листинг 1.11).

Листинг 1.11. Получение списка дескрипторов конфигурации

```
// конфигурационный пакет запроса
TSetupPacket = packed record
 bmRequest : UCHAR;
 bRequest  : UCHAR;
 wValue    : Array [1..2] of UCHAR;
 wIndex    : Array [1..2] of UCHAR;
```

```
 wLength   : Array [1..2] of UCHAR;
End;

// структура запроса дескриптора
TDescriptorRequest = packed record
 ConnectionIndex : ULONG;
 SetupPacket     : TSetupPacket;
 Data            : Array [1..2048] of Byte;
End;

procedure TForm1.ShowDeviceDetail(hRoot : THandle; iPort : Integer);
// переменные для выполнения DeviceIoControl
Var Success : LongBool;
    Packet  : TDescriptorRequest;
    BytesReturned : Cardinal;

begin
// Получение стандартного дескриптора устройства
 ZeroMemory(@Packet, SizeOf(Packet));
 Packet.ConnectionIndex       := iPort+1;
 Packet.SetupPacket.bmRequest := $80;
 Packet.SetupPacket.bRequest  := USB_REQUEST_GET_DESCRIPTOR;
 Packet.SetupPacket.wValue [2]:= USB_DEVICE_DESCRIPTOR_TYPE;
 Packet.SetupPacket.wLength[2]:= 1; // Использовать буфер 2 Кбайта

 // IOCTL_USB_GET_DESCRIPTOR_FROM_NODE_CONNECTION
 Success:= DeviceIoControl(hRoot, GetUSBCtlCode(5),
            @Packet, sizeof(Packet),
            @Packet, sizeof(Packet),
            BytesReturned, nil
           );

 If not(Success) then begin
  Log(Format('    Ошибка получения информации об устройстве %s',
  [ SysErrorMessage(GetLastError())]));
  Exit;
 End;
```

```
 // отображение дескриптора устройства
 DisplayDescriptorInfo(Packet.Data);

// Получение дескрипторов конфигурации
 ZeroMemory(@Packet, SizeOf(Packet));
 Packet.ConnectionIndex       := iPort+1;
 Packet.SetupPacket.bmRequest := $80;
 Packet.SetupPacket.bRequest  := USB_REQUEST_GET_DESCRIPTOR;
 Packet.SetupPacket.wValue [2]:= USB_CONFIGURATION_DESCRIPTOR_TYPE;
 Packet.SetupPacket.wLength[2]:= 1; // Использовать буфер 2 Кбайта

 // IOCTL_USB_GET_DESCRIPTOR_FROM_NODE_CONNECTION
 Success:= DeviceIoControl(hRoot, GetUSBCtlCode(5),
            @Packet, sizeof(Packet),
            @Packet, sizeof(Packet),
            BytesReturned, nil
           );

 If not(Success) then begin
  Log(Format('    Ошибка получения информации об устройстве %s',
                [ SysErrorMessage(GetLastError())]));
  Exit;
 End;

 // отображение информации о дескрипторах конфигурации
 DisplayDescriptorInfo(Packet.Data);
End;
```

Описание функции `DeviceIoControl` можно найти в справочной части книги (*см. разд. 14.9*). В нашем примере ключевыми параметрами этой функции являются:

- `hRoot` — дескриптор порта, к которому подключено устройство;
- `GetUsbCtlCode(5)` — код IOCTL-функции (*см. разд. 12.1.5*);
- `Packet` — структура типа `TDescriptorRequest`.

Подробности обращения к драйверу USB-устройства и внутреннюю организацию самих драйверов мы будем обсуждать позднее (*см. гл. 12*).

Как видно из листинга, в результате выполнения запроса возвращается буфер данных `Packet.Data`. Для разбора и отображения этого буфера можно использовать процедуру `DisplayDescriptorInfo`, код которой показан в листинге 1.12.

Листинг 1.12. Разбор списка дескрипторов

```
procedure TForm1.DisplayDescriptorInfo(Data : Array of byte);
var iData : Integer; lenDescr, typDescr : Byte; PData : Pointer;
begin
 // Проходим по массиву дескрипторов
 // Первый байт - размер дескриптора в байтах
 // Второй байт - тип дескриптора
 iData:= 0;

 Repeat
  lenDescr:= Data[iData+0];
  typDescr:= Data[iData+1];
  PData   := @Data[iData+0];

  Case typDescr of
   $01: begin  // Стандартный дескриптор
     With TDeviceDescriptor(PData^) do begin
      Log(Format('      bcdUSB            =%s', [BCD2Str(bcdUSB)   ]));
      Log(Format('      bDeviceClass      =%d', [bDeviceClass      ]));
      Log(Format('      bDeviceSubClass   =%d', [bDeviceSubClass   ]));
      Log(Format('      bDeviceProtocol   =%d', [bDeviceProtocol   ]));
      Log(Format('      bMaxPacketSize0   =%d', [bMaxPacketSize0   ]));
      Log(Format('      idVendor          =%d', [idVendor          ]));
      Log(Format('      idProduct         =%d', [idProduct         ]));
      Log(Format('      bcdDevice         =%s', [BCD2Str(bcdDevice)]));
      Log(Format('      iManufacturer     =%d', [iManufacturer     ]));
      Log(Format('      iProduct          =%d', [iProduct          ]));
      Log(Format('      iSerialNumber     =%d', [iSerialNumber     ]));
      Log(Format('      bNumConfigurations=%d', [bNumConfigurations]));
     End;
    End;
```

```
   $02: begin  // Дескриптор конфигурации
     With TUsbConfigurationDescriptor(PData^) do begin
      Log(Format('      iConfiguration=%d', [iConfiguration]));
      Log(Format('      bNumInterfaces=%d', [bNumInterfaces]));
      Log(Format('      bmAttributes  =%d', [bmAttributes  ]));
      Log(Format('      MaxPower(мА)  =%d', [MaxPower*2    ]));
     End;
    End;

   $03: begin
             // Дескриптор строки
    End;

    $04: begin
            // Дескриптор интерфейса
    End;

    $05: begin
           // Дескриптор конечной точки
    End;

   End; {end of case}
 // переходим к следующему дескриптору
   iData:= iData + lenDescr;
  Until (lenDescr = 0) or (iData > High(Data));
end;
```

Каждый дескриптор отображается в соответствии с содержащимися в нем данными. Для получения доступа к содержимому дескриптора мы используем указатель на текущее положение в буфере данных, приводя его к соответствующему типу. Для перевода BCD-чисел в строку номера версии используется функция `BCD2Str`, код которой можно найти в *приложении 1*. Полный код этой программы содержится на компакт-диске.

1.3. Система Plug and Play (PnP)

Протокол Plug and Play (дословно, "подключил и играй") позволяет достаточно просто подключать новое оборудование. Перед началом работы сис-

тема (BIOS[3] при начальной загрузке, Windows при запуске) опрашивает устройства, узнает их требования к системным ресурсам и пытается бесконфликтно разделить ресурсы между устройствами. Согласно спецификации USB, любое USB-устройство должно соответствовать спецификации PnP. Шина USB поддерживает динамическое подключение и отключение устройств "по определению". Нумерация устройств шины является постоянным процессом, отслеживающим изменения физической топологии.

1.3.1. Конфигурирование USB-устройств

При начальном подключении или после сброса производится начальное конфигурирование. Хабы определяют подключение и отключение USB-устройств к своим портам и сообщают состояние портов при запросе от хоста. Хост разрешает работу порта и адресуется к USB-устройству через канал управления, используя нулевой порт.

Хост определяет, является ли новое подключенное USB-устройство хабом или функцией и назначает ему *уникальный адрес* USB. Хост создает канал управления, используя назначенный адрес и нулевой номер конечной точки. При подключении хаба хост определяет подключенные к нему USB-устройства, назначает им адреса и устанавливает каналы. Если подключается функция, то уведомление о подключении передается диспетчером заинтересованному ПО.

Когда USB-устройство отключается, хаб автоматически запрещает соответствующий порт и сообщает об отключении контроллеру, который удаляет сведения о данном устройстве из всех структур данных. Если отключается функция, уведомление посылается заинтересованному ПО. Если отключается хаб, процесс удаления выполняется для всех подключенных к нему USB-устройств.

1.3.2. Нумерация USB-устройств

Нумерация устройств (enumeration), подключенных к шине, осуществляется динамически по мере их подключения (или включения их питания) без какого-либо вмешательства пользователя или клиентского ПО. Процедура нумерации выполняется следующим образом.

1. *Включение USB-устройства.* Пользователь подключает USB-устройство к порту хаба (корневого или любого другого) или подает питание на уже подключенное к порту USB-устройство. USB-устройство переходит в состояние *питание подано* (powered).

[3] BIOS (сокр. от *Basic Input/Output System*) — базовая система ввода/вывода.

2. *Хаб определяет подключение USB-устройства.* Хаб, производящий постоянный мониторинг каждого порта, определяет, что к порту подключено USB-устройство. Определив подключение, хаб продолжает подавать питание, но данные пока не передает, т. к. USB-устройство еще не готово их принимать.

3. *Хаб информирует хост о новом USB-устройстве.* Хаб, к которому подключено USB-устройство, информирует хост о смене состояния своего порта ответом на опрос состояния. Каждый хаб имеет специальное прерывание (точнее, конечно, канал типа Interrupt) для передачи таких уведомлений. Когда хост узнает о подключении нового USB-устройства, он посылает запрос `GET_STATUS` для получения дополнительной информации.

4. *Хаб проверяет режим USB-устройства.* Хаб проверяет, является ли USB-устройство низкоскоростным или полноскоростным, и отправляет эту информацию в ответ на запрос `GET_STATUS`. Спецификация USB 1.*x* позволяет хабу производить определение скоростного режима и после сброса, но USB 2.0 требует знания режима до сброса.

5. *Хаб подает USB-устройству сигнал сброса.* Когда хост узнает о появлении нового USB-устройства, то посылает хабу запрос `SET_FEATURE`, который говорит хабу произвести сброс порта, к которому подключено USB-устройство, другие хабы и порты шины не затрагиваются.

6. *Хост определяет возможность работы USB-устройства в режиме HS.*

7. *Хаб устанавливает соединение между USB-устройством и шиной.* Хост проверяет, что сброс USB-устройства произведен. Для этого хост посылает USB-устройству запрос `GET_STATUS`, если ответа нет, то запрос повторяется. Состояние USB-устройства после сброса называется *основным состоянием* (default state). В этом состоянии регистры USB-устройства сброшены, а само оно готово к обмену по нулевому каналу.

8. *Хост определяет конфигурацию нулевой точки.* Хост посылает запрос `GET_DESCRIPTOR` для того, чтобы узнать размер максимального пакета для основного канала. Хост посылает запрос по адресу 0, конечной точке номер 0. Так как в один момент времени хост будет работать только с одним (обнаруженным) USB-устройством, то на этот запрос откликнется только оно, даже если к шине подключено несколько USB-устройств. В ответ передается 8-байтовый дескриптор, в котором содержится максимальный размер пакета, поддерживаемый конечной точкой 0.

9. *Хост назначает USB-устройству уникальный адрес*, посылая запрос `SET_ADDRESS`. USB-устройство посылает хосту подтверждение и переходит в состояние *адресовано* (adressed). С этого момента любой обмен данными возможен только по этому адресу. Адрес остается верен до отключения USB-устройства, во время следующего включения адрес может быть другим.

10. *Хост считывает конфигурацию USB-устройства*, включая заявленный потребляемый ток от шины. Хост посылает запрос GET_DESCRIPTOR по новому адресу. Дескриптор, посылаемый USB-устройством, содержит максимальный размер пакета для нулевой конечной точки, число поддерживаемых конфигураций и другую информацию.
11. *Хост ищет и загружает драйвер USB-устройства.* После того как хост узнал всю информацию, он ищет наиболее подходящий драйвер и загружает его. При поиске драйвера Windows проверяет поля Vendor, Product ID и Release Number в INF-файлах. Если такой INF-файл не найден, Windows пытается найти драйвер согласно классу, подклассу и типу протокола, полученным от устройства.
12. *Драйвер выбирает конфигурацию.* Драйвер посылает запрос SET_CONFIGURATION. Многие USB-устройства имеют только одну возможную конфигурацию, но если поддерживается несколько, драйвер выберет либо первую, либо базовую, либо попросит пользователя выбрать нужную. Исходя из полученной информации, хост конфигурирует все имеющиеся конечные точки данного USB-устройства, которое переводится в состояние *сконфигурировано* (configured). Теперь хаб позволяет USB-устройству потреблять от шины полный ток, заявленный в конфигурации. Устройство готово.

Когда USB-устройство отключается от шины, хаб уведомляет об этом хост и работа порта запрещается, а хост обновляет свою текущую топологическую информацию.

1.3.3. PnP-идентификаторы USB-устройств

Каждое USB-устройство, спроектированное по спецификации PnP, должно иметь идентификатор, который однозначно определяет модель данного устройства. Этот идентификатор должен быть предоставлен шинному аппаратному обеспечению (и, соответственно, шинному драйверу) при поступлении запроса. Секция описания модели (*см. разд. 13.1.4*) содержит поле hw_id, играющее роль идентификатора модели.

Идентификатор устройства должен иметь строго определенный для данного класса устройств формат. Для USB-устройств идентификатор имеет следующий формат:

```
USB\Vid_vvvv&Pid_dddd&Rev_rr
```

Здесь vvvv — идентификатор поставщика (поле idVendor дескриптора устройства, *см. разд. 1.2.3*), зарегистрированный в Комитете USB-производителей; dddd — идентификатор, присвоенный производителем данный модели USB-устройства (поле idProduct); rr — номер версии разработки. Все эти поля вводятся как шестнадцатеричные числа.

В INF-файле допустимо указывать усеченные варианты идентификаторов, например:

```
USB\Vid_vvvv&Pid_dddd
USB\Class_cc&SubClass_ss_Prot_pp
USB\Class_cc&SubClass_ss
USB\Class_cc
```

Здесь `cc` — код базового класса из полученного дескриптора устройства или дескриптора интерфейса данного USB-устройства; `ss` — код подкласса; `pp` — идентификатор протокола.

Примеры USB-идентификаторов:

- `USB\VID_040A&PID_0100` — цифровая USB-камера Kodak;
- `USB\ROOT_HUB20` — USB-хаб;
- `USB\VID_067B&PID_2303` — USB-мобильный телефон.

1.3.4. Символьные имена устройств

Создавая объект "устройство", драйвер может присвоить ему имя. Тогда он помещается в пространство имен *диспетчера объектов*. Драйвер может позволить определить имя устройства явно или позволить сгенерировать его автоматически. По соглашению имена объектов помещаются в каталог пространства имен \Device, недоступный приложениям через Windows API.

Чтобы сделать объект устройства доступным для приложений, драйвер должен создать в каталоге \?? (до Windows NT4 этот каталог назывался \DosDevice, а в Windows XP называется \GLOBAL??) символьную ссылку на имя этого объекта в каталоге \Device (рис. 1.19). Эта ссылка называется *символьным именем* устройства или *DOS-именем*. Унаследованные драйверы и драйверы логических устройств обычно создают ссылку с общеизвестными именами (например, F: для устройства чтения компакт-диска \Device\CDRom0 или COM1 для последовательного порта \Device\Seral0). Устройства, создаваемые динамически при работе системы PnP, создают имена, используя GUID (Globally Unique IDentifier, глобально уникальный идентификатор), гарантируя глобальную уникальность имени.

Несколько символьных имен может указывать на одно и то же устройство, однако обратное 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 символьные имена для внутренних имен устройств (*см. разд. 11.6*).

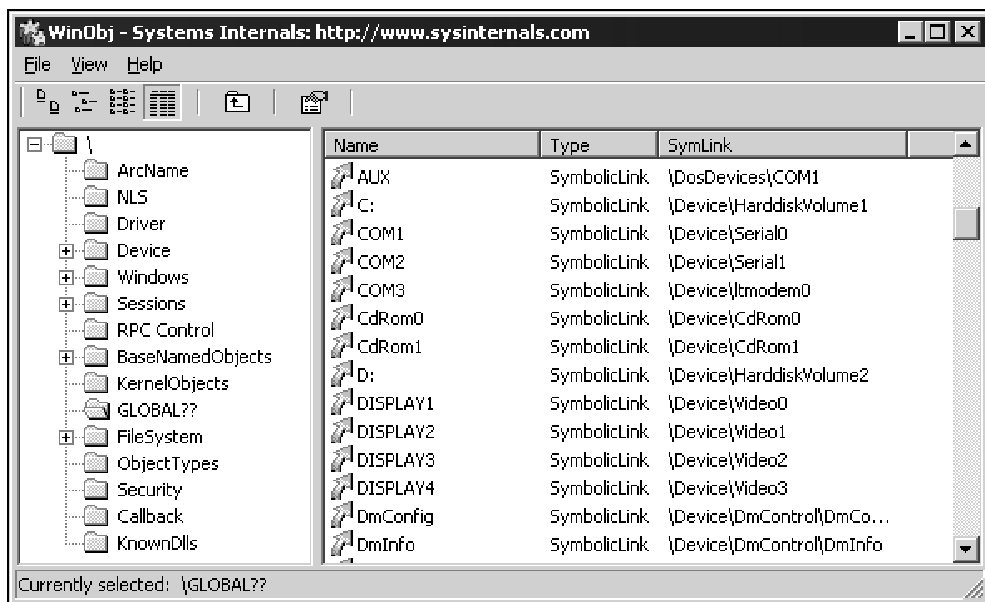


Рис. 1.19. Символьные имена Windows XP

1.4. Модель WDM

WDM (Windows Driver Model, драйверная модель Windows) — новая модель архитектуры драйверов, предложенная Microsoft для Windows 2000. Конечно, эта архитектура развивалась начиная с Windows 3.11, продолжала развиваться в Windows 98 и Windows NT, но по-настоящему полной она стала только в Windows 2000.

Архитектура WDM позволяет распределить процесс передачи данных (рис. 1.20).

С точки зрения WDM, существуют три типа драйверов.

- *Драйвер шины* (bus driver), обслуживающий контроллер шины, адаптер, мост или любые другие устройства, имеющие дочерние устройства. Драйверы шин нужны для работы системы и в общем случае поставляются Microsoft. Для каждого типа шины (PCI, PCMCIA и USB) в системе имеется свой драйвер. Сторонние разработчики создают драйверы для поддержки новых шин, например, для VMEbus, Multibus или Futurebus.
- *Функциональный драйвер* (function driver) — основной драйвер устройства, предоставляющий его функциональный интерфейс. Обязателен кроме тех случаев, когда устройство используется без драйверов (т. е. ввод/вывод осуществляется драйвером шины или драйвером фильтров шины). Функ-

циональный драйвер по определению обладает наиболее полной информацией о своем устройстве. Обычно только этот драйвер имеет доступ к специфическим регистрам устройства.

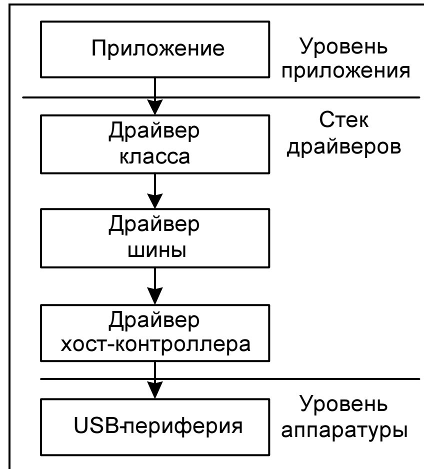


Рис. 1.20. WDM-модель для USB-интерфейса

- *Драйвер фильтра* (filter driver), поддерживающий дополнительную функциональность устройства (или существующего драйвера) или изменяющий запросы на ввод/вывод и ответы на них от других драйверов (это часто используется для коррекции устройств, предоставляющих неверную информацию о своих требованиях к аппаратным ресурсам). Такие драйверы не обязательны и их может быть несколько. Они могут работать как на более высоком уровне, чем функциональный драйвер или драйвер шины, так и на более низком. Обычно эти драйверы предоставляются производителями или независимыми поставщиками оборудования.

Согласно типам драйверов, существуют три типа объектов:

- объекты физических устройств (PDO, Physical Device Object);
- объекты функциональных устройств (FDO, Functional Device Object);
- объекты фильтров устройств (FiDO, Filter Device Object).

Объекты PDO создаются для каждого физически идентифицируемого элемента аппаратуры, подключенного к шине данных. Объект FDO подразумевает единицу логической функциональности устройства. Объекты фильтров предоставляют дополнительную функциональность.

В среде WDM один драйвер не может контролировать все аспекты устройства: драйвер шины информирует диспетчер PnP об устройствах, подключен-

ных к шине, в то время как функциональный драйвер управляет устройством. Драйверы фильтров низкого уровня позволяют исправлять информацию о требованиях устройства к системным ресурсам, а драйверы фильтров высокого уровня добавляют устройству дополнительную функциональность (например, производят дополнительную защиту клавиатуры).

Примечание

Получить список загруженных в данный момент драйверов можно с помощью утилиты Drivers (файл drivers.exe), как показано на рис. 1.21.

```
C:>Drivers
  ModuleName    Code    Data     Bss   Paged    Init         LinkDate
---------------------------------------------------------------------------
ntoskrnl.exe  431488   75904       0 1170944  171904  Thu Aug 29 13:03:24 2002
     hal.dll   32896   42624       0   28672   14336  Thu Aug 29 12:05:02 2002
   KDCOM.DLL    2560     256       0    1280     512  Sat Aug 18 00:49:10 2001
 BOOTVID.dll    5632    3584       0       0     512  Sat Aug 18 00:49:09 2001
    ACPI.sys  103936   11008       0   40192    4736  Thu Aug 29 12:09:03 2002
  WMILIB.SYS     512       0       0    1280     256  Sat Aug 18 01:07:23 2001
        ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...
    Cdfs.SYS    6528     640       0   42880    4480  Thu Aug 29 12:58:50 2002
asyncmac.sys    8576    1024       0       0    1152  Sat Aug 18 00:55:29 2001
 USBSTOR.SYS    6656     128       0   10368    1536  Thu Aug 29 12:32:50 2002
   ntdll.dll  466944   20480       0       0       0  Thu Aug 29 14:40:40 2002
---------------------------------------------------------------------------
       Total 5785536  818400       0 4474912  532192
```

Рис. 1.21. Получение списка загруженных драйверов

В поставку Windows входят два низкоуровневых драйвера, избавляя программиста от множества рутинной работы:

- драйвер хоста отвечает за обмен данными с аппаратурой;
- драйвер шины USB отвечает за управление транзакциями, питанием и распознавание устройств.

С точки зрения прикладного программиста, наибольший интерес представляют *драйвер класса* (class driver) и интерфейс обращения к этому драйверу. Здесь операционная система Windows делает еще один шаг на пути унификации интерфейсов. Все USB-устройства делятся на группы, согласно общим свойствам, выполняемым функциям и требованиям к ресурсам. Для

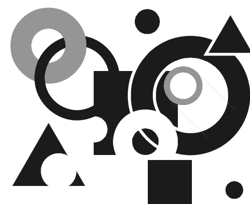
каждой группы устройств Windows предоставляет готовый драйвер, который автоматически устанавливается при обнаружении принадлежности устройства к одной из групп. Таким образом, в большинстве случаев никаких дополнительных драйверов не требуется. *Подробную информацию о классе CDC см. в гл. 5 и 9, о классе HID см. в гл. 6 и 10.*

Если же стандартный драйвер не подходит, необходимо писать собственный драйвер поддержки (*см. гл. 12*).

Список основных поддерживаемых на данный момент типов устройств приведен в табл. 1.12.

Таблица 1.12. *Список основных поддерживаемых типов устройств (Windows 2000/XP)*

Класс	**Драйвер**	**Примечания**
Хабы (hub device)	hubclass.sys	Хабы
HID-устройства (Human Interface Device)	hidclass.sys	Мышки, джойстики, клавиатуры (*см. гл. 10*)
Аудио (audio)	sysaudio.sys	Звуковые колонки, виртуальные MIDI-устройства
Устройства хранения данных (mass storage device)	usbstor.sys	Флэш-диски
Принтеры (printer)	usbprint.sys	Принтеры
Устройства коммуникации (communication)	mdismp.sys, usb8023.sys, другие	Модемы, сетевые карты, виртуальные последовательные порты (*см. гл. 5 и 9*)



Глава 2

Программирование на языке С для микроконтроллера

В этой главе мы приведем сведения, необходимые для разработки примеров для микроконтроллера на языке С. В частности, мы рассмотрим некоторые вопросы программирования для микроконтроллера AT89C5131, на базе которого будут строиться примеры нашей книги. У некоторых читателей предыдущей книги [4], до этого использующих только ассемблер, примеры на языке С вызвали затруднения. Надеемся эта глава (к сожалению, довольно краткая) поможет им разобраться в наших примерах более детально. Читатели, знакомые с языком С, могут пропустить эту главу.

2.1. Общие сведения о языке С для микроконтроллеров

Язык С является высокоуровневым языком программирования (в отличие от ассемблера). Программирование на языке высокого уровня имеет ряд значительных преимуществ:

- код программы получается более компактным и читабельным;
- код программы, созданный для одного микроконтроллера, достаточно просто переносится на другие;
- множество библиотек (включая математические библиотеки для работы с плавающей точкой) позволяют экономить время на разработку программы.

С другой стороны, при этом теряется возможность тонкой оптимизации, но для современных микроконтроллеров, обладающих значительными ресурсами памяти, это не является большой проблемой. Кроме того, для частей кода, требующих, например, минимального времени выполнения, можно использовать ассемблерные вставки.

Конечно, в нашей книге мы не сможем привести полный синтаксис языка С. Мы приведем лишь основные сведения и некоторые тонкости, связанные с особенностями микроконтроллерного программирования.

Обычно программа для микроконтроллера состоит из файла, содержащего код инициализации, одного или нескольких файлов самой программы, заголовочных файлов и одной или нескольких стандартных библиотек. Файл инициализации обычно пишется на языке ассемблера (например, CSTARTUP.S03 в наших примерах) и содержит описание стека, вектора прерываний и код инициализации микроконтроллера. Микроконтроллеры не содержат операционной системы, что накладывает соответствующие требования на код программы:

- точка входа в программу всегда располагается с нулевого адреса, поэтому первой командой в файле инициализации всегда является переход на собственно код программы (`LJMP init_C`);
- после инициализации производится переход на основную процедуру программы `main`;
- код процедуры `main` чаще всего содержит "вечный" цикл, т. к. программа в микроконтроллере не может остановиться, а процессор не может не выполнять никаких действий:

```
while (1) { код основного цикла программы }
```

Код программы и данные располагаются в разных областях памяти и, более того, требуют разных способов доступа к ним. Например:

- `code` — область данных, куда записывается код программы. Программа записывается через внешний или внутренний программатор. Например, программа для микроконтроллера AT89C5131 может записываться через USB-интерфейс;
- `data` — память обычных данных. В этой области располагаются переменные, для доступа к которым используются обычные регистры;
- `idata` — внешняя память данных. В этой области располагаются переменные, для доступа к которым используется косвенная адресация;
- `xdata` — флэш-память. Переменные из этой области памяти требуют специальных методов чтения и записи, т. к. располагаются в энергонезависимой флэш-памяти.

Многие компиляторы требуют явного описания типа переменной, поэтому синтаксис описания переменной немного отличается от стандартного, например:

```
data  byte  i;
idata float f;
code  byte  c;
```

Особенности микроконтроллеров могут порождать специальные типы данных. Например, битовые поля микроконтроллера AT89C5131 позволяют использовать битовые переменные:

```
bit b;
```

Более того, разрешается адресовать конкретные биты специальных регистров:

```
bit   RI   = 0x98;
bit   P3_0 = 0xB0;
```

Для описания регистров, расположенных в определенных ячейках памяти, используются специальные описатели, например:

```
sfr    P0 = 0x80;
sfr    P1 = 0x90;
sfr    P2 = 0xA0;
sfr    P3 = 0xB0;
sfr    P4 = 0xC0;
```

Для специальных регистров разрешается адресовать каждый бит регистра отдельно:

```
P3.3 = 1;
P1.0 = 0;
```

Для описания функций, которые будут вызываться по прерыванию, используется ключевое слово `interrupt` с указанием смещения точки входа в таблице векторов:

```
/* Вектор 0, External Interrupt 0 */
interrupt [0x03] void extern0_int (void);
```

2.2. Использование стандартных библиотек

Для использования дополнительных библиотек необходимо включить в код программы соответствующий заголовочный файл, а при компоновке программы подключить нужную библиотеку. Для краткости часто в качестве названия библиотеки используют имя заголовочного файла.

Так, например, для использования математических функций обычно используется библиотека math.h, для работы со строками — библиотека string.h.

2.3. Программирование для AT89C5131

Примеры программ в нашей книге будут написаны для AT89C5131. Почему именно этот микроконтроллер? Причин несколько. Во-первых, его легко приобрести в России. Во-вторых, он имеет неплохие характеристики и воз-

можность прямого программирования через USB-порт, и, при этом, не требует специального оборудования. Кроме того, для его программирования не требуется специальных компиляторов.

Для создания кода мы будем использовать компилятор ICC8051.EXE и компоновщик XLINK.EXE компании IAR Systems (1991). Конечно, эти программные продукты устарели, но, тем не менее, позволяют создавать достаточно прозрачные и удобные для чтения примеры. Ну и, наконец, мы к ним просто привыкли. Заметим, что опыт [3] показывает, что примеры, написанные для IAR, легко модифицируются для любых других компиляторов.

2.3.1. Файл инициализации

В листинге 2.1 показано содержимое файла инициализации CSTARTUP.S03, который мы будем использовать в наших примерах.

Листинг 2.1. Файл инициализации CSTARTUP.S03

```
                         NAME CSTARTUP
;----------------------------------------------------------------;
;                    Параметры компилирования                    ;
;----------------------------------------------------------------;
LSTCND         +
LOCSYM         +
;----------------------------------------------------------------;
;                         EXTERN DEFINITION                      ;
;----------------------------------------------------------------;
        EXTERN  ?C_EXIT     ; Куда перейти после завершения
                            ; программы

        EXTERN  _R                      ; Банк рабочих регистров
        EXTERN  main                    ; Первая функция C
        EXTERN  exit                    ; Последняя функция C
;----------------------------------------------------------------;
; Сегмент стека C. Должен находиться во встроенном ОЗУ данных. ;
;----------------------------------------------------------------;
        RSEG    CSTACK
stack_begin:
        DS      30
        COMMON  INTVEC
```

```
;-----------------------------------------------------------------;
; В этой области располагаются функции-обработчики прерываний    ;
; С с указанными [векторами], и ассемблерные процедуры такого   ;
; же назначения, если они удовлетворяют соответствующим          ;
; требованиям.                                                    ;
;-----------------------------------------------------------------;
startup:
        LJMP    init_C
        RSEG    RCODE        ; Должен загружаться после INTVEC
        RSEG    D_CDATA
init_C:
        MOV     SP,#stack_begin - 1
;-----------------------------------------------------------------;
; Выбрать банк рабочих регистров (R0...R7). Символ _R получает   ;
; значение во время линковки программы.                          ;
;-----------------------------------------------------------------;
                  MOV     PSW,#_R
;-----------------------------------------------------------------;
; Если до вызова main необходима инициализация периферии с       ;
; помощью ассемблерных фрагментов программы, или если необходимо;
; разрешить прерывания, то эти коды нужно поместить сюда.        ;
;-----------------------------------------------------------------;
    MOV    SCON  ,#01010000B
    MOV    PCON  ,#10000000B
    MOV    BRDCON,#00011110B
    MOV    BRL   ,#100
    MOV    CKCON ,#00000001B
;-----------------------------------------------------------------;
;                              main()
;-----------------------------------------------------------------;
                             LCALL main
;-----------------------------------------------------------------;
;                             exit()                              ;
;-----------------------------------------------------------------;
Exit:
   SJMP Exit
END
```

2.3.2. Структуры дескрипторов

Описание структур дескрипторов показано в листинге 2.2.

Листинг 2.2. Структуры дескрипторов

```
/* Структура дескриптора устройства */
struct usb_st_device_descriptor
{
  byte   bLength;
  byte   bDescriptorType;
  uint16 bscUSB;
  byte   bDeviceClass;
  byte   bDeviceSubClass;
  byte   bDeviceProtocol;
  byte   bMaxPacketSize0;
  uint16 idVendor;
  uint16 idProduct;
  uint16 bcdDevice;
  byte   iManufacturer;
  byte   iProduct;
  byte   iSerialNumber;
  byte   bNumConfigurations;
};

/* Структура дескриптора конфигурации */
struct usb_st_configuration_descriptor
{
  byte   bLength;
  byte   bDescriptorType;
  uint16 wTotalLength;
  byte   bNumInterfaces;
  byte   bConfigurationValue;
  byte   iConfiguration;
  byte   bmAttibutes;
  byte   MaxPower;
};
```

```
/* Структура дескриптора интерфейса */
struct usb_st_interface_descriptor
{
  byte bLength;
  byte bDescriptorType;
  byte bInterfaceNumber;
  byte bAlternateSetting;
  byte bNumEndpoints;
  byte bInterfaceClass;
  byte bInterfaceSubClass;
  byte bInterfaceProtocol;
  byte iInterface;
};

/* Структура дескриптора конечной точки */
struct usb_st_endpoint_descriptor
{
  byte   bLength;
  byte   bDescriptorType;
  byte   bEndpointAddress;
  byte   bmAttributes;
  uint16 wMaxPacketSize;
  byte   bInterval;
};
```

Для описания дескриптора строки мы будем использовать макро-подстановку (листинг 2.3).

Листинг 2.3. Дескриптор строки

```
/* Дескриптор строки */
#define structSTRING_DESCRIPTOR(NAME, LEN) struct NAME \
                                  { \
                                    byte   bLength;\
                                    byte   bDescriptorType;\
                                    uint16 wstring[LEN];\
                                  }
```

Пример описания дескрипторов показан в листинге 2.4.

Листинг 2.4. Пример описания дескрипторов

```
/* Дескриптор устройства */
code struct usb_st_device_descriptor usb_device_descriptor =
{
  sizeof(usb_device_descriptor), DEVICE,
  USB_SPECIFICATION, DEVICE_CLASS, DEVICE_SUB_CLASS,
  DEVICE_PROTOCOL, EP_CONTROL_LENGTH, VENDOR_ID,
  PRODUCT_ID, RELEASE_NUMBER, MAN_INDEX, PRD_INDEX,
  SRN_INDEX, NB_CONFIGURATION
};
/* Дескриптор строки серийного номера */
#define USB_SERIAL_NUMBER {'1'<<8, '.'<<8, '0'<<8, '.'<<8, '0'<<8}
#define USB_SN_LENGTH   5
code structSTRING_DESCRIPTOR(usb_st_serial_number, USB_SN_LENGTH)
 usb_serial_number =
    { sizeof(usb_serial_number), STRING, USB_SERIAL_NUMBER };
```

2.3.3. Структура проекта

Стандартный проект IAR C состоит из нескольких файлов:

- CSTARTUP.S03 — файл кода загрузчика и описания основных сегментов;
- xxx.c51 — собственно файл программы;
- xxx.h — один или несколько заголовочных файлов;
- CL8051S.R03 — библиотека стандартных функций для 8051;
- LNK8051.XCL — командный файл настроек компоновщика.

Для компиляции программы используется командный файл make.bat, код которого показан в листинге 2.5.

Листинг 2.5. Командный файл для компиляции проекта

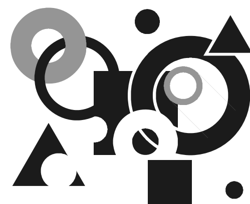
```
@Echo off
Cls
// Удаляем файлы ошибок
If exist err.txt  del err.txt
```

```
If exist errs.txt del errs.txt
// Компилируем CSTARTUP.S03
If exist LIB\CSTARTUP.r03 del LIB\CSTARTUP.r03
Bin\A8051.exe CSTARTUP.S03 LIB\LST\CSTARTUP.lst
If not exist CSTARTUP.r03 goto m_exit
copy CSTARTUP.r03 LIB\CSTARTUP.r03
del CSTARTUP.r03

// Компилируем основной файл
Bin\ICC8051.EXE test.c51 -l LIB\LST\test.LST -xDFT -a LIB\ASM\test.A51 -g
-C -ms -q -e> err.txt
If exist LIB\test.r03 del LIB\test.r03
If not exist test.r03 goto m_exit
copy test.r03 LIB\test.r03
del test.r03

// Линкуем
Bin\XLink.exe -f lnk8051.xcl > errs.txt

:m_exit
If exist err.txt  type err.txt
If exist errs.txt type errs.txt
```

Как видно из листинга 2.5, проект содержит следующую структуру каталогов:

- BIN — программы компиляторов и компоновщика;
- INC — стандартные заголовочные файлы и библиотеки;
- LIB — скомпилированные библиотеки для компоновки;
- LIB\ASM — скомпилированный ASM-код;
- LIB\LST — листинги модулей.

Два последних каталога используются для отладки кода. Полные коды командных файлов, файлов LNK8051.XCL и CSTARTUP.S03 можно найти на прилагаемом компакт-диске.



Глава 3

Инструменты

USB значительно сложнее, чем, например, последовательный интерфейс RS-232. Хорошая помехозащищенность и надежность оборачиваются сложностью отладки аппаратуры и кода. В этой главе мы опишем некоторые инструменты, делающие работу с USB более простой и комфортной.

3.1. Программаторы

Программатор предназначен для записи программы в память микроконтроллера. Существует несколько способов программирования, различающихся типом соединения, например, возможно программирование по SPI- или по USB-интерфейсу. Последний способ представляется наиболее удобным, т. к. не требует дополнительных разъемов, проводов и т. п.

Программатор состоит из трех составляющих: программы программатора, драйвера и кабеля для подключения.

3.1.1. Программатор Flip

Flip (рис. 3.1) является стандартным программатором, предоставляемым Atmel для работы со своими микроконтроллерами.

Следует обратить внимание на версию программы. Для корректной работы с USB, как говорится на сайте компании Atmel, необходима версия не ниже 2.2.0 с обязательным обновлением AtIsp.dll от 15 марта 2004 года (к сожалению, никаких версий внутри файла не прописано, поэтому узнать установлено ли обновление можно только косвенно по размеру файла, он должен быть 331 Кбайт). Однако мы рекомендуем использовать версию не ниже 2.4.4, т. к. предыдущие работают очень медленно.

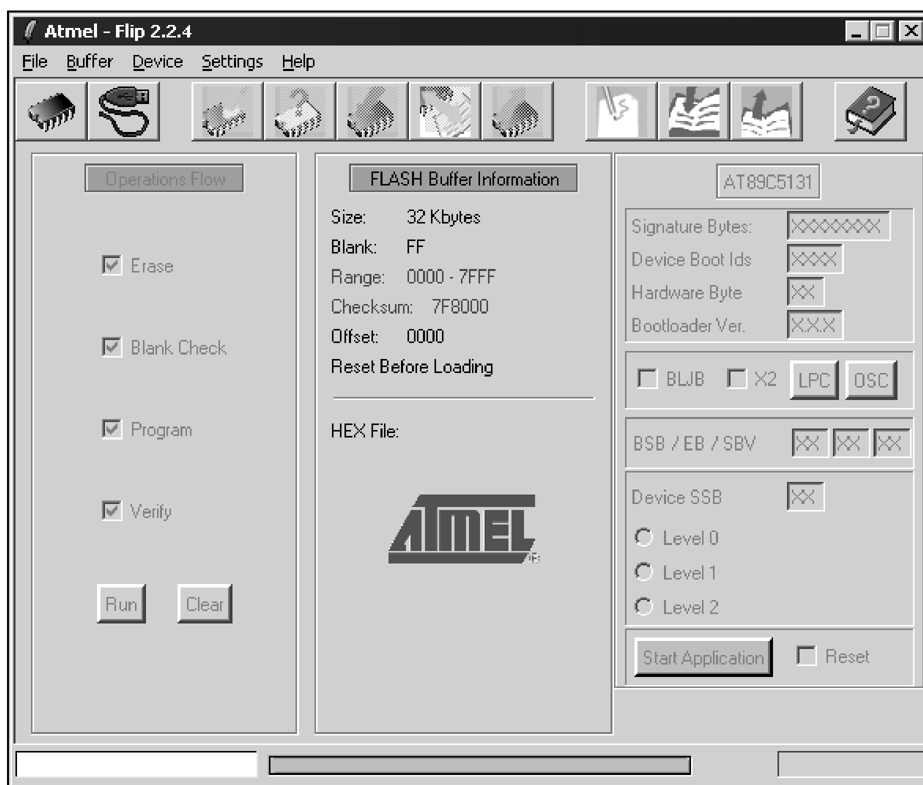


Рис. 3.1. Программатор Flip

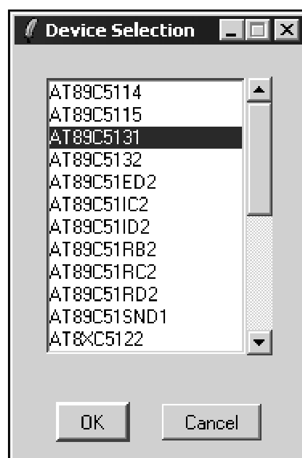


Рис. 3.2. Выбор типа микроконтроллера в программе Flip

К достоинствам программы можно отнести поддержку довольно большого числа микроконтроллеров (рис. 3.2) и наличие довольно большого числа функций:

- ❒ очистка, проверка, чтение и запись EEPROM- и флэш-памяти;
- ❒ возможность ручного редактирования буферов памяти (рис. 3.3);

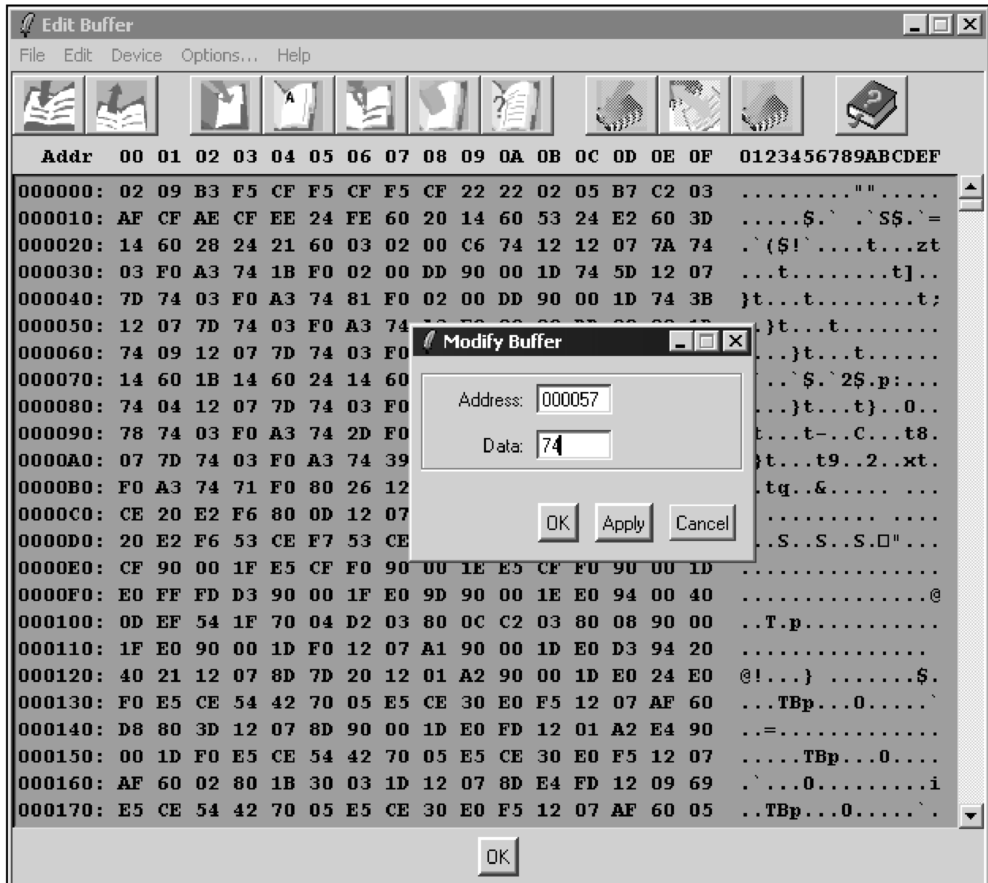


Рис. 3.3. Редактирование буферов памяти в программе Flip

- ❒ возможность загрузки и сохранения буферов в HEX-формат;
- ❒ отслеживание изменений в последнем загруженном файле (программа предлагает перезагрузить файл заново, если он изменился).

К недостаткам мы бы отнесли некоторую запутанность интерфейса, весьма краткий файл подсказки и необходимость каждый раз открывать USB-соединение, если устройство перезапустили (а это стандартная последовательность действий: загружаем программу, проверяем, исправляем, снова загружаем и т. д.).

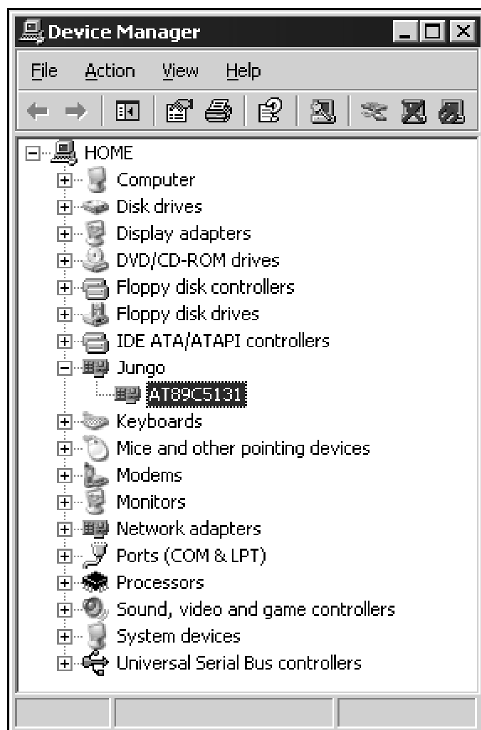


Рис. 3.4. Новое устройство — загрузчик AT89C5131

После инсталляции программы обязательно требуется выбрать в меню **Start→Programs→FLIP x.y.z→Install USB Driver**. Эта команда копирует в системный каталог необходимые файлы драйверов и INF-файлы. После подключения устройства (и запуска стартового загрузчика, *см. ниже*) оно должно появиться в списке устройств в ветке Jungo (рис. 3.4). Наличие драйверов Jungo (*см. разд. 3.3.2*) связано с тем, что USB-драйверы для Atmel предоставлены разработчиками WinDriver (рис. 3.5).

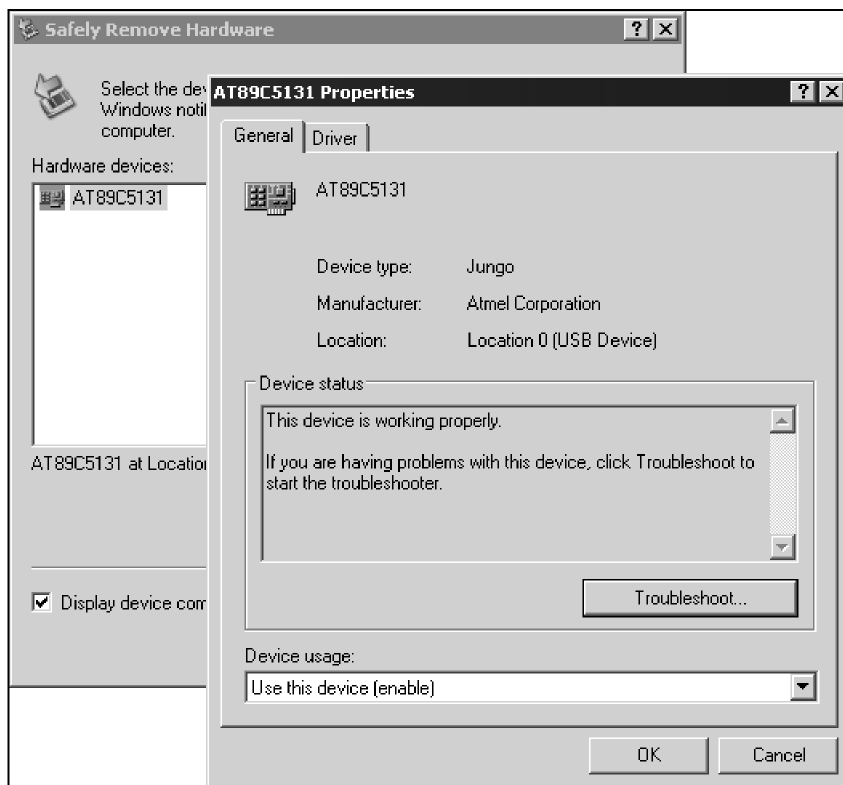


Рис. 3.5. Драйвер Jungo для USB

3.1.2. Программатор ER-Tronik

Еще один программатор удалось найти на немецком сайте **http://www.er-tronik.de**. Он предназначен только для программирования микроконтроллера AT89C5131 и только по интерфейсу USB (рис. 3.6). Конечно, узкая функциональность является существенным минусом, но, с другой стороны, дает, как нам кажется, и существенные плюсы. Простой и понятный интерфейс показался очень удобным:

- три кнопки в правой части представляют весь требуемый для работы набор функций (READ — прочитать, ERASE — очистить, WRITE — записать);
- совершенно очевидный редактор кода (в отличие от Flip, где для редактирования требуется двойной щелчок на байте, после чего открывается отдельное окно для редактирования);

- два раздельных окна для FLASH и EEPROM не требуют переключения между типами памяти, а настройки в правой части позволяют работать как с одним типом памяти, так и с обоими одновременно;
- особенно удобным показалась возможность загружать содержимое памяти из файла, список которых сохраняется в специальном меню, позволяя очень быстро загрузить файл еще раз (рис. 3.7).

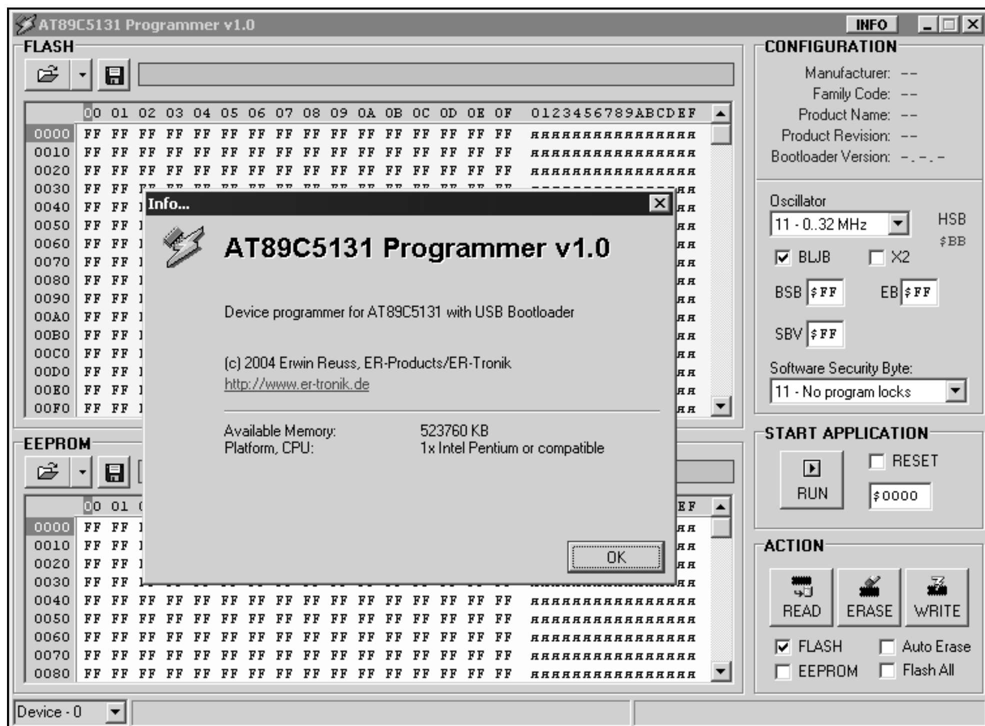


Рис. 3.6. Программатор ER-Tronik

К минусам можно отнести, пожалуй, только отсутствие документации, и USB-драйвер, при установке которого выдается сообщение о возможной несовместимости с Windows XP. Впрочем, никаких проблем при работе в Windows XP обнаружено не было (рис. 3.8).

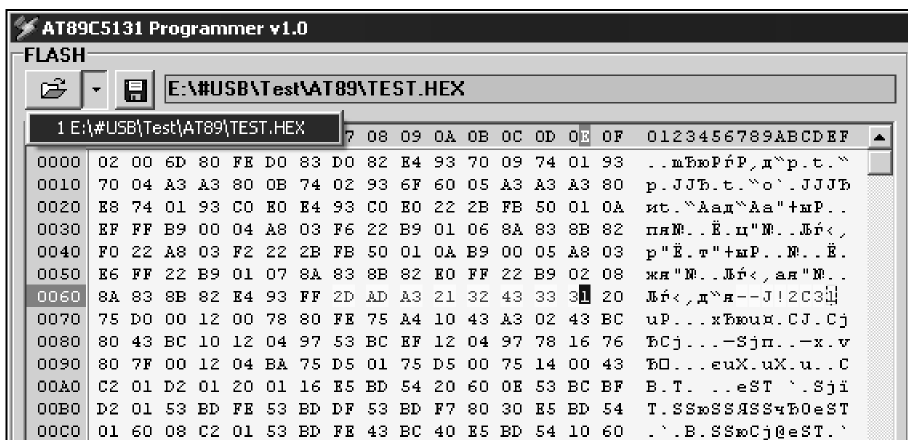


Рис. 3.7. Удобство и простота — преимущества ER-Tronik

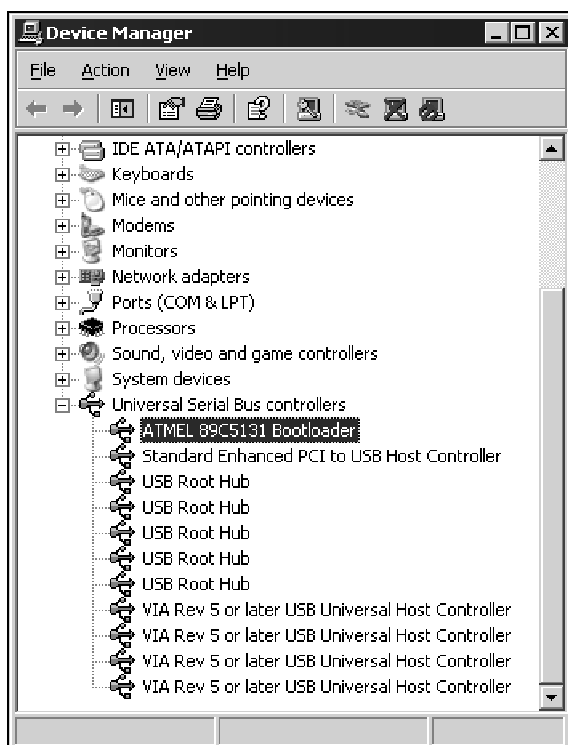


Рис. 3.8. Драйвер программатора ER-Tronik

3.2. Инструменты создания драйверов

Процесс написания драйверов достаточно сложен и трудоемок, и, конечно, на рынке ПО появились программы, облегчающие написание и тестирование драйверов.

3.2.1. NuMega Driver Studio

Этот программный комплекс включает помощник, интегрирующийся со средой разработки Microsoft Visual Studio. Последовательно отвечая на вопросы помощника, можно получить работоспособный скелет драйвера. Для компиляции полученного кода требуются библиотеки и классы NuMega и Microsoft DDK.

3.2.2. Jungo WinDriver

Пакет предназначен для разработки драйверов устройств, использующих стандарты PCI, Compact PCI, USB, ISA, ISA PnP, EISA и работающих под управлением операционных систем Windows 9*x*/ME/NT/2000. Позволяет обращаться к физической памяти, портам, устанавливать собственные обработчики аппаратных прерываний. Не требует наличия Windows DDK и программирования на уровне ядра. Используется графическая оболочка для диагностики оборудования и автоматической генерации кода на языке C (C++) или Pascal (Delphi).

3.2.3. Jungo KernelDriver

Пакет предназначен для разработки драйверов устройств, использующих стандарты PCI, Compact PCI, USB, ISA, ISA PnP, EISA и работающих на уровне ядра под управлением операционных систем Windows 9*x*/ME/NT/2000. Обеспечивает более высокую производительность, чем WinDriver. Требует наличия Windows DDK. Используется графическая оболочка для диагностики оборудования и автоматической генерации кода.

3.3. Средства Microsoft Visual Studio

Поставка Microsoft Visual Studio 6 включает несколько программ, которые могут оказаться полезными при разработке USB-устройств и драйверов.

3.3.1. Depends (Dependency Walker)

Программа Depends (рис. 3.9) позволяет посмотреть внутреннее содержание DLL-модулей: импортируемые и экспортируемые функции, использование других модулей, версии модулей и т. д.

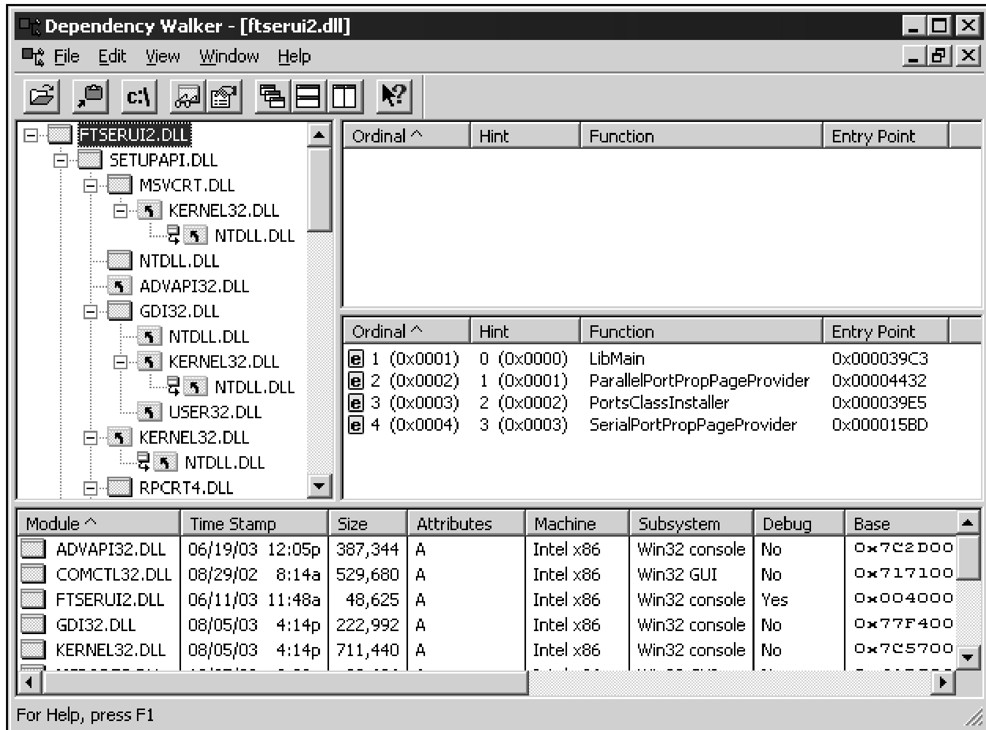


Рис. 3.9. Программа Depends

3.3.2. Error Lookup

Программа Error Lookup (рис. 3.10) отображает строковое значение ошибки по ее номеру.

3.3.3. GuidGen

Программа GuidGen (рис. 3.11) позволяет сгенерировать уникальный идентификатор GUID для ключа в реестре или для идентификатора драйвера (устройства).

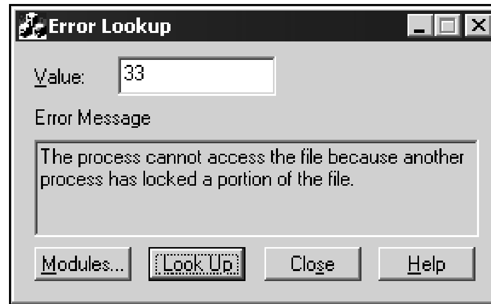


Рис. 3.10. Программа Error Lookup

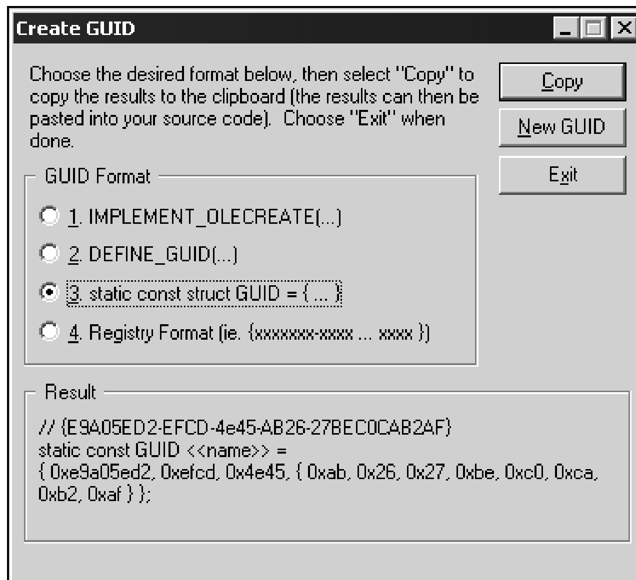


Рис. 3.11. Программа GuidGen

Более простая консольная программа UUIDGEN создает один идентификатор при каждом запуске.

3.4. Средства Microsoft DDK

Microsoft DDK (Driver Development Kit, комплект разработчика драйверов) включает множество утилит, которые помогут в работе с USB и созданием INF-файлов.

3.4.1. DeviceTree

Программа DeviceTree (рис. 3.12) позволяет отобразить дерево драйверов и соответствующих устройств. Отображение дерева устройств производится с двух точек зрения: с точки зрения принадлежности объектов устройств драйверам (режим D) и с точки зрения взаимной подчиненности объектов устройств при выполнении нумерации устройств.

Для каждого драйвера отображаются список обрабатываемых кодов (рабочих процедур), размер, атрибуты драйвера и множество других параметров. Однако полнота информации оборачивается другой стороной — при построении дерева устройств программа может привести к сбою или аварийной перезагрузке компьютера (о чем программа честно предупреждает при старте).

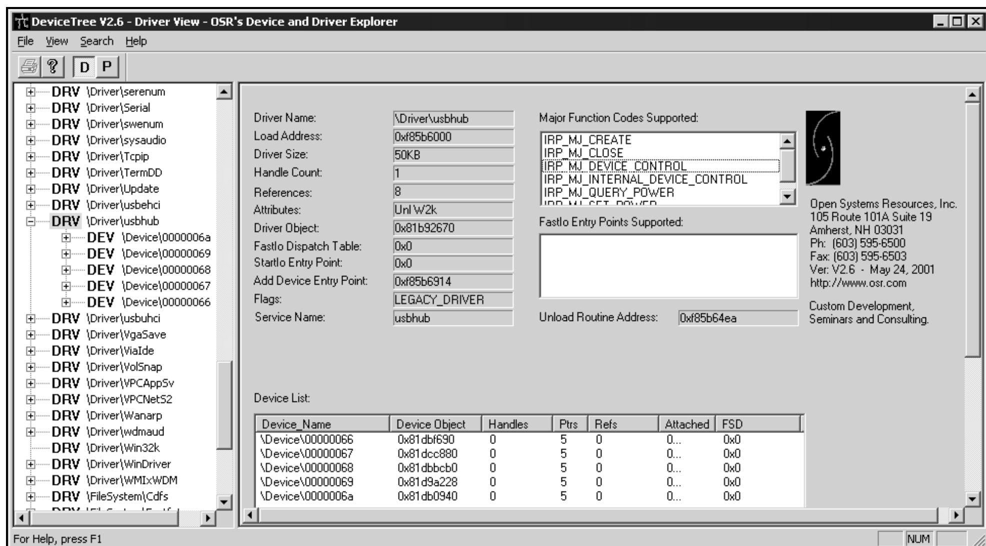


Рис. 3.12. Программа DeviceTree

3.4.2. DevCon

Консольная программа DevCon позволяет получить множество информации о системе. Ниже приведены примеры использования этой программы с различными ключами.

3.4.2.1. Ключ *classes*

Ключ `classes` отображает список всех зарегистрированных классов системы (листинг 3.1).

Листинг 3.1. Программа DevCon с ключом `classes`

```
// Windows XP. Список приводится с сокращениями.
>F:\WINXPDDK\tools\devcon\i386\devcon.exe classes
Listing 49 setup class(es).
WCEUSBS             : Windows CE USB Devices
USB                 : Universal Serial Bus controllers
CDROM               : DVD/CD-ROM drives
Computer            : Computer
DiskDrive           : Disk drives
Display             : Display adapters
fdc                 : Floppy disk controllers
hdc                 : IDE ATA/ATAPI controllers
Keyboard            : Keyboards
MEDIA               : Sound, video and game controllers
Modem               : Modems
Monitor             : Monitors
Mouse               : Mice and other pointing devices
MTD                 : PCMCIA and Flash memory devices
Ports               : Ports (COM & LPT)
Printer             : Printers
System              : System devices
Unknown             : Other devices
FloppyDisk          : Floppy disk drives
Processor           : Processors
HIDClass            : Human Interface Devices
LegacyDriver        : Non-Plug and Play Drivers
```

3.4.2.2. Ключ *driverfiles*

Ключ `driverfiles` отображает список драйверов, соответствующих выбранному классу устройств, или весь список драйверов (опция `*`). Пример вызова показан в листинге 3.2.

Листинг 3.2. Программа DevCon с ключом `driverfiles`

```
// Windows XP. Список приводится с сокращениями.
>F:\WINXPDDK\tools\devcon\i386\devcon.exe driverfiles *
ACPI\AUTHENTICAMD_-_X86_FAMILY_6_MODEL_10\_0
    Name: AMD Athlon(tm) XP 2200+
    Driver installed from f:\winxp\inf\cpu.inf [Processor_Inst].
    1 file(s) used by driver:
        F:\WINXP\System32\DRIVERS\processr.sys
ACPI\FIXEDBUTTON\2&DABA3FF&0
    Name: ACPI Fixed Feature Button
    Driver installed from f:\winxp\inf\machine.inf [NO_DRV]. No files
    used by driver.
ACPI\PNP0000\3&61AAA01&0
    Name: Programmable interrupt controller
    Driver installed from f:\winxp\inf\machine.inf [NO_DRV_PIC]. No files
    used by driver.
ACPI\PNP0100\3&61AAA01&0
    Name: System timer
    Driver installed from f:\winxp\inf\machine.inf [NO_DRV_X]. No files
    used by driver.
ACPI\PNP0200\3&61AAA01&0
    Name: Direct memory access controller
    Driver installed from f:\winxp\inf\machine.inf [NO_DRV_X]. No files
    used by driver.
ACPI\PNP0303\3&61AAA01&0
    Name: Standard 101/102-Key or Microsoft Natural PS/2 Keyboard
    Driver installed from f:\winxp\inf\keyboard.inf [STANDARD_Inst].
    2 file(s) used by driver:
        F:\WINXP\System32\DRIVERS\i8042prt.sys
        F:\WINXP\System32\DRIVERS\kbdclass.sys
ACPI\PNP0400\1
    Name: Printer Port (LPT1)
    Driver installed from f:\winxp\inf\msports.inf [LptPort]. 1 file(s)
    used by driver:
```

3.4.2.3. Ключ *hwids*

Ключ `hwids` отображает список аппаратных идентификаторов устройств системы (листинг 3.3).

Листинг 3.3. Программа DevCon с ключом `hwids`

```
// Windows XP. Список приводится с сокращениями.
>F:\WINXPDDK\tools\devcon\i386\devcon.exe hwids *
HID\VID_1241&PID_1111\6&25B17C15&0&0000
    Name: HID-compliant mouse
    Hardware ID's:
        HID\Vid_1241&Pid_1111&Rev_0100
        HID\Vid_1241&Pid_1111
        HID_DEVICE_SYSTEM_MOUSE
        HID_DEVICE_UP:0001_U:0002
        HID_DEVICE
PCI\VEN_1106&DEV_3038&SUBSYS_30381106&REV_81\3&61AAA01&0&80
    Name: VIA Rev 5 or later USB Universal Host Controller
    Hardware ID's:
        PCI\VEN_1106&DEV_3038&SUBSYS_30381106&REV_81
        PCI\VEN_1106&DEV_3038&SUBSYS_30381106
        PCI\VEN_1106&DEV_3038&CC_0C0300
        PCI\VEN_1106&DEV_3038&CC_0C03
    Compatible ID's:
        PCI\VEN_1106&DEV_3038&REV_81
        PCI\VEN_1106&DEV_3038
        PCI\VEN_1106&CC_0C0300
        PCI\VEN_1106&CC_0C03
        PCI\VEN_1106
        PCI\CC_0C0300
        PCI\CC_0C03
USB\ROOT_HUB\4&319D8414&0
    Name: USB Root Hub
    Hardware ID's:
```

```
USB\ROOT_HUB&VID1106&PID3038&REV0081
USB\ROOT_HUB&VID1106&PID3038
USB\ROOT_HUB
```

3.4.2.4. Ключ *rescan*

Ключ `rescan` указывает системе начать поиск новых устройств. В отличие от аналогичной процедуры, запускаемой из менеджера устройств, никаких оконных сообщений отображаться не будет.

3.4.2.5. Ключ *stack*

Ключ `stack` собирает информацию обо всех устройствах в стеке выбранного класса (листинг 3.4).

Листинг 3.4. Программа DevCon с ключом `stack`

```
// Windows XP. Список приводится с сокращениями.
>F:\WINXPDDK\tools\devcon\i386\devcon.exe stack *USB*
USB\ROOT_HUB\4&1EE3D36F&0
    Name: USB Root Hub
    Setup Class: {36FC9E60-C465-11CF-8056-444553540000} USB
    Class upper filters:
        hhdusbh
    Controlling service:
        usbhub
USB\ROOT_HUB\4&319D8414&0
    Name: USB Root Hub
    Setup Class: {36FC9E60-C465-11CF-8056-444553540000} USB
    Class upper filters:
        hhdusbh
    Controlling service:
        usbhub
USB\ROOT_HUB\4&350B3C2C&0
    Name: USB Root Hub
    Setup Class: {36FC9E60-C465-11CF-8056-444553540000} USB
    Class upper filters:
        hhdusbh
```

```
    Controlling service:
        usbhub
USB\ROOT_HUB\4&C2A22CA&0
    Name: USB Root Hub
    Setup Class: {36FC9E60-C465-11CF-8056-444553540000} USB
    Class upper filters:
        hhdusbh
    Controlling service:
        usbhub
USB\ROOT_HUB20\4&205A5C46&0
    Name: USB Root Hub
    Setup Class: {36FC9E60-C465-11CF-8056-444553540000} USB
    Class upper filters:
        hhdusbh
    Controlling service:
        usbhub
5 matching device(s) found.
```

3.4.2.6. Ключ *status*

Ключ `status` отображает состояние драйверов системы (листинг 3.5).

Листинг 3.5. Программа DevCon с ключом `status`

```
// Windows XP. Список приводится с сокращениями.
>F:\WINXPDDK\tools\devcon\i386\devcon.exe status *
PCI\VEN_1106&DEV_3038&SUBSYS_30381106&REV_81\3&61AAA01&0&80
    Name: VIA Rev 5 or later USB Universal Host Controller
    Driver is running.
HID\VID_1241&PID_1111\6&25B17C15&0&0000
    Name: HID-compliant mouse
    Driver is running.
ACPI\PNP0000\3&61AAA01&0
    Name: Programmable interrupt controller
    Device is currently stopped.
```

3.4.3. ChkInf и GenInf

Программа GenInf (рис. 3.13) позволяет сгенерировать INF-файл, последовательно отвечая на вопросы помощника, а программа ChkInf — проверить правильность INF-файла. *Пример использования этой программы приведен в разд. 13.2.*

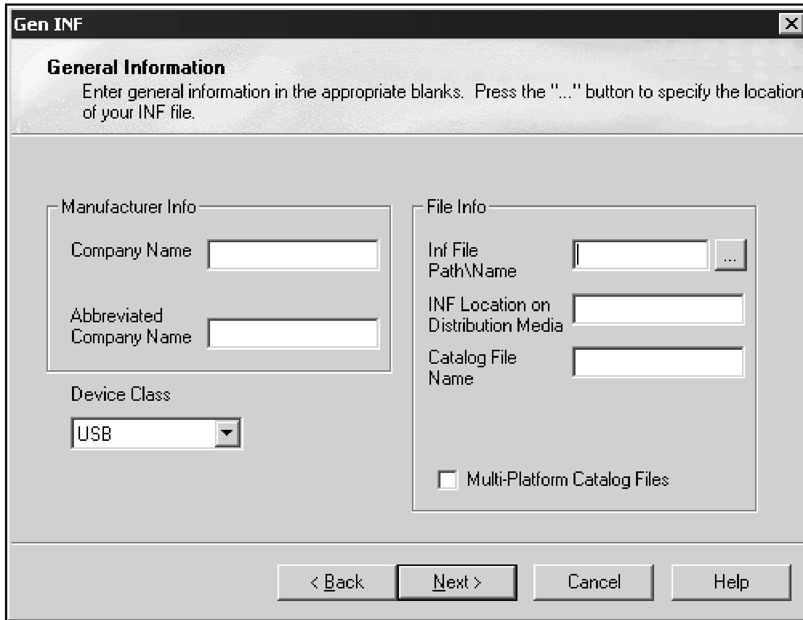


Рис. 3.13. Программа GenInf

3.5. Средства CompuWare Corporation

В поставку NuMega SoftICE Driver Suite (теперь CompuWare Corporation) кроме известного отладчика SoftICE входит несколько полезных утилит.

3.5.1. Monitor

Программа Monitor (рис. 3.14) позволяет динамически загружать, запускать, останавливать и выгружать драйверы, а также производить мониторинг загрузки драйверов.

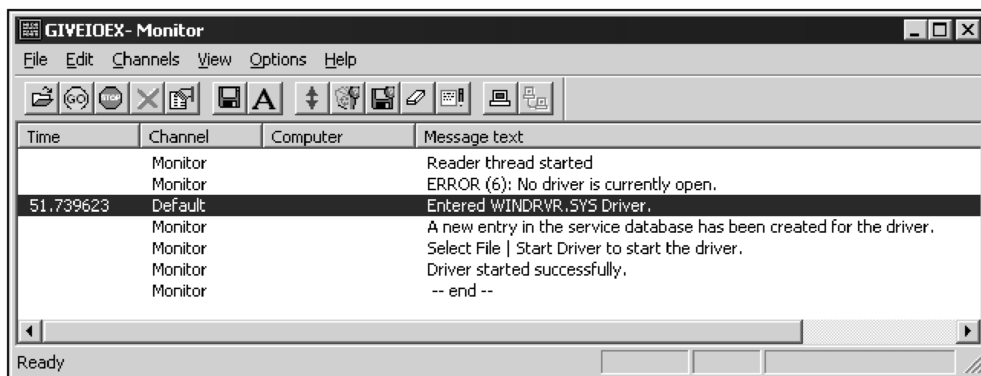


Рис. 3.14. Программа Monitor

3.5.2. SymLink

Программа SymLink (рис. 3.15) отображает список символьных имен, зарегистрированных в системе.

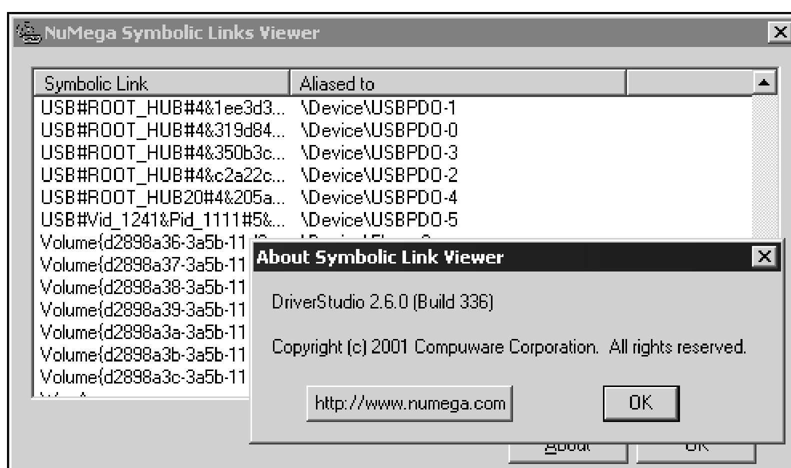


Рис. 3.15. Программа SymLink

3.5.3. EzDriverInstaller

Программа EzDriverInstaller (рис. 3.16) позволяет запустить или остановить драйвер, записанный в выбранном INF-файле.

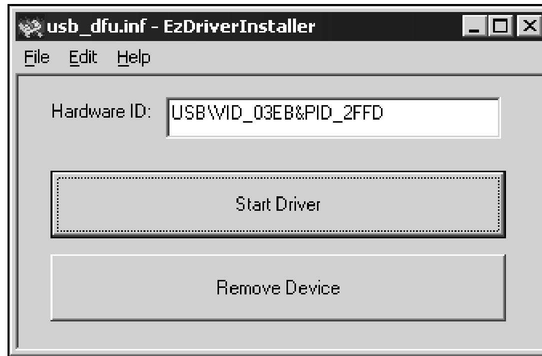


Рис. 3.16. Программа EzDriverInstaller

3.5.4. WdmSniff

Программа WdmSniff (рис. 3.17) позволяет производить мониторинг пакетов IRP практически любого драйвера (рис. 3.18).

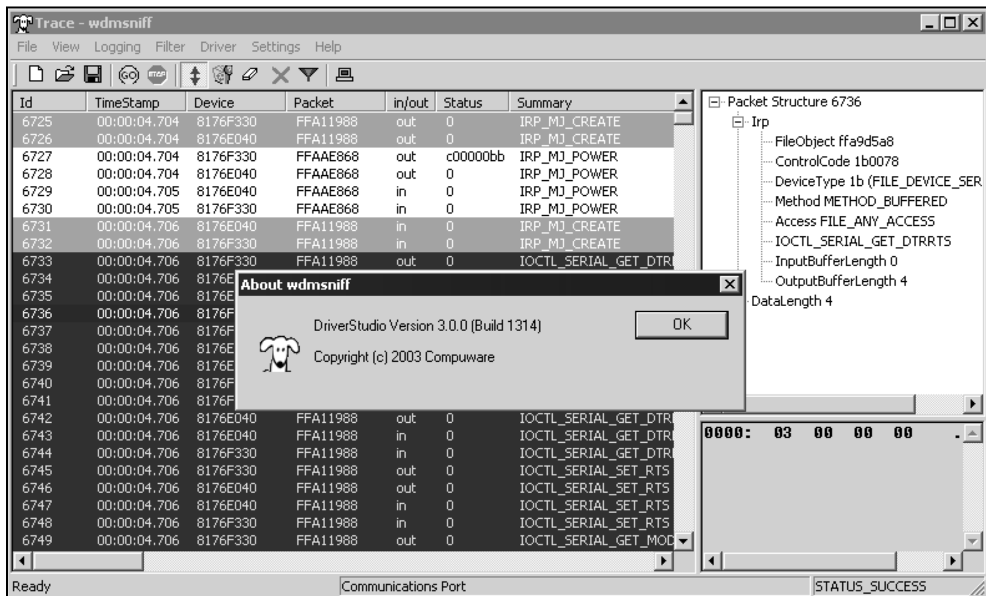


Рис. 3.17. Программа WdmSniff

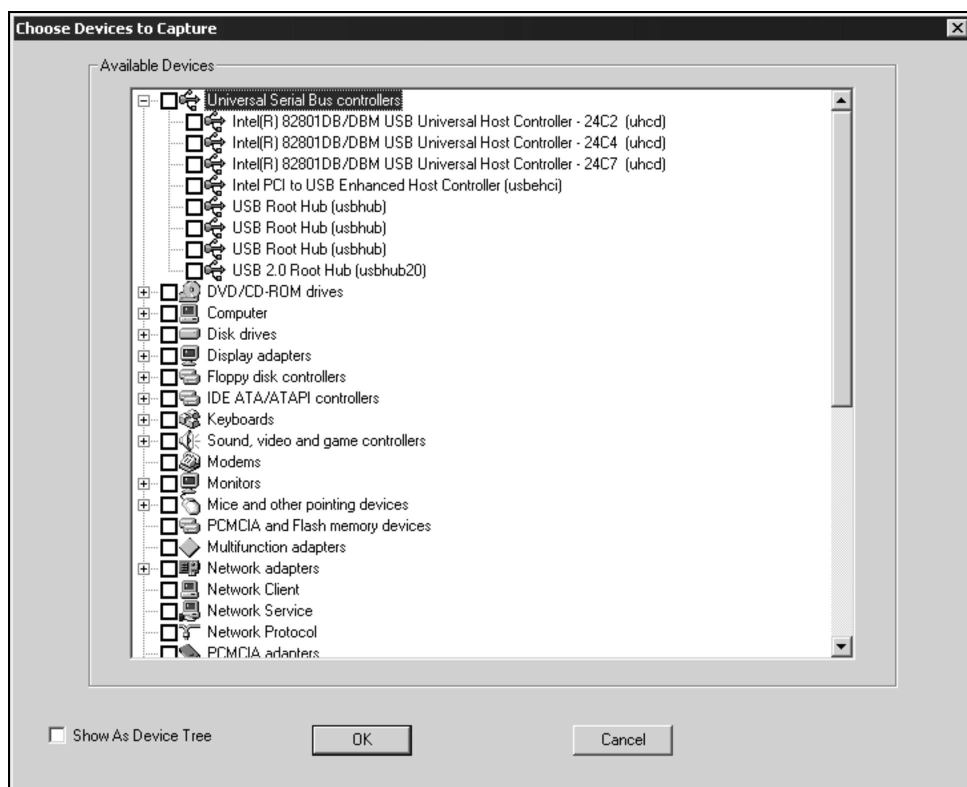


Рис. 3.18. Выбор устройств в программе WdmSniff

3.6. Средства SysInternals

3.6.1. WinObj

Программа WinObj (рис. 3.19) отображает список символьных имен, зарегистрированных в системе.

3.7. Средства USB Forum

3.7.1. HID Descriptor Tool

Программа HID Descriptor Tool (рис. 3.20) позволяет просто и достаточно удобно создавать и редактировать дескрипторы репортов (*см. гл. 6*) для HID-устройств.

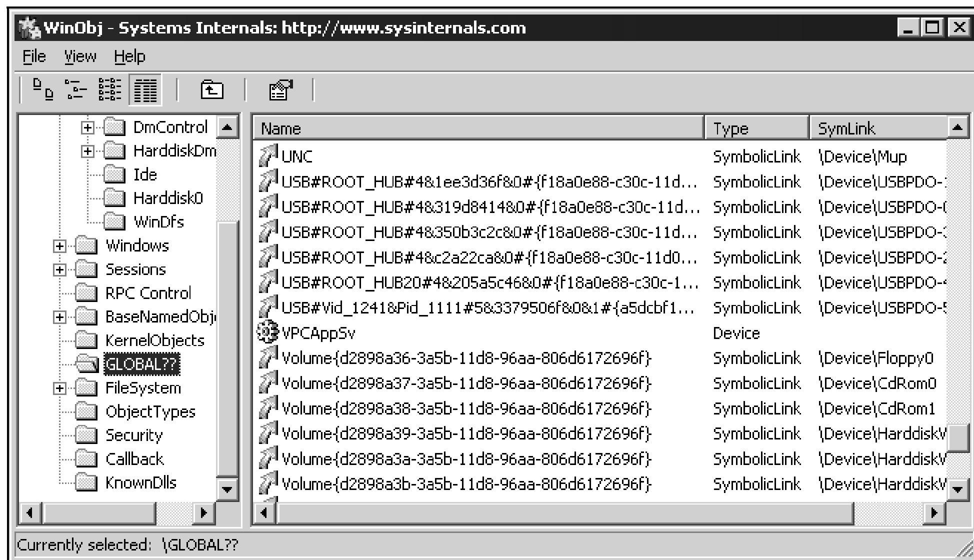


Рис. 3.19. Программа WinObj

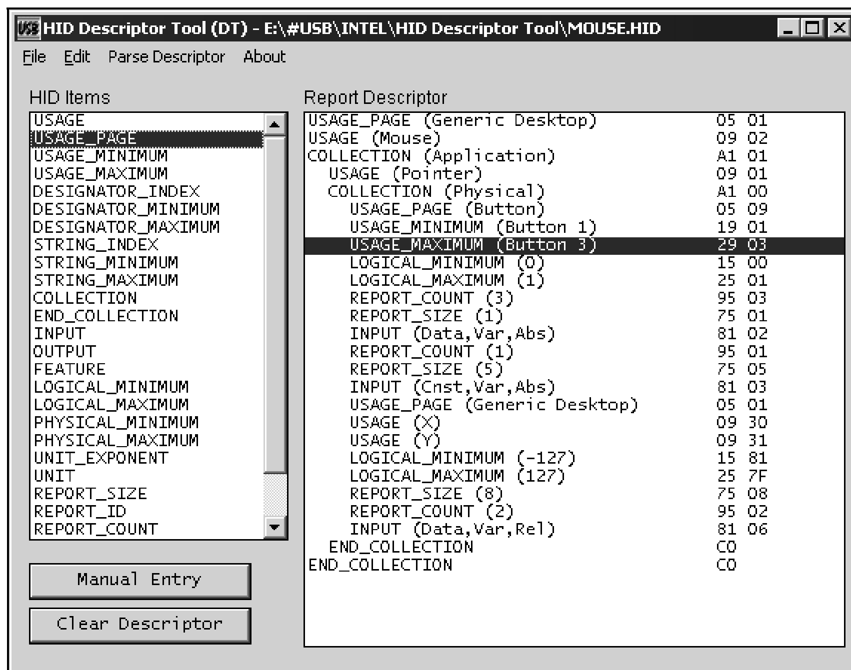


Рис. 3.20. Программа HID Descriptor Tool

Созданные дескрипторы можно сохранить в специальном формате HID для последующего редактирования, а также сохранить в виде фрагмента на языке ассемблера или C (листинг 3.6). Это незаменимый инструмент при разработке HID-устройств.

Листинг 3.6. Сохранение дескриптора репорта в разных форматах

```
// Сохранение в формате заголовочного файла C
// E:\#USB\INTEL\HID Descriptor Tool\MOUSE.HID.h
char ReportDescriptor[50] = {
    0x05, 0x01,            // USAGE_PAGE (Generic Desktop)
    0x09, 0x02,            // USAGE (Mouse)
    0xa1, 0x01,            // COLLECTION (Application)
    0x09, 0x01,            //   USAGE (Pointer)
    0xa1, 0x00,            //   COLLECTION (Physical)
    0x05, 0x09,            //     USAGE_PAGE (Button)
    0x19, 0x01,            //     USAGE_MINIMUM (Button 1)
    0x29, 0x03,            //     USAGE_MAXIMUM (Button 3)
    0x15, 0x00,            //     LOGICAL_MINIMUM (0)
    0x25, 0x01,            //     LOGICAL_MAXIMUM (1)
    0x95, 0x03,            //     REPORT_COUNT (3)
    0x75, 0x01,            //     REPORT_SIZE (1)
    0x81, 0x02,            //     INPUT (Data,Var,Abs)
    0x95, 0x01,            //     REPORT_COUNT (1)
    0x75, 0x05,            //     REPORT_SIZE (5)
    0x81, 0x03,            //     INPUT (Cnst,Var,Abs)
    0x05, 0x01,            //     USAGE_PAGE (Generic Desktop)
    0x09, 0x30,            //     USAGE (X)
    0x09, 0x31,            //     USAGE (Y)
    0x15, 0x81,            //     LOGICAL_MINIMUM (-127)
    0x25, 0x7f,            //     LOGICAL_MAXIMUM (127)
    0x75, 0x08,            //     REPORT_SIZE (8)
    0x95, 0x02,            //     REPORT_COUNT (2)
    0x81, 0x06,            //     INPUT (Data,Var,Rel)
    0xc0,                  //   END_COLLECTION
```

```
    0xc0                    // END_COLLECTION
};
// Сохранение в виде данных ассемблера
        db  5h,  1h         ; USAGE_PAGE (Generic Desktop)
        db  9h,  2h         ; USAGE (Mouse)
        db a1h,  1h         ; COLLECTION (Application)
        db  9h,  1h         ;   USAGE (Pointer)
        db a1h,  0h         ;   COLLECTION (Physical)
        db  5h,  9h         ;     USAGE_PAGE (Button)
        db 19h,  1h         ;     USAGE_MINIMUM (Button 1)
        db 29h,  3h         ;     USAGE_MAXIMUM (Button 3)
        db 15h,  0h         ;     LOGICAL_MINIMUM (0)
        db 25h,  1h         ;     LOGICAL_MAXIMUM (1)
        db 95h,  3h         ;     REPORT_COUNT (3)
        db 75h,  1h         ;     REPORT_SIZE (1)
        db 81h,  2h         ;     INPUT (Data,Var,Abs)
        db 95h,  1h         ;     REPORT_COUNT (1)
        db 75h,  5h         ;     REPORT_SIZE (5)
        db 81h,  3h         ;     INPUT (Cnst,Var,Abs)
        db  5h,  1h         ;     USAGE_PAGE (Generic Desktop)
        db  9h, 30h         ;     USAGE (X)
        db  9h, 31h         ;     USAGE (Y)
        db 15h, 81h         ;     LOGICAL_MINIMUM (-127)
        db 25h, 7fh         ;     LOGICAL_MAXIMUM (127)
        db 75h,  8h         ;     REPORT_SIZE (8)
        db 95h,  2h         ;     REPORT_COUNT (2)
        db 81h,  6h         ;     INPUT (Data,Var,Rel)
        db c0h              ;   END_COLLECTION
        db c0h              ; END_COLLECTION
```

3.8. USB Command Verifier

Еще одна программа с сайта **http://www.usb.org** позволяет протестировать устройство на его совместимость со стандартом. Обязательное условие рабо-

ты этой программы — наличие порта USB 2.0. Программа позволяет проводить практически все возможные тесты совместимости (рис. 3.21).

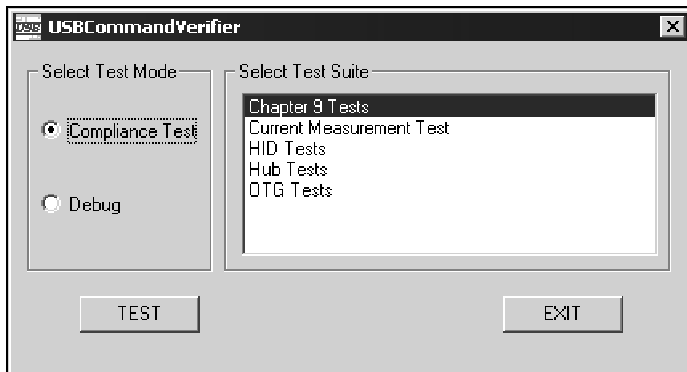


Рис. 3.21. Программа USB Command Verifier

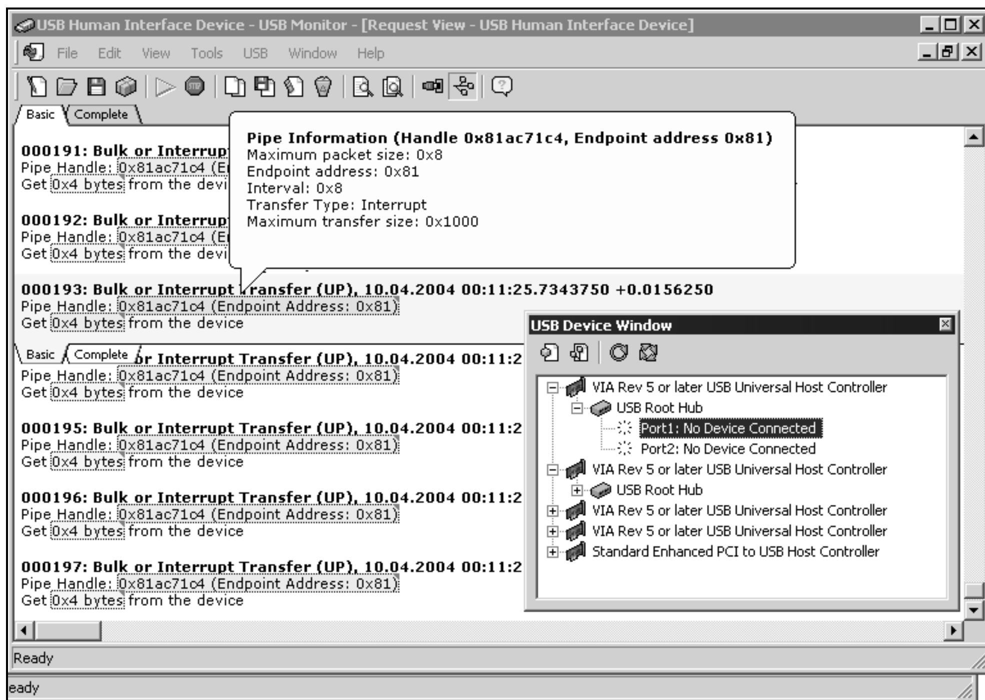


Рис. 3.22. Программа USB Monitor

3.9. Средства HDD Software

Программа USB Monitor (рис. 3.22) разработана компанией HDD Software (**http://www.hhdsoftware.com**) и является, пожалуй, наиболее мощной программой мониторинга USB-пакетов. Программа имеет простой и удобный интерфейс.

3.10. Средства Sourceforge

Для мониторинга USB-трафика можно использовать программу SnoopyPro (рис. 3.23), разрабатываемую в рамках проектов с открытым исходным кодом. Программа доступна на сайте **http://sourceforge.net/projects/usbsnoop**.

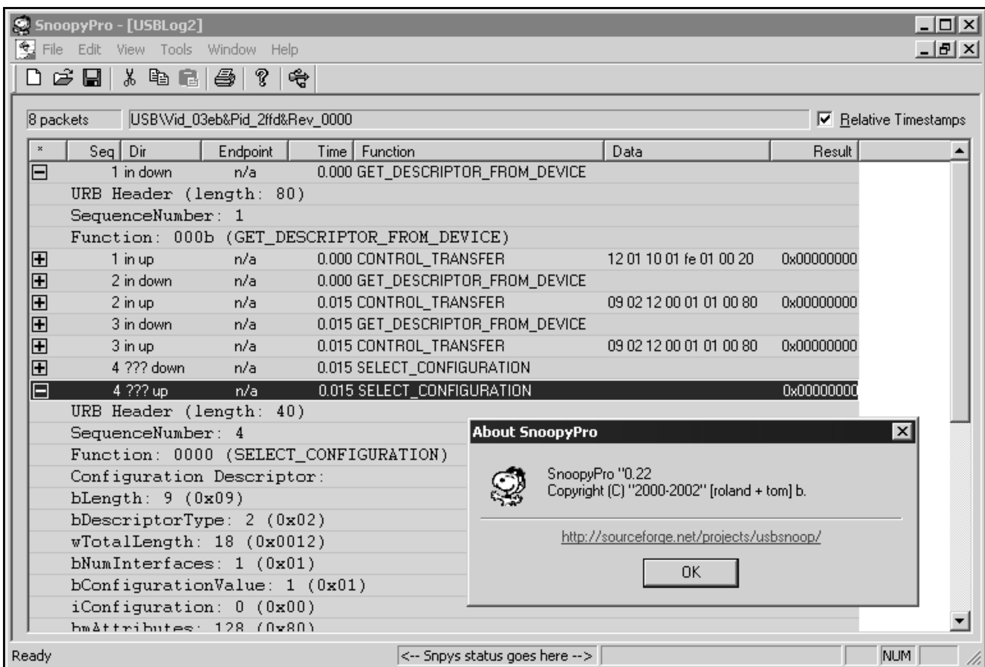


Рис. 3.23. Программа SnoopyPro

3.11. Программа мониторинга Bus Hound

Программа Bus Hound компании PeriSoft позволяет отследить трафик не только USB-устройств, но и множества других (рис. 3.24, 3.25). Кроме того, программа позволяет выполнять прямые команды (рис. 3.26).

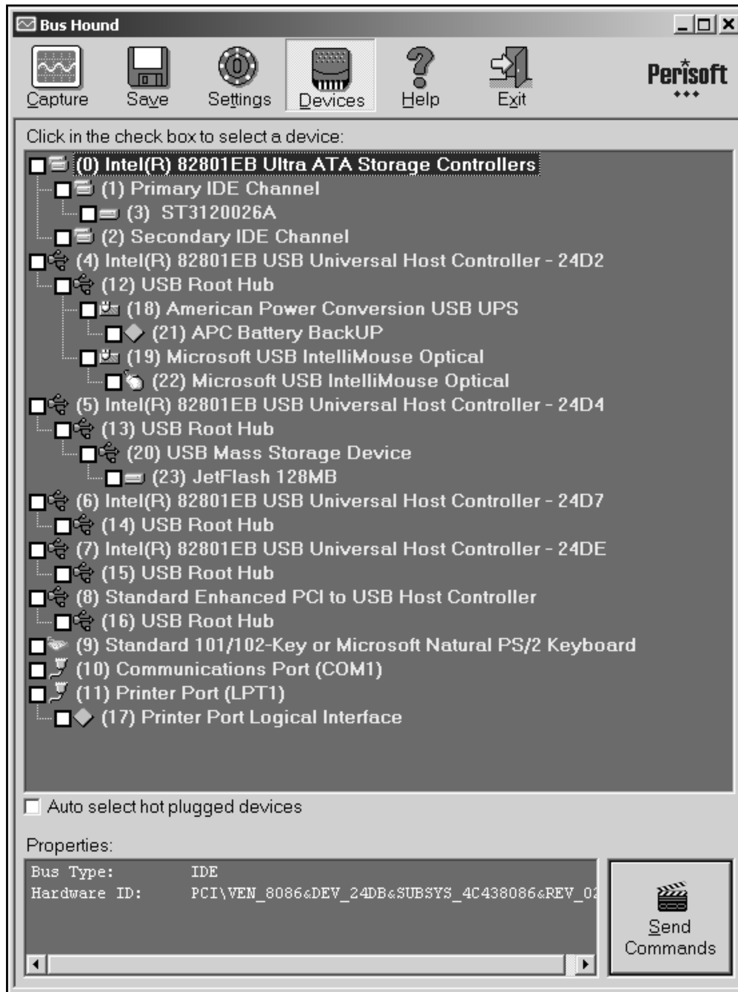


Рис. 3.24. Программа Bus Hound (выбор устройства)

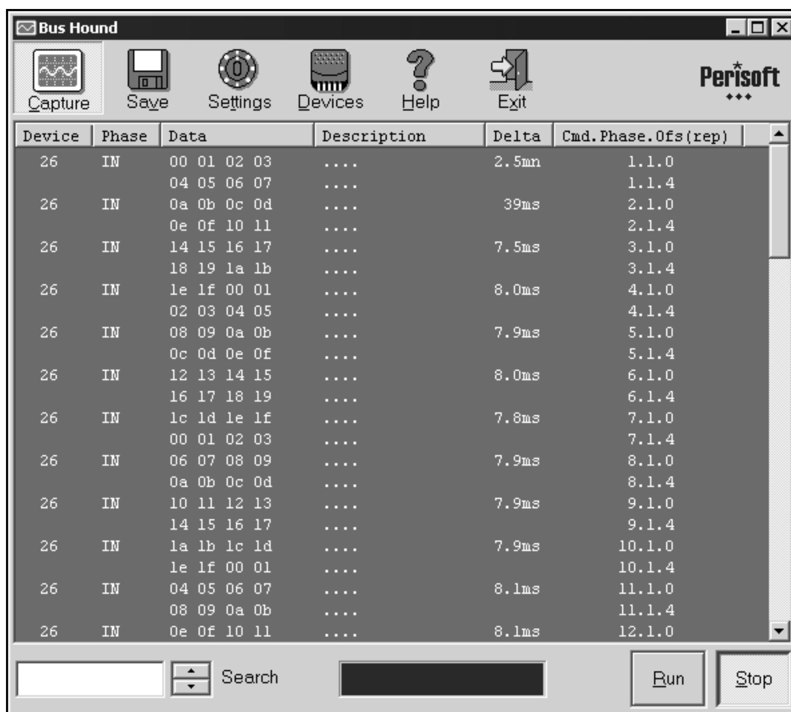


Рис. 3.25. Программа Bus Hound (мониторинг трафика)

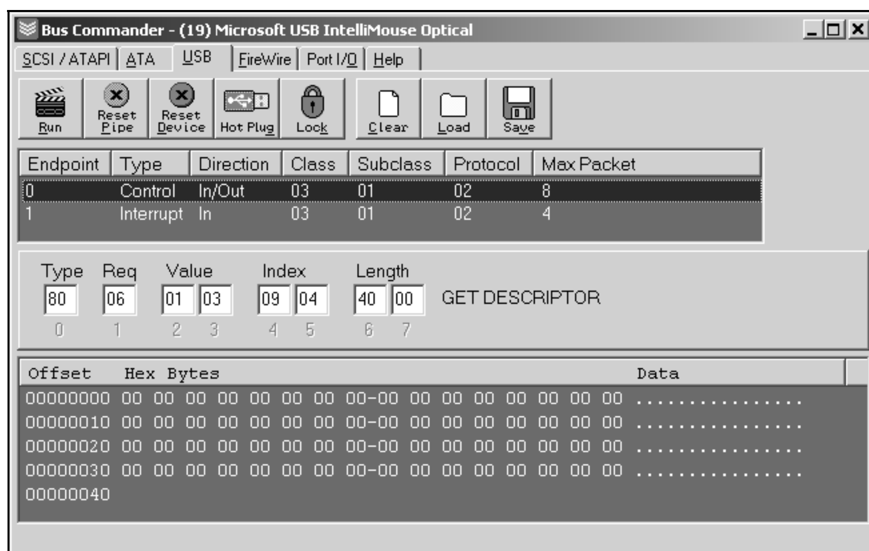
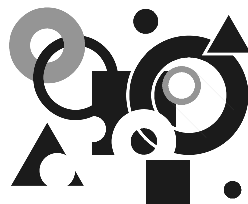


Рис. 3.26. Программа Bus Hound (выполнение команд)

Глава 4



Принципы использования функций Win32 в .NET

На смену платформе Win32 пришла платформа .NET, имеющая свои особенности программирования аппаратных средств. В этой главе мы опишем основные принципы использования функций работы с аппаратурой в Windows для платформы .NET.

Важно

Мы подразумеваем, что читатели имеют знания в области программирования на платформе .NET. Объем книги не позволяет приводить определения базовых понятий языка C# и платформы .NET, такие как поля, атрибуты и типы данных. Эти сведения можно найти в соответствующей литературе [11].

4.1. Общие сведения

Microsoft .NET Framework позволяет разработчикам создавать программы, использующие множество современных технологий. Новая платформа предоставляет разработчикам множество преимуществ [11]:

- единая программная модель. Весь прикладной сервис представлен общей объектно-ориентированной программной моделью;
- отсутствие проблем с версиями;
- упрощенная установка и удаление программ;
- возможность создавать мультиплатформенные приложения;
- упрощенное повторное использование кода;
- автоматическое управление памятью (сбор мусора);
- проверка безопасности типов;
- взаимодействие с существующим кодом.

Последний пункт стоит обсудить более подробно. С одной стороны, .NET позволяет создавать программы, работающие на нескольких платформах. При компиляции кода для .NET Framework компилятор генерирует код на *общем промежуточном языке* (CIL, Common Intermediate Language), а уже при исполнении этот код транслируется в команды процессора. Это означает, что приложение можно развернуть на любой машине, на которой работает *исполняющая часть* .NET (CLR, Common Language Runtime). Код, работающий с помощью .NET Framework, называется *управляемым кодом* (managed code), а код, работающий вне платформы .NET, называется *неуправляемым кодом* (unmanaged code).

С другой стороны, понятно, что программы, написанные с помощью .NET-библиотек (FCL, Framework Class Library), должны иметь доступ к реальному оборудованию. В этой главе мы будем обсуждать методы доступа к оборудованию для Windows.

Доступ к оборудованию для Windows означает возможность вызова Win32-функций. В некоторых случаях мы будем использовать новые технологии, предоставляемые Windows XP. В любом случае программа, работающая с оборудованием, будет привязана к Windows.

4.2. Импорт функций Win32

Как мы уже говорили, программа, написанная на .NET, может вызывать функции Win32. Для этого функции должны быть декларированы с помощью атрибута `DllImport`, как показано в листинге 4.1. Функции, использующие указатели, считаются несовместимыми с управляемым кодом, поэтому при их описании требуется указывать ключевое слово `unsafe`.

Атрибут `DllImport` имеет несколько полезных полей:

- `CharSet` (тип `CharSet`) позволяет задавать кодировку символов строк, используемых функцией;
- `EntryPoint` (тип `string`) может задавать имя импортируемой функции. По умолчанию имя импортируемой функции совпадает с именем описываемой функции;
- `SetLastError` (тип `bool`) указывает, что после вызова функции будет вызываться функция Win32 `GetLastError`. Код ошибки можно получить с помощью метода `Marshal.GetLastWin32Error`.

Листинг 4.1. Пример импорта функций Win32

```
[DllImport("hid.dll", SetLastError=true)]
static extern unsafe void HidD_GetHidGuid(ref GUID lpHidGuid);
```

```
[DllImport("hid.dll")]
static public extern bool HidD_FlushQueue(int HidDeviceObject);
```

Еще один полезный пример показан в листинге 4.2.

Листинг 4.2. Пример импорта функции `Beep`

```
// Описание
[DllImport("kernel32.dll")]
static extern bool Beep (int freq, int duration);
// Вызов
Beep(1000, 100);
//
// Или еще один вариант
//
// Описание
[DllImport("user32.dll")]
static extern void MessageBeep(int uType);
// Вызов
MessageBeep(0);
```

Параметры функций, описанные как указатели, импортируются в C# с помощью ключевого слова `ref`. В Win32 довольно часто используются функции, самостоятельно определяющие размер используемых буферов. Для этого при первом вызове им передается `null`, а при втором — буфер нужного размера. Однако, в отличие от C или Delphi, в качестве параметра, описанного с помощью `ref`, нельзя передавать `null`, поэтому в C# приходится делать два дублирующих описания. В первом описании обычно используется параметр типа `int *`, что позволяет передавать `null`, а во втором используется указатель `ref` , что позволяет получать данные.

4.3. Структуры

Для работы с Win32-функциями используются структуры данных, разработанные для Win32-платформы. Структуры .NET Framework имеют свои особенности, требующие дополнительных усилий для обеспечения совместимости с Win32.

4.3.1. Атрибут *StructLayout*[1]

Для повышения производительности приложения CLR дано право устанавливать порядок размещения полей типа (в частности, и структур). Разумеется, при обращении к функциям Win32 такое поведение недопустимо. К счастью, разработчики .NET предоставляют способ управления полями с помощью атрибута `System.Runtime.InteropServices.StructLayoutAttribute`.

Допускается указание следующих типов размещения полей:

- `LayoutKind.Auto` — автоматическое расположение полей. Попытки использовать такие объекты вне управляемого кода сгенерируют исключение;
- `LayoutKind.Sequential` — последовательное расположение полей. Поля будут располагаться согласно последовательности их описания. Поля выравниваются согласно значению атрибута `StructLayoutAttribute.Pack`;
- `LayoutKind.Explicit` — последовательное расположение полей. Каждое поле должно иметь атрибут `FieldOffsetAttribute` для указания его расположения (смещения).

По умолчанию для ссылочных типов (классов) компилятор Microsoft C# выбирает `LayoutKind.Auto`, а для размерных типов (структур) — `LayoutKind.Sequential`. Пример описания структур с атрибутом `StructLayout` показан в листинге 4.3.

Листинг 4.3. Пример описания структур с атрибутом `StructLayout`

```
// Последовательное описание полей
[StructLayout(LayoutKind.Sequential)]
public struct Point
{
   public int x;
   public int y;
}
// Прямое указание последовательности полей
[StructLayout(LayoutKind.Explicit)]
public struct Rect
{
   [FieldOffset(0)]  public int left;
```

[1] При описании атрибута .NET позволяет опускать слово Attribute. Таким образом, описания `StructLayoutAttribute` и `StructLayout` равносильны.

```
    [FieldOffset(4)]  public int top;
    [FieldOffset(8)]  public int right;
    [FieldOffset(12)] public int bottom;
}
// С помощью смещений можно создавать объединение
[StructLayout(LayoutKind.Explicit)]
public struct MyUnion
{
    [FieldOffset(0)]
    public int i;
    [FieldOffset(0)]
    public double d;
}
```

Как видно из листинга, атрибуты позволяют гибко управлять расположением полей и даже создавать *объединения* (union). Кроме того, атрибут `StructLayout` имеет дополнительные поля:

- `CharSet` позволяет управлять набором символов. Допускается указание `CharSet.Unicode` (будет использоваться `LPWSTR`), `CharSet.Ansi` (будет использоваться `LPSTR`) или `CharSet.Auto` (будет использоваться `Ansi` для Windows 98 и Windows ME и Unicode для остальных платформ);
- `Pack` позволяет указывать выравнивание полей, допускает указание значений 0, 1, 2, 4, 8, 16, 32, 64 или 128. Значение 0 означает выравнивание, принятое по умолчанию для данной платформы. Если это поле не указано, берется значение 8, за исключением неуправляемых структур, имеющих значение выравнивания 4. Разумеется, это поле работает только при типе расположения `LayoutKind.Sequential`;
- `Size` позволяет указывать полный размер объекта. Это поле позволяет неуправляемому коду определить размер объекта, если это не возможно сделать другими способами (например, структура содержит описание массива байт `byte[]`). Указываемый размер должен быть больше или равен сумме всех размеров полей объекта.

4.3.2. Атрибут *MarshalAs*

Атрибут `System.Runtime.InteropServices.MarshalAsAttribute` позволяет указать представление данных, передаваемых между управляемым и неуправляемым кодом.

Наиболее используемыми являются следующие значения этого атрибута:

- `Bool` — 4-байтовый тип `Boolean`. Соответствует типу Win32 `BOOL`. `True` означает ненулевое значение, `false` — ноль;
- `ByValArray` — массив. Для этого типа должно быть указано значение поля `SizeConst`, равное числу элементов в массиве. Дополнительно поле `ArraySubType` может указывать тип элементов массива;
- `ByValTStr` — строки конечной длины в стиле C (соответствуют описанию вида `char s[5]`). Для этого значения должны дополнительно указываться атрибут кодировки `StructLayoutAttribute.CharSet` и поле размера `SizeConst`;
- `Error` — соответствует типу Win32 `HRESULT`;
- `I1` — соответствует типу `bool` языка C. Значение `true` равно 1, значение `false` равно 0;
- `I2`, `I4`, `I8` — соответственно 2-, 4- и 8-байтовые целые числа со знаком;
- `R4`, `R8` — 4- и 8-байтовые числа с плавающей точкой;
- `U1`, `U2`, `U4`, `U8` — целые беззнаковые числа;
- `VariantBool` — аналог `VARIANT_BOOL`. Значение `true` соответствует −1, значение `false` соответствует 0.

Поле `SizeConst` должно заполняться для типов `ByValArray` и `ByValTStr` и позволяет указать размер описываемого массива.

Пример описания данных с атрибутом `MarshalAs` показан в листинге 4.4.

Листинг 4.4. Пример описания данных с атрибутом `MarshalAs`

```
// GUID
[StructLayout(LayoutKind.Sequential)]
public unsafe struct GUID
{
 public int Data1;
 public System.UInt16 Data2;
 public System.UInt16 Data3;
 // Неуправляемый код не может определить размер
 // данных по описанию byte[], поэтому указываем
 // размер явно.
 [MarshalAs(UnmanagedType.ByValArray, SizeConst=8)]
 public byte[] data4;
}
```

Для структур, использующих передачу данных по указателю, должен указываться префикс `unsafe`.

Использование структур, содержащих атрибуты `SizeConts`, сопряжено со сложностью определения их размера. Многие функции, которые мы будем использовать, требуют передачи размера структуры в качестве параметра. Однако этот размер определяется на основе описания для неуправляемого языка (C или Delphi). Несмотря на кажущееся подобие описаний, разница все же имеется. Рассмотрим описание структуры на языке C:

```
typedef struct _SP_DEVICE_INTERFACE_DETAIL_DATA_A {
    DWORD  cbSize;
    CHAR   DevicePath[0];
}SP_DEVICE_INTERFACE_DETAIL_DATA_A;
```

Очевидно, что размер этой структуры равен 5. На языке Delphi описание будет похоже и определение `sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA)` также даст 5:

```
SP_DEVICE_INTERFACE_DETAIL_DATA = packed record
 cbSize: DWORD;
 DevicePath: array [0..0] of AnsiChar;
end;
```

А вот в C# все будет сложнее. Отсутствие указателей приводит к необходимости явно указывать размер поля `DevicePath`:

```
[StructLayout(LayoutKind.Sequential, CharSet= CharSet.Ansi)]
public unsafe struct PSP_DEVICE_INTERFACE_DETAIL_DATA
{
 public int  cbSize;
 [MarshalAs(UnmanagedType.ByValTStr, SizeConst= 256)]
 public string DevicePath;
}
```

Явное указание размера приводит к тому, что размер этой структуры будет определяться как 260. Вообще говоря, это более "правильная" цифра, однако функции, использующие эту структуру, требуют размера 5, а не 260. Из-за этого в C# часто приходится указывать размер структур явно, а не с помощью определения `sizeof`.

4.4. Прямой доступ к данным

Строки, как и все объекты в .NET, являются экземплярами класса. Однако часто надо получить реальное (побайтовое) представление строки. Для этого

приходится использовать незащищенный код, как показано в листинге 4.5. В этом примере мы напрямую модифицируем содержимое строки, получив доступ к ней с помощью указателя `pfixed`.

Листинг 4.5. Пример прямого доступа к содержимому строки

```
// Преобразование строки к верхнему регистру
public static unsafe void ToUpper(string str)
{
  fixed (char *pfixed = str)
    for (char *p=pfixed; *p != 0; p++)
      *p = char.ToUpper(*p);
}
```

4.5. Обработка сообщений Windows

Позже (*см. разд. 11.3.2 и 11.8*) мы будем рассматривать функции, которые используют очередь сообщений Windows для обмена данными с устройством. Перехватить сообщения Win32 можно или с помощью интерфейса `System.Windows.Forms.IMessageFilter`, как показано в листинге 4.6, или с помощью перекрытия виртуального метода `WndProc` (для приложений типа Windows Forms), как показано в листинге 4.7. К сожалению, сообщения приходится обрабатывать по коду. Параметры сообщения доступны с помощью свойств `m.LParam` и `m.WParam`.

Листинг 4.6. Использование `IMessageFilter`

```
// класс фильтра
public class MyMessageFilter : System.Windows.Forms.IMessageFilter
{
   public bool PreFilterMessage(ref Message m)
   {
      // Нажатие левой кнопки мыши
      if (m.Msg == 513)
      {
         MessageBox.Show("WM_LBUTTONDOWN is: " + m.Msg);
         return true;
      }
```

```
            return false;
        }
    }

    // Объект фильтра
    static private MyMessageFilter msgFliter = new MyMessageFilter();

    [STAThread]
    static void Main()
    {
        // Регистрация объекта фильтра
        Application.AddMessageFilter(msgFliter);
        Application.Run(new Form1());
    }
```

Листинг 4.7. Обработка сообщений Win32

```
public class Form1 : System.Windows.Forms.Form
{
  protected override void WndProc(ref Message m)
  {
     // Обработка сообщения с кодом m.Msg
     base.WndProc (ref m);

  }
}
```

4.6. Общие сведения о WMI

Одной из базовых технологий компании Microsoft, предназначенной для централизованного управления и слежения за работой различных частей компьютерной сети под управлением Windows, является WMI (Windows Management Instrumentation).

Для представления данных используется модель CIM (Common Information Model), которая представляет собой стандартную систему именования для физических и логических компонентов компьютера, таких как логический раздел жесткого диска, экземпляр исполняемого приложения или кабель. Такая система нужна для того, чтобы любой пользователь мог обращаться ко всем элементам CIM, используя одни и те же термины для описания

этих элементов и связи с ними. CIM — объектно-ориентированная схема, поэтому для описания ее компонентов используются объектно-ориентированная терминология и правила:

- CIM содержит классы, которые представляют собой шаблоны управляемых элементов;
- объект — экземпляр класса, представляющий базовый компонент системы;
- пространство имен — набор классов для специализированной области управления;
- классы содержат свойства и методы.

Для использования классов WMI в C# требуется подключить библиотеку `System.Management.dll`. Общий сценарий вызова WMI-функций показан в листинге 4.8. Для вызова WMI с удаленной машины используется код, показанный в листинге 4.9.

Листинг 4.8. Вызов WMI-функций

```
using System;
using System.Management;

namespace WMI1
{
  class Class1
  {
    [STAThread]
    static void Main(string[] args)
    {
       ManagementScope sc =
          new  ManagementScope(@"\\.\root\cimv2", null);
       ManagementPath  ph = new  ManagementPath (<имя_ветки>);
       ManagementClass mc = new  ManagementClass(sc, ph, null);

       foreach (ManagementObject obj in mc.GetInstances())
       {
          работаем с obj.GetPropertyValue(имя_свойства);
       }
    }
  }
}
```

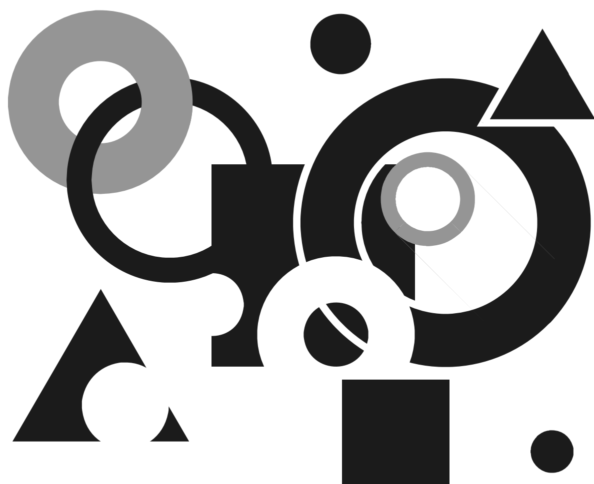
Листинг 4.9. Вызов WMI-функций на удаленной машине

```
using System;
using System.Management;

static void Main(string[] args)
{
  ConnectionOptions options = new ConnectionOptions();
  options.Username = @"domain\username";
  options.Password = "password";
  ManagementScope scope =
         new ManagementScope(@"\\machine_name\root\cimv2", options);
  scope.Connect();

  try
  {
       // работа с функциями WMI
  }
  catch (Exception e)
  {
  }
}
```

Подробную информацию об использовании WMI-функций для работы с USB-устройствами *см. в гл. 11.*

4.7. Интернет-ресурсы к этой главе

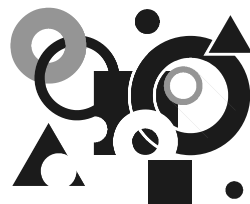
- Сценарии WMI для начинающих:
  - **http://www.osp.ru/win2000/2001/05/070.htm**;
- Справочная информация по WMI:
  - **http://msdn.microsoft.com/library/en-us/wmisdk/wmi/wmi_reference.asp**;
- Внутри WMI:
  - **http://www.citforum.ru/operating_systems/articles/wmiin.shtml**;
- Управление приложениями с помощью WMI:
  - **http://msdn.microsoft.com/library/rus/cpguide/html/cpconmanagingapplicationsusingwmi.asp**.



ЧАСТЬ II

Классы USB

Глава 5



Класс CDC

В этой главе мы рассмотрим один из наиболее интересных классов USB-устройств, называемый CDC (Communication Device Class, класс коммуникационных устройств). Спецификация CDC ориентирована на поддержку коммуникационных сервисов, таких как ISDN-модемы или виртуальные COM-порты. В нашей книге мы будем рассматривать коммуникационные устройства, использующие последовательный порт (для простоты мы будем называть их CDC-устройствами, хотя согласно полной спецификации это может показаться не совсем верным термином). Следует также отметить, что описание, приведенное в нашей книге, не является дословным переводом спецификации: мы допустим некоторые упрощения, позволяющие толковать спецификацию с практической точки зрения. Первоисточник можно найти на сайте **http://www.usb.org**.

5.1. Методы преобразования интерфейсов USB/RS-232

Появление USB постепенно приводит к вытеснению последовательного порта (RS-232). Так, например, последовательный порт уже давно отсутствует на всех современных ноутбуках. Тем не менее, многие приложения продолжают его использовать. Причин для этого много. Часто решающим фактором становится невозможность переработки программы, нежелание дополнительных трудозатрат или некоторая сложность работы с USB относительно привычного последовательного порта. В таких случаях бывает удобно использование преобразователя интерфейсов.

Основная задача преобразования интерфейса RS-232 в USB — получить со стороны программы обычный последовательный порт, позволяющий работать с интерфейсом "по старинке". Существует несколько путей решения этого вопроса.

Первый, самый простой, способ — приобретение специального переходника (рис. 5.1). Второй — использование специальных микросхем преобразования интерфейсов, например, микросхем FTDI (**http://www.ftdi.com**). И, наконец, можно использовать специальный класс USB-устройств, описываемый спецификацией CDC.

Рис. 5.1. Переходник USB-to-COM

5.2. Общие сведения об интерфейсе RS-232

Официальное название интерфейса RS-232 — интерфейс между оконечным оборудованием данных и оконечным оборудованием канала передачи данных, применяющий последовательный обмен двоичными данными (Interface Between Data Terminal Equipment and Data Circuit-Termination Equipment Employing Serial Binary Data Interchange). В этом разделе мы приведем сведения, необходимые нам для дальнейшей работы с CDC-устройствами. Подробную информацию об интерфейсе RS-232 см. в [3].

Последовательный интерфейс используется для связи двух устройств между собой. Данные в одну сторону передаются по одному проводу с помощью последовательности бит. 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 и другое периферийное оборудование;
- *DCE* (Data Communications Equipment) — аппаратура канала данных. Функция DCE состоит в обеспечении возможности передачи информа-

ции между двумя или большим числом DTE. Для этого DCE должен обеспечить соединение с DTE с одной стороны, и с каналом передачи — с другой. Роль DCE чаще всего выполняет модем, например, как показано на рис. 5.2.

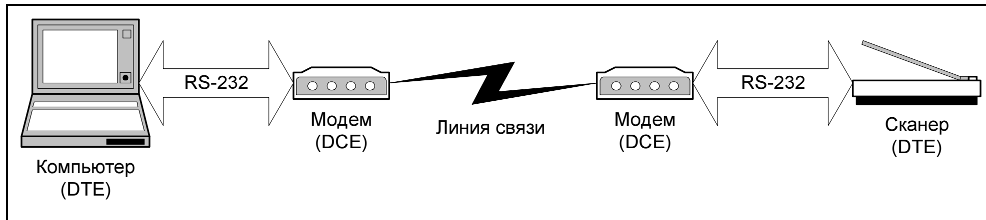


Рис. 5.2. Полная схема соединения по RS-232

DTE-устройства могут быть соединены напрямую, без DCE, с помощью *нуль-модемного кабеля*, как показано на рис. 5.3.

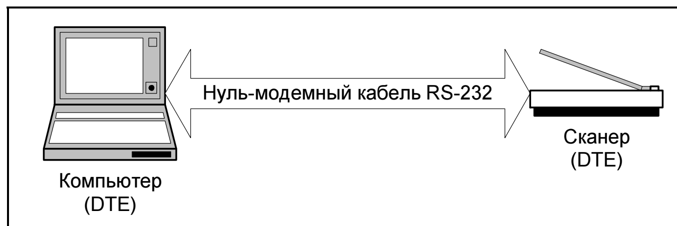


Рис. 5.3. Соединения по RS-232 нуль-модемным кабелем

5.2.1. Линии обмена

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

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

Ниже мы приведем краткое описание основных сигналов. Подробное описание сигналов можно найти в [3] или специальной литературе.

5.2.1.1. 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 др.), 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.

5.2.1.2. 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.

5.2.1.3. 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.

5.2.1.4. 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.

5.2.1.5. 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.

5.2.1.6. 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.

5.2.1.7. 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.

5.2.1.8. Обнаружение несущей (CF/DCD)

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

ли 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 сам.

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.

5.2.1.9. 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 непереданные 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.

5.3. Спецификация CDC

Спецификация CDC содержит описание коммуникационного оборудования, такого как аналоговые модемы, ISDN-адаптеры, цифровые и аналоговые телефоны, среднескоростные модемы и т. п. Спецификация не диктует, каким образом устройства должны использовать USB-интерфейс, и не перекрывает существующие стандарты.

5.3.1. Стандартные дескрипторы

Для определения типа USB-устройства используются поля дескриптора устройства (*см. разд. 1.2.3*).

Интересующее нас USB-устройство должно иметь следующие значения полей:

- `bDeviceClass` — `0x02`;
- `bDeviceSubClass` — `0x00`;
- `bDeviceProtocol` — `0x00`.

Дескриптор конфигурации имеет обычный вид (*см. разд. 1.2.3*). CDC-устройство должно определять два интерфейса (*см. разд. 1.2.3*). Первый, *коммуникационный интерфейс* (communication interface) должен иметь следующие значения полей:

- `bInterfaceClass` — `0x02`;
- `bInterfaceSubClass` — значения, указанные в табл. 5.1;
- `bInterfaceProtocol` — значения, указанные в табл. 5.2.

Второй интерфейс, *интерфейс данных* (data interface), должен иметь следующие значения полей:

- `bInterfaceClass` — `0x0A`;
- `bInterfaceSubClass` — `0x00`;
- `bInterfaceProtocol` — значения, указанные в табл. 5.3.

Таблица 5.1. *Значения полей* `bInterfaceSubClass` *коммуникационного интерфейса*

Код	**Описание подкласса**
`0x00`	Зарезервировано
`0x01`	Direct Line-устройство
`0x02`	Абстрактное устройство
`0x03`	Телефон
`0x04`	Мультиканальное устройство
`0x05`	CAPI-устройство
`0x06`	Ethernet-сетевое устройство
`0x07`	ATM-сетевое устройство
`0x08—0x7F`	Зарезервировано для будущего использования
`0x80—0xFE`	Определяется производителем

Таблица 5.2. *Значения полей* `bInterfaceProtocol` *коммуникационного интерфейса*

Код	**Описание подкласса**
`0x00`	Нет специального протокола
`0x01`	AT-команды
`0x02—0xFE`	Зарезервировано для будущего использования
`0xFF`	Протокол определяется производителем устройства

Таблица 5.3. *Значения полей* `bInterfaceProtocol` *интерфейса данных*

Код	**Описание подкласса**
`0x00`	Нет специального протокола
`0x01—0x2F`	Зарезервировано для будущего использования
`0x30`	ISDN
`0x31`	HDLC
`0x90`	Протокол со сжатием данных V42bis
`0x93`	CAPI
`0xFF`	Определяется производителем устройства

Для реализации преобразователя USB в RS232 используется *абстрактное устройство* (подкласс 0x02), возможно поддерживающее набор AT-команд (протокол 0x01). Для эмуляции последовательного порта CDC-устройство использует три конечные точки:

- конечную точку 0 (управляющую), использующую режим Interrupt IN, для передачи управляющих команд (коммуникационный интерфейс);
- две конечные точки, использующих режимы Bulk IN и Bulk OUT, для передачи данных (интерфейс данных).

5.3.2. Функциональные дескрипторы

CDC-устройство может передавать один или несколько специальных дескрипторов (функциональные дескрипторы, functional descriptor), описывающих его функциональные возможности. Функциональные дескрипторы имеют формат, показанный в табл. 5.4.

Таблица 5.4. *Структура функционального дескриптора CDC-устройства*

Смещение	Поле	Размер	Описание
0	bFunctionLength	1	Размер дескриптора
1	bDescriptorType	1	Тип дескриптора: CS_INTERFACE (значение 0x24) или CS_ENDPOINT (значение 0x25)
2	bDescriptorSubtype	1	Подтип дескриптора (см. табл. 5.5)
3	Data 0,...Data *N*	1	Данные, если они есть

Таблица 5.5. *Основные подтипы функциональных дескрипторов CDC-устройства*

Код	Описание
0x00	Заголовочный функциональный дескриптор, который обозначает начало блока функциональных дескрипторов для интерфейса (Header Functional Descriptor)
0x01	Дескриптор режима команд (Call Management Descriptor)
0x02	Дескриптор абстрактного устройства (Abstract Control Management Descriptor)
0x06	Функциональный дескриптор группирования (Union Functional Descriptor)
0x11—0xFF	Зарезервировано

5.3.2.1. Заголовочный функциональный дескриптор

Заголовочный функциональный дескриптор (header functional descriptor) обозначает начало блока функциональных дескрипторов для интерфейса. Структура этого дескриптора показана в табл. 5.6.

Таблица 5.6. *Структура заголовочного функционального дескриптора*

Смещение	Поле	Размер	Значение	Описание
0	bFunctionLength	1	0x05	Размер дескриптора
1	bDescriptorType	1	0x24	CS_INTERFACE

***Таблица 5.6** (окончание)*

Смещение	Поле	Размер	Значение	Описание
2	`bDescriptorSubtype`	1	`0x00`	Заголовочный дескриптор
3	`bcdCDC`	2	`0x10, 0x01`	Версия USB-спецификации в BCD-формате

5.3.2.2. Дескриптор режима команд

Дескриптор режима команд (call management descriptor) описывает процесс вызовов коммуникационного интерфейса. Структура этого дескриптора показана в табл. 5.7.

***Таблица 5.7.** Структура дескриптора режима команд*

Смещение	Поле	Размер	Значение	Описание
0	`bFunctionLength`	1	`0x05`	Размер дескриптора
1	`bDescriptorType`	1	`0x24`	`CS_INTERFACE`
2	`bDescriptorSubtype`	1	`0x01`	Дескриптор режима команд
3	`bmCapabilities`	1		Описывает возможности конфигурации
4	`bDataInterface`	1		Число интерфейсов в этой конфигурации (считая с нуля)

Из байта `bmCapabilities` используются только два младших бита (остальные биты зарезервированы и должны быть заполнены нулями):

- [7..2] резерв;
- [1] режим приема и передачи команд (*см. разд. 5.3.3*):
  - 0 — устройство может принимать и передавать команды только через коммуникационный интерфейс;
  - 1 — устройство может принимать и передавать команды через интерфейс данных;
- [0] если этот бит 0, то бит D1 игнорируется.

5.3.2.3. Дескриптор абстрактного устройства

Дескриптор абстрактного устройства (abstract control management descriptor) описывает команды, поддерживаемые коммуникационным интерфейсом. Структура этого дескриптора показана в табл. 5.8.

Таблица 5.8. *Структура дескриптора абстрактного устройства*

Смещение	Поле	Размер	Значение	Описание
0	`bFunctionLength`	1	`0x04`	Размер дескриптора
1	`bDescriptorType`	1	`0x24`	`CS_INTERFACE`
2	`bDescriptorSubtype`	1	`0x02`	Дескриптор абстрактного устройства
3	`bmCapabilities`	1		Битовая маска поддерживаемых команд

Байт `bmCapabilities` состоит из следующих битов:

- [7..4] резерв, должны быть нулями;
- [3] устройство поддерживает нотификацию `Network_Connection` (не рассматривается в этой книге);
- [2] устройство поддерживает запрос `Send_Break`;
- [1] устройство поддерживает запросы `Set_Line_Coding`, `Set_Control_Line_State`, `Get_Line_Coding` и нотификацию `Serial_State`;
- [0] устройство поддерживает запросы `Set_Comm_Feature`, `Clear_Comm_Feature` и `Get_Comm_Feature`.

5.3.2.4. Дескриптор группирования

Дескриптор группирования (union interface functional descriptor) описывает отношения интерфейсов в группе. Один из интерфейсов описывается как *главный интерфейс* (master или controlling interface), остальные интерфейсы описываются как *зависимые* (slave interface). Запросы и нотификации, проходящие через главный интерфейс, применяются ко всей группе интерфейсов. Структура этого дескриптора показана в табл. 5.9 (все интерфейсы нумеруются с нуля).

Таблица 5.9. *Структура дескриптора абстрактного устройства*

Смещение	Поле	Размер	Значение	Описание
0	bFunctionLength	1		Размер дескриптора
1	bDescriptorType	1	0x24	CS_INTERFACE
2	bDescriptorSubtype	1	0x06	Дескриптор группирования
3	bMasterInterface	1		Номер главного интерфейса
4	bSlaveInterface0	1		Первый зависимый интерфейс
...	...	...	...	...
N+3	bSlaveInterface*N*-1	1		Интерфейс *N*

5.3.3. Специальные запросы

Кроме обычных запросов, абстрактное устройство реализует специальные запросы, основные из которых представлены в табл. 5.10.

Таблица 5.10. *Основные специальные запросы абстрактного устройства*

Запрос	Код	Описание
SEND_ENCAPSULATED_COMMAND	0x00	Посылка команды
GET_ENCAPSULATED_RESPONSE	0x01	Прием команды
SET_LINE_CODING	0x20	Конфигурирование линии
GET_LINE_CODING	0x21	Возвращает текущее значение конфигурации линии
SET_CONTROL_LINE_STATE	0x22	Посылка сигнала вызова устройства (протокол RS-232)
SEND_BREAK	0x23	Посылка специального сигнала разрыва линии (протокол RS-232)

5.3.3.1. Запрос *SET_LINE_CODING*

Запрос SET_LINE_CODING (код 0x20) позволяет хосту установить параметры передачи данных, которые требуются некоторым приложениям для корректной работы с портом. Формат данных запроса совпадает с данными запроса GET_LINE_CODING.

5.3.3.2. Запрос *GET_LINE_CODING*

Запрос GET_LINE_CODING (код 0x21) позволяет хосту определить текущие настройки линии передачи. Формат данных этого запроса показан в табл. 5.11.

Таблица 5.11. *Формат данных запроса* GET_LINE_CODING

Смещение	Поле	Размер	Описание
0	dwDTERate	4	Скорость передачи, бит/с
4	bCharFormat	1	Число стоп-бит: • 0 — 1 стоп-бит • 1 — 1,5 стоп-бита • 2 — 2 стоп-бита
5	bParityType	1	Четность: • 0 – Нет контроля четности (None) • 1 – Нечет (Odd) • 2 – Четность (Even) • 3 – Постоянная 1 (Mark) • 4 – Постоянный 0 (Space)
6	bDataBits	1	Число бит данных (5, 6, 7, 8 или 16)

Примечание

Обычный последовательный порт не позволяет устанавливать режим "16 бит данных".

5.3.3.3. Запрос *SET_CONTROL_LINE_STATE*

Запрос SET_CONTROL_LINE_STATE (код 0x22) устанавливает состояние линии передачи. В запросе передается два байта данных, состоящих из следующих битов:

- [15..2] резерв, должны быть нулями;
- [1] состояние сигнала RTS;
- [0] состояние сигнала DTR.

5.3.3.4. Запрос *SEND_BREAK*

Запрос `SEND_BREAK` (код `0x23`) замораживает передачу данных. В отличие от стандартных функций последовательного порта (*см. разд. 15.3*) этот запрос имеет расширенную функциональность. Поле `wValue` этого запроса (*см. разд. 1.2.1*) указывает величину тайм-аута в миллисекундах. Если значение тайм-аута равно `0xFFFF`, то передача данных останавливается до получения повторного запроса `SEND_BREAK` с нулевым значением `wValue`.

5.3.4. Нотификации

Для уведомления об изменениях в состоянии и осуществления обратной связи с хостом устройство может использовать специальные нотификации. Возможность посылки той или иной нотификации определяется соответствующими битами в функциональных дескрипторах устройства (*см. разд. 5.3.2*). Мы приведем описание двух, наиболее интересных для нас, нотификаций.

5.3.4.1. Нотификация *RING_DETECT*

Нотификация `RING_DETECT` уведомляет хост о появлении сигнала вызова. Нотификация имеет код `0x09`, поля содержат следующие значения:

- `bmRequestType — 0xA1`;
- `bRequest — RING_DETECT (0x09)`;
- `wValue — 0`;
- `wIndex` — номер интерфейса;
- `wLength — 0`;
- данных нет.

5.3.4.2. Нотификация *SERIAL_STATE*

Нотификация `SERIAL_STATE` используется для уведомления об изменении состояния последовательного интерфейса. Нотификация имеет код `0x20`, поля содержат следующие значения:

- `bmRequestType — 0xA1`;
- `bRequest — SERIAL_STATE (0x20)`;
- `wValue — 0`;
- `wIndex` — номер интерфейса;
- `wLength — 2`.

Поле данных `Data` содержит двухбайтовую маску состояния последовательного интерфейса:

- [15..7] — зарезервировано;
- [D6] — `bOverRun`, флаг переполнения (потеря символа);
- [D5] — `bParity`, ошибка четности принятого символа;
- [D4] — `bFraming`, ошибка кадра (отсутствует стоп-бит);
- [D3] — `bRingSignal`, сигнал вызова;
- [D2] — `bBreak`, установлен сигнал прерывания;
- [D1] — `bTxCarrier`, состояние линии `DSR`;
- [D0] — `bRxCarrier`, состояние линии `DCD`.

Поддержка этой нотификации определяется битом 1 в дескрипторе абстрактного устройства (*см. разд. 5.3.2.3*).

5.4. Поддержка CDC в Windows

Класс CDC поддерживается всеми версиями Windows, начиная с Windows 98 SE до Windows XP. Для установки драйвера устройства требуется специальный INF-файл (*см. разд. 9.3 и гл. 13*). После установки драйвера в системе появляется виртуальный последовательный порт. Для работы с этим последовательным портом используются стандартные функции Windows.

5.4.1. Обзор функций Windows для работы с последовательными портами

Полное описание функций Windows приводится в справочной части книги (*см. гл. 14*), а здесь мы рассмотрим основные принципы их использования.

5.4.1.1. Основные операции с портом

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`, `WriteFile`. В Windows NT/2000/XP можно использовать асинхронные функции `ReadFileEx` и `WriteFileEx`. Кроме файловых функций, Windows предоставляет специфические для коммуникационных портов специальные функции.

Для использования порта необходимо получить его идентификатор — *дескриптор порта*, с помощью которого будут происходить все остальные обращения к порту. После использования необходимо 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.

5.4.1.2. Функции настройки порта

В отличие от файловых операций, перед началом работы с портом необходимо настроить параметры обмена: скорость обмена, параметры четности, число стоп-бит, тайм-ауты и т. д. 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. 15.*

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 специального формата. 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 DLL-библиотеке содержится функция `CommConfigDialog`, отображающая стандартный диалог 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 диалоге, 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 диалога 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.

5.4.1.3. 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` (*см. гл. 15*). 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.

Специальная функция `SetCommBreak` позволяет приостанавливать передачу данных. Вызов `SetCommBreak` замораживает передачу данных для указанного порта до тех пор, пока не будет вызвана `ClearCommBreak`.

5.4.1.4. Получение состояния линий модема

Для получения состояния индикаторов модема в Windows используется функция `GetCommModemStatus`. *Подробное описание этой функции можно найти в гл. 15.*

5.4.1.5. Работа с CDC на платформе .NET

Библиотека .NET FrameWork не содержит классов для работы с последовательными портами. Для обмена данными с CDC-устройством можно воспользоваться двумя способами: либо использовать специальный ActiveX-компонент MSCOMM, либо использовать вызовы Win32-функций. Более подробно мы рассмотрим эти способы в практической части книги.

5.4.2. Соответствие функций Windows и USB-запросов

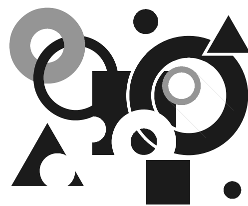
При обнаружении нового устройства (*см. разд. 11.8*) Windows производит следующую последовательность запросов:

- получение дескриптора устройства (`Get_Descriptor_Device`);
- задание адреса устройства (`Set_Address`);

- получение дескрипторов устройства (`Get_Descriptor_Device`);
- задание номера конфигурации (`Set_Configuration`);
- получение настроек линии связи (`Get_Line_Coding`);
- установка состояния линии связи (`Set_Control_Line_State`).

1. При открытии и закрытии порта CDC-устройства хост передает запрос `Set_Control_Line_State`, устанавливая соответствующее состояние линии передачи. Для установки параметров порта (при вызове функции `SetCommState`) используются запросы получения и установки настроек линии связи (`Get_Line_Coding`, `Set_Line_Coding`) и запрос установки состояния линий (`Set_Control_Line_State`).

При передаче данных кроме самих данных хост передает запрос `Set_Control_Line_State`.4При вызове `SetCommBreak` хост передает запрос `SEND_BREAK` с параметром `wValue`, равным `0xFFFF`, замораживая, таким образом, передачу данных до вызова `ClearCommBreak`, который передает запрос `SEND_BREAK` с параметром `wValue`, равным `0x0000`.



Глава 6

Класс HID

Согласно спецификации, HID-устройство (Human Interface Device) — это устройство связи с пользователем:

- ❒ клавиатуры и указатели, например, мышь, трекбол, джойстик;
- ❒ устройства контроля, такие как кнопки управления, переключатели, задвижки;
- ❒ устройства контроля в телефонах, видеомагнитофонах и игровых приставках, например, рулевое управление, игровые педали;
- ❒ устройства, не требующие взаимодействия с человеком, но передающие данные в HID-формате, например, считыватели штрихкода, термометры, вольтметры.

Вообще говоря, класс HID позволяет создать двухсторонний низкоскоростной обмен с любым устройством, даже не попадающим под жесткое определение устройства ввода. Поскольку Windows 98/2000/XP имеют встроенный драйвер для HID-устройств, то необходимость его трудоемкой разработки отпадает. С точки зрения разработчика программы самого HID-устройства, необходимо поддерживать ряд структур, описывающих HID-интерфейс, а также обеспечивать обмен по каналу данных прерываний. Основным ограничением является скорость обмена. Максимально достигаемая скорость составляет 64 Кбит/с, что значительно меньше, чем полная скорость USB — 12 Мбит/с, хотя для многих приложений, например, управления и индикации, вполне достаточно.

6.1. Спецификация HID-устройств

Разработка спецификации HID-устройств была основана на следующих требованиях:

- ❒ максимально компактный программный код, требуемый со стороны устройства;

- возможность программного обеспечения пропускать незнакомую информацию, поступающую от устройства;
- протокол должен быть наращиваемым и устойчивым;
- устройство должно само описывать свои свойства, позволяя создавать базовое (классовое) программное обеспечение.

HID-устройством может быть любое устройство, способное функционировать согласно правилам, определенным спецификацией:

- полноскоростное HID-устройство может передавать вплоть до 60 000 байт/с, т. е. 64 байта в каждом кадре 1 мс *(см. разд. 1.1.1)*, низкоскоростное — 800 байт/с, т. е. 8 байтов каждые 10 мс;
- HID-устройство может устанавливать частоту своего опроса для выяснения, имеет ли оно новые данные для пересылки;
- весь обмен с HID-устройством происходит с помощью определенной структуры, которая называется *репортом* (report). Один репорт может содержать до 65 535 байтов данных. Микропрограмма HID-устройства должна содержать *дескриптор репорта* (report descriptor), который описывает структуру данных репорта. Репорт имеет достаточно гибкую структуру для описания любого типа устройства и формата передачи данных;
- HID-устройство должно иметь конечную точку типа Interrupt IN *(см. разд. 1.1.12)* для выдачи данных в хост. Дополнительно HID-устройство может иметь конечную точку типа Interrupt OUT для получения периодических данных от хоста;
- HID-устройство должно содержать дескриптор класса и один или более дескрипторов репорта;
- HID-устройство должно поддерживать специфический для класса управляющий запрос `Get_Report`, а также опционально поддерживать дополнительный запрос `Set_Report`. Эти запросы мы рассмотрим далее;
- для передачи данных в хост HID-устройство должно положить данные репорта в буфер соответствующей конечной точки и разрешить передачу. Для получения данных от хоста HID-устройство должно разрешить соответствующую конечную точку, а затем, после прихода пакета, забрать данные из буфера.

HID-устройство должно иметь один или более дескрипторов репорта, которые запрашиваются только после того, как хост определил, что подключенное USB-устройство относится к классу HID.

Обмен между хостом и HID-устройством может производиться с помощью трех видов репортов:

- Input и Output (входные и выходные репорты) используются для передачи и приема периодических данных, например, нажатий клавиш;

- Feature (специальные репорты) используются там, где очень важно время доставки, например, для установки различных параметров устройства и его инициализации.

Обмен данными с хостом (точнее, с драйвером) производится либо по основному каналу сообщений (канал нулевой конечной точки), либо по каналу прерываний (рис. 6.1).

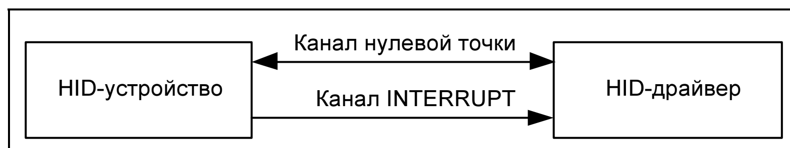


Рис. 6.1. Каналы обмена HID-устройства и драйвера

Канал нулевой точки используется для следующих операций:

- приема и передачи управляющих посылок;
- передачи данных с помощью запроса `Get_Report` (*см. ниже*);
- приема данных от хоста.

Канал прерываний используется для передачи асинхронных данных: данные с конечной точки читаются только в том случае, если HID-устройство подтверждает наличие новых данных и необходимость их передачи.

Спецификация HID определяет два типа HID-устройств: устройства, участвующие в начальной загрузке и не участвующие. Первый тип называется *загрузочные устройства* (boot device). К загрузочным устройствам относятся, например, мышь и клавиатура, работа которых начинается с самого начала включения компьютера. Загрузочные устройства должны поддерживать специальные запросы (см. табл. 6.6).

6.2. Порядок обмена данными с HID-устройством

Когда хост запрашивает входной репорт, HID-устройство выдает пакет данных с помощью передачи по прерыванию (т. е. передачи типа INTERRUPT, *см. разд. 1.1.10*). Периодичность генерации таких запросов указывается в дескрипторе конечной точки.

При генерации выходных репортов хост посылает данные в устройство, используя *управляющие посылки* (control transfers) или *передачу по прерываниям* (interrupt transfers). Возможность проводить передачи по прерыванию в HID-

устройство доступна начиная с Windows 98 SE, а более ранние версии Windows 98 будут использовать для выходных репортов управляющие передачи. Если HID-устройство не имеет конечной точки с типом Interrupt Output, то драйвер операционной системы будет использовать управляющие посылки.

Специальные репорты (т. е. репорты типа Feature) имеют направление передачи данных как от хоста к HID-устройству, так и обратно. Для них всегда используются управляющие посылки. Для того чтобы послать репорт этого типа, хост инициирует запрос `Set_Report`, предшествующий пакету данных, а далее, в фазе статуса, хост принимает от HID-устройства подтверждение об успешном либо неуспешном принятии данных. Для того чтобы получить специальный репорт, хост инициирует запрос `Get_Report`, HID-устройство при этом отвечает пакетом данных, а в фазе статуса хост возвращает в HID-устройство информацию об успешно проведенной транзакции. Еще одно преимущество специальных репортов — это возможность задавать каждому репорту его номер (Report ID). При этом у программиста появляется возможность мультиплексировать запросы, если существует необходимость создания интерфейса передачи команд управления и данных через нулевую конечную точку (*см. разд. 10.9*).

6.3. Установка драйвера HID-устройства

Для установки драйвера HID-устройства Менеджер устройств операционной системы использует INF-файлы (*см. гл. 13*). Может использоваться встроенный в операционную систему INF-файл (hiddev.inf для Windows 98 или input.inf для Windows 2000). Альтернативно программист может использовать свои собственные INF-файлы, в которых будет прописана необходимая информация. Преимущество этого подхода состоит в отображении понятного названия в окне Менеджера устройств вместо общего термина "Стандартное устройство".

6.4. Идентификация HID-устройства

HID-устройство идентифицируется кодом класса 3 в дескрипторе интерфейса и имеет два специфических дескриптора: HID-дескриптор и дескриптор репорта (на самом деле существует еще физический *дескриптор устройства* (physical descriptor), но мы его рассматривать не будем). HID-дескриптор включается в список дескрипторов при запросе конфигурации. Обмен репортами возможен только после того, как драйвер определит тип данного HID-устройства и какие интерфейсы оно поддерживает.

6.4.1. Идентификация загрузочных устройств

Идентификация типа устройства производится по полю `bInterfaceSubClass` в дескрипторе интерфейса:

- `0` — подкласс не указан (незагрузочный интерфейс);
- `1` — загрузочный интерфейс;
- `2—255` — зарезервировано.

Таким образом, HID-устройство может иметь одновременно и загрузочный интерфейс, и обычный. Например, USB-клавиатура может работать при загрузке в стандартном режиме (данные обрабатывает BIOS), а при загрузке драйвера добавлять специальные клавиши.

Если HID-устройство определено как загрузочное, то поле `bInterfaceProtocol` в дескрипторе интерфейса обозначает поддерживаемый стандартный протокол:

- `0` — протокол определяется пользовательским HID-репортом;
- `1` — клавиатура;
- `2` — мышь;
- `3—255` — зарезервировано.

Для незагрузочных HID-устройств поле `bInterfaceProtocol` должно быть равно 0.

Примечание

Если HID-устройство определено как клавиатура или мышь, Windows позволяет открывать его с помощью функции `CreateFile`, запрашивать дескрипторы устройства и конечных точек, но чтение данных с помощью функции `ReadFile` блокируется. Единственная возможность обмена данными — функции `Get_Feature` и `Set_Feature`, конечно, при условии, что HID-устройство поддерживает эти интерфейсы.

6.4.2. Дескриптор конфигурации HID-устройства

По запросу `Get_Descriptor(Configuration)` HID-устройство должно передавать хосту дескриптор конфигурации, все дескрипторы интерфейсов, все дескрипторы конечных точек и все HID-дескрипторы. Ни *строковые дескрипторы* (string descriptor), ни дескрипторы репортов не должны включаться в этот список. HID-дескриптор должен располагаться между дескриптором интерфейса и дескрипторами конечных точек:

```
Configuration descriptor
    Interface descriptor (specifying HID Class)
```

```
        HID descriptor (associated with above Interface)
            Endpoint descriptor (for HID Interrupt Endpoint)
```

Дескрипторы `HID` и `REPORT` возвращаются по запросу `Get_Descriptor` со специальными кодами запроса (*см. ниже*). Важно понимать, что дескриптор конфигурации относится к самому HID-устройству, дескриптор интерфейса и HID — к интерфейсу, а дескриптор конечной точки — к конечной точке. Это довольно очевидное утверждение оказывается важным при обработке запросов на получение дескрипторов, т. к. поле `bmRequestType` имеет разное значение для каждого из типов. Именно по этой причине в списке объединенных кодов запросов (см. листинг 1.5) присутствуют три разных идентификатора запросов `Get_Descriptor` (*см. разд. 1.2.2*):

```
#define GET_DESCRIPTOR_DEVICE  0x8006
#define GET_DESCRIPTOR_INTERF  0x8106
#define GET_DESCRIPTOR_ENDPNT  0x8206
```

6.4.3. HID-дескриптор

Формат HID-дескриптора показан в табл. 6.1, а его описание на языках C и Pascal — в листинге 6.1.

Таблица 6.1. *Структура HID-дескриптора*

Смещение	Поле	Размер	Описание
0	`bLength`	1	Размер дескриптора в байтах
1	`bDescriptorType`	1	Тип дескриптора (`$21`)
2	`bcdHID`	2	Версия HID
4	`bCountryCode`	1	Числовой код страны для локализированных устройств
5	`bNumDescriptor`	1	Число дескрипторов репортов
6	`bReportType`	1	Номер дескриптора репорта, используемый при вызове `Set_Descriptor`
7	`wReportLength`	2	Размер дескриптора репорта
9	`bReportType`	1	Номер дополнительного дескриптора
10	`wReportLength`	2	Размер дополнительного дескриптора

Листинг 6.1. HID-дескриптор

```
// Описание на языке C
typedef struct _USB_HID_DESCRIPTOR
{
  UCHAR   bLength;
  UCHAR   bDescriptorType;
  USHORT  bcdHID;
  UCHAR   bCountryCode;
  UCHAR   bNumDescriptors;
  // далее следуют один или несколько дескрипторов
  struct _HID_DESCRIPTOR_DESC_LIST {
  {
   UCHAR   bReportType;
   USHORT  wReportLength;
 } DescriptorList[1];
} USB_HID_DESCRIPTOR;
// Описание на языке Pascal
Type
 THidDescriptorList = packed record
  bReportType   : BYTE;
  wReportLength : WORD;
 End;
 TUsbHidDescriptor = packed record
  bLength         : BYTE;
  bDescriptorType : BYTE;
  bcdHID          : WORD;
  bCountryCode    : BYTE;
  bNumDescriptors : BYTE;
  DescriptorList  : Array of THidDescriptorList;
 End;
// Описание на языке C для микропроцессора
struct usb_st_hid_descriptor
{
  byte   bLength;
  byte   bDescriptorType;
```

```
  uint16 bscHID;
  byte   bCountryCode;
  byte   bNumDescriptors;
  byte   bRDescriptorType;
  uint16 wDescriptorLength;
};
```

Первые три поля HID-дескриптора содержат стандартные значения:

- `bLength` — `9`;
- `bDescriptorType` — `$21`;
- `bcdHID` — `$100`.

Поле `bCountryCode` содержит код языка, если HID-устройство локализовано, и 0 — если нет. Клавиатуры могут использовать это поля для передачи языка клавиш. HID-дескриптор возвращается по запросу `Get_Descriptor($21)`.

6.4.4. Дескриптор репорта

Дескриптор репорта (тип `REPORT`) не похож на остальные дескрипторы. Он имеет сложную табличную структуру. Длина дескриптора зависит от числа и типа данных, которые он передает.

Содержимое дескриптора репорта складывается из частей, передающих информацию о HID-устройстве. Первая часть каждого слагаемого содержит три поля: тип, тег и размер. Такая структура позволяет, с одной стороны, идентифицировать любую составляющую репорта, а с другой, пропустить неизвестные части, сразу переместившись к следующей составляющей. Дескриптор репорта имеет код `$22` и, соответственно, возвращается по запросу `GetDescriptor($22)`.

6.5. Структура дескриптора репорта

Дескриптор репорта определяет структуру данных, передаваемых от HID-устройства к хосту и обратно. HID-дескриптор состоит из *элементов* (items), каждый из которых несет определенную информацию о HID-устройстве. Элемент может содержать байты данных. Размер данных и структура элемента зависят от *базового типа* (fundamental type) элемента.

6.5.1. Элементы репорта

Существуют два основных базовых типа: *короткие элементы* (short item) и *длинные элементы* (long item).

6.5.1.1. Элементы короткого типа

Элементы короткого типа имеют однобайтный префикс, содержащий тип, тег и размер (рис. 6.2):

- [7:4] — поле тега (`bTag`, код функции элемента);
- [3:2] — тип (`bType`, *см. разд. 6.5.2*):
  - 00 — основной (main);
  - 01 — глобальный (global);
  - 10 — локальный (local);
- [0:1] — число байт данных в элементе (`bSize`):
  - 00 — нет байт данных;
  - 01 — 1 байт;
  - 10 — 2 байта;
  - 11 — 4 байта.

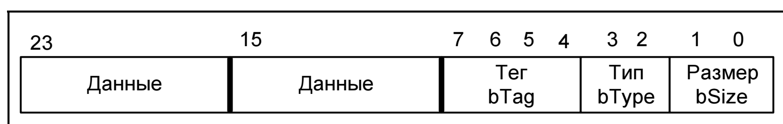


Рис. 6.2. Структура элемента дескриптора репорта

6.5.1.2. Элементы длинного типа

Элементы длинного типа имеют однобайтный префикс `0xFE`, поле `bSize` всегда равно 2, а длина данных указывается в следующем после префикса байте (рис. 6.3).

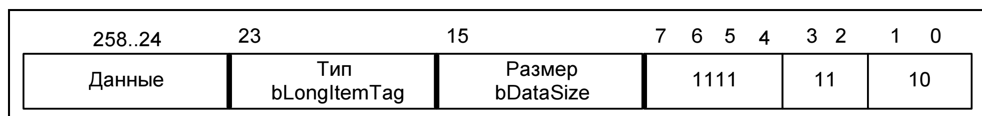


Рис. 6.3. Структура длинного элемента дескриптора репорта

6.5.2. Типы элементов репорта

Спецификация определяет три типа элементов:

- *основные элементы* (main items) — определяют группу данных в дескрипторе;
- *глобальные элементы* (global items) — определяют данные репорта;
- *локальные элементы* (local items) — определяют характеристики конкретных данных.

6.5.2.1. Основные элементы

Основные элементы определяют или группируют элементы в дескрипторе репорта. Существуют пять элементов основного типа:

- Input, Output и Feature — определяют поля соответствующих репортов;
- Collection и End Collection — не определяют поля, но объединяют группу связанных полей внутри репорта.

Рис. 6.4. Биты данных элементов Input, Output и Feature

Биты поля данных для элементов Input, Output и Feature показаны в табл. 6.2 и на рис. 6.4.

Таблица 6.2. *Биты данных элементов Input, Output и Feature*

Бит	Если бит равен 0	Если бит равен 1
0	Data	Constant
1	Array	Variable
2	Absolute	Relative
3	No wrap	Wrap
4	Linear	Non-linear
5	Preferred state	No preferred state
6	No null position	Null state
7	Non-volatile	Volatile
8	Bit Field	Buffered Bytes
9–31	Зарезервировано	

Data/Constant

Data означает, что данные, относящиеся к этой группе, могут изменяться. Constant означает, что данные изменяться не могут (только для чтения).

Array/Variable

Этот бит управляет представлением данных. Например, если клавиатура имеет 8 клавиш, то установка режима Variable будет означать, что клавиатура содержит по одному биту на каждую клавишу. В дескрипторе репорта нужно будет указать, что *размер элемента данных* (report size) равен 1, а *число элементов данных* (report count) равно 8. Установка режима Array будет означать, что каждая клавиша имеет индекс, который передается в репорте, если клавиша нажата. Для восьми клавиш индекс будет кодироваться тремя битами, поэтому поле `Report Size` должно быть равно 3, а `Report Count` должно быть равно числу клавиш, которые разрешено нажимать одновременно.

Absolute/Relative

Absolute означает, что значения, переданные в репорте, берутся "как есть". Relative означает, что данные передаются как относительное смещение от предыдущего переданного пакета. Например, джойстик передает абсолютные данные, а мышь — относительные.

No Wrap/Wrap

Wrap означает, что значение "перескочит" через границу, если оно будет продолжать увеличиваться после достижения максимума или уменьшаться после достижения минимума. Этот бит не применяется для типа данных Array.

Linear/Non-linear

Linear означает, что измеренные данные и переданные линейно зависимы. Этот бит не применяется для типа данных Array.

Preferred/No Preferred State

Preferred означает, что HID-устройство имеет специальное состояние, возвращаемое, когда пользователь не взаимодействует с ним. Например, такое состояние имеет кратковременное нажатие клавиш, и клавиатура возвращает пакет "нет нажатых клавиш", тогда как переключатель не имеет такого состояния, всегда возвращая свое текущее состояние. Этот бит не применяется для типа данных Array.

No Null Position/Null State

Null State означает, что HID-устройство может передавать нулевое состояние, которое может не интерпретироваться корректно, например, не входить в рабочий диапазон данных. No Null Position означает, что HID-устройство не имеет такого состояния и всегда передает корректные и интерпретируемые данные. Этот бит не применяется для типа данных Array.

Non-volatile/Volatile

Этот бит применяется только для Output- и Feature-репортов. Volatile означает, что HID-устройство может изменить значение данных само, без взаимодействия с хостом. Non-volatile означает, что HID-устройство может изменять значение данных только по требованию хоста. Этот бит не применяется для типа данных Array.

Bit Field/Buffered Bytes

Bit Field означает, что каждый бит или группа бит в байте могут представлять отдельную часть данных. Buffered Bytes означает, что данные содержатся в одном или более байтах. Значение элемента размера данных (Report Size) для таких данных должно быть равно 8. Этот бит не применяется для типа данных Array.

Примечание

Более подробное описание можно найти в спецификации HID.

Следует учитывать, что поле данных урезается до последнего ненулевого байта, например:

```
81 00 — INPUT (Data,Ary,Abs)
81 02 — INPUT (Data,Var,Abs)
82 02 01 — INPUT (Data,Var,Abs,Buf)
```

Элементы Collection и End Collection могут использоваться в репортах любого типа для объединения связанных элементов в группы. Спецификация определяет три типа групп: *прикладные* (application), *физические* (physical) и *логические* (logical). Производители могут определять свои группы.

В табл. 6.3 показаны значения поля данных для этих элементов.

Таблица 6.3. *Значение поля данных для элементов Collection и End Collection*

Элемент и его префикс	Поле данных	Описание
Collection (0xA1)	0x00	Физическая
	0x01	Прикладная
	0x02	Логическая
	0x03—0x7F	Резерв
	0x80—0xFF	Определяется производителем
End Collection (0xC0)	Нет	Закрывает группу

Прикладная группа объединяет элементы, имеющие одно функциональное назначение. Например, загрузочное устройство может определять две прикладные группы элементов — группу для работы на стадии загрузки и обычную группу. В отдельные группы могут быть объединены Input- и Feature-репорты.

Физическая группа объединяет элементы, представляющие данные геометрической точки (например, позицию указателя мыши, см. листинг 8.3). Логическая группа объединяет элементы различного типа в единое целое. Например, могут быть объединены элементы "буфер данных" и "число байт" в этом буфере.

Более детальное описание групп можно найти в спецификации HID, а нам важно знать, что любое описание репорта должно находиться внутри прикладной группы, т. е. внутри элементов Collection (Application) и End Collection.

6.5.2.2. Глобальные элементы

Глобальные элементы репорта описывают характеристики данных в этом репорте, такие как максимум и минимум величин, размеры и число репор-

тов. Глобальные элементы действуют в пределах всего репорта до следующего глобального элемента. Спецификация определяет 12 глобальных элементов (табл. 6.4).

Таблица 6.4. *Глобальные элементы дескриптора репорта*

Элемент	Код*	Описание
`Usage Page`	`000001`*nn*	Назначение данных
`Logical Minimum`	`000101`*nn*	Минимум в логических единицах
`Logical Maximum`	`001001`*nn*	Максимум в логических единицах
`Physical Minimum`	`001101`*nn*	Логический минимум в физических единицах
`Physical Maximum`	`010001`*nn*	Логический максимум в физических единицах
`Unit exponent`	`010101`*nn*	Десятичный порядок единиц
`Unit`	`011001`*nn*	Единицы
`Report Size`	`011101`*nn*	Размер элемента в битах
`Report ID`	`100001`*nn*	Идентификатор репорта
`Report Count`	`100101`*nn*	Число полей данных в репорте
`Push`	`101001`*nn*	Сохраняет копию глобальных настроек в стеке
`Pop`	`101101`*nn*	Восстанавливает глобальные настройки из стека
Резерв	`110001`*nn*—`111101`*nn*	Резерв

* *nn* обозначает число байтов, следующих за префиксом.

Наиболее интересными являются элементы `Report Count` и `Report Size`, определяющие число байтов в репорте и число битов, используемых для каждого элемента данных. Например:

- два поля по 8 бит определяются значениями `Report Count` — 2, `Report Size` — 8;
- десять полей по 4 бита определяются значениями `Report Count` — 10, `Report Size` — 4;
- одно поле размером 16 бит определяется значениями `Report Count` — 1, `Report Size` — 16.

Каждый репорт может иметь идентификатор, задаваемый полем `Report ID` (*см. разд. 10.9*). Отличие логического минимума и максимума от физического проще всего понять на примере. Пусть устройство передает значение температуры от 0 до 200 °C с сеткой в 2 °C. Физический и логический минимум будет равен 0, а физический максимум — 200. Однако логический максимум будет равен 100.

Элемент `Unit exponent` позволяет определить десятичный порядок значений, описанных в репорте. Допустимы значения от −8 до +7 (см. табл. 7.5). Значение 0 (обозначающее $10^0 = 1$) оставляет данные без изменений. Например, число 1234 и `Unit exponent = 0x0E` преобразуется в 12,34. В общем случае для вычисления применяется следующая формула преобразования:

```
Value = Value_L × ((PhMax - PhMin) × (10^UExp)) / (LogMax - LogMin)
```

где:

- `Value_L` — значение в логических единицах;
- `PhMax`, `PhMin` — физический максимум и минимум;
- `UExp` — экспонента;
- `LogMax`, `LogMin` — логический максимум и минимум.

Таблица 6.5. *Значения элемента* `Unit exponent`

Экспонента	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Код	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
Экспонента	**–8**	**–7**	**–6**	**–5**	**–4**	**–3**	**–2**	**–1**
Код	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F

Элементы `Push` и `Pop` позволяют сохранить и восстановить глобальные настройки. Часто эти элементы позволяют значительно сократить размер дескриптора за счет сокращения повторных описаний.

Элемент `Usage Page` определяет назначение HID-устройства. Спецификация определяет довольно много назначений (листинг 6.2). Для HID-устройств, не попадающих в указанные типы, определяется тип `Vendor Defined Page`. В зависимости от выбранного назначения интерпретируется назначение полей данных, задаваемое элементом `Usage`.

Листинг 6.2. Значения поля `Usage Page`

```
USAGE_PAGE (Generic Desktop)          05 01
USAGE_PAGE (Simulation Controls)      05 02
```

```
USAGE_PAGE (VR Controls)                05 03
USAGE_PAGE (Sports Controls)            05 04
USAGE_PAGE (Gaming Controls)            05 05
USAGE_PAGE (Keyboard)                   05 07
USAGE_PAGE (LEDs)                       05 08
USAGE_PAGE (Button)                     05 09
USAGE_PAGE (Ordinals)                   05 0A
USAGE_PAGE (Telephony Devices)          05 0B
USAGE_PAGE (Consumer Devices)           05 0C
USAGE_PAGE (Digitizers)                 05 0D
USAGE_PAGE (Unicode)                    05 10
USAGE_PAGE (Alphnumeric Display)        05 14
USAGE_PAGE (Monitor)                    05 80
USAGE_PAGE (Monitor Enumerated Values)  05 81
USAGE_PAGE (Monitor Enumerated Values)  05 81
USAGE_PAGE (VESA Virtual Controls)      05 82
USAGE_PAGE (VESA Command)               05 83
USAGE_PAGE (Power Device)               05 84
USAGE_PAGE (Battery System)             05 85
USAGE_PAGE (Vendor Defined Page 1)      06 00 FF
```

6.5.2.3. Локальные элементы

Логические элементы характеризуют переключатели, кнопки и другие элементы, состояние которых передается с помощью репорта.

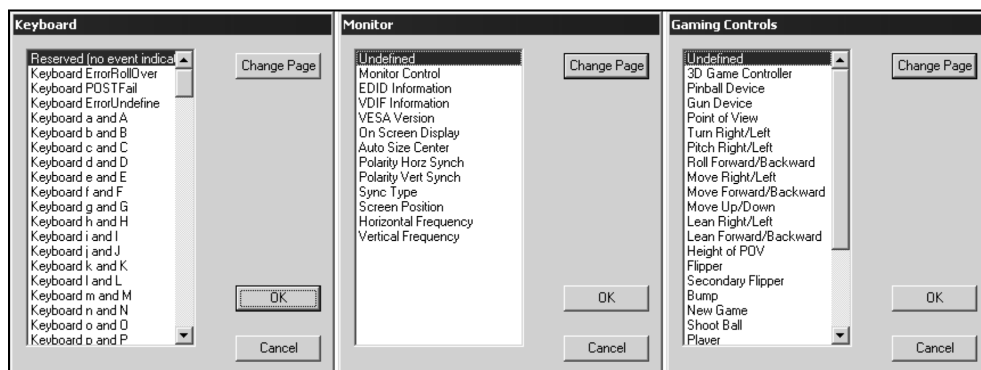


Рис. 6.5. Наборы элементов `Usage` для разных `Usage Page`

Локальные элементы применяются ко всем элементам внутри основных элементов, пока не найдется новое значение, но действие локальных элементов не переходит границу основного элемента.

Наиболее часто используемым является элемент `Usage` (код `000010`*nn*), обозначающий метод использования данных репорта. Интерпретация кода, заданного в этом элементе, зависит от элемента `Usage Page` (рис. 6.5).

6.5.3. Примеры дескрипторов

Листинг 6.3 содержит пример дескриптора репорта мыши (*см. разд. 10.7.1*), листинг 6.4 — клавиатуры (*см. разд. 10.7.2*), листинг 6.5 — монитора. Использование нестандартного HID-дескриптора мы будем рассматривать при создании своего HID-устройства (*см. гл .10*).

Листинг 6.3. Дескриптор репорта мыши

```
USAGE_PAGE (Generic Desktop)       05 01
USAGE (Mouse)                      09 02
COLLECTION (Application)           A1 01
  USAGE (Pointer)                  09 01
  COLLECTION (Physical)            A1 00
    USAGE_PAGE (Button)            05 09
    USAGE_MINIMUM (Button 1)       19 01
    USAGE_MAXIMUM (Button 3)       29 03
    LOGICAL_MINIMUM (0)            15 00
    LOGICAL_MAXIMUM (1)            25 01
    REPORT_COUNT (3)               95 03
    REPORT_SIZE (1)                75 01
    INPUT (Data,Var,Abs)           81 02
    REPORT_COUNT (1)               95 01
    REPORT_SIZE (5)                75 05
    INPUT (Cnst,Var,Abs)           81 03
    USAGE_PAGE (Generic Desktop)   05 01
    USAGE (X)                      09 30
    USAGE (Y)                      09 31
    LOGICAL_MINIMUM (-127)         15 81
    LOGICAL_MAXIMUM (127)          25 7F
    REPORT_SIZE (8)                75 08
```

```
    REPORT_COUNT (2)               95 02
    INPUT (Data,Var,Rel)           81 06
  END_COLLECTION                   C0
END_COLLECTION                     C0
```

Листинг 6.4. Дескриптор репорта клавиатуры

```
USAGE_PAGE (Generic Desktop)                   05 01
USAGE (Keyboard)                               09 06
COLLECTION (Application)                       A1 01
  USAGE_PAGE (Keyboard)                        05 07
  USAGE_MINIMUM (Keyboard LeftControl)         19 E0
  USAGE_MAXIMUM (Keyboard Right GUI)           29 E7
  LOGICAL_MINIMUM (0)                          15 00
  LOGICAL_MAXIMUM (1)                          25 01
  REPORT_SIZE (1)                              75 01
  REPORT_COUNT (8)                             95 08
  INPUT (Data,Var,Abs)                         81 02
  REPORT_COUNT (1)                             95 01
  REPORT_SIZE (8)                              75 08
  INPUT (Cnst,Var,Abs)                         81 03
  REPORT_COUNT (5)                             95 05
  REPORT_SIZE (1)                              75 01
  USAGE_PAGE (LEDs)                            05 08
  USAGE_MINIMUM (Num Lock)                     19 01
  USAGE_MAXIMUM (Kana)                         29 05
  OUTPUT (Data,Var,Abs)                        91 02
  REPORT_COUNT (1)                             95 01
  REPORT_SIZE (3)                              75 03
  OUTPUT (Cnst,Var,Abs)                        91 03
  REPORT_COUNT (6)                             95 06
  REPORT_SIZE (8)                              75 08
  LOGICAL_MINIMUM (0)                          15 00
  LOGICAL_MAXIMUM (101)                        25 65
  USAGE_PAGE (Keyboard)                        05 07
```

```
  USAGE_MINIMUM (Reserved (no event indicated))  19 00
  USAGE_MAXIMUM (Keyboard Application)           29 65
  INPUT (Data,Ary,Abs)                           81 00
END_COLLECTION                                   C0
```

Листинг 6.5. Дескриптор репорта монитора

```
USAGE_PAGE (Monitor)                             05 80
USAGE (Monitor Control)                          09 01
COLLECTION (Application)                         A1 01
  REPORT_ID (1)                                  85 01
  LOGICAL_MINIMUM (0)                            15 00
  LOGICAL_MAXIMUM (255)                          26 FF 00
  REPORT_SIZE (8)                                75 08
  REPORT_COUNT (128)                             95 80
  USAGE (EDID Information)                       09 02
  FEATURE (Data,Var,Abs,Buf)                     B2 02 01
  REPORT_ID (2)                                  85 02
  REPORT_COUNT (243)                             95 F3
  USAGE (VDIF Information)                       09 03
  FEATURE (Data,Var,Abs,Buf)                     B2 02 01
  REPORT_ID (3)                                  85 03
  USAGE_PAGE (VESA Virtual Controls)             05 82
  REPORT_COUNT (1)                               95 01
  REPORT_SIZE (16)                               75 10
  LOGICAL_MAXIMUM (200)                          26 C8 00
  USAGE (Brightness)                             09 10
  FEATURE (Data,Var,Abs)                         B1 02
  REPORT_ID (4)                                  85 04
  LOGICAL_MAXIMUM (100)                          25 64
  USAGE (Contrast)                               09 12
  FEATURE (Data,Var,Abs)                         B1 02
  REPORT_COUNT (6)                               95 06
  LOGICAL_MAXIMUM (255)                          26 FF 00
  USAGE (Video Gain Red)                         09 16
```

```
  USAGE (Video Gain Green)                      09 18
  USAGE (Video Gain Blue)                       09 1A
  USAGE (Video Black Level Red)                 09 6C
  USAGE (Video Black Level Green)               09 6E
  USAGE (Video Black Level Blue)                09 70
  FEATURE (Data,Var,Abs)                        B1 02
  REPORT_ID (5)                                 85 05
  LOGICAL_MAXIMUM (127)                         25 7F
  USAGE (Horizontal Position )                  09 20
  USAGE (Horizontal Size )                      09 22
  USAGE (Vertical Position )                    09 30
  USAGE (Vertical Size )                        09 32
  USAGE (Trapezoidal Distortion)                09 42
  USAGE (Tilt)                                  09 44
  FEATURE (Data,Var,Abs)                        B1 02
END_COLLECTION                                  C0
```

6.6. Запросы к HID-устройству

HID-устройство должно отвечать на стандартные запросы:

- `Get_Status` (для типа запроса Device);
- `Set_Address`;
- `Get_Descriptor` (включая специфические дескриптор репорта и HID-дескриптор);
- `Get_Configuration`;
- `Set_Configuration`.

Запрос `Set_Descriptor` позволяет хосту изменять дескрипторы HID-устройства. Обработка этого запроса необязательна.

Кроме стандартных запросов, HID-устройство может обрабатывать специфические HID-запросы. Такие запросы имеют следующие значения полей:

- поле `bmRequestType` может принимать одно из значений: `10100001B` или `00100001B`;
- поле `bRequest` задает тип специального запроса:
  - `0x01` — `GET_REPORT`;
  - `0x02` — `GET_IDLE`;

- 0x03 — GET_PROTOCOL;
- 0x04—0x08 — резерв;
- 0x09 — SET_REPORT;
- 0x0A — SET_IDLE;
- 0x0B — SET_PROTOCOL.

Из этого списка обязательным является только запрос GET_REPORT[1], а запросы GET_PROTOCOL и SET_PROTOCOL обрабатываются, если HID-устройство является загрузочным (табл. 6.6).

Таблица 6.6. *Специальные запросы HID-устройства*

Запрос	Код	Источник данных	Длина данных	Обязателен
GET_REPORT	0x01	Устройство	Длина репорта	Да
GET_IDLE	0x02	Устройство	1	Нет
GET_PROTOCOL	0x03	Устройство	1	Да, для загрузочных устройств
SET_REPORT	0x09	Хост	Длина репорта	Нет
SET_IDLE	0x0A	Хост	0	Нет
SET_PROTOCOL	0x0B	Хост	0	Да, для загрузочных устройств

6.6.1. Запрос *GET_REPORT*

Запрос GET_REPORT позволяет хосту принять данные через нулевую конечную точку. Поля запроса имеют следующие значения:

- bmRequestType — 10100001b;
- bRequest — GET_REPORT (код 0x01);
- wValue — тип репорта и его идентификатор;
- wIndex — номер интерфейса;
- wLength — длина репорта;
- Data — данные репорта.

[1] Если используется только передача данных от HID-устройства к хосту, можно обойтись и без обработки GET_REPORT.

Старший байт поля wValue содержит тип репорта:

- 01 — Input;
- 02 — Output;
- 03 — Feature;
- 04—FF — резерв.

Младший байт поля wValue содержит идентификатор репорта или ноль, если идентификатор репорта не используется. Обработка этого запроса обязательна для всех HID-устройств.

6.6.2. Запрос *SET_REPORT*

Запрос SET_REPORT позволяет хосту передать данные HID-устройству через нулевую конечную точку. Поля запроса имеют следующие значения:

- bmRequestType — 00100001b;
- bRequest — SET_REPORT (код 0x09);
- wValue — тип репорта и его идентификатор;
- wIndex — номер интерфейса;
- wLength — длина репорта;
- Data — данные репорта.

Значение полей этого запроса такое же, как для запроса GET_REPORT. HID-устройство может игнорировать запросы SET_REPORT.

6.6.3. Запрос *GET_IDLE*

Запрос GET_IDLE позволяет хосту прочитать текущее значение *длительности* (idle rate) для репорта. Поля запроса имеют следующие значения:

- bmRequestType — 10100001b;
- bRequest — GET_IDLE (код 0x02);
- wValue — идентификатор репорта в младшем байте;
- wIndex — номер интерфейса;
- wLength — 1;
- Data — значение длительности (*см. разд. 6.6.4*).

6.6.4. Запрос *SET_IDLE*

Запрос SET_IDLE позволяет хосту задать значение длительности для репортов. Поля запроса содержат следующие значения:

- bmRequestType — 00100001b;
- bRequest — SET_IDLE (код 0x0A);
- wValue — *длительность* (idle duration) в старшем байте и идентификатор репорта в младшем байте;
- wIndex — номер интерфейса;
- wLength — 0.

Величина длительности позволяет управлять передачей репортов в случае отсутствия изменений в передаваемых данных. Нулевое значение длительности означает *бесконечную паузу* (indefinite), в этом случае устройство будет передавать репорты только в случае изменений состояния. Ненулевое значение задает длительность интервала с кратностью 4 мс, в течение которого конечная точка отвечает на запросы пакетами NAK, если в передаваемых данных нет изменений. Спецификация определяет точность выдерживания интервала ±10%.

Если поле Report ID равно нулю, то установка производится для всех репортов, иначе — только для репорта с данным идентификатором.

Для клавиатуры рекомендуется устанавливать интервал 500 мс (это будет означать время ожидания перед началом дублирования символов, repeat rate), а для джойстиков и мыши — бесконечный (эти устройства будут передавать данные только в случае изменения состояния).

6.6.5. Запрос *GET_PROTOCOL*

Запрос GET_PROTOCOL позволяет хосту прочитать текущее значение выбранного протокола. Поля запроса содержат следующие значения:

- bmRequestType — 10100001b;
- bRequest — GET_PROTOCOL (код 0x03);
- wValue — 0;
- wIndex — интерфейс;
- wLength — 1;
- Data — тип протокола:
  - 0 — загрузочный протокол;
  - 1 — обычный протокол.

Этот запрос используется только для загрузочных устройств.

6.6.6. Запрос *SET_PROTOCOL*

Запрос `SET_PROTOCOL` позволяет производить переключение между загрузочным протоколом и обычным протоколом. Поля запроса имеют следующие значения:

- `bmRequestType` — `00100001b`;
- `bRequest` — `SET_PROTOCOL` (код `0x0B`);
- `wValue` — тип протокола:
  - `0` — загрузочный протокол;
  - `1` — обычный протокол;
- `wIndex` — интерфейс;
- `wLength` — `0`.

Этот запрос используется только для загрузочных устройств.

6.7. Инструменты

Для удобства создания драйверов для HID-устройств USB Форум разработал две программы: HID Descriptor Tool и USB Compliance Tool. Их можно загрузить с сайта **http://www.usb.org**.

Первая программа (рис. 6.6) позволяет автоматизировать процесс написания дескриптора репорта и проверить его правильность прежде, чем копировать его в HID-устройство.

Вторая программа представляет собой набор инструментальных средств, позволяющих выполнить серию основных тестов на любом USB-устройстве, плюс содержит дополнительные тесты для HID-устройств, хабов и устройств связи. USB Compliance Tool загружает свой собственный драйвер для испытуемого USB-устройства, с помощью которого можно выбрать и послать в него стандартный набор запросов.

6.8. Драйверы для HID-устройств в Windows

Любые операции с устройством (не только с HID, но и с любым другим) производятся с помощью дескриптора устройства, получаемого с помощью вызова функции `CreateFile`. Основным параметром этой функции является имя устройства. Получение имени USB-устройства мы будем обсуждать позже (*см. гл. 11*), а пока предположим, что у нас есть открытый дескриптор устройства.

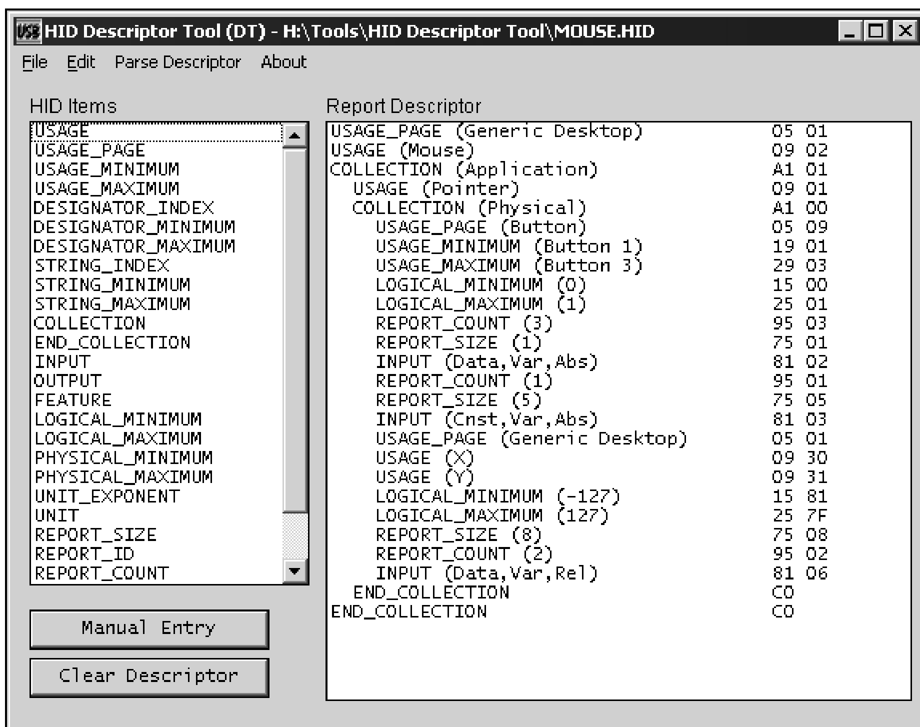


Рис. 6.6. Программа генерации и проверки дескрипторов репортов

Заголовки основных функций, предоставляемых HID-драйвером, приведены в листинге 6.6.

Листинг 6.6. Заголовки функций драйвера для HID-устройства (Delphi)

```
// возвращает GUID, связанный с HID
procedure HidD_GetHidGuid(var HidGuid: TGUID) stdcall;
// возвращает атрибуты HID-устройства (идентификаторы
// производителя и продукта)
function HidD_GetAttributes(HidDeviceObject: THandle; var HidAttrs:
THIDDAttributes): LongBool; stdcall;
// возвращает указатель на буфер, содержащий информацию
// о возможностях устройства
function HidD_GetPreparsedData(HidDeviceObject: THandle;
var PreparsedData: PHIDPPreparsedData): LongBool; stdcall;
```

```
// возвращает структуру, описывающую возможности устройства

function HidP_GetCaps(PreparsedData: PHIDPPreparsedData;
var Capabilities: THIDPCaps): NTSTATUS; stdcall;

// читает Feature-репорт из устройства

function HidD_GetFeature(HidDeviceObject: THandle; var Report;
Size: Integer): LongBool; stdcall;

// передает Feature-репорт в устройство

function HidD_SetFeature(HidDeviceObject: THandle; var Report;
Size: Integer): LongBool; stdcall;
```

Обмен данными с устройством можно производить с помощью обычных Windows API функций `ReadFile` и `WriteFile`, соответственно для входных и выходных репортов и функций HID API `HidD_GetFeature` и `HidD_SetFeature` для специальных репортов.

> ***Важно!***
>
> При использовании функции `ReadFile` пользовательская программа "проваливается" в системный драйвер для HID-устройств и будет находиться там до тех пор, пока не получит от HID-устройства запрошенное количество данных. Не помогает даже использование функции `ReadFileEx`. При написании программы необходимо таким образом разместить вызовы `ReadFile`, чтобы она не "вешала" основное приложение при ожидании данных с HID-устройства.

HID-функции содержатся в модуле Hid.Dll (непосредственно файл драйвера называется hidclass.sys). Для использования этих функций в Visual Studio нужно подключить модуль hidsdi.h. В Borland Delphi придется либо подключать эти функции вручную (как показано в листинге 6.7), либо использовать готовые классы, например, Hid.Pas из библиотеки JEDI (**http://delphi-jedi.org**). Работу с этими функциями на C# мы рассмотрим позже (*см. гл. 10*).

Листинг 6.7. Подключение HID-функций

```
const
  HidModuleName = 'HID.dll';


procedure HidD_GetHidGuid(var HidGuid: TGUID) stdcall;
{$EXTERNALSYM HidD_GetHidGuid}


function HidD_GetPreparsedData(HidDeviceObject: THandle;
var PreparsedData: PHIDPPreparsedData): LongBool; stdcall;
{$EXTERNALSYM HidD_GetPreparsedData}
```

```
function HidD_FreePreparsedData(PreparsedData: PHIDPPreparsedData): Long-
Bool; stdcall;
{$EXTERNALSYM HidD_FreePreparsedData}
... ... ... ... ... ... ... ... ... ... ... ... ... ...
procedure HidD_GetHidGuid; external HidModuleName name 'HidD_GetHidGuid';

function HidD_GetPreparsedData; external HidModuleName name
'HidD_GetPreparsedData';

function HidD_FreePreparsedData; external HidModuleName name
'HidD_FreePreparsedData';
```

В качестве примера приведем небольшую программу, получающую список HID-устройств и отображающую свойства одного из них (листинг 6.8).

Листинг 6.8. Использование HID-функций

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, ExtCtrls, StdCtrls, Buttons;

type
  TForm1 = class(TForm)
    lbLog: TListBox;
    Panel1: TPanel;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    procedure DisplayHIDInformation(HidName : String);
  public
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.dfm}

Uses SetupApi, Hid;

// Отображение списка HID-устройств
// и их свойств
procedure TForm1.Button1Click(Sender: TObject);
var HidGuid : TGuid; PnPHandle : HDevInfo;
    DevData: TSPDevInfoData;
    DeviceInterfaceData: TSPDeviceInterfaceData;
    FunctionClassDeviceData: PSPDeviceInterfaceDetailData;
    Success: LongBool;
    DevIndex: DWORD;
    BytesReturned: DWORD;
    HidName : String;
begin
 // Очистить лог
 lbLog.Items.Clear;
 // Получить GUID для класса HID
 HidD_GetHidGuid(HidGuid);
 // ПОлучаем дескриптор PnP для класса HID
 PnPHandle := SetupDiGetClassDevs(@HidGuid, nil,
             0, DIGCF_PRESENT or DIGCF_DEVICEINTERFACE);
 // Если ошибка, то выходим
 If PnPHandle = Pointer(INVALID_HANDLE_VALUE) then Exit;

 Try
  // Индекс текущего устройства
  DevIndex := 0;
  // Цикл по всем устройствам в классе HID
  Repeat
   DeviceInterfaceData.cbSize := SizeOf(TSPDeviceInterfaceData);
```

```
  // Получить информацию об интерфейсах устройства номер DevIndex
  Success := SetupDiEnumDeviceInterfaces(
               PnPHandle, nil, HidGuid, DevIndex, DeviceInterfaceData);
  If Success then begin
   DevData.cbSize := SizeOf(DevData);
   BytesReturned  := 0;
   // Получаем подробности об устройстве с интерфейсом
   // DeviceInterfaceData
   // Сначала вызываем с нулевым размером буфера, получаем
   // размер необходимого буфера, потом вызываем повторно,
   // сформировав правильный буфер
   SetupDiGetDeviceInterfaceDetail(PnPHandle,
          @DeviceInterfaceData, nil, 0, BytesReturned, @DevData);
   If (BytesReturned <> 0) and
        (GetLastError = ERROR_INSUFFICIENT_BUFFER) then begin
    // Создаем буфер
    FunctionClassDeviceData := AllocMem(BytesReturned);
    FunctionClassDeviceData.cbSize := 5;
    // Получаем информацию
    If SetupDiGetDeviceInterfaceDetail(PnPHandle,
          @DeviceInterfaceData, FunctionClassDeviceData,
          BytesReturned, BytesReturned, @DevData) then begin
     // Отобразить PnP-имя устройства
     HidName:= StrPas(PChar(@FunctionClassDeviceData.DevicePath));
     lbLog.Items.Add(HidName);
     // Отобразить информацию об устройстве
     DisplayHIDInformation(HidName);
    End;
    // Освободить буфер
    FreeMem(FunctionClassDeviceData);
   End;
  End;
  // Следующее устройство
  Inc(DevIndex);
 Until not Success;
Finally
```

```
  SetupDiDestroyDeviceInfoList(PnPHandle);
 End;
end;

// Отображение информации о HID-устройстве
procedure TForm1.DisplayHIDInformation(HidName : String);
var HidHandle : THandle;
    CanReadWriteAccess : Boolean;
    Attributes : THIDDAttributes;
    NumInputBuffers : Integer;
    Buffer  : array [0..253] of WideChar;

    // Получение строки по дескриптору
    Function GetString(StrDescriptor : Byte): WideString;
    var Buffer : array [0..253] of WideChar;
    begin
     Result := 'Ошибка';
     if StrDescriptor <> 0 then
      if HidD_GetIndexedString(HidHandle,
              StrDescriptor, Buffer, SizeOf(Buffer)) then
       Result:= Buffer;
    end;

begin
 // Сначала пробуем открыть HID-устройство
 // в режиме r/w
 lbLog.Items.Add(' Пробуем открыть HID-устройство...');
 HidHandle:= CreateFile(PChar(@HidName[1]),
              GENERIC_READ or GENERIC_WRITE,
              FILE_SHARE_READ or FILE_SHARE_WRITE,
              nil,
              OPEN_EXISTING, 0, 0
 );

 // Устройство поддерживает запись?
 CanReadWriteAccess:= HidHandle <> INVALID_HANDLE_VALUE;
```

```
// Если не получилось, пробуем открыть
// в режиме только чтения данных
If not CanReadWriteAccess then begin
 HidHandle:= CreateFile(PChar(@HidName[1]),
              0,
              FILE_SHARE_READ or FILE_SHARE_WRITE,
              nil,
              OPEN_EXISTING, 0, 0
 );
End else begin
 lbLog.Items.Add(' Устройство открыто в режиме read/write');
End;

// Если не получилось - ошибка и выход
If HidHandle = INVALID_HANDLE_VALUE then begin
 lbLog.Items.Add(' Ошибка открытия устройства');
 Exit;
End else begin
 lbLog.Items.Add(' Устройство открыто в режиме read only!');
End;

// Получаем атрибуты устройства
Attributes.Size := SizeOf(THIDDAttributes);
If HidD_GetAttributes(HidHandle, Attributes) then begin
 lbLog.Items.Add(
   Format('   VendorID=%d, ProductID=%d,
   VersionNumber=%d',[Attributes.VendorID,
   Attributes.ProductID, Attributes.VersionNumber]));
End else begin
 lbLog.Items.Add(' Ошибка HidD_GetAttributes');
End;

// Получаем число буферов
If HidD_GetNumInputBuffers(HidHandle, NumInputBuffers) then begin
 lbLog.Items.Add(
      Format('   Число входных буферов=%d',[NumInputBuffers]));
```

```
 End else begin
  lbLog.Items.Add(' Ошибка HidD_GetNumInputBuffers');
 End;

 // Получаем идентификатор изготовителя
 FillChar(Buffer, SizeOf(Buffer), #0);
 If HidD_GetManufacturerString(HidHandle, Buffer, SizeOf(Buffer)) then
begin
  lbLog.Items.Add(Format('   Производитель=%s',[Buffer]));
 End else begin
  lbLog.Items.Add(' Ошибка HidD_GetManufacturerString');
 End;

 // Получаем идентификатор продукта
 FillChar(Buffer, SizeOf(Buffer), #0);
 If HidD_GetProductString(HidHandle, Buffer, SizeOf(Buffer)) then begin
  lbLog.Items.Add(Format('   Продукт=%s',[Buffer]));
 End else begin
  lbLog.Items.Add(' Ошибка HidD_GetProductString');
 End;

 // Получаем серийный номер
 FillChar(Buffer, SizeOf(Buffer), #0);
 If HidD_GetSerialNumberString(HidHandle, Buffer, SizeOf(Buffer)) then
begin
  lbLog.Items.Add(Format('   Серийный номер=%s',[Buffer]));
 End else begin
  lbLog.Items.Add(' Ошибка HidD_GetSerialNumberString');
 End;

 // Освободить дескриптор устройства
 CloseHandle(HidHandle);
end;

end.
```

Вид формы и результат работы показаны на рис. 6.7.

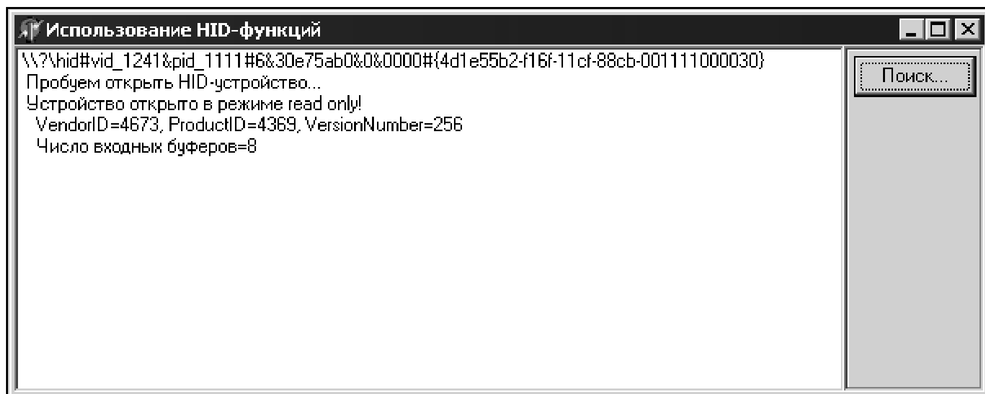
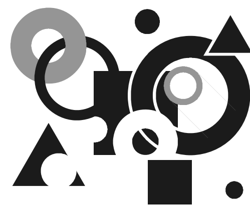


Рис. 6.7. Результат программы, демонстрирующей HID-функции

Заметим, что имя HID-устройства непохоже на обычное имя устройства, как, например, COM1 или LPT. Это имя присваивается менеджером системы Plug and Play и выглядит, например, так:

```
\\?\hid#vid_1241&pid_1111#6&30e75ab0&0&0000#{4d1e55b2-f16f-11cf-
88cb-001111000030}
```

Для получения имени HID-устройства мы использовали функции Setup API (*см. гл. 11*).



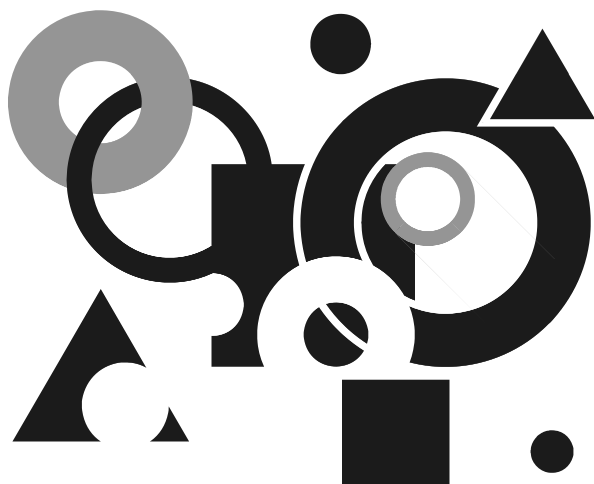
Глава 7

Другие классы USB

Класс USB описывает группу USB-устройств с общими свойствами или назначением. Специфицирование интерфейсов общения схожих по функциональности устройств позволяет операционной системе иметь один драйвер нижнего уровня, а различия в функциональности скрывать на уровне драйверов верхнего уровня.

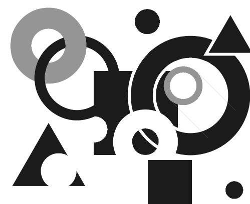
В настоящее время Windows поддерживает следующие классы USB:

- Hub Device Class — класс хабов поддерживается драйвером usbhub.sys;
- Human Interface Device (HID) Class — класс HID-устройств (*см. гл. 6 и 10*). Этот класс поддерживается драйвером hidclass.sys;
- Audio Class — класс аудиоустройств поддерживается драйвером sysaudio.sys;
- Mass Storage Class — класс устройств хранения данных поддерживается драйвером usbstor.sys;
- Printer Class — класс принтеров, подключаемых к USB-порту, поддерживается драйвером usbprint.sys;
- Communications Device Class (CDC) — класс коммуникационных устройств (*см. гл. 5 и 9*). Драйверы этого класса различаются в зависимости от типа устройства;
- Content Security Class — класс устройств защиты. Поддерживается частично на уровне использования в других драйверах. Например, многие аудиодрайверы имеют поддержку шифрования трафика;
- Imaging Class — цифровые камеры и сканеры поддерживаются драйвером usbscan.sys в Windows XP и драйверами usbscn9x.sys для ранних систем. Эти драйверы поддерживают архитектуру WIA (Windows Imaging Architecture).



ЧАСТЬ III

Практика программирования USB



Глава 8

Создание USB-устройства на основе AT89C5131

Мы выбрали Atmel AT89C5131 по нескольким причинам. Во-первых, это не дорогой, но достаточно быстродействующий микроконтроллер с широко известным ядром 8051, имеющий 6 конечных точек. Во-вторых, для реализации схемы требуется минимум дополнительной обвязки. Не маловажно и наличие бесплатного ассемблера, компилятора языка C, программатора и драйверов для Windows/Linux. Очень удобна возможность программирования микроконтроллера не по SPI, а "напрямую" по USB-каналу. В нашей книге мы ограничимся тем минимумом информации, который потребуется для реализации простого USB-устройства, а полное описание микроконтроллера и дополнительную информацию о нем можно найти на сайте Atmel (**http://www.atmel.com**).

Внимание

При описании микросхем мы будем использовать часто употребляемые обозначения, такие как UART, SRAM, FLASH, АЦП и т. д. Объем книги не позволяет привести расшифровку этих обозначений и мы надеемся, что читатель, готовящийся к созданию своего USB-устройства, обладает достаточными знаниями в этой области.

8.1. Общая информация об AT89C5131

Микроконтроллер AT89C5131 имеет следующие характеристики.

- Ядро 80C52X2 (6 тактов на инструкцию):
  - максимальная частота ядра 40 МГц;
  - двойной указатель данных;
  - полнодуплексный улучшенный UART (EUART);

   - три шестнадцатиразрядных таймера-счетчика: T0, T1 и T2;
   - 256 байт сверхоперативной памяти.
- 32 Кбайт встроенной флэш-памяти с внутрисхемным программированием через USB или UART.
- 4 Кбайт EEPROM для загрузочного сектора (3 Кбайт) и данных (1 Кбайт).
- 1 Кбайт встроенного расширенного ОЗУ (XRAM).
- USB 1.1 и USB 2.0 FS модуль с прерыванием на завершение передачи:
   - конечная точка 0 для управления передачей: буфер FIFO емкостью 32 байта;
   - 6 программируемых конечных точек с направлениями ввода и вывода и с режимами передачи данных (Bulk), прерываний (Interrupt) и изохронным (Isochronous):
      - ◊ конечные точки 1, 2, 3: буфер FIFO емкостью 32 байта;
      - ◊ конечные точки 4, 5: буфер FIFO емкостью 2×64 байта с двойной буферизацией (режим Ping-pong);
      - ◊ конечная точка 6: буфер FIFO емкостью 2×512 байта с двойной буферизацией (режим Ping-pong);
   - прерывания по приостановке/возобновлению;
   - сброс при подаче питания и сброс USB-шины;
   - генерация 48 МГц для полноскоростного функционирования шины;
   - отключение от USB-шины по запросу микроконтроллера.
- Пятиканальный программируемый счетный массив с шестнадцатиразрядным счетчиком, быстродействующим выходом, сравнением/захватом фронтов, функциями ШИМ и сторожевого таймера.
- Программируемый сторожевой таймер (однократно разрешает после сброса): от 50 мс до 6 с при 4 МГц.
- Интерфейс подключения клавиатуры с генерацией прерывания на порте P1 (8 разрядов).
- SPI-интерфейс.
- 34 линии ввода-вывода.
- 4 вывода для подключения светодиода с программируемым источником тока: 2, 6 или 10 мА.
- Четырехуровневая система прерываний с приоритетами (11 источников).
- Режимы холостого хода и экономичный.

- Встроенный генератор 0: 32 МГц с аналоговой схемой ФАПЧ для синтеза 48 МГц.
- Стабилизатор напряжения и выход опорного источника: 3,3 В, 4 мА.
- Низкий диапазон напряжения источника питания:
  - 3,0—3,6 В;
  - максимальный рабочий ток 30 мА (при 40 МГц);
  - потребление 100 мкА в экономичном режиме.
- Диапазон напряжения питания USB (не доступно в первой версии):
  - 3,6—5,5 В;
  - максимальный рабочий ток 30 мА (при 40 МГц);
  - ток потребления в экономичном режиме 200 мкА.
- Коммерческий и промышленный температурные диапазоны.
- Корпуса: PLCC52, VQFP64, MLF48, SO28.

Ориентировочная стоимость этого микроконтроллера $9.

Кроме того, выпускается аналогичный контроллер AT89C5132, имеющий два десятибитных АЦП, 64 Кбайт флэш-памяти и 2,5 Кбайт SRAM.

8.2. Структурная схема AT89C5131

AT89C5131 содержит специальный аппаратный модуль, который позволяет ему обеспечивать обмен данными по USB (рис. 8.1). Для этого необходимы опорные синхроимпульсы с частотой 48 МГц, которые вырабатываются контроллером синхронизации. Эти синхроимпульсы используются для формирования тактовых импульсов с частотой 12 МГц из принятого дифференциального потока данных USB и передачи данных на высокой скорости, соответствующей требованиям к USB-устройствам. Формирование синхроимпульсов выполняется цифровой системой ФАПЧ (DPLL, Digital Phase Locked Loop). Коэффициент деления задается битами USBCDx регистра USBCLK.

Блок последовательного интерфейса (SIE, Serial Interface Engine) выполняет следующие функции:

- кодирование и декодирование NRZI;
- вставку и извлечение бита;
- формирование битов проверки на четность (CRC);
- автоматическое формирование сигналов ACK и NACK;
- идентификацию типа передатчика;

- контроль адресов;
- восстановление синхроимпульсов (при помощи DPLL[1]).

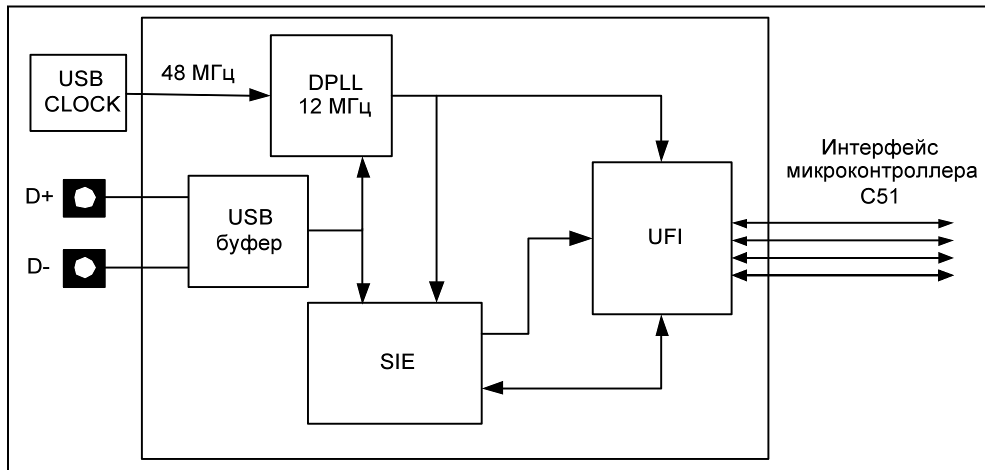


Рис. 8.1. Структурная схема USB-модуля в микроконтроллере AT89C5131

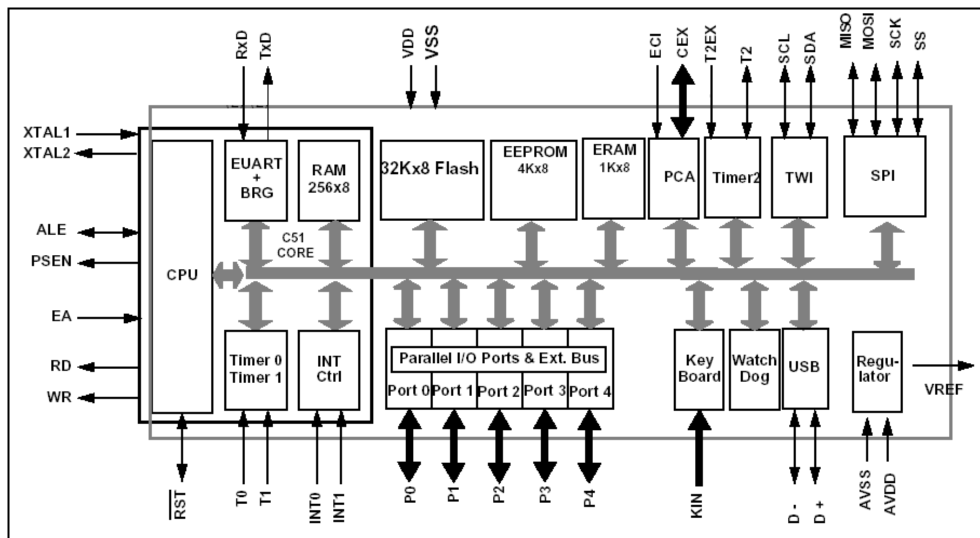


Рис. 8.2. Структурная схема AT89C5131

[1] DPLL (Digital Phase-Locked Loop) — цифровая система фазовой автоподстройки частоты.

Функциональный интерфейсный модуль (UFI, Universal Function Interface) обеспечивает интерфейс между микроконтроллерами и SIE. Он управляет обменом на пакетном уровне с минимальными программными затратами, выполняющими запись и считывание буфера FIFO конечной точки.

Для экономии места остальные функциональные составляющие, показанные на рис. 8.2 (источники прерываний, USB-регистры и т. д.) мы рассмотрим при описании регистров микроконтроллера, необходимых нам для работы.

8.3. USB-регистры AT89C5131

Микроконтроллер AT89C5131 содержит несколько ключевых регистров, с помощью которых производится конфигурирование и обмен по USB. Обычные регистры, присутствующие во всех контроллерах семейства 8051, рассматривать не будем.

8.3.1. Регистр *USBCON*

Регистр `USBCON` (байт, адрес 0xBC) — основной управляющий регистр USB-модуля. После сброса регистр принимает значение `00000000b`.

Регистр содержит следующие биты.

- [7] *USBE* — бит включения модуля USB. Установка данного бита включает USB-контроллер. Сброс — отключает и сбрасывает USB-контроллер.
- [6] *SUSPCLK* — бит приостановки синхронизации USB. Установка бита отключает вход синхроимпульсов с частотой 48 МГц (продолжение детектирования все еще возможно). Сброс — включает вход синхроимпульсов.
- [5] *SDRMWUP* — бит передачи удаленного пробуждения. Устанавливается для вызова внешнего прерывания USB-контроллера при удаленном пробуждении. Резюме исходящего потока передается, только если бит `RMWUPE` установлен, все синхроимпульсы активизированы и шина USB находилась в состоянии приостановки (состояние SUSPEND) не менее 5 мс. Сбрасывается программно.
- [4] *DETACH* — установка этого бита симулирует команду `Detach` на линии.
- [3] *UPRSM* — резюме исходящего потока (только чтение). Устанавливается аппаратно после установки бита `SDRMWUP`, если бит `RMWUPE` был также установлен. Сбрасывается аппаратно после передачи резюме исходящего потока.

- [2] *RMWUPE* — бит разрешения удаленного пробуждения. Устанавливается для разрешения запроса резюме исходящего потока ведущего USB-устройства. Сбрасывается после отображения резюме исходящего потока в `RSMINPR`.

Замечание

Не устанавливайте этот бит, если у ведущего USB-устройства для прибора не установлена функция `DEVICE_REMOTE_WAKEUP`.

- [1] *CONFG* — конфигурационный бит. Устанавливается после корректной обработки запроса `SET_CONFIGURATION` с ненулевым значением. Сбрасывается программно после получения запроса `SET_CONFIGURATION` с нулевым значением. Сбрасывается аппаратно при аппаратном сбросе или после обнаружения сброса на шине USB.
- [0] *FADDEN* — бит разрешения функции адресации. Устанавливается программным обеспечением прибора после успешного завершения транзакции `SET_ADDRESS`. В последующем он не должен сбрасываться программно. Сбрасывается аппаратно при аппаратном сбросе или после обнаружения сброса на шине USB. Когда этот бит сброшен, используется функция адресации по умолчанию (нулевая конечная точка).

Листинг 8.1 показывает описания и макросы для работы с регистром `USBCON`.

Листинг 8.1. Регистр `USBCON`

```
// Описание регистра
sfr USBCON = 0xBC;
// Описание констант для доступа к битам регистра
#define MSK_USBE      0x80   // Бит включения модуля USB
#define MSK_SUSPCLK   0x40   // Бит приостановки синхронизации USB
#define MSK_SDRMWUP   0x20   // Бит передачи удаленного пробуждения
#define MSK_DETACH    0x10   // Отключение от линии
#define MSK_UPRSM     0x08   // Бит резюме исходящего потока (r/o)
#define MSK_RMWUPE    0x04   // Бит разрешения удаленного пробуждения
#define MSK_CONFG     0x02   // Конфигурационный бит
#define MSK_FADDEN    0x01   // Бит разрешения функции адресации
// Включение/выключение модуля USB
#define Usb_enable()              (USBCON |= MSK_USBE)
#define Usb_disable()             (USBCON &= ~MSK_USBE)
```

```
// Подключение/отключение от линии
#define Usb_detach()                (USBCON |= MSK_DETACH)
#define Usb_attach()                (USBCON &= ~MSK_DETACH)
// Конфигурационный бит
#define Usb_set_CONFG()             (USBCON |= MSK_CONFG)
#define Usb_clear_CONFG()           (USBCON &= ~MSK_CONFG)
// Бит разрешения функции адресации
#define Usb_set_FADDEN()            (USBCON |= MSK_FADDEN)
#define Usb_clear_FADDEN()          (USBCON &= ~MSK_FADDEN)
// Бит приостановки синхронизации USB
#define Usb_set_suspend_clock()     (USBCON |= MSK_SUSPCLK)
#define Usb_clear_suspend_clock()   (USBCON &= ~MSK_SUSPCLK)
```

8.3.2. Регистр *USBADDR*

Регистр `USBADDR` (байт, адрес 0xC6) — регистр USB-адреса. После сброса регистр принимает значение `00000000b`. Регистр содержит следующие биты.

- [7] *FEN* — бит активизации функции. Устанавливается для активизации функции. Программное обеспечение прибора установит этот бит после приема сброса USB и примет участие в текущем конфигурационном процессе с установленным по умолчанию адресом (`FEN` сбросится в 0).
- [6:0] *UADD6*:*UADD0* — биты USB-адреса. Эти биты содержат заданный по умолчанию адрес после включения питания или сброса шины USB. Запись их состояния произойдет после принятия программным обеспечением прибора запроса `SET_ADDRESS`.

Листинг 8.2 показывает описания и макросы для работы с регистром `USBADR`.

Листинг 8.2. Регистр `USBADDR`

```
// Описание регистра
sfr USBADDR = 0xC6;
// Описание констант для доступа к битам регистра
#define MSK_FEN       0x80
// Конфигурирование USB-адреса
#define Usb_configure_address(x)      (USBADDR = (0x80 | x))
#define Usb_address()                 (USBADDR & 0x7F)
```

8.3.3. Регистр *USBINT*

Регистр USBINT (байт, адрес 0xBD) — регистр флагов основных USB-прерываний. После сброса регистр принимает значение 00000000b.

Регистр содержит следующие биты.

- [7:6]. Зарезервированы. Всегда считываются как 0. Не пытайтесь установить эти биты.
- [5] *WUPCPU* — флаг прерывания пробуждения центрального процессора (ЦП). Устанавливается аппаратно, когда находящийся в режиме приостановки USB-контроллер перезапускается сигналом активности шины. Установка этого бита вызывает USB-прерывание, когда установлен бит EWUPCPU в регистре USBIEN. Сбрасывается программно после переключения всех синхроимпульсов.
- [4] *EORINT* — флаг прерывания окончания сброса. Устанавливается аппаратно при обнаружении USB-контроллером окончания сброса. Установка этого бита вызывает USB-прерывание, когда установлен бит EEORINT в регистре USBIEN. Сбрасывается программно.
- [3] *SOFINT* — флаг прерывания при обнаружении начала кадра. Устанавливается аппаратно после приема пакета начала кадра SOF. Установка этого бита вызывает USB-прерывание, когда установлен бит ESOFINT в регистре USBIEN. Сбрасывается программно.
- [2:1]. Зарезервированы. Всегда считываются как 0. Не пытайтесь установить эти биты.
- [0] *SPINT* — флаг прерывания при приостановке. Устанавливается аппаратно при обнаружении приостановки (шина не занята в течение трех кадровых периодов). Установка этого бита вызывает USB-прерывание, когда установлен бит ESPINT в регистре USBIEN. Сбрасывается программно.

Листинг 8.3 показывает описания и макросы для работы с регистром USBINT.

Листинг 8.3. Регистр USBINT

```
// Описание регистра
sfr USBINT = 0xBD;
// Описание констант для доступа к битам регистра
#define MSK_SPINT     0x01 // Флаг прерывания при приостановке
#define MSK_SOFINT    0x08 // Флаг прерывания при обнаружении начала
// кадра
#define MSK_EORINT    0x10 // Флаг прерывания окончания сброса
```

```
#define MSK_WUPCPU    0x20 // Флаг прерывания пробуждения ЦП
// Флаг прерывания окончания сброса
#define Usb_clear_reset()            (USBINT &= ~MSK_EORINT)
#define Usb_reset()                  (USBINT & MSK_EORINT)
// Флаг прерывания пробуждения ЦП
#define Usb_clear_resume()           (USBINT &= ~MSK_WUPCPU)
#define Usb_resume()                 (USBINT & MSK_WUPCPU)
// Флаг прерывания при обнаружении начала кадра
#define Usb_clear_sof()              (USBINT &= ~MSK_SOFINT)
#define Usb_sof()                    (USBINT & MSK_SOFINT)
// Флаг прерывания при приостановке
#define Usb_clear_suspend()          (USBINT &= ~MSK_SPINT)
#define Usb_suspend()                (USBINT & MSK_SPINT)
```

8.3.4. Регистр *USBIEN*

Регистр `USBIEN` (байт, адрес 0xBE) — регистр разрешений основных USB-прерываний. После сброса регистр принимает значение `00000000b`.

Регистр содержит следующие биты.

- [7:6]. Зарезервированы. Всегда считываются как 0. Не пытайтесь установить эти биты.
- [5] *EWUPCPU* — бит разрешения прерывания при пробуждении ЦП. Установка этого бита разрешает прерывание при пробуждении ЦП. Сброс бита запрещает прерывание при пробуждении ЦП.
- [4] *EEORINT* — бит разрешения прерывания по окончанию сброса. Установка этого бита разрешает прерывание по окончанию сброса. Сброс этого бита запрещает прерывание по окончанию сброса.
- [3] *ESOFINT* — бит разрешения прерывания при обнаружении начала кадра. Установка этого бита разрешает прерывание при обнаружении начала кадра. Сброс этого бита запрещает прерывание при обнаружении начала кадра.
- [2:1] Зарезервированы. Всегда считываются как 0. Не пытайтесь установить эти биты.
- [0] *ESPINT* — бит разрешения прерывания при обнаружении приостановки. Установка этого бита разрешает прерывание при обнаружении приостановки. Сброс этого бита запрещает прерывание при обнаружении приостановки.

Листинг 8.4 показывает описания и макросы для работы с регистром USBIEN.

Листинг 8.4. Регистр USBIEN

```
// Описание регистра
sfr USBIEN = 0xBE;
// Описание констант для доступа к битам регистра
#define MSK_ESPINT    0x01
#define MSK_ESOFINT   0x08
#define MSK_EEORINT   0x10
#define MSK_EWUPCPU   0x20
// Макросы для доступа к битам регистра
#define Usb_enable_reset_int()        (USBIEN |= MSK_EEORINT)
#define Usb_enable_resume_int()       (USBIEN |= MSK_EWUPCPU)
#define Usb_enable_sof_int()          (USBIEN |= MSK_ESOFINT)
#define Usb_enable_suspend_int()      (USBIEN |= MSK_ESPINT)
#define Usb_disable_reset_int()       (USBIEN &= ~MSK_EEORINT)
#define Usb_disable_resume_int()      (USBIEN &= ~MSK_EWUPCPU)
#define Usb_disable_sof_int()         (USBIEN &= ~MSK_ESOFINT)
#define Usb_disable_suspend_int()     (USBIEN &= ~MSK_ESPINT)
```

8.3.5. Регистр *UEPNUM*

Регистр UEPNUM (байт, адрес 0xC7) — регистр номера USB конечной точки. После сброса регистр принимает значение 00000000b.

Регистр содержит следующие биты.

- [7:4]. Зарезервированы. Всегда считываются как 0. Не пытайтесь установить эти биты.
- [3:0] *EPNUM* — биты номера конечной точки. Задают номер конечной точки, к которой будет происходить обращение при считывании и записи регистров UEPSTAX, UEPDATX, UBYCTLX или UEPCONX. Значения могут быть 0, 1, 2, 3, 4, 5 или 6.

Листинг 8.5 показывает описания и макросы для работы с регистром UEPNUM.

Листинг 8.5. Регистр UEPNUM

```
// Описание регистра
sfr UEPNUM = 0xC7;
// Макрос выбора номера конечной точки
#define Usb_select_ep(e)              (UEPNUM = e)
```

8.3.6. Регистр *UEPCONX*

Регистр `UEPCONX` (байт, адрес 0xD4) — управляющий регистр конечной USB-точки, номер которой задан в регистре `UEPNUM`. После сброса регистр принимает значение `00000000b`.

Регистр содержит следующие биты.

- [7] *EPEN* — бит активизации конечной точки. Установка бита включает конечную точку в соответствии с конфигурацией прибора. Нулевая конечная точка всегда активизируется после аппаратного сброса или сброса шины USB и участвует в конфигурации прибора. Сброс бита отключает конечную точку в соответствии с конфигурацией прибора.
- [6:4] Зарезервированы. Всегда считываются как 0. Не пытайтесь установить эти биты.
- [3] *DTGL* — бит изменения статуса данных (только чтение). Устанавливается аппаратно при приеме пакета Data1. Сбрасывается аппаратно при приеме пакета Data0.
- [2] *EPDIR* — бит установки направленности конечных точек. Установка бита устанавливает пакетные (Bulk), прерывающие (Interrupt) и изохронные (Isochronous) конечные точки в режим приема (IN Direction). Сброс бита устанавливает пакетные, прерывающие и изохронные конечные точки в режим передачи (OUT Direction). Бит не влияет на управляющие конечные точки.
- [1:0] *EPTYPE1*:*EPTYPE0* — биты установки типа конечных точек (для нулевой конечной точки всегда должен быть установлен "управляющий" тип):
  - 00 — управляющая конечная точка;
  - 01 — изохронная конечная точка;
  - 10 — пакетная конечная точка;
  - 11 — прерывающая конечная точка.

Заметим, что если новый пакет данных принимается до изменения состояния бита `DTGL` с 0 на 1 или с 1 на 0, то возможна потеря пакета. Когда это происходит у пакетной конечной точки, встроенное программное обеспечение прибора должно предположить, что ведущее USB-устройство повторило передачу правильно принятого пакета, т. к. ведущее USB-устройство не приняло правильного подтверждения (ACK), после этого программа должна отказаться от передачи нового пакета (конечная точка сбрасывается к Data0 только при конфигурировании).

Листинг 8.6 показывает описания и макросы для работы с регистром `UEPCONX`. После начального конфигурирования конечной точки необходимо выполнить ее сброс (см. регистр `UEPRST`).

Листинг 8.6. Регистр `UEPCONX`

```
// Описание регистра
sfr UEPCONX = 0xD4;
// типы конечных точек
#define CONTROL              0x00
#define ISOCHRONOUS_OUT      0x01
#define BULK_OUT             0x02
#define INTERRUPT_OUT        0x03
#define ISOCHRONOUS_IN       0x05
#define BULK_IN              0x06
#define INTERRUPT_IN         0x07
// Описание констант для доступа к битам регистра
#define MSK_EPEN      0x80 /* включение конечной точки */
// Конфигурирование типа конечной точки
#define Usb_configure_ep_type(x)      (UEPCONX = x)
// Конфигурирование конечной точки
void usb_configure_endpoint(byte ep_num, byte ep_type)
{
  // Переключиться на нужную конечную точку
  Usb_select_ep(ep_num);
  // Установить тип
  Usb_configure_ep_type(ep_type);
}
```

8.3.7. Регистр *UEPSTAX*

Регистр `UEPSTAX` (байт, адрес 0xCE) — регистр управления и статуса конечной USB-точки, номер которой задан в регистре `UEPNUM`. После сброса регистр принимает значение `00000000b`.

Регистр содержит следующие биты.

- [7] *DIR* — бит управления направлением конечной точки. Этот бит учитывается только тогда, когда конечной точке присвоен тип контрольной. Должен быть установлен для стадии данных. Для остальных случаев должен быть сброшен. Этот бит должен быть установлен при прерывании RXSETUP до изменения состояния любого другого бита. Также он определяет фазу статуса (IN для проверки записи и OUT для контроля чтения). Этот бит должен быть сброшен для стадии статуса контрольной исходящей транзакции.
- [6]. Зарезервирован. Всегда считывается как 0. Не пытайтесь установить этот бит.
- [5] *STALLRQ* — ит запроса остановки установления связи. Установка бита приведет к посылке ответа STALL (остановки) ведущему USB-устройству для следующей установки связи. В противном случае этот бит должен быть сброшен.
- [4] *TXRDY* — управляющий бит готовности передачи пакета. Бит должен быть установлен после записи пакета в буфер FIFO конечной точки для передачи IN данных. Данные должны записываться в буфер FIFO конечной точки только после сброса этого бита. Установка этого бита без записи данных в буфер FIFO приведет к посылке пакета нулевой длины, который рекомендуется в общем случае и может потребоваться для обозначения передачи в тех случаях, когда длина последнего пакета данных равна максимальной (например, для контрольного считывания передачи). Сбрасывается аппаратно сразу после посылки пакета изохронной конечной точкой или после получения контрольной, пакетной или прерывающей конечной точкой подтверждения от ведущего устройства.
- [3] *STLCRC* — флаг прерывания при приостановке посылки и флаг прерывания при обнаружении ошибки CRC.
  - Для контрольных, пакетных и прерывающих конечных точек он устанавливается аппаратно после посылки при помощи запроса STALLRQ приостановки установления связи (после этого произойдет прерывание конечной точки, если оно разрешено в регистре `UEPIEN`) и сбрасывается аппаратно после приема пакета SETUP.
  - Для изохронных конечных точек этот бит устанавливается аппаратно при обнаружении ошибки в принятых данных (ошибка CRC в приня-

тых данных), после этого произойдет прерывание конечной точки, если оно разрешено в регистре `UEPIEN` и сбрасывается аппаратно после приема неповрежденных данных.

- [2] *RXSETUP* — флаг прерывания при получении пакета SETUP. Устанавливается аппаратно после получения допустимого пакета SETUP от ведущего USB-устройства. После этого произойдет прерывание, если оно разрешено в регистре `UEPIEN`. Сбрасывается программно после считывания SETUP данных из буфера FIFO конечной точки.
- [1] *RXOUT* — флаг прерывания при принятии исходящих данных. Устанавливается аппаратно после принятия исходящего пакета. После этого произойдет прерывание, если оно разрешено в регистре `UEPIEN` и все текущие исходящие пакеты отклонены до сброса этого бита. Однако в управляющих конечных точках ранее принятая транзакция SETUP может переписать содержимое буфера FIFO конечной точки, даже если был принят пакет данных при установленном этом флаге. Сбрасывается программно после считывания исходящих данных из буфера FIFO конечной точки.
- [0] *TXCMPL* — флаг прерывания по окончанию передачи входящих данных. Устанавливается аппаратно после передачи входящего пакета изохронной точкой или после получения подтверждения приема (ACK) от ведущего USB-устройства контрольной, пакетной или прерывающей конечной точкой. После этого произойдет прерывание, если оно разрешено в регистре `UEPIEN`. Сбрасывается программно перед следующей установкой бита `TXRDY`.

Листинг 8.7 показывает описания и макросы для работы с регистром `UEPSTAX`.

Листинг 8.7. Регистр UEPSTAX

```
// Описание регистра
sfr UEPSTAX = 0xCE;
// Описание констант для доступа к битам регистра
   // Флаг прерывания по окончанию передачи входящих данных
#define MSK_TXCMPL    0x01
   // Флаг прерывания при принятии исходящих данных
#define MSK_RXOUT     0x02
#define MSK_RXOUTB0   0x02
   // Флаг прерывания при получении пакета SETUP
#define MSK_RXSETUP   0x04
```

```
   // Флаг прерывания при приостановке посылки
   // Флаг прерывания при обнаружении ошибки CRC
#define MSK_STALLED   0x08
   // Управляющий бит готовности передачи пакета
#define MSK_TXRDY     0x10
   // Бит запроса остановки установления связи
#define MSK_STALLRQ   0x20
#define MSK_RXOUTB1   0x40
#define MSK_RXOUTB0B1 0x42
   // Бит управления направлением конечной точки
#define MSK_DIR       0x80
// Макросы для работы с регистром UEPSTAX
#define Usb_set_stall_request()       (UEPSTAX |= MSK_STALLRQ)
#define Usb_clear_stall_request()     (UEPSTAX &= ~MSK_STALLRQ)
#define Usb_clear_stalled()           (UEPSTAX &= ~MSK_STALLED)
#define Usb_stall_sent()              (UEPSTAX & MSK_STALLED)
#define Usb_stall_requested()         (UEPSTAX & MSK_STALLRQ)
#define Usb_clear_rx_setup()          (UEPSTAX &= ~MSK_RXSETUP)
#define Usb_setup_received()          (UEPSTAX & MSK_RXSETUP)
#define Usb_clear_DIR()               (UEPSTAX &= ~MSK_DIR)
#define Usb_set_DIR()                 (UEPSTAX |= MSK_DIR)
#define Usb_set_tx_ready()            (UEPSTAX |= MSK_TXRDY)
#define Usb_clear_tx_ready()          (UEPSTAX &= ~MSK_TXRDY)
#define Usb_clear_tx_complete()       (UEPSTAX &= ~MSK_TXCMPL)
#define Usb_tx_complete()             (UEPSTAX & MSK_TXCMPL)
#define Usb_tx_ready()                (UEPSTAX & MSK_TXRDY)
#define Usb_clear_rx()                (UEPSTAX &= ~MSK_RXOUT)
#define Usb_clear_rx_bank0()          (UEPSTAX &= ~MSK_RXOUTB0)
#define Usb_clear_rx_bank1()          (UEPSTAX &= ~MSK_RXOUTB1)
#define Usb_rx_complete()             (UEPSTAX & MSK_RXOUTB0B1)
```

8.3.8. Регистр *UEPRST*

Регистр `UEPRST` (байт, адрес 0xD5) — регистр сброса конечной точки. После сброса регистр принимает значение `00000000b`. Для сброса буфера FIFO конечной точки установите и сбросьте соответствующий бит перед

началом любой операции до аппаратного сброса или при получении сброса шины USB.

Регистр содержит следующие биты.

- [7]. Зарезервирован. Всегда считываются как 0. Не пытайтесь установить этот бит.
- [6] *EP6RST* — сброс буфера FIFO шестой конечной точки.
- [5] *EP5RST* — сброс буфера FIFO пятой конечной точки.
- [4] *EP4RST* — сброс буфера FIFO четвертой конечной точки.
- [3] *EP3RST* — сброс буфера FIFO третьей конечной точки.
- [2] *EP2RST* — сброс буфера FIFO второй конечной точки.
- [1] *EP1RST* — сброс буфера FIFO первой конечной точки.
- [0] *EP0RST* — сброс буфера FIFO нулевой конечной точки.

Листинг 8.8 показывает описания и макросы для работы с регистром UEPRST.

Листинг 8.8. Регистр UEPRST

```
// Описание регистра
sfr  UEPRST = 0xD5;
// Описание констант для доступа к битам регистра
#define MSK_EP6RST    0x40
#define MSK_EP5RST    0x20
#define MSK_EP4RST    0x10
#define MSK_EP3RST    0x08
#define MSK_EP2RST    0x04
#define MSK_EP1RST    0x02
#define MSK_EP0RST    0x01
// Макрос для сброса конечной точки
#define Usb_reset_endpoint(ep_num) UEPRST=0x01<<ep_num;UEPRST=0x00
```

8.3.9. Регистр *UEPINT*

Регистр UEPINT (байт, адрес 0xF8) — регистр прерываний конечных точек (только чтение). После сброса регистр принимает значение 00000000b.

Флаг прерывания от конечной точки устанавливается аппаратно после установки прерывания в регистре UEPSTAX и если прерывание от этой конечной

точки разрешено в регистре `UEPIEN`. Флаг должен быть сброшен программно. Регистр содержит следующие биты.

- [7] Зарезервирован. Всегда считываются как 0. Не пытайтесь установить этот бит.
- [6] *EP6INT* — флаг прерывания от шестой конечной точки.
- [5] *EP5INT* — флаг прерывания от пятой конечной точки.
- [4] *EP4INT* — флаг прерывания от четвертой конечной точки.
- [3] *EP3INT* — флаг прерывания от третьей конечной точки.
- [2] *EP2INT* — флаг прерывания от второй конечной точки.
- [1] *EP1INT* — флаг прерывания от перовой конечной точки.
- [0] *EP0INT* — флаг прерывания от нулевой конечной точки.

Листинг 8.9 показывает описания и макросы для работы с регистром `UEPRST`.

Листинг 8.9. Регистр `UEPRST`

```
// Описание регистра
sfr  UEPINT  = 0xF8;
// Проверка наличия прерывания от конечных точек
#define Usb_endpoint_interrupt()      (UEPINT != 0)
```

8.3.10. Регистр *UEPIEN*

Регистр `UEPIEN` (байт, адрес 0xC2) — регистр разрешений прерываний конечных USB точек. После сброса регистр принимает значение `00000000b`.

Для разрешения прерывания от конечной точки необходимо установить соответствующий бит, а для запрещения — сбросить. Регистр содержит следующие биты.

- [7] Зарезервирован. Всегда считываются как 0. Не пытайтесь установить этот бит.
- [6] *EP6INTE* — бит разрешения прерывания шестой конечной точки.
- [5] *EP5INTE* — бит разрешения прерывания пятой конечной точки.
- [4] *EP4INTE* — бит разрешения прерывания четвертой конечной точки.
- [3] *EP3INTE* — бит разрешения прерывания третьей конечной точки.
- [2] *EP2INTE* — бит разрешения прерывания второй конечной точки.

- [1] *EP1INTE* — бит разрешения прерывания перовой конечной точки.
- [0] *EP0INTE* — бит разрешения прерывания нулевой конечной точки.

Листинг 8.10 показывает описания и макросы для работы с регистром UEPIEN.

Листинг 8.10. Регистр UEPIEN

```
// Описание регистра
sfr  UEPINT  = 0xC2;
// Описание констант для доступа к битам регистра
#define MSK_EP6INTE    0x40
#define MSK_EP5INTE    0x20
#define MSK_EP4INTE    0x10
#define MSK_EP3INTE    0x08
#define MSK_EP2INTE    0x04
#define MSK_EP1INTE    0x02
#define MSK_EP0INTE    0x01
// Разрешение/запрещение прерываний от конечных точек
#define Usb_enable_ep_int(e)          (UEPIEN |= (0x01 << e))
#define Usb_disable_ep_int(e)         (UEPIEN &= ~(0x01 << e))
```

8.3.11. Регистр *UEPDATX*

Регистр UEPDATX (байт, адрес 0xCF) — регистр данных буфера FIFO конечной точки, номер которой задан в регистре UEPNUM. После сброса состояние регистра не регламентировано.

Листинг 8.11 показывает описания и макросы для работы с регистром UEPDATX.

Листинг 8.11. Регистр UEPDATX

```
// Описание регистра
sfr    UEPDATX    = 0xCF;
// Чтение данных с конечной точки
#define Usb_read_byte()               (UEPDATX)
// Передача данных в конечную точку
#define Usb_write_byte(x)             (UEPDATX = x)
```

8.3.12. Регистр *UBYCTLX*

Регистр `UBYCTLX` (байт, адрес 0xE2) — регистр счетчика байтов конечной точки, номер которой задан в регистре `UEPNUM`. После сброса регистр принимает значение `00000000b`.

Регистр содержит следующие биты.

- [7] Зарезервирован. Всегда считывается как 0. Не пытайтесь установить этот бит.
- [6:0] *BYCT6*:*BYCT0* — счетчик байтов принятых пакетов данных. Значение этого счетчика равно количеству байтов данных, принятых после получения идентификатора (PID) данных.

Листинг 8.12 показывает описания и макросы для работы с регистром `UBYCTLX`.

Листинг 8.12. Регистр `UBYCTLX`

```
// Описание регистра
sfr  UBYCTLX = 0xE2;
// Получение счетчика
#define Usb_get_nb_byte ()        (UBYCTLX)
```

8.3.13. Регистр *UFNUML*

Регистр `UFNUML` (байт, адрес 0xBA) — регистр младших битов номера кадра (только чтение). После сброса регистр принимает значение `0x00`. Регистр содержит младшие 8 бит 11-битного номера кадра.

8.3.14. Регистр *UFNUMH*

Регистр `UFNUMH` (байт, адрес 0xBB) — регистр старших битов номера USB-кадра (только чтение). После сброса регистр принимает значение `0x00`.

Регистр содержит следующие биты.

- [7:6] Зарезервированы. Всегда считываются как 0. Не пытайтесь установить эти биты.
- [5] *CRCOK* — бит отсутствия ошибки CRC в принятом кадре. Устанавливается аппаратно после принятия неповрежденного номера кадра в стартовом или кадровом пакете. Обновляется после каждого принятия стартового или кадрового пакета.

- [4] *CRCERR* — бит наличия ошибки CRC в принятом кадре. Устанавливается аппаратно после принятия поврежденного номера кадра в стартовом или кадровом пакете. Обновляется после каждого принятия стартового или кадрового пакета.
- [3] Зарезервирован. Всегда считывается как 0. Не пытайтесь установить этот бит.
- [2:0] *FNUM10:FNUM8* — номер кадра. Старшие 3 бита одинадцатибитного номера кадра. Они доступны в последнем принятом пакете SOF. Биты `FNUM` не изменяются, если принят поврежденный SOF.

8.4. Схемотехника AT89C5131

На рис. 8.3 показана схема расположения выводов AT89C5131 в 52-контактном корпусе. Схема включения AT89C5131 очень проста (рис. 8.4).

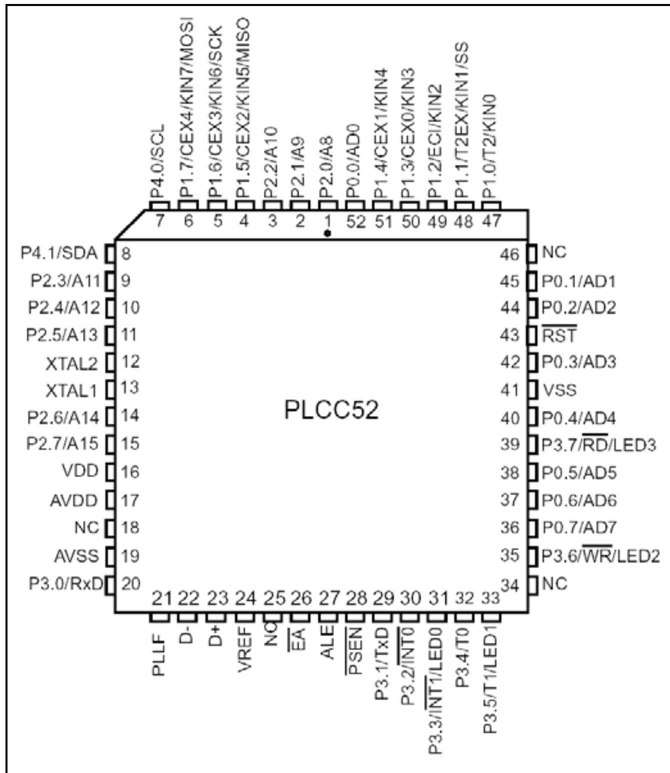


Рис. 8.3. Схема расположения выводов AT89C5131

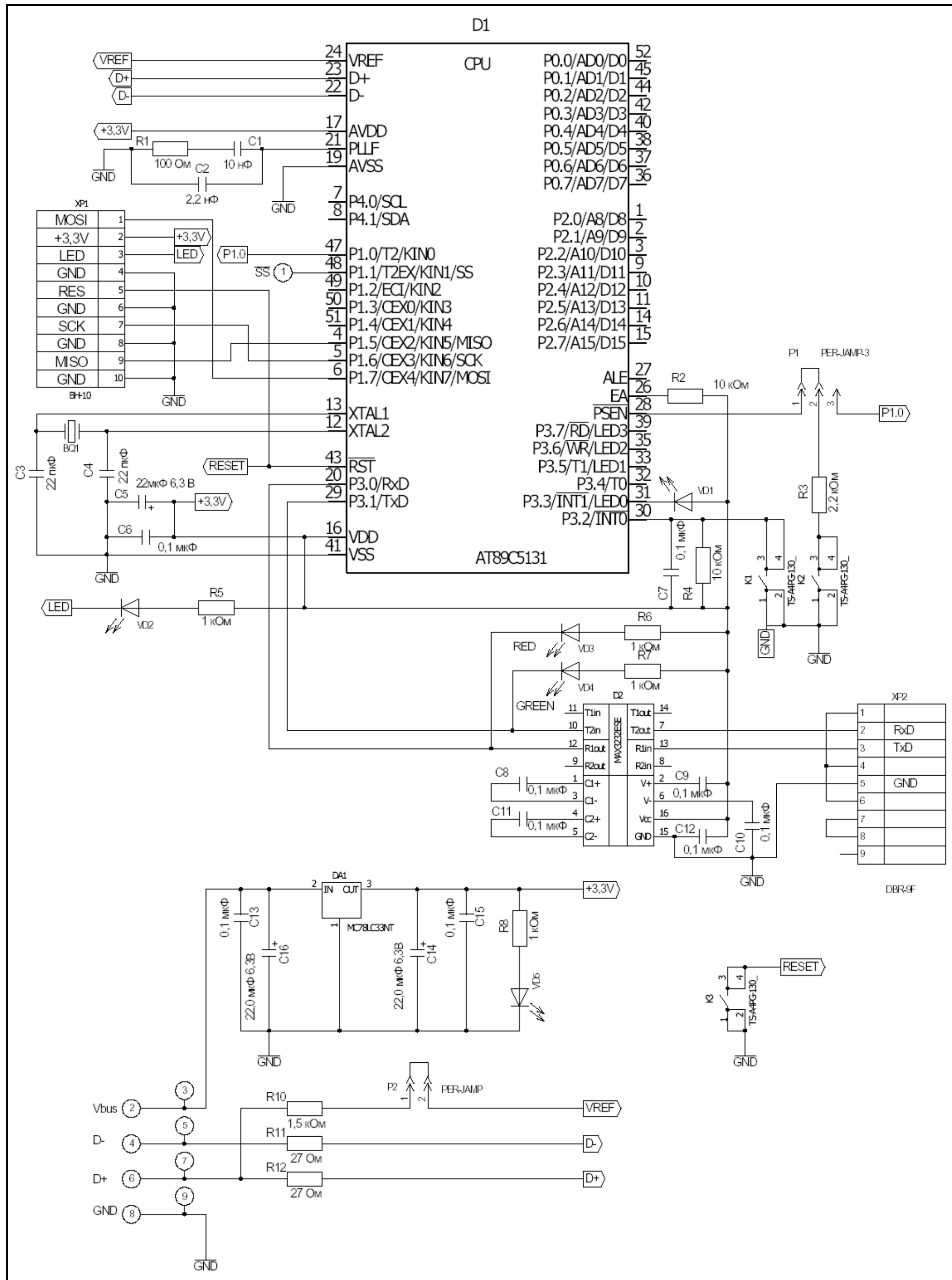


Рис. 8.4. Схема тестовой платы для AT89C5131

8.5. Базовый проект для AT89C5131

Программа, записываемая в микроконтроллер, должна выполнять следующие действия:

- инициализировать USB-интерфейс и внутренние переменные;
- производить обмен по шине USB, включая процедуру нумерации и обработку запросов от системы PnP;
- производить выполнение других операций, необходимых программисту.

Для простоты понимания код нашей программы мы будем реализовывать в несколько этапов. Сначала мы сделаем простую программу, позволяющую работать с нулевой конечной точкой. Затем, мы добавим обработку строковых дескрипторов, что позволит Windows более детально "рассказать" о подключенном USB-устройстве.

8.5.1. Первая версия программы для AT89C5131

В листинге 8.13 приведена функция `main` (основная функция программы на языке С, *см. гл. 2*), реализующая основной алгоритм программы микроконтроллера.

Листинг 8.13. Функция `main` программы для AT89C5131

```
void main()
{
 /* выполнить инициализацию USB */
 UsbInit();

 /* основной цикл программы */
 for (;;)
 {
  /* если устройство отключено от шины */
  if (!usb_connected)
  {
    /* если получен сигнал побудки */
    if (Usb_resume())
    {
      /* установить флаг активности */
      usb_connected = TRUE;
```

```
        /* сброс режима SUSPEND */
        Usb_clear_suspend_clock();
        Usb_clear_suspend();
        Usb_clear_resume();
        Usb_clear_sof();
      }
    /* если устройство подключено к шине */
    } else {
     /* если получен сигнал "засыпания" */
     if (Usb_suspend())
     {
       usb_connected = FALSE;
       Usb_clear_suspend();
       Usb_set_suspend_clock();
     }

     /* если получен сигнал сброса */
     if (Usb_reset())
     {
       Usb_clear_reset();
     }

     /* сигнал SOF */
     if (Usb_sof())
     {
      Usb_clear_sof();
     }

     /* обнаружено прерывание от конечной точки */
     if (Usb_endpoint_interrupt())
     {
      /* переключиться на 0 конечную точку */
      Usb_select_ep(0);
      /* если получен пакет SETUP (см. разд. 1.1.14) */
      if (Usb_setup_received())
```

```
    {
     /* начать процесс нумерации */
     UsbControlPacketProcessed();
    }
   }
         // переключиться на первую конечную точку
         // переключиться на вторую конечную точку
         //.....................................
  }
 } /* end for ;; */
}
```

В начале работы программы вызывается функция `UsbInit` (ее код приведен в листинге 8.14). Эта функция достаточно проста и заключается в выполнении следующей последовательности действий:

1) инициализации переменных;
2) инициализации внутреннего генератора (его функции описаны в документации на микросхему);
3) включении USB-интерфейса;
4) инициировании процесса нумерации (для этого USB-устройство отключается от шины и не ранее чем через 5 мс подключается);
5) конфигурировании нулевой конечной точки (разумеется, нулевая точка конфигурируется как управляющая);
6) сбросе нулевой конечной точки (reset);
7) разрешении прерываний от нулевой конечной точки (при этом прерывание как таковое не вызывается, т. к. бит `IE` сброшен, но соответствующие биты в регистре `UEPINT` выставляются).

После выполнения этих операций USB-устройство готово к конфигурированию через канал нулевой конечной точки.

Листинг 8.14. Функция инициализации USB

```
// Описание делителей PLL в зависимости от используемого кварца
#define PLL_3MHz            0xF0
#define PLL_4MHz            0xC0
#define PLL_6MHz            0x70
#define PLL_8MHz            0x50
```

```
#define PLL_12MHz            0x30
#define PLL_16MHz            0x20
#define PLL_18MHz            0x72
#define PLL_20MHz            0xB4
#define PLL_24MHz            0x10
#define PLL_32MHz            0x21
#define PLL_40MHz            0xB9

// Функция инициализации USB
void UsbInit()
{
/* Инициализация переменных */
 usb_configuration_nb = 0x00;
 usb_connected = FALSE;

 /* конфигурирование синхронизатора */
 Pll_set_div(PLL_24MHz); /* используется кварц 24 МГц */
 Pll_enable();           /* включение генератора      */

 /* Включение USB */
 Usb_enable();

 /* Отключиться-подключиться для начала нумерации */
 Usb_detach();
 delay5();
 Usb_attach();
 delay5();

/* Конфигурирование нулевой конечной точки */
 usb_configure_endpoint(0, CONTROL|MSK_EPEN);
/* Сброс нулевой конечной точки */
 usb_reset_endpoint(0);
/* Разрешение прерываний от нулевой конечной точки */
 Usb_enable_ep_int(0);
}
```

После инициализации программа циклически выполняет обработку сигналов побудки, прерываний, сброса и т. д. При этом контролируется возникновение прерываний от конечных точек с помощью проверки регистра `UEPINT`. Конечно, основной цикл можно заменить на пустышку `for (;;){ }`, а всю работу перенести в обработчики прерываний, но, как нам кажется, линейная структура программы более читабельна и понятна.

Важно!

Особо следует обратить внимание на строку инициализации синхронизатора:

```
Pll_set_div(PLL_24MHz);
```

Константа `PLL_24MHz` задает частоту используемого кварцевого генератора. В нашей плате используется кварц на 24 МГц.

Принцип обработки USB-сигналов легко понять из листинга, а мы займемся процедурой обработки пакетов нулевой точки. Вообще, согласно спецификации USB общие схемы обработки пакетов нулевой точки имеют вид, как показано в листинге 8.15, а реализация функции `UsbControlPacketProcessed` показана в листинге 8.17. Реализация вспомогательных функций для AT89C5131 показана в листинге 8.16.

Листинг 8.15. Общая схема обработки пакетов нулевой конечной точки

```
// Чтение пакета SETUP (8 байт)
  for (i=0; i<8; i++)
   SetupPacket.b[i]= Usb_read_byte();
// Завершение фазы SETUP. Сброс FIFO.
  EndSetupStage();

// === Запрос с передачей данных
// Переключение на передачу
 Usb_set_DIR();
// Запись ответа в FIFO
 Usb_write_byte(0x00);
// Передача данных из FIFO
 SendDataFromFIFO();
// Ожидание пустого OUT-пакета
 WaitForOutZeroPacket();

// === Запрос без передачи данных
// Ожидание пустого IN-пакета
 SendInZeroPacket();
```

```
// === Запрос приема данных
// Ожидание завершения приема
  WaitForFillFIFO();
// Чтение данных
  ACC= Usb_read_byte();
... ... ... ... ... ...
  ACC= Usb_read_byte();
// Завершение чтения
  EndReadData();
// Ожидание пустого IN-пакета
  SendInZeroPacket();
```

Листинг 8.16. Процедура обработки пакетов нулевой точки

```
void EndSetupStage()
{
  Usb_clear_rx_setup();
}
void SendDataFromFIFO()
{
  Usb_set_tx_ready();
  while ((!(Usb_tx_complete())) || (Usb_setup_received()));
  Usb_clear_tx_complete();
}
void WaitForOutZeroPacket()
{
  while ((!(Usb_rx_complete())) || (Usb_setup_received()));
  Usb_clear_rx();
  Usb_clear_DIR();
}
void SendInZeroPacket()
{
  SendDataFromFIFO();
  Usb_clear_DIR();
}
void WaitForFillFIFO()
```

```
{
  while (!(Usb_rx_complete()));
}

void EndReadData()
{
  Usb_clear_rx();
}
```

Листинг 8.17. Процедура обработки пакетов нулевой точки

```
void UsbControlPacketProcessed()
{
  /* ------------------------------ */
  /*       Читаем пакет SETUP       */
  /* Состоит из 8 байт (см. 1.2.1): */
  /*  [0] byte bmRequestType        */
  /*  [1] byte bRequest             */
  /*  [2] word wValue               */
  /*  [4] word wIndex               */
  /*  [6] word wLength              */
  /* ------------------------------ */
  for (i=0; i<8; i++)
   SetupPacket.b[i]= Usb_read_byte();

  /* ---------------------------------- */
  /* Завершение фазы SETUP. Сброс FIFO. */
  /* ---------------------------------- */
  EndSetupStage();

  /* ------------------------------ */
  /*         Обработка запроса      */
  /* ------------------------------ */
  switch (SetupPacket.wRequest)
  {
    case GET_STATUS_DEVICE:
     GetStatusDevice();
     break;
```

```
    case GET_STATUS_INTERF:
     GetStatusInterface();
     break;

    case GET_STATUS_ENDPNT:
     GetStatusEndpoint();
     break;

    case GET_DESCRIPTOR_DEVICE:
    case GET_DESCRIPTOR_INTERF:
    case GET_DESCRIPTOR_ENDPNT:
     GetDescriptor();
     break;

    case GET_CONFIGURATION:
     GetConfiguration();
     break;

    case SET_CONFIGURATION:
     SetConfiguration();
     break;

    case SET_ADDRESS:
     SetAddress();
     break;

    default:
     STALL();
     break;
  }
}
```

Как видно из листинга, мы читаем 8 байтов запроса и в соответствии с кодом запроса (листинг 8.18) производим вызов того или иного обработчика. Все байты запроса мы сохраняем в специальный буфер, описанный таким образом, чтобы можно было легко получить доступ как к отдельным байтам, так и к полям запроса (листинг 8.19).

Листинг 8.18. Коды запросов

```
/* коды запросов */
#define GET_STATUS_DEVICE      0x8000
#define GET_STATUS_INTERF      0x8100
#define GET_STATUS_ENDPNT      0x8200
#define CLEAR_FEATURE_DEVICE   0x0001
#define CLEAR_FEATURE_INTERF   0x0101
#define CLEAR_FEATURE_ENDPNT   0x0201
#define SET_FEATURE_DEVICE     0x0003
#define SET_FEATURE_INTERF     0x0103
#define SET_FEATURE_ENDPNT     0x0203
#define SET_ADDRESS            0x0005
#define GET_DESCRIPTOR_DEVICE  0x8006
#define GET_DESCRIPTOR_INTERF  0x8106
#define GET_DESCRIPTOR_ENDPNT  0x8206
#define SET_DESCRIPTOR         0x0007
#define GET_CONFIGURATION      0x8008
#define SET_CONFIGURATION      0x0009
#define GET_INTERFACE          0x810A
#define SET_INTERFACE          0x010B
#define SYNCH_FRAME            0x820C
#define GET_REPORT             0xA101
```

Листинг 8.19. Описание типа буфера для запроса `SETUP`

```
/* Структура пакета SETUP */
typedef struct{
  byte bmRequestType;
  byte bRequest;
  word wValue;
  word wIndex;
  word wLength;
} SETUP_PACKET;

typedef union
{
```

```
  SETUP_PACKET setup;
  byte         b[8];
  word         wRequest;
} UsbSetupPacket;
```

Важно отметить, что если запрос не поддерживается USB-устройством, то для него обязательно должны быть выполнена процедура отклонения `STALL()`, как, например, в ветке `default`. Код этой процедуры приведен в листинге 8.20.

Листинг 8.20. Процедура отклонения запроса

```
void STALL(){
  Usb_clear_rx_setup();
  Usb_set_stall_request();
  while (!Usb_stall_sent());
  Usb_clear_stall_request();
  Usb_clear_stalled();
  Usb_clear_DIR();
}
```

Самый сложный обработчик — обработчик запроса `GET_DESCRIPTOR`. Для обработки этого запроса следует выполнить следующую последовательность действий:

1) определить тип запрашиваемого дескриптора согласно старшему байту поля `wValue`;
2) в зависимости от типа запрашиваемого дескриптора получить указатель на нужную структуру (`pbuffer`) и размер передаваемого блока данных (`data_to_transfer`);
3) разделить передаваемые данные на блоки размером максимального пакета (константа `EP_CONTROL_LENGTH`);
4) произвести поблочную передачу данных хосту.

Код функции, производящей обработку запроса `GET_DESCRIPTOR`, приведен в листинге 8.21. Основная сложность — разделение пакета на блоки по `EP_CONTROL_LENGTH` байт. При этом если размер пакета окажется кратным этому числу, то необходимо дополнительно передать пустой пакет, уведомляющий хост о завершении передачи данных.

Листинг 8.21. Обработка запроса GET_DESCRIPTOR

```
void GetDescriptor()
{
 data byte    data_to_transfer;
 data uint16  wLength;
 bit          zlp;
 data byte   *pbuffer;

 zlp = FALSE;

 /* Тип запрашиваемого дескриптора находится  */
 /* в старшем байте поля wValue, т. е. байте 3 */
 switch (SetupPacket.b[3])
 {
    /* дескриптор устройства */
    case DEVICE:
    {
      data_to_transfer = sizeof(usb_device_descriptor);
      pbuffer = &(usb_device_descriptor.bLength);
      break;
    }
    /* дескриптор конфигурации */
    case CONFIGURATION:
    {
      data_to_transfer = sizeof(usb_configuration);
      pbuffer = &(usb_configuration.cfg.bLength);
      break;
    }
    default:
    {
      STALL();
      return;
    }
  }
```

```
/* Чтение требуемого числа байтов */
/* (поле wLenght)                */
wLength = wSWAP(SetupPacket.setup.wLength);


/* Требуется больше чем есть? */
if (wLength > data_to_transfer)
{
  /* Если число байт пакета кратно размеру пакета, */
  /* то пакет завершения формируем специально      */
  if ((data_to_transfer % EP_CONTROL_LENGTH) == 0)
  {
   zlp = TRUE;
  }
  else
  {
   zlp = FALSE;
  }
}
else
{
  /* посылаем только требуемое число байт */
  /* иногда это меньше, чем есть реально  */
  /* деление на блоки не требуется        */
  data_to_transfer = (byte)wLength;
}

/* переключение направления нулевой точки */
Usb_set_DIR();

/* шлем пока хватает данных на формирование полного пакета */
/* остаток пакета досылаем потом (если надо)               */
while (data_to_transfer > EP_CONTROL_LENGTH)
{
  /* передача буфера максимальной длины*/
  pbuffer = usb_send_ep0_packet(pbuffer, EP_CONTROL_LENGTH);
```

```
    /* сдвигаем указатель */
    data_to_transfer -= EP_CONTROL_LENGTH;

    /* ждем завершения передачи */
    Usb_set_tx_ready();
    while ((!(Usb_rx_complete())) && (!(Usb_tx_complete())));
    Usb_clear_tx_complete();
    if ((Usb_rx_complete())) /* если передача прервана хостом */
    {
      Usb_clear_tx_ready(); /* завершить передачу */
      Usb_clear_rx();
      return;
    }
  }

  /* посылаем остаточный пакет данных           */
  /* если такой пакет неполный, то он является */
  /* пакетом завершения передачи               */
  pbuffer = usb_send_ep0_packet(pbuffer, data_to_transfer);
  data_to_transfer = 0;
  Usb_set_tx_ready();
  while ((!(Usb_rx_complete())) && (!(Usb_tx_complete())));
  Usb_clear_tx_complete();
  if ((Usb_rx_complete())) /* если передача прервана хостом */
  {
    Usb_clear_tx_ready();
    Usb_clear_rx();
    return;
  }

  /* при необходимости (если все пакеты полные)       */
  /* формируем пакет завершения специально - посылкой */
  /* пакета нулевой длины                             */
  if (zlp == TRUE)
  {
    Usb_set_tx_ready();
```

```
    while ((!(Usb_rx_complete())) && (!(Usb_tx_complete())));
    Usb_clear_tx_complete();
    if ((Usb_rx_complete())) /* если передача прервана хостом */
    {
      Usb_clear_tx_ready();
      Usb_clear_rx();
      Usb_clear_DIR();
      return;
    }
  }

  while ((!(Usb_rx_complete())) && (!(Usb_setup_received())));
  if (Usb_setup_received())
  {
    return;
  }

  if (Usb_rx_complete())
  {
    Usb_clear_DIR();  /* переключение направления 0 точки */
    Usb_clear_rx();
  }

}
```

Собственно сама функция передачи пакета (листинг 8.22) выглядит достаточно очевидно, и, наверно, дополнительно комментирования не требует.

Листинг 8.22. Передача пакета на нулевую конечную точку

```
byte * usb_send_ep0_packet(byte* tbuf, byte data_length)
{
  data int  i;
  data byte b;

  Usb_select_ep(0);
```

```
  /* цикл передачи пакета заданной длины */
  for (i=0; i<data_length; i++)
  {
   b = *tbuf;
   Usb_write_byte(b); /* передача байта */
   tbuf++; /* следующий байт */
  }
  return tbuf;
}
```

Функции обработки остальных запросов выглядят значительно проще. Обработка запроса `GET_CONFIGURATION` сводится к передаче номера текущей конфигурации (листинг 8.23), сохраненного в переменной `usb_configuration_nb`. Этот номер сохраняется при обработке запроса `SET_CONFIGURATION` (листинг 8.24).

Листинг 8.23. Обработка запроса `GET_CONFIGURATION`

```
void GetConfiguration()
{
  Usb_set_DIR();

  /*
    Устройство передает один байт данных,
    содержащий код конфигурации устройства.
  */
  Usb_write_byte(usb_configuration_nb);
  SendDataFromFIFO();

  WaitForOutZeroPacket();
}
```

Кроме сохранения номера текущей конфигурации в обработчике `SET_CONFIGURATION` должна производиться инициализация других конечных точек, если они существуют (см. листинг 8.24).

Листинг 8.24. Обработка запроса `SET_CONFIGURATION`

```
void SetConfiguration()
{
  data byte configuration_number;

  /* читать номер конфигурации */
  /* из младшего байта wValue  */
  configuration_number = SetupPacket.b[2];

  /* если выбрана доступная конфигурация */
  if (configuration_number <= CONF_NB)
  {
    /* сохранить номер конфигурации */
    usb_configuration_nb = configuration_number;
  }
  else
  {
    /* ошибочный запрос - отклоняем */
    STALL();
    return;
  }

  /* фаза status */
  SendInZeroPacket();

/* Конфигурирование других конечных точек */
/* если они есть                          */
}
```

Запрос `GET_STATUS` позволяет определить состояние USB-устройства, интерфейса или конечной точки (*см. разд. 1.2.2*). Состояние, возвращаемое по запросу `GET_STATUS`, представляет собой два байта, один из которых возвращает нужную информацию, а второй зарезервирован и равен нулю. Состояние USB-устройства определяет его реакцию на сигнал побудки и тип питания, состояние интерфейса зарезервировано и всегда равно 0, а состояние конечной точки пока игнорируется. Код обработчиков запроса `GET_STATUS` показан в листинге 8.25.

Листинг 8.25. Обработка запроса GET_STATUS

```
/* ==================================================== */
/*  GetStatusXxx                                        */
/*     - слово состояния устройства                     */
/*     - слово состояния интерфейса                     */
/*     - слово состояния конечной точки                 */
/* (см. раздел 1.2.2)                                   */
/* ==================================================== */
void GetStatusDevice()
{
  Usb_set_DIR();

  /*
    Устройство передает два байта данных:
    В байте 0 используются два бита:
     [1]= 0: устройство игнорирует сигнал побудки
     [0]= 0: устройство получает питание от шины USB
    Байт 1 зарезервирован и всегда равен 0.
  */
  Usb_write_byte(0x00);
  Usb_write_byte(0x00);
  SendDataFromFIFO();

  WaitForOutZeroPacket();
}

void GetStatusInterface()
{
  Usb_set_DIR();

  /*
     Устройство передает два байта данных
     (слово состояния устройства). Оба байта
     зарезервированы и равны нулю.
  */
```

```
  Usb_write_byte(0x00);
  Usb_write_byte(0x00);
  SendDataFromFIFO();

  WaitForOutZeroPacket();
}

void GetStatusEndpoint()
{
  data byte wIndex;

  /* номер конечной точки в младшем байте wIndex */
  wIndex = SetupPacket.b[4] & MSK_EP_DIR;

  Usb_set_DIR();
  /*
     Устройство передает два байта данных.
     Используется только первый бит первого байта:
       0 - конечная точка функционирует нормально;
       1 - передача данных заблокирована.
     Остальные биты зарезервированы и равны 0.
  */
  Usb_write_byte(0x00);
  Usb_write_byte(0x00);
  SendDataFromFIFO();

  WaitForOutZeroPacket();
}
```

С помощью запроса `SET_ADDRESS` хост передает USB-устройству его адрес на шине USB. Начиная с момента получения этого запроса, USB-устройство может отвечать хосту, только если хост обращается по этому адресу. На получение своего адреса USB-устройство должно ответить установкой флага "устройство адресовано", как показано в листинге 8.26.

Листинг 8.26. Обработка запроса SET_ADDRESS

```
void SetAddress()
{
  /* выставить флаг "устройство адресовано" */
  Usb_set_FADDEN();
  SendInZeroPacket();
  /*
    Хост передает адрес устройства в младшей части
    байта wValue.
  */
  Usb_configure_address(SetupPacket.b[2]);
}
```

Теперь снова вернемся к листингу 8.21 и рассмотрим вопрос формирования дескрипторов. Мы уже описывали структуры, используемые для формирования дескрипторов (см. листинг 2.2). Сами дескрипторы мы будем хранить прямо в коде программы, описав переменные со спецификатором `code` (листинг 8.27).

> ***Важно***
>
> При описании дескрипторов следует помнить, что двухбайтовые числа должны описываться "наоборот", т. е. сначала младший байт, затем старший. Например, двухбайтовое поле длины дескриптора размером 25 байтов будет записано как `0x1900`. Для удобства записи можно использовать специальный макрос `wSWAP`:
>
> ```
> #define wSWAP(x)       ((((x)>>8)&0x00FF)|(((x)<<8)&0xFF00))
> ```

Листинг 8.27. Описание дескрипторов устройства и конфигурации

```
/* Типы дескрипторов */
#define DEVICE                0x01
#define CONFIGURATION         0x02
#define STRING                0x03
#define INTERFACE             0x04
#define ENDPOINT              0x05

/* Типы питания устройства */
#define USB_CONFIG_BUSPOWERED     0x80
```

```
#define USB_CONFIG_SELFPOWERED    0x40
#define USB_CONFIG_REMOTEWAKEUP   0x20

/* Константы, составляющие дескриптор устройства */
#define USB_SPECIFICATION     0x1001
#define USB_SPECIFICATION     0x1001
#define DEVICE_CLASS          0x02
#define DEVICE_SUB_CLASS      0
#define DEVICE_PROTOCOL       0
#define EP_CONTROL_LENGTH     32
#define VENDOR_ID             0xEB03 /* Atmel = 03EBh */
#define PRODUCT_ID            0x0921
#define RELEASE_NUMBER        0x0000

/* Дескриптор устройства */
code struct usb_st_device_descriptor usb_device_descriptor =
{
  sizeof(usb_device_descriptor),
  DEVICE,             /* тип дескриптора, =1           */
  USB_SPECIFICATION,  /* спецификация                  */
  DEVICE_CLASS,       /* класс устройства              */
  DEVICE_SUB_CLASS,
  DEVICE_PROTOCOL,    /* протокол USB                  */
  EP_CONTROL_LENGTH,  /* размер пакета 0-й точки       */
  VENDOR_ID,          /* ID производителя и устройства */
  PRODUCT_ID,
  RELEASE_NUMBER,     /* версия устройства             */
  0,                  /* дескр. строки изготовителя    */
  0,                  /* дескр. строки продукта        */
  0,                  /* дескр. строки серийного номера*/
  1                   /* число конфигураций            */
};

/* Константы, составляющие дескриптор конфигурации */
#define CONF_LENGTH           wSWAP(18) /* 9+9= 18 байт */
#define CONF_NB               1
```

```
#define CONF_ATTRIBUTES        USB_CONFIG_BUSPOWERED
#define MAX_POWER              50  /* = 100 мА */

/* Константы для дескриптора интерфейса */
#define INTERFACE0_CLASS        0xFF
#define INTERFACE0_SUB_CLASS    0x00
#define INTERFACE0_PROTOCOL     0xFF

/* Полный дескриптор конфигурации */
code struct
{
  struct usb_st_configuration_descriptor  cfg;
  struct usb_st_interface_descriptor      ifc;
 }
 usb_configuration =
 {
/* CONFIGURATION */
    { 9,
      CONFIGURATION,    /* =2 */
      CONF_LENGTH,      /* длина всех дескрипторов */
      1,                /* число интерфейсов       */
      CONF_NB,          /* номер конфигурации      */
      0,                /* дескр. строки конфигур. */
      CONF_ATTRIBUTES,  /* атрибуты                */
      MAX_POWER         /* максимальный ток        */
    },
/* INTERFACE 0 */
    { 9,
      INTERFACE,        /* =4 */
      0,    /* 0-й интерфейс               */
      0,    /* номер альтернативного инт. */
      0,    /* только 0-я конечная точка  */
      INTERFACE0_CLASS,
      INTERFACE0_SUB_CLASS,
      INTERFACE0_PROTOCOL,
      0     /* дескр. строки интерфейса   */
    }
};
```

Как уже говорилось (*см. разд. 1.2.3*), по запросу дескриптора конфигурации возвращаются дескрипторы конфигурации, интерфейсов и конечных точек. Пока в нашем примере нет конечных точек (кроме нулевой), поэтому мы будем возвращать только два дескриптора. Следует обратить внимание, что поле `wTotalLength` (константа `CONF_LENGTH`) содержит длину всех дескрипторов, возвращаемых по запросу дескриптора конфигурации (в нашем случае возвращается два дескриптора, общая длина которых 18 (9 + 9) байтов).

Теперь остается собрать весь код, откомпилировать (конечно, полный исходный код и полученный HEX-файл вы можете найти на компакт-диске) и загрузить в микроконтроллер. После подачи сигнала сброса Windows должна обнаружить новое устройство (рис. 8.5) и начать поиск и установку драйверов (рис. 8.6).

Рис. 8.5. Windows обнаружила новое устройство

Рис. 8.6. Windows производит поиск и установку драйверов

8.5.2. Добавляем строковые дескрипторы

У нашего USB-устройства есть один недостаток, от которого мы постараемся избавиться: при обнаружении нового устройства Windows может сообщить только общую информацию об устройстве и обобщенное название "USB Device". Конечно, нам хотелось бы, чтобы Windows сообщала какое именно устройство найдено.

Для решения этой проблемы нам надо добавить обработку запросов строковых дескрипторов или, другими словами, выполнить следующие действия:

1) в описании дескриптора устройства (`usb_device_descriptor`) поля индексов дескрипторов строк изготовителя (поле `iManufacturer`), продукта (поле `iProduct`) и серийного номера (поле `iSerialNumber`) надо заменить *уникальными* (в пределах USB-устройства) номера дескрипторов строк (листинг 8.28);
2) добавить описание структур самих дескрипторов и дескриптора идентификатора языка (листинг 8.29);
3) в функцию `UsbGetDescriptor` добавить обработку запроса типа дескриптора `STRING` (листинг 8.30).

Листинг 8.28. Дескриптор устройства со строками идентификации

```
#define LANG_ID             0x00
#define MAN_INDEX           0x01
#define PRD_INDEX           0x02
#define SRN_INDEX           0x03

code struct usb_st_device_descriptor usb_device_descriptor =
{
  ... ... ... ... ... ... ... ... ... ... ... ... ...
  PRODUCT_ID,
  RELEASE_NUMBER,      /* версия устройства            */
  MAN_INDEX,           /* дескр. строки изготовителя   */
  PRD_INDEX,           /* дескр. строки продукта       */
  SRN_INDEX,           /* дескр. строки серийного ном. */
  1                    /* число конфигураций           */
};
```

Листинг 8.29. Описание дескрипторов строк и языка

```
/* Макрос для создания дескриптора строки */
#define structSTRING_DESCRIPTOR(NAME, LEN) struct NAME \
                                           { \
                                             byte   bLength; \
                                             byte   bDescriptorType; \
                                             uint16 wstring[LEN]; \
                                            }
/* Дескриптор строки производителя */
#define USB_MANUFACTURER_NAME {'P'<<8, 'V'<<8, 'A'<<8, \
                               'S'<<8, 'o'<<8, 'f'<<8, 't'<<8}
#define USB_MN_LENGTH         7
code structSTRING_DESCRIPTOR(usb_st_manufacturer, USB_MN_LENGTH)
 usb_manufacturer =
  { sizeof(usb_manufacturer),  STRING, USB_MANUFACTURER_NAME };

/* Дескриптор строки продукта */
#define USB_PRODUCT_NAME      {'P'<<8, 'V'<<8, 'A'<<8, \
                               'S'<<8, 'o'<<8, 'f'<<8, 't'<<8, \
                               ' '<<8,  \
                               't'<<8, 'e'<<8, 's'<<8, 't'<<8, \
                               ' '<<8, \
                               'b'<<8, 'o'<<8, 'a'<<8, 'r'<<8, 'd'<<8}
#define USB_PN_LENGTH         18
code structSTRING_DESCRIPTOR(usb_st_product, USB_PN_LENGTH)
 usb_product =
  { sizeof(usb_product), STRING, USB_PRODUCT_NAME };


/* Дескриптор строки серийного номера */
#define USB_SERIAL_NUMBER     {'1'<<8, '.'<<8, '0'<<8, '.'<<8, '0'<<8}
#define USB_SN_LENGTH         5
code structSTRING_DESCRIPTOR(usb_st_serial_number, USB_SN_LENGTH)
 usb_serial_number =
  { sizeof(usb_serial_number), STRING, USB_SERIAL_NUMBER };
```

```
/* Дескриптор строки идентификатора языка */
#define LANGUAGE_ID           0x0904

code structSTRING_DESCRIPTOR(usb_st_language_descriptor, 1)
 usb_language =
 { sizeof(usb_language), STRING, LANGUAGE_ID };
```

Листинг 8.30. Обработка запроса дескриптора строки

```
void GetDescriptor()
{
... ... ... ... ... ... ... ... ...
switch (SetupPacket.b[3])
 {
... ... ... ... ... ... ... ... ...
    /* дескриптор строки */
    case STRING:
    {
      /* Индекс строки находится в младшем */
      /* байте поля wValue, т. е. байте 2   */
      switch (SetupPacket.b[2])
      {
        case LANG_ID:
        {
          data_to_transfer = sizeof (usb_language);
          pbuffer = &(usb_language.bLength);
          break;
        }
        case MAN_INDEX:
        {
          data_to_transfer = sizeof (usb_manufacturer);
          pbuffer = &(usb_manufacturer.bLength);
          break;
        }
```

```
        case PRD_INDEX:
        {
          data_to_transfer = sizeof (usb_product);
          pbuffer = &(usb_product.bLength);
          break;
        }
        case SRN_INDEX:
        {
          data_to_transfer = sizeof (usb_serial_number);
          pbuffer = &(usb_serial_number.bLength);
          break;
        }
        default:
        {
          STALL();
          return;
        }
      }
      break;
    }
... ... ... ... ... ... ... ... ...
}
```

Загрузив новую программу, можно увидеть, что Windows выдает значительно больше информации о новом устройстве (рис. 8.7, 8.8).

Следует отметить, что при поиске INF-файла эти имена не используются, а поиск подходящего драйвера производится по идентификаторам продукта (PID, Product IDentificator) и производителя (VID, Vendor IDentificator).

Рис. 8.7. Windows показывает имя обнаруженного устройства

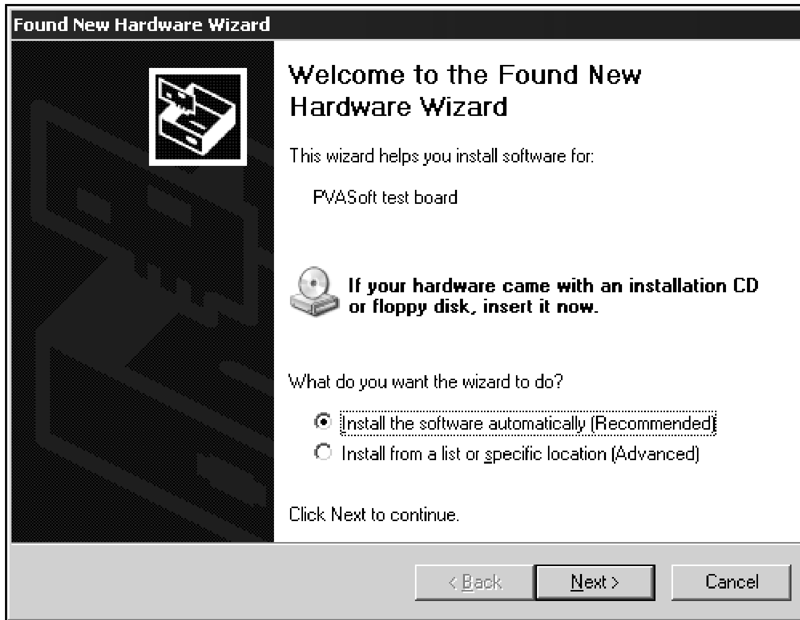


Рис. 8.8. Использование имени устройства при установке драйверов

В заключение раздела еще одно замечание. При описании строковых дескрипторов мы использовали немного странную запись

```
#define USB_PRODUCT_NAME      {'P'<<8, 'V'<<8, 'A'<<8, 'S'<<8,...
```

Необходимость в сдвиге каждого символа строки на 8 битов влево связана с форматом дескриптора — строка дескриптора должна состоять из символов Unicode, т. е. двухбайтовых символов. С помощью строк в формате Unicode можно передавать не только английские, но и русские строки. Проще всего сделать это, набрав нужный текст в редакторе Word и сохранив его в файл типа "Кодированный текст". Затем символы этого файла можно просмотреть в любом шестнадцатеричном редакторе (например, в Far по клавише <F4> в режиме просмотра) и записать в обратном порядке следования байт.

Например, строка "Тестовая плата" будет кодироваться следующим образом:

```
0000000000:  FEFF 0422 0435 0441 ¦ 0442 043E 0432 0430   Тестова
0000000010:  044F 0020 043F 043B ¦ 0430 0442 0430 000D  я плата
0000000020:  000A                ¦
```

Первые два байта (`0xFEFF`) и последние четыре (`0x000D, 0x000A`) являются заголовком файла и символами перевода строки, и мы их отбрасываем, а

остальные записываем в виде последовательности слов с обратным порядком байт.

```
#define USB_PRODUCT_NAME {0x2204, 0x3504, 0x4104, 0x4204, 0x3E04, 0x3204,
0x3004, 0x4F04, 0x2000, 0x3F04, 0x3B04, 0x3004, 0x4204, 0x3004}
```

Запуск программы с новым дескриптором показывает, что мы старались не зря (рис. 8.9).

Рис. 8.9. Устройство с русским именем

8.5.3. Добавление конечных точек

Для добавления конечных точек необходимо добавить дескрипторы этих точек в дескриптор конфигурации, не забыв, кроме того, изменить поле `bNumEndpoints`.

Листинг 8.31 показывает описание структуры дескриптора конечной точки, а листинг 8.32 — дескриптор конфигурации с двумя конечными точками.

Листинг 8.31. Структура дескриптора конечной точки для AT89C5131

```
struct usb_st_endpoint_descriptor
{
  byte   bLength;
  byte   bDescriptorType;
  byte   bEndpointAddress;
  byte   bmAttributes;
  uint16 wMaxPacketSize;
  byte   bInterval;
};
```

Листинг 8.32. Дескриптор конфигурации с двумя конечными точками

```
#define EP_0_ADDRESS      (0|CONTROL)

/* первая конечная точка */
#define EP_1_CONFIG       (BULK|EP_CONFIG_IN)  /* конфигурация */
```

```
#define EP_1_ADDRESS      (2|EP_DIRECT_IN)     /* адрес         */
#define EP_1_ATTRIBUTES   BULK                 /* атрибуты      */

/* вторая конечная точка */
#define EP_2_CONFIG       (BULK|EP_CONFIG_OUT) /* конфигурация */
#define EP_2_ADDRESS      (2|EP_DIRECT_OUT)    /* адрес         */
#define EP_2_ATTRIBUTES   BULK                 /* атрибуты      */
code struct
{
  struct usb_st_configuration_descriptor  cfg;
  struct usb_st_interface_descriptor      ifc;
  struct usb_st_endpoint_descriptor       ep1;
  struct usb_st_endpoint_descriptor       ep2;
 }
 usb_configuration =
 {
/* CONFIGURATION */
    { 9,
... ... ... ... ... ... ... ... ... ... ... ... ... ...
    },
/* INTERFACE 0 */
    { 9,
... ... ... ... ... ... ... ... ... ... ... ... ... ...
      2,               /* конечные точки  (кроме 0)   */
... ... ... ... ... ... ... ... ... ... ... ... ... ...
    },
/* Дескриптор первой конечной точки */
    {
      7,
      ENDPOINT,          /* дескриптор ENDPOINT        */
      EP_1_ADDRESS,      /* номер конечной точки       */
      EP_1_ATTRIBUTES,   /* атрибуты конечной точки    */
      wSWAP(64),         /* максимальный размер пакета */
      0                  /* частота опроса             */
    },
```

```
/* Дескриптор второй конечной точки */
    {
      7,
      ENDPOINT,           /* дескриптор ENDPOINT         */
      EP_2_ADDRESS,       /* номер конечной точки        */
      EP_2_ATTRIBUTES,    /* атрибуты конечной точки     */
      wSWAP(64),          /* максимальный размер пакета */
      0                   /* частота опроса              */
    }
   };
```

Мы специально скрыли уже известные нам поля дескриптора, выделив только изменившиеся. Как мы уже описывали (*см. разд. 1.2.3*), главными полями дескриптора конечной точки являются поля номера и атрибутов. Для формирования значения этих полей используются константы, показанные в листинге 8.33.

Листинг 8.33. Константы для описания конечных точек

```
/* Типы конечных точек  */
#define CONTROL              0x00
#define ISOCHRONOUS          0x01
#define BULK                 0x02
#define INTERRUPT            0x03
/* для вычисления номера конечной точки */
#define EP_DIRECT_OUT        0x00
#define EP_DIRECT_IN         0x80
/* для вычисления конфигурации точки */
#define EP_CONFIG_OUT        0x00
#define EP_CONFIG_IN         0x04
```

Такое несколько усложненное описание адреса конечной точки связано со структурой регистра `UEPCONX`, в котором для описания типа точки используются биты [1:0], а бит 2 используется для указания направления передачи, структурой поля `bmAttributes`, в котором для описания типа используются биты [1:0] и структурой номера конечной точки, в котором направление конечной точки задается битом 7. Так, для формирования описания конечной

точки типа Bulk с направлением передачи IN мы используем следующие константы:

```
// Конфигурация - биты [1:0] и 2
#define EP_1_CONFIG       (BULK | EP_CONFIG_IN)
// Номер конечной точки — биты [3:0] и 7
#define EP_1_ADDRESS      (2 | EP_DIRECT_IN)
// Атрибуты — биты [1:0]
#define EP_1_ATTRIBUTES    BULK
```

Инициализация всех конечных точек, кроме нулевой, производится после получения запроса `SET_CONFIGURATION` (листинг 8.34).

Листинг 8.34. Инициализация конечных точек

```
void SetConfiguration()
{
... ... ... ... ... ... ... ... ... ... ... ...
  /* фаза status */
  SendInZeroPacket();
  /* Конфигурирование других конечных точек */
  usb_configure_endpoint(1 , BULK_IN);
  usb_reset_endpoint(1);
  usb_configure_endpoint(2 , BULK_OUT);
  usb_reset_endpoint(2);
  usb_configure_endpoint(3 , INTERRUPT_IN);
  usb_reset_endpoint(3);
... ... ... ... ... ... ... ... ... ... ... ...
}
```

Конечно, можно описывать конечные точки и просто константами, не используя макроподстановок:

```
#define ENDPOINT_NB_1         0x81
#define EP_CONFIG_1           0x81
#define EP_ATTRIBUTES_1       0x02  /* bulk */
#define EP_SIZE_1             ((unsigned int)32) << 8
#define EP_INTERVAL_1         0x00
```

8.6. Загрузка программы

Упрощенно говоря, внутри процессора существуют два загрузчика: пользовательский и аппаратный (HBL, Hardware BootLoader). Пользовательский загрузчик позволяет запускать программы, записанные в память микроконтроллера, а аппаратный позволяет осуществить запись самой программы.

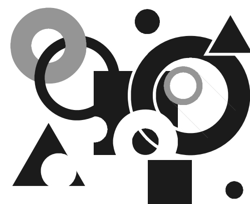
Очевидная последовательность действий такова: стартовать аппаратный загрузчик, записать программу в микроконтроллер, стартовать записанную программу.

Для старта HBL требуется выполнить следующую последовательность действий:

1) отключить прибор от шины USB, разомкнув перемычку P2 (линия VREF);
2) удерживая кнопки K3 (линия Reset) и K2 (линия PSEN), подключить прибор к шине USB, замкнув перемычку P2;
3) отпустить кнопку K3;
4) отпустить кнопку K2.

Если HBL успешно стартовал, Windows обнаружит новое устройство со следующими характеристиками:

- Vendor ID — 03EB;
- Product ID — 2FFD;
- при установке драйверов FLIP новое устройство будет иметь имя "Jungo AT89C5131" (см. рис. 3.4), а при установке драйверов ER-Tronik — имя "ATMEL 89C5131 Bootloader" (см. рис. 3.8).



Глава 9

Реализация класса CDC

В этой главе мы создадим простое CDC-устройство и разберемся, как производится обмен данными с ним из Windows.

9.1. Реализация CDC

Мы будем использовать наиболее простое абстрактное устройство (*см. разд. 5.3*):

- коммуникационный интерфейс, значения полей:
  - `bInterfaceClass` — `0x02`;
  - `bInterfaceSubClass` — `0x02`;
  - `bInterfaceProtocol` — `0x01`;
- интерфейс данных, значения полей:
  - `bInterfaceClass` — `0x0A`;
  - `bInterfaceSubClass` — `0x00`;
  - `bInterfaceProtocol` — `0x00`.

В качестве примера мы создадим CDC-устройство, реализующее обмен виртуального и реального последовательных портов (рис. 9.1).

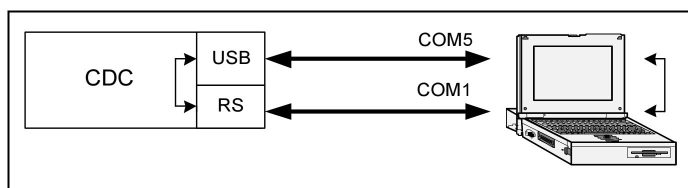


Рис. 9.1. Схема примера CDC-устройства

9.2. Дескрипторы устройства

Описание дескриптора устройства и дескриптора конфигурации показано в листинге 9.1.

Листинг 9.1. Дескриптор устройства и дескриптор конфигурации CDC-устройства

```
/* Константы для формирования дескриптора */
#define USB_SPECIFICATION     0x1001
#define DEVICE_CLASS          0x02
#define DEVICE_SUB_CLASS      0
#define DEVICE_PROTOCOL       0
#define EP_CONTROL_LENGTH     32
#define VENDOR_ID             0xEB03
#define PRODUCT_ID            0x0920
#define RELEASE_NUMBER        0x0000
#define NB_CONFIGURATION      1

/* Идентификаторы строк */
#define LANG_ID              0x00
#define MAN_INDEX            0x01
#define PRD_INDEX            0x02
#define SRN_INDEX            0x03

/* Дескриптор устройства */
code struct usb_st_device_descriptor usb_device_descriptor =
{
  sizeof(usb_device_descriptor), /* размер дескриптора */
  DEVICE,                        /* тип дескриптора    */
  USB_SPECIFICATION,
  DEVICE_CLASS,
  DEVICE_SUB_CLASS,
  DEVICE_PROTOCOL,
  EP_CONTROL_LENGTH,
  VENDOR_ID,
  PRODUCT_ID,
```

```
  RELEASE_NUMBER,
  MAN_INDEX,
  PRD_INDEX,
  SRN_INDEX,
  NB_CONFIGURATION
};

/* Дескриптор конфигурации. Состоит из нескольких */
/* дескрипторов, передаваемых последовательно     */
code struct
{
  struct usb_st_configuration_descriptor  cfg;
  struct usb_st_interface_descriptor      ifc0;
  byte                                    CDC[19];
  struct usb_st_endpoint_descriptor       ep3 ;
  struct usb_st_interface_descriptor      ifc1;
  struct usb_st_endpoint_descriptor       ep1 ;
  struct usb_st_endpoint_descriptor       ep2 ;
}
  usb_configuration =
  {
    {
      9,
      CONFIGURATION,
      CONF_LENGTH,
      NB_INTERFACE,
      CONF_NB,
      CONF_INDEX,
      CONF_ATTRIBUTES,
      MAX_POWER
    },
    {
      9,
      INTERFACE,
      INTERFACE0_NB,
      ALTERNATE0,
```

```
    NB_ENDPOINT0,
    INTERFACE0_CLASS,
    INTERFACE0_SUB_CLASS,
    INTERFACE0_PROTOCOL,
    INTERFACE0_INDEX
  },
  {
    0x05, 0x24, 0x00, 0x10, 0x01, /* заголовочный дескриптор */
    0x05, 0x24, 0x01, 0x03, 0x01, /* дескриптор режима команд */
    0x04, 0x24, 0x02, 0x06, /* дескриптор абстр. устройства */
    0x05, 0x24, 0x06, 0x00, 0x01 /* дескриптор группирования */
  },
  {
    7,
    ENDPOINT,
    ENDPOINT_NB_3,
    EP_ATTRIBUTES_3,
    EP_SIZE_3,
    EP_INTERVAL_3
  },
  {
    9,
    INTERFACE,
    INTERFACE1_NB,
    ALTERNATE1,
    NB_ENDPOINT1,
    INTERFACE1_CLASS,
    INTERFACE1_SUB_CLASS,
    INTERFACE1_PROTOCOL,
    INTERFACE1_INDEX
  },
  {
    7,
    ENDPOINT,
    ENDPOINT_NB_1,
```

```
      EP_ATTRIBUTES_1,
      EP_SIZE_1,
      EP_INTERVAL_1
    },
    {
      7,
      ENDPOINT,
      ENDPOINT_NB_2,
      EP_ATTRIBUTES_2,
      EP_SIZE_2,
      EP_INTERVAL_2
    }
  };
```

Новым в этом листинге является дескриптор CDC, состоящий из четырех дескрипторов (*см. разд. 5.3*).

Согласно спецификации CDC, устройство будет иметь четыре конечные точки:

- конечная точка 0 (обязательная для USB-устройств, *см. разд. 1.1.12*);
- конечная точка 1, тип Bulk IN;
- конечная точка 2, тип Bulk OUT;
- конечная точка 3, тип Interrupt IN.

9.2.1. Инициализация конечных точек

Инициализация конечных точек (напомним, что она будет вызываться при получении запроса `SET_CONFIGURATION`) показана в листинге 9.2.

Листинг 9.2. Инициализация конечных точек CDC-устройства

```
usb_configure_endpoint(1, BULK_IN);
usb_reset_endpoint(1);
usb_configure_endpoint(2, BULK_OUT);
usb_reset_endpoint(2);
usb_configure_endpoint(3, INTERRUPT_IN);
usb_reset_endpoint(3);
```

9.2.2. Обработка CDC-запросов

Кроме стандартных запросов нам нужно будет обрабатывать специфические запросы CDC (*см. разд. 5.3.3*). Модифицированная процедура `UsbControlPacketProcessed()` показана в листинге 9.3.

Листинг 9.3. Добавляем обработку запросов CDC

```
#define GET_LINE_CODING           0xA121
#define SET_LINE_CODING           0x2120
#define SET_CONTROL_LINE_STATE    0x2122
#define SEND_BREAK                0x2123
#define SEND_ENCAPSULATED_COMMAND 0x2100
#define GET_ENCAPSULATED_COMMAND  0xA101

void UsbControlPacketProcessed()
{
... ... ... ... ... ... ... ... ...
  switch (SetupPacket.wRequest)
  {
... ... ... ... ... ... ... ... ...
    /* Обработка специальных CDC-запросов */
    case SET_LINE_CODING:
      cdc_set_line_coding();
      break;
    case GET_LINE_CODING:
      cdc_get_line_coding();
      break;
    case SET_CONTROL_LINE_STATE:
      cdc_set_control_line_state();
      break;
    case SEND_BREAK:
      cdc_send_break();
      break;
    case SEND_ENCAPSULATED_COMMAND:
      cdc_send_encapsulated_command();
      break;
```

```
    case GET_ENCAPSULATED_COMMAND:
      cdc_get_encapsulated_command();
      break;
    default:
     STALL();
     break;
  }
}
```

Запрос `cdc_get_line_coding()` возвращает, а `cdc_set_line_coding()` принимает от хоста семь байтов данных, содержащих информацию о настройках канала связи. Формат этого пакета был описан ранее (*см. разд. 5.3.3*), а пример кода показан в листинге 9.4.

Листинг 9.4. Обработка запросов `cdc_get_line_coding` и `cdc_set_line_coding`

```
// Буфер для хранения конфигурации линии
data byte line_coding[7];

// Обработка запроса SET_LINE_CODING
void cdc_set_line_coding()
{
  WaitForFillFIFO();

  line_coding[0] = Usb_read_byte();
  line_coding[1] = Usb_read_byte();
  line_coding[2] = Usb_read_byte();
  line_coding[3] = Usb_read_byte();
  line_coding[4] = Usb_read_byte();
  line_coding[5] = Usb_read_byte();
  line_coding[6] = Usb_read_byte();
  EndReadData();

  SendInZeroPacket();
}

// Обработка запроса GET_LINE_CODING
void cdc_get_line_coding()
```

```
{
  Usb_set_DIR();

  Usb_write_byte(line_coding[0]);
  Usb_write_byte(line_coding[1]);
  Usb_write_byte(line_coding[2]);
  Usb_write_byte(line_coding[3]);
  Usb_write_byte(line_coding[4]);
  Usb_write_byte(line_coding[5]);
  Usb_write_byte(line_coding[6]);
  SendDataFromFIFO();

  WaitForOutZeroPacket();
}
```

Запрос `SEND_BREAK`, генерируемый вызовами `SetCommBreak` и `ClearCommBreak`, обрабатывается очень просто (листинг 9.5). Если параметр `wValue` равен `0xFFFF`, это означает замораживание передачи, а значение `wValue`, равное `0x0000`, означает восстановление активности. Другие значения `wValue` означают замораживание передачи на определенное количество мс, но такие значения не поддерживаются Windows API.

Листинг 9.5. Обработка запроса `cdc_send_break`

```
void cdc_send_break()
{
  SendInZeroPacket();
  // можно добавить обработку wValue
}
```

9.2.3. Конфигурирование RS-порта и CDC-линии

Разбирать тонкости конфигурирования последовательного портра для AT89C5131 мы не будем. Интересующиеся могут посмотреть в спецификацию процессора. Мы приведем лишь листинг инициализации (листинг 9.6). Здесь же мы будем инициализировать буфер, содержащий параметры CDC-линии для запроса `GET_LINE_CODING`.

Листинг 9.6. Инициализация последовательного порта AT89C5131 и CDC-линии

```
void CdcInit()
{
 /* Конфигурирование RS232 */
 SCON    = MSK_UART_8BIT;
 CKCON0 |= MSK_X2;
 PCON   |= MSK_SMOD1;
 BDRCON |= 2;
 BDRCON |= 0x1C;
 BDRCON &= ~1;
 BRL     = 100;
 SCON   |= MSK_UART_ENABLE_RX|MSK_UART_TX_READY|MSK_UART_ENABLE_RX;

 /* Параметры CDC-линии */
 line_coding[0] = 0x00;
 line_coding[1] = 0xC2;
 line_coding[2] = 0x01;
 line_coding[3] = 00;
 line_coding[4] = 0;    /* стоп-бит  */
 line_coding[5] = 0;    /* четность */
 line_coding[6] = 8;    /* размер байта */
}
```

9.2.4. Прием и передача данных

Прием данных от хоста осуществляется с помощью конечной точки 2 (Bulk OUT). В нашем примере мы будем получать данные от хоста и передавать их на последовательный порт микропроцессора. Реализация этой процедуры показана в листинге 9.7.

Листинг 9.7. Прием данных от хоста

```
void main()
{
... ... ... ... ... ... ... ...
for (;;)
 {
... ... ... ... ... ... ... ...
```

```
   if (Usb_endpoint_interrupt())
   {
... ... ... ... ... ... ... ...
    /* получение данных со 2 конечной точки */
    Usb_select_ep(2);
    bcount = UBYCTLX;
    if (bcount > 0)
    {
     for (i=0; i<bcount;i++)
     {
       SendRS232(Usb_read_byte());
     }
    }
    Usb_clear_rx();
   }
... ... ... ... ... ... ... ...
  } /* end if connected */
 } /* end for ;; */
}
// Передача данных по последовательному порту AT89C5131
void SendRS232(char ch)
{
 while(!TI);
 TI=0;
 SBUF = ch;
 while(TI);
}
```

Передача данных осуществляется с помощью конечной точки 1 (Bulk IN) и состоит из следующих шагов (листинг 9.8):

- заполнение буфера FIFO конечной точки;
- передача данных;
- передача дополнительного пустого пакета, если размер данных совпадает с максимальным размером пакета конечной точки.

Аналогичную процедуру мы уже рассматривали при передаче дескрипторов (см. листинг 8.21).

Листинг 9.8. Передача данных хосту

```
void main()
{
... ... ... ... ... ... ... ...
 for (;;)
 {
  main_task();
... ... ... ... ... ... ... ...
   /* обнаружено прерывание от конечной точки */
   if (Usb_endpoint_interrupt())
   {
... ... ... ... ... ... ... ...
    Usb_select_ep(1);
    switch (end_point1_ready)
    {
     case 1:
     {
       if (Usb_tx_complete())
       {
        Usb_clear_tx_complete();
        Usb_set_tx_ready();
        end_point1_ready = 2;
       }
       break;
     }
     case 2:
     {
       if (Usb_tx_complete())
       {
        Usb_clear_tx_complete();
        end_point1_ready = 0;
       }
       break;
     }
    }
```

```
... ... ... ... ... ... ... ...
   }/* if interrupt */
  } /* end if connected */
 } /* end for ;; */
}

void main_task()
{
 /* устройство не готово к передаче данных */
 if ((usb_configuration_nb == 0) || (Usb_suspend()))
  return;

 if (end_point1_ready == 0) /* предыдущий пакет отправлен */
 {
   Usb_select_ep(1);
   for (i=0; i<32; i++) Usb_write_byte(i);
   Usb_set_tx_ready();
   if (i==32) end_point1_ready= 2; else end_point1_ready= 1;
 }
}
```

9.3. Установка драйвера

Для установки CDC-устройства потребуется специальный INF-файл, содержащий описание драйвера, т. е. указание использовать драйверы виртуального последовательного порта. Файл для Windows 2000 и XP приведен в листинге 9.9. Этот файл можно найти на сайте Atmel. Непосредственно сам файл драйвера имеет имя usbser.sys.

Листинг 9.9. Пример INF-файла для CDC-устройства

```
; Установочный файл для Windows 2000 и XP
; для CDC-устройства на основе AT89C5131
[Version]
Signature="$Windows NT$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
```

```
Provider=%ATMEL%
LayoutFile=layout.inf
DriverVer=10/15/1999,5.0.2153.1

[Manufacturer]
%ATMEL%=ATMEL

[ATMEL]
%ATMEL_CDC%=Reader, USB\VID_03EB&PID_2009


[Reader_Install.NTx86]
;Windows2000

[DestinationDirs]
DefaultDestDir=12
Reader.NT.Copy=12

[Reader.NT]
CopyFiles=Reader.NT.Copy
AddReg=Reader.NT.AddReg

[Reader.NT.Copy]
usbser.sys

[Reader.NT.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,usbser.sys
HKR,,EnumPropPages32,,"MsPorts.dll,SerialPortPropPageProvider"

[Reader.NT.Services]
AddService = usbser, 0x00000002, Service_Inst

[Service_Inst]
DisplayName = %Serial.SvcDesc%
ServiceType = 1 ; SERVICE_KERNEL_DRIVER
```

```
StartType = 3 ; SERVICE_DEMAND_START
ErrorControl = 1 ; SERVICE_ERROR_NORMAL
ServiceBinary = %12%\usbser.sys
LoadOrderGroup = Base

[Strings]
ATMEL = "ATMEL, Inc."
ATMEL_CDC = "AT89C5131 CDC USB to UART MGM"
Serial.SvcDesc = "USB Serial emulation driver"
```

Файл для Windows 9*x*, а также некоторые рекомендации по использованию CDC можно найти на сайте **http://www.at91.com**, в частности, в файле **http://www.at91.com/pdf/applicationNotes/USB%20Application%20Note.pdf**.

9.4. Программирование обмена данными с CDC-устройством на языке Delphi

Работа с CDC-устройством фактически сводится к обращению к последовательному порту. Единственное отличие — этот порт виртуальный, поэтому некоторые функции будут работать не совсем привычным образом или могут быть не реализованы вовсе.

Подробное освещение всех функций работы с последовательным портом дано в [2]. В рамках этой книги мы рассмотрим только асинхронный обмен с CDC-устройством и некоторые тонкости, связанные с тем, что порт является виртуальным.

Общая схема работы выглядит следующим образом:

- открытие порта в асинхронном режиме;
- задание параметров порта (скорость передачи, четность и т. д.);
- создание потока для чтения данных;
- передача полученных данных в основной поток для обработки и отображения;
- асинхронная запись данных в порт возможна в любой момент, пока порт открыт;
- закрытие порта при завершении работы программы.

Для открытия порта используется функция `CreateFile` (*см. разд. 14.1*), возвращающая дескриптор (handle) открытого порта. Задание параметров виртуального порта, вообще говоря, процедура не обязательная, если CDC-

устройство не обрабатывает запрос `SET_LINE_CODING`. Пример открытия порта показан в листинге 9.10[1].

Листинг 9.10. Открытие порта CDC-устройства

```
FHandle : THandle;   {дескриптор порта}

{Открытие последовательного порта }
 FHandle:= CreateFile(
  PChar(@FPortName[1]), {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 }
);
```

Особо следует обратить внимание на флаг `FILE_FLAG_OVERLAPPED`, указывающий, что порт будет открыт в асинхронном режиме, и, следовательно, все остальные операции с портом также следует проводить в асинхронном режиме.

Для задания параметров порта используются стукрута `DCB` (*см. разд. 15.5*) и функция `SetCommState` (*см. разд. 15.16*). Поля этой структуры описаны в справочной части книги, а вызов установки параметров выглядит очень просто:

```
SetCommState(FHandle, DCB);
```

Кроме того, для формирования этой структуры можно использовать стандартный диалог Windows (рис. 9.2), вызвав функцию `CommConfigDialog`:

```
Var ComCfg : TCommConfig;
Begin
 ZeroMemory(@ComCfg, SizeOf(TCommConfig));
 ComCfg.dwSize:= SizeOf(TCommConfig);
 ComCfg.dcb   := DCB;
 Result:= CommConfigDialog(PChar(@FPortName[1]), 0, ComCfg);
```

[1] Примеры для языка C++ можно найти в справочной части книги.

```
 If Result then begin
  DCB:= ComCfg.dcb;
End;
```

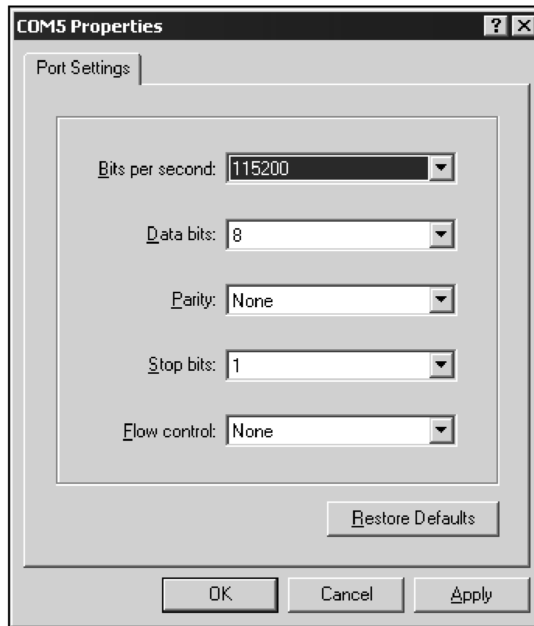


Рис. 9.2. Стандартный диалог Windows настройки порта

Для чтения данных в асинхронном режиме правильнее всего производить опрос порта в отдельном потоке, чтобы не останавливать выполнение основной программы. Реализация основной функции потока чтения показана в листинге 9.11.

Листинг 9.11. Реализация основной функции потока чтения данных

```
{Основная функция потока}
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}
    LineStatus : Cardinal; {текущее состояние линий порта}
```

```
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 isConnected 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);
        { Число полученных, но еще не прочитанных байт}
        AvaibleBytes:= CurrentState.cbInQue;
        { Проверка числа доступных байт}
        If AvaibleBytes > 0 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;
```

Порядок работы потока следующий:

1) создается *событие* (event) для ожидания данных;
2) устанавливается *маска ожидаемых событий* (`SetCommMask`);
3) пока поток активен, выполняется процесс ожидания и чтения данных, если они появились:
   - вызов `WaitForSingleObject` после `WaitCommEvent` вернет управление потоку, если произошко одно из событий, заданных маской `SetCommMask`;

- получение результата асинхронной операции производится с помощью вызова функции GetOverlappedResult;
- по значению переменной Mask (она передавалась как параметр в функцию WaitCommEvent) можно узнать какие именно события произошли;
- вызов функции GetState возвращает структуру описания состояния порта (мы разберем ее чуть позже);
- поле cbInQue струкруты CurrentState позволяет узнать сколько байтов данных доступно для чтения;
- в соответствии с доступным числом байтов мы производим асинхронное (т. к. указан параметр @ReadOL) чтение данных порта;
- после чтения порции данных мы вызываем передачу полученных данных в основной поток (при этом мы используем вызов функции Synchronize для синхронизации потока чтения с основным потоком);

4) при завершении работы потока дескриптор события закрывается, а маска событий сбрасывается.

Функция GetState, которую мы использовали для получения количества доступных байтов, на самом деле вызывает функцию WinAPI ClearCommError (*см. разд. 15.9*), которая выполняет сброс ошибок и, одновременно, возвращает состояние порта:

```
{Возвращает структуру состояния порта и код ошибок}
Function TComPort.GetState(var CodeError : Cardinal) : TComStat;
Begin
 ClearCommError(FHandle, CodeError, @Result);
End;
```

Передача данных в устройство сводится к асинхронному вызову функции WriteFile (листинг 9.12).

Листинг 9.12. Передача данных в устройство

```
{Передача буфера данных}
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;
```

Для простоты все функции, относящиеся к работе с портом, мы вынесли в отдельный класс, показанный в листинге 9.13, а листинг 9.14 содержит пример работы с этим классом. Кроме тех функций, которые мы уже разобрали, этот класс содержит функции получения списка всех портов `EnumComPorts`, функции управления линиями DTR и RTS, а также события, срабатывающие на изменения состояния линий CTS, DSR и RING.

Листинг 9.13. Класс для работы с последовательным портом

```
unit ComPort;

interface

uses
  Windows, SysUtils, Classes, Forms;

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
);

{тип события при получении байта}
TReadPacketEvent = procedure (const Packet : Pointer; const Size :
Integer; const ErrCode: Cardinal) of object;
{событие "изменение состояния CTS"}
TCTSChangeEvent = procedure (const CTS : Boolean) of object;
{событие "изменение состояния DSR"}
TDSRChangeEvent = procedure (const DSR : Boolean) of object;
{событие "изменение состояния RING"}
TRINGChangeEvent = procedure (const RING : Boolean) of object;

TComPort = class; {опережающее описание}

{читающий поток}
TReadThread = class(TThread)
  FOwner    : TComPort;
  FInBuffer : Pointer; {входной буфер}
  FErrCode  : Cardinal;{сохраняет код ошибок}
  FCount    : Integer; {реально прочитанное число байтов}
  FCTS      : Boolean; {состояние CTS}
  FDSR      : Boolean; {состояние DSR}
  FRING     : Boolean; {состояние RING}
 Protected
  Procedure Execute; override;
  Procedure DoReadPacket;
```

```
   Procedure DoChangeCTS;
   Procedure DoChangeDSR;
   Procedure DoChangeRING;
  Public
   Constructor Create(AOwner : TComPort);
   Destructor  Destroy; override;
 End;

 TComProp = class(TPersistent)
  Private
   FBaudRate    : TBaudRate; {скорость обмена (бод)}
   FByteSize    : TByteSize; {число бит в байте}
   FParity      : TParity;   {четность}
   FStopbits    : TStopbits; {число стоп-бит}
   function  GetDCB: TDCB;
   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
    FPortName    : String;    {имя порта}
    FHandle      : THandle;    {дескриптор порта}
    FOnReadPacket: TReadPacketEvent;{событие "получение пакета"}
    FReadThread  : TReadThread; {читающий поток}
    FInBufSize   : Cardinal; {размер входной очереди }
```

```
    FComProp     : TComProp; {свойства порта}
    FOnCTSChangeEvent  : TCTSChangeEvent;  {изменение CTS  }
    FOnDSRChangeEvent  : TDSRChangeEvent;  {изменение DSR  }
    FOnRINGChangeEvent : TRINGChangeEvent; {изменение RING }
    Procedure DoOpenPort; {открытие порта}
    Procedure DoClosePort; {закрытие порта}
  Private
    procedure SetReadActive(const Value: Boolean);
    function  GetReadActive: Boolean;
    procedure SetInBufSize(const Value: Cardinal);
  Public
    Constructor Create(AOwner : TComponent); override;
    Destructor  Destroy; override;
  Public
    {Открывает/закрывает порт}
    Procedure Open(PortName : String);
    Procedure Close;
    {Возвращает True, если порт открыт}
    Function  IsConnected : 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;
    {Передает буфер. В случае успеха возвращает True}
    Function  WriteBuffer(const P : PChar; const Size : Integer) :
    Boolean;
    {Передает строку. В случае успеха возвращает True}
    Function  WriteString(const S : String) : Boolean;
    {Диалог настройки параметров порта}
    Function  CommDialog : Boolean;
    {Применение параметров порта}
    Procedure ApplyComSettings;
    // Break
    Procedure SetBreak;
    Procedure ClrBreak;
    Procedure SetupBuffer;
```

```
  Public
    {Активность чтения порта}
    Property ReadActive : Boolean  read GetReadActive write
    SetReadActive;
  Published
    {Параметры порта}
    Property ComProp : TComProp read FComProp write FComProp;
    {Размер входного буфера}
    Property InBufSize  : Cardinal read FInBufSize write SetInBufSize;
   Published
    {Событие, вызываемое при получении байта}
    Property OnReadPacket : TReadPacketEvent read FOnReadPacket write
    FOnReadPacket;
    {Событие, вызываемое при изменении состояния CTS}
    Property OnCTSChangeEvent : TCTSChangeEvent read FOnCTSChangeEvent
    write FOnCTSChangeEvent;
    {Событие, вызываемое при изменении состояния DSR}
    Property OnDSRChangeEvent : TDSRChangeEvent read FOnDSRChangeEvent
    write FOnDSRChangeEvent;
    {Событие, вызываемое при изменении состояния RING}
    Property OnRINGChangeEvent : TRINGChangeEvent read FOnRINGChangeEvent
    write FOnRINGChangeEvent;
   Public {Дополнительные функции}
    {Получение базового адреса}
    Function GetBaseAddress : Word;
    {Управление линией DTR}
    Procedure ToggleDTR(State : Boolean);
    {Управление линией RTS}
    Procedure ToggleRTS(State : Boolean);
    {Перечисление всех COM портов}
    Class procedure EnumComPorts(Ports: TStrings);
    Class procedure EnumComPortsEx(Ports: TStrings);
 End;

implementation

uses
  Dialogs, WinSpool, Math;
```

```
{Добавление ресурса с иконкой для палитры компонент}
{$R *.dcr}

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);
```

```
 FComProp  := TComProp.Create; {свойства порта}
 FHandle:= INVALID_HANDLE_VALUE;
 FReadThread:= nil;
 FInBufSize := 10;
End;

Destructor  TComPort.Destroy;
Begin
 {Закрываем порт и соединение}
 DoClosePort;
 FComProp.Free;
 Inherited Destroy;
End;

Procedure TComPort.ApplyComSettings;
Begin
 If not IsConnected 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(@FPortName[1]), 0, ComCfg);
 If Result then begin
  FComProp.DCB:= ComCfg.dcb;
  ApplyComSettings;
 End;
End;

{Изменение размера входного буфера}
procedure TComPort.SetInBufSize(const Value: Cardinal);
```

```
begin
 {Разрешаем менять только при выключенном соединении}
 If IsConnected then Exit;
 {Нельзя задать нулевое или неверное значение размера}
 If Value <= 0 then Exit;
 {Запоминаем новый размер буфера}
 FInBufSize:= Value;
end;

{Установка соединения (дублирует Connect:= True)}
Procedure TComPort.Open(PortName : String);
Begin
 FPortName:= PortName;
 DoOpenPort;
End;

{Закрытие соединения (дублирует Connect:= False)}
Procedure TComPort.Close;
Begin
 DoClosePort;
End;

Function  TComPort.IsConnected : Boolean;
Begin
 Result:= (FHandle <> INVALID_HANDLE_VALUE);
End;

{открытие порта}
Procedure TComPort.DoOpenPort;
Begin
 If IsConnected then Exit;
 {Запрещаем подключение в среде разработки}
 If csDesigning in ComponentState then Exit;

 {Открытие последовательного порта }
 FHandle:= CreateFile(
```

```
            PChar(@FPortName[1]), {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 }
           ) ;
 {Если ошибка открытия порта - выход}
 If not IsConnected then Exit;
 {Задание параметров порта}
 ApplyComSettings;
 {Создание читающего потока}
 If not Assigned(FReadThread) then
  FReadThread:= TReadThread.Create(Self);
End;

{закрытие порта}
Procedure TComPort.DoClosePort;
Begin
 If not IsConnected then Exit;
 {Замораживаем поток чтения}
 ReadActive:= False;
 {Уничтожение читающего потока}
 If Assigned(FReadThread) then begin
  FReadThread.FreeOnTerminate:= True;
  FReadThread.Terminate;
 End;
 FReadThread:= nil;
 {Освобождение дескриптора порта}
 CloseHandle(FHandle);
 {Сброс дескриптора порта}
 FHandle:= INVALID_HANDLE_VALUE;
End;
```

```
{Возвращает структуру состояния порта и код ошибок}
Function TComPort.GetState(var CodeError : Cardinal) : TComStat;
Begin
 ClearCommError(FHandle, CodeError, @Result);
End;

{Передача строки}
Function  TComPort.WriteString(const S : String) : Boolean;
Begin
 Result:= WriteBuffer(PChar(S), Length(S));
End;

{Передача буфера данных}
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;

{Возвращает состояние модема}
Function TComPort.GetModemState : Cardinal;
Begin
 GetCommModemStatus(FHandle, Result);
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;

{ недокументированный код функции - получение базового адреса в dx}
Function TComPort.GetBaseAddress : Word;
Begin
 EscapeCommFunction(FHandle, 10);
 Asm
  mov  Result, dx
 End;
End;
```

```
{Управление линией DTR}
Procedure TComPort.ToggleDTR(State : Boolean);
const Funcs: Array[Boolean] of Cardinal = (CLRDTR, SETDTR);
Begin
 If IsConnected then
  EscapeCommFunction(FHandle, Funcs[State]);
End;

{Управление линией RTS}
Procedure TComPort.ToggleRTS(State : Boolean);
const Funcs: Array[Boolean] of Cardinal = (CLRRTS, SETRTS);
Begin
 If IsConnected then
  EscapeCommFunction(FHandle, Funcs[State]);
End;

{ 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 begin
     Delete(TempStr, Length(TempStr), 1); // удалить знак ":"
     Ports.Add(TempStr);
    End;
   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;

{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 CurrentState : TComStat;
    AvaibleBytes, ErrCode, RealRead : Cardinal;
    ReadOL : TOverLapped; {структура для асинхронного чтения}
    Signaled, Mask : DWORD;
    BytesTrans : DWORD; {не используется для WaitCommEvent}
    LineStatus : Cardinal; {текущее состояние линий порта}
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 or EV_CTS or EV_DSR or EV_RING);

    While (not Terminated) and isConnected 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);
```

```
    { Число полученных, но еще не прочитанных байт}
    AvaibleBytes:= CurrentState.cbInQue;
    { Проверка числа доступных байт}
    If AvaibleBytes > 0 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;

   GetCommModemStatus(FHandle, LineStatus);

   If (Mask and EV_CTS) <> 0 then begin
     FCTS:= (LineStatus and MS_CTS_ON) <> 0;
     Synchronize(DoChangeCTS);
   End;
   If (Mask and EV_DSR) <> 0 then begin
     FDSR:= (LineStatus and MS_DSR_ON) <> 0;
     Synchronize(DoChangeDSR);
   End;
   If (Mask and EV_RING) <> 0 then begin
     FRING:= (LineStatus and MS_RING_ON) <> 0;
     Synchronize(DoChangeRING);
   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;

Procedure TReadThread.DoChangeCTS;
Begin
 With FOwner do begin
  If Assigned(FOnCTSChangeEvent) then
   FOnCTSChangeEvent(FCTS);
 End;
End;

Procedure TReadThread.DoChangeDSR;
Begin
 With FOwner do begin
  If Assigned(FOnDSRChangeEvent) then
   FOnCTSChangeEvent(FDSR);
 End;
End;
```

```
Procedure TReadThread.DoChangeRING;
Begin
 With FOwner do begin
  If Assigned(FOnRINGChangeEvent) then
   FOnCTSChangeEvent(FRING);
 End;
End;

{ 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;

procedure TComPort.SetBreak;
begin
 SetCommBreak(FHandle);
end;

Procedure TComPort.ClrBreak;
begin
 ClearCommBreak(FHandle);
end;

Procedure TComPort.SetupBuffer;
begin
 SetupComm(FHandle, 20, 20);
end;

end.
```

Листинг 9.14. Пример работы с классом `TComPort`

```
unit Unit1;

interface

uses
```

```
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, ComPort, StdCtrls, Buttons, ComCtrls, ExtCtrls;

type
  TMainForm = class(TForm)
    StatusBar: TStatusBar;
    Panel1: TPanel;
    GroupBox1: TGroupBox;
    cbPort: TComboBox;
    btnOpenPort: TBitBtn;
    btnClosePort: TBitBtn;
    btnConfig: TBitBtn;
    Panel2: TPanel;
    GroupBox2: TGroupBox;
    Panel3: TPanel;
    lbReadData: TListBox;
    cbReadActive: TCheckBox;
    GroupBox3: TGroupBox;
    GroupBox4: TGroupBox;
    btnClearLog: TSpeedButton;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnOpenPortClick(Sender: TObject);
    procedure btnClosePortClick(Sender: TObject);
    procedure btnConfigClick(Sender: TObject);
    procedure cbReadActiveClick(Sender: TObject);
    procedure btnClearLogClick(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
```

```
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  private
    procedure SetControlsState;
    procedure OnReadPacket(const Packet : Pointer; const Size : Integer;
    const ErrCode: Cardinal);
    procedure OnCTSChangeEvent(const CTS : Boolean);
    procedure OnDSRChangeEvent(const DSR : Boolean);
    procedure OnRINGChangeEvent(const RING : Boolean);
  public
   Port : TComPort;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.dfm}

procedure TMainForm.FormCreate(Sender: TObject);
begin
 // Получение списка портов системы
 TComPort.EnumComPorts(cbPort.Items);
 // Создание объекта для чтения порта
 Port:= TComPort.Create(Self);
 Port.OnReadPacket     := OnReadPacket;
 Port.OnCTSChangeEvent := OnCTSChangeEvent;
 Port.OnDSRChangeEvent := OnDSRChangeEvent;
 Port.OnRINGChangeEvent:= OnRINGChangeEvent;

 // Настройка порта
 Port.ComProp.BaudRate:= br9600;
 Port.ComProp.ByteSize:= bs8;
```

```
 // Установить состояние элементов
 SetControlsState;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
 // Освобождение объекта для чтения порта
 Port.Free;
end;

procedure TMainForm.SetControlsState;
begin
 btnOpenPort.Enabled := not Port.IsConnected;
 cbPort.Enabled      := not Port.IsConnected;
 btnClosePort.Enabled:= Port.IsConnected;
 btnConfig.Enabled   := Port.IsConnected;
end;

procedure TMainForm.btnOpenPortClick(Sender: TObject);
begin
 // Открытие порта
 If cbPort.Text <> '' then
  Port.Open(cbPort.Text);
 If Port.IsConnected then
  StatusBar.SimpleText:= Format('Порт %s открыт', [cbPort.Text]);
 Port.SetupBuffer;
 SetControlsState;
 cbReadActiveClick(nil);
end;

procedure TMainForm.btnClosePortClick(Sender: TObject);
begin
 // Закрытие порта
 Port.Close;
```

```
 StatusBar.SimpleText:= '';
 SetControlsState;
end;

procedure TMainForm.btnConfigClick(Sender: TObject);
begin
 // Диалог параметров порта
 Port.CommDialog;
end;

procedure TMainForm.OnReadPacket(const Packet : Pointer; const Size :
Integer; const ErrCode: Cardinal);
Var B : Byte; i : Integer; S : String;
begin
 S:= '';
 For i:= 0 to Size-1 do begin
  B:= Byte(Pointer(LongInt(Packet)+i)^);
  S:= S + Format(' %.2X',[B]);
 End;
 lbReadData.Items.Add(S);
 lbReadData.ItemIndex:= lbReadData.Items.Count-1;
end;

procedure TMainForm.OnCTSChangeEvent(const CTS : Boolean);
begin
 lbReadData.Items.Add('CTS');
end;

procedure TMainForm.OnDSRChangeEvent(const DSR : Boolean);
begin
 lbReadData.Items.Add('DSR');
end;

procedure TMainForm.OnRINGChangeEvent(const RING : Boolean);
begin
```

```
 lbReadData.Items.Add('RING');
end;

procedure TMainForm.cbReadActiveClick(Sender: TObject);
begin
 Port.ReadActive:= cbReadActive.Checked;
end;

procedure TMainForm.btnClearLogClick(Sender: TObject);
begin
 lbReadData.Clear;
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
 Port.ApplyComSettings;
end;

procedure TMainForm.Button2Click(Sender: TObject);
begin
 Port.SetBreak;
end;

procedure TMainForm.Button3Click(Sender: TObject);
begin
 Port.ClrBreak;
end;

procedure TMainForm.Button4Click(Sender: TObject);
begin
 Port.SetupBuffer;
end;

end.
```

9.5. Программирование обмена с CDC-устройством на языке C#

Для работы с последовательным портом на платформе .NET можно использовать либо специальный ActiveX-компонент MSCOMM, либо импортированные Win32-функции.

9.5.1. Использование компонента MSCOMM

Для использования компонента MSCOMM его следует подключить с помощью меню **Tools→Add/Remove ToolBox Items** (рис. 9.3). После этого он появится в панели компонентов, и его можно будет добавить на форму (рис. 9.4).

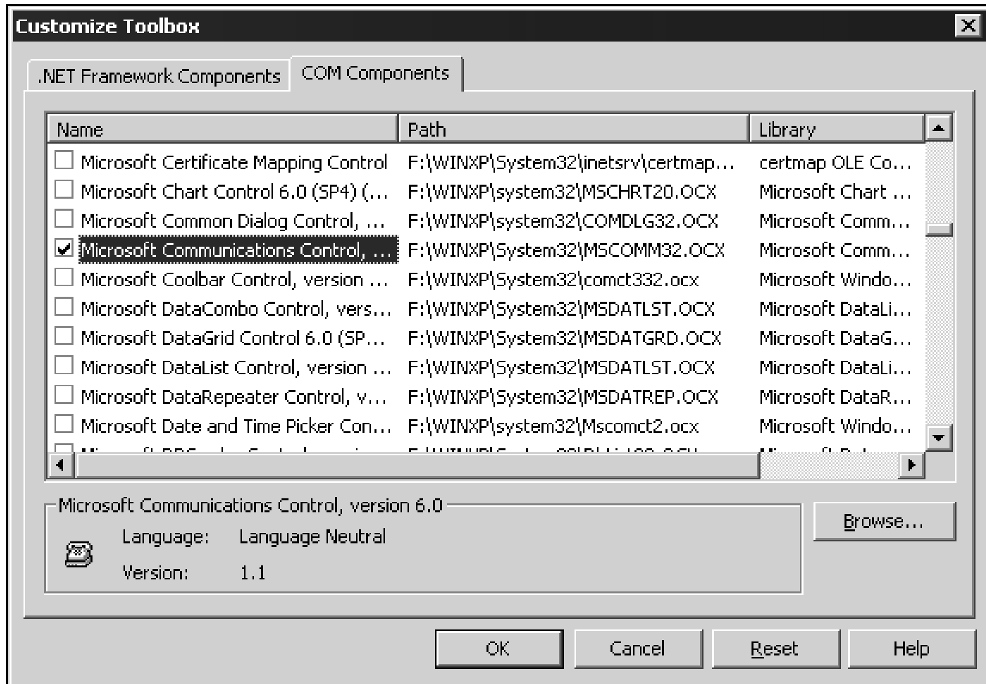


Рис. 9.3. Стандартный диалог Windows настройки компонентов

Пример использования компонента MSCOMM показан в листинге 9.15.

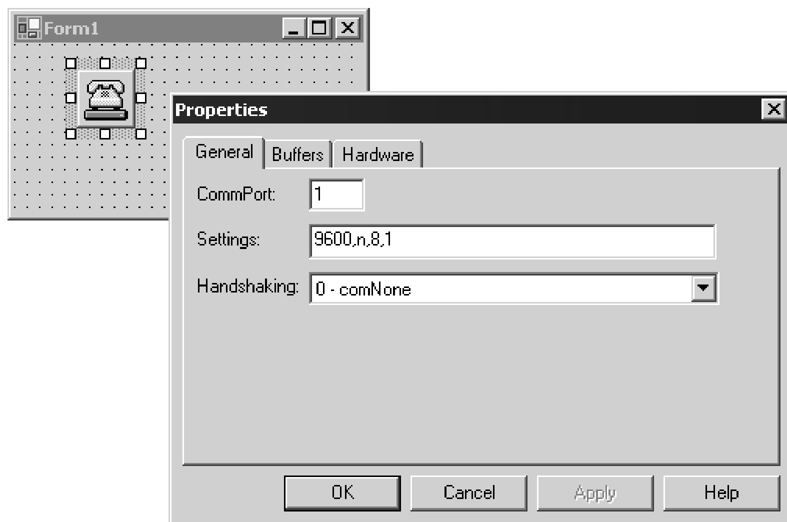


Рис. 9.4. Добавление компонента MSCOMM на форму

Важно!

В Windows 95/98 вызов `MSComm.PortOpen` приводит к утечке памяти в размере примерно 4 Кбайт на 100 вызовов. Эта ошибка исправлена в модулях из Visual Studio 97 SP2.

Листинг 9.15. Использование MSCOMM

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace MSCOMMTest
{
  public class Form1 : System.Windows.Forms.Form
  {
    private AxMSCommLib.AxMSComm axMSComm1;
    private System.Windows.Forms.ListBox lbLog;
    private System.ComponentModel.Container components = null;
```

```
    public Form1()
    {
      InitializeComponent();
      InitPort();
    }
... ... ... ... ... ... ... ... ... ... ... ... ...
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }

    void InitPort()
    {
      // Задаем номер порта
      axMSComm1.CommPort = 1;
      // Закрыть порт, если он открыт
      if (axMSComm1.PortOpen) axMSComm1.PortOpen = false;
      // Включить вызов события при получении данных
       axMSComm1.RThreshold = 1;
      // Настройка порта: 9600, no parity, 8 bits, 1 stop bit
      axMSComm1.Settings = "9600,n,8,1";
      // Режим приема данных - бинарный или текстовый
      axMSComm1.InputMode =
             MSCommLib.InputModeConstants.comInputModeBinary;
      //     MSCommLib.InputModeConstants.comInputModeText;
      // Режим ожидания данных
      axMSComm1.InputLen = 0;
      // Не игнорировать 0x00
      axMSComm1.NullDiscard = false;
      // Добавляем обработчик получения данных
      axMSComm1.OnComm += new System.EventHandler(this.OnComm);
      // Открываем порт
      axMSComm1.PortOpen = true;
    }
```

```
//  Обработчик получения данных
 private void OnComm(object sender, EventArgs e)
 {
   Application.DoEvents();
   // Обрабока изменения состояния линии CTS
   if (axMSComm1.CommEvent ==
                (short)MSCommLib.OnCommConstants.comEvCTS)
   {
      Log("Изменение состояния линии CTS");
   }
    // Если есть данные в буфере
   if (axMSComm1.InBufferCount > 0)
   {
     // Если включен режим comInputModeText,
     // мы получим строку
     if (axMSComm1.Input is String)
     {
        string Input = (string) axMSComm1.Input;
        Log(Input);
     }
     // Если включен режим comInputModeBinary, мы получим
     // System.Array
     else
     {
       byte [] Input = (byte[])axMSComm1.Input;
       foreach (byte b in      Input)
       {
         Log(string.Format("{0}", b));
       }
     }
   }
 }
 private void Log(string str)
 {
```

```
            lbLog.Items.Add(str);
            lbLog.SelectedIndex = lbLog.Items.Count-1;
        }
    }
}
```

9.5.2. Использование функций Win32

Код на C# полностью соответствует коду Win32, за исключением того, что все функции необходимо импортировать из библиотки kernel (листинг 9.16).

Листинг 9.16. Обмен данными с помощью функций Win32

```
// Реализация класса для работы с портом
using System;
using System.Runtime.InteropServices;

namespace ReadCDC
{
  public enum Parity : byte
  {
     No    = 0,
     Odd   = 1,
     Even  = 2,
     Mark  = 3,
     Space = 4
  }

  public enum StopBits : byte
  {
     Bits1   = 0,
     Bits1_5 = 1,
     Bits2   = 2
  }

  class CommPort
  {
```

```
private int  PortNum;
private int  BaudRate;
private byte ByteSize;
private Parity   parity;
private StopBits stopBits;
private int  hPortHanle = INVALID_HANDLE_VALUE;

public CommPort(
       int PortNum, int BaudRate, byte ByteSize,
       Parity parity, StopBits stopBits
)
{
   this.PortNum    = PortNum;
   this.BaudRate   = BaudRate;
   this.ByteSize   = ByteSize;
   this.parity             = parity;
   this.stopBits   = stopBits;
}

public bool Open()
{
  // Открытие порта
  hPortHanle = CreateFile("COM" + PortNum,
      GENERIC_READ | GENERIC_WRITE,0, 0,OPEN_EXISTING,0,0);
  if(hPortHanle == INVALID_HANDLE_VALUE)
  {
     return false;
  }
  // Настройка порта
  DCB dcbCommPort = new DCB();
  GetCommState(hPortHanle, ref dcbCommPort);
  dcbCommPort.BaudRate = BaudRate;
  dcbCommPort.Parity   = (byte)parity;
  dcbCommPort.ByteSize = ByteSize;
  dcbCommPort.StopBits = (byte)stopBits;
  if (!SetCommState(hPortHanle, ref dcbCommPort))
```

```
    {
      return false;
    }
    return true;
  }
  // Возврашает true, если порт открыт
  public bool IsOpen()
  {
     return(hPortHanle!=INVALID_HANDLE_VALUE);
  }
  // Закрытие порта
  public void Close()
  {
    if (IsOpen())
    {
      CloseHandle(hPortHanle);
    }
  // Чтение данных
  public byte[] Read(int NumBytes)
  {
     byte[] BufBytes;
     byte[] OutBytes;
     BufBytes = new byte[NumBytes];
     if (hPortHanle!=INVALID_HANDLE_VALUE)
     {
        int BytesRead=0;
        ReadFile(hPortHanle, BufBytes, NumBytes,
                                     ref BytesRead, 0);
        OutBytes = new byte[BytesRead];
        Array.Copy(BufBytes, OutBytes, BytesRead);
     }
     else
     {
        throw(new ApplicationException("Порт не был открыт"));
     }
     return OutBytes;
   }
```

```
// Передача данных
public void Write(byte[] WriteBytes)
{
   if (hPortHanle!=INVALID_HANDLE_VALUE)
   {
      int BytesWritten = 0;
      WriteFile(hPortHanle, WriteBytes, WriteBytes.Length,
                                ref BytesWritten, 0);
   }
   else
   {
      throw(new ApplicationException("Порт не был открыт"));
   }
 }
 // Описание констант Win32 API
 private const uint GENERIC_READ  = 0x80000000;
 private const uint GENERIC_WRITE = 0x40000000;
 private const int OPEN_EXISTING  = 3;
 private const int INVALID_HANDLE_VALUE = -1;

 [StructLayout(LayoutKind.Sequential)]
 public struct DCB
 {
   public int DCBlength;
   public int BaudRate;
   /*
      public int fBinary;
      public int fParity;
      public int fOutxCtsFlow;
      public int fOutxDsrFlow;
      public int fDtrControl;
      public int fDsrSensitivity;
      public int fTXContinueOnXoff;
      public int fOutX;
```

```
      public int fInX;
      public int fErrorChar;
      public int fNull;
      public int fRtsControl;
      public int fAbortOnError;
      public int fDummy2;
    */
    public uint flags;
    public ushort wReserved;
    public ushort XonLim;
    public ushort XoffLim;
    public byte ByteSize;
    public byte Parity;
    public byte StopBits;
    public char XonChar;
    public char XoffChar;
    public char ErrorChar;
    public char EofChar;
    public char EvtChar;
    public ushort wReserved1;
  }
  [DllImport("kernel32.dll")]
  private static extern int CreateFile(
    string lpFileName,
    uint dwDesiredAccess,
    int dwShareMode,
    int lpSecurityAttributes,
    int dwCreationDisposition,
    int dwFlagsAndAttributes,
    int hTemplateFile
  );
  [DllImport("kernel32.dll")]
  private static extern bool GetCommState(
    int hFile,              // дескриптор файла (порта)
```

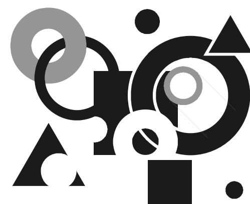
```
      ref DCB lpDCB   // структура DCB
    );
    [DllImport("kernel32.dll")]
    private static extern bool SetCommState(
      int hFile,              // дескриптор файла (порта)
      ref DCB lpDCB   // структура DCB
    );
    [DllImport("kernel32.dll")]
    private static extern bool ReadFile(
      int hFile,          // дескриптор файла (порта)
      byte[] lpBuffer,            // буфер
      int nNumberOfBytesToRead,       // размер буфера
      ref int lpNumberOfBytesRead,// реально прочитано
      int  lpOverlapped       // 0 для синхронных операций
    );
    [DllImport("kernel32.dll")]
    private static extern bool WriteFile(
      int hFile,     // дескриптор файла (порта)
      byte[] lpBuffer,                // буфер данных
      int nNumberOfBytesToWrite,      // число байт данных
      ref int lpNumberOfBytesWritten, // реально передано
      int lpOverlapped          // 0 для синхронных операций
    );
    [DllImport("kernel32.dll")]
    private static extern bool CloseHandle(
      int hObject   // дескриптор файла (порта)
    );
  }
}

// Пример использования класса
CommPort port = new CommPort(1, 9600, 8, Parity.No, StopBits.Bits1);
if (port.Open())
{
  byte[] data = port.Read(100);
```

```
  foreach (byte b in data)
  {
    listBox1.Items.Add(string.Format("{0}", b));
  }
  port.Close();
}
```

9.6. Проблемы CDC

К сожалению, текущая реализация CDC имеет некоторые проблемы. На сайте Microsoft (или в MSDN) можно найти несколько статей, посвященных проблемам CDC-драйвера для Windows XP. Наибольший интерес представляют статья Q308349 — падение Windows 98 при использовании драйвера usbser.sys (Windows 98 Shutdown Hangs for Five Minutes with USB Modem Present). Форум usb.org и вовсе не рекомендует пользоваться стандартными CDC-драйверами, а использовать драйверы сторонних разработчиков. Microsoft предлагает единственное решение всех проблем — установку обновлений операционной системы.

Глава 10



Реализация класса HID

В этой главе мы реализуем HID-устройство, описание которого дано в *гл. 6*. HID-устройства представляют большой интерес для разработчика, т. к. позволяют создать разнообразные USB-устройства без написания драйверов. Кроме того, как мы увидим (*см. гл. 11*), эти устройства поддерживаются библиотекой DirectX.

10.1. Реализация HID на AT89C5131

За основу нашего HID-устройства мы возьмем базовый проект (*см. разд. 8.5*). Для реализации необходимо выполнить следующие действия (*минимальные требования см. в разд. 6.1*):

- создать дескриптор репорта (*см. разд. 6.5*) либо вручную, либо с помощью одной из утилит (*см. разд. 6.7*);
- добавить в структуру `usb_configuration` HID-дескриптор (тип `usb_st_hid_descriptor`);
- добавить описание конечной точки типа Interrupt IN;
- добавить код для передачи данных в хост через конечную точку (в данном случае мы будем передавать всего один байт);
- в код функции `GetDescriptor` добавить обработку запроса дескриптора `HID` (код 0x21) и дескриптора `REPORT` (код 0x22).

Описание новых дескрипторов приведено в листинге 10.1.

Листинг 10.1. Описание дескрипторов HID-устройства

```
#define USB_SPECIFICATION      0x0100
#define DEVICE_CLASS           0x00
#define DEVICE_SUB_CLASS       0x00
```

```
#define DEVICE_PROTOCOL       0x00
#define EP_CONTROL_LENGTH     8
#define VENDOR_ID             0xEB03
#define PRODUCT_ID            0x0357
#define RELEASE_NUMBER        0x0000

#define LANG_ID              0x00
#define MAN_INDEX            0x01
#define PRD_INDEX            0x02
#define SRN_INDEX            0x03

code struct usb_st_device_descriptor usb_device_descriptor =
{
  sizeof(usb_device_descriptor),
  DEVICE,             /* =1 */
  USB_SPECIFICATION,  /* спецификация     */
  DEVICE_CLASS,       /* класс устройства */
  DEVICE_SUB_CLASS,
  DEVICE_PROTOCOL,    /* протокол USB     */
  EP_CONTROL_LENGTH,  /* размер пакета 0-й точки */
  VENDOR_ID,          /* ID производителя и устройства */
  PRODUCT_ID,
  RELEASE_NUMBER,     /* версия устройства */
  MAN_INDEX,          /* дескриптор строки изготовителя   */
  PRD_INDEX,          /* дескриптор строки продукта       */
  SRN_INDEX,          /* дескриптор строки серийного ном. */
  1                   /* число конфигураций */
};

#define CONF_LENGTH           wSWAP(34)
#define CONF_NB               1
#define CONF_ATTRIBUTES       USB_CONFIG_BUSPOWERED
#define MAX_POWER             50  /* = 100 mA */

/* INTERFACE 0 DESCRIPTOR */
#define INTERFACE0_CLASS       0x03 /* HID */
```

```
#define INTERFACE0_SUB_CLASS   0xFF
#define INTERFACE0_PROTOCOL    0xFF

#define SIZE_OF_REPORT        23

/* первая конечная точка */
#define EP_1_CONFIG    (INTERRUPT|EP_CONFIG_IN)  /* конфигурация */
#define EP_1_ADDRESS   (1|EP_DIRECT_IN)          /* адрес        */
#define EP_1_ATTRIBUTES INTERRUPT                /* атрибуты     */

code struct
{
  struct usb_st_configuration_descriptor  cfg;
  struct usb_st_interface_descriptor      ifc;
  struct usb_st_hid_descriptor            hid;
  struct usb_st_endpoint_descriptor       ep1;
 }
 usb_configuration =
 {
/* CONFIGURATION */
    { 9,
      CONFIGURATION,   /* =2 */
      CONF_LENGTH,     /* длина всех дескрипторов */
      1,               /* число интерфейсов       */
      CONF_NB,         /* номер конфигурации      */
      0,               /* дескриптор строки конфигурации  */
      CONF_ATTRIBUTES, /* атрибуты                */
      MAX_POWER        /* 100 мА                  */
    },
/* INTERFACE 0 */
    { 9,
      INTERFACE,       /* =4 */
      0,               /* 0-й интерфейс              */
      0,               /* номер альтернативного интерфейса */
      1,               /* 1 конечная точка (кроме 0) */
      INTERFACE0_CLASS,
```

```
      INTERFACE0_SUB_CLASS,
      INTERFACE0_PROTOCOL,
      0                 /* дескриптор строки интерфейса   */
    },
/* Дескриптор HID */
    { 9,
      HID,              /* дескриптор HID */
      0x0001,           /* Версия HID     */
      0,           /* Числовой код страны для локализованных устройств */
      1,                /* Число дескрипторов репортов */
      REPORT,           /* Номер дескриптора репорта   */
      wSWAP(SIZE_OF_REPORT) /* Размер дескриптора репорта */
    },
/* Дескриптор первой конечной точки */
    {
      7,
      ENDPOINT,           /* дескриптор ENDPOINT        */
      EP_1_ADDRESS,       /* номер конечной точки       */
      EP_1_ATTRIBUTES,    /* атрибуты конечной точки    */
      wSWAP(8),           /* максимальный размер пакета */
      100                 /* частота опроса             */
    }
   };

code struct{
  byte  rep[SIZE_OF_REPORT];
 }
 HIDReport =
    /* HID Report */
    {
      0x06, 0x00, 0xff,  /* USAGE_PAGE (Generic Desktop)      */
      0x09, 0x01,        /* USAGE (Vendor Usage 1)            */
      0xa1, 0x01,        /* COLLECTION (Application)          */
      0x19, 0x01,        /*   USAGE_MINIMUM (Vendor Usage 1)  */
      0x29, 0x01,        /*   USAGE_MAXIMUM (Vendor Usage 1)  */
      0x15, 0x00,        /*   LOGICAL_MINIMUM (0)             */
```

```
     0x26, 0x00, 0xFF,  /*   LOGICAL_MAXIMUM (255)          */
/* размер байта - 8 */
     0x75, 0x08,        /*   REPORT_SIZE (8)                */
/* число байт - 1 */
     0x95, 0x01,        /*   REPORT_COUNT(1)                */
     0x81, 0x02,        /*   INPUT (Data,Var,Abs)           */
     0xc0               /* END_COLLECTION                   */
    };
```

Комбинация параметров `RepostSize=8` и `ReportCount=1` указывает на передачу одного байта данных. Для передачи специальных дескрипторов мы модифицируем код функции `GetDescriptor` (листинг 10.2).

Листинг 10.2. Передача специальных дескрипторов HID-устройства

```
#define HID                     0x21
#define REPORT                  0x22

void GetDescriptor()
{
... ... ... ... ... ... ... ... ...
 /* Тип запрашиваемого дескриптора находится  */
 /* в старшем байте поля wValue, т. е. байте 3 */
 switch (SetupPacket.b[3])
 {
... ... ... ... ... ... ... ... ...
    /* HID-дескриптор */
    case HID:
    {
      data_to_transfer = sizeof(usb_configuration.hid);
      pbuffer = &(usb_configuration.hid.bLength);
      break;
    }
    /* дескриптор репорта */
    case REPORT:
    {
      data_to_transfer = SIZE_OF_REPORT;
```

```
      pbuffer = &(HIDReport.rep[0]);
      break;
    }
... ... ... ... ... ... ... ... ...
}
```

Инициализация конечной точки показана в листинге 10.3, а процедура передачи данных — в листинге 10.4.

Листинг 10.3. Инициализация конечной точки HID-устройства

```
void SetConfiguration()
{
... ... ... ... ... ... ... ... ...
  /* Конфигурирование других конечных точек */
  usb_configure_endpoint(1, INTERRUPT_IN);
  usb_reset_endpoint(1);
  Usb_enable_ep_int(1);
}
```

Листинг 10.4. Передача данных по первой конечной точке

```
void main()
{
... ... ... ... ... ... ... ... ...
 /* основной цикл программы */
 for (;;)
 {
  Fill_EP1_FIFO();
... ... ... ... ... ... ... ... ...
   /* обнаружено прерывание от конечной точки */
   if (Usb_endpoint_interrupt())
   {
... ... ... ... ... ... ... ... ...
    /* переключиться на 1 конечную точку */
    Usb_select_ep(1);
    if (Usb_tx_complete())
```

```
    {
      Usb_clear_tx_complete();
      end_point1_ready = FALSE;
    }
   } /* end if interrupt */
 } /* end for ;; */
}

void Fill_EP1_FIFO()
{
 /* устройство не готово к передаче данных */
 if ((usb_configuration_nb == 0) || (Usb_suspend()))
  return;

 if (!end_point1_ready) /* предыдущий пакет отправлен */
 {
   Usb_select_ep(1);
   Usb_write_byte(point1_state);
   Usb_set_tx_ready();

   point1_state++;
   end_point1_ready = TRUE;
 }
}
```

Переменная `point_state` нужна для удобства тестирования, чтобы передаваемый байт изменял свое значение. Флаг `end_point1_ready` показывает, что конечная точка 1 готова к передаче нового пакета.

Итак, мы реализовали простейшее HID-устройство, передающее на компьютер один байт. Для экономии места мы сначала реализуем еще несколько более сложных HID-устройств, а затем займемся программой чтения данных для Windows.

10.2. Передача нескольких байтов

Для передачи более чем одного байта достаточно поменять HID-репорт и, соответственно, процедуру передачи самих данных. Пример показан в лис-

тинге 10.5. Здесь мы передаем семь байтов: поле `REPORT_SIZE` указывает, что мы передаем восемь битов, а поле `REPORT_COUNT` — что мы будем передавать семь таких частей.

Листинг 10.5. Передача семи байтов

```
// HID-дескриптор
code struct{
  byte  rep[SIZE_OF_REPORT];
 }
 HIDReport =
    /* HID Report */
    {
      0x06, 0x00, 0xff,  /* USAGE_PAGE (Generic Desktop)      */
      0x09, 0x01,        /* USAGE (Vendor Usage 1)            */
      0xa1, 0x01,        /* COLLECTION (Application)          */
      0x19, 0x01,        /*   USAGE_MINIMUM (Vendor Usage 1)  */
      0x29, 0x01,        /*   USAGE_MAXIMUM (Vendor Usage 1)  */
      0x15, 0x00,        /*   LOGICAL_MINIMUM (0)             */
      0x26, 0x55, 0x00,  /*   LOGICAL_MAXIMUM (55)           */
      0x75, 0x08,        /*   REPORT_SIZE (8)                 */
      0x95, 0x07,        /*   REPORT_COUNT(7)                 */
      0x81, 0x02,        /*   INPUT (Data,Var,Abs)            */
      0xc0               /* END_COLLECTION                    */
    };

// Процедура передачи
void Fill_EP1_FIFO()
{
 /* устройство не готово к передаче данных */
 if ((usb_configuration_nb == 0) || (Usb_suspend()))
  return;

 if (!end_point1_ready) /* предыдущий пакет отправлен */
 {
   Usb_select_ep(1);
    Usb_write_byte(point1_state);
```

```
    Usb_write_byte(0x02);
    Usb_write_byte(0x03);
    Usb_write_byte(0x04);
    Usb_write_byte(0x05);
    Usb_write_byte(0x06);
    Usb_write_byte(0x07);
   Usb_set_tx_ready();
   point1_state++;
   end_point1_ready = TRUE;
 }
}
```

10.3. Feature-репорты

Репорты типа Feature (специальные репорты) используются там, где важно время доставки данных, например, для установки различных параметров устройства и его инициализации.

Для перевода нашего примера на Feature-репорты достаточно изменить HID-дескриптор, а процедуру передачи данных `Fill_EP1_FIFO` заменить обработкой запроса `GET_REPORT`, как это показано в листинге 10.6 (весь код, относящийся к первой конечной точке, мы удаляем).

Листинг 10.6. Передача Feature-репорта

```
// Описание кода запроса
#define GET_REPORT             0xA101

// HID-дескриптор
code struct{
  byte  rep[SIZE_OF_REPORT];
 }
 HIDReport =
    /* HID Report */
    {
      0x06, 0x00, 0xff,  /* USAGE_PAGE (Generic Desktop)      */
      0x09, 0x01,        /* USAGE (Vendor Usage 1)            */
      0xa1, 0x01,        /* COLLECTION (Application)          */
```

```
      0x19, 0x01,         /*   USAGE_MINIMUM (Vendor Usage 1)  */
      0x29, 0x01,         /*   USAGE_MAXIMUM (Vendor Usage 1)  */
      0x15, 0x00,         /*   LOGICAL_MINIMUM (0)             */
      0x26, 0xff, 0x00,   /*   LOGICAL_MAXIMUM (255)           */
      0x75, 0x08,         /*   REPORT_SIZE (8)                 */
      0x95, 0x07,         /*   REPORT_COUNT(7)                 */
      0xB1, 0x02,         /*   FEATURE (Data,Var,Abs)          */
      0xc0                /* END_COLLECTION                    */
    };
void UsbControlPacketProcessed()
{
... ... ... ... ... ... ... ... ...
  switch (SetupPacket.wRequest)
  {
... ... ... ... ... ... ... ... ...
    case GET_REPORT:
     hid_get_report();
     break;
    default:
     STALL();
     break;
  }
}

void hid_get_report()
{
 if (usb_configuration_nb == 0){
  STALL();
  return;
 }

 Usb_clear_rx_setup();
 Usb_set_DIR();
    Usb_write_byte(0x11);
    Usb_write_byte(0x12);
```

```
    Usb_write_byte(point1_state);
    Usb_write_byte(0x14);
    Usb_write_byte(0x15);
    Usb_write_byte(0x16);
    Usb_write_byte(0x17);
    point1_state++;
 SendDataFromFIFO();
 WaitForOutZeroPacket();
}
```

10.4. Передача данных от хоста (*SET_REPORT*)

Получение данных производится с помощью обработки запроса `SET_REPORT`. Ничего нового в этом коде нет, поэтому мы просто приведем его без дополнительных комментариев (листинг 10.7).

Листинг 10.7. Обработка запроса `SET_REPORT`

```
// Описание кода запроса
#define SET_REPORT            0x2109

void UsbControlPacketProcessed()
{
... ... ... ... ... ... ... ... ...
  switch (SetupPacket.wRequest)
  {
... ... ... ... ... ... ... ... ...
    case SET_REPORT:
     hid_set_report();
     break;
    default:
     STALL();
     break;
  }
}
```

```
void hid_set_report()
{
 WaitForFillFIFO();

/* Читаем данные */
 Usb_read_byte();
 Usb_read_byte();
 Usb_read_byte();
 Usb_read_byte();
 Usb_read_byte();
 Usb_read_byte();
 Usb_read_byte();

 EndReadData();
 SendInZeroPacket();
}
```

10.5. Установка HID-устройства

Скомпилируем и загрузим одну из наших программ в тестовое HID-устройство. В Windows XP не потребуется даже перезагрузки — драйверы будут установлены автоматически. Windows 98 также устанавливает драйверы автоматически, но для их старта необходимо подать HID-устройству сигнал сброса. На рис. 10.1 показано обнаруженное HID-устройство в списке устройств системы.

Отметим, что HID-устройство как бы состоит из двух частей: HID-интерфейса (устройство USB Human Interface Device, мы описали его, указав тип 3 в дескрипторе интерфейса) и пользовательского интерфейса (устройство HID-compliant device, тип которого мы описали в дескрипторе репорта). Вся работа (чтение конфигурации, прием и передача данных) будет производиться со второй частью HID-устройства.

10.6. Обмен данными с HID-устройством

В этом разделе мы создадим небольшую программу для обмена данными с HID-устройством. Мы специально не приводили код этой программы в предыдущих разделах, хотя и подразумевали ее наличие. Мы объединим программу работы с разными типами репортов, сделав одну, универсальную программу.

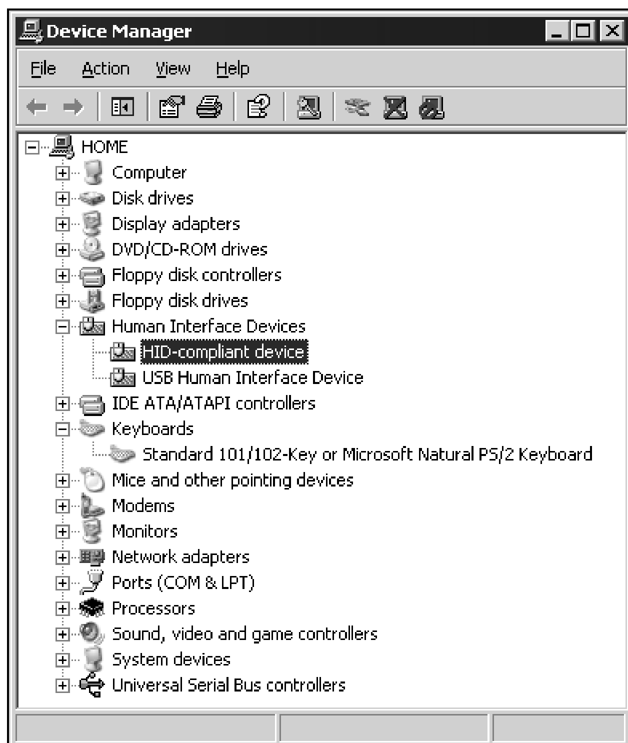


Рис. 10.1. HID-устройство обнаружено и успешно установлено

10.6.1. Получение имени HID-устройства

Для чтения данных с HID-устройства нужно получить его дескриптор с помощью функции `CreateFile`. В отличие от, например, последовательных портов HID-устройства не имеют простых и понятных имен (COM1, COM2 и т. д.). Для получения имени устройства нужно воспользоваться функциями модуля Setup API и выполнить следующую последовательность действий:

1) получить GUID класса HID с помощью вызова `HidD_GetHidGuid`;
2) получить дескриптор PnP для класса HID с помощью вызова `SetupDiGetClassDevs`;
3) произвести цикл по всем HID-устройствам, вызывая `SetupDiGetDeviceInterfaceDetail`, и найти нужное устройство (для простоты мы будем производить чтение данных с каждого найденного устройства).

Соответствующий код показан в листинге 10.8 (*подробную информацию о функциях Setup API см. в разд. 11.1*).

Листинг 10.8. Поиск HID-устройств

```
// Отображение списка HID-устройств и их свойств
procedure TForm1.Button1Click(Sender: TObject);
var HidGuid : TGuid; PnPHandle : HDevInfo;
    DevData: TSPDevInfoData;
    DeviceInterfaceData: TSPDeviceInterfaceData;
    FunctionClassDeviceData: PSPDeviceInterfaceDetailData;
    Success: LongBool;
    DevIndex: DWORD;
    BytesReturned: DWORD;
    HidName : String;
begin
 // Очистить лог
 lbLog.Items.Clear;
 // Получить GUID для класса HID
 HidD_GetHidGuid(HidGuid);
 // Получаем дескриптор PnP для HID-класса
 PnPHandle:= SetupDiGetClassDevs(@HidGuid, nil, 0,
                  DIGCF_PRESENT or DIGCF_DEVICEINTERFACE);
 // Если ошибка, то выходим
 If PnPHandle = Pointer(INVALID_HANDLE_VALUE) then Exit;

 Try
  // Индекс текущего устройства
  DevIndex := 0;
  // Цикл по всем устройствам в HID-классе
  Repeat
   DeviceInterfaceData.cbSize := SizeOf(TSPDeviceInterfaceData);
   // Получить информацию об интерфейсах устройства номер DevIndex
   Success := SetupDiEnumDeviceInterfaces(PnPHandle, nil,
                      HidGuid, DevIndex, DeviceInterfaceData);
   If Success then begin
    DevData.cbSize := SizeOf(DevData);
```

```
    BytesReturned  := 0;
    // Получаем подробности об устройстве с
    // интерфейсом DeviceInterfaceData
    // Сначала вызываем с нулевым размером буфера,
    // получаем размер необходимого буфера, потом
    // вызываем повторно, сформировав правильный буфер
    SetupDiGetDeviceInterfaceDetail(PnPHandle, @DeviceInterfaceData,
                 nil, 0, BytesReturned, @DevData);
    If (BytesReturned <> 0) and
                (GetLastError = ERROR_INSUFFICIENT_BUFFER) then begin
     // Создаем буфер
     FunctionClassDeviceData := AllocMem(BytesReturned);
     FunctionClassDeviceData.cbSize := 5;
     // Получаем информацию
     If SetupDiGetDeviceInterfaceDetail(PnPHandle, @DeviceInterfaceData,
                 FunctionClassDeviceData, BytesReturned,
                 BytesReturned, @DevData) then begin
      // Отобразить имя PnP-имя устройства
      HidName:= StrPas(PChar(@FunctionClassDeviceData.DevicePath));
      lbLog.Items.Add(HidName);
      // Чтение репортов
      ReadHIDReports(HidName); /* см. листинг 10.9 */
     End;
     // Освободить буфер
     FreeMem(FunctionClassDeviceData);
    End;
   End;
   // Следующее устройство
   Inc(DevIndex);
  Until not Success;
 Finally
  SetupDiDestroyDeviceInfoList(PnPHandle);
 End;
end;
```

Компонент `lbLog` типа `TListBox` служит для отображения сообщений.

Получив имя (переменная `HidName`), мы вызываем функцию `ReadHIDReports(HidName)`, задачей которой является чтение данных с найденного HID-устройства. Эту функцию мы будем обсуждать в следующем разделе.

10.6.2. Получение атрибутов устройства и чтение репортов

Для работы с HID-устройством следует получить дескритор устройства, вызвав функцию `CreateFile` (*см. разд. 14.1*), также как мы делали это для CDC-устройства. Так как мы ничего не знаем про свойства найденного HID-устройства, то сначала пробуем открыть его для чтения и записи, а если не получится, то просто для чтения.

С успешно открытым HID-устройством можно работать. Первым делом необходимо получить его атрибуты, вызвав функцию `HidD_GetAttributes`. Это даст нам информацию, записанную в дескрипторе устройства: идентификаторы производителя (Vendor ID) и HID-устройства (Product ID), а также версию программы, записанную в HID-устройство (Version Number). Для получения строковых значений идентификаторов (если они описаны в дескрипторе) можно воспользоваться функциями `HidD_GetManufacturerString`, `HidD_GetProductString` и `HidD_GetSerialNumberString`.

В принципе, именно в этот момент времени можно произвести проверку свойств HID-устройства и, если его идентификаторы не совпадают с нужными, перейти к следующему (напомним, что листинг 10.8 содержит цикл по всем найденным USB-устройствам).

Для чтения данных мы должны определить, какие репорты поддерживает HID-устройство. Для этого мы используем функцию `HidP_GetCaps`, возвращающую размеры Input-, Output и Feature-репортов. Если размер соответствующего репорта равен нулю, HID-устройство не поддерживает репорты этого типа. Перед вызовом `HidP_GetCaps` требуется вызов `HidD_GetPreparsedData`, создающий специальный буфер типа `THIDPPreparsedData` (на самом деле этот тип является простым указателем на буфер данных).

Для чтения Input-репорта вызывается обычная функция `ReadFile` (в Windows 2000/XP можно использовать `ReadFileEx`). Для чтения Feature-репортов используется функция `HidD_GetFeature`.

Полный код функции чтения репортов показан в листинге 10.9.

Листинг 10.9. Чтение репортов

```
// Чтение репортов
procedure TForm1.ReadHIDReports(HidName : String);
```

```
var HidHandle : THandle;
    CanReadWriteAccess : Boolean;
    Attributes : THIDDAttributes;
    NumInputBuffers : Integer;
    Buffer  : array [0..253] of WideChar;
    InputReport : Array [0..255] of Byte;
    i : Integer; S : String;
    BytesRead : Cardinal;
    Capabilities : HIDP_CAPS;
    PreparsedData: PHIDPPreparsedData;
begin
 // Сначала пробуем открыть устройство
 // в режиме r/w
 lbLog.Items.Add(' Пробуем открыть HID-устройство...');
 HidHandle:= CreateFile(PChar(@HidName[1]),
              GENERIC_READ or GENERIC_WRITE,
              FILE_SHARE_READ or FILE_SHARE_WRITE,
              nil,
              OPEN_EXISTING,
              0,
              0
 );

 // Устройство поддерживает запись?
 CanReadWriteAccess:= HidHandle <> INVALID_HANDLE_VALUE;

 // Если не получилось, пробуем открыть
 // в режиме только чтения данных
 If not CanReadWriteAccess then begin
  HidHandle:= CreateFile(PChar(@HidName[1]),
               0,
               FILE_SHARE_READ or FILE_SHARE_WRITE,
               nil,
               OPEN_EXISTING, 0, 0
  );
 End else begin
```

```
 lbLog.Items.Add(' Устройство открыто в режиме read/write');
End;

// Если не получилось - ошибка и выход
If HidHandle = INVALID_HANDLE_VALUE then begin
 lbLog.Items.Add(' Ошибка открытия устройства');
 Exit;
End else begin
 lbLog.Items.Add(' Устройство открыто в режиме read only!');
End;

// Получаем атрибуты устройства
Attributes.Size := SizeOf(THIDDAttributes);
If HidD_GetAttributes(HidHandle, Attributes) then begin
 lbLog.Items.Add(Format(
   '   VendorID=%d, ProductID=%d, VersionNumber=%d',
       [Attributes.VendorID, Attributes.ProductID,
        Attributes.VersionNumber]
  ));
End else begin
 lbLog.Items.Add(' Ошибка HidD_GetAttributes');
End;

// Получаем число буферов
If HidD_GetNumInputBuffers(HidHandle, NumInputBuffers) then begin
 lbLog.Items.Add(Format(
         '   Число входных буферов=%d',[NumInputBuffers]));
End else begin
 lbLog.Items.Add(' Ошибка HidD_GetNumInputBuffers');
End;

// Получаем идентификатор изготовителя
FillChar(Buffer, SizeOf(Buffer), #0);
If HidD_GetManufacturerString(HidHandle,
                 Buffer, SizeOf(Buffer)) then begin
 lbLog.Items.Add(Format('   Производитель=%s',[Buffer]));
```

```
End else begin
 lbLog.Items.Add(' Ошибка HidD_GetManufacturerString');
End;

// Получаем идентификатор продукта
FillChar(Buffer, SizeOf(Buffer), #0);
If HidD_GetProductString(HidHandle, Buffer, SizeOf(Buffer)) then begin
 lbLog.Items.Add(Format('   Продукт=%s',[Buffer]));
End else begin
 lbLog.Items.Add(' Ошибка HidD_GetProductString');
End;

// Получаем серийный номер
FillChar(Buffer, SizeOf(Buffer), #0);
If HidD_GetSerialNumberString(
               HidHandle, Buffer, SizeOf(Buffer)) then begin
 lbLog.Items.Add(Format('   Серийный номер=%s',[Buffer]));
End else begin
 lbLog.Items.Add(' Ошибка HidD_GetSerialNumberString');
End;

If HidD_GetPreparsedData(HidHandle, PreparsedData) then begin
 HidP_GetCaps(PreparsedData, Capabilities);
 lbLog.Items.Add(Format(
   ' UsagePage              =%x',[Capabilities.UsagePage]
 ));
 lbLog.Items.Add(Format(
   ' InputReportByteLength  =%d',[Capabilities.InputReportByteLength]
 ));
 lbLog.Items.Add(Format(
   ' OutputReportByteLength =%d',[Capabilities.OutputReportByteLength]
 ));
 lbLog.Items.Add(Format(
   ' FeatureReportByteLength=%d',[Capabilities.FeatureReportByteLength
 ]));
End else begin
```

```
  lbLog.Items.Add(' Ошибка HidD_GetPreparsedData');
 End;

// Чтение данных с помощью HidD_GetFeature
 If Capabilities.FeatureReportByteLength > 0 then begin
  FillChar(InputReport, SizeOf(InputReport), #0);
  // InputReport[0] должен быть 0, если устройство
  // имеет один репорт.
  If HidD_GetFeature(HidHandle, InputReport,
             Capabilities.FeatureReportByteLength) then begin
   // отображение полученных данных
   S:= '';
   For i:= 1 to Capabilities.FeatureReportByteLength-1 do begin
    S:= S + ' ' + IntToHex(InputReport[i], 2);
   End;
   lbLog.Items.Add(' Прочитано:' + S);
  End else begin
   lbLog.Items.Add(
     ' Ошибка HidD_GetFeature ('+SysErrorMessage(GetLastError)+')');
  End;
 End else begin
  lbLog.Items.Add(' FeatureReport не поддерживается');
 End;

// Чтение данных с помощью ReadFile
 If Capabilities.InputReportByteLength > 0 then begin
  FillChar(InputReport, SizeOf(InputReport), #0);
  If ReadFile(HidHandle, InputReport,
       Capabilities.InputReportByteLength, BytesRead, nil) then begin
   // отображение полученных данных
   S:= '';
   For i:= 1 to BytesRead do begin
    S:= S + ' ' + IntToHex(InputReport[i], 2);
   End;
   lbLog.Items.Add(' Прочитано:' + S);
  End else begin
```

```
   lbLog.Items.Add(
       ' Ошибка ReadFile ('+SysErrorMessage(GetLastError)+')');
  End;
 End else begin
  lbLog.Items.Add(' InputReport не поддерживается');
 End;

 // Освободить блок PreparsedData
 HidD_FreePreparsedData(PreparsedData);
 // Освободить дескриптор устройства
 CloseHandle(HidHandle);
end;
```

Заметим, что после вызова `HidD_GetPreparsedData` полученный буфер необходимо освободить с помощью функции `HidD_FreePreparsedData`.

Запустив программу (ее полный код можно найти на компакт-диске), мы увидим информацию, показанную на рис. 10.2.

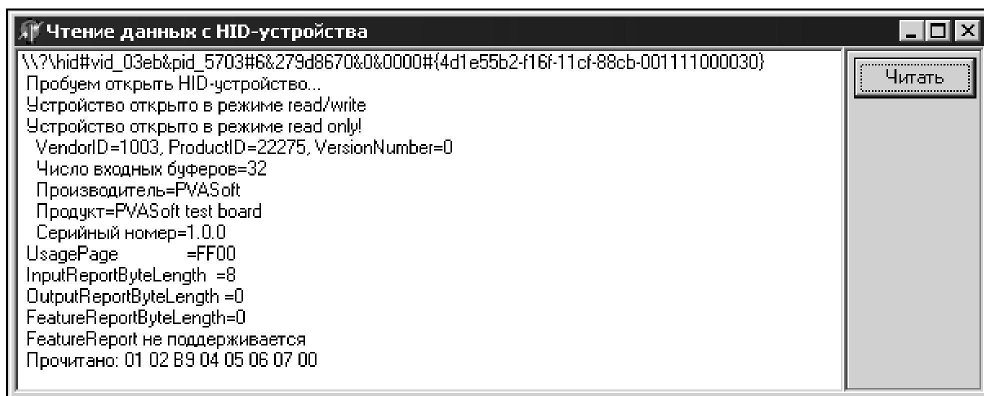


Рис. 10.2. Окно чтения Input-репорта с HID-устройства

Внимательно посмотрев на рисунок, можно заметить, что функция `HidP_GetCaps` вернула размер буфера на единицу больше, чем мы описывали в дескрипторе устройства. Это связано с тем, что в первом байте буфера передается номер репорта (Report ID). Более подробно этот вопрос мы рассмотрим в следующих разделах.

В заключение приведем код функции `GetHIDString` для получения строки по ее идентификатору (листинг 10.10).

Листинг 10.10. Получение строки по идентификатору

```
// Получение строки по идентификатору
Function GetHIDString(StrDescriptor : Byte): WideString;
var Buffer : array [0..253] of WideChar;
begin
 Result := 'Ошибка';
 if StrDescriptor <> 0 then
  if HidD_GetIndexedString(HidHandle, StrDescriptor,
         Buffer, SizeOf(Buffer)) then
 Result:= Buffer;
end;
```

10.6.3. Передача данных от хоста к HID-устройству

Передача данных может осуществляться либо с помощью функции `WriteFile` (для Windows XP можно использовать функцию `WriteFileEx`), либо (для Feature-репортов) с помощью специальной HID-функции `HidD_SetFeature`. Пример передачи данных в устройство показан в листинге 10.11.

Листинг 10.11. Передача данных с помощью `HidD_SetFeature`

```
Var
    OutputReport : Array [0..255] of Byte;

If Capabilities.FeatureReportByteLength > 0 then begin
  FillChar(OutputReport, SizeOf(OutputReport), #0);

  OutputReport[0]:=  0; // первый байт должен быть 0, если
                        // устройство имеет один репорт
  OutputReport[1]:=  1;
  OutputReport[2]:=  2;
  OutputReport[3]:=  3;
  OutputReport[4]:=  4;
```

```
 OutputReport[5]:=  5;
 OutputReport[6]:=  6;
 OutputReport[7]:=  7;

 If HidD_SetFeature(HidHandle, OutputReport,
              Capabilities.FeatureReportByteLength) then begin
  lbLog.Items.Add('HidD_SetFeature выполнен успешно');
 End else begin
  lbLog.Items.Add(' Ошибка HidD_SetFeature ('+
               SysErrorMessage(GetLastError)+')');
 End;
End else begin
 lbLog.Items.Add('FeatureReport не поддерживается');
End;
```

10.7. Примеры HID-устройств

Для примера мы приведем реализацию двух знакомых всем устройств — мыши и клавиатуры.

10.7.1. Реализация устройства "мышь"

Для реализации этого устройства необходимо заменить HID-дескриптор нашего примера, рассмотренного в *разд. 10.2*, на дескриптор, приведенный в листинге 6.3. Кроме того, мы будем имитировать нажатие правой кнопки мыши с помощью кнопки на порту P1.0 (листинг 10.12).

Листинг 10.12. Реализация устройства "мышь"

```
void Fill_EP1_FIFO()
{
 /* устройство не готово к передаче данных */
 if ((usb_configuration_nb == 0) || (Usb_suspend()))
  return;

 if (!end_point1_ready) /* предыдущий пакет отправлен */
 {
   Usb_select_ep(1);
```

```
    if (!P1.0)
     Usb_write_byte(0x01); /* 1 - правая кнопка мыши */
    else
     Usb_write_byte(0x00);
    Usb_write_byte(0x00);
    Usb_write_byte(0x00);

   Usb_set_tx_ready();

   end_point1_ready = TRUE;
 }
}
```

Протокол передачи, описанный HID-дескриптором, выглядит следующим образом:

- байт 0, бит 0 — состояние кнопки 1;
- байт 0, бит 1 — состояние кнопки 2;
- байт 0, бит 2 — состояние кнопки 3;
- байт 0, биты с 4 по 7 — резерв, зависит от устройства;
- байт 1 — смещение по X;
- байт 2 — смещение по Y.

Единственная тонкость реализации — нам придется удалить передачу дескрипторов строк и соответствующие индексы строк в дескрипторе устройства заменить нулями. По крайней мере, в Windows XP устройство, передающее строковые дескрипторы (кроме строки серийного номера), не опознается драйвером мыши. Полный код примера можно найти на компакт-диске.

10.7.2. Реализация устройства "клавиатура"

Пример дескриптора клавиатуры показан в листинге 10.13.

Листинг 10.13. Дескриптор клавиатуры

```
code struct{
  byte  rep[SIZE_OF_REPORT];
 }
 HIDReport =  /* HID Report */
```

```
{
  0x05,0x01, /* Usage Page (Generic Desktop)      */
  0x09,0x06, /* Usage (Keyboard)                  */
  0xA1,0x01, /* Collection (Application)          */
   0x05,0x07, /* Usage Page (Keyboard)             */
   0x19,224,  /* Usage Minimum (224)               */
   0x29,231,  /* Usage Maximum (231)               */
   0x15,0x00, /* Logical Minimum (0)               */
   0x25,0x01, /* Logical Maximum (1)               */
   0x75,0x01, /* Report Size (1)                   */
   0x95,0x08, /* Report Count (8)                  */
   0x81,0x02, /* Input (Data, Variable, Absolute)  */
   0x81,0x01, /* Input (Constant)                  */
   0x19,0x00, /* Usage Minimum (0)                 */
   0x29,101,  /* Usage Maximum (101)               */
   0x15,0x00, /* Logical Minimum (0)               */
   0x25,101,  /* Logical Maximum (101)             */
   0x75,0x08, /* Report Size (8)                   */
   0x95,0x06, /* Report Count (6)                  */
   0x81,0x00, /* Input (Data, Array)               */
   0x05,0x08, /* Usage Page (LED)                  */
   0x19,0x01, /* Usage Minimum (1)                 */
   0x29,0x05, /* Usage Maximum (5)                 */
   0x15,0x00, /* Logical Minimum (0)               */
   0x25,0x01, /* Logical Maximum (1)               */
   0x75,0x01, /* Report Size (1)                   */
   0x95,0x05, /* Report Count (5)                  */
   0x91,0x02, /* Output (Data, Variable, Absolute) */
   0x95,0x03, /* Report Count (3)                  */
   0x91,0x01, /* Output (Constant)                 */
   0xC0       /* End Collection                    */
};
```

Для передачи состояния клавиатуры (кода клавиш и состояния индикаторов) используется передача данных по первой конечной точке (листинг 10.14). Как видно из дескриптора, код нажатых клавиш передается в масси-

ве, начиная с третьего байта. Коды клавиш соответствуют HID-кодам соответствующих значений в элементах `Usage Index`, например:

- клавиша <a> имеет код `0x04`;
- клавиша <b> имеет код `0x05`;
- клавиша <F1> имеет код `0x3A`.

Листинг 10.14. Передача кода нажатой клавиши

```
void Fill_EP1_FIFO()
{
 /* устройство не готово к передаче данных */
 if ((usb_configuration_nb == 0) || (Usb_suspend()))
  return;

 if (!end_point1_ready) /* предыдущий пакет отправлен */
 {
    Usb_select_ep(1);
     Usb_write_byte(0x00);
     Usb_write_byte(0x00);
     if (!P1.0) {
      Usb_write_byte(4);  /* 4=a, 5=b,...*/
     } else {
      Usb_write_byte(0x00);
     }
     Usb_write_byte(0x00);
     Usb_write_byte(0x00);
     Usb_write_byte(0x00);
     Usb_write_byte(0x00);
     Usb_write_byte(0x00);
   Usb_set_tx_ready();
   end_point1_ready = TRUE;
 }
}
```

Кроме того, клавиатура может обрабатывать запрос `SET_REPORT` для получения сведений об изменениях в состоянии индикаторов.

Заметим также, что приведенный дескриптор клавиатуры отличается от приведенного в листинге 6.4. Здесь демонстрируется одно из преимуществ HID-протокола: конечная программа (в данном случае драйвер клавиатуры) интерпретирует данные согласно дескриптору, а HID-устройство может передавать их в удобном для него порядке. В следующих разделах мы будем обсуждать этот вопрос более подробно.

10.8. Использование HID-протокола

Ранее (*см. гл. 6*) мы приводили достаточно обширное описание HID-протокола, но до сих пор мы пользовались только минимальным набором функций. В этом разделе мы разберем функции, делающие HID-протокол по настоящему гибким.

До начала обсуждения необходимо сделать несколько замечаний относительно используемых функций. Часть функций, в случае успешного выполнения, возвращает значение `TRUE` (при этом функции могут возвращать как просто `Boolean`, так и `LongBool`). Другая часть функций возвращает специальные HID-коды. Для первой категории проверка правильности выполнения выглядит очевидно, а в случае ошибки код и описание ошибки получаются с помощью вызова:

```
SysErrorMessage(GetLastError)
```

А для функций, возвращающих HID-коды, используется специальный обработчик, код которого показан в листинге 10.15.

Листинг 10.15. Обработка HID-кодов ошибки

```
function GetHIDP_Result(Code : Longint) : String;
begin
 Result:= 'Unknown result code';
 Case Code of
  Longint($00110000):
          Result:= 'HIDP_STATUS_SUCCESS';
  Longint($80110001):
          Result:= 'HHIDP_STATUS_NULL';
  Longint($C0110001):
          Result:= 'HHIDP_STATUS_INVALID_PREPARSED_DATA';
  Longint($C0110002):
          Result:= 'HHIDP_STATUS_INVALID_REPORT_TYPE';
  Longint($C0110003):
          Result:= 'HHIDP_STATUS_INVALID_REPORT_LENGTH';
```

```
  Longint($C0110004):
          Result:= 'HHIDP_STATUS_USAGE_NOT_FOUND';
  Longint($C0110005):
          Result:= 'HHIDP_STATUS_VALUE_OUT_OF_RANGE';
  Longint($C0110006):
          Result:= 'HHIDP_STATUS_BAD_LOG_PHY_VALUES';
  Longint($C0110007):
          Result:= 'HHIDP_STATUS_BUFFER_TOO_SMALL';
  Longint($C0110008):
          Result:= 'HHIDP_STATUS_INTERNAL_ERROR';
  Longint($C0110009):
          Result:= 'HHIDP_STATUS_I8042_TRANS_UNKNOWN';
  Longint($C011000A):
          Result:= 'HHIDP_STATUS_INCOMPATIBLE_REPORT_ID';
  Longint($C011000B):
          Result:= 'HHIDP_STATUS_NOT_VALUE_ARRAY';
  Longint($C011000C):
          Result:= 'HHIDP_STATUS_IS_VALUE_ARRAY';
  Longint($C011000D):
          Result:= 'HHIDP_STATUS_DATA_INDEX_NOT_FOUND';
  Longint($C011000E):
          Result:= 'HHIDP_STATUS_DATA_INDEX_OUT_OF_RANGE';
  Longint($C011000F):
          Result:= 'HHIDP_STATUS_BUTTON_NOT_PRESSED';
  Longint($C0110010):
          Result:= 'HHIDP_STATUS_REPORT_DOES_NOT_EXIST';
  Longint($C0110020):
          Result:= 'HHIDP_STATUS_NOT_IMPLEMENTED';
 End;
End;
```

Для указания типа репорта используются следующие константы:

```
HidP_Input   = 0;
HidP_Output  = 1;
HidP_Feature = 2;
```

10.8.1. Интерпретация данных

Снова вернемся к дескриптору, используемому при передаче одного байта данных (листинг 10.16).

Листинг 10.16. Дескриптор для передачи одного байта данных

```
code struct{
  byte  rep[SIZE_OF_REPORT];
 }
 HIDReport =
    /* HID Report */
    {
      0x06, 0x00, 0xff,  /* USAGE_PAGE (Generic Desktop)      */
      0x09, 0x01,        /* USAGE (Vendor Usage 1)            */
      0xa1, 0x01,        /* COLLECTION (Application)          */
      0x19, 0x01,        /*   USAGE_MINIMUM (Vendor Usage 1)  */
      0x29, 0x01,        /*   USAGE_MAXIMUM (Vendor Usage 1)  */
      0x75, 0x08,        /*   REPORT_SIZE (8)                 */
      0x95, 0x01,        /*   REPORT_COUNT(1)                 */
      0x81, 0x02,        /*   INPUT (Data,Var,Abs)            */
      0xc0               /* END_COLLECTION                    */
    };
```

Модифицируем этот дескриптор, добавив в него описание минимума и максимума передаваемого значения:

```
      0x15, 0x00,        /*   LOGICAL_MINIMUM (0)   */
      0x25, 0x64,        /*   LOGICAL_MAXIMUM (100) */
      0x35, 0x00,        /*   P_MINIMUM (0)         */
      0x46, 0xC8, 0x00,  /*   P_MAXIMUM (200)       */
```

Легко видеть, что реальные значения должны находиться в пределах от 0 до 200, а логические — от 0 до 100, т. е. логические значения в два раза больше. Однако наша программа чтения данных будет показывать обычные, непреобразованные, величины. Для чтения *преобразованных величин* (scaled value) используется специальная функция `HidP_GetScaledUsageValue`. Разумеется, эта функция работает, только если пакет данных описан как `variable`.

Вызов функции выглядит следующим образом:

```
var
  UsageValue  : Integer;
  Usage, UsagePage, ReportLen : Word;

 UsageValue:= 0;
 Result:= GetHIDP_Result( HidP_GetScaledUsageValue(
     HidP_Input,    // тип репорта
     UsagePage ,    // Usage Page, как в дескрипторе
     0,             // порядковый номер коллекции
     Usage,         // Usage, как в дескрипторе
     UsageValue,    // результат
     PreparsedData, // блок данных, полученных из HidP_GetCaps
     InputReport,   // репорт
     ReportLen      // длина репорта
));
```

Порядковый номер коллекции используется в том случае, когда в репорте присутствуют две или более коллекций с одинаковыми значениями `Usage Page` и `Usage`. Вызов `HidP_GetCaps` мы приводили в листинге 10.9. Чтение репорта (в данном случае Input) производится с помощью функции `ReadFile` (листинг 10.17).

Листинг 10.17. Чтение Input-репорта

```
FillChar(InputReport, SizeOf(InputReport), #0);
If not ReadFile(HidHandle, InputReport,
                             ReportLen, BytesRead, nil) then begin
  // Ошибка ReadFile: SysErrorMessage(GetLastError)
End;
```

Как видно, переменная результата `UsageValue` имеет тип `Integer`, поэтому при передаче данных можно использовать как двухбайтовые значения, так и четырехбайтовые (листинги 10.18, 10.19).

Листинг 10.18. Передача двухбайтовых значений

```
// часть дескриптора
0x15, 0x00,        /*   LOGICAL_MINIMUM (0)            */
0x26, 0xE8, 0x03,  /*   LOGICAL_MAXIMUM (1000)         */
```

```
0x35, 0x00,        /*   P_MINIMUM (0)                  */
0x46, 0xF4, 0x01,  /*   P_MAXIMUM (500)                */
0x75, 0x10,        /*   REPORT_SIZE (16)               */
// передача двухбайтового значения
Usb_select_ep(1);
Usb_write_byte(0x90); /* 0x0190 = 400 преобразуется в 200 */
Usb_write_byte(0x01);
Usb_set_tx_ready();
```

Листинг 10.19. Передача четырехбайтовых значений

```
// часть дескриптора
0x15, 0x00,                    /*   L_MINIMUM (0)       */
0x27, 0x00, 0x71, 0x02, 0x00,  /*   L_MAXIMUM (160 000) */
0x35, 0x00,                    /*   P_MINIMUM (0)       */
0x47, 0x80, 0x38, 0x01, 0x00,  /*   P_MAXIMUM (80 000)  */
0x75, 0x20,                    /*   REPORT_SIZE (32)    */
// передача 4-байтового значения
Usb_select_ep(1);
Usb_write_byte(0x45); /* 0x00012345 = 74565 преобразуется в 37282 */
Usb_write_byte(0x23);
Usb_write_byte(0x01);
Usb_write_byte(0x00);
Usb_set_tx_ready();
```

10.8.2. Коллекции

Все дескрипторы, которые мы рассматривали, имеют в описании два элемента, определяющих *коллекцию* (collection): открывающий элемент `Collection` и закрывающий элемент `End Collection`. Эти элементы могут использоваться в репортах любого типа для объединения связанных элементов в группы. Спецификация определяет три типа коллекций: *прикладные коллекции* (application collection), *физические* (physical) и *логические* (logical), но можно определять и свои типы коллекций.

Одним из наиболее интересных свойств коллекций является возможность вкладывать коллекции друг в друга, создавая древовидное представление данных.

В дескрипторе, приведенном в листинге 10.20, логическая коллекция вложена в прикладную коллекцию.

Листинг 10.20. Вложенные коллекции

```
0x06, 0x00, 0xff,    /* USAGE_PAGE (Generic Desktop)     */
0x09, 0x01,          /* USAGE (Vendor Usage 1)           */
0xA1, 0x01,          /* COLLECTION (Application)         */
 0xA1, 0x02,         /*  COLLECTION (Logic)              */
 0x05, 0x09,         /*   USAGE_PAGE (Button)            */
 0x19, 0x01,         /*   USAGE_MINIMUM (Button 1)       */
 0x29, 0x08,         /*   USAGE_MAXIMUM (Button 8)       */
 0x15, 0x00,         /*   LOGICAL_MINIMUM (0)            */
 0x25, 0x01,         /*   LOGICAL_MAXIMUM (1)            */
 0x75, 0x01,         /*   REPORT_SIZE (1)                */
 0x95, 0x08,         /*   REPORT_COUNT (8)               */
 0x81, 0x02,         /*   INPUT (Data,Var,Abs)           */
 0xC0,               /*  END_COLLECTION                  */
0x05, 0x01,          /*  USAGE_PAGE (Generic)            */
0x19, 0x01,          /*  USAGE_MINIMUM (Vendor Usage 1) */
0x29, 0x01,          /*  USAGE_MAXIMUM (Vendor Usage 1) */
0x15, 0x00,          /*  LOGICAL_MINIMUM (0)             */
0x25, 0x55,          /*  LOGICAL_MAXIMUM (0x55)          */
0x75, 0x08,          /*  REPORT_SIZE (8)                 */
0x95, 0x07,          /*  REPORT_COUNT(7)                 */
0x81, 0x02,          /*  INPUT (Data,Var,Abs)            */
0xC0                 /* END_COLLECTION                   */
```

Чтение коллекций Windows-приложениями производится с помощью функции `HidP_GetLinkCollectionNodes`, возвращающей связанный список коллекций (листинг 10.21). Обработка этого списка производится с помощью рекурсивной процедуры, код которой показан в листинге 10.22.

Листинг 10.21. Чтение списка коллекций

```
  PHIDPLinkCollectionNode = ^THIDPLinkCollectionNode;
  THIDPLinkCollectionNode = record
    LinkUsage:         Word;
    LinkUsagePage:     Word;
    Parent:            Word;
    NumberOfChildren: Word;
    NextSibling:       Word;
    FirstChild:        Word;
    CollectionType:    BYTE;
    IsAlias:           BYTE;
    Reserved:          Word;
    UserContext:       Pointer;
  end;

// Тип данных для хранения списка коллекций
type
   THIDPLinkCollection =  array of THIDPLinkCollectionNode;

procedure THIDDeviceDataCollectionsForm.CreateTree;
var HidHandle : THandle;
    PreparsedData: PHIDPPreparsedData;
    Capabilities : HIDP_CAPS;
    NumberLinkCollectionNodes : DWORD;
begin
 // Открываем устройство
 HidHandle:= CreateFile(PChar(@HidName[1]),
               0,
               FILE_SHARE_READ or FILE_SHARE_WRITE,
               nil,
               OPEN_EXISTING, 0, 0
 );
 // Если не получилось - ошибка и выход
 If HidHandle = INVALID_HANDLE_VALUE then begin
  StatusBar.Panels[0].Text:= 'Ошибка открытия устройства';
```

```
  Exit;
 End;
 // Получаем блок описания устройства
 If HidD_GetPreparsedData(HidHandle, PreparsedData) then begin
  HidP_GetCaps(PreparsedData, Capabilities);
  NumberLinkCollectionNodes:= Capabilities.NumberLinkCollectionNodes;
 // создаем массив коллекций
  SetLength(FLinkCollection, NumberLinkCollectionNodes);
 // получаем список коллекций
  HidP_GetLinkCollectionNodes(@FLinkCollection[0],
               NumberLinkCollectionNodes, PreparsedData);
  // вызываем отображение коллекций
  ShowCollection(FLinkCollection, NumberLinkCollectionNodes);
 End else begin
  StatusBar.Panels[0].Text:= 'Ошибка HidD_GetPreparsedData';
 End;

 // Освободить блок PreparsedData
 HidD_FreePreparsedData(PreparsedData);
 // Освободить дескриптор устройства
 CloseHandle(HidHandle);
end;
```

Листинг 10.22. Отображение списка коллекций (рекурсивная процедура)

```
procedure THIDDeviceDataCollectionsForm.ShowCollection(
    LinkCollection : THIDPLinkCollection;
    NumberLinkCollectionNodes : Integer);
var ParentNode : TTreeNode; Index : Integer;
begin
 // Добавляем основной узел коллекции
 ParentNode:= CollectionTree.Items.Add(nil, HIDName);

 // Рекурсивный обход
 Index:= 0;
 Repeat
```

```
  EnumerateNodes(LinkCollection, Index, ParentNode,
                      LinkCollection[Index].NumberOfChildren);
  Index:= LinkCollection[Index].NextSibling;
 Until (Index = 0);
end;

procedure THIDDeviceDataCollectionsForm.EnumerateNodes(
      LinkCollection : THIDPLinkCollection;
      Index : Integer;
      Parent : TTreeNode;
      NumberOfChildren : Integer);
   // Возвращает строку описания узла и добавляет его в дерево
   Function AddNode(
           ACollNode : THIDPLinkCollectionNode;
           AParent : TTreeNode) : TTreeNode;
   Var S : String;
   Begin
    S:= FormatCollectionNode(LinkCollection[Index]);
    Result:= CollectionTree.Items.AddChild(Parent, S);
    With LinkCollection[Index] do
     Result.Data:= Pointer(Longint(LinkUsage) +
                        Longint(LinkUsagePage shl 16));
   End;
var i : Word; Node : TTreeNode;
begin
 // Добавляем узел
 Node:= AddNode(LinkCollection[Index], Parent);

 // Обходим всех детей, если они есть
 If NumberOfChildren = 0 then Exit;

 For i:= 0 to NumberOfChildren-1 do begin
   If LinkCollection[Index].FirstChild <> 0 then
    EnumerateNodes(LinkCollection, LinkCollection[Index].FirstChild,
                Node, LinkCollection[Index].NumberOfChildren);
```

```
    if LinkCollection[Index].NextSibling <> 0 then
      Index:= LinkCollection[Index].NextSibling
    else
      Break;
 End;
End;
```

Обход узлов дерева производится сверху вниз и слева направо. При этом каждый узел имеет следующие поля (листинг 10.21):

- `LinkUsage` — тег `Usage`;
- `LinkUsagePage` — тег `Usage Page`;
- `Parent` — номер коллекции родителя;
- `NumberOfChildren` — число дочерних коллекций;
- `NextSibling` — индекс следующей коллекции;
- `FirstChild` — индекс первой дочерней коллекции;
- `CollectionType` — тип коллекции;

Правило обхода коллекций проще всего понять на примере рисунка, взятого из MSDN (рис. 10.3). В приведенном на рисунке случае обход коллекций будет осуществляться в порядке ABCD.

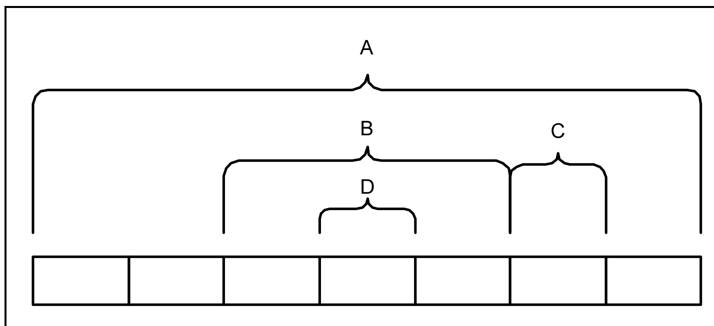


Рис. 10.3. Структура вложенных коллекций

Для нашего HID-устройства программа отобразит одну вложенную коллекцию (рис. 10.4). Полный код тестовой программы можно найти на компакт-диске.

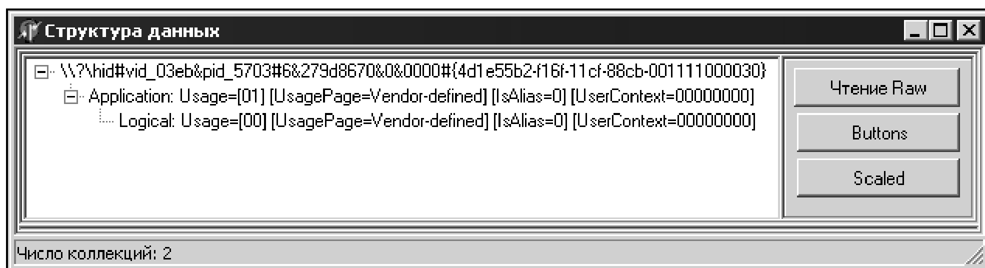


Рис. 10.4. Программа отображения коллекций

10.8.3. Массивы и кнопки

Ранее (*см. разд. 10.7.2*), при описании дескриптора клавиатуры, мы использовали описание массива:

```
0x19,0x00, /* Usage Minimum (0)                */
0x29,101,  /* Usage Maximum (101)              */
0x15,0x00, /* Logical Minimum (0)              */
0x25,101,  /* Logical Maximum (101)            */
0x75,0x08, /* Report Size (8)                  */
0x95,0x06, /* Report Count (6)                 */
0x81,0x00, /* Input (Data, Array)              */
```

В общем случае объединение объектов в массив означает, что каждый из них имеет индекс, который передается в репорте, если данный объект выбран. Например, каждая клавиша имеет индекс, который передается в репорте, если она нажата. В нашем примере индекс будет кодироваться восемью битами, поэтому поле `Report Size` должно быть равно восьми, а `Report Count` равно числу клавиш, которые разрешено нажимать одновременно.

Рассмотрим еще один пример:

```
0x05, 0x07,        /* USAGE_PAGE (Generic Desktop)      */
0x09, 0x01,        /* USAGE (Vendor Usage 1)            */
0xa1, 0x01,        /* COLLECTION (Application)          */
0x19, 0x01,        /*   USAGE_MINIMUM (Vendor Usage 1)  */
0x29, 0x65,        /*   USAGE_MAXIMUM (Keyboard)        */
0x15, 0x00,        /*   LOGICAL_MINIMUM (0)             */
0x25, 0x06,        /*   LOGICAL_MAXIMUM (6)             */
```

```
    0x75, 0x04,        /*   REPORT_SIZE (4)                 */
    0x95, 0x08,        /*   REPORT_COUNT(8)                 */
    0x81, 0x00,        /*   INPUT (Data,Arr,Abs)            */
    0xc0               /* END_COLLECTION                    */
```

В этом дескрипторе каждая кнопка кодируется четырьмя битами, одновременно допускается нажимать до восьми кнопок. Очевидно, что один байт содержит информацию о двух кнопках, а размер пакета будет равен четырем байтам. Соответственно при передаче данных один из индексов сдвигается влево на четыре бита.

Хотя массив и содержит только коды нажатых клавиш, передача данных должна содержать полный пакет, заполненный нулями или индексами нажатых клавиш (по крайней мере, для Windows). Другими словами, для нашего примера мы должны передавать либо четыре нуля (или правильнее будет сказать — восемь "полунулей"), либо какие-то из индексов, причем не важно на каком месте и в каком порядке они будут указаны:

```
  Usb_write_byte(0);
  Usb_write_byte(2+(3<<4)); // передаем индексы 2 и 3
  Usb_write_byte(0);
  Usb_write_byte(0);
```

> ***Важно!***
>
> В любом случае, независимо от количества нажатых клавиш размер передаваемого пакета должен быть равен четырем байтам.

Вызов функции `ReadFile`, которой мы уже пользовались для чтения пакетов, вернет нам совершенно корректный набор данных:

```
FillChar(InputReport, SizeOf(InputReport), #0);
If not ReadFile(HidHandle, InputReport, ReportLen,
                 BytesRead, nil) then begin
  Memo1.Lines.Add('Ошибка ReadFile
                  ('+SysErrorMessage(GetLastError)+')');
 End;
 S:= 'INPUT:';
 For i:= 0 to ReportLen-1 do
   S:= S + Format(' %d ', [InputReport[i]]);
```

В данном примере мы получим строку

```
INPUT: 0  0  50  0  0
```

А вот для получения собственно кодов клавиш можно воспользоваться специальной функцией `HidP_GetUsages`. Перед ее вызовом следует определить размер возвращаемого буфера с помощью функции `HidP_MaxUsageListLength`. Пример вызова показан в листинге 10.23.

Листинг 10.23. Чтение массива кодов клавиш

```
MaxUsageListLength:= HidP_MaxUsageListLength(
   HidP_Input,
   UsagePage,
   PreparsedData
 );

If MaxUsageListLength > 0 then begin
   UsageLength:= MaxUsageListLength;
   SetLength(UsageList, UsageLength);
   S:= GetHIDP_Result( HidP_GetUsages(
         HidP_Input,
         UsagePage ,
         0, // LinkCollection: Word;
         @UsageList[0],
         UsageLength,
         PreparsedData,
         InputReport,
         ReportLen));
   If UsageLength > 0 then begin
    S:= 'UsageList:';
    For i:= 0 to UsageLength-1 do
     S:= S + Format(' %d ', [Word(UsageList[i])]);
   End;
 End;
```

В нашем случае строка `S` будет содержать два числа, являющихся переданными кодами клавиш:

```
UsageList: 2 3
```

Полный код этого примера можно найти на компакт-диске.

10.9. HID-устройство с несколькими репортами

Спецификация HID позволяет создавать устройства с несколькими репортами. Пример дескриптора такого HID-устройства показан в листинге 10.24.

Листинг 10.24. Дескриптор устройства с двумя репортами

```
code struct{
  byte  rep[SIZE_OF_REPORT];
 }
 HIDReport =
    /* HID Report */
    {
      0x06, 0x00, 0xff,  /* USAGE_PAGE (Generic Desktop)      */

      0x09, 0x01,        /* USAGE (Vendor Usage 1)            */
      0xa1, 0x01,        /* COLLECTION (Application)          */
      0x19, 0x01,        /*   USAGE_MINIMUM (Vendor Usage 1)  */
      0x29, 0x01,        /*   USAGE_MAXIMUM (Vendor Usage 1)  */
      0x15, 0x00,        /*   LOGICAL_MINIMUM (0)             */
      0x26, 0xff, 0x00,  /*   LOGICAL_MAXIMUM (255)           */
      0x85, 0x01,        /*   REPORT_ID (1)                   */
      0x75, 0x08,        /*   REPORT_SIZE (8)                 */
      0x95, 0x07,        /*   REPORT_COUNT(7)                 */
      0x81, 0x02,        /*   INPUT (Data,Var,Abs)            */
      0xc0               /* END_COLLECTION                    */
      ,
      0x09, 0x01,        /* USAGE (Vendor Usage 1)            */
      0xa1, 0x01,        /* COLLECTION (Application)          */
      0x19, 0x01,        /*   USAGE_MINIMUM (Vendor Usage 1)  */
      0x29, 0x01,        /*   USAGE_MAXIMUM (Vendor Usage 1)  */
      0x15, 0x00,        /*   LOGICAL_MINIMUM (0)             */
      0x26, 0xff, 0x00,  /*   LOGICAL_MAXIMUM (255)           */
      0x85, 0x02,        /*   REPORT_ID (2)                   */
      0x75, 0x08,        /*   REPORT_SIZE (8)                 */
```

```
    0x95, 0x04,         /*   REPORT_COUNT(4)                    */
    0x81, 0x02,         /*   INPUT (Data,Var,Abs)               */
    0xc0                /* END_COLLECTION                       */
  };
```

Как видно из листинга, мы используем два репорта:

- первый, с индексом `1`, передает семь байт;
- второй, с индексом `2`, передает четыре байта.

Соответственно при передаче данных из HID-устройства мы будем передавать пакет данных, согласно `ReportID` (листинг 10.25), который передается в младшем байте поля `wValue` (*см. разд. 6.6.2*).

Листинг 10.25. Передача данных для устройства с двумя репортами

```
void hid_get_report()
{
 if (usb_configuration_nb == 0){
  STALL();
  return;
 }

 Usb_clear_rx_setup();
 Usb_set_DIR();

 switch (SetupPacket.b[2]) /* ReportID */
 {
  case 1:
  {
    Usb_write_byte(0x01);
    Usb_write_byte(0x02);
    Usb_write_byte(point1_state);
    Usb_write_byte(0x04);
    Usb_write_byte(0x05);
    Usb_write_byte(0x06);
    Usb_write_byte(0x07);
    break;
  }
```

```
  case 2:
  {
    Usb_write_byte(0x11);
    Usb_write_byte(0x12);
    Usb_write_byte(point1_state);
    Usb_write_byte(0x14);
    break;
  }
 }
 SendDataFromFIFO();
 WaitForOutZeroPacket();
}
```

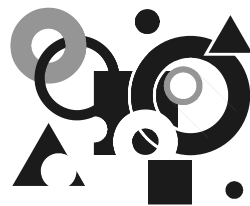
Если нужно, можно анализировать также тип репорта, который передается в байте `SetupPacket.b[3]`.

Со стороны хоста, для указания используемого репорта индекс передаваемого или запрашиваемого репорта записывается в первый байт буфера данных. При передаче данных это выглядит нормально:

```
  FillChar(OutputReport, SizeOf(OutputReport), #0);
 // задаем ReportID
  OutputReport[0]:= 1;
 // заполняем данные
  OutputReport[1]:= 1;
... ... ... ... ... ...
  OutputReport[7]:= 7;
  // передаем
  HidD_SetFeature(HidHandle, OutputReport, FeatureReportLen);
```

А вот при чтении эта конструкция выглядит несколько странно, но к ней нужно просто привыкнуть:

```
  FillChar(InputReport, SizeOf(InputReport), #0);
  // Хотя мы будем читать данные, мы заполняем нулевой байт
  InputReport[0]:= 1; // Report ID
  // Читаем данные
  HidD_GetFeature(HidHandle, InputReport, FeatureReportLen);
```



Глава 11

Специальные функции Windows

В этой главе мы расскажем о специальных функциях Windows (Setup API, WMI и т. п.), а также об использовании функций DirectX для работы с USB-устройствами.

11.1. Функции Setup API

Функции Setup API находятся в библиотеке setupapi.dll. Для работы с ними на языке C необходимо подключить файл setupapi.h. На языке Delphi необходимо либо подключать нужные функции самостоятельно (как показано в листинге 11.1), либо использовать модуль SetupApi.pas из библиотеки JEDI.

Листинг 11.1. Подключение функций Setup API в Delphi

```
function SetupDiGetClassDevs(ClassGuid: PGUID; const Enumerator: PChar;
hwndParent: HWND; Flags: DWORD): HDEVINFO; stdcall;

{$EXTERNALSYM SetupDiGetClassDevs}

function SetupDiGetClassDevs; external SetupApiModuleName name
'SetupDiGetClassDevs';
```

На языке C# (и вообще для платформы .NET) придется использовать импорт функций Win32 (*см. гл. 4*).

11.1.1. Перечисление USB-устройств

Как мы уже говорили, физическое имя устройства в Windows это не просто COM1 или F: — имя представляет собой PnP-идентификатор, выглядящий примерно так:

```
\\?\hid#vid_1241&pid_1111#6&30e75ab0&0&0000#{4d1e55b2-f16f-11cf-88cb-
001111000030}
```

Это имя присваивается менеджером PnP. Для его получения можно использовать функции Setup API (или функции WMI, *см. разд. 11.2*).

Для получения списка устройств используется следующая последовательность действий:

1. Получение дескриптора класса устройств (дескриптор либо корневого класса, либо класса с конкретным GUID) с помощью вызова функции `SetupDiGetClassDevs`.
2. Вызов функции `SetupDiEnumDeviceInfo` для получения описателя очередного устройства.
3. Вызов функции `SetupDiGetDeviceRegistryProperty` для получения информации об устройстве.
4. Повторение шагов 2—3 до тех пор, пока функция `SetupDiEnumDeviceInfo` не вернет `FALSE`.
5. Освобождение дескриптора с помощью вызова `SetupDiDestroyDeviceInfoList`.

Для получения списка всех устройств функции `SetupDiGetClassDevs` вместо идентификатора класса передается `nil`, а в поле флагов записывается `DIGCF_ALLCLASSES`, а для получения дескриптора конкретного класса в эту функцию передается идентификатор нужного класса (см. табл. 13.1).

Для примера мы создадим небольшую программу, позволяющую получить либо список всех устройств в системе, либо устройства одного из следующих классов:

- видеоадаптеры;
- USB-устройства;
- HID-устройства.

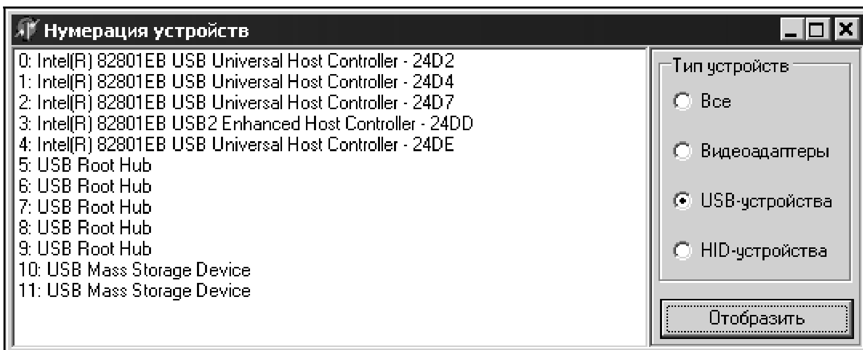


Рис. 11.1. Программа получения списка устройств

Получение списка видеоадаптеров сделано исключительно для упрощения тестирования (USB-устройства могут отсутствовать, а видеоадаптер есть всегда). Отметим также, что классы USB-устройств и HID-устройств различаются. К первому классу относятся хост, корневой хаб, флэш-диски и т. д., а HID-устройства мы рассматривали ранее (*см. гл. 6 и 10*).

Наша тестовая программа будет состоять из одной формы, показанной на рис. 11.1, а исходный код на языке Delphi приведен в листинге 11.2.

Листинг 11.2. Получение списка устройств с помощью функций Setup API (Delphi)

```
unit Unit1;

interface

uses
 Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
 Forms, Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    Panel1: TPanel;
    rgDeviceType: TRadioGroup;
    Button2: TButton;
    procedure Button2Click(Sender: TObject);
  private
  public
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

Uses SetupApi;
```

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Guid : TGUID;
  PnPHandle: HDevInfo;
  DeviceInfoData : SP_DEVINFO_DATA;
  DeviceInterfaceData: SP_DEVICE_INTERFACE_DATA;
  i : Cardinal;
  Success: LongBool;
  DataT, buffersize: Cardinal;
  buffer : PByte;
Const
  // GUID класса "видеоадаптеры"
  DisplayGuid : TGUID = '{4d36e968-e325-11ce-bfc1-08002be10318}';
  // GUID класса "HID-устройства"
  HidGuid     : TGUID = '{745a17a0-74d3-11d0-b6fe-00a0c90f57da}';
  // GUID класса "USB-устройства"
  USBGuid     : TGUID = '{36FC9E60-C465-11CF-8056-444553540000}';
begin
 // очистка лога
 ListBox1.Clear;

 If rgDeviceType.ItemIndex = 0 then begin  // Все устройства
  PnPHandle:= SetupDiGetClassDevs(nil, nil, 0,
                  DIGCF_PRESENT or DIGCF_ALLCLASSES);
 End else begin
  // Устройства конкретного класса
  Case rgDeviceType.ItemIndex of
   1: Guid:= DisplayGuid; // Все видеоадаптеры
   2: Guid:= USBGuid;     // Все USB-устройства
   3: Guid:= HidGuid;     // Все HID-устройства
  End;
  PnPHandle:= SetupDiGetClassDevs(@Guid, nil, 0, DIGCF_PRESENT);
 End;

 // Цикл по всем устройствам класса
 Try
```

```
i:= 0;
DeviceInfoData.cbSize:= SizeOf(SP_DEVINFO_DATA);
DeviceInterfaceData.cbSize := SizeOf(SP_DEVICE_INTERFACE_DATA);
Repeat
 // Получаем очередное устройство
 Success:= SetupDiEnumDeviceInfo(PnPHandle, i, DeviceInfoData);
 If Success then begin
  buffer:= nil;
  buffersize:= 0;
  // Получаем информацию об устройстве
  while not SetupDiGetDeviceRegistryProperty
        (
          PnPHandle,
          DeviceInfoData,
          SPDRP_DEVICEDESC,
          DataT,
          buffer,
          buffersize,
          buffersize
        )
  do begin
   if (GetLastError() = ERROR_INSUFFICIENT_BUFFER) then begin
    if (buffer <> nil) then FreeMem(buffer);
    buffer:= AllocMem(buffersize);
   end else begin
    break;
   end;
  end;
  // Отобразить информацию об устройстве
  ListBox1.Items.Add(Format('%d: %s',[i, StrPas(PChar(buffer))]));
  if (buffer <> nil) then FreeMem(buffer);
 End;
 Inc(i);
 Application.ProcessMessages;
until not Success;
```

```
 Finally
  // Освободить дескриптор класса
  SetupDiDestroyDeviceInfoList(PnPHandle);
 End;
end;
end.
```

Немного странный цикл при вызове функции `SetupDiGetDeviceRegistry-Property` объясняется довольно просто: первый вызов этой функции производится с нулевым размером буфера (`buffersize:= 0`), поэтому функция возвращает ошибку "недостаточный размер буфера" (`ERROR_INSUFFICIENT_BUFFER`) и нужный размер буфера в переменной `buf-fersize`. На втором проходе функции передается буфер нужного размера. Результат выполнения программы виден на рис. 11.1.

Код на языке C# выглядит не многим сложнее (листинг 11.3). Единственная сложность — корректное описание импорта необходимых функций[1]. Следует также помнить о правилах использования функций Win32 на платформе .NET (*см. гл. 4*).

Листинг 11.3. Получение списка устройств с помощью функций Setup API (C#)

```
using System;
using System.Runtime.InteropServices;

namespace DeviceEnumerator
{
 class Class1
 {
  [STAThread]
  static unsafe void Main(string[] args)
  {
   // получаем дескриптор (передаем null - получение
   // всех устройств системы)
   int PnPHandle = SetupAPI.SetupDiGetClassDevs(
    null,
```

[1] Для экономии места мы не будем приводить здесь код импорта функций. Соответствующие описания можно найти в справочной части книги.

```
 null,
 null,
 SetupAPI.ClassDevsFlags.DIGCF_ALLCLASSES |
                  SetupAPI.ClassDevsFlags.DIGCF_PRESENT
);

int result  = -1;
int DeviceIndex = 0;
// цикл по устройствам
while (result != 0)
{
 // получение информации об устройстве
 SetupAPI.SP_DEVINFO_DATA DeviceInfoData =
                                new SetupAPI.SP_DEVINFO_DATA();
 DeviceInfoData.cbSize = Marshal.SizeOf(DeviceInfoData);
 result = SetupAPI.SetupDiEnumDeviceInfo(
     PnPHandle,
     DeviceIndex,
     ref DeviceInfoData
 );

 // Получение свойства DEVICEDESC (описание)
 if (result == 1)
 {
  int RequiredSize = 0;
  SetupAPI.DATA_BUFFER Buffer = new SetupAPI.DATA_BUFFER();
  result = SetupAPI.SetupDiGetDeviceRegistryProperty(
   PnPHandle,
   ref DeviceInfoData,
   SetupAPI.RegPropertyType.SPDRP_DEVICEDESC,
   null,
   ref Buffer,
   1024,
   ref RequiredSize
   );
```

```
      // Печатаем результат
      Console.WriteLine("{0}", Buffer.Buffer);
     }
     // Следующее устройство
     DeviceIndex++;
    }
   }
  }
 }
```

Описание типа буфера выглядит следующим образом:

```
// Device interface detail data
[StructLayout(LayoutKind.Sequential, CharSet= CharSet.Ansi)]
public unsafe struct DATA_BUFFER
{
  [MarshalAs(UnmanagedType.ByValTStr, SizeConst= 1024)]
  public string Buffer;
}
```

Как видно, размер нашего буфера зафиксирован, и поэтому мы не вызываем функцию `SetupDiGetDeviceRegistryProperty` с нулевыми параметрами.

При необходимости получить устройства конкретного класса, как и в Delphi, надо вызвать функцию `SetupDiGetClassDevs` с GUID этого класса (листинг 11.4).

Листинг 11.4. Получение списка USB-устройств и их свойств с помощью функций Setup API (C#)

```
using System;
using System.Runtime.InteropServices;

namespace DeviceEnumerator
{
 class Class1
 {
  [STAThread]
  static unsafe void Main(string[] args)
  {
```

```
// DisplayGuid : TGUID = '{4d36e968-e325-11ce-bfc1-08002be10318}';
// HidGuid     : TGUID = '{745a17a0-74d3-11d0-b6fe-00a0c90f57da}';
// USBGuid     : TGUID = '{36FC9E60-C465-11CF-8056-444553540000}';

   // Создаем GUID класса USB-устройств
   Guid guid = new Guid("{36FC9E60-C465-11CF-8056-444553540000}");
   // Получаем дескриптор
   int PnPHandle = SetupAPI.SetupDiGetClassDevs(
    ref guid,
    null,
    null,
    SetupAPI.ClassDevsFlags.DIGCF_PRESENT
   );

   int result  = -1;
   int DeviceIndex = 0;
   // Цикл по всем устройствам класса
   while (result != 0)
   {
    SetupAPI.SP_DEVINFO_DATA DeviceInfoData =
                    new SetupAPI.SP_DEVINFO_DATA();
    DeviceInfoData.cbSize = Marshal.SizeOf(DeviceInfoData);
    result = SetupAPI.SetupDiEnumDeviceInfo(
         PnPHandle,
         DeviceIndex,
         ref DeviceInfoData
    );

    if (result == 1)
    {
     Console.WriteLine("{0}:\n\t{1}\n\t{2}\n\t{3}\n\t{4}",
      GetRegistryProperty(PnPHandle, ref DeviceInfoData,
                SetupAPI.RegPropertyType.SPDRP_DEVICEDESC),
      GetRegistryProperty(PnPHandle, ref DeviceInfoData,
                SetupAPI.RegPropertyType.SPDRP_CLASS),
      GetRegistryProperty(PnPHandle, ref DeviceInfoData,
```

```
                SetupAPI.RegPropertyType.SPDRP_CLASSGUID),
      GetRegistryProperty(PnPHandle, ref DeviceInfoData,
                SetupAPI.RegPropertyType.SPDRP_DRIVER),
      GetRegistryProperty(PnPHandle, ref DeviceInfoData,
                SetupAPI.RegPropertyType.SPDRP_MFG)
     );
    }
    DeviceIndex++;
   }
  }

public unsafe static string GetRegistryProperty(int PnPHandle, ref
SetupAPI.SP_DEVINFO_DATA DeviceInfoData, SetupAPI.RegPropertyType Property)
  {
   int RequiredSize = 0;
   SetupAPI.DATA_BUFFER Buffer = new SetupAPI.DATA_BUFFER();

   int result = SetupAPI.SetupDiGetDeviceRegistryProperty(
    PnPHandle,
    ref DeviceInfoData,
    Property,
    null,
    ref Buffer,
    1024,
    ref RequiredSize
    );

   return Buffer.Buffer;

  }
 }
}
```

С помощью функции `GetRegistryProperty` можно получить многие другие свойства устройства.

11.1.2. Получение состояния USB-устройства

Функция `CM_Get_DevNode_Status` позволяет получить дополнительную информацию об устройстве. В частности, включено ли устройство в текущий конфигурации оборудования (рис. 11.2).

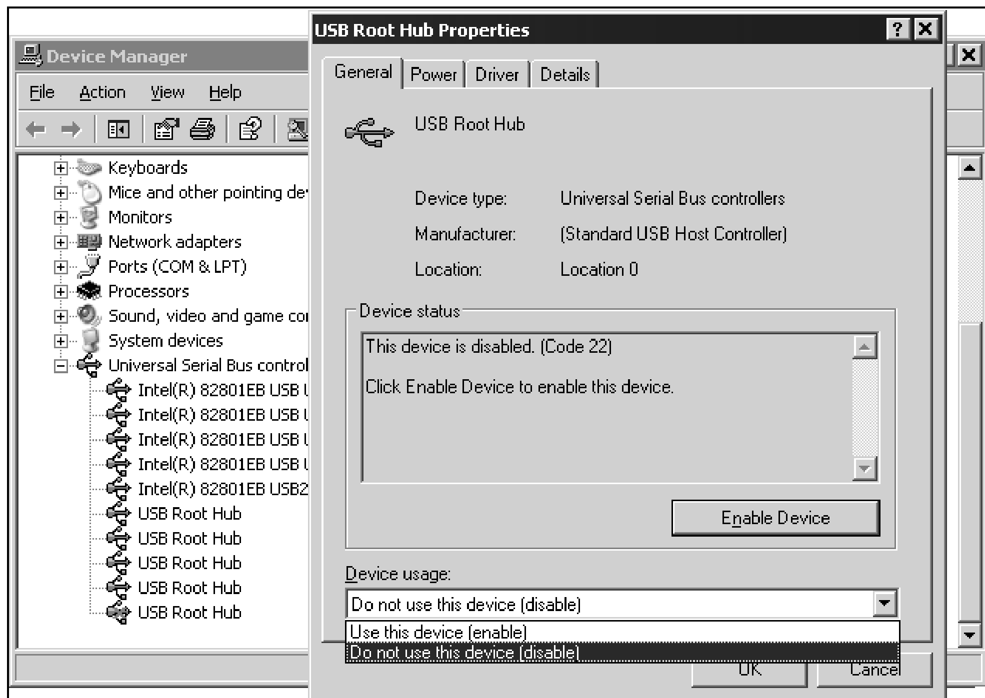


Рис. 11.2. Включение/выключение устройства из конфигурации оборудования

В листинге 11.5 показано получение двух свойств устройства — `IsEnable`, возвращающее `true`, если устройство включено в текущую конфигурацию оборудования, и `IsDisableable`, если выключение устройства доступно.

Листинг 11.5. Получение свойств состояния USB-устройства (C#)

```
using System;
using System.Runtime.InteropServices;

namespace DeviceEnumerator
{
```

```
class Class1
{
 [STAThread]
 static unsafe void Main(string[] args)
 {
  Guid UsbGuid =
           new Guid("{36FC9E60-C465-11CF-8056-444553540000}");

  int PnPHandle = SetupAPI.SetupDiGetClassDevs(
   ref UsbGuid,
   null,
   null,
   SetupAPI.ClassDevsFlags.DIGCF_PRESENT
  );

  int result  = -1;
  int DeviceIndex = 0;

  while (result != 0)
  {
   SetupAPI.SP_DEVINFO_DATA DeviceInfoData =
                new SetupAPI.SP_DEVINFO_DATA();
   DeviceInfoData.cbSize = Marshal.SizeOf(DeviceInfoData);
   result = SetupAPI.SetupDiEnumDeviceInfo(
                PnPHandle, DeviceIndex, ref DeviceInfoData);

   if (result == 1)
   {
    Console.WriteLine("{0}:\n\tIsEnabled={1}\n\tIsDisableable={2}",
     GetRegistryProperty(PnPHandle, ref DeviceInfoData,
                 SetupAPI.RegPropertyType.SPDRP_DEVICEDESC),
     IsEnable(DeviceInfoData),
     IsDisableable(DeviceInfoData)
    );
   }
```

```
    DeviceIndex++;
   }

   Marshal.FreeHGlobal((System.IntPtr)PnPHandle);
  }
... ... ... ... ... ... ... ... ... ... ... ...
  public unsafe static string GetRegistryProperty()
... ... ... ... ... ... ... ... ... ... ... ...

  public unsafe static bool IsEnable(
             SetupAPI.SP_DEVINFO_DATA DevData)
  {
   int Status = 0;
   int Problem = 0;

   SetupAPI.CM_Get_DevNode_Status(ref Status,
                ref Problem, DevData.DevInst, 0);
   return !(((Status & SetupAPI.DN_HAS_PROBLEM)!=0) &&
                    (Problem == SetupAPI.CM_PROB_DISABLED));
  }

  public unsafe static bool IsDisableable(
               SetupAPI.SP_DEVINFO_DATA DevData)
  {
   int Status = 0;
   int Problem = 0;

   SetupAPI.CM_Get_DevNode_Status(ref Status, ref Problem,
                     DevData.DevInst, 0);
   return ((Status & SetupAPI.DN_DISABLEABLE)!=0) &&
             (Problem != SetupAPI.CM_PROB_HARDWARE_DISABLED);
  }
 }
}
```

Описание и формат импорта функции `CM_Get_DevNode_Status` можно найти в справочной части книги.

11.2. Перечисление USB-устройств с помощью WMI

Общие сведения о WMI мы уже приводили (*см. разд. 4.6*), а сейчас попробуем применить эти знания для получения информации о USB-устройствах системы.

Нас будут интересовать три WMI-класса:

- `Win32_USBController` — хранит информацию о USB-контроллерах;
- `Win32_USBHub` — хранит информацию о хабах;
- `Win32_USBControllerDevice` — хранит информацию о USB-устройствах.

Пример работы с этими классами показан в листинге 11.6.

Листинг 11.6. Вызов WMI-функций `Win32_USB`

```
using System;
using System.Management;

namespace WMI1
{
 class Class1
 {
  [STAThread]
  static void Main(string[] args)
  {
   // Получаем объект для доступа к WMI
   ManagementScope sc = new  ManagementScope(@"\\.\root\cimv2", null);

   // Получаем список USB-контроллеров
   Console.WriteLine("\nWin32_USBController:");
   ManagementPath  ph_controller = new  ManagementPath
   (@"Win32_USBController");
   ManagementClass mc_controller = new  ManagementClass(sc,
   ph_controller, null);

   foreach (ManagementObject controller in mc_controller.GetInstances())
   {
```

```
 // Идентификация
 PrintPropety(controller, "Name");
 PrintPropety(controller, "Caption");
 PrintPropety(controller, "SystemName");
 // Описание
 PrintPropety(controller, "Description");
 // Производитель
 PrintPropety(controller, "Manufacturer");
 // ID
 PrintPropety(controller, "DeviceID");
 PrintPropety(controller, "PNPDeviceID");
 // Дополнительные сведения
 PrintPropety(controller, "InstallDate");
 PrintPropety(controller, "Status");
 // WMI-класс
 PrintPropety(controller, "CreationClassName");
 PrintPropety(controller, "SystemCreationClassName");
}

// Получаем список USB-хабов
Console.WriteLine("\nWin32_USBHub:");
ManagementPath  ph_hub = new  ManagementPath (@"Win32_USBHub");
ManagementClass mc_hub = new  ManagementClass(sc, ph_hub, null);

foreach (ManagementObject hub in mc_hub.GetInstances())
{
 Console.WriteLine("Name: {0}", hub.GetPropertyValue("Name"));
}

// Получаем список устройств
Console.WriteLine("\nWin32_USBControllerDevice:");
ManagementPath  ph_dev = new  ManagementPath
(@"Win32_USBControllerDevice");
ManagementClass mc_dev = new  ManagementClass(sc, ph_dev, null);

foreach (ManagementObject dev in mc_dev.GetInstances())
{
```

```
    Console.WriteLine("Antecedent: {0}",
    dev.GetPropertyValue("Antecedent"));
    Console.WriteLine("Dependent : {0}", dev.GetPropertyValue
    ("Dependent" ));
   }

   Console.WriteLine("Press OK for continue...");
   Console.ReadLine();
  }

  // Получение свойства объекта
  private static object GetProperty(ManagementObject obj, string Name)
  {
   return obj.GetPropertyValue(Name);
  }

  // Печать свойства
  private static void PrintPropety(ManagementObject obj, string Name)
  {
   Console.WriteLine("{0}: {1}", Name, GetProperty(obj, Name));
  }
 }
}
```

11.3. Специальные функции Windows XP

В Windows XP было добавлено несколько функций, существенно расширяющих возможности работы с оборудованием, и, в частности, с USB.

11.3.1. *HidD_GetInputReport* — чтение HID-репортов

Функция `HidD_GetInputReport`, появившаяся в Windows XP, позволяет прочитать Input-репорт HID-устройства, аналогично тому, как это делалось с помощью функции `ReadFile`. Пример вызова показан в листинге 11.7.

Листинг 11.7. Чтение репортов с помощью `HidD_GetInputReport`

```
FillChar(InputReport, SizeOf(InputReport), #0);
InputReport[0]:= 0; // можно задать Report ID
If HidD_GetInputReport(HidHandle, @InputReport[0], ReportLen) then
begin
  S:= 'HidD_GetInputReport:';
  For i:= 1 to ReportLen do begin
   S:= S + Format(' %2x', [InputReport[i]]);
  End;
End else begin
  //Ошибка HidD_GetInputReport SysErrorMessage(GetLastError)
End;
```

Аналогично для передачи репортов в Windows XP можно использовать функцию `HidD_SetOutputReport`.

11.3.2. Получение данных Raw Input

В предыдущих моделях работы получение данных от устройств ввода различалось в зависимости от типа устройства. Так, например, ввод с клавиатуры преобразовывался в скан-коды и передавался пользовательской программе в виде сообщения `WM_CHAR`. Ввод мыши преобразовывался в сообщения `WM_MOUSEMOVE`. Дополнительно генерировалось сообщение `WM_APPCOMMAND`. Для остальных устройств алгоритм получения данных другой — необходимо было получить дескриптор устройства, периодически производить чтение устройства или проверять готовность порта и т. д.

Модель Raw Input — попытка унифицировать алгоритм ввода данных от устройств различных типов. Эта модель имеет несколько преимуществ:

- приложению не нужно открывать устройство или порты;
- приложение получает данные от устройства напрямую и обрабатывает их когда это необходимо самому приложению;
- приложение может отличать данные от устройств одинакового типа, например, от двух мышек;
- приложение может выбирать, какие именно данные от устройства нужны этому приложению или данные какого именно типа устройств;
- поддерживаются все типы HID-устройств.

Передача данных согласно модели Raw Input производится через очередь сообщений Windows с помощью сообщения `WM_INPUT`, которое имеет код `0x00FF`. Для облегчения работы выделяются три типа данных, передаваемых этим сообщением: данные мыши (код 0), сообщения клавиатуры (код 1) и сообщения других HID-устройств (код 2):

```
TRawInputType = (RIM_TYPEMOUSE, RIM_TYPEKEYBOARD, RIM_TYPEHID);
```

По умолчанию приложение не получает никаких данных Raw Input, для их получения приложение должно произвести подписку на интересущий его набор данных. Подписка производится с помощью вызова функции `RegisterRawInputDevices`. Ее описание для языка Delphi имеет вид:

```
function RegisterRawInputDevices(
    pRawInputDevices : PRawInputDevice;
    puiNumDevices : Integer;
    cbSize : Integer) : LongBool; stdcall;
function RegisterRawInputDevices;
external 'user32' name 'RegisterRawInputDevices';
```

Для языка C описание выглядит следующим образом:

```
WINUSERAPI BOOL WINAPI RegisterRawInputDevices(
    IN PCRAWINPUTDEVICE pRawInputDevices,
    IN UINT uiNumDevices,
    IN UINT cbSize
);
```

Первым параметром этой функции является набор структур `RawInputDevice`, описывающих один из наборов данных. Второй параметр передает количество таких структур, а третий — размер структуры `RawInputDevice`.

Структура `RawInputDevice` описывает тип данных, на который производится подписка:

```
Type
 PRawInputDevice = ^TRawInputDevice;
 TRawInputDevice = record
   UsagePage : Word;
   Usage     : Word;
   Flags     : DWord;
   Target    : THandle;
 End;
```

Поля `UsagePage` и `Usage` соответствуют полям в HID-репорте, параметр `Target` должен содержать дескриптор окна, которому будет посылаться со-

общение WM_INPUT, а параметр Flags содержит дополнительные требования к регистрационным данным:

- RIDEV_APPKEYS — доступно начиная с Windows XP SP1. Если этот флаг установлен, приложение полностью обрабатывает команды клавиатуры. Этот флаг может быть установлен только совместро с флагом RIDEV_NOLEGACY для клавиатуры;
- RIDEV_CAPTUREMOUSE — если этот флаг выставлен, нажатие на клавиши мыши не активизирует другие окна;
- RIDEV_EXCLUDE, RIDEV_PAGEONLY — будет разбираться далее;
- RIDEV_INPUTSINK — если этот флаг выставлен, приложение будет получать сообщения, даже если оно не активно (не имеет фокуса);
- RIDEV_NOHOTKEYS — если этот флаг выставлен, приложение не перехватывает "горячие" комбинации клавиш, такие как <Alt>+<Tab> и <Ctrl>+ +<Alt>+<Del>;
- RIDEV_NOLEGACY — может устанавливаться для мыши и клавиатуры. Если этот флаг установлен, система не будет генерировать "старые" сообщения, такие как WM_KEYDOWN, WM_LBUTTONDOWN и т. п.;
- RIDEV_REMOVE — удаляет подписку на выбранный тип данных.

С помощью флагов RIDEV_PAGEONLY и RIDEV_EXCLUDE подписку на данные можно производить более гибко. Флаг RIDEV_PAGEONLY подписывает приложение на получение всех данных для указанного UsagePage (при этом поле Usage должно быть равно нулю). Флаг RIDEV_EXCLUDE отключает подписку для некоторого набора данных. Таким образом, можно подписаться на "всю страницу", а затем отключить подписку на какие-то конкретные типы данных.

Константы флагов описываются следующим образом:

```
RIDEV_REMOVE        0x00000001;
RIDEV_EXCLUDE       0x00000010;
RIDEV_PAGEONLY      0x00000020;
RIDEV_NOLEGACY      0x00000030;
RIDEV_INPUTSINK     0x00000100;
RIDEV_CAPTUREMOUSE  0x00000200;
RIDEV_NOHOTKEYS     0x00000200;
RIDEV_APPKEYS       0x00000400;
RIDEV_EXMODEMASK    0x000000F0;
```

Пример вызова регистрационной функции показан в листинге 11.8, а полный код этой программы можно найти на компакт-диске.

Листинг 11.8. Регистрация на получение данных

```
function THIDDeviceReadRawInputForm.GetFlagsValue : Cardinal;
begin
 Result:= 0;
 // Flags - элемент типа TCheckListBox
 If Flags.Checked[0] then Result:= Result or $00000001;
 If Flags.Checked[1] then Result:= Result or $00000010;
 If Flags.Checked[2] then Result:= Result or $00000020;
 If Flags.Checked[3] then Result:= Result or $00000030;
 If Flags.Checked[4] then Result:= Result or $00000100;
 If Flags.Checked[5] then Result:= Result or $00000200;
 If Flags.Checked[6] then Result:= Result or $00000200;
 If Flags.Checked[7] then Result:= Result or $00000400;
 If Flags.Checked[8] then Result:= Result or $000000F0;
end;

procedure THIDDeviceReadRawInputForm.btnRegisterClick(
                                           Sender: TObject);
var  Rid : TRawInputDevice;
begin
 Rid.UsagePage:= UsagePage.Value;
 Rid.Usage    := Usage.Value;
 Rid.Flags    := GetFlagsValue;
 Rid.Target   := Self.Handle; // дескриптор самого окна
 // Регистрируем
 If RegisterRawInputDevices(@Rid, 1, sizeof(Rid))  then begin
  StatusBar.Panels[0].Text:= 'Ok';
 End else begin
  StatusBar.Panels[0].Text:= 'Error:'+SysErrorMessage(GetLastError);
 End;
end;
```

При получении сообщения необходимо вызвать функцию `GetRawInputData`. Для определения размера полученных данных обычно делается вызов этой функции с нулевыми параметрами. Второй вызов производится для получения собственно данных (листинг 11.9).

Листинг 11.9. Обработка сообщения `WM_INPUT`

```
procedure THIDDeviceReadRawInputForm.WndProc(var Message: TMessage);
var i, dwSize : DWord; Data : Array of Byte; P : PRawInputRecord; B :
Byte;
    S : String;  RidInfo : TRidDeviceInfo; pcbSize : DWord;
    HidHandle : THandle;
begin
 Inherited WndProc(Message);

 // Обрабатываем сообщение WM_INPUT
 If Message.Msg = WM_INPUT then begin
  XorIndic;

  // Сначала передаем нулевой буфер, чтобы узнать
  // необходимый размер буфера
  GetRawInputData(Message.lParam, RID_INPUT, nil, dwSize,
  sizeof(TRawInputHeader));

  // Создаем буфер
  SetLength(Data, dwSize);

  // Получаем данные
  If GetRawInputData(Message.lParam, RID_INPUT, Data, dwSize,
  sizeof(TRawInputHeader)) then begin
    P:= PRawInputRecord(@Data[0]);

    HidHandle:= P^.Header.hDevice;

    S:= '';
    Case TRawInputType(P^.Header.dwType) of
     // Данные мыши
     RIM_TYPEMOUSE: Begin
      S:= Format('X=%d Y=%d ulButtons=%d Wheel=%d',[
             P^.Mouse.lLastX,
             P^.Mouse.lLastY,
             P^.Mouse.ulButtons and ($FFFF-$0400),
```

```
          (P^.Mouse.ulButtons and $0400) shr 10
        ]);
    End;
    // Данные клавиатуры
    RIM_TYPEKEYBOARD: Begin
     S:= Format('MakeCode=%d VKey=%d Message=%d',[
           P^.Keyb.MakeCode,
           P^.Keyb.VKey,
           P^.Keyb.Message
        ]);
    End;
    // Данные обычного HID-устройства
    RIM_TYPEHID: Begin
     For i:= 0 to P^.Hid.dwSizeHid-1 do begin
      B:= Data[dwSize - P^.Hid.dwSizeHid + i];
      S:= S + Format(' %.2X ',[B]);
     End;
    End;
   End;
   DataMemo.Text:= S;
End else begin
 // Если ошибка...
 DataMemo.Text:= 'Error:'+SysErrorMessage(GetLastError);
 Exit;
End;

If not FInfo then begin
  With TLog.Create(HIDDeviceInfo) do begin
    ClearInfo;

    pcbSize:= SizeOf(RidInfo);
    FillChar(RidInfo, SizeOf(RidInfo), 0);
    RidInfo.cbSize:= pcbSize;
    If GetRawInputDeviceInfoA(HidHandle, RIDI_DEVICEINFO, @RidInfo,
    pcbSize) then begin
     Case TRawInputType(RidInfo.dwType) of
```

```
  RIM_TYPEMOUSE: Begin //   Mouse : TRidDeviceInfoMouse;
    With RidInfo.Mouse do begin
    AddInfoLineInt('RidInfo.Mouse.dwId'              , dwId);
    AddInfoLineInt('RidInfo.Mouse.dwNumberOfButtons ',
                                   dwNumberOfButtons );
    AddInfoLineInt('RidInfo.Mouse.dwSampleRate'      ,
                                   dwSampleRate);
    End;
  End;
  RIM_TYPEKEYBOARD: Begin //   Keyb  : TRidDeviceInfoKeyboard;
    With RidInfo.Keyb do begin
     AddInfoLineInt('RidInfo.Keyb.dwType', dwType);
     AddInfoLineInt('RidInfo.Keyb.dwSubType', dwSubType);
     AddInfoLineInt('RidInfo.Keyb.dwKeyboardMode',
                     dwKeyboardMode);
     AddInfoLineInt('RidInfo.Keyb.dwNumberOfFunctionKeys',
                     dwNumberOfFunctionKeys);
     AddInfoLineInt('RidInfo.Keyb.dwNumberOfIndicators'  ,
                     dwNumberOfIndicators);
     AddInfoLineInt('RidInfo.Keyb.dwNumberOfKeysTotal'   ,
                     dwNumberOfKeysTotal);
    End;
  End;
  RIM_TYPEHID: Begin //  Hid   : TRidDeviceInfoHid;
    With RidInfo.Hid do begin
     AddInfoLineInt('RidInfo.Hid.dwVendorId', dwVendorId);
     AddInfoLineInt('RidInfo.Hid.dwProductId', dwProductId);
     AddInfoLineInt('RidInfo.Hid.dwVersionNumber',
                             dwVersionNumber);
     AddInfoLineHex('RidInfo.Hid.usUsagePage', usUsagePage);
     AddInfoLineHex('RidInfo.Hid.usUsage'      , usUsage);
    End;
  End;
 End;
End else begin
 AddInfoLineErr('Ошибка RIDI_DEVICEINFO');
End;
```

```
      // Получить размер буфера, необходимый
      // для хранения имени устройства
      GetRawInputDeviceInfoA(HidHandle, RIDI_DEVICENAME, nil,
                                                     pcbSize);
      // Создаем буфер
      SetLength(Data, pcbSize);
      // Получить имя устройства
      If GetRawInputDeviceInfoA(HidHandle, RIDI_DEVICENAME, Data,
                                      pcbSize) then begin
       AddInfoLineStr('RidInfo.DeviceName', StrPas(@Data[0]));
      End;

     End; {with log}
     FInfo:= True;
   End; {if not info}

 End;
End;
```

Получаемые данные представляют собой струтуру `RawInputRecord`, состоящую из заголовка и собственно пакета данных:

```
 TRawInputHeader = record // заголовок
    dwType  : DWord;      // тип данных
    dwSize  : DWord;      // размер
    hDevice : THandle;    // дескриптор устройства (см. далее)
    wParam  : Longint;
 End;

 PRawInputRecord = ^TRawInputRecord;
 TRawInputRecord = record
   Header : TRawInputHeader;
   Case TRawInputType of
    RIM_TYPEMOUSE   :  (  Mouse : TRawInputMouse;  );
    RIM_TYPEKEYBOARD:  (  Keyb  : TRawInputKeys;   );
    RIM_TYPEHID     :  (  Hid   : TRawInputHid;    );
 End;
```

В зависимости от типа блок данных имеет разный формат. Для данных мыши это структура `TRawInputMouse`:

```
TRawInputMouse = record
   usFlags      : Word;
   ulButtons    : DWord;
   ulRawButtons : DWord;
   lLastX       : Longint;
   lLastY       : Longint;
   ulExtraInformation : DWord;
End;
```

Для клавиатуры данные содержатся в структуре типа `TRawInputKeys`:

```
TRawInputKeys = record
   MakeCode     : Word;
   Flags        : Word;
   Reserved     : Word;
   VKey         : Word;
   Message      : Word;
   ExtraInformation : DWord;
End;
```

Для всех других типов устройств данные передаются с помощью структуры `TRawInputHid`:

```
TRawInputHid = record
   dwSizeHid : DWord;     // размер в байтах каждого репорта
   dwCount   : DWord;     // число пакетов
   bRawData  : Pointer;
End;
```

Поле `hDevice` в заголовке передает дескриптор устройства. Передав его в функцию `GetRawInputDeviceInfoA`, мы получаем дополнительную информацию об устройстве. Эта функция имеет описание:

```
function GetRawInputDeviceInfoA(
    hDevice   : THandle;
    uiCommand : DWord;
    pData     : Pointer;
    var pcbSize : DWord) : LongBool; stdcall;
function GetRawInputDeviceInfoA;
external 'user32' name 'GetRawInputDeviceInfoA';
```

Константы для поля `uiCommand` описываются следующим образом:

```
RIDI_PREPARSEDDATA = $20000005;
RIDI_DEVICENAME    = $20000007;
RIDI_DEVICEINFO    = $2000000b;
```

Результат работы программы показан на рис. 11.3, а ее полный код можно найти на компакт-диске.

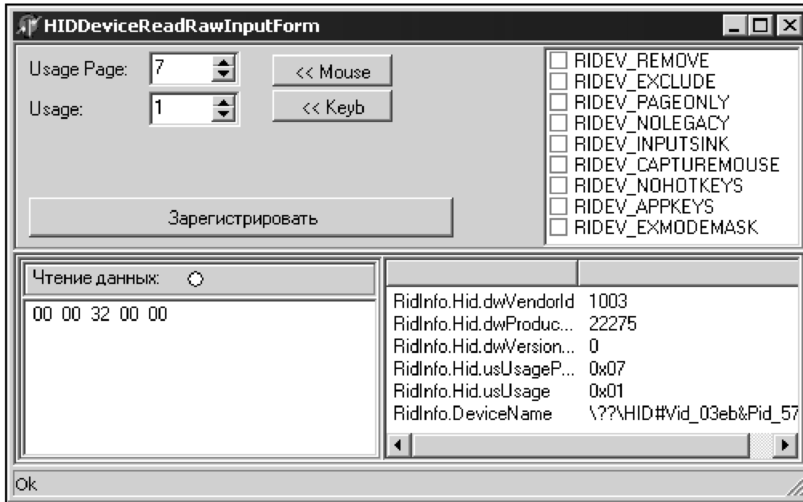


Рис. 11.3. Получение данных Raw Input

11.4. Функции DirectX

В этом разделе мы постараемся опровергнуть довольно распространенное мнение, что библиотека DirectX служит только для расширения для графических функций. Для нас интерес представляют функции класса `DirectInput`.

`DirectInput` — это набор API, обеспечивающий аппаратно-независимый ввод данных в систему в режиме реального времени. События, обрабатываемые этим API, формируются клавиатурой, мышью и джойстиком. Под джойстиком в `DirectInput` подразумеваются любые устройства, отличные от мыши и клавиатуры. Кроме того, `DirectInput` работает с HID-устройствами (в документации это оговаривается довольно невнятно). Примеры этого раздела мы будем реализовывать на языке C#.

Для использования функций `DirectInput` на платформе .NET требуется подключение библиотеки Microsoft.DirectX.DirectInput.dll, которое произ-

водится добавлением этой DLL в ссылки (reference) проекта. Сам DLL-файл можно найти в системном каталоге WINXP\assembly\GAC\Microsoft.DirectX.DirectInput.

Объекты `DirectInput` расположены в пространстве имен Microsoft.DirectX.DirectInput (мы приводим самые интересные классы):

- `Device` — класс устройства;
- `DeviceList` — коллекция устройств;
- `DeviceProperties` — описание свойств устройства;
- `Manager` — основной класс для доступа к устройствам;
- `SystenGuid` — содержит GUID-константы мыши и клавиатуры;
- `DeviceInstance` — структура описания устройства;
- `InputRange` — структура описания диапазона;
- `MouseState` — структура описания состояния мыши;
- `DeviceClass` — набор констант описания классов устройств;
- `DeviceType` — набор констант описания типов устройств;
- `Key` — набор констант описания клавиш клавиатуры;
- `Mouse` — набор констант описания клавиш мыши.

Класс `Manager.Device` позволяет получить доступ к устройствам. Например, получение количества доступных устройств выглядит следующим образом:

```
DeviceList dl = Microsoft.DirectX.DirectInput.Manager.Devices;
Console.WriteLine("Всего устройств: {0}", dl.Count);
```

Или можно выбрать только определенный класс устройств:

```
DeviceList dl = Manager.GetDevices(
         DeviceClass.GameControl, EnumDevicesFlags.AttachedOnly);
```

С объектом типа `DeviceList` можно работать с помощью нумератора. Каждый элемент этой коллекции является объектом типа `DeviceInstance`. Код для перечисления всех устройств и отображения некоторых свойств этих устройств выглядит следующим образом:

```
  foreach (DeviceInstance d in dl)
  {
    Console.WriteLine(
      "Type={0} SubType={1} Usage={2}
                      Page={3} Guid={4} ProductName={5}",
       d.DeviceType,
       d.DeviceSubType,
```

```
        (UInt16)d.Usage,
        (UInt16)d.UsagePage,
        d.InstanceGuid,
        d.ProductName
    );
  }
```

Для мыши этот код выведет данные: "`type=Mouse, subtype=1, usage=page=0`"; для клавиатуры: "`type=Keyboard, subtype=4, usage=page=0`"; для нашего HID-устройства (*см. гл. 10*): "`type=Device, subtype=0, usage=1, page=65280`".

Объект `DeviceInstance` пригоден для получения свойств устройства, а для обмена данными следует создать класс самого устройства, т. е. объект типа `Device`:

```
  try
  {
    dev.Acquire();
    dev.Poll();
    BufferedDataCollection dataCollection = dev.GetBufferedData();
  }
  catch (Exception e)
  {
     Console.WriteLine("Exception: " + e.Message);
  }
```

Также с помощью этого объекта можно получить идентификатор устройства:

```
  if (d.DeviceType == DeviceType.Device)
    // Для нашего HID-устройства напечатает "VID_03EB&PID_6203"
    Console.WriteLine(dev.Properties.TypeName);
```

С точки зрения `DirectInput` набор данных устройства состоит из элементов (объектов) в соответствии с дескриптором устройства. Для получения списка объектов необходимо вызвать метод `GetObjects`, передав ему необходимый фильтр:

```
// получить все объекты
DeviceObjectList objList =
               dev.GetObjects(DeviceObjectTypeFlags.All);
// получить только коллекции
DeviceObjectList objList =
               dev.GetObjects(DeviceObjectTypeFlags.Collection);
```

```
// получить только кнопки
DeviceObjectList objList =
               dev.GetObjects(DeviceObjectTypeFlags.Button);
```

Возможные фильтры описываются флагами набора `DeviceObjectTypeFlags`. Разумеется, класс `DeviceObjectList` имеет нумератор и метод `Count`:

```
foreach (DeviceObjectInstance oi in objList)
{
  Console.WriteLine(@"object:
    Name={0}
    Dim={1}
    Offset={2}
    Flags={3}
    CollectionNumber={4},
    Exponent={5}",
    oi.Name, oi.Dimension, oi.Offset,
    oi.Flags, oi.CollectionNumber, oi.Exponent
  );
}
```

Например, для мыши этот код отобразит следующие элементы:

```
objList.Count=8
object:
  Name=X-axis
  Dim=0
  Offset=0
  Flags=256
  CollectionNumber=0,
  Exponent=0
object:
  Name=Y-axis
  Dim=0
  Offset=4
  Flags=256
  CollectionNumber=0,
  Exponent=0
object:
  Name=Wheel
```

```
  Dim=0
  Offset=8
  Flags=256
  CollectionNumber=0,
  Exponent=0
object:
  Name=Button 0
  Dim=0
  Offset=12
  Flags=0
  CollectionNumber=0,
  Exponent=0
object:
  Name=Button 1
  Dim=0
  Offset=13
  Flags=0
  CollectionNumber=0,
  Exponent=0
object:
  Name=Button 2
  Dim=0
  Offset=14
  Flags=0
  CollectionNumber=0,
  Exponent=0
object:
  Name=Button 3
  Dim=0
  Offset=15
  Flags=0
  CollectionNumber=0,
  Exponent=0
object:
  Name=Button 4
  Dim=0
```

```
  Offset=16
  Flags=0
  CollectionNumber=0,
  Exponent=0
```

Перед чтением данных следует указать формат пакета, используя либо системные константы `DeviceDataFormat`, либо класс `DataFormat`:

```
// формат данных — протокол "мыши"
dev.SetDataFormat(DeviceDataFormat.Mouse);
```

При задании своего формата данных следует указать размер буфера:

```
dev.Properties.BufferSize = 4;
```

В листинге 11.10 приведен пример кода, реализующего чтение кодов клавиатуры. У нас не получилось выполнить чтение данных произвольного HID-устройства. С одной стороны в документации DirectX содержится пример формирования произвольного формата данных, но, с другой, библиотека позволяет работать только с объектами, под которыми подразумеваются единицы ввода информации: кнопки, клавиши, координаты и т. п.

Листинг 11.10. Чтение кодов клавиатуры

```
private Device device = null;
private BufferedDataCollection dataCollection = null;

public MainForm()
{
  InitializeComponent();

  // Создаем объект устройства
  device = new Device(SystemGuid.Keyboard);
  // Режим работы
  device.SetCooperativeLevel(this, CooperativeLevelFlags.Background|
  CooperativeLevelFlags.NonExclusive);
  // Размер буфера
  device.Properties.BufferSize = 11;
  // Начало обмена
  device.Acquire();
}

private void btnRead_Click(object sender, System.EventArgs e)
{
```

```
  string result = String.Empty;
  // получаем буфер данных
  dataCollection = device.GetBufferedData();

  // если есть данные - отображаем
  if(dataCollection != null)
  {
    foreach (BufferedData d in dataCollection)
    {
      // выводим смещение и код
      result += String.Format("0x{0:X2}=0x{1:X2}",
                 d.Offset, d.Data);
    }
  }
  tbResult.Text = result;
}
```

Метод `dev.SendHardwareCommand` позволяет передать данные устройству.

11.5. Диалог добавления нового оборудования

В Windows 2000/XP диалог добавления нового оборудования можно вызвать и программными средствами, как показано в листинге 11.11.

Листинг 11.11. Запуск диалога добавления нового оборудования (Windows 2000/XP)

```
type
 TcplApplet = function(
      hwndCPl: HWND;
      uMsg: DWORD; lParam1, lParam2: Longint
 ): Longint; stdcall;

// по нажатию кнопки
procedure Tform1.Button1Click(Sender: Tobject);
```

```
var
 APModule : Thandle;
 Applet   : TcplApplet;
begin
 // Загрузка CPL-библиотеки
 APModule:= LoadLibrary('hdwwiz.cpl');
 // Если ошибка загрузки - выход
 if APModule <= HINSTANCE_ERROR then Exit;
 // Точка входа
 Applet:= TcplApplet(GetProcAddress(APModule, 'CplApplet'));
 // Передать сообщение CPL_DBLCLK - «запустить по двойному щелчку»
 Applet(0, 5 {= CPL_DBLCLK}, 0, 0);
 // Освободить ссылку на библиотеку
 FreeLibrary(APModule);
end;
```

11.6. Работа с символьными именами устройств

Хотя прикладные программы не могут использовать внутренние имена, они могут получать, добавлять и удалять символьные имена для внутренних имен устройств. Эти операции выполняются с помощью функций `QueryDosDevice` и `DefileDosDevice` (*см. гл. 14*).

Листинг 11.12 демонстрирует получение списка, добавление и удаление символьных имен.

Листинг 11.12. 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.
```

Основное диалоговое окно нашей тестовой программы показано на рис. 11.4. Введем в верхнее окно строку `COM1` и нажмем кнопку **Получить NT-имя**. В окошке **NT-имя** должна отобразиться строка, похожая на `\Device\Serial0`.

Теперь введем в окошко **DOS-имя** какое-нибудь новое имя, например, `MyDevice` и нажмем кнопку **Добавить DOS-имя**. Результат мы можем увидеть, нажав кнопку **Получить полный список DOS-имен** или **Получить NT-имя**.

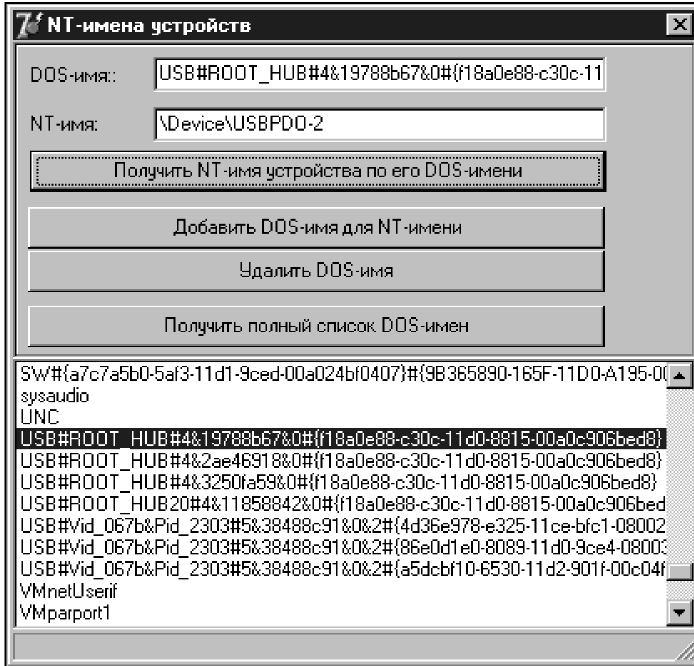


Рис. 11.4. Окно программы работы с именами устройств

В отличие от последовательных портов, имеющих всем привычные имена, USB-компоненты имеют имена, включающие GUID, например, для имени `USB#ROOT_HUB#4&19788b67&0#{f18a0e88-c30c-11d0-8815-00a0c906bed8}` с помощью нашей программы мы получим имя `\Device\USBPDO-2`.

Для просмотра имен удобно пользоваться программой WinObj (*см. разд. 3.6.1*).

11.7. Безопасное извлечение флэш-дисков

Очень часто встречается вопрос "Как программно извлечь флэш-диск?" Эта проблема может быть решена несколькими способами.

Первый способ, показанный в листинге 11.13, использует функции Setup API. Единственный недостаток этого метода — если устройство нельзя извлечь в данный момент времени, то пользователю выдается соответствующий диалог.

При необходимости извлечь устройство "по-тихому" можно использовать код, показанный в листинге 11.14. Здесь используются функции прямого обращения к драйверу (более подробно мы будем рассматривать эту функцию при реализации своих драйверов). Но этот способ имеет прямо противоположный недостаток — устройство извлекается, но операционная система не уведомляется об отключении диска. Соответственно, обращение к диску вызовет ошибку: "Отсутствует диск в устройстве".

Полный код обоих примеров можно найти на компакт-диске.

Листинг 11.13. Извлечение флэш-диска с помощью функций Setup API

```
using System;
using System.Runtime.InteropServices;

namespace DeviceEject
{
  class Class1
  {
    [STAThread]
    static unsafe void Main(string[] args)
    {
      Guid UsbGuid =
              new Guid("{36FC9E60-C465-11CF-8056-444553540000}");

      int PnPHandle = SetupAPI.SetupDiGetClassDevs(
        ref UsbGuid,
        null,
        null,
        SetupAPI.ClassDevsFlags.DIGCF_PRESENT
      );

      int  result    = -1;
      int DeviceIndex = 0;
```

```
  while (result != 0)
  {
    SetupAPI.SP_DEVINFO_DATA DeviceInfoData =
              new SetupAPI.SP_DEVINFO_DATA();
    DeviceInfoData.cbSize = Marshal.SizeOf(DeviceInfoData);
    result = SetupAPI.SetupDiEnumDeviceInfo(PnPHandle,
                          DeviceIndex, ref DeviceInfoData);

    if (result == 1)
    {
      if (IsRemovable(DeviceInfoData))
      {
        Console.WriteLine("{0}", GetRegistryProperty(PnPHandle,
                   ref DeviceInfoData,
                   SetupAPI.RegPropertyType.SPDRP_DEVICEDESC));
        if (SetupAPI.CM_Request_Device_Eject(
                          DeviceInfoData.DevInst,
                          null, null, 0, 0) == 0)
          Console.WriteLine("Устройство успешно отключено.");
      }
    }

    DeviceIndex++;
  }
  Marshal.FreeHGlobal((System.IntPtr)PnPHandle);
  Console.ReadLine();
}

public unsafe static string GetRegistryProperty(
             int PnPHandle,
             ref SetupAPI.SP_DEVINFO_DATA DeviceInfoData,
             SetupAPI.RegPropertyType Property)
{
  int RequiredSize = 0;
  SetupAPI.DATA_BUFFER Buffer = new SetupAPI.DATA_BUFFER();
```

```
      int result = SetupAPI.SetupDiGetDeviceRegistryProperty(
        PnPHandle,
        ref DeviceInfoData,
        Property,
        null,
        ref Buffer,
        1024,
        ref RequiredSize
        );

      return  Buffer.Buffer;

    }

    public unsafe static bool IsRemovable(
                       SetupAPI.SP_DEVINFO_DATA DevData)
    {
      int Status  = 0;
      int Problem  = 0;
      SetupAPI.CM_Get_DevNode_Status(ref Status, ref Problem,
                                     DevData.DevInst, 0);
      // public const int DN_REMOVABLE = 0x4000;
      return ((Status & SetupAPI.DN_REMOVABLE)!=0);
    }

  }
}
```

Листинг 11.14. Извлечение флэш-диска с помощью прямого обращения к драйверу

```
unit Unit1;

interface

uses
```

```
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    DiskName: TEdit;
    btnEject: TButton;
    Label1: TLabel;
    procedure btnEjectClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

const
 IOCTL_STORAGE_BASE      = $0000002d;
 FILE_READ_ACCESS        = $0001;
 FILE_ANY_ACCESS         = $0000;
 FILE_DEVICE_FILE_SYSTEM = $00000009;
 METHOD_BUFFERED         = 0;
 IOCTL_STORAGE_MEDIA_REMOVAL = IOCTL_STORAGE_BASE shl 16 or
                               FILE_READ_ACCESS shl 14 or
                               $0201 shl 2 or METHOD_BUFFERED;
 IOCTL_STORAGE_EJECT_MEDIA   = IOCTL_STORAGE_BASE shl 16 or
                               FILE_READ_ACCESS shl 14 or
                               $0202 shl 2 or METHOD_BUFFERED;
```

```
 FSCTL_LOCK_VOLUME     = FILE_DEVICE_FILE_SYSTEM shl 16 or
                         FILE_ANY_ACCESS shl 14 or
                         6 shl 2 or METHOD_BUFFERED;
 FSCTL_DISMOUNT_VOLUME = FILE_DEVICE_FILE_SYSTEM shl 16 or
                         FILE_ANY_ACCESS shl 14 or
                         8 shl 2 or METHOD_BUFFERED;

procedure TForm1.btnEjectClick(Sender: TObject);
var
 dwBytesReturned: DWord;
 hVolume: THandle;
 PMRBuffer: Bool;
 DevName : String;
begin
 // Получаем полное имя диска
 DevName:= Format('\\.\%s:', [DiskName.Text]);
 // Открываем устройство
 hVolume:= CreateFile(PChar(@DevName[1]), GENERIC_READ or GENERIC_WRITE,
              FILE_SHARE_READ or FILE_SHARE_WRITE, nil, OPEN_EXISTING,
              0, 0);
 PMRBuffer := False;
 If hVolume <> INVALID_HANDLE_VALUE then try
   // Выдаем команды
   If
     DeviceIoControl(hVolume, FSCTL_LOCK_VOLUME, nil, 0, nil, 0,
     dwBytesReturned, nil) and
     DeviceIoControl(hVolume, FSCTL_DISMOUNT_VOLUME, nil, 0, nil, 0,
     dwBytesReturned, nil) and
     DeviceIoControl(hVolume, IOCTL_STORAGE_MEDIA_REMOVAL, @PMRBuffer,
     sizeof(PMRBuffer), nil, 0, dwBytesReturned, nil) and
     DeviceIoControl(hVolume, IOCTL_STORAGE_EJECT_MEDIA, nil, 0, nil, 0,
     dwBytesReturned, nil)
   Then begin
      ShowMessage('Устройство успешно извлечено');
   End
   Else begin
     // Ошибка извлечения устройства
```

```
    ShowMessage(SysErrorMessage(GetLastError))
   End;
 Finally
   // Освободить дескриптор
   CloseHandle(hVolume)
 End
 Else begin
   // Ошибка открытия устройства
   ShowMessage(SysErrorMessage(GetLastError));
 End;
end;

end.
```

11.8. Обнаружение добавления и удаления устройств

При добавлении последовательных портов или флэш-дисков (вообще говоря, при изменениях в аппаратной конфигурации) Windows рассылает сообщение `WM_DEVICECHANGE`, передающее структуру `TWMDeviceChange`. Эта структура описывается так:

```
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 3—7), поэтому нам придется позаимствовать их из файла dbt.h набора заголовков Visual Studio. Полный файл dbt.pas можно найти на 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.

В Windows 2000 добавлена еще одна константа

```
DBT_DEVNODES_CHANGED = $0007
```

Это событие сообщает об изменении в списке узлов дерева устройств в Менеджере Устройств. Вообще говоря, изменение в списке самих устройств вызывает изменение узла дерева устройств, поэтому это сообщение рассылается достаточно часто. Именно с помощью него можно отследить добавление устройства даже в случае неуспешной его установки.

Пример обработки сообщения `WM_DEVICECHANGE` показан в листинге 11.15.

Листинг 11.15. 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;
    // получение имени диска
    function GetDiskName(unitmask : Longint) : string;
  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:'+WideCharToString(@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. Имя:
    '+GetDiskName(lpdbv.dbcv_unitmask));
   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:
    '+WideCharToString(@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. Имя:
    '+GetDiskName(lpdbv.dbcv_unitmask));
   End;
  End;
 End;

end;

// Маска имени диска. Возвращаемое значение состоит из битов,
// соответствующих именам дисков:
// бит 0=A, бит 1=B, бит 3=C и т.д.
function TForm1.GetDiskName(unitmask : Longint) : string;
var i : Integer;
begin
 For i:= 0 to 26 do begin
  if ((unitmask and 1) <> 0) then Break;
  unitmask:= unitmask shr 1;
 End;
 Result:= Char(Integer('A')+i);
end;

end.
```

Как видно из листинга, мы будем обрабатывать два события — добавление или удаление порта (последовательного или параллельного) и добавление или удаление логического диска. Работу первого сообщения можно проверить, установив флажок **Выключено** (Disable) для порта COM1 и затем

включив его снова (рис. 11.5). Работу второго сообщения можно проверить, подключив к USB-порту флэш-диск.

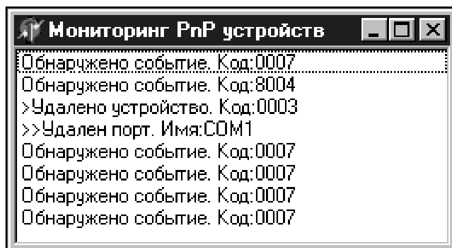


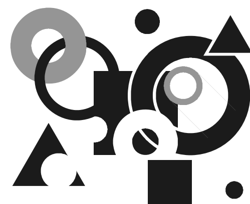
Рис. 11.5. Обнаружение изменений в аппаратной конфигурации

Обратите внимание на получение имени подключаемого или отключаемого диска. Буква диска кодируется соответствующим битом в маске `unitmask`. Первый бит обозначает диск A, второй — диск B и т. д.

11.9. Интернет-ресурсы

- Первые шаги. DirectInput.
  - **http://www.firststeps.ru/mfc/directx/drinput/drinput1.html**
- Beginning Game Development, Derek Pierson.
  - **http://msdn.microsoft.com/coding4fun/gamedevelopment/beginning4/default.aspx**

Глава 12



Разработка драйвера

Стандартные классы (такие как CDC, HID и т. п.) удобны, но имеют свои ограничения: по максимальной скорости обмена, размерам пакета, набору функций и т. д. Если возможностей стандартного драйвера не хватает, необходимо создавать свой. В этой главе мы рассмотрим процесс создания драйвера USB-устройства.

12.1. Основные процедуры драйвера WDM

Оговорим сразу, что мы рассматриваем именно драйверы модели WDM, а не драйверы Windows 98 или Windows NT. Это важно, так как драйверы WDM, с одной стороны должны содержать дополнительные процедуры для поддержки PnP, а с другой более логичны по структуре. Итак, в этом разделе мы перечислим основные процедуры драйвера WDM. Подробное описание процесса создания драйвера не входит в рамки этой книги, поэтому мы ограничимся исключительно практическим интересом, а всех желающих отсылаем к списку литературы [7, 8].

В общем случае драйвер должен реализовать следующий набор процедур:

- точка входа (имя этой процедуры совпадает с именем драйвера);
- процедура `DriverEntry`, вызывающаяся при загрузке драйвера в память;
- рабочие процедуры драйвера.

Точка входа может быть пустой процедурой, не выполняющей никаких действий. Основное требование — ее имя должно совпадать с именем драйвера. Остальные процедуры мы рассмотрим более подробно.

12.1.1. Процедура *DriverEntry*

Процедура `DriverEntry` вызывается при загрузке драйвера в память. Заголовок этой процедуры показан в листинге 12.1.

Листинг 12.1. Заголовок процедуры DriverEntry

```
NTSTATUS DriverEntry(
 IN PDRIVER_OBJECT  pDriverObject,// Адрес объекта драйвера
 IN PUNICODE_STRING pRegistryPath // Путь в реестре к подразделу драйвера
)
```

Получив от Диспетчера ввода/вывода указатель на структуру DRIVER_OBJECT, драйвер должен заполнить в ней следующие поля:

- поле pDriverObject→DriverStartIo — адрес процедуры StartIo, которая необходима для организации обработки очереди необработанных запросов;
- поле pDriverObject→DriverExtension→AddDevice — адрес процедуры AddDevice, которая будет вызываться при инициализации нового устройства;
- поля в массиве pDriverObject→MajorFunction[IRP_MJ_xxx] — драйвер должен регистрировать точки входа в собственные рабочие процедуры (*см. ниже*);
- поле pDriverObject→DriverUnload — адрес процедуры Unload, которая будет вызываться перед выгрузкой драйвера.

Пример регистрации рабочих процедур показан в листинге 12.2, а набор констант IRP_MJ_xxx из файла ntddk.h — в листинге 12.3.

Листинг 12.2. Пример регистрации рабочих процедур драйвера[1]

```
// пример из драйвера GiveIOEx
DriverObject->MajorFunction[IRP_MJ_CREATE] = GiveioCreateDispatch;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = GiveioDeviceControl;
```

Листинг 12.3. Описание констант IRP_MJ_xxx (ntddk.h)

```
#define IRP_MJ_CREATE                   0x00
#define IRP_MJ_CREATE_NAMED_PIPE        0x01
#define IRP_MJ_CLOSE                    0x02
#define IRP_MJ_READ                     0x03
#define IRP_MJ_WRITE                    0x04
#define IRP_MJ_QUERY_INFORMATION        0x05
```

[1] Здесь и далее мы будем пользоваться примером драйвера GiveIoEx [3].

```
#define IRP_MJ_SET_INFORMATION          0x06
#define IRP_MJ_QUERY_EA                 0x07
#define IRP_MJ_SET_EA                   0x08
#define IRP_MJ_FLUSH_BUFFERS            0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION   0x0b
#define IRP_MJ_DIRECTORY_CONTROL        0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL      0x0d
#define IRP_MJ_DEVICE_CONTROL           0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL  0x0f
#define IRP_MJ_SHUTDOWN                 0x10
#define IRP_MJ_LOCK_CONTROL             0x11
#define IRP_MJ_CLEANUP                  0x12
#define IRP_MJ_CREATE_MAILSLOT          0x13
#define IRP_MJ_QUERY_SECURITY           0x14
#define IRP_MJ_SET_SECURITY             0x15
#define IRP_MJ_POWER                    0x16
#define IRP_MJ_SYSTEM_CONTROL           0x17
#define IRP_MJ_DEVICE_CHANGE            0x18
#define IRP_MJ_QUERY_QUOTA              0x19
#define IRP_MJ_SET_QUOTA                0x1a
#define IRP_MJ_PNP                      0x1b
#define IRP_MJ_PNP_POWER                IRP_MJ_PNP
#define IRP_MJ_MAXIMUM_FUNCTION         0x1b
```

Если драйверу интересен только один тип запросов, то можно воспользоваться константой `IRP_MJ_MAXIMUM_FUNCTION`, как показано в листинге 12.4. Все запросы будут обрабатываться процедурой `IrpHandler`, за исключением запросов `IRP_MJ_DEVICE_CONTROL`, которые будут поступать в процедуру `IrpDeviceControl`.

Листинг 12.4. Обработка одного типа запроса

```
int i;
for (i=0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
 DriverObject->MajorFunction[i] = IrpHandler;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IrpDeviceControl;
```

12.1.2. Процедура *AddDevice*

Основной обязанностью процедуры `AddDevice` (точнее говоря, той процедуры, которая была зарегистрирована в поле `DriverExtension→AddDevice` при вызове `DriverEntry`) является создание объекта устройства с использованием вызова `IoCreateDevice`.

Заголовок процедуры `AddDevice` показан в листинге 12.5, а скелетный пример организации самой рабочей процедуры — в листинге 12.6.

При необходимости подключения к объекту физического устройства (этот объект создается шинным драйвером), драйвер может вызвать функцию `IoAttachDevice`, передав ей указатель, взятый из параметра `PhysicalDeviceObject`.

Листинг 12.5. Заголовок процедуры `AddDevice`

```
NTSTATUS XxxAddDevice(
 IN PDRIVER_OBJECT DriverObject,
 IN PDEVICE_OBJECT PhysicalDeviceObject
);
```

Листинг 12.6. Пример процедуры `AddDevice`

```
NTSTATUS FilterAddDevice(
 IN PDRIVER_OBJECT DriverObject,
 IN PDEVICE_OBJECT PhysicalDeviceObject
)
{
 NTSTATUS                status = STATUS_SUCCESS;
 PDEVICE_OBJECT          deviceObject = NULL;
 PDEVICE_EXTENSION       deviceExtension;
 ULONG                   deviceType = FILE_DEVICE_UNKNOWN;

 // Вызов IoIsWdmVersionAvailable(1, 0x20) возвращает TRUE
 // в операционных системах после Windows 2000.
 if (!IoIsWdmVersionAvailable(1, 0x20)) {
  deviceObject = IoGetAttachedDeviceReference(PhysicalDeviceObject);
  deviceType = deviceObject->DeviceType;
  ObDereferenceObject(deviceObject);
 }
```

```
// Создание объекта драйвера-фильтра
status = IoCreateDevice (DriverObject,
 sizeof (DEVICE_EXTENSION),
 NULL,  // без имени
 deviceType,
 FILE_DEVICE_SECURE_OPEN,
 FALSE,
 &deviceObject
);

if (!NT_SUCCESS (status)) {
 // выход, если ошибка
 return status;
}

// отладочная информация
DebugPrint (("AddDevice PDO (0x%x) FDO (0x%x)\n",
  PhysicalDeviceObject, deviceObject));

deviceExtension = (PDEVICE_EXTENSION) deviceObject->DeviceExtension;
deviceExtension->NextLowerDriver =
  IoAttachDeviceToDeviceStack(deviceObject, PhysicalDeviceObject);

// Ошибка означает сбой в системе Plug and Play
if(NULL == deviceExtension->NextLowerDriver) {
 IoDeleteDevice(deviceObject);
 return STATUS_UNSUCCESSFUL;
}

deviceObject->Flags |= deviceExtension->NextLowerDriver->Flags &
                      (DO_BUFFERED_IO | DO_DIRECT_IO |
                      DO_POWER_PAGABLE
);

deviceObject->DeviceType =
             deviceExtension->NextLowerDriver->DeviceType;
```

```
 deviceObject->Characteristics =
             deviceExtension->NextLowerDriver->Characteristics;
 deviceExtension->Self = deviceObject;

 // Установка начального состояния фильтра
 INITIALIZE_PNP_STATE(deviceExtension);

 DebugPrint(("AddDevice: %x to %x->%x \n", deviceObject,
                       deviceExtension->NextLowerDriver,
                       PhysicalDeviceObject));

 deviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
 return STATUS_SUCCESS;
}
```

12.1.3. Процедура *Unload*

Обычно загруженный драйвер остается в системе до перезагрузки. Для того чтобы сделать драйвер выгружаемым, необходимо реализовать и зарегистрировать процедуру выгрузки `Unload`. Диспетчер ввода/вывода произведет вызов этой процедуры в момент ручной либо автоматической выгрузки драйвера.

Заголовок этой процедуры показан в листинге 12.7.

Листинг 12.7. Заголовок процедуры `Unload`

```
VOID GiveioUnload(
 IN PDRIVER_OBJECT pDriverObject // указатель на объект драйвера
)
```

Процедура `Unload` выполняет стандартный набор действий:

1. При необходимости драйвер может сохранять текущие настройки в системном реестре. При последующей загрузке драйвера эти данные могут быть использованы в процедуре `DriverEntry`.
2. Если разрешены прерывания для обслуживаемого устройства, то процедура `Unload` должна произвести их запрещение и отключение от объекта прерываний.

3. Символьная ссылка должна быть удалена из пространства имен, видимого пользовательскими приложениями. Это выполняется при помощи вызова функции `IoDeleteSymbolicLink`.
4. Объект драйвера должен быть удален вызовом функции `IoDeleteDevice`.
5. В случае, если драйвер управляет многокомпонентным контроллером, необходимо повторить шаги 3 и 4 для каждого устройства, подключенного к контроллеру, а затем удалить сам объект контроллера при помощи вызова функции `IoDeleteController`.
6. Следует выполнить освобождение памяти, выделенной драйверу, во всех типах оперативной памяти.

Драйверы модели WDM выполняют почти все из этих действий в обработчике `IRP_MJ_PNP` запросов с субкодом `IRP_MN_REMOVE`.

Листинг 12.8 показывает пример написания процедуры `Unload`. Обратите внимание на преобразование имени драйвера в кодировку Unicode.

Листинг 12.8. Пример процедуры `Unload`

```
// пример из драйвера GiveIOEx
#define DEVICE_NAME_STRING L"giveioex"

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);
}
```

Важно отметить, что процедура `Unload` не вызывается в момент перезагрузки или выключения системы. При необходимости выполнения действий во время выключения следует делать это в обработчике запросов `IRP_MJ_SHUTDOWN`, причем объект устройства должен быть, с помощью вызова функции `IoRegisterShutdownNotification`, занесен в очередь объектов, получающих уведомление о выключении.

12.1.4. Рабочие процедуры драйвера

Клиенты драйвера (т. е. пользовательские приложения или модули ядра) общаются с драйвером с помощью специальных структур данных, называемых *пакетами IRP* (Input/output Request Packet, пакет запроса ввода/вывода). При появлении запроса от приложения пользователя Диспетчер ввода/вывода вызывает соответствующий обработчик драйвера, который был зарегистрирован в массиве `DriverObject→MajorFunction[]`, как показано в листинге 12.2.

Пакеты IRP являются структурами данных переменной длины и состоят из стандартного заголовка, содержащего общую учетную информацию, и одного или нескольких блоков параметров, называемых *ячейкой стека ввода/вывода* (I/O stack location). Структура пакета IRP показана на рис. 12.1.

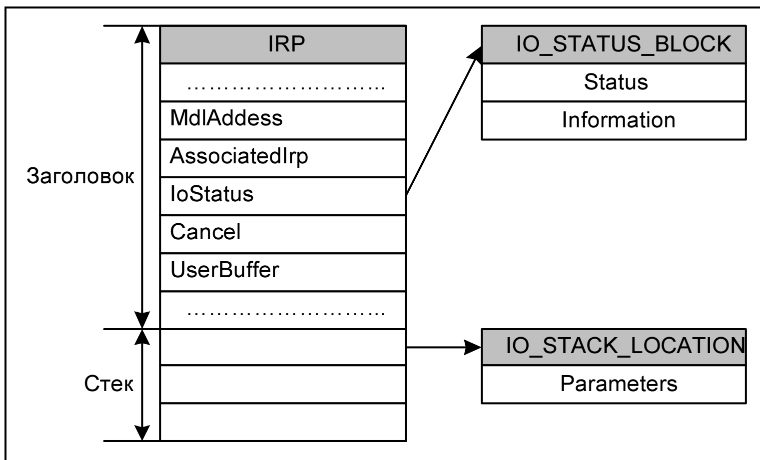


Рис. 12.1. Структура пакета IRP

12.1.4.1. Заголовок пакета

Описывать все поля заголовка пакета IRP не имеет смысла. Код драйвера может работать со следующими полями:

- `PMDL MdlAddess` — указатель на MDL-список (Memory Descriptor List, список дескрипторов памяти) в случае, если устройство поддерживает прямой ввод/вывод;
- `PVOID AssociatedIrp.SystemBuffer` — указатель на системный буфер для случая, если устройство поддерживает буферизированный ввод/вывод;
- `IO_STATUS_BLOCK IoStatus` — код состояния (статус) запроса;

- BOOLEAN Cancel — индикатор того, что пакет IRP должен быть аннулирован;
- PVOID UserBuffer — адрес пользовательского буфера для ввода/вывода.

Поле структуры IoStatus фиксирует состояние данной операции ввода/вывода: когда драйвер готов завершить обработку пакета IRP, он устанавливает поле IoStatus.Status в значение STATUS_xxx. В поле IoStatus.Information записывается 0, если произошла ошибка или другое, определенное операцией ввода/вывода значение, чаще всего — число переданных или полученных байт данных (которое может быть равно и нулю).

12.1.4.2. Ячейки стека ввода/вывода

Основное назначение ячеек ввода/вывода состоит в хранении параметров запроса на ввод/вывод. Диспетчер ввода/вывода создает пакет IRP с числом ячеек в стеке, равном числу драйверных слоев, участвующих в обработке запроса. Любому драйверу в иерархии разрешен доступ к его собственной ячейке стека. Когда драйвер передает IRP-пакет нижнему драйверному уровню, он автоматически перемещает указатель стека ввода/вывода пакета таким образом, что он указывает на стековую ячейку для этого драйвера. Когда обработка пакета драйвером нижнего уровня завершена, указатель стека снова возвращается в исходное положение и указывает на ячейку стека для лежащего выше драйвера. Для получения указателя на текущую ячейку существует специальный системный вызов IoGetCurrentStackLocation.

Получив указатель на свою ячейку стека, т. е. указатель на структуру IO_STACK_LOCATION, драйвер может работать со следующими полями:

- UCHAR MajorFunction — код IRP_MJ_xxx, описывающий назначение операции;
- UCHAR MinorFunction — субкод операции;
- PDEVICE_OBJECT DeviceObject — указатель на объект устройства, которому был адресован данный запрос IRP;
- PFILE_OBJECT FileObject — файловый объект для данного запроса, если он задан.

В зависимости от значения MajorFunction поле Parameters представляется по-разному (т. е. оно описано как объединение (union)), например:

- для типа IRP_MJ_CONTROL в поле Parameters доступны следующие поля:
  - ULONG OutputBufferLength;
  - ULONG InputBufferLength;
  - ULONG IoControlCode;
  - PVOID Type3InputBuffer;

- для типов `IRP_MJ_READ` и `IRP_MJ_WRITE` в поле `Parameters` доступны следующие поля:
  - `ULONG Length`;
  - `ULONG Key`;
  - `LARGE_INTEGER ByteOffset`.

12.4.1.3. Рабочие процедуры драйвера

Тема книги не позволяет вдаваться в подробности *рабочих процедур драйвера* (dispatch routines). С практической точки зрения достаточно следующей информации:

- список поддерживаемых драйвером рабочих процедур (т. е. набор обрабатываемых кодов `IRP_MJ_xxx`) формируется драйвером в процессе выполнения процедуры `DriverEntry`;
- перед вызовом `DriverEntry` Диспетчер ввода/вывода заполняет весь массив адресом процедуры `_IopInvalidDeviceRequest`, обеспечивая, таким образом, корректную обработку неподдерживаемых рабочих процедур;
- все процедуры драйвера используют один формат параметров и тип вызова, применяя прототип, показанный в листинге 12.9.

Листинг 12.9. Прототип процедуры драйвера

```
NTSTATUS
 DispatchProcedurePrototype(
  // указатель на объект устройства, для которого
  // предназначен пакет IPR
  IN PDEVICE_OBJECT DeviceObject,
  // указатель на пакет IPR
  IN PIRP pIrp
 )
{
 // Возращаемое значение
 NTSTATUS            ntStatus = STATUS_SUCCESS;
 // Указатель на текущую ячейку стека
 PIO_STACK_LOCATION  irpSp;
 // Длина входного буфера
 ULONG               inBufLength;
```

```
 // Длина выходного буфера
 ULONG              outBufLength;
 // Указатель на входной буфер
 PULONG             LongBuffer;

 // Получить указатель на ячейку стека
 irpSp       = IoGetCurrentIrpStackLocation( pIrp );
 // Длина входного буфера
 inBufLength = irpSp->Parameters.DeviceIoControl.InputBufferLength;
 // Указатель на входной буфер для метода доступа METHOD_BUFFERED
 LongBuffer  = (PULONG) pIrp->AssociatedIrp.SystemBuffer;
 ... ... ... ... ... ... ... ... ... ... ... ... ... ...
 return ntStatus;
}
```

Рабочая процедура драйвера возвращает результат типа `NTSTATUS`:

- `STATUS_SUCCESS` — запрос обработан;
- `STATUS_PENDING` — ожидается обработка запроса;
- `STATUS_xxx` — код ошибки.

Существуют три варианта завершения рабочей процедуры.

- Отклонение запроса. Производится в случае, когда рабочая процедура не может обработать запрос, например, вследствие неустранимой ошибки. В этом случае следует выполнить следующие действия (листинг 12.10):
  - в поле `IoStatus.Status` записывается код ошибки;
  - поле `IoStatus.Information` обнуляется;
  - производится вызов `IoCompleteRequest` для завершения обработки запроса;
  - рабочая процедура возвращает тот же код ошибки, который был записан в поле `IoStatus.Status`.
- Завершение работы с запросом. Многие запросы могут быть полностью обработаны без обращения к физическому устройству, за которое отвечает драйвер, например, получение дескриптора устройства или конфигурирование самого драйвера. В случае необходимости завершить обработку запроса следует выполнить следующие шаги (листинг 12.11):
  - записать в поле `IoStatus.Status` код успешного завершения `STATUS_SUCCESS`;

- записать в поле IoStatus.Information корректное значение (зависит от запроса);
- выполнить вызов IoCompleteRequest для завершения обработки запроса;
- возвратить код успешного завершения STATUS_SUCCESS.

❒ Передача запроса на обработку. В простейшем случае драйвер может сразу завершить обработку запроса, например, в ответ на запрос чтения данных сразу же считать данные с последовательного порта. Однако асинхронный механизм ввода/вывода Windows предусматривает и другой метод работы. Рабочая процедура может поместить пакет IRP в очередь для последующей обработки и сразу же вернуть сообщение о том, что обработка пакета не была завершена. Для этого следует выполнить следующие шаги (листинг 12.12):
- выполнить вызов IoMarkIrpRending, информируя Диспетчер ввода/вывода о том, что пакет поставлен в очередь на обработку;
- выполнить вызов IoStartPacket, чтобы поместить пакет в системную очередь для последующей обработки процедурой StartIO;
- возвратить код незавершенной обработки пакета STATUS_PENDING.

Листинг 12.10. Отклонение запроса рабочей процедурой

```
NTSTATUS
 DispatchProcedurePrototype(
  IN PDEVICE_OBJECT DeviceObject,
  IN PIRP pIrp
 )
{
 // Возращаемое значение
 NTSTATUS ntStatus = STATUS_SUCCESS;

 ... попытка обработки запроса ...

 // Обнаружили ошибку - отклоняем запрос
 // код ошибки — буфер слишком мал
 ntStatus = STATUS_BUFFER_TOO_SMALL;
 // ни одного байта не передано
 pIrp->IoStatus.Information = 0;
 // статус обработки
```

```
 pIrp->IoStatus.Status = ntStatus;
 // завершение обработки
 IoCompleteRequest(pIrp, IO_NO_INCREMENT);
 // код завершения
 return ntStatus;
}
```

Листинг 12.11. Завершение обработки запроса рабочей процедурой

```
NTSTATUS
 DispatchProcedurePrototype(
  IN PDEVICE_OBJECT DeviceObject,
  IN PIRP pIrp
 )
{
 // Возращаемое значение
 NTSTATUS ntStatus = STATUS_SUCCESS;

 ... обработка запроса ...

 // запрос успешно обработан
 ntStatus = STATUS_SUCCESS;
 // ни одного байта не передано
 pIrp->IoStatus.Information = 0;
 // статус обработки
 pIrp->IoStatus.Status = ntStatus;
 // завершение обработки
 IoCompleteRequest(pIrp, IO_NO_INCREMENT);
 // код завершения
 return ntStatus;
}
```

Листинг 12.12. Постановка запроса в очередь для обработки

```
NTSTATUS
 DispatchProcedurePrototype(
  IN PDEVICE_OBJECT DeviceObject,
```

```
  IN PIRP pIrp
 )
{
 ... предварительная обработка запроса ...

 // постановка запроса в очередь на обработку
 IoMarkIrpPending(pIrp);
 IoStartPacket(pDeviceObject, pIrp, 0, NULL);
 // код завершения
 return STATUS_PENDING;
}
```

Следует отметить два важных обстоятельства:

- вызов `IoCompleteRequest(pIrp,…)` может освободить память, занимаемую собственно пакетом, поэтому оператор `return(pIrp->IoStatus.Status);` может привести к непредсказуемым результатам;
- после возврата из рабочей процедуры Диспетчер ввода/вывода завершает все запросы, не помеченные статусом `STATUS_PENDING`, однако не уведомляет об этом вышестоящие в очереди драйверы. Для корректного уведомления драйвер должен вызывать `IoCompleteRequest` в конце обработки запроса.

Набор кодов запросов `IRP_MJ_xxx` и соответствующие им функции Windows API пользовательского режима приведен в табл. 12.1. Единственным обязательным для обработки кодом является код `IRP_MJ_CREATE`, генерируемый при вызове `CreateFile`. При необходимости освобождения ресурсов при вызове `CloseHandle` драйвер должен обрабатывать код `IRP_MJ_CLOSE`. Необходимость обработки остальных кодов зависит от функциональности драйвера.

Таблица 12.1. *Коды запросов IRP и соответствующие функции пользовательского режима*

IRP-код	**Вызов Windows API[2] или действия**
`IRP_MJ_CREATE`	`CreateFile`
`IRP_MJ_CLEANUP`	Очистка ожидающих обработки пакетов IRP при закрытии дескриптора драйвера при обработке вызова `CloseHandle`

[2] Описание и параметры функций Windows API можно найти в справочной части книги.

Таблица 12.1 *(окончание)*

IRP-код	Вызов Windows API или действия
IRP_MJ_CLOSE	CloseHandle
IRP_MJ_READ	ReadFile
IRP_MJ_WRITE	WriteFile
IRP_MJ_DEVICE_CONTROL	DeviceIoControl
IRP_MJ_INTERNAL_ DEVICE_CONTROL	Действия по управлению устройством, доступные только для клиентов, работающих в режиме ядра (недоступно для вызовов пользовательского режима)
IRP_MJ_QUERY_ INFORMATION	Передача длины файла в ответ на вызов GetFileSize
IRP_MJ_SET_ INFORMATION	Установка длины файла по вызову SetFileSize
IRP_MJ_FLUSH_BUFFERS	Запись или очистка служебных буферов при отработке вызовов, например: • FlushFileBuffres • FlushConsoleInputBuffer • PurgeComm
IRP_MJ_SHUTDOWN	Действия, которые нужно выполнить драйверу в процессе подготовки системы к завершению работы
IRP_MJ_PNP	Посылается системой PnP во время нумерации устройств, распределения ресурсов и т. д.
IRP_MJ_DEVICE_CHANGE	Посылается при изменении состава оборудования

12.1.5. Обслуживание запросов IOCTL

Как показано в табл. 12.1, набор функций, соответствующих запросам IRP_MJ_xxx, ограничивается набором самих констант. Дополнительные запросы к драйверу формируются с помощью рабочей процедуры драйвера для кода IRP_MJ_DEVICE_CONTROL и, соответственно, функции DeviceIoControl.

Заголовок функции DeviceIoControl показан в листинге 12.13, а более подробное описание дано в справочной части книги.

Листинг 12.13. Заголовок функции `DeviceIoControl`

```
BOOL DeviceIoControl(
 HANDLE hDevice,              // дескриптор драйвера
 DWORD dwIoControlCode,       // код операции
 LPVOID lpInBuffer,           // входной буфер
 DWORD nInBufferSize,         // размер входного буфера
 LPVOID lpOutBuffer,          // выходной буфер
 DWORD nOutBufferSize,        // размер выходного буфера
 LPDWORD lpBytesReturned,     // число переданных байт
 LPOVERLAPPED lpOverlapped    // асинхронная информация
);
```

Дескриптор драйвера получается при вызове функции `CreateFile`, или, другими словами, при открытии драйвера. Входной и выходной буферы позволяют обмениваться данными с драйвером.

Наибольший интерес представляет параметр `dwIoControlCode`, передающий в драйвер код выполняемой операции, называемый IOCTL (Input/Output ConTroL code, код операции ввода/вывода). Коды IOCTL, передаваемые в драйвер, могут быть определены разработчиком драйвера и имеют строго определенный формат (рис. 12.2).

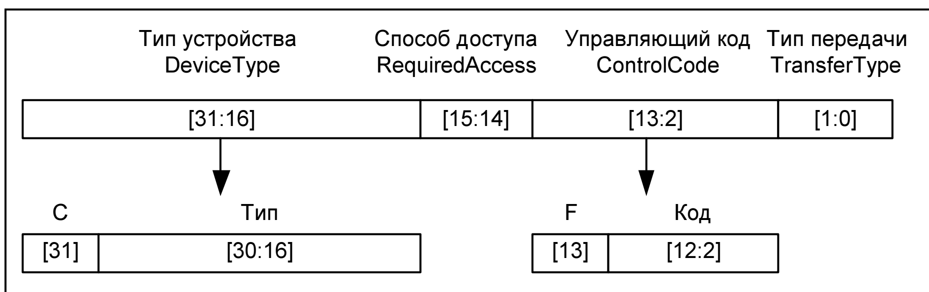


Рис. 12.2. Формат кода IOCTL

Для формирования кода IOCTL в Windows DDK существует специальное макроопределение `CTL_CODE`, параметры и пример использования которого показаны в листингах 12.14 и 12.15. Описание параметров этого макроса дано в табл. 12.2. Для пользовательских программ на языке Delphi можно использовать функцию, код которой показан в листинге 12.16.

Листинг 12.14. Макроопределение `CTL_CODE`

```
#define CTL_CODE( DeviceType, Function, Method, Access ) (\
    ((DeviceType) << 16) | ((Access) << 14) |
    ((Function) << 2) | (Method)\
)
```

Листинг 12.15. Формирование кода IOCTL в драйвере

```
// 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)

#define IOCTL_DISK_SET_PARTITION_INFO\
   CTL_CODE(IOCTL_DISK_BASE, 0x008, METHOD_BUFFERED,\
            FILE_READ_DATA | FILE_WRITE_DATA)
```

Листинг 12.16. Формирование кода IOCTL в Delphi

```
Const
 GIVEIO_TYPE     = 40000;
 METHOD_BUFFERED =     0;
 FILE_ANY_ACCESS = $0000;

// Функция формирования кода IOCTL
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;
```

```
// Пример формирования кода для листинга 12.14
Var
 IOCTL_IOPM_GET_ALL_ACCESS: Cardinal;
... ... ... ...
IOCTL_IOPM_GET_ALL_ACCESS:= Get_Ctl_Code($900);
```

Таблица 12.2. *Параметры макроопределения* `CTL_CODE`

Параметр	**Описание**
`DeviceType`	Код драйвера: • 0x0000–0x7FFFF — зарезервированы Microsoft • 0x8000–0xFFFF — определяются пользователем
`ControlCode`	Определяемые драйвером коды IOCTL: • 0x000–0x7FF — зарезервированы Microsoft • 0x800–0xFFF — определяются пользователем
`TransferType`	Способ получения доступа к буферу: • 0: `METHOD_BUFFERED` • 1: `METHOD_IN_DIRECT` • 2: `METHOD_OUT_DIRECT` • 3: `METHOD_NEITHER`
`RequiredAccess`	Тип доступа: • 0x0000: `FILE_ANY_ACCESS` • 0x0001: `FILE_READ_ACCESS` • 0x0002: `FILE_WRITE_ACCESS` • 0x0003: `FILE_READ_ACCESS | FILE_WRITE_ACCESS`

Заметим, что из табл. 12.2 видно, что флаги `C` и `F`, показанные на рис. 12.2, будут равны `0`, если код IOCTL является зарезервированным кодом Microsoft.

Более подробное обсуждение значения констант, составляющих код IOCTL, выходит за рамки нашей книги, их можно найти в [8, 9] или в MSDN. Нас же, прежде всего, интересует способ обработки пользовательских кодов. Листинг 12.17 показывает пример обработки кодов IOCTL в рабочей процедуре драйвера.

Листинг 12.17. Обработка кода IOCTL в рабочей процедуре драйвера

```
// в процедуре DriverEntry регистритуем рабочую процедуру драйвера
// для обработки IOCTL кодов
NTSTATUS DriverEntry(
 IN PDRIVER_OBJECT DriverObject,
 IN PUNICODE_STRING RegistryPath
)
{
... ... ... ... ... ...
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IOCTL_DeviceControl;
... ... ... ... ... ...
 return STATUS_SUCCESS;
}

// Рабочая процедура драйвера
NTSTATUS
IOCTL_DeviceControl(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP pIrp
    )
{
 // результат
 NTSTATUS ntStatus = STATUS_SUCCESS;
 // получить указатель на стек
 PIO_STACK_LOCATION irpSp = IoGetCurrentIrpStackLocation(pIrp);
 // получить код IOCTL
 ULONG ioctlCode = irpSp->Parameters.DeviceIoControl.IoControlCode;
 // размер входного буфера
 ULONG inSize = irpSp->Parameters.DeviceIoControl.InputBufferLength;
 // размер выходного буфера
 ULONG outSize = irpSp->Parameters.DeviceIoControl.OutputBufferLength;

 // обработка кодов IOCTL
 switch (ioctlCode)
 {
```

```
  // коды IOCTL_xxx формируются разработчиком драйвера
  case IOCTL_CODE1:
  {
   // проверяем входные параметры
   if ((inSize == 0) || (outsize == 0))
   {
    ntStatus = STATUS_INVALID_PARAMETER;
    break;
   }
   ... ... обработка IOCTL_CODE1 ... ...
   break;
  }

  case IOCTL_CODE2:
  {
   // проверяем входные параметры
   if (inSize < 100)
   {
    ntStatus = STATUS_BUFFER_TOO_SMALL;
    break;
   }
   ... ... обработка IOCTL_CODE2 ... ...
   break;
  }
 }
 pIrp->IoStatus.Status = ntStatus;
 IoCompleteRequest( pIrp, IO_NO_INCREMENT );
 return ntStatus;
}
```

Как видно из заголовка функции `DeviceIoControl`, драйверу передаются входной и выходной буферы с данными. Драйвер получает указатели на эти данные в полях пакета IRP, зависящих от метода доступа, "зашифрованного" в коде номера IOCTL (см. табл. 12.2).

- При использовании метода `METHOD_BUFFERED` Диспетчер ввода/вывода предоставляет единственный буфер в нестраничной памяти, достаточный для размещения входного и выходного буферов инициатора вызова. Ад-

рес этой области размещается в поле `AssociatedIrp.SystemBuffer` (листинг 12.9). Затем производится копирование входного буфера с данными инициатора запроса в эту область. В поле `UserBuffer` заносится оригинальный адрес буфера для получения данных инициатора запроса. По завершении обработки запроса Диспетчер ввода/вывода копирует содержимое выделенной области данных в выходной буфер инициатора запроса. Таким образом, драйверу предоставляется один буфер, даже если инициатор запроса указал два разных буфера.

- При использовании метода `METHOD_IN_DIRECT` и `METHOD_OUT_DIRECT` Диспетчер ввода/вывода производит *фиксацию* (lock) выходного буфера инициатора запроса в физической памяти. Затем он производит построение списка дескрипторов памяти для выходного буфера и сохраняет указатель на этот список в поле `MdlAddress` пакета IRP. Кроме того, Диспетчер ввода/вывода выделяет временную область в нестраничном пуле и сохраняет этот адрес в поле `AssociatedIrp.SystemBuffer` пакета IRP. Производится копирование содержимого входного буфера инициатора запроса в выделенный системный буфер, а в поле `UserBuffer` производится запись значения `NULL`. После этого пакет IRP поступает в вызываемую рабочую процедуру драйвера.
- При использовании метода `METHOD_NEITHER` Диспетчер ввода/вывода помещает адрес входного буфера инициатора запроса в поле `Parameters.DeviceIoControl.Type3InputBuffer` в текущей ячейке стека пакета IRP текущей операции ввода/вывода. В поле `UserBuffer` производится запись адреса выходного буфера инициатора запроса, где инициатор ожидает получить результаты выполнения операции. Оба этих адреса указывают в область памяти инициатора запроса.

12.2. Загрузка драйвера и обращение к процедурам драйвера

Теперь, когда мы выяснили, что основная деятельность драйвера производится в рабочих процедурах, разберемся с вопросом как получить доступ к этим процедурам (при описании мы несколько упростим картину, позволив себе подойти к этому вопросу с практической точки зрения).

12.2.1. Процедура работы с драйвером

Процедура работы с драйвером выглядит следующим образом:

- загрузка драйвера с помощью вызова `CreateFile` (вызывает также рабочую процедуру `IRP_MJ_CREATE`);

- вызов либо стандартных рабочих процедур (например, процедуру `IRP_MJ_READ` с помощью функции `ReadFile`, см. табл. 12.1), либо пользовательских процедур, описанных в рабочей процедуре `IRP_MJ_DEVICE_CONTROL` с помощью вызова `DeviceIoControl` (*см. разд. 12.1.5*);
- закрытие драйвера с помощью вызова `CloseHandle`.

Описание и параметры использованных функций описаны в справочной части книги.

Загрузка драйвера производится по имени драйвера или устройства, например, как показано в листинге 12.18.

Листинг 12.18. Загрузка драйвера по имени драйвера или по имени устройства

```
// Загрузка по имени устройства
hComHandle:= CreateFile(
 '\\.\COM1' // 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_FLAG_OVERLAPPED, // асинхронный режим доступа
 0 // Должно быть 0 для COM портов
);
// Загрузка по имени драйвера
hDevice:= CreateFile('\\.\giveioex',
 GENERIC_READ or GENERIC_WRITE,
 0,nil,
 OPEN_EXISTING,
 FILE_ATTRIBUTE_NORMAL,
 0
);
```

В качестве имени устройства указывается одно из символьных имен физического устройства (*см. разд. 1.3.4*). Например, COM1, хотя и кажется именем устройства, на самом деле является символьным именем устройства \Device\Serial0. Для того чтобы драйвер был загружен, системе нужно знать физическое расположение файла драйвера. Для этого в специальной ветке реестра содержится запись о драйвере.

12.2.2. Регистрация драйвера

Для того чтобы драйвер мог быть обнаружен системой (т. е. по имени драйвера был найден и загружен физический файл), он должен быть зарегистрирован в специальной ветке реестра.

Регистрация драйвера может производиться с помощью INF-файла (*см. разд. 13.1*), с помощью специальных утилит или программно, с помощью специальной компоненты Windows, называемой SCM-менеджер (Service Control Manager, менеджер управления сервисами).

12.2.2.1. Регистрация с помощью SCM-менеджера

Преимущество использования SCM-менеджера состоит в том, что его функции позволяют динамически загружать и выгружать драйверы непосредственно из пользовательских программ, не прибегая к использованию Мастера установки оборудования. Таким образом, приложение само может определять время присутствия драйвера в операционной системе. Однако отметим, что не все драйверы могут быть загружены и запущены средствами SCM-менеджера.

Листинг 12.19 показывает регистрацию драйвера с помощью SCM-менеджера, состоящую из следующих шагов:

1) регистрация сервиса:
   - открытие SCM-менеджера с помощью вызова функции `OpenSCMManager` с флагом `SC_MANAGER_ALL_ACCESS`;
   - регистрация драйвера как SCM-сервиса с помощью вызова функции `CreateService`;
   - закрытие дескриптора сервиса с помощью вызова функции `CloseServiceHandle`;
   - закрытие дескриптора менеджера с помощью вызова функции `CloseServiceHandle`;
2) старт сервиса:
   - открытие SCM-менеджера с помощью вызова функции `OpenSCMManager` с флагом `SC_MANAGER_CONNECT`;
   - открытие сервиса, соответствующего драйверу, с помощью функции `OpenService` с флагом `SERVICE_START`;
   - старт сервиса с помощью функции `StartService`;
   - закрытие дескриптора сервиса;
   - закрытие дескриптора SCM-менеджера.

Процедура останова и удаления драйвера, показанная в листинге 12.20, является "симметричной" копией процедуры старта:

- останов сервиса:
  - открытие SCM-менеджера с помощью вызова функции `OpenSCMManager` с флагом `SC_MANAGER_CONNECT`;
  - открытие сервиса, соответствующего драйверу, с помощью функции `OpenService` с флагом `SERVICE_STOP`;
  - останов сервиса с помощью функции `ControlService` с флагом `SERVICE_CONTROL_STOP`;
  - закрытие дескриптора сервиса;
  - закрытие дескриптора SCM-менеджера;
- удаление сервиса:
  - открытие SCM-менеджера с помощью вызова функции `OpenSCMManager` с флагом `SC_MANAGER_ALL_ACCESS`;
  - открытие сервиса, соответствующего драйверу с помощью функции `OpenService` с флагом `SERVICE_ALL_ACCESS`;
  - удаление записи о драйвере с помощью функции `DeleteService`;
  - закрытие дескриптора сервиса;
  - закрытие дескриптора SCM-менеджера.

Листинг 12.19. Регистрация драйвера в реестре и старт сервиса

```
Const
 // имя драйвера
 DriverName : PChar= 'giveioex'#0;
 // имя файла драйвера
 FileDriver : String = 'giveioex.sys'#0;

// Процедура регистрации драйвера
Function TGiveIOEx.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\giveioex}
hSCMan:= WinSvc.OpenSCManager(
           Nil,         { локальный }
           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;

// Старт зарегистрированного сервиса
Function TGiveIOEx.StartService : Boolean;
var lpServiceArgVectors : PChar;
```

```
    hSCMan, hService : SC_HANDLE;
    DriverPath : String;
Begin
 Result:= False;

// Старт 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;
```

Листинг 12.20. Останов сервиса и удаление драйвера

```
// Останов сервиса
Function TGiveIOEx.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;

// Удаление сервиса
Function TGiveIOEx.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;
```

На рис. 12.3 виден результат успешной регистрации драйвера, после которой обращение к драйверу, которое мы рассмотрим далее, выглядит довольно просто (листинг 12.21).

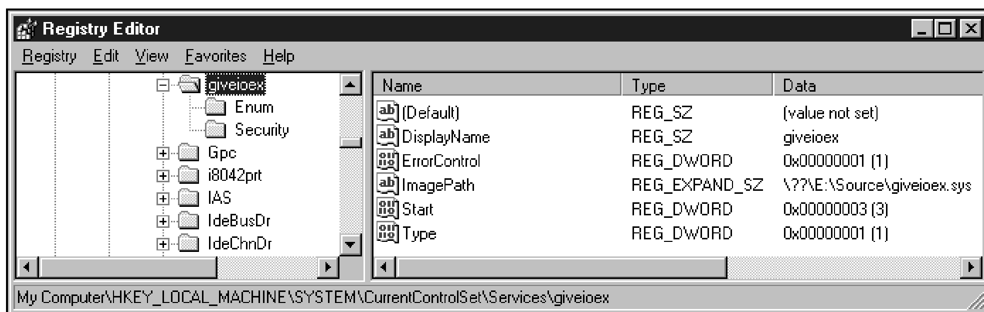


Рис. 12.3. Драйвер зарегистрирован

12.2.2.2. Параметры драйвера в реестре

При регистрации драйвера в реестре создаются несколько стандартных записей (рис. 12.3).

- Параметр `DisplayName` (тип `REG_SZ`). Значение этого параметра описывает текст, используемый в служебных программах операционной системы, в частности, в программах Панели управления. В случае если данный параметр не указан, используется имя драйвера.
- Параметр `ErrorControl` (тип `REG_DWORD`). Этот параметр описывает способ обработки ошибки при загрузке или инициализации драйвера. Возможные варианты приведены в табл. 12.3.
- Параметр `ImagePath` (тип `REG_EXPAND_SZ`). Значение этого параметра описывает полный путь к файлу, содержащему исполняемый код драйвера. По умолчанию значение этого параметра равно `%system%\Drivers\имя_драйвера`.
- Параметр `Start` (тип `REG_DWORD`). Значение этого параметра описывает стадию загрузки операционной системы, когда следует загружать драйвер. Возможные варианты приведены в табл. 12.4.
- Параметр `Type` (тип `REG_DWORD`). Значение этого параметра описывает тип драйвера. Некоторые значения этого параметра приведены в табл. 12.5.
- Параметры подраздела `Enum`. Подраздел `Enum` в ветке описания драйвера присутствует постоянно для драйверов, загруженных с помощью Мастера установки оборудования. Для драйверов, загружаемых при помощи SCM-сервиса, он появляется только после их удачного старта. В этом разделе присутствуют параметры `Count` (число обнаруженных устройств), `NextInstance` и параметры `1`, `2` и т. д. Параметры `1`, `2` и т. п. появляются

только для удачно стартовавших драйверов, а их значения указывают на ветку HKLM\System\CurrentControlSet\Enum (где отражаются все когда-либо удачно стартовавшие драйверы).

Таблица 12.3. *Значения параметра* ErrorControl

Значение	Символьное имя	Описание
0x00	SERVICE_ERROR_IGNORE	Ошибки игнорируются, загрузка продолжается без уведомлений об ошибках в данном драйвере
0x01	SERVICE_ERROR_NORMAL	Ошибки игнорируются, но сообщения об ошибках выводятся, при этом загрузка продолжается
0x02	SERVICE_ERROR_SEVERE	Порядок загрузки нарушается и начинается заново с использованием набора параметров последней успешной загрузки, а если он уже используется, то ошибка игнорируется
0x03	SERVICE_ERROR_CRITICAL	Порядок загрузки нарушается и начинается заново с использованием набора параметров последней успешной загрузки, а если он уже используется, то загрузка прерывается и выводится сообщение об ошибке

Таблица 12.4. *Значения параметра* Start

Значение	Символьное имя	Описание
0x00	SERVICE_BOOT_START	Драйвер запускается загрузчиком ОС
0x01	SERVICE_SYSTEM_START	Драйвер запускается на стадии загрузки компонентов ядра ОС
0x02	SERVICE_AUTO_START	Драйвер будет запущен средствами SCM-менеджера после загрузки компонентов ядра ОС
0x03	SERVICE_DEMAND_START	Драйвер запущен пользовательским приложением при помощи средств SCM-менеджера
0x04	SERVICE_DISABLED	Драйвер не может быть запущен

Таблица 12.5. Значения параметра `Type`

Значение	Символьное имя	Описание
0x01	SERVICE_KERNEL_DRIVER	Драйвер режима ядра
0x02	SERVICE_FILE_SYSTEM_ DRIVER	Драйвер файловой системы
0x04	SERVICE_ADAPTER	Драйвер адаптера

12.2.3. Обращение к рабочим процедурам

Обращение к стандартным рабочим процедурам производится с помощью стандартных функций Windows API, а к пользовательским — с помощью вызова функции `DeviceIoControl`.

Соответствие рабочих процедур и функций Windows API приведено в табл. 12.1. Например, вызов рабочей процедуры с кодом `IRP_MJ_READ` производится с помощью вызова функции `ReadFile`.

Вызов функции `DeviceIoCOntrol` всегда вызывает рабочую процедуру `IRP_MJ_DEVICE_CONTROL`. Различение пользовательских процедур производится с помощью *подкода* (sub code), передаваемого при вызове `DeviceIoControl` (*см. разд. 12.1.5*).

Листинг 12.21 демонстрирует вызов пользовательской рабочей процедуры с кодом $900.

Листинг 12.21. Загрузка и обращение к рабочим процедурам драйвера

```
Procedure TGiveIOEx.GiveIoForProcess(dwProcessId : Cardinal);
var hDevice : SC_HANDLE; Result : Cardinal;
begin
 // Загрузить драйвер с именем giveioex
 hDevice:= CreateFile('\\.\giveioex',
   GENERIC_READ or GENERIC_WRITE,
   0,nil,
   OPEN_EXISTING,
   FILE_ATTRIBUTE_NORMAL,
   0
 );
```

```
 // Обращение к рабочей процедуре с кодом IRP_MJ_DEVICE_CONTROL
 // и подкодом $900
 DeviceIoControl(hDevice, Get_Ctl_Code($900),
    @dwProcessId, SizeOf(dwProcessId), nil, 0, Result, nil);
 // Закрытие драйвера
 CloseHandle(hDevice);
end;
```

12.2.4. Хранение драйвера внутри исполняемого файла

В заключение этого раздела сделаем небольшое замечание. В вышеприведенном случае для работы программы требуется два файла — файл программы и файл драйвера. Для удобства установки и распространения программы можно использовать специальный ресурсный файл, позволяющий "встроить" файл драйвера в файл программы, и распространять один файл вместо двух. Для этого нужно в любом редакторе ресурсов создать бинарный ресурс, загрузив его из файла драйвера. Полученный файл ресурса (он имеет расширение res) с помощью директивы `{$R имя_файла.res}` включают в модуль программы, содержащий обращение к драйверу, а затем выполняют следующие шаги (листинг 12.22):

- создают файл драйвера;
- записывают в созданный файл содержимое ресурса;
- с созданным файлом драйвера работают как обычно;
- удаляют файл драйвера.

Таким образом, динамическое создание и уничтожение файла драйвера будет производиться прозрачно для программы.

Листинг 12.22. Динамическое создание файла драйвера

```
// Подключение файла ресурсов
{$R GiveIoEx.RES}

// получение пути к системной директории
function GetSystemDir : String;
var Buffer: array[0..1023] of Char;
begin
 SetString(Result, Buffer, GetSystemDirectory(Buffer, SizeOf(Buffer)));
end;
```

```
// получение имени драйвера
function GetDriverPath : String;
begin
 Result:= GetSystemDir + '\giveioex.sys';
end;

// создание файла драйвера из ресурса
Function TGiveIOEx.CreateFileFromResource : Boolean;
Var S: TFileStream;  Rsrc: HRSRC; Res: THandle;
    Data: Pointer;
Begin
 Result := False;
 // 101 - номер ресурса в ресурсном файле
 Rsrc := FindResource(HInstance, MakeIntResource(101), RT_RCDATA);
 If Rsrc = 0 then Exit;

 Res:= LoadResource(HInstance, Rsrc);
 Try
  Data := LockResource(Res);
  If Data <> nil then
   Try
    S:= TFileStream.Create(GetDriverPath, fmCreate);
    Try
     S.WriteBuffer(Data^, SizeOfResource(HInstance, Rsrc));
    Finally
     S.Free;
    End;
    Result:= True;
   Finally
    UnlockResource(Res);
   End;
  Finally
   FreeResource(Res);
  End;
End;
```

```
// удаление файла драйвера
Procedure TGiveIOEx.RemoveFile;
var DriverPath : String;
Begin
 DriverPath:= GetDriverPath;
 Windows.DeleteFile(PChar(DriverPath));
End;
```

Естественно, такая процедура возможна только для динамической загрузки драйвера с помощью SCM-менеджера.

12.3. Создание драйвера с помощью Driver Studio

Ранее мы описали несколько инструментов, с помощью которых можно быстро создать драйвер USB-устройства (*см. гл. 3*). В этом разделе мы рассмотрим процесс создания драйвера с помощью NuMega Driver Studio.

Для работы нам потребуется установленные Microsoft Visual Studio и Microsoft DDK (дальше мы будем пользоваться сокращениями VS и DDK соответственно).

Установка Driver Studio (сокращенно DS) довольно проста. После инсталляции в VS появляется дополнительное меню со следующими кнопками (рис. 12.4):

- запуск помощника создания драйвера;
- запуск помощника создания сетевого драйвера;
- изменение переменных окружения;
- компиляция с помощью утилиты BUILD из DDK.

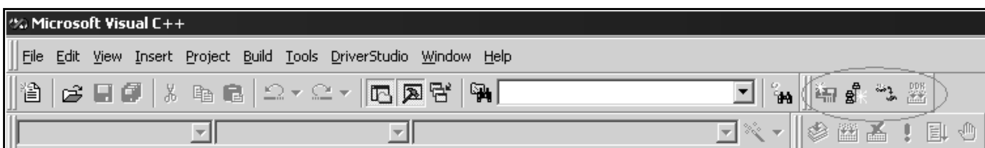


Рис. 12.4. Меню Driver Studio в MS Visual Studio

Сразу отметим, что для компиляции наших драйверов можно использовать либо компиляцию из среды VS, либо воспользоваться командными файлами

make_XP.bat и make2000.bat. Эти файлы можно найти на прилагаемом к книге компакт-диске.

В случае возникновения проблем с компиляцией (особенно в Windows 2000) проверьте правильность переменных окружения. Должны существовать переменные (мы приводим только основные) со следующими значениями:

- `BASEDIR` — путь к папке DDK, например, `F:\WINXPDDK`;
- `DRIVERWORKS` — путь к папке DS в коротком формате, например, `F:\PROGRA~1\NuMega\SOFTIC~1\DRIVER~3`;
- `MSDevDir` — путь к папке MSDev, например, `F:\VStudio\Common\MSDev98`.

Настройку переменных DS можно провести прямо из среды VS, нажав третью кнопку в меню (рис. 12.5).

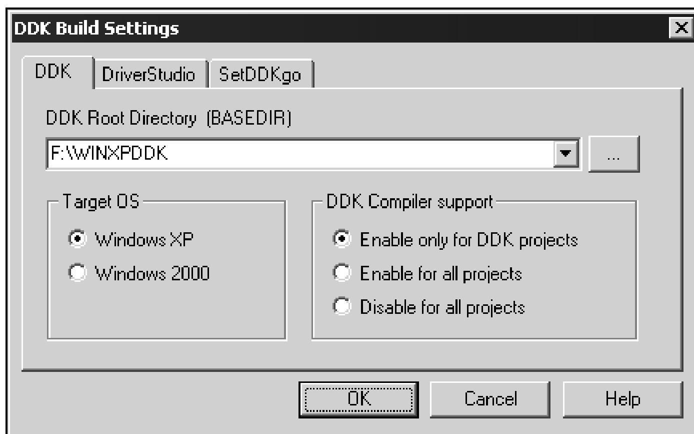


Рис. 12.5. Настройка переменных Driver Studio

После установки потребуется скомпилировать библиотеки DS (вообще говоря, при первой компиляции драйвера эти библиотеки должны компилироваться автоматически, но в некоторых случая этого не происходит). Сделать это можно из командной строки, с помощью командного файла bldlib.bat, находящегося в папке DriverWorks. Для компиляции нужно указать два параметра — тип компиляции и тип библиотек:

- компиляция Debug-версии: `bldlib.bat checked wdm`
- компиляции Release-вервии: `bldlib.bat free wdm`

Откомпилировать библиотеку можно и с помощью файла проекта VdwLibs.dsw, находящегося в папке DriverWorks\source.

После компиляции в папке DriverWorks\lib\i386 должен появиться файл vdw_wdm.lib.

12.3.1. Несколько слов о библиотеке Driver Studio

Библиотека классов DS представляет собой надстройку над чистым WDM API. Это избавляет программиста от использования довольно запутанных низкоуровневых функций, позволяя, тем не менее, выполнение любых необходимых операций.

В нашей книге мы не сможем описать все классы DS, поэтому мы ограничимся только теми, которые потребуются нам для написания драйвера USB-устройства.

12.3.1.1. Класс *KDriver*

Класс `KDriver` является базовым классом драйвера. Его задача — предоставить стандартные базовые функции (такие как `DriverEntry`, `AddDevice`, `Unload` и т. д.). Для этого класса не важно, к какому оборудованию обращается драйвер. Для управления оборудованием создается экземпляр класса `KDevice`.

Для реализации специфических драйверов библиотека предоставляет несколько наследников `KDriver` (рис. 12.6).

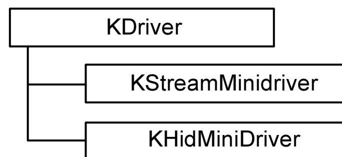


Рис. 12.6. Наследование классов `KDriver`

12.3.1.2. Класс *KDevice*

Класс `KDevice` является базовым классом объекта устройства. Когда драйвер получает запрос IRP, объект `KDriver` не обрабатывает запрос самостоятельно. Он передает его объекту `KDevice`, который отвечает за непосредственную связь с оборудованием.

Сам по себе класс `KDevice` используется редко. Чаще используют один из его потомков (рис. 12.7), например, `KPnpDevice`. Каждый драйвер должен иметь хотя бы один объект устройства.

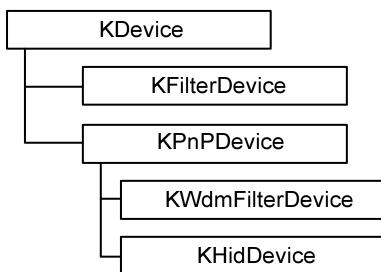


Рис. 12.7. Наследование классов `KDevice`

Объект устройства содержит виртуальные методы для обработки запросов на чтение (`ReadFile`), запись (`WriteFile`) или выполнение специальных функций (`DeviceIoControl`). Конкретный экземпляр объекта устройства должен перекрыть эти методы для придания объекту нужной функциональности.

12.3.1.3. Класс *KIrp*

Класс `KIrp` представляет собой оболочку для структуры пакета запроса ввода/вывода IRP (*см. разд. 12.1.4*). Как мы увидим далее, создание пакета запроса с помощью этого класса значительно проще, чем создание структуры IRP напрямую.

12.3.1.4. Класс *KRegistryKey*

Класс `KRegistryKey` позволяет обращаться к данным драйвера, сохраненным в реестре. Данные сохраняются в ветке HKLM\SYSTEM\CurrentControlSet\Services\<имя драйвера>\Parameters\.

Перечисление секций и данных доступно с помощью функций `EnumerateSubkey` и `EnumerateValue`. Для доступа к конкретным данным класс имеет методы `QueryValue` (чтение), `WriteValue` (запись), `DeleteValue` (удаление).

12.3.1.5. Класс *KLowerDevice*

Класс `KLowerDevice` является базовым классом для устройства нижнего уровня. Библиотека предоставляет несколько наследников этого класса, реализующих интерфейсы устройств нижнего уровня (рис. 12.8).

Класс `KUsbLowerDevice` предоставляет основные функции для обращения к USB-интерфейсу (листинг 12.23).

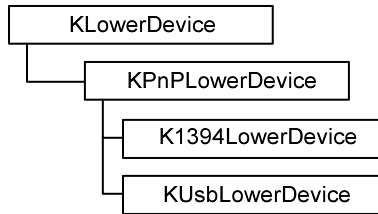


Рис. 12.8. Наследование классов `KLowerDevice`

Листинг 12.23. Некоторые функции `KUsbLowerDevice`

```
class KUsbLowerDevice : public KPnpLowerDevice
{
public:
 // Функция выбора конфигурации
 AC_STATUS ActivateConfiguration(
            UCHAR ConfigurationValue, // номер конфигурации
            ULONG MaxConfigSize=1024  // максим. размер кофигурации
           );
 // Получение строкового дескриптора
 NTSTATUS GetStringDescriptor(
           UCHAR Index,          // номер строки
           PWCHAR pStr,          // буфер для строки
           UCHAR MaxLen,         // размер буфера
           SHORT LangId = 0x109  // код языка
          );
// Получение счетчика кадров
 ULONG GetCurrentFrameNumber(void);
// Передача запроса SET_FEATURE
 NTSTATUS SetFeature(
           USHORT Feature,  // код функции
           PIO_COMPLETION_ROUTINE CompletionRoutine=NULL,
           PVOID Context=NULL
          );
// Передача запроса CLEAR_FEATURE
 NTSTATUS ClearFeature(
```

```
          USHORT Feature, // код функции
          PIO_COMPLETION_ROUTINE CompletionRoutine=NULL,
          PVOID Context=NULL
         );
// передача пакета запроса ввода/вывода драйверу нижнего уровня
 NTSTATUS SubmitUrb(
          PURB pUrb,
          PIO_COMPLETION_ROUTINE CompletionRoutine = NULL,
          PVOID Context=NULL,
          ULONG mSecTimeOut=0
         );
```

Параметр `CompletionRoutine` позволяет указать адрес функции, которая будет вызвана при завершении операции. Это позволяет организовать асинхронное выполнение операций ввода/вывода.

Параметр `pUrb` позволяет передавать указатель на буфер, который содержит данные, передаваемые или принимаемые от устройства. Передаваемый пакет данных мы будем называть URB-пакет.

12.3.1.6. Классы USB

Мы уже описали несколько классов, относящихся к USB: класс `KHidMiniDriver` (наследник `KDriver`), класс `KHidDevice` (наследник `KDevice`) и класс `KUsbLowerDevice` (наследник `KLowerDevice`). Эти классы имеют предопределенную функциональность. Конечно же, библиотека DS предоставляет классы для реализации пользовательских интерфейсов.

Объект класса `KUsbInterface` предоставляет функции для работы с USB-интерфейсами. Драйвер может создавать столько интерфейсов, сколько их описано в дескрипторе конфигурации.

Объект класса `KUsbPipe` предоставляет функции для работы с конечными точками. Драйвер должен создавать конечные точки только с теми значениями параметров, которые указаны в дескрипторе конфигурации. Создание несуществующих конечных точек приведет к краху системы.

Принцип работы с этими классами следующий:

- в конструкторе экземпляра `KDriver` создается экземпляр класса `KPnpDevice`;
- в конструкторе экземпляра `KPnpDevice` создаются:
  - экземпляр `m_Lower` класса `KUsbLowerDevice`, предоставляющий доступ к функциям низкого уровня;

- один или несколько экземпляров класса интерфейса `KUsbInterface`;
- при необходимости создаются экземпляры классов конечных точек `KUsbPipe`;

- в функции драйвера `OnStartDevice` выполняется активизация одной из конфигураций с помощью вызова `m_Lower.ActivateConfiguration()`;
- при получении запроса ввода/вывода, например, запроса чтения, производится следующая последовательность действий:
  - с помощью методов конечной точки создается и инициализируется пакет `PURB`, например, вызовом `BuildBulkTransfer`, `BuildControlTransfer` или `BuildInterruptTransfer`;
  - созданный запрос передается драйверу нижнего уровня с помощью вызова метода конечной точки `SubmitUrb`;
  - полученные данные (сохраненные в пакете URB) передаются пользовательской программе, инициировавшей запрос.

12.3.2. Другие классы Driver Studio

Для общности приведем несколько классов, которые могут оказаться полезными при разработке драйвера.

- `KIoRegister` — порт ввода/вывода. Этот класс позволяет читать и записывать в порт 8-, 16-, и 32-битные значения.
- `KIoRange` — диапазон адресов ввода/вывода. Практически представляет собой массив экземпляров класса `KIoRegister`.
- `KMemoryRegister` представляет собой отдельную ячейку памяти в адресном пространстве устройства.
- `KMemoryRange` представляет собой диапазон адресов памяти.
- `KEvent` — объект синхронизации работы потоков.
- `KTimer` — объект таймера.
- `KSystemThread` — класс для создания нового потока в драйвере.
- `KList` — класс списка.
- `KFile` — класс для работы с файлами.
- `KUnicode` — класс для работы со строками.

12.3.3. Создание шаблона драйвера с помощью Driver Studio

Итак, для создания драйвера все готово. Нажимаем первую кнопку меню...

12.3.3.1. Шаг 1. Задание имени и пути проекта

На первом шаге создания проекта необходимо задать имя проекта и путь, по которому будут располагаться файлы драйвера (рис. 12.9).

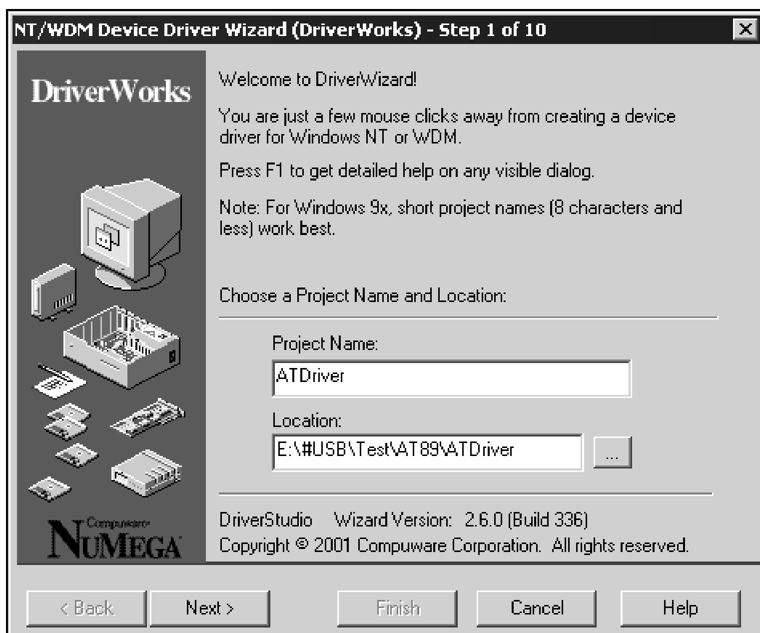


Рис. 12.9. Первый шаг создания драйвера

12.3.3.2. Шаг 2. Выбор архитектуры драйвера

На втором шаге необходимо выбрать архитектуру создаваемого драйвера: **Windows NT 4.0** или **WDM** (рис. 12.10). Первый вариант уже практически не используется, поэтому мы выбираем модель WDM.

12.3.3.3. Шаг 3. Выбор шины

На третьем шаге необходимо выбрать шину, на которой располагается устройство, для которого создается драйвер (рис. 12.11). В нашем случае это USB (заметим, что если драйвер создается для последовательного или параллельного интерфейса, то следует выбрать **None**, т. к. обращение к таким портам производится напрямую, без дополнительного драйвера шины).

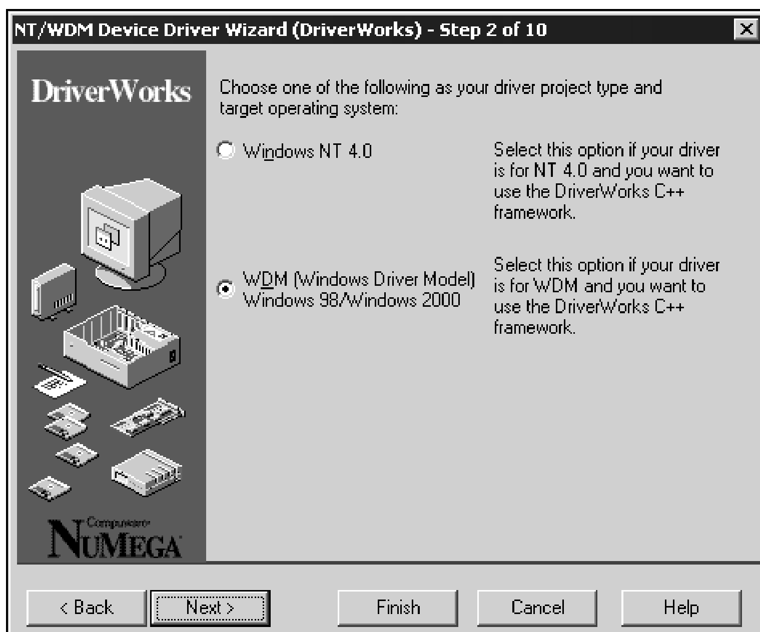


Рис. 12.10. Второй шаг создания драйвера

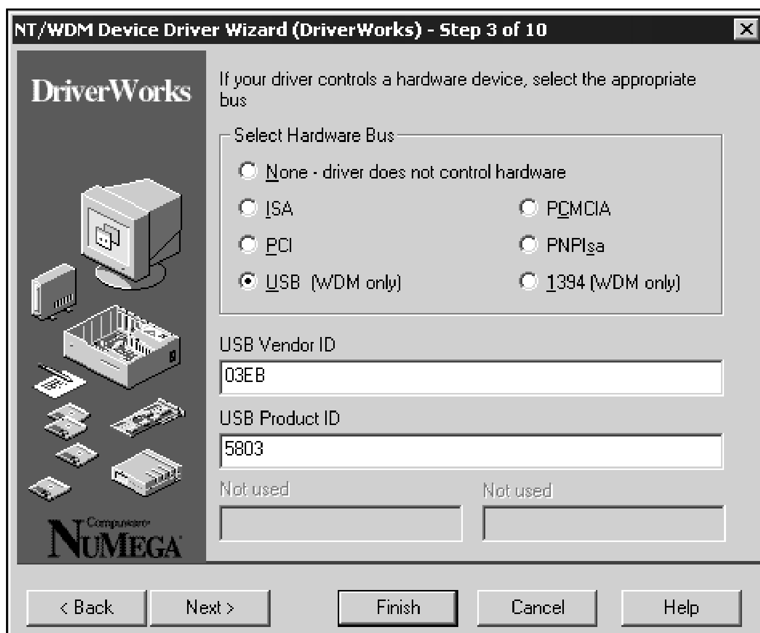


Рис. 12.11. Третий шаг создания драйвера

Для USB-устройств следует указать параметры устройства: идентификатор производителя (USB Vendor ID) и идентификатор продукта (USB Product ID). Конечно, следует указывать те же идентификаторы, которые были указаны в дескрипторе устройства.

Заметим, что указанные значения идентификаторов используются только для формирования INF-файла, и их можно легко изменить и после генерации файлов драйвера. Так, в нашем случае в INF-файл будет записана строка:

```
[Manufacturer]
%MfgName%=Mfg0
[Mfg0]
%DeviceDesc%=ATDriver_DDI, USB\VID_03EB&PID_5803
```

12.3.3.4. Шаг 4. Задание набора конечных точек

На четвертом шаге необходимо задать набор конечных точек, поддерживаемых устройством (рис. 12.12). Свойства конечных точек должны в точности совпадать со свойствами, описанными в дескрипторе конфигурации, иначе запуск драйвера приведет к краху системы.

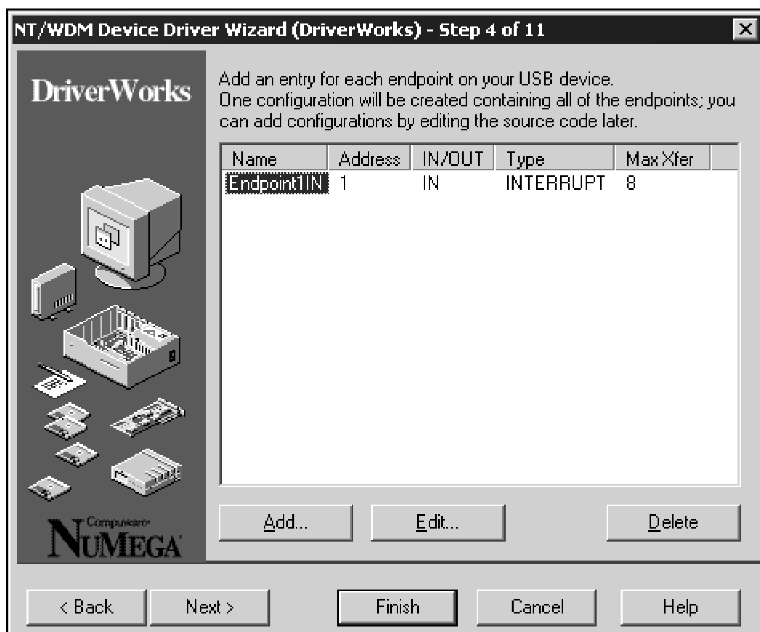


Рис. 12.12. Четвертый шаг создания драйвера

Для добавления новой конечной точки следует нажать кнопку **Add** и заполнить требуемые поля свойств конечной точки (рис. 12.13).

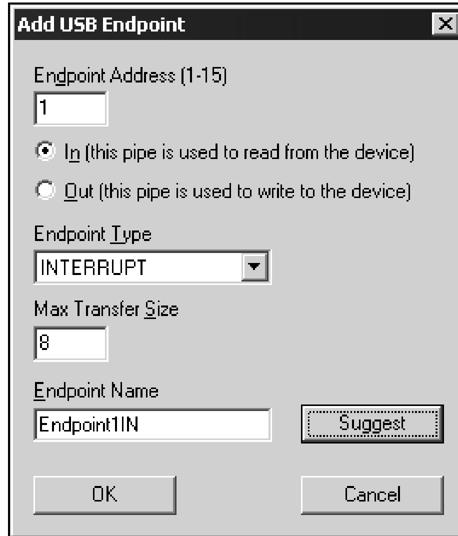


Рис. 12.13. Добавление конечной точки

Кнопка **Suggest** в диалоге добавления конечной точки позволяет автоматически сгенерировать имя переменной для конечной точки.

12.3.3.5. Шаг 5. Задание имени класса и файла

На пятом шаге (рис. 12.14) необходимо задать имена, которые будут присвоены файлу C++, который содержит класс драйвера (**File Name**), и самому классу драйвера (**Driver Class**). Эти имена создаются на основе имени проекта и обычно их оставляют без изменений.

12.3.3.6. Шаг 6. Выбор функций драйвера

На шестом шаге необходимо выбрать функции, которые будет поддерживать драйвер (рис. 12.15). Возможные варианты функциональности:

- **Read** — драйвер будет обрабатывать запросы на чтение;
- **Write** — драйвер будет обрабатывать запросы на запись;
- **Flush** — драйвер поддерживает функцию сброса буферов;
- **Device Control** — драйвер будет поддерживать пользовательские запросы с помощью функции `DeviceIoControl`;

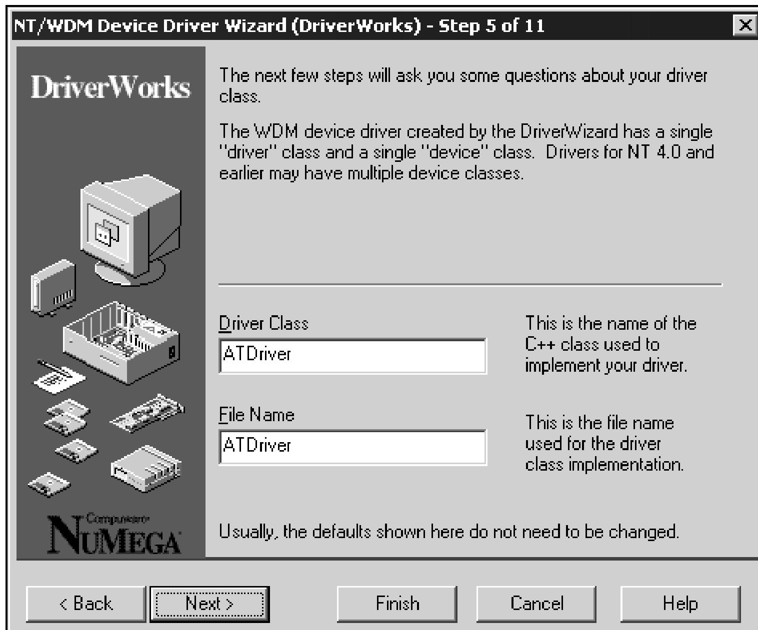


Рис. 12.14. Пятый шаг создания драйвера

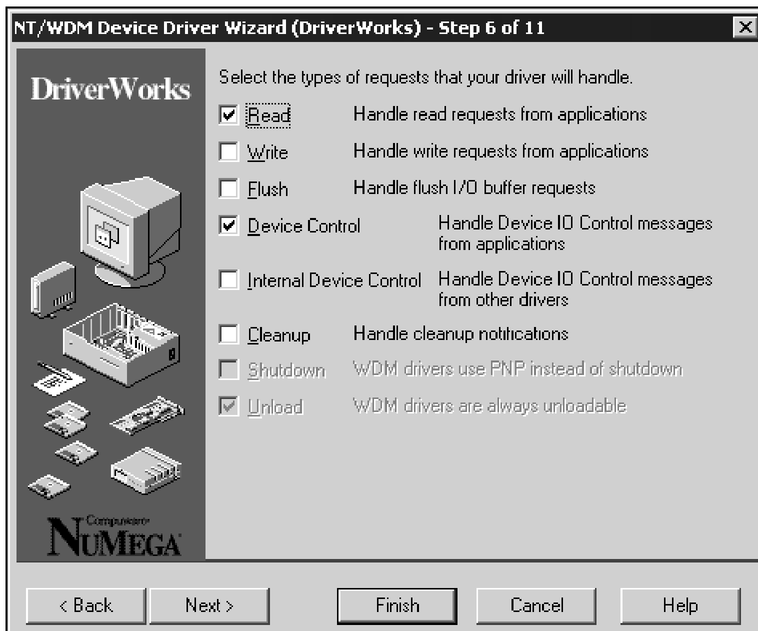


Рис. 12.15. Шестой шаг создания драйвера

- **Internal Device Control** — драйвер будет обрабатывать запросы от других драйверов;
- **Cleanup** — драйвер будет обрабатывать запросы на очистку буфера обмена.

Для нашего драйверы мы оставим только обработку функций чтения и пользовательских запросов.

12.3.3.7. Шаг 7. Выбор способа обработки запросов

На седьмом шаге необходимо выбрать способ обработки запросов (рис. 12.16). Опция **Select queuing method** позволяет выбрать, каким образом будут буферизироваться запросы на ввод/вывод:

- **None** — запросы не буферизируются в очереди;
- **DriverManaged** — драйвер содержит одну или более одной очередей, в которой сохраняются запросы на ввод/вывод, пришедшие от других драйверов или системы;
- **SystemManaged** — драйвер использует только одну очередь сообщений.

Также надо выбрать, будут ли буферизироваться запросы на чтение и запись.

Для драйверов USB-устройств выбор не предоставляется, поэтому просто пропускаем этот шаг.

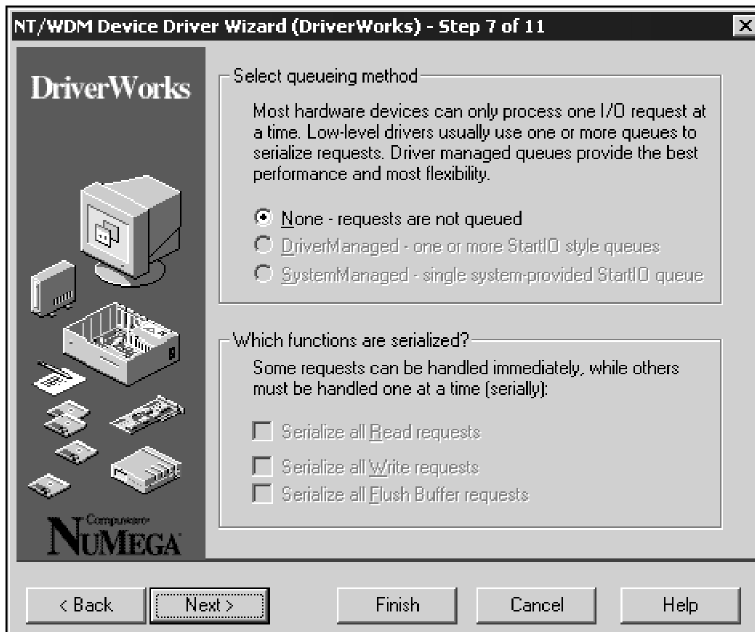


Рис. 12.16. Седьмой шаг создания драйвера

12.3.3.8. Шаг 8. Создание сохраняемых параметров драйвера

На седьмом шаге можно задать параметры, которые драйвер будет загружать из реестра Windows при старте (рис. 12.17).

Рис. 12.17. Восьмой шаг создания драйвера

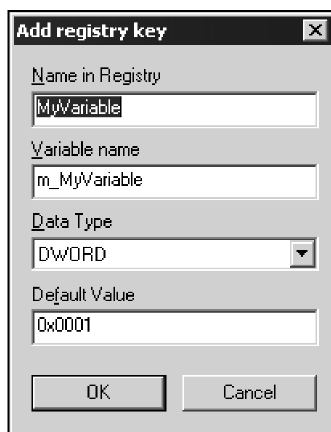


Рис. 12.18. Добавление нового сохраняемого параметра

Для добавления нового параметра нужно нажать кнопку **Add**. При этом задается параметр реестра, имя переменной, тип параметра и его значение по умолчанию (рис. 12.18). Запись и считывание параметров из реестра позволяет драйверу задавать какие-либо конфигурационные параметры, сохранять данные, необходимые для его запуска или работы.

12.3.3.9. Шаг 9. Свойства драйвера

Девятый шаг создания драйвера состоит из нескольких частей. В верхней части окна мастера задается имя класса драйвера. С помощью кнопки **Rename** можно задать другое имя.

На вкладке **Interface** (рис. 12.19) можно задать имя драйвера (как он будет открываться с помощью функции `CreateFile`) и способ обращения к драйверу — по символьному имени (*см. разд. 1.3.4*) или по уникальному идентификатору (GUID).

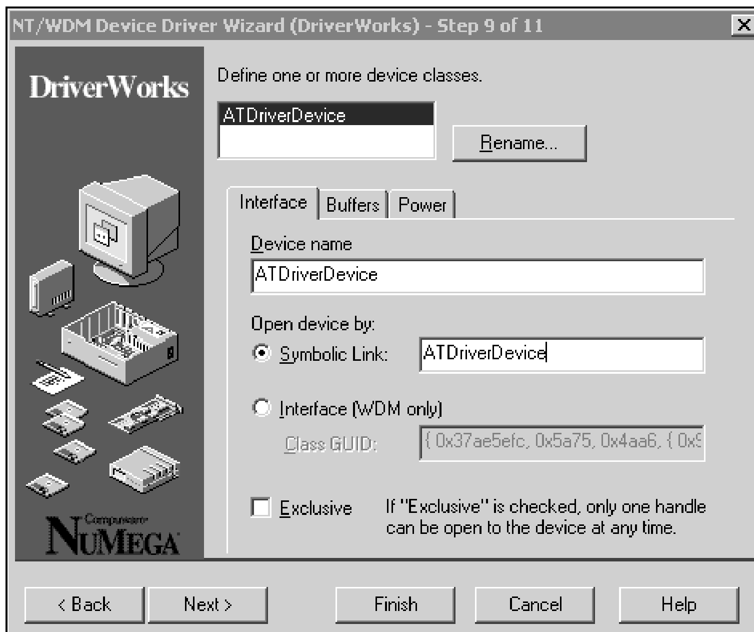


Рис. 12.19. Закладка **Interface** девятого шага создания драйвера

На вкладке **Buffers** (рис. 12.20) можно выбрать способ передачи буферов памяти.

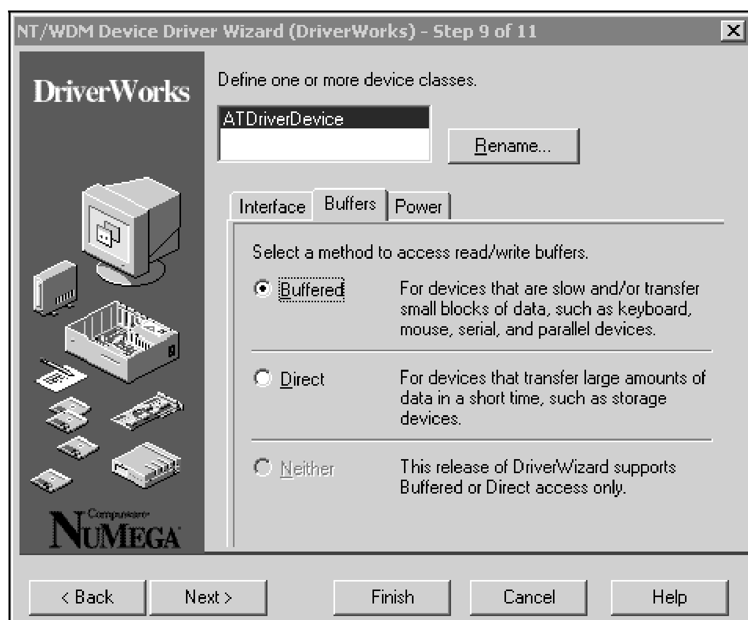


Рис. 12.20. Вкладка **Buffers** девятого шага создания драйвера

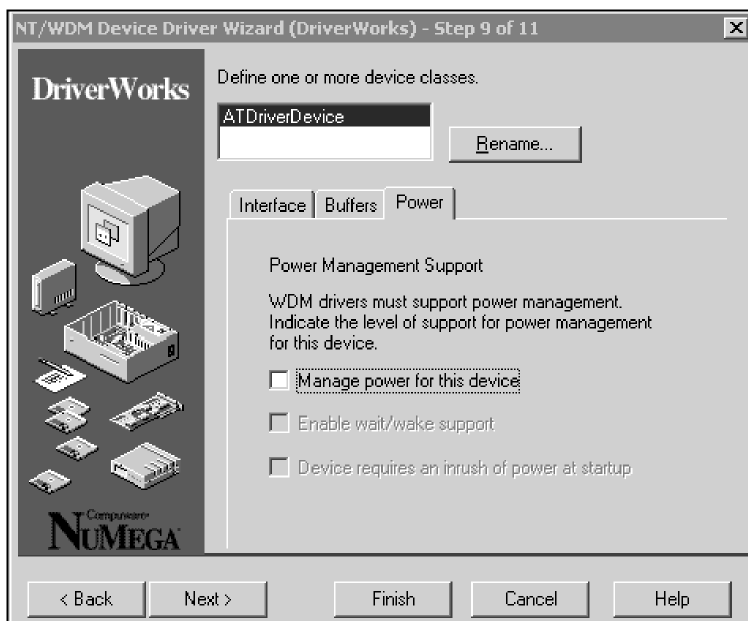


Рис. 12.21. Вкладка **Power** девятого шага создания драйвера

При выборе опции **Buffered** будут создаваться промежуточные буферы внутри драйвера, а затем данные будут копироваться в буфер пользовательской программы, а выбор опции **Direct** заставит драйвер работать с буферами пользовательской программы напрямую. Подробную информацию о доступе к буферам можно найти в *разд. 12.1.5*.

На вкладке **Power** (рис. 12.21) можно выбрать способ управления энергопотреблением.

12.3.3.10. Шаг 10. Задание кодов IOCTL

На десятом шаге задаются коды функции `DeviceIoControl` (рис. 12.22), если конечно была выбрана поддержка этой функции на 6 шаге.

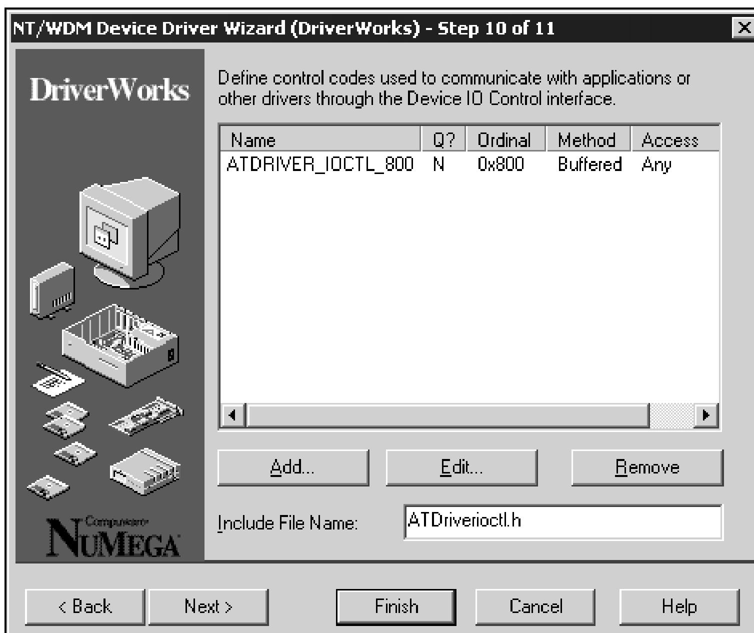


Рис. 12.22. Десятый шаг создания драйвера

Кнопка **Add** позволяет добавить код пользовательской функции (рис. 12.23).

12.3.3.11. Шаг 11. Дополнительные настройки

На последнем шаге создания драйвера (рис. 12.24) выбирается, нужно ли создавать тестовое приложение для драйвера (**Create test console application**) и дополнительные настройки отладки и создания лога событий.

Вообще говоря, тестовое приложение нам не нужно, т. к. для создания приложения мы будем использовать Delphi, но для простейшего тестирования драйвера можно использовать и приложение C++.

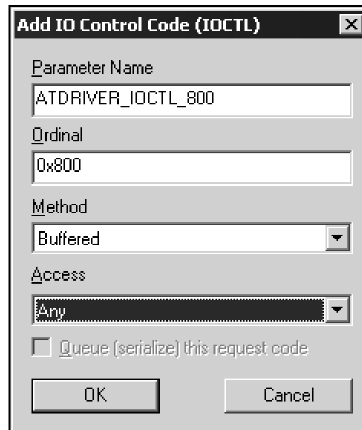


Рис. 12.23. Добавление кода пользовательской функции

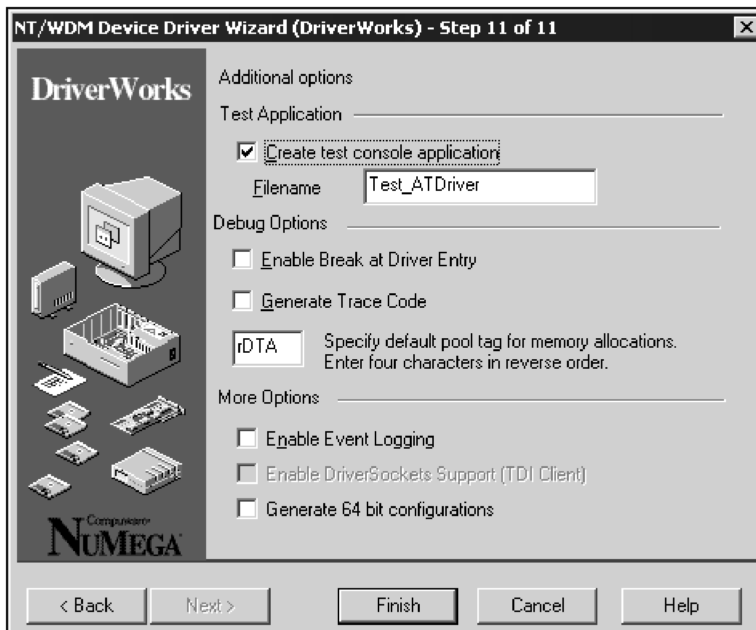


Рис. 12.24. Последний шаг создания драйвера

12.3.4. Доработка шаблона драйвера

Получившийся драйвер является заготовкой для настоящего драйвера и требует небольшой доработки. Точнее, требуется выполнить следующий набор действий:

1) дописать реализацию функций ввода/вывода;
2) сгенерированный драйвер содержит только один интерфейс, поэтому при необходимости требуется создать нужное число экземпляров класса `KUsbInterface`;
3) по умолчанию активизируется конфигурация номер 1; если устройство имеет другие настройки или несколько конфигураций, нужно обеспечить их функционирование;
4) дописать реализацию функций IOCTL.

12.3.5. Базовые методы класса устройства

Листинг 12.24 показывает код класса `ATDriver` (мы позволили себе несколько сократить листинг, выделив только важные места кода). Обратим внимание на переменную `m_Unit`. При загрузке драйвера (функция `DriverEntry`) эта переменная обнуляется, а при каждом вызове `AddDevice` увеличивается на единицу. Это позволяет создавать экземпляры класса `ATDriverDevice` с уникальными в пределах системы именами. Первое устройство, для которого будет загружен драйвер, будет иметь имя ATDriverDevice0, второе ATDriverDevice1 и т. д.

Листинг 12.24. Код класса `ATDriver` (с сокращениями)

```
#define VDW_MAIN
#include <vdw.h>
#include <kusb.h>
#include "ATDriver.h"
#include "ATDriverDevice.h"

#pragma hdrstop("ATDriver.pch")

// Generated by DriverWizard version DriverStudio 2.6.0 (Build 336)
////////////////////////////////////////////////////////////////////
//  ATDriver::DriverEntry
// Вызывается при загрузке драйвера
////////////////////////////////////////////////////////////////////
```

```
NTSTATUS ATDriver::DriverEntry(PUNICODE_STRING RegistryPath)
{
  m_Unit = 0;
  return STATUS_SUCCESS;
}
// End INIT section
/////////////////////////////////////////////////////////////////////
#pragma code_seg()
/////////////////////////////////////////////////////////////////////
//  ATDriver::AddDevice
// Вызывается при подключении устройства
/////////////////////////////////////////////////////////////////////
NTSTATUS ATDriver::AddDevice(PDEVICE_OBJECT Pdo)
{
  ATDriverDevice * pDevice = new (
   static_cast<PCWSTR>(KUnitizedName(L"ATDriverDevice", m_Unit)),
   FILE_DEVICE_UNKNOWN,
   static_cast<PCWSTR>(KUnitizedName(L"ATDriverDevice", m_Unit)),
   0,
   DO_BUFFERED_IO
  )
  ATDriverDevice(Pdo, m_Unit);

  m_Unit++;
  DbgPrint("Unit number is %d", m_Unit);

  return status;
}
```

Собственно работа с устройством сосредоточена в методах класса `ATDriverDevice`. Как мы уже описывали (*см. разд. 12.3.1*), в конструкторе этого класса создаются экземпляры объектов драйвера нижнего уровня (`m_Lower`), интерфейса (`m_Interface`) и конечной точки (`m_Endpoint1IN`). При необходимости здесь же можно создать другие интерфейсы и конечные точки, если они поддерживаются устройством. В листинге 12.25 показан конструктор класса нашего устройства (для простоты мы позволили себе сократить проверки).

Листинг 12.25. Конструктор класса `ATDriverDevice`

```
ATDriverDevice::ATDriverDevice(PDEVICE_OBJECT Pdo, ULONG Unit) :
                                                KPnpDevice(Pdo, NULL)
{
  DbgPrint("ATDriverDevice::ATDriverDevice START");
  NTSTATUS status;

  DbgPrint("Unit number is %d", Unit);
  // Запомнить номер устройства
  m_Unit = Unit;

 // Создание объекта драйвера нижнего уровня
 status = m_Lower.Initialize(this, Pdo);
 // Создание объекта интерфейса
 status = m_Interface.Initialize(
           m_Lower,  // Объект драйвера нижнего уровня
           0,        // Номер интерфейса
           1,        // Номер конфигурации
           0         // Номер альтернативного интерфейса
          );
 // Создание конечной точки
 status = m_Endpoint1IN.Initialize(m_Lower, 0x81 /* 8 */);
 // Информирование драйвера нижнего уровня о создании
 // нового интерфейса
 SetLowerDevice(&m_Lower);
 // Инициалиазция политики PnP
 SetPnpPolicy();
}
```

> ***Важно!***
>
> По умолчанию конструктор конечной точки принимает три параметра: указатель на объект драйвера нижнего уровня, номер конечной точки и максимальный размер пакета. Последний параметр нужно самостоятельно удалить из конструктора, т. к. размер пакета будет указываться при создании пакета IRP.

Следующий важный метод — `OnStartDevice`, вызываемый при начале работы устройства. Код этого метода показан в листинге 12.26. Если устройство

поддерживает несколько конфигураций, можно изменить номер активной конфигурации.

Листинг 12.26. Метод `ATDriverDevice::OnStartDevice`

```
NTSTATUS ATDriverDevice::OnStartDevice(KIrp I)
{
  NTSTATUS status = STATUS_UNSUCCESSFUL;
  AC_STATUS acStatus = AC_SUCCESS;

  I.Information() = 0;

  // Активизация конфигурации 1
  acStatus = m_Lower.ActivateConfiguration(
              1    // Номер конфигурации
             );

 if (acStatus == AC_SUCCESS)
 {
  // Конфигурация установлена успешно
  status = STATUS_SUCCESS;
 }
 if (acStatus == AC_FAILED_TO_OPEN_PIPE_OBJECT)
 {
  // Конфигурация установлена, но созданных конечных
  // точек в ней не существует
  status = STATUS_SUCCESS;
 }

  return status;
}
```

Все остальные методы, кроме метода чтения данных (`Read`) и обработчика пользовательской функции (`ATDRIVER_IOCTL_800_Handler`), мы оставляем без изменений. Метод `Read` мы разберем в следующем разделе, а метод пользовательских функций мы уже рассматривали (*см. разд. 12.1.5*) и повторяться не будем. Его можно использовать, например, для чтения строковых дескрипторов или других целей.

12.3.6. Реализация чтения данных

Метод `Read` вызывается, когда пользовательское приложение вызывает функцию `ReadFile`. Код, генерируемый по умолчанию, не производит никаких действий, возвращая пакет нулевой длины.

Нашей задачей является создание пакета URB для организации чтения данных с конечной точки `m_Endpoint1IN`. Листинг 12.27 показывает реализацию метода `Read` для чтения данных с конечной точки типа `Interrupt`.

Листинг 12.27. Метод `Read` для конечной точки типа `Interrupt`

```
NTSTATUS ATDriverDevice::Read(KIrp I)
{
  // Если запрошено ноль байт, то всегда возвращаем успех
  if (I.ReadSize() == 0)
  {
    I.Information() = 0;
    return I.PnpComplete(this, STATUS_SUCCESS);
  }
  NTSTATUS status = STATUS_SUCCESS;

  // Указатель на буфер для чтения данных
  PVOID   pBuffer = I.BufferedReadDest();
  // Число запрошенных для чтения байт
  ULONG   dwTotalSize = I.ReadSize(CURRENT);
  // Будет сохранять число реально прочитанных байтов
  ULONG   dwBytesRead = 0;
  // Если конечная точка не активна, то чтение невозможно
  if (!m_Endpoint1IN.IsOpen())
  {
    I.Information() = 0;
    return I.PnpComplete(this, STATUS_INSUFFICIENT_RESOURCES);
  }
  // Максимальное число байтов для выбранной конечной точки
  ULONG dwMaxSize = m_Endpoint1IN.MaximumTransferSize();
  // Если запрошено больше, чем можно прочитать, то
  // ограничим запрос максимальным размером
  if (dwTotalSize > dwMaxSize) dwTotalSize = dwMaxSize;
```

```
  // Создает запрос для чтения данных типа Interrupt
  PURB pUrb = m_Endpoint1IN.BuildInterruptTransfer(
               pBuffer,     // Буфер для чтения
               dwTotalSize, // Сколько данных нужно прочитать
               TRUE         // разрешаем использование коротких пакетов
              );
  pUrb->UrbBulkOrInterruptTransfer.TransferFlags =
          (USBD_TRANSFER_DIRECTION_IN | USBD_SHORT_TRANSFER_OK);
  // Передаем сформированный запрос драйверу нижнего уровня
  status = m_Endpoint1IN.SubmitUrb(pUrb);
  // Возвращаем реально прочитанное число байтов
  dwBytesRead = pUrb->UrbBulkOrInterruptTransfer.TransferBufferLength;
  // Освободить пакет
  delete pUrb;
  // Вернуть реально прочитанное число байтов
  I.Information() = dwBytesRead;
  // Подтвердить успешное завершение операции
  return I.PnpComplete(this, status);
}
```

12.3.7. Установка драйвера

Скомпилировав драйвер, мы получим файл ATDriver.sys. Теперь, подключив наше устройство, нужно установить этот драйвер, указав путь к файлу ATDriver.inf и файлу самого драйвера (рис. 12.25).

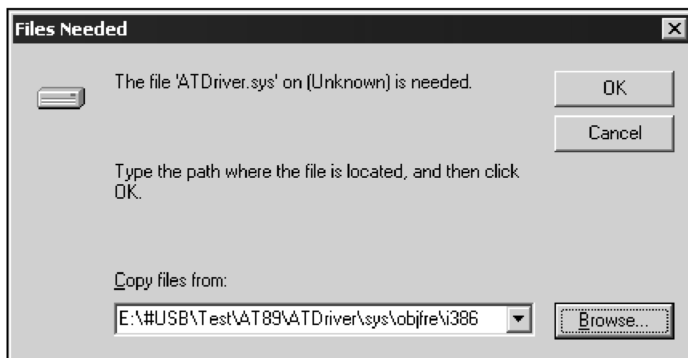


Рис. 12.25. Указание файла драйвера

Если все прошло успешно система обнаружит драйвер и установит его (рис. 12.26). Теперь можно заняться созданием программы для чтения данных.

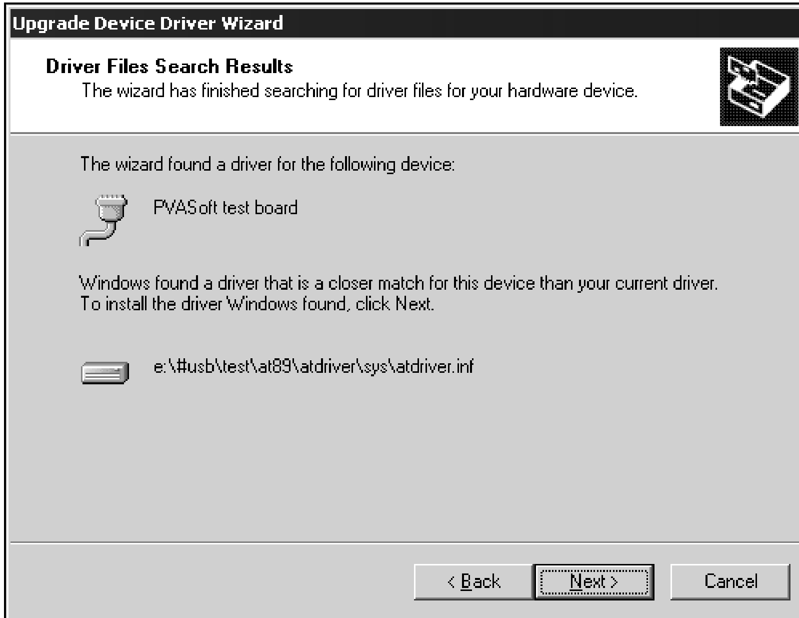


Рис. 12.26. Драйвер обнаружен

12.3.8. Программа чтения данных

Мы достаточно подробно разобрали методы загрузки и обращения к драйверам (*см. разд. 12.2*), поэтому, нам кажется, дополнительных комментариев к листингу 12.28 не требуется.

Листинг 12.28. Модуль для загрузки и обращения к драйверу `ATDriverDevice`

```
unit DrvLoader;

interface

Uses Windows, SysUtils;

type
 TAT89DriverLoader = class
```

```
  private
   hDevice : THandle; // хранить дескриптор драйвера
  Public
   Constructor Create;
   Destructor  Destroy; override;
  Public
   // Регистрация драйвера
   Function CreateService(SysPath : String) : Boolean;
  // Удаление драйвера
   Function RemoveService : Boolean;
  // Старт драйвера
   Function StartService  : Boolean;
  // Останов драйвера
   Function StopService   : Boolean;
  public
  // Открытие драйвера
   Function  Open(DevIndex : Integer) : Boolean;
  // Закрытие драйвера
   Procedure Close;
  public
  // Вызов функции DeviceIoControl
   Function Func1 : Cardinal;
  // Вызов функции чтения
   Function Read  : String;
 End;

implementation

Uses WinSvc, Dialogs;

// Конструктор
Constructor TAT89DriverLoader.Create;
Begin
 Inherited Create;
 hDevice:= INVALID_HANDLE_VALUE;
End;
```

```
// Деструктор
Destructor  TAT89DriverLoader.Destroy;
Begin
 Close;
 Inherited Destroy;
End;

// Описание драйвера
Const
 // Имя драйвера
 DriverName : PChar= 'ATDriverDevice'#0;
 // Имя файла драйвера
 FileDriver : String = 'ATDriver.sys'#0;
 // Полное имя драйвера
 DriverPath : String = '\\.\ATDriverDevice';

Function TAT89DriverLoader.CreateService(SysPath : String) : Boolean;
var hSCMan, hService: SC_HANDLE;
    DriverPath : String;
Begin
 Result:= False;

 hSCMan:= WinSvc.OpenSCManager(
           Nil,         { local }
           Nil,         { SERVICES_ACTIVE_DATABASE }
           SC_MANAGER_ALL_ACCESS
          );
 If hSCMan = 0 then Exit;

 // Получаем полное имя к файлу драйвера
 // Если драйвер находится в текущем каталоге,
 // где запускается программа, то не добавляем SysPath
 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(SysErrorMessage(GetLastError), mtError, [mbOK], 0);
  CloseServiceHandle(hSCMan);
  Exit;
 End;

 // Регистрация успешна
 WinSvc.CloseServiceHandle(hService);
 WinSvc.CloseServiceHandle(hSCMan);
 Result:= True;
End;

// Удаление регистрации драйвера
Function TAT89DriverLoader.RemoveService : Boolean;
var 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;
 // Удалить
 WinSvc.DeleteService(hService);
 WinSvc.CloseServiceHandle(hService);
 WinSvc.CloseServiceHandle(hSCMan);
 Result:= True;
End;
```

```
// Старт сервиса драйвера
Function TAT89DriverLoader.StartService  : Boolean;
var lpServiceArgVectors : PChar;
    hSCMan, hService : SC_HANDLE;
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;

// Останов сервиса драйвера
Function TAT89DriverLoader.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;
 // Останов
 WinSvc.ControlService(hService, SERVICE_CONTROL_STOP, serviceStatus);
 WinSvc.CloseServiceHandle(hService);
 WinSvc.CloseServiceHandle(hSCMan);
 Result:= True;
End;

// Открытие драйвера. Параметр DevIndex передает номер
// устройства (согласно параметру m_Unit в листинге 14.4)
Function TAT89DriverLoader.Open(DevIndex : Integer) : Boolean;
Var DevName : String;
Begin
 If DevIndex <> -1 then DevName:= DriverPath + IntToStr(DevIndex)
  Else DevName:= DriverPath;

 hDevice:= CreateFile(@DevName[1],
            GENERIC_READ or GENERIC_WRITE,
            0,nil,
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,
            0
          );
 Result:= hDevice <> INVALID_HANDLE_VALUE;
 If not Result then begin
  MessageDlg(SysErrorMessage(GetLastError), mtError, [mbOK], 0);
 End;
End;
// Закрыть дескриптор драйвера
Procedure TAT89DriverLoader.Close;
Begin
 If hDevice <> INVALID_HANDLE_VALUE then
  CloseHandle(hDevice);
 hDevice:= INVALID_HANDLE_VALUE;
End;
```

```
// Параметры вызова DeviceIoControl
Const
 FILE_DEVICE_UNKNOWN = $22;
 METHOD_BUFFERED     = 0;
 FILE_ANY_ACCESS     = $0000;

// Формирование CTL кода
function Get_Ctl_Code(Nr: Integer): Integer;
begin
  Result:=
     (FILE_DEVICE_UNKNOWN shl 16) or
     (FILE_ANY_ACCESS     shl 14) or
     (Nr                  shl  2) or
     METHOD_BUFFERED;
end;

// Выполнение функции DeviceIoControl
Function TAT89DriverLoader.Func1 : Cardinal;
Var Res : Cardinal;
Begin
 If hDevice = INVALID_HANDLE_VALUE then Exit;
 DeviceIoControl(hDevice, Get_Ctl_Code($800),...);
End;

// Функция чтения 8 байт
Function TAT89DriverLoader.Read  : String;
Var A : Array[1..8] of Byte;
    ReadBytes : Cardinal;
    i : Integer;
Begin
 // Инициализация буфера
 For i:= 1 to 8 do A[i]:= 0;

 Result:= '';
 If hDevice = INVALID_HANDLE_VALUE then Exit;
```

```
 // Чтение
 If not ReadFile(hDevice, A, 8, ReadBytes, nil) then
  MessageDlg(SysErrorMessage(GetLastError), mtError, [mbOK], 0);

 // Шестнадцатеричная строка
 For i:= 1 to 8 do
  Result:= Result + IntToHex(A[i],2)+' ';
End;

end.
```

Код тестовой программы, использующей модуль `DrvLoader`, показан в листинге 12.29, а вид формы и результат работы — на рис. 12.27 (для контроля мы запустили программу DebugView, показывающую отладочные сообщения драйвера).

Листинг 12.29. Тестовая программа чтения данных с USB-устройства

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, DrvLoader, ExtCtrls, Spin;

type
  TForm1 = class(TForm)
    btnLoad: TButton;
    btnUnload: TButton;
    Bevel1: TBevel;
    btnOpen: TButton;
    btnClose: TButton;
    Bevel2: TBevel;
    btnFunc1: TButton;
    EOutput: TEdit;
    Button1: TButton;
```

```
    SpinEdit1: TSpinEdit;
    procedure btnLoadClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnUnloadClick(Sender: TObject);
    procedure btnOpenClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure btnFunc1Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
  public
    Driver : TAT89DriverLoader;
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

// Создание компонента TDriverLoader
procedure TForm1.FormCreate(Sender: TObject);
begin
 Driver:= TAT89DriverLoader.Create;
end;

// Уничтожение компонента TDriverLoader
procedure TForm1.FormDestroy(Sender: TObject);
begin
 Driver.Free;
end;

// Регистрация сервиса в реестре и старт
procedure TForm1.btnLoadClick(Sender: TObject);
begin
 Driver.CreateService(ExtractFilePath(ParamStr(0)));
```

```
 Driver.StartService;
end;

// Останов сервиса и удаление из реестра
procedure TForm1.btnUnloadClick(Sender: TObject);
begin
 Driver.StopService;
 Driver.RemoveService;
end;

// Открытие драйвера
procedure TForm1.btnOpenClick(Sender: TObject);
begin
 If Driver.Open(SpinEdit1.Value)
  then MessageDlg('Драйвер открыт успешно', mtConfirmation, [mbOK], 0)
  else MessageDlg('Ошибка открытия драйвера', mtError, [mbOK], 0);
end;

// Закрытие драйвера
procedure TForm1.btnCloseClick(Sender: TObject);
begin
 Driver.Close;
end;

// Выполнение DeviceIoControl($800)
procedure TForm1.btnFunc1Click(Sender: TObject);
begin
 EOutput.Text:= IntToStr(Driver.Func1);
end;

// Чтение данных
procedure TForm1.Button1Click(Sender: TObject);
begin
 EOutput.Text:= Driver.Read;
end;

end.
```

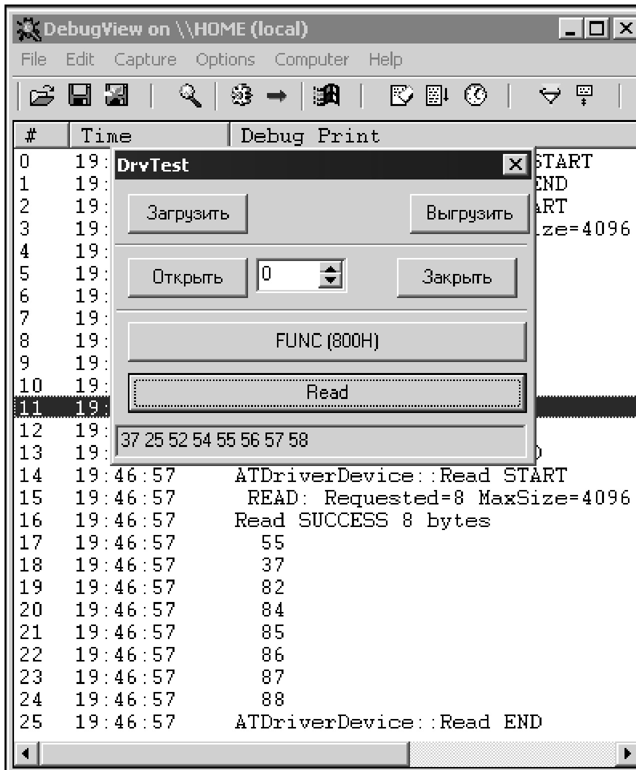


Рис. 12.27. Результат чтения данных с USB-устройства

12.3.9. Чтение данных с конечных точек других типов

В предыдущем разделе мы пользовались конечной точкой типа `Interrupt`. При необходимости можно легко заменить тип конечной точки. Листинг 12.30 показывает константы для дескриптора конфигурации конечной точки типа `Bulk`, а листинг 12.31 — изменения в коде драйвера.

Листинг 12.30. Константы для конечной точки типа `Bulk`

```
/* первая конечная точка */
#define EP_1_CONFIG        (BULK|EP_CONFIG_IN)  /* конфигурация */
#define EP_1_ADDRESS       (1|EP_DIRECT_IN)     /* адрес         */
#define EP_1_ATTRIBUTES    BULK                 /* аттрибуты     */
```

Листинг 12.31. Изменение в коде драйвера для чтения конечной точки типа `Bulk`

```
NTSTATUS ATDriverDevice::Read(KIrp I)
{
... ... ... ... ... ... ... ... ... ... ... ... ...
  PURB pUrb = m_Endpoint1IN.BuildBulkTransfer(
                pBuffer,         // Буфер
                dwTotalSize,     // Сколько байт читать
                TRUE,            // Направление (TRUE = IN)
                NULL,
                FALSE,           // Разрешить короткие чтения
                NULL
              );
... ... ... ... ... ... ... ... ... ... ... ... ...
  pUrb->UrbBulkOrInterruptTransfer.TransferFlags =
                 (USBD_TRANSFER_DIRECTION_IN | USBD_SHORT_TRANSFER_OK);
... ... ... ... ... ... ... ... ... ... ... ... ...
  dwBytesRead = pUrb->UrbBulkOrInterruptTransfer.TransferBufferLength;
... ... ... ... ... ... ... ... ... ... ... ... ...
}
```

Для использования конечных точек типа `Control` достаточно изменить метод формирования пакета запроса:

```
  PURB pUrb = m_Endpoint1IN.BuildControlTransfer(
                pBuffer,         // Буфер
                dwTotalSize,     // Сколько данных читать?
                TRUE             // Направление (TRUE = IN)
              );
```

12.3.10. "Чистый" драйвер USB-устройства

Использование Driver Studio избавило нас от множества рутинной работы, однако не стоит забывать, что эта программа стоит довольно существенных денег, в то время как использование Windows DDK доступно всем разработчикам.

Основную структуру WDM-драйвера мы уже описали. В этом разделе мы приведем схему разработки такого драйвера для шины USB.

Общий "план" драйвера показан на рис. 12.28.

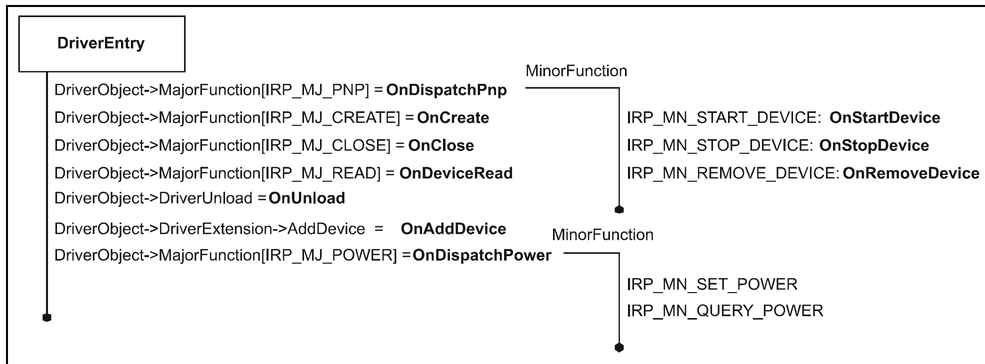


Рис. 12.28. Общий "план" драйвера USB-устройства

Основными обработчиками драйвера являются:

- `DriverEntry` — точка входа в драйвер;
- `OnAddDevice` — вызывается при добавлении нового устройства;
- `OnCreate` — вызывается при создании устройства;
- `OnClose` — вызывается при закрытии устройства;
- `OnUnload` — вызывается при выгрузке драйвера;
- `OnDispatchPower` — диспетчер управления энергопотреблением;
- `OnDispatchPnp` — диспетчер для обработки командами PnP:
    - `OnStartDevice` — старт устройства;
    - `OnStopDevice` — останов устройства;
    - `OnRemoveDevice` — удаление устройства;
- `OnDeviceRead` — вызывается при обработке функции `ReadFile`.

Регистрация основных обработчиков производится в процедуре `DriverEntry` (листинг 12.32).

Листинг 12.32. Регистрация обработчиков драйвера

```
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
```

```
{
  NTSTATUS ntStatus = STATUS_SUCCESS;

  // Основные обработчики драйвера
  DriverObject->MajorFunction[IRP_MJ_CREATE]= OnCreate;
  DriverObject->MajorFunction[IRP_MJ_CLOSE ]= OnClose;
  DriverObject->MajorFunction[IRP_MJ_READ  ]= OnDeviceRead;
  DriverObject->MajorFunction[IRP_MJ_PNP   ]= OnDispatchPnp;
  DriverObject->MajorFunction[IRP_MJ_POWER ]= OnDispatchPower;
  DriverObject->DriverUnload                = OnUnload;
  DriverObject->DriverExtension->AddDevice  = OnAddDevice;

  return ntStatus;
}
```

В отличие от NT-драйвера, в WDM-драйвере создание объекта устройства и регистрация символьных имен должны выполняться внутри процедуры `OnAddDevice` (листинг 12.33).

Листинг 12.33. Функция `OnAddDevice` драйвера USB-устройства

```
NTSTATUS OnAddDevice(
    IN PDRIVER_OBJECT DriverObject,
    IN PDEVICE_OBJECT PhysicalDeviceObject
    )
{
 NTSTATUS                ntStatus = STATUS_SUCCESS;
 PDEVICE_OBJECT          deviceObject = NULL;
 PDEVICE_EXTENSION       pdx;

 WCHAR NameBuffer[] = L"\\Device\\" DEVICE_NAME_STRING;
 WCHAR DOSNameBuffer[] = L"\\DosDevices\\" DEVICE_NAME_STRING;
 UNICODE_STRING uniNameString, uniDOSString;

 // Создание буферов для имен
 RtlInitUnicodeString(&uniNameString, NameBuffer);
```

```
 RtlInitUnicodeString(&uniDOSString , DOSNameBuffer);
 DbgPrint("UniName=%s DosName=%s", uniDOSString, uniNameString);

 // Инициализация объекта драйвера
 ntStatus = IoCreateDevice(
             DriverObject,
             sizeof (DEVICE_EXTENSION),
             &uniNameString,
             FILE_DEVICE_UNKNOWN,
             0,
             FALSE,
             &deviceObject
          );

 // Создание символьного имени драйвера
ntStatus = IoCreateSymbolicLink(&uniDOSString, &uniNameString);

 // Инициализация блока данных объекта устройства
 DbgPrint("Init device extension");
 pdx = (PDEVICE_EXTENSION) (deviceObject->DeviceExtension);
 pdx->OpenHandles = 0;

 // Драйвер будет использовать прямой ввод/вывод для запросов
 // чтения и записи
 deviceObject->Flags |= DO_DIRECT_IO;

 // Сохраняем ссылку на драйвер нижнего уровня. Ему мы будем
 // пересылать запросы на ввод/вывод
 pdx->StackDeviceObject =
       IoAttachDeviceToDeviceStack(deviceObject, PhysicalDeviceObject);

 return ntStatus;
}
```

В WDM предусмотрен довольно удобный механизм хранения данных, относящихся к конкретному экземпляру драйвера: при создании объекта устрой-

ства в функцию `IoCreateDevice` передается размер блока пользовательских данных (`DeviceExtension`). В любом другом обработчике можно легко получить указатель на этот буфер с помощью следующего кода:

```
PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION )fdo->DeviceExtension;
```

Внутренняя структура этих данных — забота программиста, драйвер только лишь создает блок памяти нужного размера. В нашем примере мы используем описание, показанное в листинге 12.34.

Листинг 12.34. Структура `DEVICE_EXTENSION`

```
typedef struct _DEVICE_EXTENSION
{
  // Объект устройства в стеке IRP
  PDEVICE_OBJECT  StackDeviceObject;
  // Число устройств для этого драйвера
  ULONG           OpenHandles;
  // TRUE, если устройство стартовано
  BOOLEAN         Started;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Обработчики `OnCreate` и `OnClose`, в общем случае, могут не выполнять никаких специальных действий за исключением управления счетчиком копий драйвера (листинг 12.35).

Листинг 12.35. Процедуры `OnCreate` и `OnClose`

```
NTSTATUS OnCreate(
    IN PDEVICE_OBJECT fdo,
    IN PIRP Irp
    )
{
 NTSTATUS ntStatus;
 PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION )fdo->DeviceExtension;

// счетчик открытых устройств
 pdx->OpenHandles++;

 Irp->IoStatus.Status      = STATUS_SUCCESS;
```

```
 Irp->IoStatus.Information = 0;
 ntStatus = Irp->IoStatus.Status;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);
 return ntStatus;
}

NTSTATUS OnClose(
    IN PDEVICE_OBJECT fdo,
    IN PIRP Irp
    )
{
 NTSTATUS ntStatus;
 PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION )fdo->DeviceExtension;

// счетчик открытых устройств
 pdx->OpenHandles--;

 Irp->IoStatus.Status = STATUS_SUCCESS;
 Irp->IoStatus.Information = 0;
 ntStatus = Irp->IoStatus.Status;
 IoCompleteRequest (Irp, IO_NO_INCREMENT);
 return ntStatus;
}
```

Обработчик `OnUnload` может выполнять некоторые действия по освобождению памяти, буферов и т. д., а диспетчер энергопотребления `OnDispatchPower` используется только в случае, если устройство поддерживает соответствующие интерфейсы. В нашем случае эти обработчики не выполняют никаких специальных действий (листинг 12.36).

Листинг 12.36. Обработчики `OnUnload` и `OnDispatchPower`

```
VOID OnUnload(
    IN PDRIVER_OBJECT DriverObject
    )
{
}
```

```
NTSTATUS OnDispatchPower(
    IN PDEVICE_OBJECT fdo,
    IN PIRP           Irp
    )
{
 PIO_STACK_LOCATION irpStack, nextStack;
 PDEVICE_EXTENSION pdx = fdo->DeviceExtension;
 NTSTATUS ntStatus;

 Irp->IoStatus.Status = STATUS_SUCCESS;
 Irp->IoStatus.Information = 0;
 irpStack = IoGetCurrentIrpStackLocation(Irp);

 nextStack = IoGetNextIrpStackLocation(Irp);
 RtlCopyMemory(nextStack, irpStack, sizeof(IO_STACK_LOCATION));

 PoStartNextPowerIrp(Irp);
 ntStatus = PoCallDriver(pdx->StackDeviceObject, Irp);

 if (ntStatus == STATUS_PENDING)
 {
  IoMarkIrpPending(Irp);
 }

 return ntStatus;
}
```

Основную работу по поддержке PnP производит процедура-диспетчер `OnDispatchPnp`, распределяющая вызовы в соответствии с кодом `MinorFunction` (листинг 12.37).

Листинг 12.37. Диспетчер PnP-запросов `OnDispatchPnp`

```
NTSTATUS OnDispatchPnp(
   IN PDEVICE_OBJECT fdo,
   IN PIRP           Irp
   )
```

```
{
   PIO_STACK_LOCATION irpStack;
   PDEVICE_EXTENSION pdx = fdo->DeviceExtension;
   ULONG fcn;
   NTSTATUS ntStatus;

   /* Получаем текущую позицию в стеке драйверов */
   irpStack = IoGetCurrentIrpStackLocation (Irp);
   /* Номер функции */
   fcn = irpStack->MinorFunction;

   switch (fcn)
   {
      /* Обработка старта устройства */
      case IRP_MN_START_DEVICE:
      {
         ntStatus = OnStartDevice(fdo);
         if (ntStatus == STATUS_SUCCESS)
         {
            pdx->Started = TRUE;
         }
         break;
      }
      /* Обработка останова устройства */
      case IRP_MN_STOP_DEVICE:
      {
         // Сначала передаем запрос драйверу ниже по стеку
         IoSkipCurrentIrpStackLocation(Irp);
         IoCallDriver(pdx->StackDeviceObject, Irp);
         // отрабатываем останов нашего устройства
         ntStatus = OnStopDevice(fdo);
         break;
      }
      /* Удаление устройства из системы */
      case IRP_MN_REMOVE_DEVICE:
```

```
      {
         ntStatus = OnRemoveDevice(fdo,Irp);
         break;
      }
      // Все остальные запросы передаем драйверу дальше по стеку
      default:
      {
        IoSkipCurrentIrpStackLocation(Irp);
        ntStatus = IoCallDriver(pdx->StackDeviceObject, Irp);
      }
   }
   return ntStatus;
}
```

Для непосредственного выполнения запросов к устройству необходимо выполнить следующие действия:

- выделить блок *нестраничной памяти* (non paged memory) с помощью вызова `ExAllocatePool`;
- сформировать запрос с помощью одной из функций формирования запросов (например, `UsbBuildGetDescriptorRequest`, `UsbBuildSelectConfigurationRequest`, `UsbBuildInterruptOrBulkTransferRequest` и т. п.);
- передать запрос USBD с помощью функции `DoCallUSBD` (листинг 12.38);
- обработать результат запроса;
- освободить память.

В случае асинхронного вызова два последних действия переносятся в функцию, вызываемую при завершении выполнения операции.

Листинг 12.38. Передача запроса USBD

```
NTSTATUS DoCallUSBD(
    IN PDEVICE_OBJECT fdo,
    IN PURB Urb
    )
{
   NTSTATUS ntStatus, status = STATUS_SUCCESS;
   PDEVICE_EXTENSION pdx;
```

```
    PIRP irp;
    KEVENT event;
    IO_STATUS_BLOCK ioStatus;
    PIO_STACK_LOCATION nextStack;

    pdx = fdo->DeviceExtension;

    // объект синхронизации
    KeInitializeEvent(&event, NotificationEvent, FALSE);

    irp = IoBuildDeviceIoControlRequest(
              IOCTL_INTERNAL_USB_SUBMIT_URB,
              pdx->StackDeviceObject,
              NULL,
              0,
              NULL,
              0,
              TRUE, /* INTERNAL */
              &event,
              &ioStatus
        );

    // получение следующей позиции в стеке драйверов
    nextStack = IoGetNextIrpStackLocation(irp);

    // Формирование параметров для вызова
    nextStack->Parameters.Others.Argument1 = Urb;
   // Вызов
    ntStatus = IoCallDriver(pdx->StackDeviceObject, irp);

   // Если запрос еще выполняется...
    if (ntStatus == STATUS_PENDING)
    {
       status = KeWaitForSingleObject(
                     &event,
                     Suspended,
```

```
                    KernelMode,
                    FALSE,
                    NULL);
   } else {
      ioStatus.Status = ntStatus;
   }
 // Обработка результата запроса
   ntStatus = ioStatus.Status;

   if (NT_SUCCESS(ntStatus))
   {
      if (!(USBD_SUCCESS(Urb->UrbHeader.Status)))
         ntStatus = STATUS_UNSUCCESSFUL;
   }

   return ntStatus;
}
```

Обработчик `OnStartDevice` может содержать код, работающий с нулевой конечной точкой, код получения дескрипторов и код конфигурирования устройства (листинг 12.39).

Листинг 12.39. Обработчик `OnStartDevice`

```
NTSTATUS OnStartDevice(
   IN PDEVICE_OBJECT fdo
  )
{
 PDEVICE_EXTENSION pdx;
 NTSTATUS ntStatus;
 PUSB_DEVICE_DESCRIPTOR deviceDescriptor = NULL;
 PURB urb;
 ULONG size;

 pdx = fdo->DeviceExtension;
 urb = ExAllocatePool( NonPagedPool,
              sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST));
```

```
size = sizeof(USB_DEVICE_DESCRIPTOR);
deviceDescriptor = ExAllocatePool(NonPagedPool, size);

UsbBuildGetDescriptorRequest(
  urb,
  (USHORT) sizeof (struct _URB_CONTROL_DESCRIPTOR_REQUEST),
  USB_DEVICE_DESCRIPTOR_TYPE,
  0,
  0,
  deviceDescriptor,
  NULL,
  size,
  NULL
);

// Передача запроса на выполнение
ntStatus = DoCallUSBD(fdo, urb);

// Отображение дескриптора
if (NT_SUCCESS(ntStatus)) {
 DbgPrint("Device Descriptor:");
 DbgPrint("bLength           %d  ", deviceDescriptor->bLength);
 DbgPrint("bDescriptorType    0x%x", deviceDescriptor->bDescriptorType);
 DbgPrint("bcdUSB            0x%x", deviceDescriptor->bcdUSB);
 DbgPrint("bDeviceClass       0x%x", deviceDescriptor->bDeviceClass);
 DbgPrint("bDeviceSubClass    0x%x", deviceDescriptor->bDeviceSubClass);
 DbgPrint("bDeviceProtocol    0x%x", deviceDescriptor->bDeviceProtocol);
 DbgPrint("bMaxPacketSize0    0x%x", deviceDescriptor->bMaxPacketSize0);
 DbgPrint("idVendor          0x%x", deviceDescriptor->idVendor);
 DbgPrint("idProduct         0x%x", deviceDescriptor->idProduct);
 DbgPrint("bcdDevice         0x%x", deviceDescriptor->bcdDevice);
 DbgPrint("iManufacturer     0x%x", deviceDescriptor->iManufacturer);
 DbgPrint("iProduct          0x%x", deviceDescriptor->iProduct);
 DbgPrint("iSerialNumber     0x%x", deviceDescriptor->iSerialNumber);
}
```

```
 // Освободить занятую память
 ExFreePool(deviceDescriptor);
 ExFreePool(urb);

 if (NT_SUCCESS(ntStatus)) {
  // Конфигурируем устройство
  ntStatus = OnConfigureDevice(fdo);
 }

 return ntStatus;
}
```

Конфигурирование устройства (процедура `OnConfigureDevice`) состоит из следующих действий (листинг 12.40):

1) передача запроса `GET_CONFIGURATION` с минимальным размером буфера (для получения нужного размера буфера);
2) получение полного дескриптора;
3) создание конечных точек;
4) конфигурирование конечных точек.

Листинг 12.40. Конфигурирование устройства

```
NTSTATUS
OnConfigureDevice(
  IN  PDEVICE_OBJECT fdo
 )
{
 PDEVICE_EXTENSION pdx;
 NTSTATUS ntStatus;
 PURB urb = NULL;
 ULONG size;
 PUSB_CONFIGURATION_DESCRIPTOR configurationDescriptor = NULL;
 UCHAR alternateSetting, MyInterfaceNumber;
 PUSBD_INTERFACE_INFORMATION interfaceObject;
 USBD_INTERFACE_LIST_ENTRY   interfaceList;

 pdx = fdo->DeviceExtension;
```

```
// Память для URB
 urb = ExAllocatePool(NonPagedPool,
                 sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST));

// Получить только дескриптор конфигурации
 size = sizeof(USB_CONFIGURATION_DESCRIPTOR) + 16;
 configurationDescriptor = ExAllocatePool(NonPagedPool, size);

 UsbBuildGetDescriptorRequest(urb,
   (USHORT) sizeof (struct _URB_CONTROL_DESCRIPTOR_REQUEST),
   USB_CONFIGURATION_DESCRIPTOR_TYPE,
   0,
   0,
   configurationDescriptor,
   NULL,
   sizeof (USB_CONFIGURATION_DESCRIPTOR),
   NULL
);
 ntStatus = DoCallUSBD(fdo, urb);

// Определение нужного размера буфера
 size = configurationDescriptor->wTotalLength + 16;

// Освободить старый буфер и отвести новый, нужного размера
 ExFreePool(configurationDescriptor);
 configurationDescriptor = NULL;
 configurationDescriptor = ExAllocatePool(NonPagedPool, size);

// Получение полного дескриптора конфигурации
 UsbBuildGetDescriptorRequest(urb,
  (USHORT) sizeof (struct _URB_CONTROL_DESCRIPTOR_REQUEST),
  USB_CONFIGURATION_DESCRIPTOR_TYPE,
  0,
  0,
  configurationDescriptor,
  NULL,
```

```
  size,
  NULL
 );
 ntStatus = DoCallUSBD(fdo, urb);

// Получение конфигурации
 interfaceList.InterfaceDescriptor =
   USBD_ParseConfigurationDescriptorEx(
     ConfigurationDescriptor,
     ConfigurationDescriptor,
     -1, -1, -1, -1, -1
   );

// Создание запроса для получения конфигурации
 urb = USBD_CreateConfigurationRequestEx(
         ConfigurationDescriptor, &interfaceList
       );

// Получение указателя на буфер описания интерфейса
 interfaceObject =
  (PUSBD_INTERFACE_INFORMATION)
         (&(urb->UrbSelectConfiguration.Interface));

// Конфигурирование конечных точек интерфейса
 for (j=0; j<interfaceList[0].InterfaceDescriptor->bNumEndpoints; j++)
{
  PUSBD_PIPE_INFORMATION pipe;
  pipe = &interfaceObject->Pipes[j];

  pipe->MaximumTransferSize = 1024; // установить при необходимости

  DbgPrint("PipeType 0x%x\n", pipe->PipeType);
  DbgPrint("EndpointAddress 0x%x\n", pipe ->EndpointAddress);
  DbgPrint("MaxPacketSize 0x%x\n", pipe ->MaximumPacketSize);
  DbgPrint("Interval 0x%x\n", pipe ->Interval);
  DbgPrint("Handle 0x%x\n", pipe ->PipeHandle);
```

```
  DbgPrint("MaximumTransferSize 0x%x\n", pipe ->MaximumTransferSize);
}

 ntStatus = DoCallUSBD(fdo, urb);

 ExFreePool(urb);
 ExFreePool(configurationDescriptor);

 return ntStatus;
}
```

В обработчике `OnStopDevice` производится передача запроса `SET_CONFIGURATION` с нулевым номером конфигурации, т. е. сброс конфигурации (листинг 12.41).

Листинг 12.41. Обработчик `OnStopDevice`

```
NTSTATUS OnStopDevice(
   IN  PDEVICE_OBJECT fdo
   )
{
 PDEVICE_EXTENSION pdx;
 NTSTATUS ntStatus = STATUS_SUCCESS;
 PURB urb;
 ULONG size;

 pdx = fdo->DeviceExtension;

 size= sizeof(struct _URB_SELECT_CONFIGURATION);
 urb = ExAllocatePool(NonPagedPool, size);

 if (urb)
 {
  NTSTATUS status;

  UsbBuildSelectConfigurationRequest(urb, (USHORT) size, NULL);
  status = DoCallUSBD(fdo, urb);
```

```
  ExFreePool(urb);
 } else {
  ntStatus = STATUS_NO_MEMORY;
 }

 return ntStatus;
}
```

Обработчик `OnRemoveDevice` содержит код, "обратный" коду `OnAddDevice`. В этом обработчике удаляется символьное имя устройства и освобождается объект устройства (листинг 12.42).

Листинг 12.42. Обработчик `OnRemoveDevice`

```
NTSTATUS OnRemoveDevice(
   IN PDEVICE_OBJECT fdo,
   IN PIRP Irp
   )
{
 NTSTATUS ntStatus;
 PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
 ULONG i;


 WCHAR NameBuffer[] = L"\\Device\\" DEVICE_NAME_STRING;
 WCHAR DOSNameBuffer[] = L"\\DosDevices\\" DEVICE_NAME_STRING;
 UNICODE_STRING uniNameString, uniDOSString;

 // Создание буферов для имен
 RtlInitUnicodeString(&uniNameString, NameBuffer);
 RtlInitUnicodeString(&uniDOSString , DOSNameBuffer);
 // Удаление символьного имени драйвера
 ntStatus = IoDeleteSymbolicLink(&uniNameString);

 IoDetachDevice(pdx->StackDeviceObject);
 IoDeleteDevice(fdo);
```

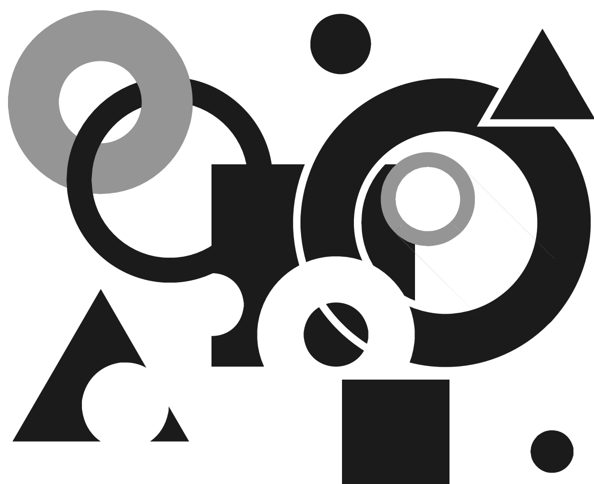
```
 IoSkipCurrentIrpStackLocation(Irp);
 ntStatus = IoCallDriver(pdx->StackDeviceObject, Irp);

 return ntStatus;
}
```

Чтение данных производится в обработчике `OnDeviceRead` (листинг 12.43).

Листинг 12.43. Чтение данных с устройства

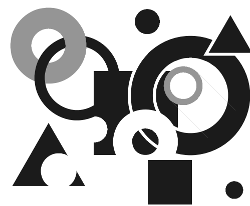
```
NTSTATUS OnDeviceRead(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP pIrp
    )
{
 NTSTATUS                    ntStatus   = STATUS_SUCCESS;
 PUSBD_PIPE_INFORMATION      pipeInfo   = NULL;
 USBD_PIPE_HANDLE            pipeHandle = NULL;
 PURB                        urb        = NULL;
 ULONG                       urbSize    = 0;

 pipeInfo = ... ; // одна из конечных точек
 pipeHandle = pipeInfo->PipeHandle;

 urbSize = sizeof(struct _URB_BULK_OR_INTERRUPT_TRANSFER);

 urb = ExAllocatePool(NonPagedPool,urbSize);

 transferFlags = USBD_SHORT_TRANSFER_OK;
 if (USB_ENDPOINT_DIRECTION_IN(pipeInfo->EndpointAddress))
  transferFlags |= USBD_TRANSFER_DIRECTION_IN;

// Формирование запроса
 UsbBuildInterruptOrBulkTransferRequest(
  urb,
  (USHORT) urbSize,
```

```
  pipeHandle,
  NULL,
  Irp->MdlAddress,
  bufferLength,
  transferFlags,
  NULL
 );

 // Выполнение запроса
 ntStatus = DoCallUSBD(fdo, urb);

 if (NT_SUCCESS(ntStatus))
 {
  Irp->IoStatus.Information =
          urb->UrbBulkOrInterruptTransfer.TransferBufferLength;
 }

// Освободить память
 ExFreePool(urb);

// Вернуть результат запроса
 return ntStatus;
}
```



ЧАСТЬ IV

Справочник



Глава 13

Формат INF-файла

При установке 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.

> ***Важно***
>
> В операционных системах Windows 9*x* размер INF-файла не может превосходить 64 Кбайт. Для Windows NT/2000/XP ограничений нет. Максимальная длина любого поля в INF-файле составляет 512 символов.

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 необходим при создании и установке драйвера устройства.

13.1. Структура INF-файла

Инсталляционный INF-файл поделен на секции, каждая из которых начинается с идентификатора (имени секции), заключенного в квадратные скобки. Часть секций является обязательной, присутствие других секций зависит от назначения драйвера. Порядок следования секций не играет роли, важно лишь, чтобы секции имели корректные имена и были правильно соотнесены в перекрестных ссылках. Секция продолжается до начала следующей секции или до обнаружения конца файла.

Имя секции не должно содержать более 28 символов для Windows 9*x* и более 255 символов для Windows NT/2000/XP. Имя секции может содержать пробелы, если ссылка на такую секцию заключена в кавычки, однако, лучше ограничиваться именами без пробелов.

Записи внутри каждой секции описывают некоторые действия либо ссылаются на другие секции. Запись в секции представляет строку следующего формата:

```
entry = value[, value[ ,value…]]
```

где `entry` является ключевым словом либо маркером (ссылкой на значение, которая заключается между двумя символами "%"). В операционных системах Windows 9*x* все запятые должны присутствовать в количестве, указанном в документации, а в секциях Windows NT замыкающие перечисление запятые можно опускать, если сами значения опущены.

Символ ";" означает в следующей за ним позиции начало комментариев, которые продолжаются до конца строки. Исключение составляют строки, в которых символ ";" заключен в кавычки. Комментарии не принимаются в рассмотрение при анализе файла.

При необходимости продолжить запись на следующей строке в последней позиции текущей строки ставится символ "\".

13.1.1. Секция *Version*

Секция `Version` содержит основную информацию о INF-файле. Рассмотрим значения ключей этой секции:

- `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 (символ "$" в начале и конце строки обязателен):
  - `"$Windows NT$"` — операционные системы NT;
  - `"$Windows 95$"` — системы Windows 9*x*;
  - `"$Chicago$"` — любая версия Window;
- `Class = class-name` — описывает тип устройства. Значение может быть одной из стандартных строк. 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 (табл. 13.1). Длина значения ключа `Class` не может превышать 32 символа;

- `ClassGuid = {nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}` — описывает GUID типа устройства (см. табл. 13.1). Обязательно указывается для нового типа устройства;
- `Provider = %INF-creator%` — содержит имя компании-разработчика INF-файла;
- `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` — его версию. 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.

Таблица 13.1. *Некоторые значения ключей* `Class` *и* `ClassGuid` *секции* `Version`

Имя	Описание	GUID
HIDClass	HID устройства	{745a17a0-74d3-11d0-b6fe-00a0c90f57da}
Ports	Порты (COM и LPT)	{4d36e978-e325-11ce-bfc1-08002be10318}
Image	Цифровые камеры, сканеры и т. д.	{6bdd1fc6-810f-11d0-bec7-08002be2092f}
Multifunction	Многофункциональные устройства, такие как PCMCIA-модемы или сетевые адаптеры	{4d36e971-e325-11ce-bfc1-08002be10318}
System	Системные устройства	{4d36e97d-e325-11ce-bfc1-08002be10318}
USB	Хост-контроллер USB, хабы, но не USB-периферия	{36fc9e60-c465-11cf-8056-444553540000}
NoDriver	Драйвер отсутствует	{4d36e976-e325-11ce-bfc1-08002be10318}
Unknown	Другие устройства	{4d36e97e-e325-11ce-bfc1-08002be10318}

13.1.2. 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. 13.4.5*). 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][.Suit
eMask]]]]
```

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
... ... ... ... ...
```

13.1.3. 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]] ...
```

13.1.3.1. Ключ *DefaultDescDir*

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

13.1.3.2. Ключи *file-list-section*

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

13.1.3.3. 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. 13.2). Кроме того, следует учитывать, что в целях совместимости с Windows 98/ME значение 65 535 приравнивается к −1.

Таблица 13.2. *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 9*x*/ME
12	Директория драйверов, т. е. %windir%\system32\drivers для NT-систем и %windir%\system\IoSubsys для Windows 9*x*/ME
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 не обязательно 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
16 406	All Users\Start Menu
16 407	All Users\Start Menu\Programs
16 408	All Users\Start Menu\Programs\Startup

Таблица 13.2 (окончание)

Значение	Описание
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

13.1.3.4. Ключ *subdir*

Ключ `subdir` обозначает директорию относительно `dirid`, например:

```
; 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!

13.1.4. Секция описания модели

Секции описания моделей должны соответствовать ссылкам на модели из секции `Manufacturer` (*см. разд. 13.4.3*). Формат строк этой секции имеет вид:

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

- `device-description` — это любая уникальная строка описания или ссылка на строку в секции `String`;
- `install-section-name` содержит имя секции, описывающей устройство;
- `hw-id` содержит строку, соответствующую `Hardware ID` в идентификаторе PnP и может иметь один из форматов, перечисленных далее:
  - `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 Microsoft-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.

Пусть, например, 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
```

13.1.5. Секция *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. 13.4.5*).

Формат строк этой секции имеет вид:

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

- 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;
- subkey — содержит имя ветки реестра относительно reg-root или ссылку на строку секции String;

- value-entry-name — содержит имя ключа, создаваемого в реестре в ветке reg-root\subkey. Может быть или строкой, заключенной в кавычки, или ссылкой на значение из секции String;
- flags — необязательное, обозначает тип и свойства добавляемого значения. Значение представляет собой маску из констант FLG_xxx (табл. 13.3). Мы приведем самые существенные из этих констант;
- value содержит значение ключа реестра reg-root\subkey\value-enntry-name и должно соответствовать типу, указанному флагами flags.

Таблица 13.3. *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-битного 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
$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)

```
; 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,"NextI
D",3,01,00,00,00
HKLM,Software\Microsoft\Windows\CurrentVersion\RunOnce,TapiSetup2,,"tapiu
pr.exe"
```

13.1.6. Секция *xxx.LogConfig*

Секция `xxx.LogConfig` описывает системные ресурсы, требуемые драйвером. Ссылка на имя этой секции дается из секции описания модели (*см. разд. 13.4.5*).

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.

13.1.7. 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. 13.4.5*). 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 при операции копирования файлов имена исходного и конечного файлов одинаковы, 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 9x, 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. 13.4).

Таблица 13.4. *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

Таблица 13.4 *(окончание)*

Флаги секции CopyFiles	Описание флагов
$00000002 COPYFLG_NOSKIP	Не позволять пользователю отказаться от копирования файла

13.1.8. Секция *Strings*

Секция Strings используется для создания INF-файлов на нескольких языках. Другие секции могут ссылаться на ключи этой секции с помощью символов "%". Строка, заключенная между символами "%", 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)
```

13.1.9. 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. 13.1).

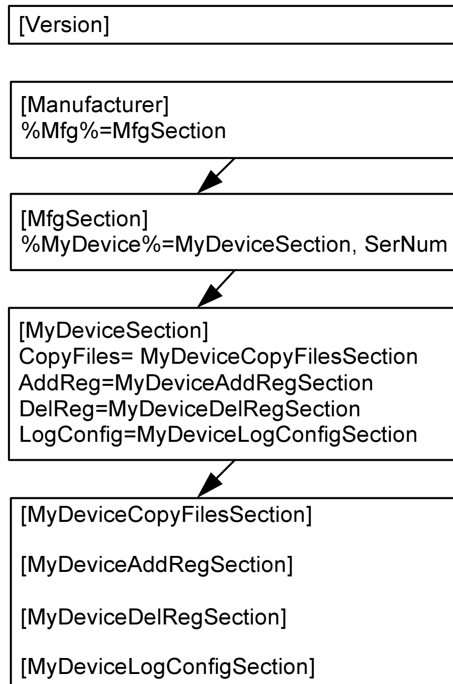


Рис. 13.1. Связи основных секций INF-файла

13.2. Создание и тестирование INF-файлов

В комплект Windows DDK входит довольно удобная утилита для создания INF-файлов, которая называется GetInf (рис. 13.2). Найти ее можно в каталоге %DDK%\tools\geninf\x86\ в Windows XP DDK или %DDK%\tools\ в Windows 2000 DDK.

Последовательно отвечая на вопросы утилиты (рис. 13.3), довольно легко получить вполне приемлемый вариант INF-файла.

Для проверки правильности написания INF-файла в Windows DDK предусмотрена утилита ChkInf. Для ее работы требуется установка интерпретатора ActivePerl, загрузить который можно с сайта **http://www.activestate.com**. Утилита выдает результат в формате HTML.

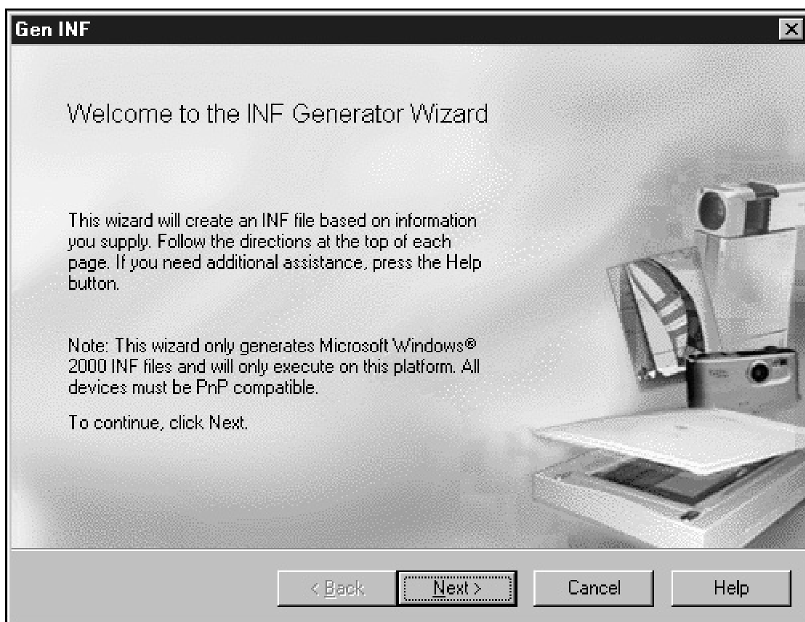


Рис. 13.2. Утилита генерации INF-файлов (Windows XP DDK)

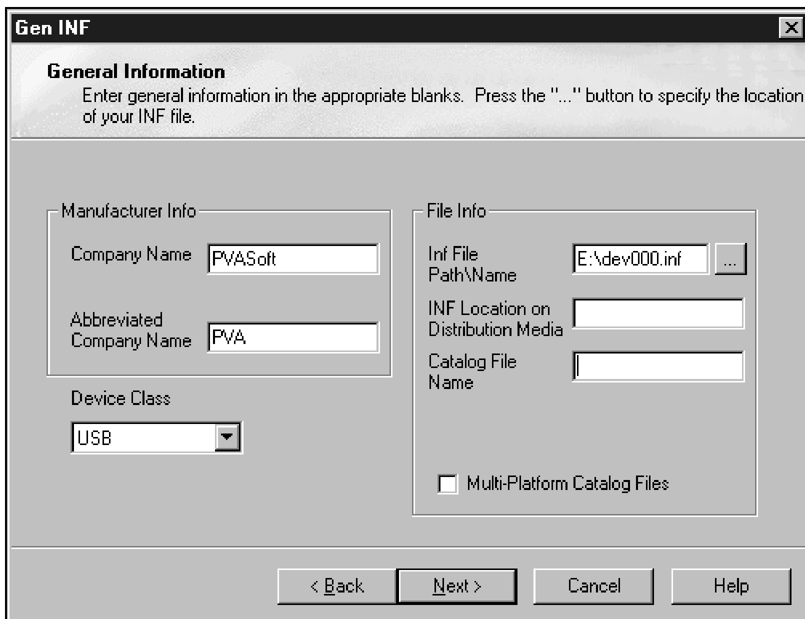


Рис. 13.3. Один из диалогов помощника создания INF-файла

13.3. Установка устройств с помощью INF-файла

Мы уже рассматривали процедуру установки USB-устройств (*см. разд. 9.3 и 10.5*), в этом разделе мы опишем общие этапы этого процесса. Итак, обнаружив подключение нового устройства, диспетчер PnP выполняет следующие шаги:

- PnP-менеджер режима ядра уведомляет PnP-менеджер пользовательского режима об обнаружении нового устройства со специфическими идентификаторами PnP (код производителя, модель, версия и т. д.);
- PnP-менеджер пользовательского режима составляет список возможно подходящих драйверов, проверяя, в частности, системный каталог с доступными INF-файлами;
- если подходящий INF-файл не обнаружен, система откладывает все последующие действия до момента, пока в систему войдет пользователь с достаточным уровнем привилегий. Этому пользователю предлагается диалог Мастера Установки (Add Hardware Wizard). Пользователь должен указать месторасположение подходящих INF-файлов;
- при обнаружении подходящего INF-файла производится его обработка: выполняется копирование указанных файлов драйвера, модификация реестра и т. д.;
- на основе директив INF-файла PnP-менеджер режима ядра загружает все фильтр-драйверы нижнего уровня, затем функциональный драйвер и, наконец, верхние фильтр-драйверы, предназначенные для обслуживания нового устройства. Драйверу, находящемуся на вершине стека, направляются PnP-запросы (пакеты IRP с кодом `IRP_MJ_PNP`), включая `IRP_MN_START_DEVICE`.

13.4. Ветки реестра для USB

Рассмотрим следующую ветку реестра (рис. 13.4):

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\PCI
```

Она содержит ветку имен USB-контроллеров, например:

```
VEN_1002&DEV_4742&SUBSYS_00000000&REV_5C
```

Каждая из этих веток имен содержит ветку, имя которой похоже на такое: `3&225b1d41&0&0008`

Наиболее интересны следующие ключи этой ветки:

- `DeviceDesc` — описание контроллера;
- `HardwareID` — полное аппаратное имя;
- `Mfg` — имя производителя.

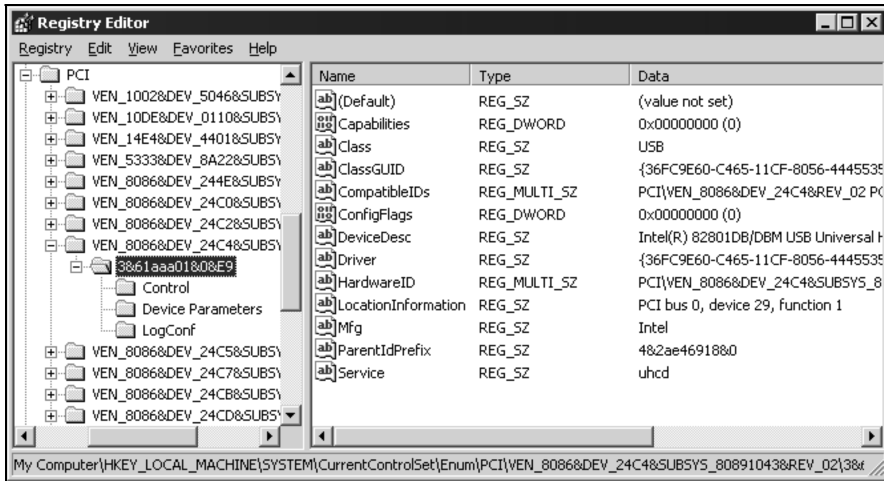


Рис. 13.4. Ветка реестра USB-котроллеров

Следующая ветка реестра содержит ключи, описывающие USB-устройства, когда-либо присутствующие в системе:

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB`

Отметим, что к этим устройствам относятся также корневые хабы (рис. 13.5), но не относятся устройства, имеющие другой идентификатор класса (например, для класса `USBSTOR` создается другая ветка реестра).

Ветка `Device Parameters` содержит ключ `SymbolicName`, значение которого равно символьному имени (*см. разд. 1.3.4*) соответствующего устройства.

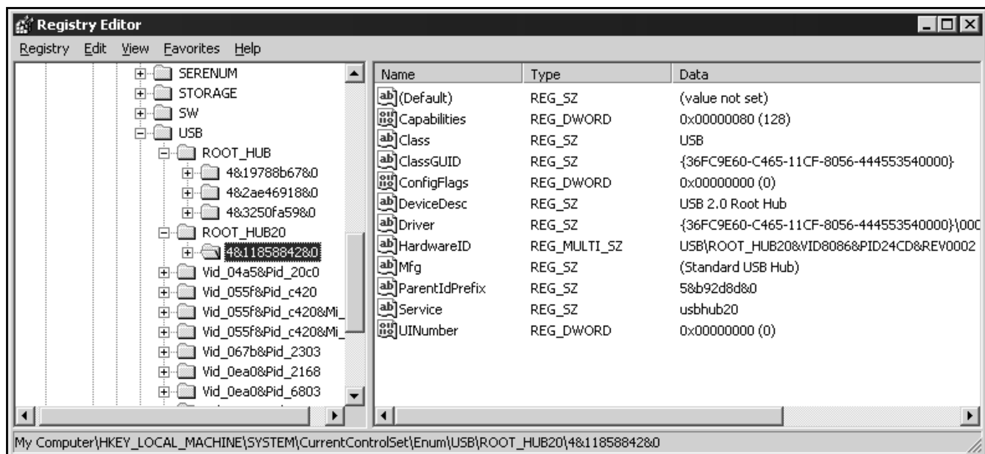
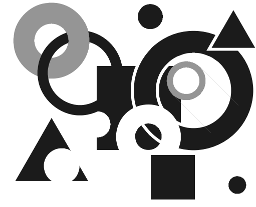


Рис. 13.5. Ветка реестра корневых хабов



Глава 14

Базовые функции Windows

14.1. Функции *CreateFile* и *CloseHandle*: открытие и закрытие объекта

Функция CreateFile открывает объект, а функция CloseHandle — закрывает. Объектом может являться файл, драйвер, порт, устройство и т. д. Объект может быть открыт в режиме разделения или эксклюзивно. В этом случае попытка открыть ресурс еще раз завершиться с ошибкой. 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 объекта
 DWORD   dwDesiredAccess,        // способ доступа
 DWORD   dwShareMode,            // тип совместного доступа
 LP_SECURITY_ATTRIBUTES lpSA,    // атрибуты защиты
 DWORD   dwCreationDisposition,  // параметры создания
 DWORD   dwFlagsAndAttributes,   // атрибуты
 HANDLE   hTemplateFile           // дескриптор template-файла
);

BOOL CloseHandle(
 HANDLE  hObject                // дескриптор объекта
);
```

Формат заголовков CreateFile и CloseHandle на языке Delphi имеет следующий вид:

```
function CreateFile(
 lpFileName: PChar;            // имя объекта
```

```
 dwDesiredAccess,             // способ доступа
 dwShareMode: DWORD;          // тип совместного доступа
 lpSA : PSecurityAttributes;  // атрибуты защиты
 dwCreationDisposition,       // параметры создания
 dwFlagsAndAttributes: DWORD; // атрибуты
 hTemplateFile: Thandle       // дескриптор template-файла
): THandle;
function CloseHandle(
 hObject: THandle             // дескриптор объекта
): BOOL;
```

Для языка C# используется импорт этих функций из библиотеки `kernel32`:

```
[DllImport("kernel32", SetLastError=true)]
static extern unsafe IntPtr CreateFile(
 string FileName,                  // имя объекта
 uint DesiredAccess,               // способ доступа
 uint ShareMode,                   // тип совместного доступа
 uint SecurityAttributes,          // атрибуты защиты
 uint CreationDisposition,         // параметры создания
 uint FlagsAndAttributes,          // атрибуты
 int hTemplateFile                 // дескриптор template-файла
);
[DllImport("kernel32", SetLastError=true)]
static extern unsafe bool CloseHandle(
 IntPtr hObject   // handle to object
);
```

14.1.1. Дополнительные сведения

Для нотации языка C строка, передаваемая в функцию `CreateFile`, должна иметь дублированные символы "\", например, `\\.\COM1` должна выглядеть как `\\\\.\\COM1`.

В языке C# специальный символ "@" перед строкой позволяет отменить стандартную нотацию языка C и использовать имя в его обычном виде.

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 может использо-

ваться простое имя "COM1"—"COM9" без префиксов, однако для COM10 и более 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 значение `FILE_ATTRIBUTE_NORMAL or FILE_FLAG_OVERLAPPED` для асинхронного доступа к порту;
- последний параметр обязательно должен иметь значение `nil`.

Для виртуальных портов, создаваемых CDC-устройством, обычно используются номера портов больше 4.

14.1.2. 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 дескриптора выполнено успешно, и возвращает `0`, если произошла ошибка. Расширенную информацию об ошибке можно получить с помощью вызова `GetLastError`.

14.1.3. Пример вызова

В листинге 14.1 приведена структура программы, использующей функции `CreateFile`/`CloseHandle` для доступа к коммуникационному порту COM1.

Листинг 14.1. Использование функций `CreateFile` и `CloseHandle`

[Delphi]

```
{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);
```

[C++]

```
// nPort - номер открываемого порта
CString sPort;
sPort.Format(_T("\\\\.\\COM%d"), nPort);
HANDLE hComm = CreateFile(
  sPort,
  GENERIC_READ | GENERIC_WRITE,
  0,
  NULL,
  OPEN_EXISTING,
```

```
  FILE_FLAG_OVERLAPPED,
  NULL
);
if (hComm == INVALID_HANDLE_VALUE)
{
  AfxThrowSerialException();
}
// Закрытие порта
CloseHandle(hComm);
```

14.2. Функция *ReadFile*: чтение данных

Функция `ReadFile` производит синхронное или асинхронное чтение данных. 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
  nNBTR: DWORD;                  // число байт для чтения
  var nNBR: DWORD;               // реально прочитанное число байт
  lpOverlapped: Poverlapped      // параметры асинхронного чтения
): BOOL;
```

Для языка C# используется импорт этой функции из библиотеки kernel:

```
[DllImport("kernel32", SetLastError=true)]
static extern unsafe bool ReadFile(
 IntPtr hFile,                 // дескриптор, полученный от CreateFile
 void* pBuffer,                // буфер для чтения
 int nNBTR,                    // число байт для чтения
```

```
 int* nNBR,                   // реально прочитанное число байт
 int Overlapped               // параметры асинхронного чтения
);
```

Для асинхронных операций последний параметр передает структуру `OVERLAPPED`[1]:

```
[DllImport("kernel32", SetLastError=true)]
private static extern bool ReadFile(
  int hFile,
  byte[] lpBuffer,
  int nNumberOfBytesToRead,
  ref int lpNumberOfBytesRead,
  ref OVERLAPPED lpOverlapped
);
```

Первый параметр передает дескриптор объекта, полученный с помощью функции `CreateFile`. Указатель на буфер для чтения данных задается с помощью 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 байтов, прочитанных в асинхронном режиме, может использоваться функция `GetOverlappedResult`.

Последний параметр передает настройки для асинхронного чтения данных. Если объект был открыт с параметром `FILE_FLAG_OVERLAPPED`, этот параметр обязательно должен указывать на правильную структуру типа `OVERLAPPED`, если же объект был открыт без использования `FILE_FLAG_OVERLAPPED`, то этот указатель обязательно должен быть `NULL`.

14.2.1. Дополнительные сведения

В Windows NT/2000/XP можно использовать функцию `ReadFileEx` для асинхронного чтения данных (*см. разд. 14.4*).

Для платформы .NET следует использовать опцию компилятора `unsafe`, т. к. эта функция требует использования указателя как параметра.

---

[1] Мы приводим описание структуры для языка C#. Для языков C и Delphi описание этих структур находится в заголовочном файле windows.

Структура `OVERLAPPED` описывается в C# следующим образом:

```
[StructLayout(LayoutKind.Sequential)]
private struct OVERLAPPED {
  public int  Internal;
  public int  InternalHigh;
  public int  Offset;
  public int  OffsetHigh;
  public int hEvent;
}
```

14.2.2. 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`.

14.2.3. Пример вызова

Листинг 14.2 показывает пример использования функции `ReadFile` для синхронного чтения данных.

Листинг 14.2. 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 доступный 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;
```

14.3. Функция *WriteFile*: передача данных

Функция `WriteFile` производит синхронную или асинхронную запись данных в файл (порт, драйвер). Формат заголовка `WriteFile` на языке С имеет следующий вид:

```
BOOL WriteFile(
  HANDLE       hHandle,     // дескриптор, полученный от CreateFile
  LPCVOID      Buffer,      // буфер данных
  DWORD        nNBTW,       // длина буфера
  LPDWORD      nNBW,        // реально отправленное число байт
  LPOVERLAPPED Overlapped   // параметры асинхронной записи
);
```

Формат заголовка `WriteFile` на языке Delphi имеет следующий вид:

```
function WriteFile(
  hHandle: THandle;         // дескриптор, полученный от CreateFile
  const  Buffer;            // буфер данных
  nNBTW      : DWORD;       // длина буфера
  var lpNBW  : DWORD;       // реально отправленное число байт
  Overlapped : POverlapped  // параметры асинхронной записи
): BOOL;
```

Для языка C# используется импорт этой функции из библиотеки kernel:

```
[DllImport("kernel32", SetLastError=true)]
static extern bool WriteFile(
```

```
  int hHandle,
  byte[] lpBuffer,
  int nNBTW,
  ref int lpNBW,
  int Overlapped  // равно 0 для синхронных операций
);
```

Для асинхронных операций последний параметр передает структуру `OVERLAPPED` (*см. разд. 14.2*):

```
[DllImport("kernel32", SetLastError=true)]
static extern bool WriteFile(
  int hHandle,
  byte[] lpBuffer,
  int nNBTW,
  ref int lpNBW,
  ref OVERLAPPED Overlapped
);
```

Первый параметр передает дескриптор объекта, 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.

14.3.1. Дополнительные сведения

В Windows NT/2000/XP можно использовать функцию `WriteFileEx` для асинхронной записи данных (*см. разд. 14.5*).

14.3.2. 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`.

14.3.3. Пример вызова

Листинг 14.3 показывает пример использования функции `WriteFile` для синхронной записи данных в коммуникационный порт, а листинг 14.4 — пример асинхронной записи.

Листинг 14.3. 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 14.4. 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);
```

14.4. Функция *ReadFileEx*: APC-чтение данных[2]

Функция `ReadFileEx` осуществляет асинхронное чтение данных. Работа этой функции похожа на вызов обычной функции `ReadFile` в режиме асинхронного 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 объекта
  LPVOID     lpBuffer,        // буфер для данных
  DWORD      dwNBTR,          // число байт для чтения
  LPOVERLAPPED lpOverlapped, // параметры асинхронного чтения
  // callback-процедура, выполняемая по завершению чтения
  LP_OVERLAPPED_COMPLETION_ROUTINE lpComletionRoutine
);
```

Формат заголовка `ReadFileEx` на языке Delphi имеет следующий вид:

```
function ReadFileEx(
  hFile                : THandle;     // дескриптор объекта
  lpBuffer             : Pointer;     // буфер для данных
  dwNBTR               : DWORD;       // число байт для чтения
  lpOverlapped         : POverlapped; // параметры асинхронного чтения
```

[2] APC (Asynchronous Procedure Calls) — асинхронный вызов процедур.

```
  // callback-процедура, выполняемая по завершению 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 байтов
 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 14.5.

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.

14.4.1. Возвращаемое значение

При успешном завершении функция `ReadFileEx` возвращает ненулевое значение, а при ошибочном — ноль, при этом код ошибки можно получить с помощью вызова `GetLastError`.

14.4.2. Дополнительные сведения

Для завершения всех асинхронных операций может использоваться функция `CancelIo`.

Функция `ReadFileEx` игнорирует параметр `hEvent` в структуре `lpOverlapped` и он может использоваться программой.

14.4.3. Пример вызова

Листинг 14.5 показывает пример использования `ReadFileEx`. Обратите внимание, что процедура, вызываемая при завершении операции, должна иметь спецификатор `stdcall`.

Листинг 14.5. Пример использования функции ReadFileEx

```
Procedure TReadThread.Execute;
Var ReadOL : TOverLapped; {структура для асинхронного чтения}

  {Callback-процедура, вызываемая 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;
```

14.5. 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 С имеет следующий вид:

```
BOOL WriteFileEx(
  HANDLE     hFile,            // дескриптор объекта
  LPCVOID    lpBuffer,         // буфер для данных
  DWORD      dwBufLen,         // число байтов для записи
  LPOVERLAPPED lpOverlapped, // параметры асинхронной записи
  // callback-процедура, выполняемая по завершению записи
  LP_OVERLAPPED_COMPLETION_ROUTINE lpComletionRoutine
);
```

Формат заголовка `WriteFileEx` на языке Delphi имеет следующий вид:

```
function WriteFileEx(
  hFile    : THandle;                // дескриптор объекта
  lpBuffer : Pointer;                // буфер для данных
  nNumberOfBytesToWrite: DWORD;      // число байтов для записи
  const lpOverlapped : TOverlapped; // параметры асинхронной записи
  lpCompletionRoutine: FARPROC
): BOOL;
```

Все параметры этой функции совпадают с параметрами функций `WriteFile` и `ReadFileEx`.

14.5.1. Возвращаемое значение

При успешном завершении функция `WriteFileEx` возвращает ненулевое значение, а при ошибочном — ноль. Код ошибки можно получить с помощью вызова `GetLastError`.

14.5.2. Пример вызова

Листинг 14.6 показывает пример использования `WriteFileEx`. Обратите внимание на спецификатор `stdcall` функции обратного вызова `OnCompletionWrite`.

Листинг 14.6. 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;
```

14.6. Функция *WaitForSingleObject*: ожидание сигнального состояния объекта

Функция `WaitForSingleObject` ожидает сигнального состояния объекта синхронизации. Функция завершается в двух случаях:

- объект перешел в сигнальное состояние;
- по завершению 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. Пауза ожидания `dwMilliseconds` задается в миллисекундах. Специальная константа `INFINITE` задает неограниченное время ожидания.

При необходимости ожидания сигнального состояния одного из нескольких объектов или всех одновременно следует воспользоваться функцией `WaitForMultipleObjects`, а не объединять `WaitForSingleObject` с помощью `OR` или `AND`.

14.6.1. Возвращаемое значение

При успешном выполнении функция `WaitForSingleObject` возвращает значение, указывающее на состояние объекта. При ошибке возвращает `WAIT_FAILED`. В табл. 14.1 приводятся возможные результаты функции.

Таблица 14.1. *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`

14.7. 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 завершению 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.

14.7.1. Возвращаемое значение

При успешном выполнении функция `WaitForMultipleObjects` возвращает значение, указывающее на состояние объектов. При ошибке возвращает `WAIT_FAILED`. В табл. 14.2 приводятся возможные результаты функции.

Таблица 14.2. *Результат 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`
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

Таблица 14.2 *(окончание)*

Код	**Описание**
`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`

14.8. 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 вызовом `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.

14.8.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

14.9. 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,     // дескрипторов 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; // дескрипторов 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.

14.9.1. Возвращаемое значение

При успешном выполнении функция DeviceIoControl возвращает ненулевое значение. В случае ошибки вернет ноль, а вызов GetLastError вернет код ошибки.

14.10. Функция *CancelIo*: прерывание операции

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;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("kernel32.dll")]
 static public extern int CancelIo(int hFile);
```

Параметр hFile задает дескриптор порта, полученный при открытии в функции CreateFile.

14.10.1. Возвращаемое значение

Ненулевое значение, если выполнение успешно.

14.11. 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*).

14.11.1. Возвращаемое значение

При успешном выполнении функция QueryDosDevice возвращает число символов, сохраненных в буфере. В случае ошибки вернет ноль, а вызов GetLastError вернет код ошибки.

Если буфер имеет недостаточный размер, функция вернет ноль, 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.

14.11.2. Пример вызова

Листинг 14.7 показывает получение списка всех имен в системе с помощью вызова `QueryDosDevice`.

Листинг 14.7. 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;
```

14.12. 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. 14.3.

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 (см. табл. 14.3). Второй параметр передает DOS-имя устройства, а третий — внутреннее имя устройства. Для выполнения этой функции требуются права администратора системы.

14.12.1. Возвращаемое значение

При успешном выполнении функция `DefineDosDevice` возвращает ненулевое значение. В случае ошибки функция вернет ноль, а вызов `GetLastError` вернет код ошибки.

14.12.2. Пример вызова

Листинг 14.8 показывает добавление имени в систему.

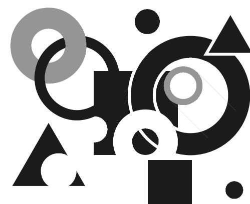
Листинг 14.8. Пример вызова `DefineDosDevice`

```
{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 14.3. *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 15

Структуры и функции Windows для последовательных портов

15.1. 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;
```

[1] Многие приведенные здесь структуры могут использоваться также для конфигурирования параллельных портов и модемов, но мы будем рассматривать их использование только для COM-портов.

```
  dcb: TDCB;
  dwProviderSubType: DWORD;
  dwProviderOffset: DWORD;
  dwProviderSize: DWORD;
  wcProviderData: array[0..0] of WCHAR;
end;
TCommConfig = _COMMCONFIG;
```

Структура содержит следующие поля:

- `dwSize`. Должно быть равно размеру структуры;
- `wVersion`. Должно равняться единице;
- `wReserved`. Не используется;
- `dcb`. Задает описание параметров COM-порта в структуре `DCB` (*см. разд. 15.5*);
- `dwProviderSubType`. Должно быть равно `PST_RS232` (значение `1`). Полный список возможных значений приводится в табл. 15.2.
- `dwProviderOffser`. Смещение, в байтах, до устройство-зависимого блока информации `wcProviderData` от начала структуры;
- `dwProviderSize`. Размер, в байтах, блока `wcProviderData`;
- `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` равно `PST_MODEM`, то данное поле содержит структуру `MODEMSETTINGS`.

Для установки параметров порта используется функция `SetCommConfig`. Так как ручное заполнение полей структуры может быть затруднительно, рекомендуется предварительно считать текущее значение полей с помощью вызова `GetCommConfig`, 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 15.1).

Листинг 15.1. 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;
```

15.2. Структура свойств порта *COMMPROP*

Структура `COMMPROP` используется функцией `GetCommProperties` для получения информации о коммуникационном драйвере. Описание структуры на языке С 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* (output buffer) в байтах. Если 0, то ограничение не используется. Для обычного режима COM-портов вернет 0;
- dwMaxRxQueue. Максимальный размер внутреннего *входного буфера* (input buffer) в байтах. Если 0, то ограничение не используется. Для обычного режима COM-порта вернет 0;
- dwMaxBaud. Максимально доступное значение скорости порта в бодах. Это поле может принимать одно из значений, приведенных в табл. 15.1. 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 из табл. 15.2;
- dwProvCapabilities. Определяет параметры совместимости порта. Возможные значения этого поля приведены в табл. 15.3;
- dwSettableParams. Определяет параметры, которые могут быть изменены. Возможные значения этого поля приведены в табл. 15.4. Обычный COM-порт возвращает значение маски 127 ($7F);
- dwSettableBaud. Определяет маску допустимых скоростей обмена. Маска состоит из тех же констант, которые использовались в dwMaxBaud. Обычно COM-порт возвращает значение маски 0x1006FFFF;
- wSettableData. Определяет маску допустимых форматов байт, приведенных в табл. 15.5. Обычный COM-порт возвращает значение маски 0x0F;
- wSettableStopParity. Определяет допустимые значения для числа стоп-бит и четности согласно битовой маске из табл. 15.6. Обычный COM-порт возвращает значение маски $1F07;
- dwCurrentTxQueue. Размер внутреннего выходного буфера. Значение 0 означает, что эта величина недоступна;
- dwCurrentRxQueue. Размер внутреннего входного буфера. Значение 0 означает, что эта величина недоступна;
- dwProvSpec1, dwProvSpec2, wcProvChar. Величины, зависящие от драйвера. Программы должны игнорировать эти поля.

Таблица 15.1. *Константы допустимых скоростей в COMMPROP*

Значение	Скорость	Величина
BAUD_075	75 бод	$00000001
BAUD_110	110 бод	$00000002
BAUD_134_5	134,5 бод	$00000004

Таблица 15.1 *(окончание)*

Значение	Скорость	Величина
BAUD_150	150 бод	$00000008
BAUD_300	300 бод	$00000010
BAUD_600	600 бод	$00000020
BAUD_1200	1200 бод	$00000040
BAUD_1800	1800 бод	$00000080
BAUD_2400	2400 бод	$00000100
BAUD_4800	4800 бод	$00000200
BAUD_7200	7200 бод	$00000400
BAUD_9600	9600 бод	$00000800
BAUD_14400	14 400 бод	$00001000
BAUD_19200	19 200 бод	$00002000
BAUD_38400	38 400 бод	$00004000
BAUD_56K	56 000 бод	$00008000
BAUD_57600	57 600 бод	$00040000
BAUD_115200	115 200 бод	$00020000
BAUD_128K	128 000 бод	$00010000
BAUD_USER	Устанавливается программно	$10000000

Таблица 15.2. *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

Таблица 15.2 *(окончание)*

Значение	**Режим**	**Величина**
PST_RS449	Порт RS-449	$005
PST_SCANNER	Сканер	$022
PST_TCPIP_TELNET	Протокол TCP/IP Telnet	$102
PST_UNSPECIFIED	Не определено	$000
PST_X25	Стандарт X.25	$103

Таблица 15.3. *Параметры совместимости порта*

Значение	**Тип**	**Маска**
PCF_16BITMODE	Поддерживается специальный 16-битный режим	$200
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 15.4. *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

Таблица 15.4 (окончание)

Значение	Параметр	Маска
SP_RLSD	Сигнал RLSD	$40
SP_STOPBITS	Число стоп-бит	$08

Таблица 15.5. *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
DATABITS_8	8 бит данных	$08
DATABITS_16	16 бит данных	$10
DATABITS_16X	Специальное аппаратное расширение	$20

Таблица 15.6. *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

Так как поле wcProvChar имеет переменную длину и может содержать или не содержать данные, то для выделения нужного блока памяти необходимо выполнить следующие шаги (листинг 15.2):

1) выделить память под структуру COMMPROP;

2) запросить информацию у системы, вызвав функцию GetCommProperties;

3) если поле wPacketLength содержит значение большее 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 15.2. 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;
```

15.3. 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 функций `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;
```

Описание структуры на языке C# имеет вид:

```
[StructLayout(LayoutKind.Sequential)]
protected internal struct COMMTIMEOUTS
{
  internal UInt32 readIntervalTimeout;
  internal UInt32 readTotalTimeoutMultiplier;
  internal UInt32 readTotalTimeoutConstant;
  internal UInt32 writeTotalTimeoutMultiplier;
  internal UInt32 writeTotalTimeoutConstant;
}
```

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 этого 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 из линии;
- `ReadTotalTimeoutMultiplier`. Задает множитель, в миллисекундах, используемый для вычисления полного времени ожидания операции чтения. Для каждой операции чтения это значение умножается на количество запрошенных для чтения байтов;
- `ReadTotalTimeoutConstant`. Задает константу, добавляемую к произведению, полученному при вычислении времени из `ReadTotalTimeoutMultiplier`;
- `WriteTotalTimeoutMultiplier` и `WriteTotalTimeoutConstant`. То же, что `ReadTotalTimeoutMultiplier` и `ReadTotalTimeoutConstant`, но для передачи. Задание всех `ReadXXX` констант в `0` указывает на отсутствие тайм-аутов чтения, а всех `WriteXXX` — отключение тайм-аутов для записи.

Пусть, например, заданы такие тайм-ауты:

- `ReadTotalTimeoutMultiplier — 2`;
- `ReadTotalTimeoutConstant — 1`;
- `ReadIntervalTimeout — 1`;
- считывается 250 символов.

Если операция чтения завершится за 501 мс ($250 \times 2 + 1$), 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 мо-

гут быть получены следующей операцией чтения. Если между началами двух последовательных символов пройдет более 1 мс, то операция чтения так же будет завершена.

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 15.3). С помощью вызова BuildCommDCBAndTimeouts можно заполнить структуру COMMTIMEOUTS из строки, формат которой совпадает с форматом команды Mode.

Листинг 15.3. Пример использования структуры COMMTIMEOUTS

```
var
  CommTimeOut : TCommTimeouts;
FillChar(CommTimeOut, SizeOf(TCommTimeOuts), 0);
CommTimeOut.ReadIntervalTimeout := MAXDWORD;
SetCommTimeOuts(SaveHandle, CommTimeOut);
```

15.4. Структура статуса порта *COMSTAT*

Структура COMSTAT содержит информацию о коммуникационном устройстве. Эта структура заполняется с помощью вызова ClearCommError. Описание структуры на языке С 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 имеет значение `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 15.4).

Листинг 15.4. Пример использования структуры COMSTAT

```
// Возвращает число байтов 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;
```

15.5. Структура *DCB*

Структура `DCB` (Device Control Block) содержит управляющие настройки коммуникационного устройства. Описание структуры на языке С 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. 15.6*);
- `BaudRate`. Задает скорость обмена. Может задаваться либо числом (в бод), либо одной из констант табл. 15.7;

- fBinary. Должно быть TRUE, т. к. Windows не поддерживает небинарный метод доступа;
- fParity. Включает режим контроля четности. Если поле равно 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 равно 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. Данный режим используется, например, модемами при восстановлении потерянной связи. Вызов EscapeCommFunction в этом режиме вызовет ошибку;
- fDsrSensitivity. Включает восприятие сигнала DSR. Если это поле равно 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 принимаемых данных. Таким образом, это поле вводит некую зависимость между управлением входным и выходным потоками информации;
- fOutX. Включает режим XON/XOFF-передачи. Если это поле равно TRUE, передача останавливается по получении символа XoffChar и возобновляется при получении символа XonChar;
- fInX. Включает режим XON/XOFF-приема. Если это поле равно 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 поля fErrorChar и fParity имеют значение TRUE, то ошибочные символы будут заменены символом ErrorChar;
- fNull. Включает игнорирование нулевых байтов при приеме. Если это поле равно 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 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 EscapeCommFunction вызовет ошибку. Этот режим не работает для 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 байт в приемном буфере перед посылкой символа XOFF. Максимально допустимое количество байт в буфере вычисляется вычитанием данного значения из размера приемного буфера в байтах;
- ByteSize. Задает число битов в байте (5—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 `fErrorChar` равном `TRUE`;

❒ `EofChar`. Задает символ конца данных;

❒ `EvtChar`. Задает символ для вызова *события* (event);

❒ `wReserved1`. Не используется.

Таблица 15.7. *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`	`14400`	14 400
`CBR_19200`	`19200`	19 200
`CBR_38400`	`38400`	38 400
`CBR_5600`	`56000`	56 000
`CBR_57600`	`57600`	57 600
`CBR_115200`	`115200`	$1C200
`CBR_128000`	`128000`	$1F400
`CBR_256000`	`256000`	$3E800

Важно отметить, что программист должен сам следить за совместимостью параметров блока `DCB`. Например:

- `ByteSize` может задаваться только от 5 до 8;
- комбинация `ByteSize`, равное `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 `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 15.5).

Листинг 15.5. 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 15.6.

Листинг 15.6. 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;
```

15.6. 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;
```

Формат заголовка `BuildCommDCB` на языке C# имеет следующий вид:

```
[DllImport("kernel32.dll")]
private static extern bool BuildCommDCB(
  string lpDef,    // строка
  ref DCB lpDCB    // указатель на блок параметров DCB
);
```

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:

- при задании скорости обмена 110 бит/с автоматически устанавливается формат обмена с двумя стоповыми битами. Это сделано для совместимости с командой mode из MS-DOS или Windows NT;
- 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 существуют 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 в 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 запол-

нения структуры DCB, затем изменить значения нужных полей, и после этого вызывать функцию конфигурирования порта.

15.6.1. 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.

15.6.2. Возвращаемое значение

Если функция выполнена успешно, возвращается ненулевое значение, иначе нулевое. Код ошибки можно получить с помощью вызова GetLastError.

15.6.3. Пример вызова

Листинг 15.7 показывает один из способов простого заполнения полей структуры DCB с помощью вызова BuildCommDCB.

Листинг 15.7. 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;
```

15.7. 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`.

15.8. Функции *SetCommBreak* и *ClearCommBreak*: управление выводом данных

Функция `SetCommBreak` замораживает передачу данных для указанного порта до тех пор, пока не будет вызвана `ClearCommBreak`.

Формат заголовков этих функций на языке С 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`.

15.8.1. Возвращаемое значение

Если выполнение успешно, возвращается ненулевое значение.

15.9. Функция *ClearCommError*: получение и сброс ошибок порта

Функция `ClearCommError` получает информацию об ошибках порта и сбрасывает флаги ошибок.

Формат заголовка `ClearCommError` на языке С 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. 15.8 (для полноты приведены все коды ошибок).

Таблица 15.8. *Коды ошибок для функции* `ClearCommError`

Маска	Значение	Величина
`CE_BREAK`	Обрыв линии (break)	$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 не верно	$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
`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 байт во входной очереди порта. Эта величина необходима перед вызовом функции `ReadFile`, т. к. перед чтением данных необходимо распределение памяти для буфера (см. листинг 14.2).

15.9.1. Возвращаемое значение

Если выполнение функции успешно, возвращает ненулевое значение.

15.10. 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;
```

Выполняет для порта с дескриптором `hFile` функцию, указанную в параметре `dwFunc`. Указание на выполнение функции передается напрямую драйверу порта. Допустимые значения параметра приведены в табл. 15.9.

15.10.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

Таблица 15.9. *Коды функций* `EscapeCommFunction`

Функция	**Описание**	**Значение**
`CLRDTR`	Сбрасывает сигнал DTR (data-terminal-ready)	6
`CLRRTS`	Сбрасывает сигнал RTS (request-to-send)	4
`SETDTR`	Устанавливает сигнал DTR	5
`SETRTS`	Устанавливает сигнал RTS	3
`SETXOFF`	Указывает передатчику выполнить операцию, как будто символ XOFF был принят	1
`SETXON`	Указывает передатчику выполнить операцию, как будто символ XON был принят	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

Таблица 15.9 (окончание)

Функция	Описание	Значение
`CLRBREAK`	Восстанавливает передачу данных, выполняя действия, аналогичные вызову `ClearCommBreak`	9
`RESETDEV`	Сброс устройства, если возможно	7
`GETCOMBASE`	Получить базовый адрес порта (недокументированно). Только в Windows 95/98/ME	10

15.11. Функции *GetCommMask* и *SetCommMask*: маска вызова событий

Функция `GetCommMask` позволяет получить флаги событий, а `SetCommMask` — задать эти флаги. Флаги задают типы событий, которые будут отслеживаться драйвером порта. Ожидание выбранных событий производится с помощью функции `WaitCommEvent`.

Формат заголовка 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 заголовка 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. 15.10.

15.11.1. Возвращаемое значение

Не ноль, если выполнение успешно.

Таблица 15.10. *Маски событий для* `GetCommMask/SetCommMask`

Значение	Описание
EV_BREAK	Прерывание (break) 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
EV_RXFLAG	Получен специальный символ, описанный в блоке DCB
EV_TXEMPTY	Буфер посылки опустошен

15.12. 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.

15.12.1. Возвращаемое значение

Если выполнение успешно, возвращается ненулевое значение.

15.12.2. 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;
```

15.12.3. Пример вызова

Листинг 15.8 показывает использование функции `WaitCommEvent` для организации потока чтения данных `TReadThread`. При инициализации устанавливается маска ожидания наличия данных `EV_RXCHAR`. Вызов `WaitCommEvent` стартует ожидание наступления этого события, а функция `WaitForSingleObject` позволяет "разбудить" поток в нужный момент времени. Функция `GetOverlappedResult` возвращает результат ожидания — маску произошедших событий (если бы в функции `SetCommMask` их было указано несколько).

Перед чтением данных производится вызов ClearCommError, позволяющий узнать об ошибках порта, и, самое главное — получить число символов, доступных для чтения. Это позволяет корректно распределить память для буфера данных функции ReadFile.

Листинг 15.8. 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;
```

15.13. Функции *GetCommConfig* и *SetCommConfig*: конфигурирование параметров порта

Функция `GetCommConfig` возвращает, а `SetCommConfig` — устанавливает текущую конфигурацию коммуникационного устройства. Конфигурация передается с помощью структуры `COMMCONFIG` (*см. разд. 15.1*).

Формат заголовка 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 заголовка 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.

15.13.1. Возвращаемое значение

Возвращает ненулевое значение, если вызов успешен.

15.13.2. 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 15.9).

Листинг 15.9. 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;
```

15.14. Функция *CommConfigDialog*: диалог конфигурирования порта

Функция `CommConfigDialog` отображает диалог конфигурирования параметров порта (рис. 15.1).

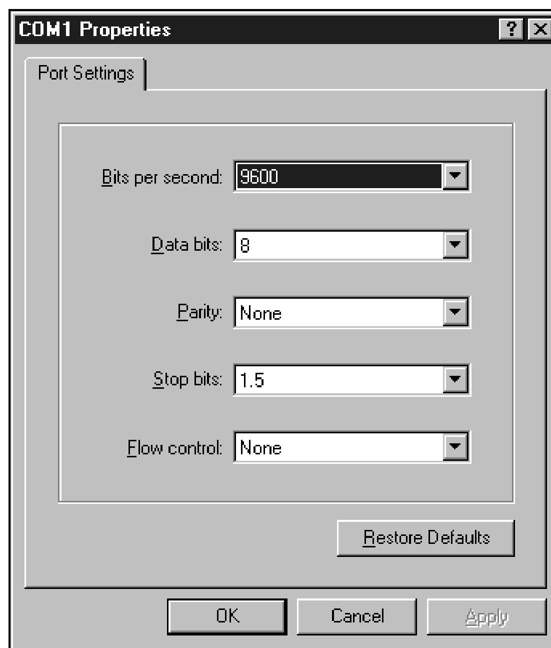


Рис. 15.1. Диалог редактирования настроек порта (`CommConfigDialog`)

Функция `CommConfigDialog` не выполняет настройки порта. Она всего лишь позволяет пользователю изменить некоторые поля в блоке `DCB`, содержащемся в структуре `COMMCONFIG`. Кнопка **Restore Defaults** (Восстановить исходные значения) позволяет установить настройки порта, заданные с помо-

щью 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`.

15.14.1. Возвращаемое значение

Возвращает ненулевое значение при успешном выполнении.

15.14.2. Дополнительные сведения

Для работы этой функции 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.

15.14.3. Пример вызова

```
Var
  comm : TCOMMCONFIG;
comm.dwSize:= sizeof(comm);
CommConfigDialog('COM1', 0, comm);
```

15.15. 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`.

15.15.1. Возвращаемое значение

Ненулевое значение, если функция выполнена успешно.

15.15.2. Пример вызова

См. листинг 15.2.

15.16. Функции *GetCommState* и *SetCommState*: состояние порта

Функция `GetCommState` (`SetCommState`) читает (устанавливает) текущие настройки порта согласно данным, записанным в структуре `DCB`.

Формат заголовка 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 заголовка на языке Delphi имеет вид:

```
function GetCommState(hFile: THandle; var   lpDCB: TDCB): BOOL;
function SetCommState(hFile: THandle; const lpDCB: TDCB): BOOL;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("kernel32.dll")]
private static extern bool GetCommState(
  int hFile,
  ref DCB lpDCB
);
[DllImport("kernel32.dll")]
private static extern bool SetCommState(
  int hFile,
  ref DCB lpDCB
);
```

Первый параметр передает дескриптор порта, полученный с помощью `CreateFile`. Второй параметр передает указатель на структуру `DCB`.

15.16.1. Возвращаемое значение

Возвращает ненулевое значение при успешном выполнении.

15.16.2. 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 15.10).

Листинг 15.10. 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;
```

15.17. Функции *GetCommTimeouts* и *SetCommTimeouts*: тайм-ауты порта

Функция `GetCommTimeouts` (`SetCommTimeouts`) читает (устанавливает) текущие настройки тайм-аутов согласно данным, записанным в структуре `COMMTIMEOUTS`.

Формат заголовка на языке С 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`.

15.17.1. Возвращаемое значение

Возвращает ненулевое значение при успешном выполнении.

15.17.2. Пример вызова

См. листинг 15.3.

15.18. 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  Flags // флаги сбрасываемых буферов
);
```

Формат заголовка на языке Delphi имеет вид:

```
function PurgeComm(hFile: THandle; Flags: DWORD): BOOL;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("kernel32.dll")]
private static extern Boolean PurgeComm(IntPtr hFile, uint flags);
```

Параметр `hFile` передает дескриптор порта, полученный с помощью `CreateFile`. Параметр `Flags` определяет маску операций согласно табл. 15.11.

Таблица 15.11. *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)

15.18.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

15.18.2. Пример вызова

Пример, показанный в листинге 15.11, 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 15.11. 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;
```

15.19. 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;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("kernel32.dll")]
internal static extern Boolean SetupComm(
          IntPtr hFile, UInt32 dwInQueue, UInt32 dwOutQueue);
```

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`.

15.19.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

15.20. Функции *GetDefaultCommConfig* и *SetDefaultCommConfig*: настройки порта по умолчанию

Функция `GetDefaultCommConfig` (`SetDefaultCommConfig`) читает (устанавливает) настройки порта по умолчанию. Эти настройки будут приняты при нажатии кнопки **Восстановить исходные значения** в диалоге, вызываемом функцией `CommConfigDialog` (*см. разд. 15.14*).

Формат заголовка 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 заголовка 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`.

15.20.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

15.21. 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;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("kernel32.dll")]
private static extern Boolean TransmitCommChar(
                              IntPtr hFile, Byte cChar);
```

Первый параметр передает дескриптор порта, полученный с помощью `CreateFile`. Второй параметр передает символ.

15.21.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

15.22. 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;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("kernel32.dll")]
private static extern Boolean GetCommModemStatus(
                       IntPtr hFile, out UInt32 pModemStat);
```

Первый параметр передает дескриптор порта, полученный с помощью `CreateFile`. Второй параметр возвращает статус модема, состоящий из масок, приведенных в табл. 15.12.

Таблица 15.12. *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

15.22.1. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение.

15.22.2. Пример вызова

Код, показанный в листинге 15.12, представляет собой функцию, возвращающую состояние всей линий модема. Вместо маски возвращается обычный для Delphi набор состояний.

Листинг 15.12. 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;
```

15.23. 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`, а иначе `PORT_INFO_2`. Эти структуры описаны 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 размера, 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 с символом ":", например, COM1:, а для виртуальных — без него, например, FILE.

15.23.1. Дополнительные сведения

Для использования этой функции в Delphi требуется подключить модуль WinSpool.

15.23.2. Возвращаемое значение

При успешном выполнении возвращает ненулевое значение. В случае ошибки вернет 0, а вызов GetLastError вернет код ошибки.

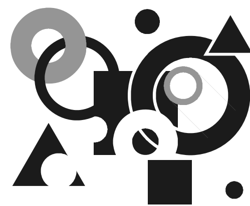
15.23.3. Пример вызова

Хотя в MSDN эта функция описывается для принтерных портов, она возвращает и коммуникационные порты тоже. Отличить одни порты от других можно, сравнив начало имени порта со строкой "COM" (листинг 15.13).

Листинг 15.13. 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;
```



Глава 16

Структуры и функции Windows Setup API

16.1. Функция *SetupDiGetClassDevs*: перечисление устройств

Функция `SetupDiGetClassDevs` возвращает информацию о списке устройств выбранного класса.

Формат заголовка на языке С имеет вид:

```
HDEVINFO SetupDiGetClassDevs(
  const GUID* ClassGuid,
  PCTSTR Enumerator,
  HWND hwndParent,
  DWORD Flags
);
```

Формат заголовка на языке Delphi имеет вид:

```
function SetupDiGetClassDevs(
  ClassGuid: PGUID;
  const Enumerator: PAnsiChar;
  hwndParent: HWND;
  Flags: DWORD
): HDEVINFO; stdcall;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("setupapi.dll", SetLastError=true)]
static extern  unsafe int SetupDiGetClassDevs(
  ref GUID  lpHidGuid,
  int*  Enumerator,
  int*  hwndParent,
```

```
  int  Flags
);
```

Первый параметр передает GUID выбранного класса. Если этот параметр `NULL`, то будут выбраны все устройства в системе.

Следующий вызов вернет все устройства в системе:

```
SetupDiGetClassDevs(nil, nil, 0, DIGCF_PRESENT or DIGCF_ALLCLASSES);
```

А такой вызов вернет устройства выбранного класса:

```
SetupDiGetClassDevs(@Guid, nil, 0, DIGCF_PRESENT);
```

Функция возвращает дескриптор (handler) выбранного набора устройств. Этот дескриптор можно использовать при вызовах функций работы с устройствами `SetupDiEnumDeviceInfo` или `SetupDiEnumDeviceInterfaces`. В MSDN этот параметр обычно имеет имя `DeviceInfoSet`.

После завершения работы с дескриптором его следует освободить с помощью вызова `SetupDiDestroyDeviceInfoList`.

16.1.1. Возвращаемое значение

При успешном выполнении функция возвращает корректный дескриптор. При ошибке возвращается `INVALID_HANDLE_VALUE`.

16.2. Функция *SetupDiDestroyDeviceInfoList*: освобождение блока описания устройства

Функция `SetupDiDestroyDeviceInfoList` освобождает дескриптор, полученный при вызове `SetupDiGetClassDevs`.

Формат заголовка на языке C имеет вид:

```
BOOL SetupDiDestroyDeviceInfoList(
  HDEVINFO DeviceInfoSet
);
```

Формат заголовка на языке Delphi имеет вид:

```
function SetupDiDestroyDeviceInfoList(
       DeviceInfoSet: HDEVINFO): LongBool; stdcall;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("setupapi.dll", SetLastError=true)]
static extern  unsafe int SetupDiDestroyDeviceInfoList(
  int DeviceInfoSet
);
```

16.2.1. Возвращаемое значение

При успешном выполнении функция возвращает ненулевое значение.

16.3. Функция *SetupDiEnumDeviceInterfaces*: информация об устройстве

Функция `SetupDiEnumDeviceInterfaces` возвращает структуру `SP_DEVICE_INTERFACE_DATA`, содержащую информацию об устройстве. Для получения конкретного блока информации используется функция `SetupDiGetDeviceInterfaceDetail`.

Описание структуры на языке C:

```
typedef struct _SP_DEVICE_INTERFACE_DATA {
  DWORD cbSize;
  GUID InterfaceClassGuid;
  DWORD Flags;
  ULONG_PTR Reserved;
} SP_DEVICE_INTERFACE_DATA;
```

Описание структуры на языке Delphi:

```
SP_DEVICE_INTERFACE_DATA = packed record
  cbSize: DWORD;
  InterfaceClassGuid: TGUID;
  Flags: DWORD;
  Reserved: ULONG_PTR;
end;
```

Описание структуры на языке C# имеет вид:

```
[StructLayout(LayoutKind.Sequential)]
public unsafe struct SP_DEVICE_INTERFACE_DATA
{
  public int cbSize;
  public GUID InterfaceClassGuid;
  public int Flags;
  public int Reserved;
}
```

Описание функции на языке C имеет вид:

```
BOOL SetupDiEnumDeviceInterfaces(
  HDEVINFO DeviceInfoSet,
  PSP_DEVINFO_DATA DeviceInfoData,
  const GUID* InterfaceClassGuid,
  DWORD MemberIndex,
  PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData
);
```

Описание функции на языке Delphi имеет вид:

```
function SetupDiEnumDeviceInterfaces(
  DeviceInfoSet: HDEVINFO;
  DeviceInfoData: PSPDevInfoData;
  const InterfaceClassGuid: TGUID;
  MemberIndex: DWORD;
  var DeviceInterfaceData: TSPDeviceInterfaceData
): LongBool; stdcall;
```

Описание функции на языке C# имеет вид:

```
[DllImport("setupapi.dll", SetLastError=true)]
static extern  unsafe int SetupDiEnumDeviceInterfaces(
  int  DeviceInfoSet,
  int  DeviceInfoData,
  ref  GUID  InterfaceClassGuid,
  int  MemberIndex,
  ref  SP_DEVICE_INTERFACE_DATA lpDeviceInterfaceData);
```

Первый параметр передает дескриптор, полученный при вызове функции `SetupDiGetClassDevs`. Второй параметр можно оставить `NULL`. Третий параметр — GUID выбранного класса устройств, четвертый — порядковый номер устройства и пятый — буфер для получения информации.

16.3.1. Возвращаемое значение

При успешном выполнении функция возвращает значение `TRUE`.

16.4. Функция *SetupDiGetDeviceInterfaceDetail*: детальная информация об устройстве

Функция `SetupDiGetDeviceInterfaceDetail` используется для получения конкретной информации из структуры `SP_DEVICE_INTERFACE_DATA`, которую возвращает функция `SetupDiEnumDeviceInterfaces`.

Описание функции на языке C имеет вид:

```
BOOL SetupDiGetDeviceInterfaceDetail(
  HDEVINFO DeviceInfoSet,
  PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
  PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData,
  DWORD DeviceInterfaceDetailDataSize,
  PDWORD RequiredSize,
  PSP_DEVINFO_DATA DeviceInfoData
);
```

Описание функции на языке Delphi имеет вид:

```
function SetupDiGetDeviceInterfaceDetail(
  DeviceInfoSet: HDEVINFO;
  DeviceInterfaceData: PSPDeviceInterfaceData;
  DeviceInterfaceDetailData: PSPDeviceInterfaceDetailData;
  DeviceInterfaceDetailDataSize: DWORD;
  var RequiredSize: DWORD;
  Device: PSPDevInfoData
): LongBool; stdcall;
```

Описание функции на языке C# имеет вид:

```
[DllImport("setupapi.dll", SetLastError=true)]
static extern  unsafe int SetupDiGetDeviceInterfaceDetail(
  int  DeviceInfoSet,
  ref SP_DEVICE_INTERFACE_DATA lpDeviceInterfaceData,
  int* aPtr,
  int detailSize,
  ref int requiredSize,
  int* bPtr);
[DllImport("setupapi.dll", SetLastError=true)]
unsafe int SetupDiGetDeviceInterfaceDetail(
     int  DeviceInfoSet,
     ref SP_DEVICE_INTERFACE_DATA lpDeviceInterfaceData,
```

```
     ref PSP_DEVICE_INTERFACE_DETAIL_DATA
                         myPSP_DEVICE_INTERFACE_DETAIL_DATA,
     int detailSize,
     ref int requiredSize,
    int* bPtr);
```

Двойное описание для C# связано с необходимостью вызывать эту функцию два раза: один раз для получения необходимой длины буфера, второй — для получения информации.

Структура `PSP_DEVICE_INTERFACE_DETAIL_DATA` описывается следующим образом. На языке C:

```
typedef struct _SP_DEVICE_INTERFACE_DETAIL_DATA {
  DWORD cbSize;
  TCHAR DevicePath[0];
} SP_DEVICE_INTERFACE_DETAIL_DATA;
```

На языке Delphi:

```
SP_DEVICE_INTERFACE_DETAIL_DATA = packed record
  cbSize: DWORD;
  DevicePath: array [0..0] of AnsiChar;
end;
```

Описание структуры на языке C# имеет вид:

```
[StructLayout(LayoutKind.Sequential, CharSet= CharSet.Ansi)]
unsafe struct PSP_DEVICE_INTERFACE_DETAIL_DATA
{
  public int  cbSize;
  [MarshalAs(UnmanagedType.ByValTStr, SizeConst= 256)]
  public string DevicePath;
}
```

16.5. Функция *SetupDiEnumDeviceInfo*: информация об устройстве

Функция `SetupDiEnumDeviceInfo` возвращает структуру `PSP_DEVINFO_DATA`, содержащую информацию об устройстве.

Описание этой структуры на языке C выглядит следующим образом:

```
typedef struct  {
  DWORD cbSize;
```

```
  GUID ClassGuid;
  DWORD DevInst;
  ULONG_PTR Reserved;
} SP_DEVINFO_DATA;
```

Описание этой структуры на языке Delphi имеет вид:

```
SP_DEVINFO_DATA = packed record
  cbSize: DWORD;
  ClassGuid: TGUID;
  DevInst: DWORD;
  Reserved: ULONG_PTR;
end;
```

Описание структуры на языке C# имеет вид:

```
[StructLayout(LayoutKind.Sequential)]
public struct SP_DEVINFO_DATA
{
  public int cbSize;
  public System.Guid ClassGuid;
  public int DevInst;
  public int Reserved;
}
```

Описание функции на языке C имеет вид:

```
BOOL SetupDiEnumDeviceInfo(
  HDEVINFO DeviceInfoSet,
  DWORD MemberIndex,
  PSP_DEVINFO_DATA DeviceInfoData
);
```

Описание функции на языке Delphi имеет вид:

```
function SetupDiEnumDeviceInfo(
  DeviceInfoSet: HDEVINFO;
  MemberIndex: DWORD;
  var DeviceInfoData: TSPDevInfoData
): LongBool; stdcall;
```

Описание функции на языке C# имеет вид:

```
[DllImport("setupapi.dll", SetLastError=true)]
unsafe int SetupDiEnumDeviceInfo(
  int  DeviceInfoSet,
```

```
  int  Index,
  ref  SP_DEVINFO_DATA DeviceInfoData
);
```

Для получения детальной информации используется функция
`SetupDiGetDeviceRegistryProperty`.

16.6. Функция *SetupDiGetDeviceRegistryProperty*: получение Plug and Play свойств устройства

Функция `SetupDiGetDeviceRegistryProperty` позволяет получить свойства устройства из структуры `SP_DEVINFO_DATA`, полученной с помощью вызова `SetupDiEnumDeviceInfo`.

Описание функции на языке C имеет вид:

```
WINSETUPAPI BOOL WINAPI SetupDiGetDeviceRegistryProperty(
  IN HDEVINFO  DeviceInfoSet,
  IN PSP_DEVINFO_DATA  DeviceInfoData,
  IN DWORD  Property,
  OUT PDWORD  PropertyRegDataType,
  OUT PBYTE  PropertyBuffer,
  IN DWORD  PropertyBufferSize,
  OUT PDWORD  RequiredSize  OPTIONAL
);
```

Описание функции на языке Delphi имеет вид:

```
function SetupDiGetDeviceRegistryProperty(
  DeviceInfoSet: HDEVINFO;
  const DeviceInfoData: TSPDevInfoData;
  Property_: DWORD;
  var PropertyRegDataType: DWORD;
  PropertyBuffer: PBYTE;
  PropertyBufferSize: DWORD;
  var RequiredSize: DWORD
): LongBool; stdcall;
```

Описание функции на языке C# имеет вид:

```
[DllImport("setupapi.dll", SetLastError=true)]
unsafe int SetupDiGetDeviceRegistryProperty(
```

```
  int                    DeviceInfoSet,
  ref SP_DEVINFO_DATA    DeviceInfoData,
  RegPropertyType        Property,
  int*                   PropertyRegDataType,
  int*                   PropertyBuffer,
  int                    PropertyBufferSize,
  ref int                RequiredSize
);
```

Параметр `Property` указывает, какое именно свойство требуется получить. Для Delphi и C используются константы, а для C# тип `RegPropertyType`, описание которого можно найти на компакт-диске.

Для установки свойств можно использовать симметричную функцию `SetupDiSetDeviceRegistryProperty`, но устанавливать разрешается далеко не все свойства. Подробности можно найти в MSDN.

16.7. Функция *CM_Get_DevNode_Status*: статус устройства

Функция `CM_Get_DevNode_Status` возвращает статус устройства, представляющего собой "узел" дерева в Менеджере Устройств.

Описание функции на языке C имеет вид:

```
CMAPI CONFIGRET WINAPI CM_Get_DevNode_Status(
  OUT PULONG  pulStatus,
  OUT PULONG  pulProblemNumber,
  IN DEVINST  dnDevInst,
  IN ULONG  ulFlags
);
```

Описание функции на языке Delphi имеет вид:

```
function CM_Get_DevNode_Status(
  var ulStatus: DWord;
  var ulProblemNumber: DWord;
  dnDevInst: DWord;
  ulFlags: Dword): DWord;
stdcall; external 'setupapi.dll';
```

Описание функции на языке C# имеет вид:

```
[DllImport("setupapi.dll", SetLastError=true)]
public static extern unsafe int CM_Get_DevNode_Status(
```

```
  ref  int pulStatus,
  ref  int pulProblemNumber,
  int  DevInst,
  int  ulFlags
);
```

Функция возвращает поле флагов `pulStatus`, состоящее из следующих флагов:

```
public const int DN_HAS_PROBLEM              = 0x0400;
public const int CM_PROB_DISABLED            = 0x0016;
public const int CM_PROB_HARDWARE_DISABLED   = 0x001D;
public const int DN_DISABLEABLE              = 0x2000;
```

Если установлен флаг `DN_HAS_PROBLEM`, то в поле `pulProblemNumber` записывается номер ошибки.

16.8. Функция *CM_Request_Device_Eject*: безопасное извлечение устройства

Функция `CM_Request_Device_Eject` позволяет подготовить устройство (например, USB Flash Disk) для безопасного извлечения.

Описание функции на языке C имеет вид:

```
CMAPI CONFIGRET WINAPI CM_Request_Device_Eject(
  IN DEVINST  dnDevInst,
  OUT PPNP_VETO_TYPE  pVetoType,
  OUT LPTSTR  pszVetoName,
  IN ULONG  ulNameLength,
  IN ULONG  ulFlags
);
```

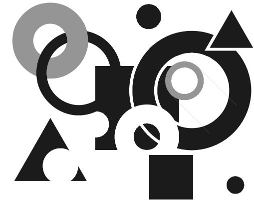
Описание функции на языке Delphi имеет вид:

```
function CM_Request_Device_EjectA(
  dnDevInst: DWord;
  pVetoType: PDWord;
  pszVetoName: PChar;
  ulNameLength: DWord;
  ulFlags: DWord): DWord;
stdcall; external 'setupapi.dll';
```

Описание функции на языке C# имеет вид:

```
[DllImport("setupapi.dll", SetLastError=true)]
public static extern unsafe int CM_Request_Device_Eject(
  int      dnDevInst,
  int *    pVetoType,
  int *    pszVetoName,
  int      ulNameLength,
  int      ulFlags
);
```

При вызове все параметры, кроме первого можно установить в `null`. Единственная неприятность при выполнении этой функции — если устройство не может быть отключено, система отображает окно с сообщением и пользователю придется закрывать его самостоятельно.



Глава 17

Структуры и функции Windows HID API

HID-функции, начинающиеся с префикса `HidD`, возвращают результат типа `LongBool`, а функции, начинающиеся с префикса `HidP` — специальные значения, приведенные в листинге 10.15 (есть исключения, например, функция `HidP_MaxDataListLength`).

17.1. Функция *HidD_Hello*: проверка библиотеки

Функция `HidD_Hello` генерирует тестовую строку. Функция может использоваться для проверки наличия HID-библиотеки. Эта функция не документирована.

Формат заголовка на языке C имеет вид:

```
ULONG HidD_Hello (OUT PCHAR Buffer, IN ULONG BufferLength);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_Hello(
 Buffer: PChar;
 BufferLength: ULONG
): ULONG;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll")]
extern bool HidD_Hello(ref byte lpBuffer, int BufferLength);
```

Первый параметр задает адрес буфера для строки, а второй — его размер.

Пример использования этой функции показан в листинге 17.1.

Листинг 17.1. Пример использования функции `HidD_Hello`

```
procedure TForm1.Button1Click(Sender: TObject);
var S : String[20];
begin
 HidD_Hello(@S[1], 20);
 S[0]:= Char(20);
 MessageDlg(S, mtWarning, [mbOK], 0);
end;
```

17.2. Функция *HidD_GetHidGuid*: получение GUID

Функция `HidD_GetHidGuid` возвращает GUID для HID-класса. Это значение используется для поиска устройств этого класса.

Формат заголовка на языке C имеет вид:

```
void __stdcall HidD_GetHidGuid(
 OUT LPGUID HidGuid
);
```

Формат заголовка на языке Delphi имеет вид:

```
procedure HidD_GetHidGuid(
 var HidGuid: TGUID
);
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll", SetLastError=true)]
static extern unsafe void HidD_GetHidGuid(
 ref GUID lpHidGuid
);
```

Единственный параметр передает переменную для результата. Функция возвращает значение `True` при успешном выполнении.

Пример использования этой функции показан в листинге 17.2.

Листинг 17.2. Пример использования функции `HidD_GetHidGuid`

```
var
 HidGuid : TGuid;
// Получить GUID для класса HID
HidD_GetHidGuid(HidGuid);
```

17.3. Функция *HidD_GetPreparsedData*: создание описателя устройства

Функция `HidD_GetPreparsedData` подготавливает буфер типа `THIDPPreparsedData`, который используется для работы некоторых других HID-функций. После использования буфер должен быть освобожден с помощью вызова `HidD_FreePreparsedData`.

Формат заголовка на языке C имеет вид:

```
BOOLEAN __stdcall HidD_GetPreparsedData(
  IN  HANDLE                 HidDeviceObject,
  OUT PHIDP_PREPARSED_DATA * PreparsedData
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_GetPreparsedData(
  HidDeviceObject: THandle;
  var PreparsedData: PHIDPPreparsedData
): LongBool;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll", SetLastError=true)]
private unsafe static extern int HidD_GetPreparsedData(
  int HidDeviceObject,
  ref int PreparsedData
);
```

Первый параметр передает дескриптор устройства, полученный с помощью вызова `CreateFile`, а второй — указатель на буфер `THIDPPreparsedData`.

Пример использования этой функции показан в листинге 17.3.

Листинг 17.3. Пример использования функции `HidD_GetPreparsedData`

```
var
  PreparsedData: PHIDPPreparsedData;

If HidD_GetPreparsedData(HidHandle, PreparsedData) then begin
  // использование PreparsedData
... ... ... ... ... ... ... ... ... ...
  // Освободить блок PreparsedData
  HidD_FreePreparsedData(PreparsedData);
End;
```

17.4. Функция *HidD_FreePreparsedData*: освобождение описателя устройства

Функция `HidD_FreePreparsedData` освобождает буфер, созданный функцией `HidD_GetPreparsedData`.

Формат заголовка на языке С имеет вид:

```
BOOLEAN __stdcall HidD_FreePreparsedData(
 IN PHIDP_PREPARSED_DATA PreparsedData
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_FreePreparsedData(
 PreparsedData: PHIDPPreparsedData
): LongBool;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll", SetLastError=true)]
static extern unsafe int HidD_FreePreparsedData(
 int PreparsedData
);
```

Пример использования этой функции показан в листинге 17.3.

17.5. Функция *HidD_GetFeature*: получение Feature-репорта

Функция `HidD_GetFeature` позволяет получить Feature-репорт от HID-устройства.

Формат заголовка на языке С имеет вид:

```
BOOLEAN __stdcall HidD_GetFeature (
 IN    HANDLE   HidDeviceObject,
 OUT   PVOID    ReportBuffer,
 IN    ULONG    ReportBufferLength
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_GetFeature(
 HidDeviceObject: THandle;
 var ReportBuffer;
```

```
 ReportBufferLength: Integer
): LongBool;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll")]
static public extern bool HidD_GetFeature(
 int HidDeviceObject,
 ref byte ReportBuffer,
 int ReportBufferLength
);
```

Первый параметр передает дескриптор устройства, полученный с помощью вызова `CreateFile`, второй — указатель на буфер репорта, третий — размер буфера. Размер буфера должен быть на единицу больше, чем размер репорта, описанный в дескрипторе HID-устройства[1]. Его можно получить с помощью функции `HidP_GetCaps`.

Если используется несколько репортов, то в первом байте буфера должен передаваться идентификатор репорта (Report ID).

Пример использования этой функции показан в листинге 17.4.

Листинг 17.4. Пример использования функции `HidD_GetFeature`

```
var
  FeatureReport : Array [0..255] of Byte;

If Capabilities.FeatureReportByteLength > 0 then begin
  FillChar(Feature, SizeOf(Feature), #0);
  //  Feature [1]:= 1; идентификатор репорта
  If HidD_GetFeature(HidHandle, Feature,
        Capabilities.FeatureReportByteLength) then begin
      // использование репорта
  End;
End;
```

[1] Спецификация USB, так же как и Windows, использует понятие "дескриптор". Следует различать дескриптор, возвращаемый функцией `CreateFile` (тип `THandle`) и дескриптор USB-устройства.

17.6. Функция *HidD_SetFeature*: передача Feature-репорта

Функция `HidD_SetFeature` используется для передачи Feature-репорта. Размер передаваемого буфера должен быть на единицу больше, чем размер, описанный в дескрипторе устройства. В первом байте буфера передается идентификатор репорта (Report ID), если он используется, и ноль, если нет.

Формат заголовка на языке C имеет вид:

```
BOOLEAN __stdcall HidD_SetFeature (
 IN     HANDLE   HidDeviceObject,
 IN     PVOID    ReportBuffer,
 IN     ULONG    ReportBufferLength
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_SetFeature(
 HidDeviceObject: THandle;
 var Report;
 Size: Integer
): LongBool;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll")]
static public extern bool HidD_SetFeature(
 int HidDeviceObject,
 ref byte lpReportBuffer,
 int ReportBufferLength
);
```

Параметры этой функции и возвращаемое значение совпадают с параметрами функции `HidD_GetFeature`.

17.7. Функция *HidD_GetNumInputBuffers*: получение числа буферов

Функция `HidD_GetNumInputBuffers` возвращает количество репортов, сохраняемое в кольцевом буфере драйвера.

Формат заголовка на языке C имеет вид:

```
BOOLEAN HidD_GetNumInputBuffers(
  IN HANDLE   HidDeviceObject,
```

```
  OUT PULONG  NumberBuffers
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_GetNumInputBuffers(
  HidDeviceObject: THandle;
  var NumBufs: Integer
): LongBool;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll")]
static public extern bool HidD_GetNumInputBuffers(
               int HidDeviceObject, ref int NumberBuffers);
```

Первый параметр передает дескриптор устройства, полученный с помощью вызова `CreateFile`, второй — переменную для возвращения результата.

17.8. Функция *HidD_SetNumInputBuffers*: установка числа буферов

Функция `HidD_SetNumInputBuffers` устанавливает количество репортов, сохраняемое в кольцевом буфере драйвера.

Формат заголовка на языке C имеет вид:

```
BOOLEAN HidD_SetNumInputBuffers(
  IN HANDLE  HidDeviceObject,
  OUT ULONG  NumberBuffers
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_SetNumInputBuffers(HidDeviceObject: THandle;
               NumBufs: Integer): LongBool;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll")]
static public extern bool HidD_SetNumInputBuffers(
                 int HidDeviceObject, int NumberBuffers);
```

Параметры этой функции и возвращаемое значение совпадают с параметрами функции `HidD_GetNumInputBuffers`.

17.9. Функция *HidD_GetAttributes*: получение атрибутов устройства

Функция `HidD_GetAttributes` возвращает атрибуты устройства, описываемые структурой `THIDDAttributes` (листинг 17.5).

Листинг 17.5. Описание структуры THIDDAttributes

[C]

```
typedef struct _HIDD_ATTRIBUTES {
 ULONG   Size;           // размер структуры = sizeof(_HIDD_ATTRIBUTES)
 USHORT  VendorID;       // идентификатор производителя
 USHORT  ProductID;      // идентификатор продукта
 USHORT  VersionNumber; // версия
 // Могут присутствовать дополнительные поля
} HIDD_ATTRIBUTES, *PHIDD_ATTRIBUTES;
```

[Delphi]

```
 PHIDDAttributes = ^THIDDAttributes;
 THIDDAttributes = record
    Size:           ULONG; // размер структуры
    VendorID:       Word;  // идентификатор производителя
    ProductID:      Word;  // идентификатор продукта
    VersionNumber: Word;  // версия
    // Могут присутствовать дополнительные поля
  end;
```

[C#]

```
[StructLayout(LayoutKind.Sequential)]
public unsafe struct HIDD_ATTRIBUTES
{
 public int Size; // размер структуры
 public System.UInt16    VendorID;       // идентификатор производителя
 public System.UInt16    ProductID;      // идентификатор продукта
 public System.UInt16    VersionNumber; // версия
 // Могут присутствовать дополнительные поля
}
```

Формат заголовка на языке C имеет вид:

```
BOOLEAN __stdcall HidD_GetAttributes (
 IN HANDLE HidDeviceObject,      // дескриптор устройства
 OUT PHIDD_ATTRIBUTES Attributes // результат
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_GetAttributes(
 HidDeviceObject: THandle;      // дескриптор устройства
 var HidAttrs: THIDDAttributes  // результат
): LongBool;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll", SetLastError=true)]
private static extern int HidD_GetAttributes(
 int hObject,                   // дескриптор устройства,
 ref HIDD_ATTRIBUTES Attributes // результат
);
```

Первый параметр передает дескриптор устройства, полученный с помощью вызова `CreateFile`, второй — буфер для возвращения результата. Поле `Size` структуры `THIDDAttributes` должно быть заполнено перед вызовом функции.

Пример использования этой функции показан в листинге 17.6.

Листинг 17.6. Пример использования функции `HidD_GetAttributes`

```
Var
  Attributes : THIDDAttributes;

Attributes.Size := SizeOf(THIDDAttributes);
If HidD_GetAttributes(HidHandle, Attributes) then begin
   // функция выполнена успешно
End;
```

17.10. Функция *HidD_GetManufacturerString*: получение строки производителя

Функция `HidD_GetManufacturerString` возвращает строку производителя, индекс которой описан в дескрипторе конфигурации устройства.

Формат заголовка на языке C имеет вид:

```
BOOLEAN HidD_GetManufacturerString(
  IN HANDLE  HidDeviceObject,
  OUT PVOID  Buffer,
  IN ULONG   BufferLength
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_GetManufacturerString(HidDeviceObject: THandle;
             Buffer: PWideChar; BufferLength: Integer): LongBool;
```

Формат заголовка на языке C# имеет вид:

```
static public extern int HidD_GetManufacturerString(
      int HidDeviceObject, ref byte lpBuffer, int BufferLength);
```

Первый параметр передает дескриптор устройства, полученный с помощью вызова `CreateFile`, второй — буфер для строки, третий — размер буфера.

Пример использования показан в листинге 17.7.

Листинг 17.7. Пример использования функции `HidD_GetManufacturerString`

```
[Delphi]
Var
  Buffer  : array [0..253] of WideChar;
  S : String;

FillChar(Buffer, SizeOf(Buffer), #0);
If HidD_GetManufacturerString(HidHandle, Buffer, SizeOf(Buffer)) then
begin
  S:= Format('   Производитель=%s',[Buffer]));
End;
[C#]
static unsafe void HidInfo(string DeviceName)
{
  IntPtr handle = Win32.CreateFile(
         DeviceName, Win32.GENERIC_READ, 0, IntPtr.Zero,
         Win32.OPEN_EXISTING, Win32.FILE_FLAG_OVERLAPPED, IntPtr.Zero);
  byte [] lpBuffer = new byte[1024];
  HidApiDeclarations.HidD_GetManufacturerString(handle.ToInt32(),
              ref lpBuffer[0], 1024);
```

```
  foreach (byte b in lpBuffer)
    Console.Write(string.Format("{0}", (char)b));
  Win32.CloseHandle(handle);
}
```

17.11. Функция *HidD_GetProductString*: получение строки продукта

Функция `HidD_GetProductString` возвращает строку продукта, индекс которой описан в дескрипторе конфигурации устройства.

Формат заголовка на языке C имеет вид:

```
BOOLEAN  HidD_GetProductString(
  IN HANDLE  HidDeviceObject,
  OUT PVOID  Buffer,
  IN ULONG  BufferLength
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_GetProductString(HidDeviceObject: THandle;
             Buffer: PWideChar; BufferLength: Integer): LongBool;
```

Первый параметр передает дескриптор устройства, полученный с помощью вызова `CreateFile`, второй — буфер для строки, третий — размер буфера.

17.12. Функция *HidD_GetSerialNumberString*: получение строки серийного номера

Функция `HidD_GetSerialNumberString` возвращает строку серийного номера, индекс которой описан в дескрипторе конфигурации устройства.

Формат заголовка на языке C имеет вид:

```
BOOLEAN  HidD_GetSerialNumberString(
  IN HANDLE  HidDeviceObject,
  OUT PVOID  Buffer,
  IN ULONG  BufferLength
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_GetSerialNumberString(HidDeviceObject: THandle;
             Buffer: PWideChar; BufferLength: Integer): LongBool;
```

Первый параметр передает дескриптор устройства, полученный с помощью вызова `CreateFile`, второй — буфер для строки, третий — размер буфера.

17.13. Функция *HidD_GetIndexedString*: получение строки по индексу

Функция `HidD_GetIndexedString` возвращает строку по индексу.

Формат заголовка на языке C имеет вид:

```
BOOLEAN HidD_GetIndexedString(
  IN HANDLE  HidDeviceObject,
  IN ULONG  StringIndex,
  OUT PVOID  Buffer,
  IN ULONG  BufferLength
  );
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_GetIndexedString(HidDeviceObject: THandle;
      Index: Integer; Buffer: PWideChar;
                      BufferLength: Integer): LongBool;
```

Первый параметр передает дескриптор устройства, полученный с помощью вызова `CreateFile`, второй — индекс запрашиваемой строки, третий — буфер для строки, четвертый — размер буфера.

Пример использования показан в листинге 17.8.

Листинг 17.8. Пример использования функции `HidD_GetIndexedString`

```
Function GetString(StrDescriptor : Byte): WideString;
var Buffer : array [0..253] of WideChar;
begin
 Result := 'Ошибка';
 if StrDescriptor <> 0 then
  if HidD_GetIndexedString(HidHandle, StrDescriptor,
          Buffer, SizeOf(Buffer)) then
              Result:= Buffer;
end;
```

17.14. Функция *HidD_GetInputReport*: получение Input-репорта

Функция `HidD_GetInputReport` позволяет получить Input-репорт. Эта функция доступна, начиная с Windows XP.

Формат заголовка на языке C имеет вид:

```
BOOLEAN __stdcall HidD_GetInputReport(
  IN HANDLE  HidDeviceObject,
  IN OUT  PVOID  ReportBuffer,
  IN ULONG  ReportBufferLength
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_GetInputReport(HidDeviceObject: THandle;
              Buffer: Pointer; BufferLength: ULONG): LongBool;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll")]
static public extern bool HidD_GetInputReport(
    int HidDeviceObject,ref byte lpReportBuffer,
    int ReportBufferLength);
```

Первый параметр передает дескриптор устройства, полученный с помощью вызова `CreateFile`, второй — буфер для репорта, третий — размер буфера. Размер буфера может быть получен с помощью вызова функции `HidP_GetCaps`.

Пример использования показан в листинге 17.9.

Листинг 17.9. Пример использования функции `HidD_GetInputReport`

```
var
  InputReport : Array [0..255] of Byte;

If HidD_GetInputReport(HidHandle, @InputReport,
                 Capabilities.InputReportByteLength) then begin
   // репорт получен успешно
End;
```

17.15. Функция *HidD_SetOutputReport*: передача Output-репорта

Функция `HidD_SetOutputReport` позволяет передать Output-репорт. Эта функция доступна, начиная с Windows XP.

Формат заголовка на языке C имеет вид:

```
BOOLEAN __stdcall HidD_SetOutputReport(
  IN HANDLE  HidDeviceObject,
  IN PVOID  ReportBuffer,
  IN ULONG  ReportBufferLength
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidD_SetOutputReport(HidDeviceObject: THandle;
                Buffer: Pointer; BufferLength: ULONG): LongBool;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll")]
static public extern bool HidD_SetOutputReport(
    int HidDeviceObject, ref byte lpReportBuffer,
    int ReportBufferLength);
```

Первый параметр передает дескриптор устройства, полученный с помощью вызова `CreateFile`, второй — буфер репорта, третий — размер буфера.

17.16. Функция *HidP_GetCaps*: получение свойств устройства

Функция `HidP_GetCaps` позволяет получить свойства устройства. Выходным параметром функции является структура `THIDPCaps` (листинг 17.10).

Формат заголовка на языке C имеет вид:

```
NTSTATUS __stdcall HidP_GetCaps(
 IN PHIDP_PREPARSED_DATA PreparsedData,
 OUT PHIDP_CAPS Capabilities
);
```

Формат заголовка на языке Delphi имеет вид:

```
function HidP_GetCaps(
 PreparsedData: PHIDPPreparsedData;
```

```
 var Capabilities: THIDPCaps
): NTSTATUS;
```

Формат заголовка на языке C# имеет вид:

```
[DllImport("hid.dll", SetLastError=true)]
private unsafe static extern int HidP_GetCaps(
 int pPHIDP_PREPARSED_DATA,
 IN  PHIDP_PREPARSED_DATA PreparsedData,
 ref HIDP_CAPS myPHIDP_CAPS
);
```

Пример использования показан в листинге 17.11.

Листинг 17.10. Структура свойств устройства

[C]

```
typedef struct _HIDP_CAPS
{
 USAGE    Usage;
 USAGE    UsagePage;
 USHORT   InputReportByteLength;
 USHORT   OutputReportByteLength;
 USHORT   FeatureReportByteLength;
 USHORT   Reserved[17];
 USHORT   NumberLinkCollectionNodes;
 USHORT   NumberInputButtonCaps;
 USHORT   NumberInputValueCaps;
 USHORT   NumberInputDataIndices;
 USHORT   NumberOutputButtonCaps;
 USHORT   NumberOutputValueCaps;
 USHORT   NumberOutputDataIndices;
 USHORT   NumberFeatureButtonCaps;
 USHORT   NumberFeatureValueCaps;
 USHORT   NumberFeatureDataIndices;
} HIDP_CAPS, *PHIDP_CAPS;
```

[Delphi]

```
PTHIDPCaps = ^ THIDPCaps;
```

```
THIDPCaps = record
  Usage:                       TUsage; // значение Usage
  UsagePage:                   TUsage; // значение Usage Page
  InputReportByteLength:       Word;   // размер Input-репорта
  OutputReportByteLength:      Word;   // размер Output-репорта
  FeatureReportByteLength:     Word;   // размер Feature-репорта
  Reserved:                    array [0..16] of Word;
  // Специфические HID-значения
  NumberLinkCollectionNodes: Word;
  NumberInputButtonCaps:       Word;
  NumberInputValueCaps:        Word;
  NumberInputDataIndices:      Word;
  NumberOutputButtonCaps:      Word;
  NumberOutputValueCaps:       Word;
  NumberOutputDataIndices:     Word;
  NumberFeatureButtonCaps:     Word;
  NumberFeatureValueCaps:      Word;
  NumberFeatureDataIndices:    Word;
end;
```

[C#]

```
[StructLayout(LayoutKind.Sequential)]
public unsafe struct HIDP_CAPS
{
 public System.UInt16  Usage;
 public System.UInt16   UsagePage;
 public System.UInt16   InputReportByteLength;
 public System.UInt16   OutputReportByteLength;
 public System.UInt16   FeatureReportByteLength;
 [MarshalAs(UnmanagedType.ByValArray, SizeConst=17)]
 public System.UInt16[] Reserved;
 public System.UInt16  NumberLinkCollectionNodes;
 public System.UInt16  NumberInputButtonCaps;
 public System.UInt16  NumberInputValueCaps;
 public System.UInt16  NumberInputDataIndices;
```

```
 public System.UInt16  NumberOutputButtonCaps;
 public System.UInt16  NumberOutputValueCaps;
 public System.UInt16  NumberOutputDataIndices;
 public System.UInt16  NumberFeatureButtonCaps;
 public System.UInt16  NumberFeatureValueCaps;
 public System.UInt16  NumberFeatureDataIndices;
}
```

Листинг 17.11. Пример использования функции `HidP_GetCaps`

```
Var
  Capabilities : HIDP_CAPS;
  PreparsedData: PHIDPPreparsedData;

If HidD_GetPreparsedData(HidHandle, PreparsedData) then begin
  HidP_GetCaps(PreparsedData, Capabilities);
  // Capabilities.UsagePage
  // Capabilities.InputReportByteLength
  // Capabilities.OutputReportByteLength
  // Capabilities.FeatureReportByteLength
End;
```

17.17. Функция *HidP_MaxDataListLength*: получение размеров репортов

Функция `HidP_MaxDataListLength` позволяет получить размер репорта, определенного параметром `ReportType`.

Формат заголовка на языке Delphi имеет вид:

```
function HidP_MaxDataListLength(
  ReportType: DWORD;
  PreparsedData: PHIDPPreparsedData
): ULONG;
```

Первый параметр передает тип репорта:

```
  HidP_Input   = 0;
  HidP_Output  = 1;
  HidP_Feature = 2;
```

Второй параметр передает указатель на структуру `THIDPPreparsedData`, получаемую вызовом `HidP_GetCaps`. Функция возвращает максимальный размер звапрашиваемого типа репорта. Значение `0` означает некорректный либо индекс типа репорта, либо буфер данных.

17.18. Функция *HidD_FlushQueue*: сброс буферов

Функция `HidD_FlushQueue` очищает все незавершенные Input-репорты во входной очереди.

Формат функции на языке C имеет вид:

```
BOOLEAN  HidD_FlushQueue(
  IN HANDLE  HidDeviceObject
);
```

Формат функции на языке Delphi имеет вид:

```
function HidD_FlushQueue(
               HidDeviceObject: THandle): LongBool; stdcall;
```

Формат функции на языке C# имеет вид:

```
[DllImport("hid.dll")]
static public extern bool HidD_FlushQueue(int HidDeviceObject);
```

Единственный параметр этой функции передает дескриптор устройства.

17.19. Функция *HidP_GetLinkCollectionNodes*: дерево коллекций

Функция `HidP_GetLinkCollectionNodes` возвращает описание дерева коллекций. Первый параметр передает указатель на буфер для возвращения результата, второй — передает размер этого буфера, а третий — блок данных, возвращаемый функцией `GetPreparsedData`.

Формат функции на языке C имеет вид:

```
NTSTATUS  HidP_GetLinkCollectionNodes(
  OUT PHIDP_LINK_COLLECTION_NODE  LinkCollectionNodes,
  IN OUT PULONG  LinkCollectionNodesLength,
  IN PHIDP_PREPARSED_DATA  PreparsedData
);
```

Формат функции на языке Delphi имеет вид:

```
function HidP_GetLinkCollectionNodes(
  LinkCollectionNodes: PHIDPLinkCollectionNode;
  var   LinkCollectionNodesLength:
  ULONG; PreparsedData: PHIDPPreparsedData
): NTSTATUS; stdcall;
```

Пример вызова и построения дерева коллекций см. в разд. 10.8.2.

17.20. Функции *HidP_GetScaledUsageValue* и *HidP_SetScaledUsageValue*: получение и задание преобразованных значений

HID-дескриптор позволяет описывать границы величин, используемых для обмена данными. Соответственно логические и физические значения величин могут отличаться (*см. разд. 10.8.1*). Функции `HidP_GetScaledUsageValue` и `HidP_SetScaledUsageValue` позволяют работать с преобразованными значениями.

Формат функций на языке C имеет вид:

```
NTSTATUS  HidP_GetScaledUsageValue(
  IN HIDP_REPORT_TYPE  ReportType,
  IN USAGE   UsagePage,
  IN USHORT  LinkCollection  OPTIONAL,
  IN USAGE   Usage,
  OUT PLONG  UsageValue,
  IN PHIDP_PREPARSED_DATA  PreparsedData,
  IN PCHAR   Report,
  IN ULONG   ReportLength
);
NTSTATUS  HidP_SetScaledUsageValue(
  IN HIDP_REPORT_TYPE  ReportType,
  IN USAGE   UsagePage,
  IN USHORT  LinkCollection  OPTIONAL,
  IN USAGE   Usage,
  IN LONG    UsageValue,
```

```
  IN PHIDP_PREPARSED_DATA  PreparsedData,
  IN OUT PCHAR  Report,
  IN ULONG      ReportLength
);
```

Формат функций на языке Delphi имеет вид:

```
function HidP_GetScaledUsageValue(ReportType: THIDPReportType; UsagePage:
TUsage; LinkCollection: Word; Usage: TUsage; var UsageValue: Integer;
PreparsedData: PHIDPPreparsedData; var Report; ReportLength: ULONG):
NTSTATUS; stdcall;

function HidP_SetScaledUsageValue(ReportType: THIDPReportType; UsagePage:
TUsage; LinkCollection: Word; Usage: TUsage; UsageValue: Integer; Prepar-
sedData: PHIDPPreparsedData; var Report; ReportLength: ULONG): NTSTATUS;
stdcall;
```

Пример использования этих функций см. в разд 10.8.1.

17.21. Функция *HidP_MaxUsageListLength: размер буфера для кодов клавиш*

Функция `HidP_MaxUsageListLength` возвращает размер буфера, необходимый для получания кодов клавиш с помощью функции `HidP_GetUsages`.

Формат функции на языке C имеет вид:

```
ULONG  HidP_MaxUsageListLength(
  IN HIDP_REPORT_TYPE  ReportType,
  IN USAGE  UsagePage  OPTIONAL,
  IN PHIDP_PREPARSED_DATA  PreparsedData
);
```

Формат функции на языке Delphi имеет вид:

```
function HidP_MaxUsageListLength(
    ReportType: THIDPReportType; UsagePage: TUsage;
        PreparsedData: PHIDPPreparsedData): ULONG; stdcall;
```

Пример вызова см. в листинге 10.23.

17.22. Функция *HidP_UsageListDifference*: различие между массивами

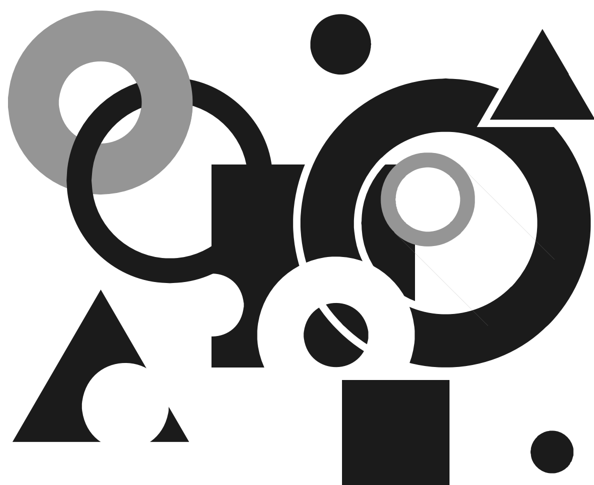
Функция `HidP_UsageListDifference` возвращает различие между двумя массивами.

Формат функции на языке C имеет вид:

```
NTSTATUS HidP_UsageListDifference(
  IN PUSAGE  PreviousUsageList,
  IN PUSAGE  CurrentUsageList,
  OUT PUSAGE  BreakUsageList,
  OUT PUSAGE  MakeUsageList,
  IN ULONG  UsageListLength
);
```

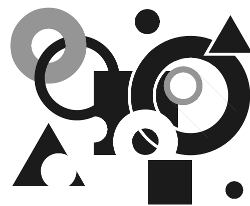
Формат функции на языке Delphi имеет вид:

```
function HidP_UsageListDifference(PreviousUsageList: PUsage;
  CurrentUsageList: PUsage; BreakUsageList: PUsage;
  MakeUsageList: PUsage; UsageListLength: ULONG
): NTSTATUS; stdcall;
```



Приложения

Приложение 1

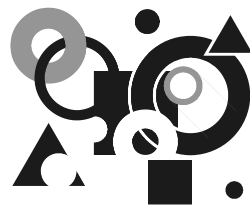


Дополнительные функции

Для преобразования BCD-чисел в строку номера версии мы используем очень простую функцию, код которой показан в листинге П1.1

Листинг П1.1 Преобразование BCD-числа в строку

```
function BCD2Str(Value : Word) : String;
begin
 Result:=
          IntToHex(Value shr 8, 2) +   // старшая часть версии
          '.' +                        // разделитель
         IntToHex(Value and $00FF, 2); // младшая часть версии
end;
```



Приложение 2

Компиляция примеров в других версиях Delphi

Весь рассмотренный в книге код на языке Delphi разрабатывался в среде Borland Delphi 6. Однако мы не использовали никаких возможностей, специфических именно для этой версии. Для компиляции примеров в других версиях Delphi потребует минимальных изменений, обусловленных изменениями в самом языке.

Для Borland Delphi версий 3—5 необходимо будет сменить тип `Cardinal` на `Integer`, а для Borland Delphi 7 — на `DWORD` и использовать ссылки на эти переменные (например, `@Data` или `@buffersize`).

Для Borland Delphi 8 потребуется добавить опцию `{$UNSAFECODE ON}` для разрешения неуправляемого кода, а для каждой функции, использующей указатели, добавить аттрибут `unsafe`.

Приложение 3

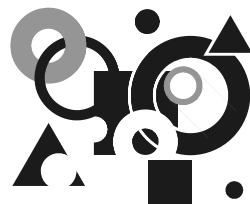


Таблица идентификаторов языков (LangID)

Список идентификаторов содержится в спецификации `USB_LANGID`. Актуальный на данный момент список приведен в табл. П3.1.

Таблица П3.1. *Таблица идентификаторов языков*

ID	Язык	ID	Язык	ID	Язык
`0x041c`	Albanian	`0x3409`	English (Philippines)	`0x0446`	Punjabi
`0x0401`	Arabic (Saudi Arabia)	`0x0425`	Estonian	`0x0418`	Romanian
`0x0801`	Arabic (Iraq)	`0x0438`	Faeroese	`0x0419`	Russian
`0x0c01`	Arabic (Egypt)	`0x0429`	Farsi	`0x044f`	Sanskrit.
`0x1001`	Arabic (Libya)	`0x040b`	Finnish	`0x0c1a`	Serbian (Cyrillic)
`0x1401`	Arabic (Algeria)	`0x040c`	French (Standard)	`0x081a`	Serbian (Latin)
`0x1801`	Arabic (Morocco)	`0x080c`	French (Belgian)	`0x0459`	Sindhi
`0x1c01`	Arabic (Tunisia)	`0x0c0c`	French (Canadian)	`0x041b`	Slovak
`0x2001`	Arabic (Oman)	`0x100c`	French (Switzerland)	`0x0424`	Slovenian
`0x2401`	Arabic (Yemen)	`0x140c`	French (Luxembourg)	`0x040a`	Spanish (Traditional Sort)
`0x2801`	Arabic (Syria)	`0x180c`	French (Monaco)	`0x080a`	Spanish (Mexican)

Таблица П3.1 *(продолжение)*

ID	Язык	ID	Язык	ID	Язык
0x2c01	Arabic (Jordan)	0x0437	Georgian.	0x0c0a	Spanish (Modern Sort)
0x3001	Arabic (Lebanon)	0x0407	German (Standard)	0x100a	Spanish (Guatemala)
0x3401	Arabic (Kuwait)	0x0807	German (Switzerland)	0x140a	Spanish (Costa Rica)
0x3801	Arabic (U.A.E.)	0x0c07	German (Austria)	0x180a	Spanish (Panama)
0x3c01	Arabic (Bahrain)	0x1007	German (Luxembourg)	0x1c0a	Spanish (Dominican Republic)
0x4001	Arabic (Qatar)	0x1407	German (Liechtenstein)	0x200a	Spanish (Venezuela)
0x042b	Armenian.	0x0408	Greek	0x240a	Spanish (Colombia)
0x044d	Assamese.	0x0447	Gujarati.	0x280a	Spanish (Peru)
0x042c	Azeri (Latin)	0x040d	Hebrew	0x2c0a	Spanish (Argentina)
0x082c	Azeri (Cyrillic)	0x0439	Hindi.	0x300a	Spanish (Ecuador)
0x042d	Basque	0x040e	Hungarian	0x340a	Spanish (Chile)
0x0423	Belarusian	0x040f	Icelandic	0x380a	Spanish (Uruguay)
0x0445	Bengali.	0x0421	Indonesian	0x3c0a	Spanish (Paraguay)
0x0402	Bulgarian	0x0410	Italian (Standard)	0x400a	Spanish (Bolivia)
0x0455	Burmese	0x0810	Italian (Switzerland)	0x440a	Spanish (El Salvador)
0x0403	Catalan	0x0411	Japanese	0x480a	Spanish (Honduras)
0x0404	Chinese (Taiwan)	0x044b	Kannada.	0x4c0a	Spanish (Nicaragua)
0x0804	Chinese (PRC)	0x0860	Kashmiri (India)	0x500a	Spanish (Puerto Rico)

Таблица П3.1 *(окончание)*

ID	Язык	ID	Язык	ID	Язык
0x0c04	Chinese (Hong Kong SAR, PRC)	0x043f	Kazakh	0x0430	Sutu
0x1004	Chinese (Singapore)	0x0457	Konkani.	0x0441	Swahili (Kenya)
0x1404	Chinese (Macau SAR)	0x0412	Korean	0x041d	Swedish
0x041a	Croatian	0x0812	Korean (Johab)	0x081d	Swedish (Finland)
0x0405	Czech	0x0426	Latvian	0x0449	Tamil.
0x0406	Danish	0x0427	Lithuanian	0x0444	Tatar (Tatarstan)
0x0413	Dutch (Netherlands)	0x0827	Lithuanian (Classic)	0x044a	Telugu.
0x0813	Dutch (Belgium)	0x042f	Macedonian	0x041e	Thai
0x0409	English (United States)	0x043e	Malay (Malaysian)	0x041f	Turkish
0x0809	English (United Kingdom)	0x083e	Malay (Brunei Darussalam)	0x0422	Ukrainian
0x0c09	English (Australian)	0x044c	Malayalam.	0x0420	Urdu (Pakistan)
0x1009	English (Canadian)	0x0458	Manipuri	0x0820	Urdu (India)
0x1409	English (New Zealand)	0x044e	Marathi.	0x0443	Uzbek (Latin)
0x1809	English (Ireland)	0x0861	Nepali (India).	0x0843	Uzbek (Cyrillic)
0x1c09	English (South Africa)	0x0414	Norwegian (Bokmal)	0x042a	Vietnamese
0x2009	English (Jamaica)	0x0814	Norwegian (Nynorsk)	0x04ff	HID (Usage Data Descriptor)
0x2409	English (Caribbean)	0x0448	Oriya.	0xf0ff	HID (Vendor Defined 1)
0x2809	English (Belize)	0x0415	Polish	0xf4ff	HID (Vendor Defined 2)
0x2c09	English (Trinidad)	0x0416	Portuguese (Brazil)	0xf8ff	HID (Vendor Defined 3)
0x3009	English (Zimbabwe)	0x0816	Portuguese (Standard)	0xfcff	HID (Vendor Defined 4)

Приложение 4

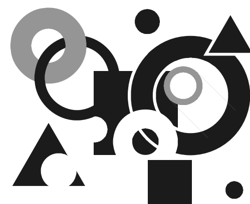


Таблица кодов производителей (Vendor ID, Device ID)

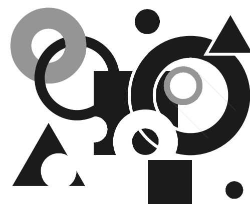
В табл. П4.1 приведены значения некоторых производителей (Vendor ID) и классов устройств (Device ID).

Таблица П4.1. *Некоторые коды Vendor ID и Device ID*

Vendor ID	Компания	Device ID	Устройство
0E11	Compaq Computer Corp.	11C0	HEWLETT PACKARD
100B	National Semiconductor	11C1	AT&T Microelectronics (Lucent)
1008	Epson	8086	Intel Corporation
1014	IBM	7020	USB Controller
100A	Phoenix Technologies	A0F8	USB Open Host Controller
101C	Western Digital	0012	USB Controller
1022	Advanced Micro Devices (AMD)	7404	AMD 755 PCI to USB Open Host Controller
1023	Trident Microsystems	740C	AMD 756 USB Controller
1025	Acer Labs Incorporated (ALI)	5237	M5237 PCI USB Host Controller
1028	Dell Computer Corp	0035	uPD9210FGC-7EA USB Host Controller
1029	Siemens Nixdorf AG	7001	SiS5571 USB Host Controller
102B	Matrox Graphics Inc	A0F8	82C750 PCI USB Controller
1032	Compaq	C861	82C861 FireLink PCI-to-USB Bridge
1033	NEC Electronics Hong Kong	0670	USB0670 USB Controller

Таблица П4.1 *(окончание)*

Vendor ID	Компания	Device ID	Устройство
1039	Silicon Integrated Systems (SiS)	0673	USB0673 USB Controller
103C	Hewlett-Packard Company	5237	ALI M5237 USB Host Controller
1043	Asustek Computer Inc	3038	VT83C572 USB Controller
1045	OPTi Inc	0571	VIA AMD-645 USB controller
104C	Texas Instruments (TI)	5801	USB Open Host Controller
104D	Sony Corp	2412	82801AA USB Controller
106B	Apple Computer Inc	2422	82801AB USB Controller
1075	Advanced Integration Research	2428	82801AB Hub Interface-to-PCI Bridge
1076	Chaintech Computer Co Ltd	2442	82801BA USB Controller
1095	CMD Technology Inc	2444	82801BA USB Controller
10B7	3COM Corp	244E	82801BA Hub Interface to PCI Bridge
10B9	Acer Labs Incorporated (ALI)	7020	82371SB PIIX3 USB Host Controller (Triton II)
1106	VIA Technologies Inc	7112	82371AB PIIX4 USB Interface
1114	Atmel Corp	719A	82443MX USB Universal Host Controller
1121	Zilog	7602	82372FB PCI to USB Universal Host Controller
1131	Philips Semiconductors		



Приложение 5

Как создать ярлык Device Manager

Запустить Менеджер Устройств можно, выбрав по правой кнопке мыши пункт меню **Свойства**. Но удобнее иметь его ярлык на рабочем столе. Запускаемый файл имеет имя devmgmt.msc и располагается в каталоге \Windows\system32\ (рис. П5.1).

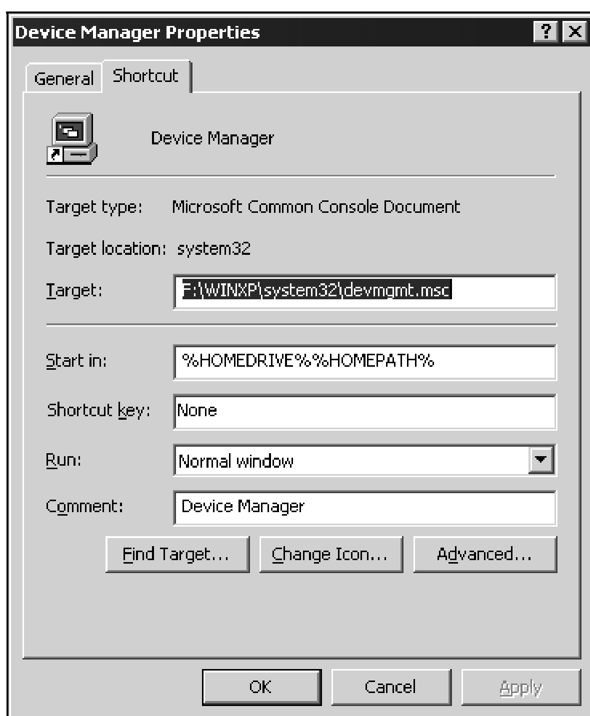
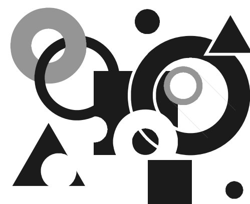


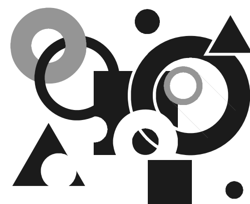
Рис. П5.1. Создание ярлыка Менеджера Устройств

Приложение 6



Часто задаваемые вопросы

- **Собрал устройство, включил, а Windows его не видит.**
  - Если переключение в режим программирования (раздел xxx) выполнено правильно, скорее всего, проблема в подключении линий D+ и D−.
- **Программа записана, но при включении устройство не опознается.**
  - Проверьте правильность настройки частоты кварцевого генератора (константы `PLL_xxx`) в функции `main` и реального кварцевого генератора.
- **Примеры не компилируются в Borland Delphi 7 и выше.**
  - *См. прил. 2.*
- **Как найти все COM-порты в системе?**
  - *См. функцию `EnumComPorts` из листинга 9.13.*
- **Хочу программировать USB напрямую по портам, как COM. Только не знаю как.**
  - И не надо. Для этого есть специальные функции, описанные в этой книге и в [4].
- **HID-устройство отображается в Менеджере Устройств Windows как работающее не корректно.**
  - Проверьте соответствие длины HID-репорта, указанной в дескрипторе конфигурации и реальной длины дескриптора (константа `SIZE_OF_REPORT` в примерах нашей книги).
- **Добавил конечные точки в дескриптор устройства, инициализирую их, а устройство не опознается Windows.**
  - Проверьте правильность константы размера дескриптора (константа `CONF_LENGTH` в примерах нашей книги).
- **Почему USB-мышь и клавиатура не открываются с помощью CreateFile?**
  - *См. прим. в разд. 6.4.1.*



Приложение 7

Описание компакт-диска

Папки	Описание	Разделы
IMessageFilter	Пример обработки сообщений Win32 в .NET с помощью IMessageFilter	4.5
AT89-BASE	Базовый проект для всех примеров	8.5
AT89-CDC	Реализация CDC-устройства	9.1—9.2
CDC-Read.Delphi	Программа обмена данными с CDC-устройством (Delphi)	9.4
CDC-Read.NET	Программа обмена данными с CDC-устройством (C#)	9.5
MSCOMM	Пример использования ActiveX-компонента MSCOMM (C#)	9.5.1
AT89-HID-int1	Реализация HID-устройства, передача 1 байта	10.1
AT89-HID-int7	Реализация HID-устройства, передача 7 байт	10.2
AT89-HID-feature	Реализация HID-устройства, использование Feature-протокола	10.3
AT89-HID-setreport	Реализация HID-устройства, передача данных от хоста к устройству	10.4
HIDTest	Пример работы с HID-функциями (Delphi)	10.6
AT89-HID-mouse	Реализация HID-мыши	10.7.1
AT89-HID-keyb	Реализация HID-клавиатуры	10.7.2
AT89-HID-minmax1	Реализация HID-устройства, интерпретация данных (byte)	10.8.1
AT89-HID-minmax2	Реализация HID-устройства, интерпретация данных (word)	10.8.1

(окончание)

Папки	**Описание**	**Разделы**
AT89-HID-minmax4	Реализация HID-устройства, интерпретация данных (long)	10.8.1
AT89-HID-coll	Реализация HID-устройства, вложенные коллекции	10.8.2
AT89-HID-buttons	Реализация HID-устройства, массив кнопок	10.8.2
Enumerator.Delphi	Нумерация USB-устройств (Delphi)	11.1, 11.2
Enumerator.NET	Нумерация USB-устройств (C#)	11.1, 11.2
HIDEnumerator	Нумерация HID-устройств (C#)	11.1
DeviceStatus	Получение состояния устройства	11.1.2
DirectInput-Scancodes	Чтение сканкодов с помощью DirectInput	11.4
DirectX-DeviceList	Получение списка устройств с помощью DirectX	11.4
DirectX-ReadObjectList	Чтение списка объектов устройства	11.4
StartAddHardware	Программный запуск процедуры добавления устройств	11.5
DeviceName	Работа с символьными именами	11.6
DeviceEject.Delphi	Безопасное извлечение устройств (Delphi)	11.7
DeviceEject.NET	Безопасное извлечение устройств (C#)	11.7
DeviceMonitor.Delphi	Обнаружение добавления или удаления устройств (Delphi)	11.8
DeviceMonitor.NET	Обнаружение добавления или удаления устройств (C#)	11.8
WDMTest	Пример реализации WDM-драйвера	12.3

Литература

1. Axelson Jan. USB Complete. — Lakeview Research, 2001.
2. John Hyde. USB Design by Example. A Practical Guide to Builing I/O Devices. — Intel Press, 2001.
3. Агуров П. В. Интерфейс USB. Практика использования и программирования. — СПб.: БХВ-Петербург, 2004.
4. Агуров П. В. Последовательные интерфейсы. Практика программирования. — СПб.: БХВ-Петербург, 2004.
5. Гук М. Аппаратные интерфейсы IBM PC. Энциклопедия. — СПб.: Питер, 2002.
6. Гук М. Аппаратные средства IBM PC. Энциклопедия. — СПб.: Питер, 2002.
7. Рихтер Д. Windows для профессионалов. — Microsoft Press, 1999.
8. Рихтер Д. Программирование на платформе Microsoft .NET Framework Windows для профессионалов. — Microsoft Press, 2003.
9. Солдатов В. П. Программирование драйверов Windows. — Москва: Бином, 2004.
10. Соломон Д., Руссанович М. Внутреннее устройство Windows 2000. Структура и алгоритмы работы компонентов Windows 2000 и NTFS 5. — Питер, Microsoft Press, 2001.

Предметный указатель

A

Abstract Control Management Descriptor 138
ACK 38

B

Boot Device 148

C

Call Management Descriptor 137
CDC 133
Control Read 40
Control Write 40
CRC 30, 32

D

DCE 128
Descriptor:
 configuration 56
 device 52
 endpoint 60
 interface 59
 qualifier 55
 report 147
 standard device 52
 string 63
DTE 128

F

FDO 75
FiDO 75
Functional Descriptor 135

H

Header Functional Descriptor 136
HID 64, 146
 коллекция 320

I

IN 29
IRP 17, 386

L

LANGID 64

N

NAK 38

O

OUT 29

P

PDO 75
PID 27
PING 29
Pipe 26

R

Raw Input 348

S

SETUP 29
Setup Packet 42
SOF 30
STALL 38
SYNC 27

U

Union Interface Functional
Descriptor 138

W

WDM 74

А

Адрес устройства 18, 70

Д

Дескриптор:
HID 151, 290
абстрактного устройства 138
группирования 138
заголовочный функциональный 136
интерфейса 59
конечной точки 60, 231
конфигурации 56
отображение 68
порта 142
порядок получения 65
режима команд 137
репорта 147, 153
специфический 64
стандартный 52
строки 63, 226
тип 44
устройства 52
уточняющий 55
функциональный 135
Драйвер:
получение списка 76
фильтра 75
функциональный 74
шины 74
USB 18

З

Запрос:
CLEAR_FEATURE 47
GET_CONFIGURATION 49, 218
GET_DESCRIPTOR 48, 150, 213
GET_IDLE 165, 167
GET_INTERFACE 50
GET_LINE_CODING 140
GET_PROTOCOL 166, 168
GET_REPORT 147, 165, 166
GET_STATUS 46, 219
IRP_MJ_xxx 393
SEND_BREAK 141
SET_ADDRESS 48, 221
SET_CONFIGURATION 50, 218
SET_CONTROL_LINE_STATE 140
SET_DESCRIPTOR 49
SET_FEATURE 47
SET_IDLE 166, 168
SET_INTERFACE 50
SET_LINE_CODING 139
SET_PROTOCOL 166, 169
SET_REPORT 147, 166, 167
SYNC_FRAME 51
на обработку 51
стандартный 43

И

Идентификатор пакета 27
Идентификация HID 149

Интерфейс 27, 58
Интерфейс USB:
  архитектура 11
  возможности 10
  логическая архитектура 13
  логические уровни 16
  механизм передачи данных 15
  свойства 9
  физическая архитектура 11

К

Кадр 15, 24
Канал 15, 26
Канал сообщений 26
Класс:
  CDC 133, 142
  DeviceObjectList 360
  Manager.Device 358
Коллекция 320
Конечная точка 15, 25
  атрибуты 62
  блокировка 47
  максимальное число точек 25
  нулевая 25
  определение 13
Контрольная сумма 30, 32
Концентратор 13
Корневой хаб 11

Л

Логическое устройство 18

М

Маркер. *См. Пакет*
Микрокадр 24

Н

Нуль-модемный кабель 129
Нумерация 65

О

Основной канал сообщений 27

П

Пакет 27
  Data0 30
  Data1 30
  Data2 30
  IN 29
  IRP 17, 386
  MData 30
  OUT 29
  PING 29
  SETUP 29, 37, 42
  SOF 30
  SPLIT 31
  квитирование 38
  маркер 37
  подтверждение 31
Передача:
  данных с подтверждением 38
  изохронная 22, 38, 42
  массивов данных 22
  по прерыванию 41
  по прерываниям 22
  прием с подтверждением 38
  приоритеты 23
  управляющая 21, 37
Побудка 14, 47
Поле bmRequestType 43
Полоса пропускания 17
Порт 13
  восходящий 13
  нисходящий 13
Посылка:
  без данных 40
  запись данных 40
  чтение данных 40
Поток 26
Прерывания 15
Приостановка 14
Процедура:
  AddDevice 382
  DriverEntry 379
  Unload 384
  рабочая 388

Р

Разрыв связи 532
Регистр AT89C5131
UBYCTLX 201
UEPCONX 193
UEPDATX 200
UEPIEN 199
UEPINT 198
UEPNUM 192
UEPRST 197
UEPSTAX 195
UFNUMH 201
UFNUML 201
USBADDR 189
USBCON 187
USBIEN 191
USBINT 190
Режимы передачи данных 15
Репорт:
Feature 148, 298
Input 147
Output 147
номер 149
определение 147

С

Сигнал:
готовность DCE (CC/DSR) 132
готовность DTE (CD/DTR) 132
готовность к передаче (CB/CTS) 131
готовность к приему (CJ) 133
запрос передачи (CA/RTS) 130
индикатор вызова (CE/RI) 132
обнаружение несущей (CF/DCD) 132
передаваемые данные (BA/TxD/TD) 130
принимаемые данные (BB/RxD/RD) 130
Символьное имя 73
Скорость обмена:
задание в Windows 523
максимально допустимая 513
Слово состояния:
интерфейса 46
конечной точки 46
устройства 46
Спецификация:
CDC 133
HID 146
Структура:
COMMCONFIG 143, 509, 510, 541, 550
COMMPROP 144, 511
COMMTIMEOUTS 143, 518, 531, 546
COMSTAT 520, 533, 551
DCB 143, 522, 529, 531, 544
MODEMDEVCAPS 517
OVERLAPPED 489
TWMDeviceChange 374

Т

Транзакция 37
планирование 18

У

Устройство 13
HID 64, 146
загрузочное 148, 150
логическое 14
основное состояние 71
свойства 14
типы 77

Ф

Файл INF 149
структура 467
Фрейм 15
Функция 14
BuildCommDCB 143, 528, 531
BuildCommDCBAndTimeouts 143, 531
CancelIo 504
ClearCommBreak 144, 531
ClearCommError 521, 532, 538
CloseHandle 142, 484
CM_Get_DevNode_Status 342
CommConfigDialog 143, 510, 542, 550
CreateEvent 537

CreateFile 142, 169, 305, 484
DefineDosDevice 364, 507
DeviceIoControl 67, 393, 502
EnumPorts 553
EscapeCommFunction 534
GetCommConfig 143, 510, 540
GetCommMask 535
GetCommModemStatus 144, 537, 552
GetCommProperties 144, 511, 544
GetCommState 530, 544
GetCommTimeouts 518, 546
GetDefaultCommConfig 143, 543, 550
GetOverlappedResult 489, 501, 537
GetRawInputData 351
GetRawInputDeviceInfoA 356
GetRegistryProperty 341
HidD_FlushQueue 585
HidD_FreePreparsedData 310, 571
HidD_GetAttributes 170, 305, 575
HidD_GetFeature 171, 305, 571
HidD_GetHidGuid 170, 302, 569
HidD_GetIndexedString 311, 579
HidD_GetInputReport 347, 580
HidD_GetManufacturerString 305, 576
HidD_GetNumInputBuffers 573
HidD_GetPreparsedData 170, 305, 570
HidD_GetProductString 305, 578
HidD_GetSerialNumberString 578
HidD_Hello 568
HidD_SetFeature 171, 573
HidD_SetNumInputBuffers 574
HidD_SetOutputReport 348, 581
HidP_GetCaps 171, 305, 581
HidP_GetLinkCollectionNodes 321, 585
HidP_GetScaledUsageValue 586
HidP_GetUsages 328
HidP_MaxDataListLength 584
HidP_MaxUsageListLength 328, 587
HidP_SetScaledUsageValue 586
HidP_UsageListDifference 587
PurgeComm 547
QueryDosDevice 364, 505
ReadFile 142, 305, 327, 488, 538
ReadFileEx 142, 489, 494
RegisterRawInputDevices 349
SetCommBreak 144, 531
SetCommConfig 143, 510, 540
SetCommMask 535, 536
SetCommState 143, 529, 544
SetCommTimeouts 143, 518, 546
SetDefaultCommConfig 143, 543, 550
Setup API 332
SetupComm 144, 548
SetupDiDestroyDeviceInfoList 333
SetupDiEnumDeviceInfo 333
SetupDiGetClassDevs 302, 333, 339
SetupDiGetDeviceInterfaceDetail 302
SetupDiGetDeviceRegistryProperty 333, 337
TransmitCommChar 551
WaitCommEvent 535, 536
WaitForMultipleObjects 499
WaitForSingleObject 498, 537
Windows для работы с портами 142
WMI 345
WriteFile 142, 491
WriteFileEx 142, 492, 496

Х

Хаб:
  корневой 11, 14
  определение 11, 13
Хост 11
Хост-контроллер 13

Э

Энергопотребление 58