А. Б. Григорьев

bhv

О чем не пишут в книгах по Delphi

+cd

Королевство Delphi

Виртуальный клуб программистов

Интеграция Windows API и VCL
Использование Windows Sockets API в Delphi
Неочевидные особенности целых чисел, вещественных чисел, строк
Коллекция коварных ошибок
Разбор и вычисление арифметических выражений в Delphi
Особенности различных версий от Delphi 3 до Delphi 2007

А. Б. Григорьев

О чем не пишут в книгах по Delphi

Санкт-Петербург
«БХВ-Петербург»
2008

УДК 681.3.068+800.92Delphi
ББК 32.973.26-018.1
Г83

Григорьев А. Б.

Г83 О чем не пишут в книгах по Delphi. — СПб.: БХВ-Петербург, 2008. — 576 с.: ил. + CD-ROM

ISBN 978-5-9775-0190-3

Рассмотрены малоосвещенные вопросы программирования в Delphi. Описаны методы интеграции VCL и API. Показаны внутренние механизмы VCL и приведены примеры вмешательства в эти механизмы. Рассмотрено использование сокетов в Delphi: различные режимы их работы, особенности для протоколов TCP и UDP и др. Большое внимание уделено разбору ситуаций возникновения ошибок и получения неверных результатов в "простом и правильном" коде. Отдельно рассмотрены особенности работы с целыми, вещественными и строковыми типами данных, а также приведены примеры неверных результатов, связанных с ошибками компилятора, VCL и др. Для каждой из таких ситуаций предложены методы решения проблемы. Подробно рассмотрен синтаксический анализ в Delphi на примере арифметических выражений. Многочисленные примеры составлены с учетом различных версий: от Delphi 3 до Delphi 2007. Прилагаемый компакт-диск содержит примеры из книги.

Для программистов

УДК 681.3.068+800.92Delphi
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Леонид Кочин*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Инны Тачиной*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 11.12.07.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 46,44.
Тираж 2500 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 978-5-9775-0190-3

© Григорьев А. Б., 2008
© Оформление, издательство "БХВ-Петербург", 2008

Отпечатано с готовых диапозитивов
в ОАО "Техническая книга"
190005, Санкт-Петербург, Измайловский пр., 29

Отпечатано с готовых диапозитивов
в СПб ГУП "Петроцентр" ОП "Пушкинская типография"
196601, Санкт-Петербург, г. Пушкин, ул. Средняя, 3/8

Отпечатано с готовых диапозитивов в ФГУП "Печатный двор"
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

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

Отпечатано с готовых диапозитивов
в ГППО "Псковская областная типография"
180007, г. Псков, Рижский пр., 17

Отпечатано по технологии CtP
в ОАО «Печатный двор» им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

Оглавление

Введение **9**

Глава 1. Windows API и Delphi **13**

1.1. Основы работы с Windows API в VCL-приложениях 13
1.1.1. Что такое Windows API 14
1.1.2. Как получить справку по функциям Windows API 15
1.1.3. Дескрипторы вместо классов 26
1.1.4. Формы VCL и окна Windows 27
1.1.5. Функции обратного вызова 32
1.1.6. Сообщения Windows 36
1.1.7. Создание окон средствами VCL 46
1.1.8. Обработка сообщений с помощью VCL 51
1.1.9. Сообщения, определяемые пользователем 70
1.1.10. Особые сообщения 72
1.1.11. Графика в Windows API 74
1.1.12. ANSI и Unicode 82
1.1.13. Строки в Windows API 84
1.2. Примеры использования Windows API 89
1.2.1. Пример EnumWnd 89
1.2.2. Пример Line 97
1.2.3. Пример CoordLabel 106
1.2.4. Пример PanelMsg 108
1.2.5. Пример NumBroadcast 113
1.2.6. Пример ButtonDel 117
1.2.7. Пример GDIDraw 120
1.2.8. Пример BitmapSpeed 131
1.3. Обобщающие примеры 137
1.3.1. Обобщающий пример 1 — Информация о процессах 137
1.3.1.1. Получение списка процессов 137
1.3.1.2. Получение списка и свойств окон 140

1.3.2. Обобщающий пример 2 — Ассоциированные файлы и предотвращение запуска второй копии приложения 145
1.3.2.1. Ассоциирование расширения с приложением 146
1.3.2.2. Командная строка 148
1.3.2.3. Поиск уже запущенной копии приложения 150
1.3.2.4. Перевод приложения на передний план 154
1.3.3. Обобщающий пример 3 — "Дырявое" окно 156
1.3.3.1. Сообщение *WM_NCHCHITTEST* 156
1.3.3.2. Регионы 158
1.3.3.3. Сообщения *WM_SIZE* и *WM_SIZING* 158
1.3.3.4. А теперь — все вместе 159
1.3.4. Обобщающий пример 4 — Линии нестандартного стиля 168
1.3.4.1. Получение координат точек прямой 168
1.3.4.2. "Резиновая" линия и растровые операции 171
1.3.4.3. Кривые Безье 173
1.3.4.4. Траектории 174
1.3.4.5. Интерактивная кривая 179

Глава 2. Использование сокетов Delphi 187

2.1. Стандартные сокеты 188
2.1.1. Соглашения об именах 189
2.1.2. Общие сведения о сокетах 190
2.1.3. Сетевые протоколы. Семиуровневая модель OSI 192
2.1.4. Стек TCP/IP 193
2.1.5. Протокол UDP 197
2.1.6. Протокол TCP 199
2.1.7. Сетевые экраны 203
2.1.8. Создание сокета 204
2.1.9. Передача данных при использовании UDP 215
2.1.10. Пример программы: простейший чат на UDP 220
2.1.11. Передача данных при использовании TCP 230
2.1.12. Примеры передачи данных с помощью TCP 237
2.1.13. Определение готовности сокета 252
2.1.14. Примеры использования функции *select* 260
2.1.15. Неблокирующий режим 268
2.1.16. Сервер на неблокирующих сокетах 272
2.1.17. Параметры сокета 282
2.1.18. Итоги первого раздела 285
2.2. Сокеты Windows 286
2.2.1. Версии Windows Sockets 286
2.2.2. Устаревшие функции WinSock 1 288
2.2.3. Информация о протоколе 290
2.2.4. Новые функции 294
2.2.5. Асинхронный режим, основанный на сообщениях 304
2.2.6. Пример сервера, основанного на сообщениях 312

2.2.7. Асинхронный режим, основанный на событиях 325
2.2.8. Пример использования сокетов с событиями 334
2.2.9. Перекрытый ввод-вывод 358
2.2.10. Сервер, использующий перекрытый ввод-вывод 371
2.2.11. Многоадресная рассылка 382
2.2.12. Дополнительные функции 388
2.3. Итоги главы 395

Глава 3. "Подводные камни" 397

3.1. Неочевидные особенности целых чисел 398
3.1.1. Аппаратное представление целых чисел 398
3.1.2. Выход за пределы диапазона при присваивании 401
3.1.3. Переполнение при арифметических операциях 403
3.1.4. Сравнение знакового и беззнакового числа 404
3.1.5. Неявное преобразование в цикле *for* 406
3.2. Неочевидные особенности вещественных чисел 407
3.2.1. Двоичные дроби 408
3.2.2. Вещественные типы Delphi 408
3.2.3. Внутренний формат вещественных чисел 411
3.2.4. "Неполноценный" *Extended* 412
3.2.5. Бесконечные дроби 414
3.2.6. "Неправильное" значение 415
3.2.7. Сравнение 416
3.2.8. Сравнение разных типов 417
3.2.9. Вычитание в цикле 418
3.2.10. Неожиданная потеря точности 418
3.2.11. Борьба с потерей точности в VCL 420
3.2.12. Машинное эпсилон 423
3.2.13. Методы решения проблем 424
3.3. Тонкости работы со строками 425
3.3.1. Виды строк в Delphi 425
3.3.2. Хранение строковых литералов 428
3.3.3. Приведение литералов к типу *PChar* 431
3.3.4. Сравнение строк 433
3.3.5. Побочное изменение 440
3.3.6. Нулевой символ в середине строки 442
3.3.7. Функция, возвращающая *AnsiString* 444
3.3.8. Строки в записях 445
3.3.9. Использование ShareMem 455
3.4. Прочие "подводные камни" 461
3.4.1. Порядок вычисления операндов 461
3.4.2. Зацикливание обработчика *TUpDown.OnClick* при открытии диалогового окна в обработчике 464
3.4.3. Access violation при закрытии формы с перекрытым методом *WndProc* 466

3.4.4. Подмена имени оконного класса, возвращаемого функцией *GetClassInfo* 471
3.4.5. Ошибка EReadError при использовании вещественных свойств....473
3.4.6. Ошибка List index out of bounds при корректном значении индекса....474
3.4.7. Неправильное поведение свойства *Anchors*....476
3.4.8. Ошибка при сравнении указателей на метод....478
3.4.9. Возможность получения адреса свойства....480
3.4.10. Невозможность использования некоторых свойств оконного компонента в деструкторе....481

Глава 4. Разбор и вычисление выражений 489

4.1. Синтаксис и семантика....489
4.2. Формальное описание синтаксиса....492
4.3. Синтаксис вещественного числа 495
4.4. Простой калькулятор 498
4.5. Учет приоритета операторов....504
4.6. Выражения со скобками 506
4.7. Полноценный калькулятор....511
4.8. Калькулятор с лексическим анализатором 517
4.9. Однопроходный калькулятор и функции с несколькими переменными 529
4.10. Еще немного теории 534

ПРИЛОЖЕНИЯ 539

Приложение 1. Сайт "Королевство Delphi" 541

Приложение 2. Содержимое компакт-диска 549

Примеры к главе 1....549
Примеры к главе 2....550
Примеры к главе 3....552
Примеры к главе 4....555

Список литературы 557

Предметный указатель 559

Введение

Среда программирования Delphi заслуженно приобрела свою популярность. Этот удобный и красивый инструмент, основанный на не менее красивом языке Паскаль, первым избавил программиста от рутинных операций, сохранив при этом гибкость, присущую универсальным языкам программирования. Не удивительно, что об этой среде написано много книг, а в Интернете можно найти множество различных сайтов, посвященных Delphi.

Автор данной книги с 1999 года является постоянным посетителем (а с 2004 года — еще и модератором) сайта "Королевство Delphi" (**www.delphikingdom.ru**; подробнее этот сайт описан в *приложении 1*), одного из самых известных русскоязычных ресурсов по Delphi. За это время изучено, какие вопросы интересуют посетителей сайта и какие ответы они хотели бы получить. Наблюдения показывают, что существует целый ряд тем, традиционно вызывающих большой интерес, но информацию по которым найти сложно. Авторы книг по Delphi стараются как можно быстрее перейти к описанию возможностей различных библиотек, оставляя в стороне другие важные вопросы.

"За бортом" остается множество тем, касающихся более низкоуровневых средств, которые в большинстве своем неплохо документированы и описаны в книгах, проиллюстрированы готовыми примерами, которые, правда, обычно даются на C++. Конечно, немного опыта — и код переведен с C++ на Delphi. Только обидно программировать в Delphi, никак не задействуя ее высокоуровневые библиотеки — хочется как-то "сшить" гибкость низкоуровневых вещей и удобство библиотек, использовав библиотеки везде, где это возможно, а низкоуровневый код — только там, где без него никак не обойтись. И вот как раз по этим вопросам и наблюдается острая нехватка информации.

Природа не терпит пустоты, поэтому в Интернете время от времени появляются рекомендации по поводу того, как же все-таки "впрячь в одну упряжку" библиотеки и низкоуровневый код. Но эти советы, за редким исключением,

страдают тем, что дают готовое решение с минимальными объяснениями, почему надо делать это именно так (иногда авторы этих рекомендаций при всем желании не смогли бы дать такое объяснение, т. к. сами дошли до этого "методом тыка"). Так что такие советы могут оказаться очень полезными в конкретной ситуации, но не дают общего понимания проблемы.

Эта книга призвана заполнить информационный вакуум по некоторым из таких вопросов. Но самое главное — это то, что мы здесь принципиально будем избегать изложения в стиле "делай так, и будет тебе счастье", т. е. уклон будет не в сторону готовых рецептов, а в сторону объяснения, как это все устроено, чтобы читатель потом сам был в состоянии искать решения для своих проблем. Для этого мы будем разбирать стандартные средства Delphi с позиции "а как оно все работает".

Глава 1 книги посвящена интеграции библиотеки VCL и Windows API, прежде всего той части Windows API, которая управляет окнами на экране. Не секрет, что в VCL отсутствуют многие возможности по управлению окнами, которые предоставляет операционная система Windows, но их можно задействовать и в VCL-приложении, если знать, как и куда можно вставить код, использующий API, так, чтобы это не нарушило работу VCL. В данной главе даются базовые сведения о Windows API и о том, как VCL использует API. Здесь приведен ряд примеров программ различной степени сложности, которые демонстрируют методы применения Windows API совместно с VCL в различных ситуациях. Особое внимание уделяется тому, как использовать Windows API, не нарушая работу VCL.

Глава 2 посвящена применению сокетов в Delphi. Сокеты — это самые низкоуровневые сетевые средства Windows, своего рода ассемблер сетевого программирования. Они хорошо документированы, но из-за обилия возможностей эта документация оказывается практически "неподъемной" для человека, который только-только начал знакомиться с сокетами и которому не нужны все эти возможности, а требуется просто научиться передавать данные с помощью TCP/IP. Книг по сокетам очень мало, а по использованию сокетов в Delphi — вообще ни одной. Между тем сокеты в Delphi имеют свою специфику из-за отличий языка C, на который ориентирована библиотека сокетов, и Delphi: макросы заменены функциями, изменены параметры некоторых функций, определения типов приспособлены к возможностям Delphi. Кроме того, стандартный модуль Delphi для импорта функций из библиотеки сокетов импортирует библиотеку не полностью и содержит некоторые неточности. Все это делает освоение сокетов "с нуля" очень сложным делом. Вторая глава книги ориентирована на человека, который не имеет опыта сетевого программирования и не знаком с терминологией. Даются все необходимые определения и пояснения, чтобы можно было полностью понять, как работа-

ют примеры, но при этом читатель не перегружается избыточной на данном этапе информацией. Попутно излагаются особенности работы с сокетами, присущие именно Delphi. После прочтения данной главы читатель получает достаточно полное представление о протоколах стека TCP/IP и об основных режимах работы сокетов, и после этого способен дальше читать документацию самостоятельно.

Глава 3 посвящена ситуациям, в которых стандартные средства ведут себя не так, как этого ожидает программист. Иногда это объясняется ошибками в реализации этих средств, но чаще тем, что программист просто не знает некоторых особенностей данной реализации. Конечно, детально изучив документацию, можно не только дать объяснение ошибкам, но и научиться предсказывать, где они могут возникнуть, но такой метод имеет очевидные сложности. В данной главе выбран другой подход — "от ошибки". Мы сначала будем рассматривать ситуацию, в которой возникает ошибка, а потом объяснять, какие особенности реализации приводят к этому. Такой порядок изложения позволяет одновременно и рассказать о том, как соответствующие средства реализованы, и предостеречь от неверного их использования.

Глава 4 целиком посвящена одному очень популярному в форумах вопросу: как вычислить арифметическое выражение, которое становится известным только во время выполнения программы (т. е., например, у нас есть арифметическое выражение в строковой переменной, а требуется вычислить его значение). С одной стороны, в Интернете можно найти немало самодельных библиотек, решающих эту задачу. Но не все из них работают достаточно корректно, потому что их авторы зачастую реализуют доморощенные методы анализа выражений, дающие в некоторых случаях сбои. С другой стороны, существует литература, в которой довольно полно изложена формальная теория создания трансляторов. Но эта теория выходит далеко за рамки простого вычисления арифметических выражений и требует соответствующих усилий для ее освоения. Мы же, с одной стороны, ограничимся только той частью теоретических сведений, которая необходима для построения вычислителя, но с другой, — будем этой теории строго следовать, создавая действительно работоспособное решение. И хотя на выходе мы получим несколько вполне работоспособных примеров, не они будут нашей главной целью. Мы будем стремиться к тому, чтобы читатель понял сам принцип построения таких анализаторов и при необходимости смог вносить в них изменения. В дальнейшем это поможет при изучении теории синтаксического анализа по специализированным книгам.

Книга ориентирована на Delphi для Win32. О Delphi для .NET мы здесь говорить не будем. При написании книги были исследованы особенности всех версий Delphi от 3-й до 2007-й, за исключением BDS 2005, и если какая-то

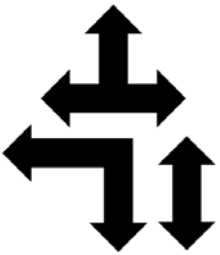
особенность или ошибка, описанная в данной книге, присутствует только в некоторых из этих версий, то это обстоятельство обязательно отмечается. Но из-за того, что с BDS 2005 автор книги не имел возможности ознакомиться, фраза "до Delphi 7 включительно" может означать "до BDS 2005" включительно, а фраза "начиная с BDS 2006" — "начиная с BDS 2005".

Автор надеется, что книга действительно окажется полезной читателю. Актуальность изложенных в ней тем подтверждается многочисленными вопросами в форумах, а полезность сведений — многочисленными ответами.

Благодарности

- Елене Филипповой, создательнице и бессменному главному администратору сайта "Королевство Delphi".
- Алексею Ковязину (российское отделение CodeGear) за помощь в выпуске этой книги.
- Юрию Зотову, натолкнувшему меня на идею четвертой главы и некоторых разделов первой и третьей глав.
- Многочисленным посетителям сайта "Королевство Delphi", при общении с которыми выяснялись интересные факты и рождались идеи, нашедшие себе место в этой книге.

ГЛАВА 1



Windows API и Delphi

Библиотека VCL, делающая создание приложений в Delphi таким быстрым и удобным, все же не позволяет разработчику задействовать все возможности операционной системы. Полный доступ к ним дает *API* (Application Programming Interface) — интерфейс, который система предоставляет программам. С его помощью можно получить доступ ко всем документированным возможностям системы.

Программированию в Windows на основе API посвящено много книг, а также материалов в Интернете. Но если все делать только с помощью API, то даже для того, чтобы создать пустое окно, потребуется написать несколько десятков строк кода, а о визуальном проектировании такого окна придется вообще забыть. Поэтому желательно как-то объединить мощность API и удобство VCL. О том, как это сделать, мы и поговорим в этой главе.

В первой части главы рассматриваются общие принципы использования API и интеграции этого интерфейса с VCL. Во второй части разбираются простые примеры, иллюстрирующие теорию. В третьей части представлено несколько обобщающих примеров использования API — небольших законченных приложений, использующих различные функции API для решения комплексных задач.

1.1. Основы работы с Windows API в VCL-приложениях

В данном разделе будет говориться о том, как совместить Windows API и компоненты VCL. Предполагается, что читатель владеет основными методами создания приложений с помощью VCL, а также синтаксисом языка Delphi, поэтому на этих вопросах мы останавливаться не будем. Так как "официаль-

ные" справка и примеры работы с API предполагают работу на C или C++, и это может вызвать трудности у человека, знакомого только с Delphi, здесь также будет уделено внимание тому, как правильно читать справку и переводить содержащийся в ней код с C/C++ на Delphi.

1.1.1. Что такое Windows API

Windows API — это набор функций, предоставляемых операционной системой каждой программе. Данные функции находятся в стандартных *динамически компонуемых библиотеках* (Dynamic Linked Library, DLL), таких как kernel32.dll, user32.dll, gdi32.dll. Указанные файлы располагаются в системной директории Window. Вообще говоря, каждая программа должна самостоятельно заботиться о том, чтобы подключить эти библиотеки. DLL могут подключаться к программе *статически* и *динамически*. В первом случае связь с библиотекой прописывается в исполняемом файле программы, и система при запуске этой программы сразу же загружает в ее адресное пространство и библиотеку. Если требуемая библиотека на диске не найдена, запуск программы будет невозможен. В случае динамического подключения программа загружает библиотеку в любой удобный для нее момент времени с помощью функции `LoadLibrary`. Если при этом возникает ошибка из-за того, что библиотека не найдена на диске, программа может самостоятельно решить, как на это реагировать.

Статическая загрузка проще динамической, но динамическая гибче. При динамической загрузке программист может, во-первых, выгрузить библиотеку, не дожидаясь окончания работы программы. Во-вторых, программа может продолжить работу, даже если библиотека не найдена. В-третьих, возможна загрузка тех DLL, имена которых неизвестны на момент компиляции. Это позволяет расширять функциональность приложения после передачи его пользователю с помощью дополнительных библиотек (в англоязычной литературе такие библиотеки обычно называются plugin).

Стандартные библиотеки необходимы самой системе и всем программам, они всегда находятся в памяти, и поэтому обычно они загружаются статически. Чтобы статически подключить в Delphi некоторую функцию Windows API, например, функцию `GetWindowDC` из модуля user32.dll, следует написать конструкцию вида

```
function GetWindowDC(Wnd: HWnd); HDC; stdcall;
  external 'user32.dll' name 'GetWindowDC';
```

В результате в специальном разделе исполняемого файла, который называется таблицей импорта, появится запись, что программа импортирует функцию `GetWindowDC` из библиотеки user32.dll. После такого объявления компилятор будет знать, как вызывать эту функцию, хотя ее реальный адрес будет внесен

в таблицу импорта только при запуске программы. Обратите внимание, что функция `GetWindowDC`, как и все функции Windows API, написана в соответствии с *моделью вызова* `stdcall`, а в Delphi по умолчанию принята другая модель — `register` (модель вызова определяет, как функции передаются параметры). Поэтому при импорте функций из стандартных библиотек необходимо явно указывать эту модель (подчеркнем, что это относится именно к стандартным библиотекам; другие библиотеки могут использовать любую другую модель вызова, разработчик библиотеки свободен в своем выборе). Далее указывается, из какой библиотеки импортируется функция и какое название в ней она имеет. Дело в том, что имя функции в библиотеке может не совпадать с тем, под которым она становится известной компилятору. Это может помочь разрешить конфликт имен при импорте одноименных функций из разных библиотек, а также встречается в других ситуациях, которые мы рассмотрим позже. Главным недостатком DLL следует считать то, что в них сохраняется информация только об именах функций, но не об их параметрах. Поэтому если при импорте функции указать не те параметры, которые подразумевались автором DLL, то программа будет работать неправильно (вплоть до зависания), а ни компилятор, ни операционная система не смогут указать на ошибку.

Обычно программе требуется много различных функций Windows API. Декларировать их все довольно утомительно. К счастью, Delphi избавляет программиста от этой работы: многие из этих функций уже описаны в соответствующих модулях, достаточно упомянуть их имена в разделе `uses`. Например, большинство общеупотребительных функций описаны в модулях `Windows` и `Messages`.

Функции API, которые присутствуют не во всех версиях Windows, предпочтительнее загружать динамически. Например, если программа статически импортирует функцию `SetLayeredWindowsAttributes`, она не запустится в Windows 9*x*, где этой функции нет — система, встретив ее упоминание в таблице импорта, прервет загрузку программы. Поэтому, если требуется, чтобы программа работала и в Windows 9*x*, эту функцию следует импортировать динамически. Отметим, что компоновщик в Delphi помещает в таблицу импорта только те функции, которые реально вызываются программой. Поэтому наличие декларации `SetLayeredWindowsAttributes` в модуле `Windows` не помешает программе запускаться в Windows 9*x*, если она не вызывает эту функцию.

1.1.2. Как получить справку по функциям Windows API

Для тех, кто решил работать с Windows API, самым необходимым инструментом становится какая-либо документация по этим функциям. Их так мно-

го, что запомнить все совершенно нереально, поэтому работа без справочника под рукой просто невозможна.

Первоисточник информации по технологиям Microsoft для разработчика — *Microsoft Developer's Network* (MSDN). Это отдельная справочная система, не входящая в комплект поставки Delphi. MSDN можно приобрести отдельно или воспользоваться online-версией, находящейся по адресу: **http://msdn.microsoft.com** (доступ к информации свободный, регистрация не требуется). MSDN содержит не только информацию об API, но и все, что может потребоваться программисту, использующему различные средства разработки от Microsoft. Кроме справочного материала, MSDN включает в себя спецификации стандартов и технологий, связанных с Windows, статьи из журналов, посвященных программированию, главы из некоторых книг. И вся эта информация крайне полезна разработчику. Кроме того, MSDN постоянно обновляется, информация в нем наиболее актуальна. Пример справки из MSDN показан на рис. 1.1.

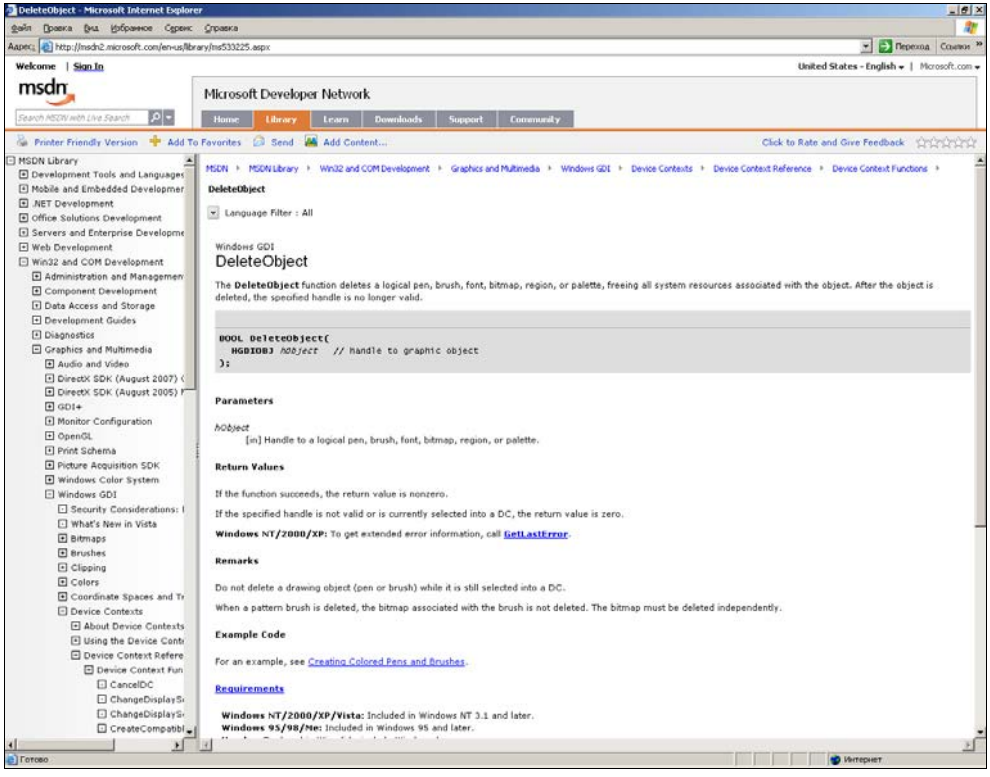


Рис. 1.1. Online-вариант MSDN (показана справка по функции `DeleteObject`)

Примечание

Отметим, что MSDN содержит также описание функций операционной системы Windows CE. Интерфейс Windows CE API на первый взгляд очень похож на Windows API, но различия между ними есть, и иногда весьма значительные. Поэтому при использовании MSDN не следует выбирать раздел API Reference — он целиком посвящен WinCE API.

В комплект поставки Delphi входит справочная система, содержащая описание функций Windows API. Справочная система в Delphi до 7-й версии включительно была построена на основе hlp-файлов. Применительно к справке по Windows API это порождало две проблемы. Во-первых, hlp-файлы имеют ограничение по числу разделов в справочной системе, поэтому объединить в одной справке информацию и по Delphi, и по Windows API было невозможно, — эти две справки приходилось читать по очереди. Чтобы открыть файл справки по Windows API, нужно было в редакторе кода поставить курсор на название какой-либо функции API и нажать клавишу <F1> — в этом случае вместо справки по Delphi открывалась справка по Windows API. Второй вариант — в меню **Программы** найти папку **Delphi**, а в ней — папку **Help\MS SDK Files** и выбрать требуемый раздел. Можно также вручную открыть файл MSTools.hlp. В ранних версиях Delphi он находится в каталоге $(Delphi)\Help, в более поздних его нужно искать в $(Program Files)\Common Files. Окно старой справки показано на рис. 1.2.

Вторая проблема, связанная со справкой на основе hlp-файлов, — это то обстоятельство, что разработчики Delphi, разумеется, не сами писали эту справку, а взяли ту, которую предоставила Microsoft. Microsoft же последнюю версию справки в формате HLP выпустила в тот момент, когда уже вышла Windows 95, но еще не было Windows NT 4. Поэтому про многие функции, прекрасно работающие в NT 4, там написано, что в Windows NT они не поддерживаются, т. к. в более ранних версиях они действительно не поддерживались. В справке, поставляемой с Delphi 7 (и, возможно, с некоторыми более ранними версиями), эта информация подправлена, но даже и там отсутствуют функции, которые появились только в Windows NT 4 (как, например, `CoCreateInstanceEx`). И уж конечно, бесполезно искать в этой справке информацию о функциях, появившихся в Windows 98, 2000, XP. Соответственно, при работе в этих версиях Delphi даже не возникает вопрос, что предпочесть для получения информации о Windows API, — справку, поставляемую с Delphi, или MSND. Безусловно, следует выбрать MSDN. Справка, поставляемая с Delphi, имеет только одно преимущество по сравнению с MSDN: ее можно вызывать из среды нажатием клавиши <F1>. Но риск получить неверные сведения слишком велик, чтобы это преимущество могло быть серьезным аргументом. Единственная ситуация, когда предпочтительна справка, поставляемая с Delphi, — это случай, если у вас нет достаточно быстрого

доступа к Интернету для работы с online-версией MSDN и нет возможности приобрести и установить его offline-версию.

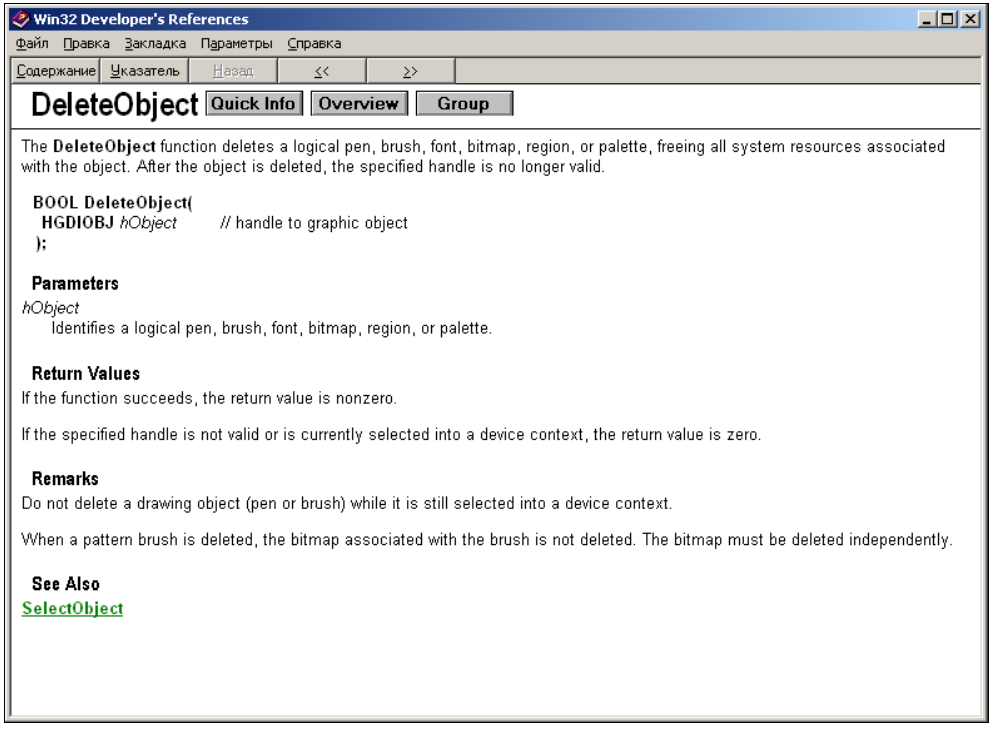


Рис. 1.2. Старая (на основе hlp-файлов) справка по Windows API (показана функция `DeleteObject`)

Начиная с BDS 2006, Borland/CodeGear реализовала новую справочную систему Borland Help (рис. 1.3). По интерфейсу она очень напоминает offline-версию MSDN, а также использует файлы в том же формате, поэтому технологических проблем интеграции справочных систем по Delphi и по Windows API больше не существует. В справку BDS 2006 интегрирована справка по Windows API от 2002—2003 годов (разные разделы имеют разную дату). Справка Delphi 2007 содержит сведения по Windows API от 2006 года, т. е. совсем новые. Таким образом, при работе с Delphi 2007 наконец-то можно полностью отказаться от offline-версии MSDN, а к online-версии обращаться лишь изредка, когда требуется информация о самых последних изменениях в Windows API (например, о тех, которые появились в Windows Vista).

Примечание

Несмотря на очень высокое качество разделов MSDN, относящихся к Windows API, ошибки иногда бывают и там. Со временем их исправляют. Поэтому, если

вы столкнулись с ситуацией, когда есть подозрение, что какая-либо функция Windows API ведет себя не так, как это описано в вашей offline-справке, есть смысл заглянуть в online-справку — возможно, там уже появились дополнительные сведения по данной функции.

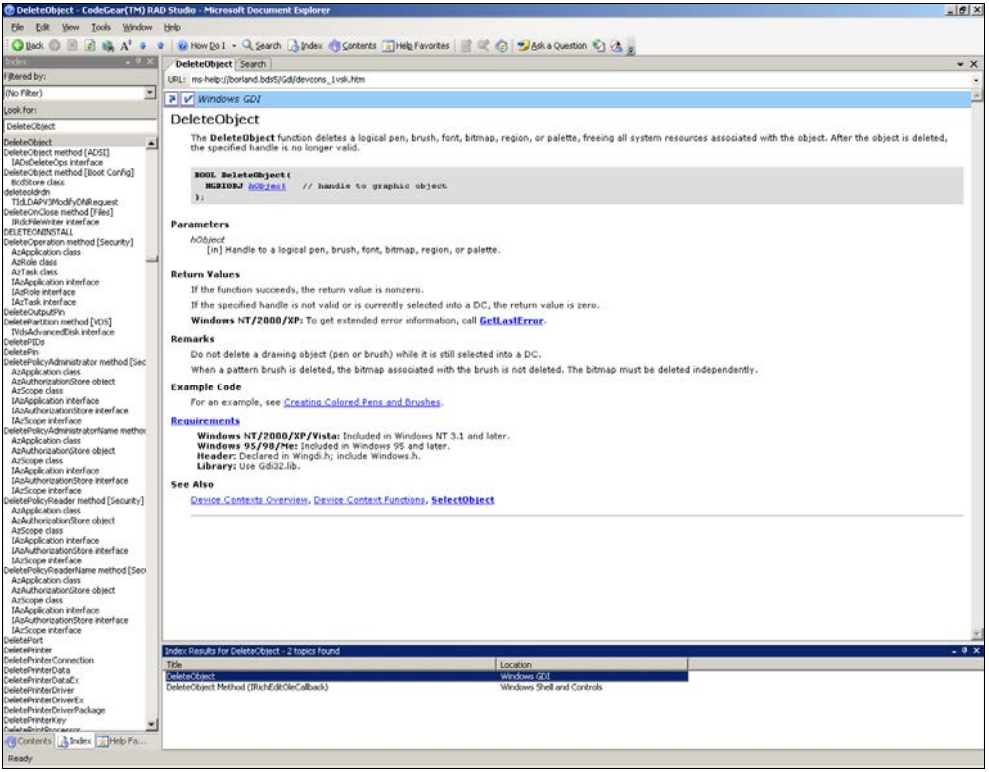


Рис. 1.3. Окно справки Delphi 2007 (функция `DeleteObject`)

Система Windows написана на C++, поэтому все описания функций Windows API, а также примеры их использования приведены на этом языке (это касается как MSDN, так и справки, поставляемой с Delphi). При этом, прежде всего, необходимо разобраться с типами данных. Большинство типов, имеющихся в Windows API, определены в Delphi. Соответствие между ними показано в табл. 1.1.

Таблица 1.1. *Соответствие типов Delphi системным типам*

Тип Windows API	Тип Delphi
`INT`	`INT`
`UINT`	`LongWord`

Таблица 1.1 *(окончание)*

Тип Windows API	Тип Delphi
WORD	Word
SHORT	SmallInt
USHORT	Word
CHAR	Чаще всего соответствует типу Char, но может трактоваться также как ShortInt, т. к. в C++ нет разницы между символьным и целочисленным типами
UCHAR	Чаще всего соответствует типу Byte, но может трактоваться также как Char
DWORD	LongWord
BYTE	Byte
WCHAR	WideChar
BOOL	LongBool
int	Integer
long	LongInt
short	SmallInt
unsigned int	Cardinal

Название типов указателей имеет префикс *P* или *LP* (Pointer или Long Pointer; в 16-разрядных версиях Windows были короткие и длинные указатели. В 32-разрядных все указатели длинные, поэтому оба префикса имеют одинаковый смысл). Например, LPDWORD эквивалентен типу ^DWORD, PUCHAR — ^Byte. Иногда после префикса *P* или *LP* стоит еще префикс *C* — он означает, что это указатель на константу. В C++ возможно объявление таких указателей, которые указывают на константное содержимое, т. е. компилятор разрешает это содержимое читать, но не модифицировать. В Delphi такие указатели отсутствуют, и при портировании эти типы заменяются обычными указателями, т. е. префикс *C* игнорируется.

Типы PVOID и LPVOID соответствуют нетипизированным указателям (Pointer).

Для передачи символов чаще всего используется тип TCHAR. Windows поддерживает две кодировки: ANSI (1 байт на символ) и Unicode (2 байта на символ; о поддержке Unicode в Windows мы будем говорить далее). Тип CHAR соответствует символу в кодировке ANSI, WCHAR — Unicode. Для программ, которые используют ANSI, тип TCHAR эквивалентен типу CHAR, для использующих

Unicode — `WCHAR`. В Delphi нет прямого аналога типу `TCHAR`, программист сам должен следить за тем, какой символьный тип требуется в данном месте.

Строки в Windows API передаются как указатели на цепочку символов, завершающихся нулем. Поэтому указатель на `TCHAR` может указывать как на единичный символ, так и на строку. Чтобы было легче разобраться, где какой указатель, в Windows API есть типы `LPTCHAR` и `LPTSTR`. Они эквивалентны друг другу, но первый принято использовать там, где требуется указатель на одиночный символ, а второй — на строку. Если строка передается в функцию только для чтения, обычно используют указатель на константу, т. е. тип `LPCTSTR`. В Delphi это соответствует `PChar` для ANSI и `PWideChar` для Unicode.

Здесь следует отметить особенность записи строковых литералов в языках C/C++. Символ `\` в литерале имеет специальное значение: после него идет один или несколько управляющих символов. Например, `\n` означает перевод строки, `\t` — символ табуляции и т. п. В Delphi таких последовательностей нет, поэтому при переводе примеров из MSDN следует явно писать коды соответствующих символов. Например, литерал `"a\nb"` в Delphi превращается в `'a'#13'b'`. После символа `\` может идти число — в этом случае оно трактуется как код символа, т. е. литерал `"a\0b\9"` в C/C++ эквивалентен литералу `'a'#0'b'#9` в Delphi. Если нужно, чтобы строковый литерал включал в себя сам символ `\`, его удваивают, т. е. литерал `"\\"` в C++ соответствует `'\'` в Delphi. Кроме того, в примерах кода, приведенных в MSDN, можно нередко увидеть, что строковые литералы обрабатываются макросами `TEXT` или `_T`, которые служат для унификации записи строковых литералов в кодировках ANSI и Unicode. При переводе такого кода на Delphi эти макросы можно просто опустить. С учетом сказанного такой, например, код (взят из примера использования Named pipes):

```
LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");
```

на Delphi будет выглядеть так:

```
var
  lpszPipeName: PChar;
...
lpszPipeName := '\\.\pipe\mynamedpipe';
```

Большинство названий типов из левой части табл. 1.1 в целях совместимости описаны в модуле `Windows`, поэтому они допустимы наравне с обычными типами Delphi. Кроме этих типов общего назначения существуют еще специальные. Например, дескриптор окна имеет тип `HWND`, первый параметр сообщения — тип `WPARAM` (в старых 16-разрядных Windows он был эквивалентен типу `Word`, в 32-разрядных — `LongInt`). Эти специальные типы также описаны в модуле `Windows`.

Записи (record) в C/C++ называются структурами и объявляются с помощью слова struct. Из-за особенностей описания структур на языке C структуры в Windows API получают два имени: одно основное имя, составленное из заглавных букв, которое затем и используется, и одно вспомогательное, получающееся из основного добавлением префикса tag. Начиная с четвертой версии Delphi приняты следующие правила именования таких типов: простое и вспомогательное имена остаются без изменений и еще добавляется новое имя, получающееся из основного присоединением общеупотребительного в Delphi префикса *T*. Например, в функции CreatePenIndirect один из параметров имеет тип LOGPEN. Это основное имя данного типа, а вспомогательное — tagLOGPEN. Соответственно, в модуле Windows определена запись tagLOGPEN и ее синонимы — LOGPEN и TLogPen. Эти три идентификатора в Delphi взаимозаменяемы. Вспомогательное имя встречается редко, программисты, в зависимости от личных предпочтений, выбирают либо основное имя типа, либо имя с префиксом *T*.

Описанные здесь правила именования типов могут внести некоторую путаницу при использовании VCL. Например, для описания растра в Windows API определен тип BITMAP (он же — tagBITMAP). В Delphi соответствующий тип имеет еще одно имя — TBitmap. Но такое же имя имеет класс TBitmap, описанный в модуле Graphics. В коде, который Delphi создает автоматически, модуль Graphics находится в списке uses после модуля Windows, поэтому идентификатор TBitmap воспринимается компилятором как Graphics.TBitmap, а не как Windows.TBitmap. Чтобы использовать Windows.TBitmap, нужно явно указать имя модуля или воспользоваться одним из альтернативных имен.

В более ранних версиях Delphi были другие правила именования типов. Например, в Delphi 2 существовал тип BITMAP, но не было TBitmap и tagBITMAP, а в Delphi 3 из этих трех типов был только TBitmap.

Все структуры в Windows API описаны без *выравнивания*, т. е. компилятор не вставляет между полями неиспользуемые байты, чтобы границы полей приходились на начало двойного или четверного слова, поэтому в Delphi для описания соответствующих структур предусмотрено слово packed, запрещающее выравнивание.

При описании структур Windows API можно иногда встретить ключевое слово union (см., например, структуру in_addr). Объединение нескольких полей с помощью этого слова означает, что все они будут размещены по одному адресу. В Delphi это соответствует вариантным записям (т. е. использованию case в record). Объединения в C/C++ гибче, чем вариантные записи Delphi, т. к. позволяют размещать вариантную часть в любом месте структуры, а не только в конце. При переносе таких структур в Delphi иногда приходится вводить дополнительные типы.

Теперь рассмотрим синтаксис описания самой функции в C++ (листинг 1.1).

Листинг 1.1. Синтаксис описания функции на C++

```
<Тип функции> <Имя функции> '('
    [ <Тип параметра> {<Имя параметра>}
      {','} <Тип параметра> {<Имя параметра>} }
    ]
 ')';
```

Как видно из листинга 1.1, при объявлении функции существует возможность указывать только типы параметров и не указывать их имена. Однако это считается устаревшим и применяется крайне редко (если не считать "параметров" типа `VOID`, о которых написано далее).

Необходимо помнить, что в C/C++ различаются верхний и нижний регистры, поэтому `HDC`, `hdc`, `hDC` и т. д. — это разные идентификаторы (автор C очень любил краткость и хотел, чтобы можно было делать не 26, а 52 переменные с именем из одной буквы). Поэтому часто можно встретить, что имя параметра и его тип совпадают с точностью до регистра. К счастью, при описании функции в Delphi мы не обязаны сохранять имена параметров, значение имеют лишь их типы и порядок следования. С учетом всего этого функция, описанная в справке как

```
HMETAFILE CopyMetaFile(HMETAFILE hmfSrc, LPCTSTR lpszFile);
```

в Delphi имеет вид

```
function CopyMetaFile(hmfSrc: HMETAFILE;
  lpszFile: LPCTSTR): HMETAFILE;
```

или, что то же самое,

```
function CopyMetaFile(hmfSrc: HMETAFILE; lpszFile: PChar): METAFILE;
```

Примечание

Компилятор Delphi допускает, чтобы имя параметра процедуры или функции совпадало с именем типа, поэтому мы в дальнейшем увидим, что иногда имя параметра и его тип совпадают, только записываются в разном регистре, чтобы прототип функции на Delphi максимально соответствовал исходному прототипу на C/C++. При этом следует учитывать, что соответствующий идентификатор внутри функции будет рассматриваться как имя переменной, а не типа, поэтому, например, объявлять локальную переменную данного типа придется с явным указанием имени модуля, в котором данный тип объявлен.

Несколько особняком стоит тип `VOID` (или `void`, что то же самое, но в Windows API этот идентификатор встречается существенно реже). Если

функция имеет такой тип, то в Паскале она описывается как процедура. Если вместо параметров у функции в скобках указан VOID, это означает, что функция не имеет параметров. Например, функция

```
VOID CloseLogFile(VOID);
```

в Delphi описывается как

```
procedure CloseLogFile;
```

Примечание

Язык C++, в отличие от C, допускает объявление функций без параметров, т. е. функцию CloseLogFile можно было бы объявить так: VOID CloseLogFile();. В C++ эти варианты объявления эквивалентны, но в Windows API вариант без явного параметра встречается существенно реже из-за несовместимости с C.

Когда тип параметра является указателем на другой тип (обычно начинается с букв LP), при описании этой функции в Delphi можно пользоваться параметром-переменной, т. к. в этом случае функции передается указатель. Например, функция

```
int GetRgnBox(HRGN hrgn, LPRECT lprc);
```

в модуле Windows описана как

```
function GetRgnBox(RGN: HRGN; var p2: TRect): Integer;
```

Такая замена целесообразна в том случае, если значение параметра не может быть нулевым указателем, потому что при использовании var передать такой указатель будет невозможно. Нулевой указатель в C/C++ обозначается константой NULL. NULL и 0 в этих языках взаимозаменяемы, поэтому в справке можно и про целочисленный параметр встретить указание, что он может быть равен NULL.

И наконец, если не удается понять, как функция, описанная в справке, должна быть переведена на Паскаль, можно попытаться найти описание этой функции в исходных текстах модулей, поставляемых вместе с Delphi. Эти модули находятся в каталоге $(DELPHI)\Source\RTL\Win (до Delphi 7) или $(BDS)\Source\Win32\RTL\Win (BDS 2006 и выше). Можно также воспользоваться подсказкой, которая всплывает в редакторе Delphi после того, как будет набрано имя функции.

Если посмотреть справку, например, по функции GetSystemMetrics, то видно, что эта функция должна иметь один целочисленный параметр. Однако далее в справке предлагается при вызове этой функции подставлять в качестве параметра не числа, а SM_ARRANGE, SM_CLEANBOOT и т. д. Подобная ситуация и со многими другими функциями Windows API. Все эти SM_ARRANGE, SM_CLEANBOOT и т. д. являются именами числовых констант. Эти константы описаны в том

же модуле, в котором описана функция, использующая их, поэтому можно не выяснять численные значения этих констант, а указывать при вызове функций их имена, например, `GetSystemMetrics(SM_ARRANGE);`.

Если по каким-то причинам все-таки потребовалось выяснить численные значения, то в справочной системе их искать не стоит — их там нет. Их можно узнать из исходных текстов модулей Delphi, в которых эти константы описаны. Так, например, просматривая Windows.pas, можно узнать, что `SM_ARRANGE = 56`.

В справке, поставляемой вместе с Delphi до 7-й версии включительно, в описании многих функций Windows API вверху можно увидеть три ссылки: **QuickInfo**, **Overview** и **Group**. Первая дает краткую информацию о функции: какой библиотекой реализуется, в каких версиях Windows работает и т. п. (напоминаю, что к информации о версиях в этой справке нужно относиться очень критично). **Overview** — это обзор какой-то большой темы. Например, для любой функции, работающей с растровыми изображениями, обзор будет объяснять, зачем в принципе нужны эти самые растровые изображения и как они устроены. Страница, на которую ведет ссылка **Overview**, обычно содержит весьма лаконичные сведения, но, нажав кнопку >>, расположенную в верхней части окна, можно получить продолжение обзора. И, наконец, **Group**. Эта ссылка приводит к списку всех функций, родственных данной. Например, для функции `CreateRectRgn` группу будут составлять все функции, имеющие отношение к регионам. Если теперь нажимать на кнопку <<, то будут появляться страницы с кратким описанием возможных применений объектов, с которыми работают функции (в приведенном примере — описание возможностей регионов). Чтобы читать их в нормальной последовательности, лучше всего нажать на кнопку << столько раз, сколько возможно, а затем пойти в противоположном направлении с помощью кнопки >>.

MSDN (а также справка BDS 2006 и выше) предоставляет еще больше полезной информации. В нижней части описания каждой функции есть раздел **Requirements**, в котором написано, какая библиотека и какая версия Windows требуется для ее использования. В самом низу описания функции расположены ссылки **See also**. Первая ссылка — обзор соответствующей темы (например, для уже упоминавшейся функции `CreateRectRgn` она называется **Regions Overview**). Вторая — список родственных функций (**Region Functions** в данном случае). Она ведет на страницу, где перечислены все функции, родственные выбранной. После этих двух обязательных ссылок идут ссылки на описание функций и типов, которые обычно используются совместно с данной функцией.

Основные типы, константы и функции Windows API объявлены в модулях `Windows` и `Messages`. Но многие функции объявлены в других модулях, кото-

рые не подключаются к программе по умолчанию, программист должен сам выяснить, в каком модуле находится требуемый ему идентификатор, и подключить этот модуль. Ни справка, поставляемая с Delphi, ни MSDN, разумеется, не могут дать необходимую информацию. Чтобы выяснить, в каком модуле объявлен нужный идентификатор, можно воспользоваться поиском по всем файлам с расширением pas, находящимся в папке с исходными кодами стандартных модулей. Этим методом можно, например, выяснить, что весьма популярная функция `ShellExecute` находится в модуле `ShellAPI`, а `CoCreateInstance` — в модуле `ActiveX` (а также в модуле `Ole2`, оставленном для совместимости со старыми версиями Delphi).

Еще несколько слов о числовых константах. В справке можно встретить числа вида, например, `0xC56F` или `0x3341`. Префикс `0x` в C/C++ означает шестнадцатеричное число. В Delphi его следует заменить на `$`, т. е. эти числа должны быть записаны как `$C56F` и `$3341` соответственно.

1.1.3. Дескрипторы вместо классов

Программируя в Delphi, мы быстро привыкаем к тому, что каждый объект реализуется экземпляром соответствующего класса. Например, кнопка реализуется экземпляром класса `TButton`, контекст устройства — классом `TCanvas`. Но когда создавались первые версии Windows, объектно-ориентированный метод программирования еще не был общепризнанным, поэтому он не был реализован. Современные версии Windows частично унаследовали этот недостаток, поэтому в большинстве случаев приходится работать "по старинке", тем более что DLL могут экспортировать только функции, но не классы. Когда мы будем говорить об объектах, создаваемых через Windows API, будем подразумевать не объекты в терминах ООП, а некоторую сущность, внутренняя структура которой скрыта от нас, поэтому с этой сущностью мы можем оперировать только как с единым и неделимым (атомарным) объектом.

Каждому объекту, созданному с помощью Windows API, присваивается уникальный номер *(дескриптор)*. Его конкретное значение не несет для программиста никакой полезной информации и может быть использовано только для того, чтобы при вызове функций из Windows API указывать, с каким объектом требуется выполнить операцию. В большинстве случаев дескрипторы представляют собой 32-значные числа, а значит, их можно передавать везде, где требуются такие числа. В дальнейшем мы увидим, что Windows API несколько вольно обращается с типами, т. е. один и тот же параметр в различных ситуациях может содержать и число, и указатель, и дескриптор, поэтому знание двоичного представления дескриптора все-таки приносит программисту пользу (хотя если бы система Windows была "спроектирована по правилам", тип дескриптора вообще не должен был интересовать программиста).

Таким образом, главное различие между методами класса и функциями Windows API заключается в том, что первые связаны с тем экземпляром класса, через который они вызываются, и поэтому не требуют явного указания на объект. Вторым необходимо указание объекта через его дескриптор, т. к. они сами по себе никак не связаны ни с одним объектом.

Компоненты VCL нередко являются оболочками над объектами Delphi. В этом случае они имеют свойство (которое обычно называется `Handle`), содержащее дескриптор соответствующего объекта. Иногда класс Delphi инкапсулирует несколько объектов Windows. Например, класс `TBitmap` включает в себя `HBITMAP` и `HPALETTE` — картинку и палитру к ней. Соответственно, он хранит два дескриптора: в свойствах `Handle` и `Palette`.

Следует учитывать, что внутренние механизмы VCL не могут включиться, если изменение объекта происходит через Windows API. Например, если спрятать окно не с помощью метода `Hide`, а путем вызова функции Windows API `ShowWindow(Handle, SW_HIDE)`, не возникнет событие `OnHide`, потому что оно запускается теми самыми внутренними механизмами VCL. Но такие недоразумения случаются обычно только тогда, когда функциями Windows API дублируется то, что можно сделать и с помощью VCL.

Все экземпляры классов, созданные в Delphi, должны удаляться. В некоторых случаях это происходит автоматически, а иногда программист должен сам позаботиться о "выносе мусора". Аналогичная ситуация и с объектами, создаваемыми в Windows API. Если посмотреть справку по функции, создающей какой-то объект, то там обязательно будет информация о том, какой функцией можно удалить объект и нужно ли это делать вручную, или система сделает это автоматически. Во многих случаях совершенно разные объекты могут удаляться одной и той же функцией. Так, функция `DeleteObject` удаляет косметические перья, геометрические перья, кисти, шрифты, регионы, растровые изображения и палитры. Обращайте внимание на возможные исключения. Например, регионы не удаляются системой автоматически, однако если вызвать для региона функцию `SetWindowRgn`, то он переходит в собственность операционной системы. Никакие дальнейшие операции с ним, в том числе и удаление, совершать нельзя.

Если системный объект используется только одним приложением, то он будет удален при завершении работы приложения. Тем не менее хороший стиль программирования требует, чтобы программа удаляла объекты явно, а не полагалась на систему.

1.1.4. Формы VCL и окна Windows

Под словом "окно" обычно подразумевается некоторая форма наподобие тех, что можно создать с помощью класса `TForm`. Однако это понятие существенно

шире. В общем случае окном называется любой объект, который имеет экранные координаты и может реагировать на мышь и клавиатуру. Например, кнопка, которую можно создать с помощью класса `TButton`, — это тоже окно.

VCL вносит некоторую путаницу в это понятие. Некоторые визуальные компоненты VCL не являются окнами, а только имитируют их, как, например, `TImage`. Это позволяет экономить ресурсы системы и повысить быстродействие программы. Механизм этой имитации мы рассмотрим позже, а пока следует запомнить, что окнами являются только те визуальные компоненты, которые имеют в числе предков класс `TWinControl`. Разработчики VCL постарались, чтобы разница между *оконными* и *неоконными визуальными компонентами* была минимальной. Действительно, на первый взгляд неоконный `TLabel` и оконный `TStaticText` кажутся практически близнецами. Разница становится заметной тогда, когда используется Windows API. С неоконными компонентами можно работать только средствами VCL, они даже не имеют свойства `Handle`, в то время как оконными компонентами можно управлять с помощью Windows API.

Отметим также еще одно различие между оконными и неоконными компонентами: неоконные компоненты рисуются непосредственно на поверхности *родительского компонента*, в то время как оконные как бы "кладутся" на родителя сверху. В частности, это означает, что неоконный `TLabel`, размещенный на форме, не может закрывать собой часть кнопки `TButton`, потому что `TLabel` рисуется на поверхности формы, а кнопка — это независимый объект, лежащий на форме и имеющий свою поверхность. А `TStaticText` может оказаться над кнопкой, потому что он тоже находится над формой.

Примечание

Чтобы разместить неоконный визуальный компонент над оконным, если в этом есть необходимость, можно поступить следующим образом. Положить на форму панель (`TPanel`) — она является оконным компонентом и может быть размещена поверх других оконных элементов. На панель теперь можно положить любой неоконный визуальный компонент, и он будет рисоваться не на поверхности формы, а на поверхности панели. Если теперь убрать у панели рамку и уменьшить ее до размеров содержащегося в ней неоконного компонента, панель станет незаметной, и все вместе это будет выглядеть так, будто неоконный компонент находится над оконным.

Каждое окно принадлежит к какому-то *оконному классу*. Не следует путать оконный класс с классами Delphi. Это некий шаблон, определяющий базовые свойства окна. Каждому такому шаблону присваивается имя, уникальное в его области видимости. Перед использованием класс необходимо зарегистрировать (функция `RegisterClassEx`). В качестве параметра эта функция принимает запись типа `TWndClassEx`, поля которой содержат параметры класса.

С каждым окном должна быть связана специальная функция, называющаяся *оконной процедурой* (подробнее мы рассмотрим ее чуть позже). Она является параметром не отдельного окна, а всего оконного класса, т. е. все окна, принадлежащие данному классу, будут использовать одну и ту же оконную процедуру. Эта процедура может размещаться либо в самом исполняемом модуле, либо в одной из загруженных им DLL. При создании класса указывается дескриптор модуля, в котором находится оконная процедура.

Примечание

Здесь следует отметить некоторую путаницу в терминах. В англоязычной справке есть слово module, служащее для обозначения файла, отображенного в адресное пространство процесса, т. е., в первую очередь, exe-файла, породившего процесс, и загруженных им DLL. И есть слово unit, которое обозначает модуль в Delphi и которое также переводится как модуль. Ранее мы говорили о модулях как об отображаемых в адресное пространство файлах — это они имеют дескрипторы. Модули Delphi не являются системными объектами и дескрипторов не имеют.

Дескриптор модуля, загруженного в память, можно получить с помощью функции `GetModuleHandle`. Функция `LoadLibrary` в случае успешного завершения также возвращает дескриптор загруженной DLL. Кроме того, Delphi предоставляет две переменные: `MainInstance` из модуля `System` и `HInstance` из модуля `SysInit` (оба этих модуля подключаются к программе автоматически, без явного указания в списке `uses`). `MainInstance` содержит дескриптор exe-файла, породившего процесс, `HInstance` — текущего модуля. В исполняемом файле `MainInstance` и `HInstance` равны между собой, в DLL `HInstance` содержит дескриптор самой библиотеки, а `MainInstance` — загрузившего ее главного модуля.

Каждое окно в Windows привязывается к какому-либо модулю (в Windows 9*x*/ME необходимо явно указать дескриптор этого модуля, NT/2000/XP определяет модуль, из которого вызвана функция создания окна, автоматически). Соответственно, оконные классы делятся на *локальные* и *глобальные*: окна локальных классов может создавать только тот модуль, в котором находится оконная процедура класса, глобальных — любой модуль данного приложения. Будет ли класс локальным или глобальным, зависит от значений полей `TWndClassEx` при регистрации класса.

Оконный класс, к которому принадлежит окно, указывается при его создании. Это может быть зарегистрированный ранее класс или один из *системных классов*. Системные классы — это `'BUTTON'`, `'COMBOBOX'`, `'EDIT'`, `'LISTBOX'`, `'MDICLIENT'`, `'SCROLLBAR'` и `'STATIC'`. Назначение этих классов понятно из их названий (класс `'STATIC'` реализует статические текстовые или графические элементы, т. е. не реагирующие на мышь и клавиатуру, но имеющие дескриптор). Кроме этих классов существуют также классы из биб-

лиотеки ComCtl32.dll, они тоже доступны всем приложениям без предварительной регистрации (подробнее об этих классах можно узнать в MSDN, в разделе **Common Controls Reference**).

Для окон в обычном понимании этого слова готовых классов не существует, их приходится регистрировать самостоятельно. В частности, VCL для форм регистрирует оконные классы, имена которых совпадают с именами соответствующих классов VCL.

Кроме имени, класс включает в себя другие параметры, такие как стиль, кисть и т. д. Они подробно перечислены в справке.

Для создания окна служат функции `CreateWindow` и `CreateWindowEx`. При создании окна в числе других параметров задается модуль, к которому оно привязано, имя оконного класса, *стиль* и *расширенный стиль*. Последние два параметра определяют поведение конкретного окна и не имеют ничего общего со стилем класса. Результат работы этих функций — дескриптор созданного ими окна.

Еще один важный параметр этих функций — дескриптор *родительского окна*. Окно является подчиненным по отношению к своему родителю. Например, если дочернее окно — это кнопка или иной элемент управления, то визуально оно располагается в другом окне, которое является для него родительским. Если дочернее окно — это `MDIChild`, то родительским для него будет `MDIForm` (если быть до конца точным, то не сама форма `MDIForm`, а специальное окно класса `MDICLIENT`, которое является дочерним по отношению к `MDIForm`; дескриптор этого окна хранится в свойстве `ClientHandle` главной формы). Другими словами, отношения "родительское — дочернее окно" отражают принадлежность одного окна другому, визуальную связь между ними. Окна, родитель которых не задан (т. е. в качестве дескриптора родителя передан ноль), располагаются непосредственно на рабочем столе. Если при создании окна задан стиль `WS_CHILD`, то его координаты отсчитываются от левого верхнего угла клиентской области родительского окна, и при перемещении родительского окна все дочерние окна будут перемещаться вместе с ним. Окно, имеющее стиль `WS_CHILD`, не может располагаться на рабочем столе, попытка создать такое окно окончится неудачей.

Визуальные компоненты VCL имеют два свойства, которые иногда путают: `Owner` и `Parent`. Свойство `Parent` указывает на объект, реализующий окно, являющееся родительским для данного визуального компонента (компоненты, не являющиеся наследником `TWinControl`, также имеют это свойство — VCL для них имитирует эту взаимосвязь, однако сами они не могут быть родителями других визуальных компонентов). Свойство `Owner` указывает на *владельца компонента*. Отношения "владелец-принадлежащий" реализуются полностью внутри VCL. Свойство `Owner` есть у любого наследника

TComponent, в том числе и у невизуальных компонентов, и владельцем других компонентов также может быть невизуальный компонент (например, TDataModule). При уничтожении компонента он автоматически уничтожает все компоненты, владельцем которых он является (здесь, впрочем, есть некоторое дублирование функций, т. к. оконный компонент также при уничтожении уничтожает все визуальные компоненты, родителем которых он является). Еще владелец отвечает за загрузку всех установленных во время разработки свойств принадлежащих ему компонентов.

Свойство Owner доступно только для чтения. Владелец компонента задается один раз при вызове конструктора и остается неизменным на протяжении всего жизненного цикла компонента (за исключением достаточно редких случаев явного вызова методов InsertComponent и RemoveComponent). Свойство Parent задается отдельно и может быть впоследствии изменено (визуально это будет выглядеть как "перепрыгивание" компонента из одного окна в другое).

Визуальный компонент может не иметь владельца. Это означает, что ответственность за его удаление лежит на программисте, создавшем его. Но большинство визуальных компонентов не может функционировать, если свойство Parent не задано. Например, невозможно отобразить на экране компонент TButton, у которого не установлено свойство Parent. Это связано с тем, что большинство оконных компонентов имеет стиль WS_CHILD, который, напомним, не позволяет разместить окно на рабочем столе. Окнами без родителя могут быть только наследники TCustomForm.

Впрочем, сделать кнопку, не имеющую родителя, можно средствами Windows API. Например, такой командой (листинг 1.2).

Листинг 1.2. Создание кнопки, не имеющей родительского окна

```
CreateWindow('BUTTON', 'Test',
  WS_VISIBLE or BS_PUSBUTTON or WS_POPUP,
  10, 10, 100, 50, 0, 0, HInstance, nil);
```

Рекомендуем в этом примере убрать стиль WS_POPUP и посмотреть, что получится — эффект достаточно забавный. Отметим, что создавать такие висящие сами по себе кнопки смысла нет, поскольку сообщения о событиях, происходящих со стандартными элементами управления, получает родительское окно, и при его отсутствии программа не может отреагировать, например, на нажатие кнопки.

Кроме обычного конструктора Create, у класса TWinControl есть конструктор CreateParented, позволяющий создавать оконные компоненты, родителями

которых являются окна, созданные без использования VCL. В качестве параметра этому конструктору передается дескриптор родительского окна. У компонентов, созданных таким образом, не нужно устанавливать свойство `Parent`.

Примечание

Путаницу между понятием родителя и владельца усиливает то, что в MSDN по отношению к окнам тоже используются термины owner и owned (принадлежащий), однако это не имеет никакого отношения к владельцу в понимании VCL. Если окно имеет стиль `WS_CHILD`, то оно обязано иметь родителя, но не может иметь владельца. Если такого стиля у окна нет, оно не может иметь родителя, но может (хотя и не обязано) иметь владельца. Владельцем в этом случае становится то окно, чей дескриптор передан в качестве родительского, т. е. родитель и владелец в терминах системы — это один и тот же параметр, который по-разному интерпретируется в зависимости от стиля самого окна. Окно, имеющее владельца, уничтожается при уничтожении владельца, прячется при его минимизации и всегда находится над владельцем. Окно, имеющее стиль `WS_CHILD`, может быть родителем, но не может быть владельцем другого окна; если передать дескриптор такого окна в качестве владельца, то реальным владельцем станет родитель дочернего окна. Чтобы не путать владельца в терминах VCL и в терминах системы, мы в дальнейшем всегда будем оговаривать, в каком смысле будет упомянуто слово "владелец".

Создание окон через Windows API требует кропотливой работы. VCL справляется с этой задачей замечательно, поэтому создавать окна самостоятельно приходится только тогда, когда использование VCL нежелательно, например, если необходимо написать как можно более компактное приложение. Во всех остальных случаях приходится только слегка подправлять работу VCL. Например, с помощью Windows API можно изменить форму окна или убрать из него заголовок, оставив рамку. Подобные действия не требуют от программиста создания нового окна, можно воспользоваться тем, что уже создано VCL.

Другой случай, когда могут понадобиться функции Windows API для окон, — если приложение должно что-то делать с чужими окнами. Например, хотя бы просто перечислить все окна, открытые в данный момент, как это делает входящая в состав Delphi утилита WinSight32. Но в этом случае также не приходится самому создавать окна, работа идет с уже имеющимися.

1.1.5. Функции обратного вызова

Прежде чем двигаться дальше, необходимо разобраться с тем, что такое *функции обратного вызова* (callback functions; этот термин иногда также переводят "функции косвенного вызова"). Эти функции в программе описываются, но обычно не вызываются напрямую, хотя ничто не запрещает сделать это. В этом они похожи на те методы класса, которые связаны с событиями.

Ничто не мешает вызывать напрямую, например, метод `FormCreate`, но делать это приходится крайне редко. С другой стороны, даже если этот метод не вызывается явно, он все равно выполняется, потому что VCL автоматически вызывает его без прямого указания программиста. Еще одно общее свойство — конкретное имя метода при косвенном вызове не важно. Можно изменить его, но если этот метод по-прежнему будет связан с событием `OnCreate`, он так же будет успешно вызываться. Разница заключается только в том, что такие методы вызываются внутренними механизмами VCL, а функции обратного вызова — самой системой Windows. Соответственно, на эти функции налагаются следующие требования: во-первых, они должны быть именно функциями, а не методами класса; во-вторых, они должны быть написаны в соответствии с моделью вызова `stdcall` (MSDN предлагает использовать модель `callback`, которая в имеющихся версиях Windows является синонимом `stdcall`). Что же касается того, как программист сообщает системе о том, что он написал функцию обратного вызова, то это в каждом случае будет по-своему.

В качестве примера рассмотрим перечисление окон с помощью функции `EnumWindows`. В справке она описана так:

```
BOOL EnumWindows(WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

Соответственно, в Windows.pas она имеет вид

```
function EnumWindows(lpEnumFunc: TFNWndEnumProc;
  lParam: LPARAM): BOOL; stdcall;
```

Параметр `lpEnumFunc` должен содержать указатель на функцию обратного вызова. Прототип этой функции описан так:

```
BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM lParam);
```

Функции с таким именем в Windows API не существует. Это так называемый *прототип функции*, согласно которому следует описывать функцию обратного вызова. На самом деле этот прототип предоставляет бóльшую свободу, чем это может показаться на первый взгляд. Во-первых, имя может быть любым. Во-вторых, система не накладывает строгих ограничений на имена и типы параметров — они могут быть любыми, при условии, что новые типы совпадают по размерам с теми, которые указываются (тип `TFNWndEnumProc`, описанный в модуле `Windows` — это не процедурный тип, а просто нетипизированный указатель, поэтому компилятор Delphi не будет контролировать соответствие передаваемой функции обратного вызова ее прототипу). Что касается типа функции и типа первого параметра, то они имеют определенный смысл, и изменение их типа вряд ли может быть полезным. Но второй параметр предназначен специально для передачи значения, которое программист волен использовать по своему усмотрению, система просто передает через него в

функцию обратного вызова то значение, которое имел параметр lParam при вызове функции EnumWindows. А программисту может показаться удобнее работать не с типом LPARAM (т. е. LongInt), а, например, с указателем или же с массивом из четырех байтов. Лишь бы были именно четыре байта, а не восемь, шестнадцать или еще какое-то число. Можно даже превратить этот параметр в параметр-переменную, т. к. при этом функции будут передаваться все те же четыре байта — адрес переменной. Впрочем, тем, кто не очень хорошо разбирается с тем, как используется стек для передачи параметров при различных моделях вызова, лучше не экспериментировать с изменением типа параметра, а строго следовать заявленному прототипу, при необходимости выполняя требуемые преобразования внутри функции обратного вызова.

Функция EnumWindows работает так: после вызова она начинает по очереди перебирать все имеющиеся в данный момент окна верхнего уровня, т. е. те, у которых нет родителя. Для каждого такого окна вызывается заданная функция обратного вызова, в качестве первого параметра ей передается дескриптор данного окна (каждый раз, естественно, новый), в качестве второго — то, что было передано самой функции EnumWindows в качестве второго параметра (каждый раз одно и то же). Получая по очереди дескрипторы всех окон верхнего уровня, функция обратного вызова может выполнить с каждым из них определенное действие (закрыть, минимизировать и т. п.). Или можно проверять все эти окна на соответствие какому-то условию, пытаясь найти нужное. А значение, возвращаемое функцией обратного вызова, влияет на работу EnumWindows. Если она возвращает False, значит, все, что нужно, уже сделано, можно не перебирать остальные окна.

Окончательный код для того случая, когда второй параметр имеет тип Pointer, иллюстрирует листинг 1.3.

Листинг 1.3. Вызов функции EnumWindows с функцией обратного вызова

```
function MyCallbackFunction(Wnd: HWND; P: Pointer): BOOL; stdcall;
begin
  {  что-то делаем  }
end;
..............
var
  MyPointer: Pointer;
..............
EnumWindows(@MyCallbackFunction, LongInt(MyPointer));
```

Что бы мы ни делали с типом второго параметра функции обратного вызова, тип соответствующего параметра EnumWindows не меняется. Поэтому необхо-

димо явное приведение передаваемого параметра к типу LongInt. Обратное преобразование типов при вызове MyCallbackFunction осуществляется автоматически.

Использование EnumWindows и функций обратного вызова демонстрируется примером EnumWnd.

Отметим, что функции обратного вызова будут вызываться до того, как завершит работу функция EnumWindows. Однако это не является распараллеливанием работы. Чтобы проиллюстрировать это, рассмотрим ситуацию, когда программа вызывает некоторую функцию А, которая, в свою очередь, вызывает функцию В. Функция В, очевидно, начнет свою работу до того, как завершит работу функция А. То же самое произойдет и с функцией обратного вызова, переданной в EnumWindows: она будет вызываться из кода EnumWindows так же, как и функция В из кода функции А. Поэтому код функции обратного вызова получит управление (и не один раз, т. к. EnumWindows будет вызывать эту функцию в цикле) до завершения работы EnumWindows.

Однако это правило действует не во всех ситуациях. В некоторых случаях система запоминает адрес переданной ей функции обратного вызова, чтобы использовать ее потом. Примером такой функции является оконная процедура: ее адрес передается системе один раз при регистрации класса, и затем система многократно вызывает эту функцию при необходимости.

В 16-разрядных версиях Windows вызов функций обратного вызова осложнялся тем, что для них необходим был специальный код, называемый *прологом*. Пролог создавался с помощью функции MakeProcInstance, удалялся после завершения с помощью FreeProcInstance. Таким образом, вызов EnumWindows должен был бы выглядеть так, как показано в листинге 1.4.

Листинг 1.4. Вызов функции EnumWindows в 16-разрядных версиях Windows

```
var
  MyProcInstnace: TFarProc;
..................
MyProcInstance := MakeProcInstance(@MyCallbackFunction, HInstance);
EnumWindows(MyProcInstance, LongInt(MyPointer));
FreeProcInstance(MyProcInstance);
```

В Delphi этот код будет работоспособным, т. к. для совместимости MakeProcInstance и FreeProcInstance оставлены. Но они ничего не делают (в чем легко убедиться, просмотрев исходный файл Windows.pas), поэтому можно обойтись и без них. Тем не менее эти функции иногда до сих пор используются, видимо, просто в силу привычки. Другой способ, с помощью которого в 16-разрядных версиях можно сделать пролог — описать функцию

с директивой `export`. Эта директива сохранена для совместимости и в Delphi, но в 32-разрядных версиях она также ничего не делает (несмотря на то, что справка, например, по Delphi 3 утверждает обратное; в справке по Delphi 4 этой ошибки уже нет).

1.1.6. Сообщения Windows

Человеку, знакомому с Delphi, должна быть ясна схема событийного управления. Программист пишет только методы реакции на различные события, а затем этот код получает управление тогда, когда соответствующее событие произойдет. Простые программы в Delphi состоят исключительно из методов реакции на события (например, `OnCreate`, `OnClick`, `OnCloseQuery`). Причем событием называется не только событие в обычном смысле этого слова, т. е. когда происходит что-то внешнее, но и ситуация, когда событие используется просто для передачи управления коду, написанному разработчиком программы, в тех случаях, когда VCL не может сама справиться с какой-то задачей. Пример такого события — `TListBox.OnDrawItem`. Устанавливая стиль списка в `lbOwnerDrawFixed` или `lbOwnerDrawVariable`, программист указывает, что ему требуется нестандартный вид элементов списка, поэтому их рисование он берет на себя. И каждый раз, когда возникает необходимость в рисовании элемента, VCL передает управление специально написанному коду. На самом деле разница между двумя типами событий весьма условна. Можно сказать, что когда пользователь нажимает клавишу, VCL не "знает", что делать, и поэтому передает управление обработчику `OnKeyPress`.

Событийное управление не есть изобретение авторов Delphi. Такой подход заложен в самой системе Windows. Только здесь события называются *сообщениями* (message), что иногда даже лучше отражает ситуацию. Windows посылает программе сообщения, связанные либо с тем, что произошло внешнее событие (нажатие кнопки мыши, клавиши на клавиатуре и т. п.), либо с тем, что самой системе потребовались от программы какие-то действия. Самое распространенное действие — предоставление информации. Например, при необходимости узнать текст заголовка окна Windows посылает этому окну специальное сообщение, в ответ на которое окно должно сообщить системе свой заголовок. Еще бывают сообщения, которые просто уведомляют программу о начале какого-то действия (например, о начале перетаскивания окна) и предоставляют возможность вмешаться. Но это вмешательство необязательно.

В Delphi для реакции на каждое событие обычно создается свой метод. В Windows одна процедура, называемая оконной, обрабатывает все сообщения, адресованные конкретному окну. (В C/C++ нет понятия "процедура", там термин "оконная процедура" не вызывает путаницы, а вот в Delphi четко определено, что такое процедура. И здесь можно запутаться: то, что в системе

называется оконной процедурой, с точки зрения Delphi будет не процедурой, а функцией. Тем не менее мы будем употреблять общепринятый термин "оконная процедура".) Каждое сообщение имеет свой уникальный номер, а оконная процедура обычно целиком состоит из оператора `case`, и каждому сообщению соответствует своя альтернатива этого оператора. Номера сообщений знать не обязательно, потому что можно использовать константы, описанные в модуле `Messages`. Эти константы начинаются с префикса, указывающего на принадлежность сообщения к какой-то группе. Например, сообщения общего назначения начинаются с `WM_`: `WM_PAINT`, `WM_GETTEXTLENGTH`. Сообщения, специфичные, например, для кнопок, начинаются с префикса `BM_`. Остальные группы сообщений также связаны либо с теми или иными элементами управления, либо со специальными действиями, например, с динамическим обменом данными (Dynamic Data Exchange, DDE). Обычной программе приходится обрабатывать довольно много сообщений, поэтому оконная процедура бывает, как правило, очень длинной и громоздкой. Оконная процедура описывается программистом как функция обратного вызова и указывается при создании оконного класса. Таким образом, все окна данного класса имеют одну и ту же оконную процедуру. Впрочем, существует возможность породить так называемый подкласс, т. е. новый класс, наследующий все свойства существующего, за исключением оконной процедуры. Несколько подробнее об этом будет сказано далее.

Кроме номера, каждое сообщение содержит два параметра: `wParam` и `lParam`. Префиксы *w* и *l* означают "Word" и "Long", т. е. первый параметр 16-разрядный, а второй — 32-разрядный. Однако так было только в старых, 16-разрядных версиях Windows. В 32-разрядных версиях оба параметра 32-разрядные, несмотря на их названия. Конкретный смысл каждого параметра зависит от сообщения. В некоторых сообщениях один или оба параметра могут вообще не использоваться, в других — наоборот, двух параметров даже не хватает. В этом случае один из параметров (обычно `lParam`) содержит указатель на дополнительные данные. После обработки сообщения оконная процедура должна вернуть какое-то значение. Обычно это значение просто сигнализирует, что сообщение не нуждается в дополнительной обработке, но в некоторых случаях оно более осмысленно, например, `WM_SETICON` должно вернуть дескриптор иконки, которая была установлена ранее.

Прототип оконной процедуры выглядит следующим образом:

```
  LRESULT CALLBACK WindowProc(
  HWND hwnd,       // дескриптор окна
  UINT uMsg,       // номер сообщения
  WPARAM wParam,   // первый параметр сообщения
  LPARAM lParam    // второй параметр сообщения
);
```

В Delphi оконная процедура объявляется следующим образом:

```
function WindowProc(hWnd: HWND; Msg: UINT; wParam: WPARAM;
  lParam: LPARAM): LRESULT; stdcall;
```

Все, что "умеет" окно, определяется тем, как его оконная процедура реагирует на сообщения. Чтобы окно можно было, например, перетаскивать мышью, его оконная процедура должна обрабатывать целый ряд сообщений, связанных с мышью. Чтобы не заставлять программиста каждый раз реализовывать стандартную для всех окон обработку событий, в системе предусмотрена функция `DefWindowProc`. Разработчик приложения в своей оконной процедуре должен предусмотреть только специфическую для данного окна обработку сообщений, а обработку всех остальных сообщений передать этой функции. Существуют также аналоги функции `DefWindowProc` для специализированных окон: `DefDlgProc` для диалоговых окон, `DefFrameProc` для родительских MDI-окон, `DefChildMDIProc` для дочерних MDI-окон.

Сообщение окну можно либо *послать* (post), либо *отправить* (send). Каждая нить, вызвавшая хоть одну функцию из библиотеки user32.dll или gdi32.dll, имеет свою *очередь сообщений*, в которую помещаются все сообщения, посланные окнам, созданным данной нитью (послать сообщение окну можно, например, с помощью функции `PostMessage`). Соответственно, кто-то должен извлекать эти сообщения из очереди и передавать их окнам-адресатам. Это делается с помощью специального цикла, который называется *петлей сообщений* (message loop). В этом непрерывном цикле, который должен реализовать разработчик приложения, сообщения извлекаются из очереди с помощью функции `GetMessage` (реже — `PeekMessage`) и передаются в функцию `DispatchMessage`. Эта функция определяет, какому окну предназначено сообщение, и вызывает его оконную процедуру. Таким образом, простейший цикл обработки сообщений выглядит так, как показано в листинге 1.5.

Листинг 1.5. Простейшая петля сообщений

```
var
 Msg: TMsg;
...
while GetMessage(Msg, 0, 0, 0) do
begin
  TranslateMessage(Msg);
  DispatchMessage(Msg);
end;
```

Блок-схема петли сообщений показана на рис. 1.4.

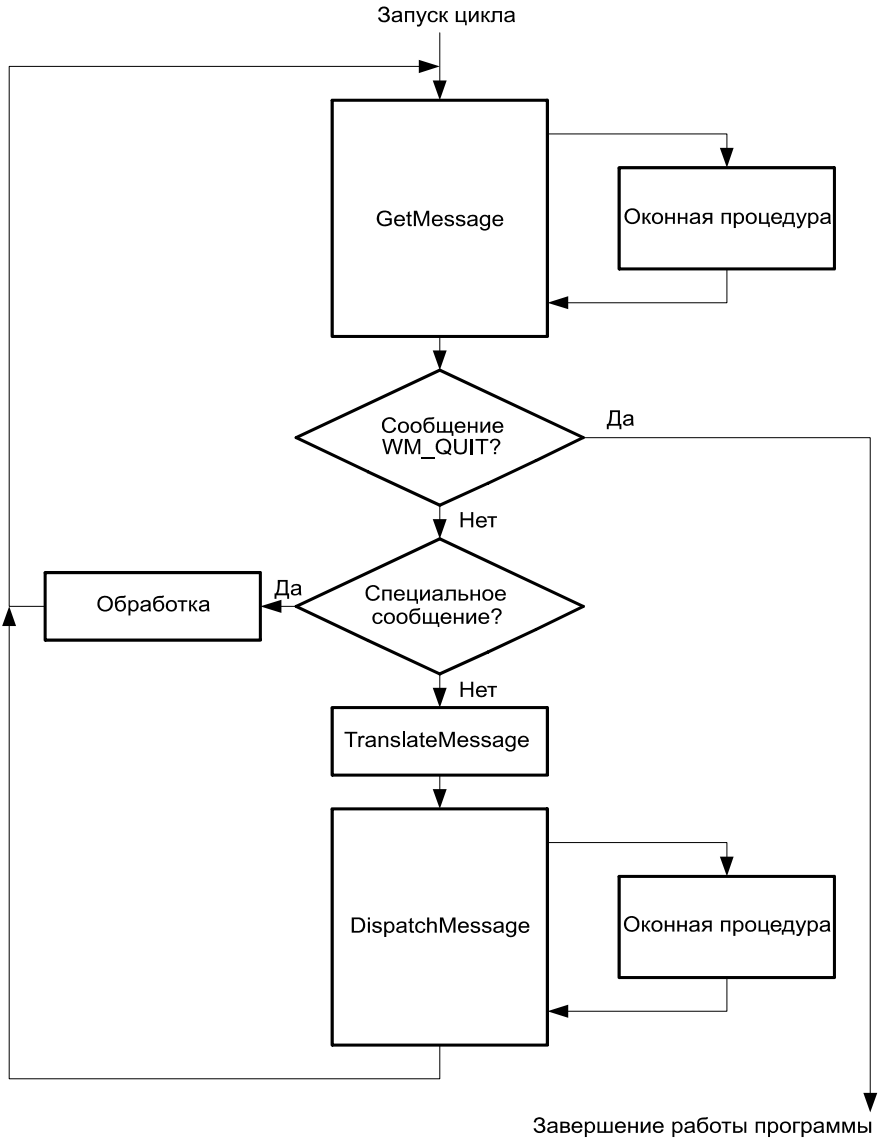


Рис. 1.4. Блок-схема петли сообщений

Функция `GetMessage` возвращает `True` до тех пор, пока не будет получено сообщение `WM_QUIT`, указывающее на необходимость завершения программы. Обычная программа для Windows, выполнив предварительные действия (регистрация класса и создание окна), входит в петлю сообщений, которую выполняет до конца своей работы. Все остальные действия выполняются в оконной процедуре при реакции на соответствующие сообщения.

Примечание

Если нить не имеет петли сообщений, сообщения, которые посылаются ее окнам, не будут обработаны. Это следует учитывать при создании таких компонентов, как, например, TTimer и TClientSocket. Эти компоненты создают невидимые окна для получения сообщений, которые необходимы им для работы. Если нить, создавшая эти объекты, не будет иметь петли сообщений, они будут неработоспособными.

Сообщение, извлеченное из очереди, GetMessage помещает в первый параметр-переменную типа TMsg. Последние три параметра служат для фильтрации сообщений, позволяя извлекать из очереди только те сообщения, которые соответствуют определенным критериям. Если эти параметры равны нулю, как это обычно бывает, фильтрация при извлечении сообщений не производится.

Функция TranslateMessage, которая обычно вызывается в петле сообщений, служит для трансляции клавиатурных сообщений (если петля сообщений реализуется только для обработки сообщений невидимым окнам, которые использует, например, COM/DCOM, или по каким-то другим причинам ввод с клавиатуры не обрабатывается или обрабатывается нестандартным образом, вызов TranslateMessage можно опустить). Когда пользователь нажимает какую-либо клавишу на клавиатуре, система посылает окну, находящемуся в фокусе, сообщение WM_KEYDOWN. Через параметры этого сообщения передается виртуальный код нажатой клавиши — двухбайтное число, которое определяется только положением нажатой клавиши на клавиатуре и не зависит от текущей раскладки, состояния клавиш <CapsLock> и т. п. Функция TranslateMessage, обнаружив такое сообщение, добавляет в очередь (причем не в конец, а в начало) сообщение WM_CHAR, в параметрах которого передается код символа, соответствующего нажатой клавише, с учетом раскладки, состояния клавиш <CapsLock>, <Shift> и т. п. Именно функция TranslateMessage по виртуальному коду клавиши определяет код символа. При этом нажатие любой клавиши приводит к генерации WM_KEYDOWN, а вот WM_CHAR генерируется не для всех клавиш, а только для тех, которые соответствуют какому-то символу (например, не генерирует WM_CHAR нажатие таких клавиш, как <Shift>, <Ctrl>, <Insert>, функциональных клавиш).

Примечание

У многих компонентов VCL есть события OnKeyDown и OnKeyPress. Первое возникает при получении компонентом сообщения WM_KEYDOWN, второе — сообщения WM_CHAR.

Если очередь сообщений пуста, функция GetMessage ожидает, пока там не появится хотя бы одно сообщение, и только после этого завершает работу. Во время этого ожидания нить не загружает процессор.

Петля сообщений может извлечь и отправить на обработку следующее сообщение только тогда, когда оконная процедура закончила обработку предыдущего. Таким образом, сообщение, обработка которого занимает много времени, блокирует обработку следующих сообщений, и все окна, созданные данной нитью, перестают реагировать на действия пользователя. Именно этим объясняется временное зависание программы, которая в одном из своих обработчиков сообщений делает математические расчеты или выполняет длительный запрос к базе данных: сообщения накапливаются в очереди, но не извлекаются из нее и не обрабатываются. Как только обработка текущего сообщения закончится, все остальные сообщения будут извлечены из очереди и обработаны.

В некоторых случаях избежать временного зависания программы помогает организация *локальной петли сообщений*. Если обработчик сообщения, для работы которого требуется много времени, содержит цикл, в него можно вставить вызовы функции `PeekMessage`, которая позволяет проверить, есть ли в очереди сообщения. Если сообщения обнаружены, нужно вызвать `DispatchMessage` для передачи их требуемому окну. В этом случае сообщения будут извлекаться из очереди и обрабатываться до завершения работы обработчика. Блок-схема программы, содержащей локальную петлю сообщений, показана на рис. 1.5 (для краткости в главной петле сообщений показаны только две самых важных функции: `GetMessage` и `DispatchMessage`, хотя и в этом случае главная петля целиком выглядит так же, как на рис. 1.4).

При использовании локальной петли сообщений существует опасность бесконечной рекурсии. Рассмотрим это на простом примере: предположим, что сложный код, содержащий локальную петлю сообщений, выполняется при нажатии некоторой кнопки на форме приложения. Пока обработчик выполняется, нетерпеливый пользователь может снова нажать кнопку, запустив вторую активацию обработчика нажатия кнопки, и так несколько раз. Конечно, организовать таким образом очень глубокую рекурсию пользователь вряд ли сможет (терпения не хватит), но часто даже то, что несколько активаций обработчика вызваны рекурсивно, может привести к неприятным последствиям. А если программа организует локальную петлю сообщений в обработчике сообщений таймера, то здесь рекурсия действительно может углубляться до переполнения стека. Поэтому при организации петли сообщений следует принимать меры против рекурсии. Например, в случае с кнопкой в обработчике ее нажатие можно запретить (`Enabled := False`), и вновь разрешить только после окончания обработки, тогда пользователь не сможет нажать кнопку во время работы локальной петли сообщений.

В очередь можно поставить сообщение, не привязанное ни к какому окну. Это делается с помощью функции `PostThreadMessage`. Такие сообщения необ-

ходимо самостоятельно обрабатывать в петле сообщений, потому что функция `DispatchMessage` их просто игнорирует.

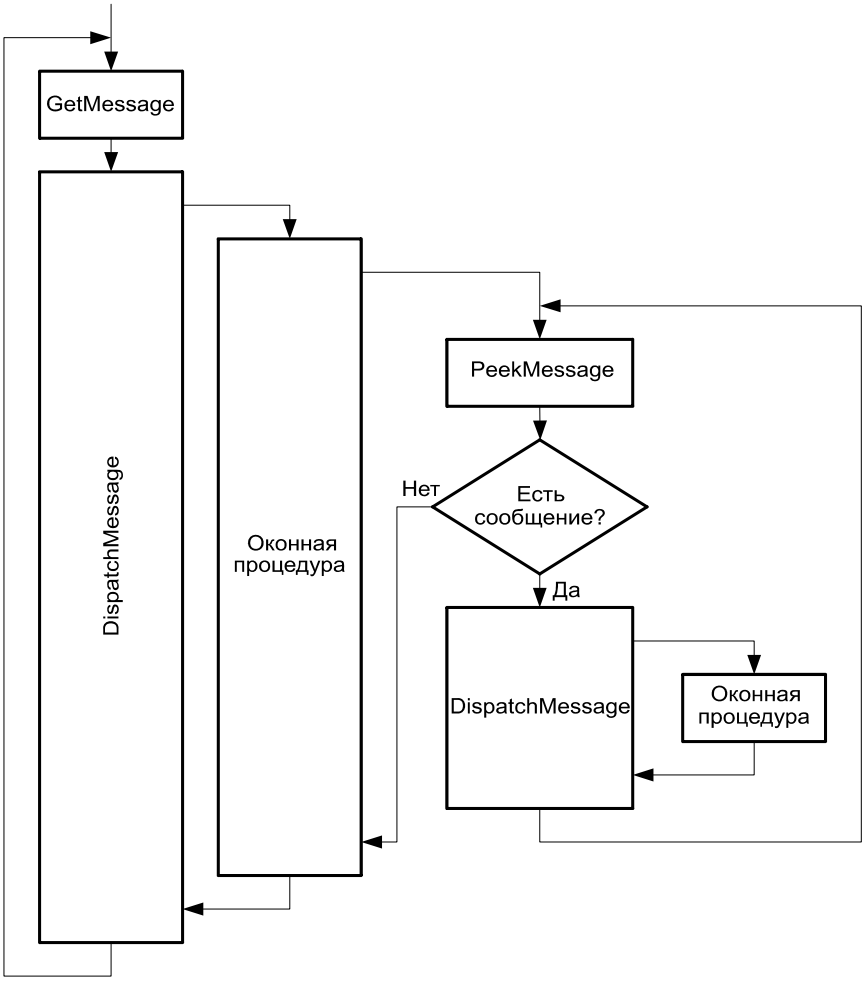


Рис. 1.5. Блок-схема программы с локальной петлей сообщений

Существуют также *широковещательные сообщения*, которые посылаются сразу нескольким окнам. Проще всего послать такое сообщение с помощью функции `PostMessage`, указав в качестве адресата не дескриптор конкретного окна, а константу `HWND_BROADCAST`. Такое сообщение получат все окна, расположенные непосредственно на рабочем столе и не имеющие при этом владельцев (в терминах системы). Существует также специальная функция `BroadcastSystemMessage` (начиная с Windows XP — ее расширенный вариант

BroadcastSystemMessageEx), которая позволяет уточнить, каким конкретно окнам будет отправлено широковещательное сообщение.

Кроме параметров wParam и lParam, каждому сообщению приписывается время отправки и координаты курсора в момент возникновения. Соответствующие поля есть в структуре TMsg, которую используют функции GetMessage и DispatchMessage, но у оконной процедуры не предусмотрены параметры для их передачи. Получить время отправки сообщения и координаты курсора при обработке сообщения можно с помощью функций GetMessageTime и GetMessagePos соответственно.

Существует также ряд функций, которые могут обрабатывать сообщения без участия DispatchMessage и оконной процедуры. Если эти функции распознают сообщение, извлеченное из очереди, как "свое", они сами выполняют все необходимые действия по его обработке, и тогда TranslateMessage и DispatchMessage вызывать не нужно. К этим функциям, в частности, относятся следующие:

- TranslateAccelerator — на основе загруженной из ресурсов таблицы распознает нажатие "горячих" клавиш меню и вызывает оконную процедуру, передавая ей сообщение WM_COMMAND или WM_SYSCOMMAND, аналогичное тому, которое посылается при выборе соответствующего пункта меню пользователем;
- TranslateMDISysAccel — аналог предыдущей функции за исключением того, что распознает "горячие" клавиши системного меню MDI-окон;
- IsDialogMessage — распознает сообщения, имеющие особый смысл для диалоговых окон (например, нажатие калвиши <Tab> для перехода между элементами управления). Используется для немодальных диалоговых окон и окон, не являющихся диалоговыми (т. е. созданными без помощи функций CreateDialogXXXX), но требующими аналогичной функциональности.

Перечисленные функции при необходимости вставляются в петлю сообщений. Листинг 1.6 показывает, как будет выглядеть петля сообщений, содержащая вызов TranslateAccelerator для родительской MDI-формы и TranslateMDISysAccel для дочерней.

Листинг 1.6. Петля сообщений с обработкой "горячих" клавиш главного меню и системного меню MDI-окон

```
while GetMessage(Msg, 0, 0, 0) do
  if not TranslateMDISysAccel(ActiveMDIChildHandle, Msg)
    and not TranslateAccelerator(MDIFormHandle, AccHandle, Msg) then
  begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
  end;
```

При отправке сообщения, в отличие от посылки, оно не ставится в очередь, а передается оконной процедуре напрямую. Отправить сообщение можно, например, с помощью функции SendMessage. Если эта функция вызывается из той же нити, которой принадлежит окно-адресат, то фактически это эквивалентно прямому вызову оконной процедуры. Если окно принадлежит другой нити, данное сообщение становится в отдельную очередь, имеющую более высокий приоритет, чем очередь для посланных сообщений. Функции GetMessage и PeekMessage сначала выбирают все сообщения из этой очереди и отправляют их на обработку, и лишь затем приступают к анализу очереди посланных сообщений.

Примечание

Поскольку сообщения, отправленные окну, передаются оконной процедуре напрямую либо диспетчеризуются внутри GetMessage или PeekMessage, то эти сообщения не попадают в функции TranslateMDISysAccel, TranslateAccelerator и TranslateMessage. Это необходимо учитывать при передаче окну сообщений, эмулирующих нажатие клавиш на клавиатуре. Такие сообщения окну нужно посылать, а не отправлять, чтобы они прошли полный цикл обработки и окно правильно на них отреагировало. Для эмуляции сообщений от клавиатуры можно также воспользоваться функцией keybd_event, но она посылает сообщение не указанному окну, а активному, что не всегда удобно.

Диалоговые окна обрабатывают сообщения по-особому. Эти окна делятся на *модальные* (создаются и показываются с помощью функций DialogBoxXXXX) и *немодальные* (создаются с помощью функций CreateDialogXXXX и затем показываются с помощью функции ShowWindow, использующейся и для обычных, не диалоговых, окон). И модальные, и немодальные окна создаются на основе шаблона, который может храниться в ресурсах приложения или в памяти. В шаблоне можно явно указать имя созданного вами оконного класса диалогового окна или (как это обычно бывает) не указывать его вообще, чтобы был выбран класс, предоставляемый системой для диалоговых окон по умолчанию. Оконная процедура диалогового класса должна передавать необработанные сообщения функции DefDlgProc.

Все диалоговые окна имеют так называемую *диалоговую процедуру* — функцию, указатель на которую передается в качестве одного из параметров функциям DialogBoxXXXX и CreateDialogXXXX. Прототипы диалоговой и оконной процедур совпадают. Функция DefDlgProc начинает свою работу с того, что вызывает диалоговую процедуру. Если та не обработала переданное ей сообщение (о чем сигнализирует возвращаемое нулевое значение), функция DefDlgProc обрабатывает его сама. Таким образом, с помощью одного оконного класса и одной оконной процедуры можно реализовывать различные диалоговые окна, используя разные диалоговые процедуры.

Функции DialogBoxXXXX создают диалоговое окно и сразу же показывают его в модальном режиме. Данные функции завершают свое выполнение только тогда, когда модальное окно будет закрыто. Внутри модальных функций организуется собственная петля сообщений. Все прочие окна на время показа модального диалога запрещаются (как если бы для них была вызвана функция EnableWindow с параметром FALSE), т. е. перестают реагировать на сообщения от мыши и клавиатуры. При этом они сохраняют способность реагировать на другие сообщения, благодаря чему могут, например, обновлять свое содержимое по таймеру (в справке написано, что ничто не мешает программисту вставить в диалоговую процедуру вызов функций, разрешающих запрещенные системой окна, но при этом теряется смысл модальных диалогов).

Если в очереди нет сообщений, модальная петля посылает родительскому окну диалога сообщение WM_ENTERIDLE, обработка которого позволяет этому окну выполнять фоновые действия. Разумеется, что обработчик WM_ENTERIDLE не должен выполняться слишком долго, иначе модальное окно зависнет.

Обычно окно использует оконную процедуру, которая задана при создании соответствующего оконного класса. Однако допускается создание так называемых *подклассов* — переопределение оконной процедуры после того, как окно создано. Это переопределение касается только заданного окна и не оказывает влияния на остальные окна, принадлежащие данному оконному классу. Осуществляется оно с помощью функции SetWindowLong с параметром GWL_WNDPROC (другие значения этого параметра позволяют менять другие свойства окна, такие как стиль и расширенный стиль). Изменять оконную процедуру можно только у окон, созданных самим процессом.

Новая оконная процедура, которая устанавливается при создании подкласса, все необработанные сообщения должна передавать не функции DefWindowProc, а той оконной процедуре, которая была установлена ранее. SetWindowLong при изменении оконной процедуры возвращает дескриптор старой процедуры (этот же дескриптор можно получить, заранее вызвав функцию GetWindowLong с аргументом GWL_WNDPROC). Обычно значение дескриптора численно совпадает с адресом старой оконной процедуры, поэтому в некоторых источниках можно встретить рекомендации использовать этот дескриптор непосредственно как указатель процедурного типа. И это даже будет работать для оконных классов, созданных самой программой. Но безопаснее все же вызов старой оконной процедуры реализовать с помощью системной функции CallWindowProc, предоставив ей "разбираться", является ли дескриптор указателем.

В качестве примера рассмотрим создание подкласса для некоторого окна, дескриптор которого содержится в переменной Wnd. Пусть нам потребовалось для этого окна нестандартным образом обрабатывать сообщение WM_KILLFOCUS.

Тогда код новой оконной процедуры и код ее установки будет выглядеть так, как показано в листинге 1.7.

Листинг 1.7. Создание подкласса для особой обработки сообщения WM_KILLFOCUS

```
var
  OldWndProc: TFNWndProc;

function NewWindowProc(hWnd: HWND; Msg: UINT; wParam: WPARAM;
  lParam: LPARAM): LRESULT; stdcall;
begin
  if Msg = WM_KILLFOCUS then
    // Обработка события
  else
    Result := CallWindowProc(OldWndProc, hWnd, Msg, wParam, lParam);
end;

...
// Установка новой оконной процедуры окну Wnd
OldWndProc := TFNWndProc(SetWindowLong(Wnd, GWL_WNDPROC,
  Longint(@NewWindowProc)));
...
```

Примечание

MSDN называет функции GetWindowLong и SetWindowLong устаревшими и рекомендует использовать вместо них GetWindowLongPtr и SetWindowLongPtr, совместимые с 64-разрядными версиями Windows. Однако до 2007-й версии Delphi включительно эти функции отсутствуют в модуле Windows, и при необходимости их следует импортировать самостоятельно.

Переопределять оконную процедуру с помощью SetWindowLong можно и у тех окон, оконная процедура которых была переопределена ранее. Таким образом создаются цепочки оконных процедур, каждая из которых вызывает предыдущую.

1.1.7. Создание окон средствами VCL

Теперь поговорим о том, как в VCL создаются окна. Речь здесь будет идти не о написании кода для создания окна с помощью VCL (предполагается, что читатель это и так знает), а о том, какие функции API и в какой момент вызывает VCL при создании окна.

Если смотреть код методов класса TWinControl, которые вызываются при создании и отображении окна, то найти там то место, когда окно создается, уда-

ется не сразу. На первый взгляд все выглядит так, будто этот код вообще не имеет отношения к созданию окна, как будто оно создается где-то совсем в другом месте, а TWinControl получает уже готовый дескриптор. На самом деле окно создает, конечно же, сам TWinControl, а спрятано его создание в свойстве Handle. Метод GetHandle, который возвращает значение свойства Handle, выглядит следующим образом (листинг 1.8).

Листинг 1.8. Реализация метода TWinControl.GetHandle

```
procedure TWinControl.HandleNeeded;
begin
  if FHandle = 0 then
  begin
    if Parent <> nil then Parent.HandleNeeded;
    CreateHandle;
  end;
end;

function TWinControl.GetHandle: HWnd;
begin
  HandleNeeded;
  Result := FHandle;
end;
```

При каждом обращении к свойству Handle вызывается метод HandleNeeded, который проверяет, создано ли уже окно, и если нет, создает его, попутно создавая, при необходимости, родительское окно. Таким образом, окно создается при первом обращении к свойству Handle.

Метод CreateHandle, который вызывается из HandleNeeded, выполняет непосредственно лишь несколько вспомогательных операций, а для создания окна вызывает еще один метод — CreateWnd (листинг 1.9).

Листинг 1.9. Реализация метода CreateWnd

```
procedure TWinControl.CreateWnd;
var
  Params: TCreateParams;
  TempClass: TWndClass;
  ClassRegistered: Boolean;
begin
  CreateParams(Params);
  with Params do
```

```
  begin
    if (WndParent = 0) and (Style and WS_CHILD <> 0) then
      if (Owner <> nil) and (csReading in Owner.ComponentState) and
        (Owner is TWinControl) then
        WndParent := TWinControl(Owner).Handle
      else
        raise EInvalidOperation.CreateFmt(SParentRequired, [Name]);
    FDefWndProc := WindowClass.lpfnWndProc;
    ClassRegistered := GetClassInfo(WindowClass.hInstance,
      WinClassName, TempClass);
    if not ClassRegistered or
      (TempClass.lpfnWndProc <> @InitWndProc) then
    begin
      if ClassRegistered then Windows.UnregisterClass(WinClassName,
        WindowClass.hInstance);
      WindowClass.lpfnWndProc := @InitWndProc;
      WindowClass.lpszClassName := WinClassName;
      if Windows.RegisterClass(WindowClass) = 0 then RaiseLastOSError;
    end;
    CreationControl := Self;
    CreateWindowHandle(Params);
    if FHandle = 0 then
      RaiseLastOSError;
    if (GetWindowLong(FHandle, GWL_STYLE) and WS_CHILD <> 0) and
      (GetWindowLong(FHandle, GWL_ID) = 0) then
      SetWindowLong(FHandle, GWL_ID, FHandle);
  end;
  StrDispose(FText);
  FText := nil;
  UpdateBounds;
  Perform(WM_SETFONT, FFont.Handle, 1);
  if AutoSize then AdjustSize;
end;
```

Собственно создание окна опять происходит не здесь, а в методе `CreateWindowHandle`, который очень прост: он состоит из одного только вызова API-функции `CreateWindowEx` с параметрами, значения которых берутся из полей записи `Params` типа `TCreateParams` (листинг 1.10).

Листинг 1.10. Запись `TCreateParams`

```
TCreateParams = record
  Caption: PChar;
```

```
  Style: DWORD;
  ExStyle: DWORD;
  X, Y: Integer;
  Width, Height: Integer;
  WndParent: HWnd;
  Param: Pointer;
  WindowClass: TWndClass;
  WinClassName: array[0..63] of Char;
end;
```

В записи Params хранятся параметры как окна, передаваемые в функцию WindowCreateEx, так и оконного класса (поля WindowClass и WndClassName). Все поля инициализируются методом CreateParams на основе значений свойств оконного компонента. Данный метод виртуальный и может быть перекрыт в наследниках, что бывает полезно, когда необходимо изменить стиль создаваемого окна. Например, добавив расширенный стиль WS_EX_CLIENTEDGE (или, как вариант, WS_EX_STATICEDGE), можно получить окно с необычной рамкой (листинг 1.11).

Листинг 1.11. Перекрытие метода CreateParams

```
procedure TForm1.CreateParams(var Params: TCreateParams);
begin
  // Вызов унаследованного метода для заполнения всех полей
  // записи Params
  inherited CreateParams(Params);
  // Добавляем флаг WS_EX_CLIENTEDGE к расширенному стилю окна
  Params.ExStyle := Params.ExStyle or WS_EX_CLIENTEDGE;
end;
```

Примечание

В *разд. 1.1.4* мы говорили, что имя оконного класса, который VCL создает для оконного компонента, совпадает с именем класса этого компонента. Здесь мы видим, что на самом деле имя оконного класса можно сделать и другим, для этого достаточно изменить значения поля Params.WinClassName.

Обратите внимание, что всем без исключения классам метод CreateWnd назначает одну и ту же оконную процедуру — InitWndProc. Это является основой в обработке сообщений с помощью VCL, именно поэтому оконная процедура назначается не в методе CreateParams, а в методе CreateWnd, чтобы в наследниках нельзя было изменить это поведение (метод CreateWnd тоже виртуальный, но при его переопределении имеет смысл только добавлять какие-то действия, а не изменять поведение унаследованного метода).

Чтобы понять, как работает процедура InitWndProc, обратите внимание на еще одну особенность метода CreateWnd: перед вызовом CreateWindowHandle (т. е. непосредственно перед созданием окна) он записывает ссылку на текущий объект в глобальную переменную CreationControl. Эта переменная затем используется процедурой InitWndProc (листинг 1.12).

Листинг 1.12. Оконная процедура InitWndProc

```
function InitWndProc(HWindow: HWnd; Message, WParam,
  LParam: Longint): Longint;
begin
  CreationControl.FHandle := HWindow;
  SetWindowLong(HWindow, GWL_WNDPROC,
    Longint(CreationControl.FObjectInstance));
  if (GetWindowLong(HWindow, GWL_STYLE) and WS_CHILD <> 0) and
    (GetWindowLong(HWindow, GWL_ID) = 0) then
    SetWindowLong(HWindow, GWL_ID, HWindow);
  SetProp(HWindow, MakeIntAtom(ControlAtom),
    THandle(CreationControl));
  SetProp(HWindow, MakeIntAtom(WindowAtom),
    THandle(CreationControl));

  asm
        PUSH    LParam
        PUSH    WParam
        PUSH    Message
        PUSH    HWindow
        MOV     EAX,CreationControl
        MOV     CreationControl,0
        CALL    [EAX].TWinControl.FObjectInstance
        MOV     Result,EAX
  end;

end;
```

Примечание

Код функции InitWndProc в листинге 1.12 взят из Delphi 7. В более поздних версиях код включает в себя поддержку окон, работающих с кодировкой Unicode, поэтому там предусмотрен выбор между ANSI- и Unicode-вариантами функций API (подробнее об ANSI- и Unicode-вариантах см. *разд. 1.1.12*). Такой код сложнее понять из-за этих дополнений. Кроме того, из листинга 1.12 убрано все, что относится к компиляции под LINUX, чтобы не засорять листинг.

Из листинга 1.12 видно, что оконная процедура `InitWndProc` не обрабатывает сама никаких сообщений, а просто переназначает оконную процедуру у окна. Таким образом, `InitWndProc` для каждого окна вызывается только один раз, чтобы переназначить оконную процедуру. Обработка того сообщения, которое привело к вызову `InitWndProc`, тоже передается в эту новую процедуру (ассемблерная вставка в конце `InitWndProc` делает именно это).

При просмотре этого кода возникают два вопроса. Первый — зачем писать такую оконную процедуру, почему бы не назначить нужную процедуру обычным образом? Здесь все дело в том, что стандартными средствами оконная процедура назначается одна на весь оконный класс, в то время как по внутренней логике VCL каждый экземпляр компонента должен иметь свою собственную оконную процедуру. Добиться этого можно только порождением подкласса уже после создания окна. Указатель на свою уникальную оконную процедуру (откуда эта процедура берется и почему она должна быть уникальной, мы поговорим в следующем разделе) каждый экземпляр хранит в поле `FObjectInstance`. Значение глобальной переменной `CreationControl` присваивается, как мы помним, непосредственно перед созданием окна, а первое свое сообщение окно получает буквально в момент создания. Так как VCL — принципиально однонитевая библиотека, ситуация, когда другой код вклинивается между присваиванием значения переменной `CreationControl` и вызовом `InitWndProc`, невозможна, так что в `InitWndProc` попадает правильная ссылка на создаваемый объект.

Второй вопрос — зачем так сложно? Почему в методе `CreateWnd` сразу после создания окна нельзя было вызвать `SetWindowLong` и установить нужную оконную процедуру там, вместо того чтобы поручать это процедуре `InitWndProc`? Здесь ответ такой: это сделано потому, что свои первые несколько сообщений (например, сообщения `WM_CREATE` и `WM_NCCREATE`) окно получает до того, как функция `CreateWindowEx` завершит свою работу. Чтобы завершить создание окна, `CreateWindowEx` отправляет несколько сообщений окну, и только после того как окно обработает их должным образом, процесс создания окна считается завершенным. Так что назначать уникальную оконную процедуру после завершения `CreateWindowEx` — это слишком поздно. Именно поэтому уникальная оконная процедура назначается таким неочевидным и несколько неуклюжим способом.

1.1.8. Обработка сообщений с помощью VCL

При использовании VCL в простых случаях самостоятельно работать с оконными сообщениями нет нужды, поскольку практически все можно сделать с помощью свойств, методов и событий компонентов. Тем не менее, некоторые сообщения приходится обрабатывать вручную. Чаще всего это приходится

делать при разработке собственных компонентов, но и в обычных приложениях это также может быть полезным.

Кроме сообщений, предусмотренных в системе, компоненты VCL обмениваются сообщениями, созданными авторами этой библиотеки. Эти сообщения имеют префиксы `CM_` и `CN_`. Они нигде не документированы, разобраться с ними можно только по исходным кодам VCL. При разработке собственных компонентов приходится обрабатывать эти сообщения, которые мы здесь не будем полностью описывать, но некоторые из них будут упоминаться в описании работы VCL с событиями.

В Windows API нет понятия главного окна — все окна, не имеющие родителя (или владельца в терминах системы), равноценны, и приложение может продолжать работу после закрытия любых окон. Но в VCL введено понятие *главной формы*: форма, которая создается первой, становится главной, и ее закрытие означает закрытие всего приложения.

Если окно не имеет ни родителя, ни владельца в терминах системы (такие окна называются *окнами верхнего уровня*), то на панели задач появляется кнопка, связанная с этим окном (окно, имеющее владельца, также может обзавестись такой кнопкой, если оно создано со стилем `WS_EX_APPWINDOW`). Обычно в приложении одно окно главного уровня, и оно играет роль главного окна этого приложения, хотя система не запрещает приложению создавать несколько окон верхнего уровня (примеры — Internet Explorer, Microsoft Word). Разработчики VCL пошли по другому пути: окно верхнего уровня, ответственное за появление кнопки на панели задач, создается объектом `Application`. Дескриптор этого окна хранится в свойстве `Application.Handle`, а само оно невидимо, т. к. имеет нулевые размеры. Как и любое другое, это окно имеет оконную процедуру и может обрабатывать сообщения. Главная форма — это отдельное окно, не имеющее, с формальной точки зрения, никакого отношения к кнопке на панели задач. Видимость связи между этой кнопкой и главной формой обеспечивается взаимодействием объекта `Application` и объекта главной формы внутри VCL. Таким образом, даже простейшее VCL-приложение создает два окна: невидимое окно объекта `Application` и окно главной формы. Окно, создаваемое объектом `Application`, мы будем называть *невидимым окном приложения*. Невидимое окно приложения по умолчанию становится владельцем (в терминах системы) всех форм, у которых явно не установлено свойство `Parent`, в том числе и главной формы.

При обработке сообщений VCL решает две задачи: выборка сообщений из очереди и передача сообщения конкретному компоненту. Рассмотрим сначала первую задачу.

Выборкой сообщений из очереди занимается объект `Application`, непосредственно за извлечение и диспетчеризацию сообщения отвечает его метод `ProcessMessage` (листинг 1.13).

Листинг 1.13. Метод TApplication.ProcessMessage

```
function TApplication.ProcessMessage(var Msg: TMsg): Boolean;
var
  Unicode: Boolean;
  Handled: Boolean;
  MsgExists: Boolean;
begin
  Result := False;
  if PeekMessage(Msg, 0, 0, 0, PM_NOREMOVE) then
  begin
    Unicode := (Msg.hwnd <> 0) and IsWindowUnicode(Msg.hwnd);
    if Unicode then
      MsgExists := PeekMessageW(Msg, 0, 0, 0, PM_REMOVE)
    else
      MsgExists := PeekMessage(Msg, 0, 0, 0, PM_REMOVE);
    if not MsgExists then Exit;
    Result := True;
    if Msg.Message <> WM_QUIT then
    begin
      Handled := False;
      if Assigned(FOnMessage) then FOnMessage(Msg, Handled);
      if not IsPreProcessMessage(Msg) and not IsHintMsg(Msg) and
        not Handled and not IsMDIMsg(Msg) and
        not IsKeyMsg(Msg) and not IsDlgMsg(Msg) then
      begin
        TranslateMessage(Msg);
        if Unicode then
          DispatchMessageW(Msg)
        else
          DispatchMessage(Msg);
      end;
    end
    else
      FTerminate := True;
  end;
end;
```

В этом коде отдельного комментария требует то, как используется функция PeekMessage. Сначала эта функция вызывается с параметром PM_NOREMOVE, — так выполняется проверка условия, что в очереди присутствует сообщение, а также выясняется, для какого окна предназначено первое сообщение в очереди. Само сообщение при этом остается в очереди. С помощью функции IsWindowUnicode производится проверка, использует ли окно-адресат коди-

ровку ANSI или Unicode, и затем, в зависимости от этого, сообщение извлекается либо функцией `PeekMessage`, либо ее Unicode-аналогом `PeekMessageW` (о Unicode-аналогах функций см. *разд. 1.1.12*). При диспетчеризации сообщения также вызывается либо функция `DispatchMessage`, либо ее Unicode-аналог `DispatchMessageW`.

Если метод `ProcessMessage` с помощью `PeekMessage` извлекает из очереди сообщение `WM_QUIT`, то он устанавливает в `True` поле `FTerminate` и завершает свою работу. Обработка всех остальных сообщений, извлеченных из очереди, состоит из следующих основных этапов (см. рис. 1.6):

1. Если назначен обработчик `Application.OnMessage`, сообщение передается ему. В этом обработчике можно установить параметр-переменную `Handled` в `True`, что означает, что сообщение не нуждается в дополнительной обработке.
2. Второй шаг — это предварительная обработка сообщения (вызов метода `IsPreProcessMessage`). Этот шаг появился только начиная с BDS 2006, в более ранних версиях его не было. Обычно предварительную обработку осуществляет то окно, которому предназначено это сообщение, но если окно-адресат не является VCL-окном, производится поиск VCL-окна по цепочке родителей. Кроме того, если какое-либо окно захватит ввод мыши, предварительную обработку сообщений будет осуществлять именно оно. Если оконный компонент, удовлетворяющий этим требованиям, найден, вызывается его метод `PreProcessMessage`, который возвращает результат логического типа. Если компонент вернул `True`, то на этом обработка сообщения заканчивается. Отметим, что ни один из стандартных компонентов VCL не использует эту возможность перехвата сообщений, она реализована для сторонних компонентов.
3. Затем, если на экране присутствует *всплывающая подсказка* (hint), проверяется, должно ли пришедшее сообщение прятать эту подсказку, и если да, то она убирается с экрана (метод `IsHintMessage`). Список сообщений, которые должны прятать окно подсказки, зависит от класса этого окна (здесь имеется в виду класс VCL, а не оконный класс) и определяется виртуальным методом `THintWindow.IsHintMsg`. Стандартная реализация этого метода рассматривает как "прячущие" все сообщения от мыши, клавиатуры, сообщения об активации и деактивации программы и о действиях пользователя с меню или визуальными компонентами. Если метод `IsHintMessage` возвращает `False`, то сообщение дальше не обрабатывается, но стандартная реализация этого метода всегда возвращает `True`.
4. Далее проверяется значение параметра `Handled`, установленное в обработчике `OnMessage` (если он назначен). Если это значение равно `True`, метод

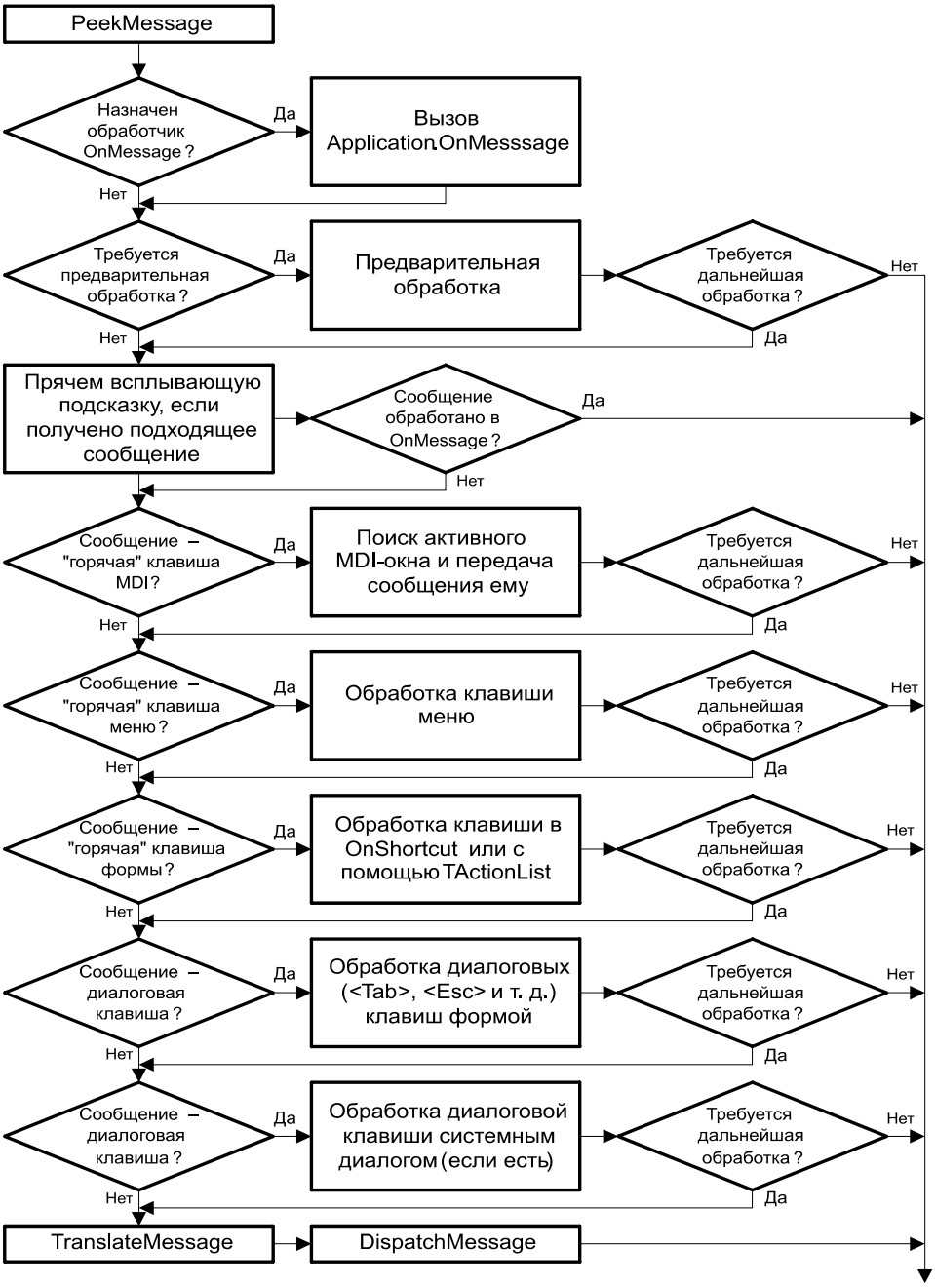


Рис. 1.6. Одна итерация петли сообщений VCL
(блок-схема метода `Application.ProcessMessage`)

ProcessMessage завершает свою работу, и обработка сообщения на этом заканчивается. Таким образом, обработка сообщения по событию OnMessage не может отменить предварительную обработку сообщения и исчезновение всплывающей подсказки.

5. Если главная форма приложения имеет стиль MDIForm, и одно из его дочерних MDI-окон в данный момент активно, сообщение передается функции TranslateMDISysAccel. Если эта функция вернет True, то обработка сообщения на этом завершается (все эти действия выполняются в методе IsMDIMsg).
6. Затем, если получено клавиатурное сообщение, оно отправляется на предварительную обработку тому же окну, что и в пункте 2 (метод IsKeyMsg). Предварительная обработка клавиатурного сообщения начинается с попытки найти полученную комбинацию клавиш среди "горячих" клавиш контекстно-зависимого меню и выполнить соответствующую команду. Если контекстно-зависимое меню не распознало сообщение как свою "горячую" клавишу, то вызывается обработчик события OnShortCut окна, осуществляющего предварительную обработку (если это окно не является формой и не имеет этого события, то вызывается OnShortCut его родительской формы). Если обработчик OnShortCut не установил свой параметр Handled в True, полученная комбинация клавиш ищется среди "горячих" клавиш сначала главного меню, а потом — среди компонентов TActionList. Если и здесь искомая комбинация не находится, возникает событие Application.OnShortCut, которое также имеет параметр Handled, позволяющий указать, что сообщение в дополнительной обработке не нуждается. Если обработчик не установил этот параметр, то сообщение передается главной форме приложения, которое пытается найти нажатую комбинацию среди "горячих" клавиш своего контекстного меню, передает его обработчику OnShortCut, ищет среди "горячих" клавиш главного меню и компонентов TActionList. Если нажатая клавиша не является "горячей", но относится к клавишам, использующимся для управления диалоговыми окнами (<Tab>, стрелки, <Esc> и т. п.), форме передается сообщение об этом, и при необходимости сообщение обрабатывается. Таким образом, на данном этапе средствами VCL эмулируются функции TranslateAccelerator и IsDialogMessage.
7. Если на экране присутствует один из стандартных диалогов (в VCL они реализуются классами TOpenDialog, TSaveDialog и т. п.), то вызывается функция IsDialogMessage, чтобы эти диалоги могли нормально функционировать (метод IsDlgMsg).
8. Если ни на одном из предыдущих этапов сообщение не было обработано, то вызываются функции TranslateMessage и DispatchMessage, которые за-

вершают обработку сообщения путем направления его соответствующей оконной функции.

Примечание

Если внимательно проанализировать шестой этап обработки сообщения, видно, что нажатая комбинация клавиш проверяется на соответствие "горячим" клавишам меню сначала активной формы, затем — главной. При этом сначала возникает событие OnShortCut активной формы, потом — Application.OnShortCut, затем — OnShortCut главной формы. Если в момент получения сообщения главная форма активна, то она дважды будет проверять соответствие клавиши "горячим" клавишам своих меню, и событие OnShortCut тоже возникнет дважды (первый раз поле Msg.Msg равно CN_KEYDOWN, второй — CM_APPKEYDOWN). Эта проверка осуществляется дважды только в том случае, если комбинация клавиш не распознается как "горячая" клавиша — в противном случае цепочка проверок обрывается при первой проверке.

Метод ProcessMessage возвращает True, если сообщение извлечено и обработано, и False, если очередь была пуста. Этим пользуется метод HandleMessage, который вызывает ProcessMessage и, если тот вернет False, вызывает метод Application.Idle для низкоприоритетных действий, которые должны выполняться только при отсутствии сообщений в очереди. Метод Idle, во-первых, проверяет, над каким компонентом находится курсор мыши, и сохраняет ссылку на него в поле FMouseControl, которое используется при последующей проверке, нужно ли прятать всплывающую подсказку. Затем, при необходимости, прячется старая всплывающая подсказка и показывается новая. После этого вызывается обработчик Application.OnIdle, если он назначен. Этот обработчик имеет параметр Done, по умолчанию равный True. Если в коде обработчика он не меняется на False, метод Idle инициирует события OnUpdate у всех объектов TAction, у которых они назначены (если Done после вызова принял значение False, HandleMessage не тратит время на инициацию событий OnUpdate).

Примечание

В BDS 2006 появилось свойство Application.ActionUpdateDelay, позволяющее снизить нагрузку на процессор, откладывая на некоторое время обновление объектов TAction. Если значение этого свойства не равно нулю, в методе Idle вместо вызова OnUpdate запускается таймер и OnUpdate вызывается по его сигналу.

Затем, независимо от значения Done, с помощью процедуры CheckSynchronize проверяется, есть ли записи в списке методов, ожидающих синхронизации (эти методы помещаются в указанный список при вызове TThread.Synchronize). Если список не пуст, выполняется первый из этих методов (при этом он, разумеется, удаляется из списка). Затем, если Done остал-

ся равным True, а список методов для синхронизации был пуст (т. е. никаких дополнительных действий выполнять не нужно), HandleMessage вызывает функцию Windows API WaitMessage. Эта функция приостанавливает выполнение нити до тех пор, пока в ее очереди не появятся сообщения.

Примечание

Вызов Synchronize приводит к тому, что соответствующий метод будет выполнен основной нитью приложения, а нить, вызвавшая Synchronize, будет приостановлена до тех пор, пока главная нить не сделает это. Отсюда видно, насколько бредовыми являются советы (заполонившие Интернет, а также встречающиеся в некоторых книгах, например, у Архангельского) помещать весь код нити в Synchronize. В этом случае дополнительная нить вообще не будет ничего делать, все будет выполняться основной нитью, и выигрыша от создания дополнительной нити просто не будет. Поэтому в Synchronize нужно помещать только те действия, которые не могут быть выполнены неосновной нитью (например, обращения к свойствам и методам VCL-компонентов).

Главная петля сообщений в VCL реализуется методом Application.Run, вызов которого автоматически вставляется в dpr-файл VCL-проекта. Application.Run вызывает в цикле метод HandleMessage, пока поле FTerminate не окажется равным True (напомним, что значение True присваивается этому полю, когда ProcessMessage извлекает из очереди сообщение WM_QUIT, а также при обработке сообщения WM_ENDSESSION и при закрытии главной формы).

Для организации локальной петли сообщений существует метод Application.ProcessMessages. Он вызывает ProcessMessage до тех пор, пока очередь не окажется пустой. Вызов этого метода рекомендуется вставлять в обработчики событий, которые работают долго, чтобы в это время программа не теряла способности реагировать на действия пользователя.

Из сказанного может сложиться впечатление, что главная нить проверяет список методов синхронизации только в главной петле сообщений, когда вызывается метод Idle. На самом деле это не так. Модуль Classes содержит переменную WakeMainThread, хранящую указатель на метод, который вызывается при помещении нового метода в список синхронизации. В конструкторе TApplication этой переменной присваивается указатель на метод TApplication.WakeMainThread, который посылает сообщение WM_NULL невидимому окну приложения. Сообщение WM_NULL — это "пустое" сообщение, на которое окно не должно реагировать (оно используется, например, при перехвате сообщений ловушкой: ловушка не может запретить передачу окну сообщения, но может изменить его на WM_NULL, чтобы окно проигнорировало сообщение). Невидимое окно приложения, тем не менее, не игнорирует это сообщение, а вызывает при его получении CheckSynchronize. Таким образом, синхронное выполнение метода не откладывается до вызова Idle, а выполня-

ется достаточно быстро, в том числе и в локальной петле сообщений. Более того, если главная нить перешла в режим ожидания получения сообщения (через вызов WaitMessage), то вызов Synchronize в другой нити прервет это ожидание, т. к. в очередь будет поставлено сообщение WM_NULL.

Процедура CheckSynchronize и переменная WakeMainThread позволяют обеспечить синхронизацию и в тех приложениях, которые не используют VCL в полном объеме. Разработчику приложения необходимо обеспечить периодические вызовы функции CheckSynchronize из главной нити, чтобы можно было вызывать TThread.Synchronize в других нитях. При этом в главной нити можно обойтись без петли сообщений. Присвоение переменной WakeMainThread собственного метода позволяет реализовать специфичный для данного приложения способ ускорения вызова метода в главной нити.

Примечание

Описанный здесь способ синхронизации работы нитей появился, начиная с шестой версии Delphi. В более ранних версиях списка методов для синхронизации не было. Вместо этого в главной нити создавалось специальное невидимое окно, а метод TThread.Synchronize с помощью SendMessage посылал этому окну сообщение CM_EXECPROC с адресом объекта, метод которого нуждался в синхронизации. Метод выполнялся в оконной процедуре данного окна при обработке этого сообщения. Такой механизм также позволял осуществить синхронизацию в приложениях без VCL, но требовал обязательного наличия петли сообщений в главной нити и не давал возможности выполнять синхронизацию, пока главная нить находилась в локальной петле сообщений. Из-за смены механизма синхронизации могут возникнуть проблемы при переносе в новые версии старых приложений: если раньше для обеспечения работы синхронизации было достаточно организовать петлю сообщений, то теперь необходимо найти место для вызова CheckSynchronize. Разумеется, при переносе полноценных VCL-приложений эти проблемы не возникают, т. к. все, что нужно, содержится в методах класса TApplication.

Принятый в Delphi 6 способ синхронизации получил дальнейшее развитие в BDS 2006. В классе TThread появился метод Queue для передачи в код главной нити вызов метода для асинхронного выполнения, т. е. такого, когда нить, вызвавшая Queue, после этого продолжает работать, не дожидаясь, пока главная нить выполнит требуемый код. Главная нить выполняет этот код параллельно тогда, когда для этого предоставляется случай (информация получена из анализа исходных кодов модулей VCL, т. к. справка Delphi, к сожалению, не описывает данный метод: в справке BDS 2006 он вообще не упомянут, в справке Delphi 2007 упомянут, но все описание состоит из одной фразы "This is Queue, a member of class TThread"). Метод Queue использует тот же список методов синхронизации, что и Synchronize, только элементы этого списка пополнились признаком асинхронного выполнения, и процедура CheckSynchronize не уведомляет нить, поместившую метод в список, о его выполнении, если метод помещен в список синхронизации методом Queue. А метод TThread.RemoveQueuedEvents позволяет удалять из списка методов синхронизации асинхронные вызовы, если нужда в их выполнении отпала.

При показе VCL-формы в модальном режиме выборка сообщений из очереди осуществляется особым образом. Модальные окна в VCL — это не то же самое, что модальные диалоги с точки зрения API. Диалог может быть создан только на основе шаблона, и его модальность обеспечивается самой операционной системой, а VCL допускает модальность для любой формы, позволяя разработчику не быть ограниченным возможностями предусмотренного системой шаблона. Достигается это следующим образом: при вызове метода `ShowModal` все окна запрещаются средствами VCL, затем окно показывается обычным образом, как немодальное, но из-за того, что все остальные окна запрещены, создается эффект модальности.

Внутри `ShowModal` создается своя петля сообщений. В этой петле в цикле вызывается метод `Application.HandleMessage` до тех пор, пока не будет установлено свойство `ModalResult` или не придет сообщение `WM_QUIT`. После завершения этой петли вновь разрешаются все окна, которые были разрешены до вызова `ShowModal`, а "модальная" форма закрывается. В отличие от системных модальных диалогов модальная форма VCL во время своей активности не посылает родительскому окну сообщение `WM_ENTERIDLE`, но благодаря тому, что "модальная" петля сообщений использует `HandleMessage`, будет вызываться `Idle`, а значит, будет возникать событие `Application.OnIdle`, которое позволит выполнять фоновые действия.

Теперь рассмотрим, как VCL обрабатывает извлеченные из очереди сообщения. Как уже было сказано ранее, для каждого класса формы VCL регистрирует одноименный оконный класс, а все окна, принадлежащие одному оконному классу, имеют общую оконную процедуру. С другой стороны, логика работы VCL требует, чтобы события обрабатывались тем экземпляром объекта, который инкапсулирует окно-адресат. Таким образом, возникает вопрос о том, как передать сообщение заданному экземпляру класса VCL.

VCL решает эту задачу следующим образом. Модуль `Classes` содержит недокументированную функцию `MakeObjectInstance`, описанную так:

```
type
  TWndMethod = procedure(var Message: TMessage) of object;

function MakeObjectInstance(Method: TWndMethod): Pointer;
```

Тип `TMessage` хранит информацию о сообщении. Все методы VCL-компонентов, связанные с обработкой сообщения, используют этот тип (чуть позже мы рассмотрим его более подробно).

Функция `MakeObjectInstance` динамически формирует новую оконную процедуру и возвращает указатель на нее (следовательно, любое VCL-приложение содержит самомодифицирующийся код). Задача этой динамически соз-

данной процедуры — передать управление тому методу, который был указан при вызове MakeObjectInstance (таким образом, различные оконные процедуры, сформированные этой функцией, отличаются только тем, метод MainWndProc какого экземпляра класса они вызывают).

Каждый экземпляр оконного компонента создает свою оконную процедуру, которая передает обработку сообщения его методу MainWndProc. Указатель на эту процедуру записывается в поле FObjectInstance. Как мы уже говорили в предыдущем разделе, при регистрации оконного класса в качестве оконной процедуры указывается InitWndProc, которая при получении первого сообщения создает подкласс, и оконной процедурой назначается та, указатель на которую хранится в поле FObjectInstance, т. е. функция, созданная с помощью MakeObjectInstance (см. листинг 1.12). Таким образом, каждый экземпляр получает свою оконную процедуру, а обработку сообщения начинает метод MainWndProc.

MainWndProc — это невиртуальный метод, обеспечивающий решение технических вопросов: удаление "мусора", оставшегося при обработке сообщения и обработку исключений. Собственно обработку сообщения он передает методу, на который указывает свойство WindowProc. Это свойство имеет тип TWndMethod и по умолчанию указывает на виртуальный метод WndProc. Таким образом, если разработчик не изменял значения свойства WindowProc, обработкой сообщения занимается WndProc.

Метод WndProc обрабатывает только те сообщения, которые должны быть обработаны специальным образом, чтобы поддержать функциональность VCL. Особым образом метод WndProc обрабатывает сообщения от мыши: он следит, в границы какого визуального компонента попадают координаты "мышиных" сообщений, и если этот компонент отличается от того, в чью область попало предыдущее сообщение, компоненту из предыдущего сообщения дается команда обработать сообщение CM_MOUSELEAVE, а новому — сообщение CM_MOUSEENTER. Это обеспечивает реакцию визуальных компонентов на приход и уход мыши (в частности, генерирование событий OnMouseEnter и OnMouseExit). Необходимость реализации такого способа отслеживания прихода и ухода мыши вместо использования системных сообщений WM_MOUSEHOVER и WM_MOUSELEAVE связана с тем, что системные сообщения пригодны только для работы с окнами, а VCL отслеживает приход и уход мыши и на неоконные визуальные компоненты. Впрочем, WM_MOUSELEAVE в WndProc тоже служит дополнительным средством проверки ухода мыши.

Примечание

Описанный здесь способ отслеживания ухода и прихода мыши реализован, начиная с BDS 2006. В более ранних версиях Delphi за это отвечал метод Application.Idle, который, как мы помним, вызывается только тогда, когда в

очереди нет сообщений. Из-за этого иногда (например, при быстром движении мышью) события ухода и прихода мыши пропускались, нарушая логику работы программы. Поэтому в BDS 2006 способ контроля прихода и ухода мыши был изменен, и ответственность за это возложена на метод `TWinControl.WndProc`. Это позволило избавиться от одного недостатка — потери событий, но породило другой: теперь перехват и самостоятельная обработка "мышиных" сообщений до того, как это сделает метод `WndProc`, может привести к потере возможности отслеживания прихода и ухода мыши. Впрочем, эта проблема проявляется только при выполнении программистом определенных осмысленных действий по внедрению кода в оконную процедуру, поэтому она гораздо менее серьезна, чем та, от которой удалось избавиться.

События мыши метод `WndProc` диспетчеризует самостоятельно, без помощи функции `DispatchMessage`. Это связано с тем, что `DispatchMessage` передает сообщение тому оконному компоненту, которому оно предназначено с точки зрения системы. Однако с точки зрения VCL этот компонент может являться родителем для неоконных визуальных компонентов, и если сообщение от мыши связано с их областью, то оно должно обрабатываться соответствующим неоконным компонентом, а не его оконным родителем. `DispatchMessage` ничего о неоконных компонентах не "знает" и не может передать им сообщения, поэтому разработчикам VCL пришлось реализовывать свой способ.

Те сообщения, которые метод `WndProc` не обрабатывает самостоятельно (а их — подавляющее большинство), он передает в метод `Dispatch`, который объявлен и реализован в классе `TObject`. На первый взгляд может показаться странным, что в самом базовом классе реализована функциональность, использующаяся только в визуальных компонентах. Эта странность объясняется тем, что разработчики Delphi встроили поддержку обработки сообщений непосредственно в язык. Методы класса, описанные с директивой `message`, служат специально для обработки сообщений. Синтаксис описания такого метода следующий:

```
procedure <Name>(var Message: <TMsgType>); message <MsgNumber>;
```

`<MsgNumber>` — это номер сообщения, для обработки которого предназначен метод. Имя метода может быть любым, но традиционно оно совпадает с именем константы сообщения за исключением того, что в нем выбран более удобный регистр символов и отсутствует символ "_" (например, метод для обработки `WM_SIZE` будет называться `WMSize`).

В качестве типа параметра `<TMsgType>` компилятор разрешает любой тип, но на практике имеет смысл только использование типа `TMessage` или "совместимого" с ним. Тип `TMessage` описан в листинге 1.14.

Листинг 1.14. Описание типа `TMessage`

```
TMessage = packed record
  Msg: Cardinal;
```

```
  case Integer of
    0: (
      WParam: Longint;
      LParam: Longint;
      Result: Longint);
    1: (
      WParamLo: Word;
      WParamHi: Word;
      LParamLo: Word;
      LParamHi: Word;
      ResultLo: Word;
      ResultHi: Word);
end;
```

Поле `Msg` содержит номер сообщения, поля `WParam` и `LParam` — значение одноименных параметров сообщения. Поле `Result` — выходное: метод, осуществляющий окончательную обработку сообщения, заносит в него то значение, которое должна вернуть оконная процедура. Поля с суффиксами `Lo` и `Hi` позволяют обращаться отдельно к младшему и старшему словам соответствующих полей, что может быть очень полезно, когда эти параметры содержат пару 16-разрядных значений. Например, у сообщения `WM_NCMOUSEMOVE` младшее слово параметра `LParam` содержит X-координату мыши, старшее — Y-координату. В случае обработки этого сообщения поле `LParamLo` будет содержать X-координату, `LParamHi` — Y-координату.

"Совместимыми" с `TMessage` можно назвать структуры, которые имеют такой же размер, а также параметр `Msg`, задающий сообщение. Эти структуры учитывают специфику конкретного сообщения. Их имена образуются из имени сообщения путем отбрасывания символа _ и добавления префикса *T*. Для уже упоминавшегося сообщения `WM_NCMOUSEMOVE` соответствующий тип выглядит, как показано в листинге 1.15.

Листинг 1.15. Тип `TWMNCMouseMove`

```
TWMNCMouseMove = packed record
  Msg: Cardinal;
  HitTest: Longint;
  XCursor: Smallint;
  YCursor: Smallint;
  Result: Longint;
end;
```

Параметр `WParam` переименован в `HitTest`, что лучше отражает его смысл в данном случае, а параметр `LParam` разбит на две 16-разрядных части: `XCursor` и `YCursor`.

Параметр метода для обработки сообщения имеет тип, соответствующий обрабатываемому сообщению (при необходимости можно описать свой тип), или тип TMessage. Таким образом, обработчик сообщения WM_NCMOUSEMOVE будет выглядеть так, как показано в листинге 1.16.

Листинг 1.16. Объявление и реализация метода для обработки сообщения WM_NCMOUSEMOVE

```
type
  TSomeForm = class(TForm)
    ................
    procedure WMNCMouseMove(var Message: TWMNCMouseMove);
      message WM_NCMOUSEMOVE;
    ................
  end;

procedure TSomeForm.WMNCMouseMove(var Message: TWMNCMouseMove);
begin
  ..............
  inherited; // Возможно, этот вызов не будет нужен
end;
```

Метод для обработки сообщения может выполнить ее полностью самостоятельно, тогда он не должен вызывать унаследованный метод обработки сообщения. Если же реакция предка на сообщение в целом устраивает разработчика, но нуждается только в дополнении, ключевое слово inherited позволяет вызвать унаследованный обработчик для данного сообщения. Таким образом, может образовываться целая цепочка вызовов унаследованных обработчиков одного и того же сообщения, каждый из которых выполняет свою часть обработки. Если у предков класса нет обработчика данного сообщения, директива inherited передает управление методу TObject.DefaultHandler.

Вернемся к методу Dispatch. Он ищет среди обработчиков сообщения класса (собственных или унаследованных) метод для обработки сообщения, заданного полем Msg параметра Message и, если находит, передает управление ему. Если ни сам класс, ни его предки не содержат обработчика данного сообщения, то обработка передается методу DefaultHandler.

Метод DefaultHandler виртуальный, в классе TObject он не выполняет никаких действий, но наследники его переопределяют. Впервые он переопределяется в классе TControl для обработки сообщений, связанных с получением и установкой заголовка окна — WM_GETTEXT, WM_GETTEXTLENGTH и WM_SETTEXT. Напомним, что класс TControl является предком для всех визуальных компонентов, а не только оконных, и появление обработчика системных сообщений в

этом классе — часть той имитации обработки сообщений неоконными компонентами, о которой мы уже говорили.

В классе `TWinControl` метод `DefaultHandler` также переопределен. Помимо передачи некоторых сообщений дочерним окнам (об этом мы будем подробнее говорить чуть позже) и обработки некоторых внутренних сообщений он вызывает оконную процедуру, адрес которой хранится в свойстве `DefWndProc`. Это свойство содержит адрес, который был присвоен полю `WindowClass.lpfnWndProc` структуры `TCreateParams` в методе `CreateParams`. По умолчанию это поле содержит адрес стандартной оконной процедуры `DefWindowProc`. Как было сказано ранее, обработка сообщений при использовании API обычно завершается вызовом этой процедуры.

В классе `TCustomForm` метод `DefaultHandler` также переопределен: если форма является MDI-формой, сообщения, присланные ей, передаются в процедуру `DefFrameProc` (за исключением `WM_SIZE`, которое передается в `DefWindowProc`) независимо от того, какое значение имеет свойство `DefWndProc`. Для всех остальных типов форм вызывается унаследованный от `TWinControl` `DefaultHandler`.

Повторим еще раз всю цепочку обработки сообщений оконными компонентами VCL (рис. 1.7). Для каждого компонента создается уникальная оконная процедура, которая передает управление методу `MainWndProc`. `MainWndProc` передает управление методу, указатель на который хранится в свойстве `WindowProc`. По умолчанию это метод компонента `WndProc`. Он осуществляет обработку некоторых сообщений, но в большинстве случаев передает управление методу `Dispatch`, который ищет среди методов компонента или его предков обработчик данного сообщения. Если обработчик не найден, управление получает метод `DefaultHandler` (он может также получить управление и в том случае, если обработчик найден, но он вызывает `inherited`). `DefaultHandler` самостоятельно обрабатывает некоторые сообщения, но большинство из них передается оконной процедуре, адрес которой хранится в свойстве `DefWndProc` (по умолчанию это стандартная функция Windows API `DefWindowProc`).

Класс `TControl` имеет метод `Perform`, с помощью которого можно заставить визуальный компонент выполнить обработку конкретного сообщения в обход оконной процедуры и системного механизма передачи сообщений. `Perform` приводит к непосредственному вызову метода, указатель на который хранится в свойстве `WindowProc`. Дальше цепочка обработки сообщений такая же, как и при получении сообщения через оконную процедуру. Для оконных компонентов вызов `Perform` по своим последствиям практически эквивалентен передаче сообщения с помощью `SendMessage` с двумя исключениями. Во-первых, при использовании `SendMessage` система обеспечивает переключение

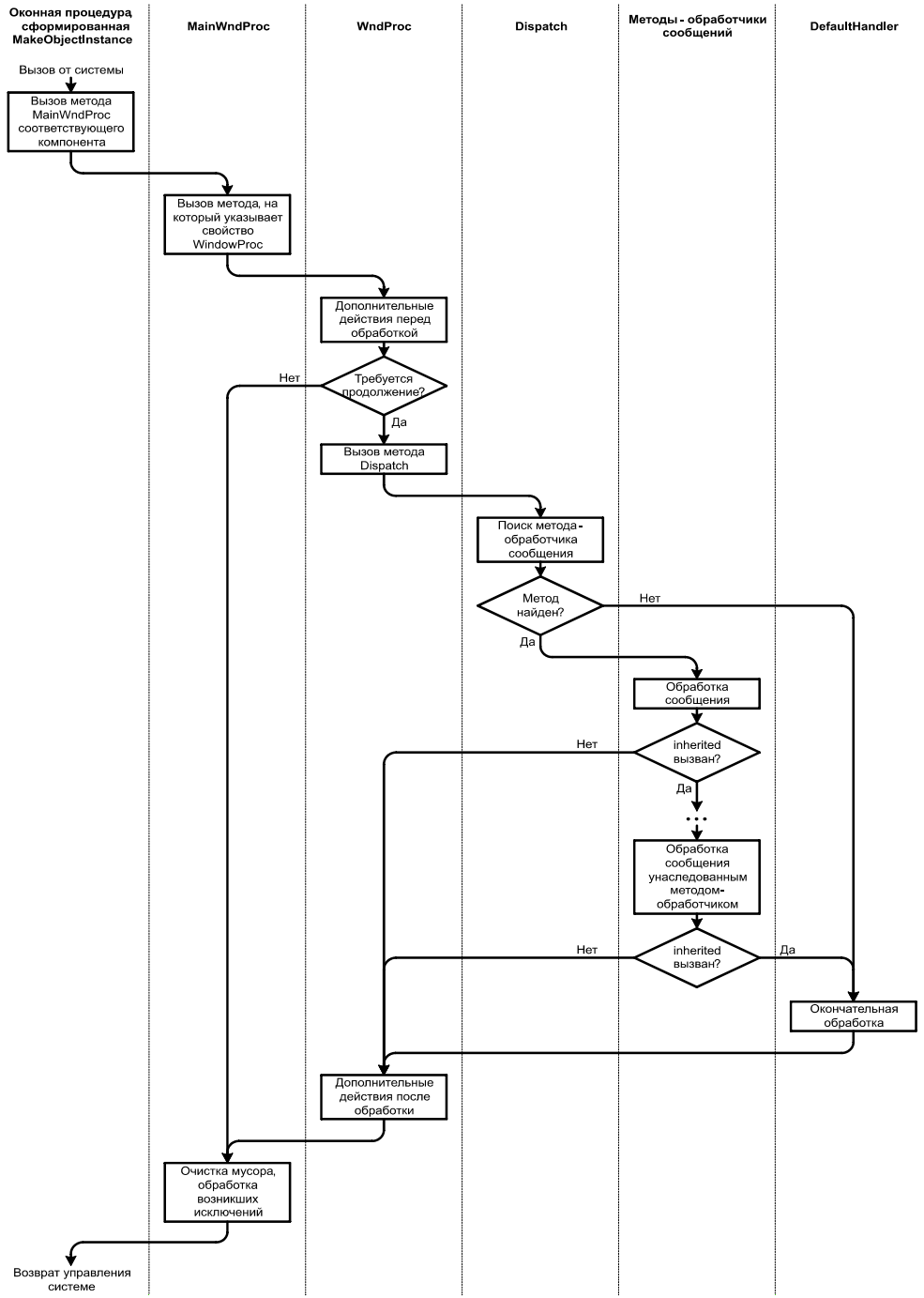


Рис. 1.7. Блок-схема оконной процедуры оконных компонентов VCL

между нитями, и сообщение будет выполнено в той нити, которая создала окно, а `Perform` никакого переключения не производит, и обработка сообщения будет выполнена той нитью, которая вызвала `Perform`. Поэтому `Perform`, в отличие от `SendMessage`, можно использовать только в главной нити (напомним, что VCL — принципиально однонитевая библиотека, и создание форм вне главной нити с ее помощью недопустимо). Во-вторых, `Perform` выполняется чуть быстрее, т. к. оконная процедура и метод `MainWndProc` исключаются из цепочки обработки сообщения.

Но основное преимущество `Perform` перед `SendMessage` заключается в том, что `Perform` пригоден для работы со всеми визуальными компонентами, а не только с оконными. Неоконные визуальные компоненты не могут иметь оконной процедуры, но цепочка обработки сообщений у них есть. В ней отсутствует оконная процедура и метод `MainWndProc`, а `DefaultHandler` не вызывает никаких стандартных оконных процедур, но во всем остальном эта цепочка полностью эквивалентна цепочке оконных компонентов. Таким образом, цепочка обработки сообщений оконных компонентов имеет две точки входа: оконную процедуру и метод `Perform`, а цепочка неоконных компонентов — только метод `Perform`. Следовательно, метод `Perform` универсален: он одинаково хорошо подходит как для оконных, так и для неоконных компонентов. Он широко применяется в VCL, т. к. позволяет единообразно работать с любыми визуальными компонентами.

Неоконным визуальным компонентам сообщения посылает их родительское окно. Например, как мы уже говорили, обработка сообщений, связанных с мышью, в классе `TWinControl` включает в себя проверку, не попадают ли координаты курсора в область какого-либо из дочерних неоконных компонентов. И если попадает, оконный компонент не обрабатывает это сообщение самостоятельно, а транслирует его соответствующему неоконному компоненту с помощью `Perform`. Эта трансляция и обеспечивает получение сообщений неоконными компонентами.

Сообщения в VCL транслируются не только неоконным, но и оконным компонентам. В Windows все сообщения, информирующие об изменении состояния стандартных элементов управления, получает их родительское окно, а не сам элемент. Например, при нажатии на кнопку уведомительное сообщение об этом получает не сама кнопка, а окно, ее содержащее. Сама кнопка получает и обрабатывает только те сообщения, которые обычно разработчику неинтересны. Это упрощает работу программиста, т. к. не требуется для каждого элемента управления писать свою оконную процедуру, все значимые сообщения получает оконная процедура родительского окна.

Рассмотрим, что происходит при нажатии кнопки на форме. Окно, содержащее эту кнопку, получает сообщение `WM_COMMAND`, уведомляющее о возникно-

вении события среди оконных компонентов. Параметры сообщения позволяют определить, какое именно событие и с каким элементом управления произошло (в данном случае событие будет BN_CLICKED). Обработчик WM_COMMAND класса TWinControl находит компонент, вызвавший сообщение, и посылает ему сообщение CN_COMMAND (как видно из префикса, это внутреннее сообщение VCL) с теми же параметрами. В нашем примере это будет экземпляр класса TButton, реализующий кнопку, которую нажал пользователь. Получив CN_COMMAND, компонент начинает обработку произошедшего с ним события (в частности, TButton инициирует событие OnClick).

Примечание

К переопределению обработчика WM_COMMAND нужно относиться осторожно, чтобы не нарушить механизм трансляции сообщений. Примером неправильного переопределения может служить класс TCustomGrid. В форумах нередко встречаются вопросы, почему элементы управления, родителем которых является TDrawGrid или TStringGrid, некорректно ведут себя: кнопки при нажатии не генерируют событие OnClick, выпадающие списки остаются пустыми и т. п. Это связано с тем, что обработчик WM_COMMAND в TCustomGrid учитывает возможность существования только одного дочернего компонента — внутреннего редактора, возникающего при включенной опции goEditing. Остальным дочерним компонентам WM_COMMAND не транслируются, и они лишены возможности корректно реагировать на происходящие с ними события. Выходом из ситуации может стать либо создание наследника от TDrawGrid или TStringGrid, который правильно транслирует WM_COMMAND, либо назначение родительским окном компонента, вставляемого в сетку, формы, панели или иного оконного компонента, который правильно транслирует это сообщение.

Рассмотрим все методы, с помощью которых можно встроить свой код в цепочку обработки сообщений оконным компонентом и перехватить сообщения. Всего существует шесть способов сделать это.

1. Как и у всякого окна, у оконного компонента VCL можно изменить оконную процедуру с помощью функции SetWindowLong. Этот способ лучше не применять, поскольку код VCL не будет ничего "знать" об этом переопределении, и сообщения, получаемые компонентом не через оконную процедуру, а с помощью Perform, не будут перехвачены. Другой недостаток данного способа — то, что изменение некоторых свойств компонента (например, FormStyle и BorderStyle у формы) невозможно без уничтожения окна и создания нового. Для программиста это пересоздание окна выглядит прозрачно, но новое окно получит новую оконную процедуру, и нужно будет выполнять перехват заново. Отследить момент пересоздания окна можно с помощью сообщения CM_RECREATEWND, обработчик которого уничтожает старое окно, а создание нового окна откладывается до момента первого обращения к свойству Handle. Если перехватить это сообщение, то, в принципе, после выполнения стандартного обработчика можно зано-

во установить перехват с помощью `SetWindowLong`, но т. к. этот способ не дает никаких преимуществ перед другими, более простыми, им все равно лучше не пользоваться.

2. Можно создать собственный метод обработки сообщения и поместить указатель на него в свойство `WindowProc`. При этом старый указатель обычно запоминается, т. к. новый обработчик обрабатывает лишь некоторые сообщения, а остальные передает старому. Достоинство этого способа — то, что метод, указатель на который помещается в `WindowProc`, не обязан принадлежать тому компоненту, сообщения которого перехватываются. Это позволяет, во-первых, создавать компоненты, которые влияют на обработку сообщений родительскими формами, а во-вторых, реализовывать нестандартную обработку сообщений стандартными компонентами, не порождая от них наследника.
3. При написании нового компонента можно перекрыть виртуальный метод `WndProc` и реализовать обработку нужных сообщений в нем. Это позволяет компоненту перехватывать сообщения в самом начале цепочки (за исключением внешних обработчиков, установленных с помощью свойства `WindowProc` — здесь разработчик компонента не властен).
4. Наиболее удобный способ самостоятельной обработки событий — написание их методов-обработчиков. Этот способ встречается чаще всего. Его недостатком является то, что номера обрабатываемых сообщений должны быть известны на этапе компиляции. Для системных сообщений и внутренних сообщений VCL это условие выполняется, но далее мы будем говорить об определяемых пользователем сообщениях, номера которых в некоторых случаях на этапе компиляции неизвестны. Обрабатывать такие сообщения с помощью методов с директивой `message` невозможно.
5. Для перехвата сообщений, которые не были обработаны с помощью методов-обработчиков, можно перекрыть виртуальный метод `DefaultHandler`.
6. И наконец, можно написать оконную процедуру и поместить указатель на нее в свойство `DefWndProc`. Этот способ по своим возможностям практически эквивалентен предыдущему, но менее удобен. Однако предыдущий способ пригоден только для создания собственного компонента, в то время как `DefWndProc` можно изменять у экземпляров существующих классов. Напомним, что этот способ не подходит для форм, у которых `FormStyle = fsMDIForm`, т. к. такие формы игнорируют значение свойства `DefWndProc`.

Для перехвата сообщений неоконных визуальных компонентов допустимы все перечисленные способы, за исключением первого и последнего.

Метод `WndProc` оконного компонента транслирует сообщения от мыши неоконным визуальным компонентам, родителем которых он является. Напри-

мер, если положить на форму компонент TImage и переопределить у этой формы метод для обработки сообщения WM_LBUTTONDOWN, то нажатие кнопки мыши над TImage не приведет к вызову этого метода, т. к. WndProc передаст это сообщение в TImage, и Dispatch не будет вызван. Но если переопределить WndProc или изменить значение свойства WindowProc (т. е. использовать второй или третий метод перехвата), то можно получать и обрабатывать и те "мышиные" сообщения, которые должны транслироваться неоконным дочерним компонентам. Это общее правило: чем раньше встраивается собственный код в цепочку обработки сообщений, тем больше у него возможностей.

Как мы уже говорили, начиная с BDS 2006 появился еще один способ перехвата сообщений — перекрытие метода PreProcessMessage. Этот способ нельзя ставить в один ряд с перечисленными ранее шестью способами, т. к. он имеет два существенных отличия от них. Во-первых, с помощью этого способа перехватываются все сообщения, попавшие в петлю сообщений, а не только те, которые посланы конкретному компоненту, из-за чего может понадобиться дополнительная фильтрация сообщений. Во-вторых, метод PreProcessMessage перехватывает сообщения, попавшие в петлю сообщений, а не в оконную процедуру компонента. С одной стороны, это дает возможность перехватывать те сообщения, которые метод Application.ProcessMessage не считает нужным передавать в оконную процедуру, но с другой стороны, не позволяет перехватывать те сообщения, которые окно получает, минуя петлю сообщений (например, те, которые отправлены с помощью SendMessage или Perform). По этим причинам область применения данного способа совсем другая, чем у способов, связанных с внедрением кода в оконную процедуру. Перекрытие PreProcessMessage сопоставимо, скорее, с использованием события Application.OnMessage.

Различные способы перехвата сообщений иллюстрируются рядом примеров на прилагающемся к книге компакт-диске: использование свойства WindowProc показано в примерах Line, CoordLabel и PanelMsg, перекрытие метода WndProc — в примере NumBroadcast, создание метода для обработки сообщения — в примере ButtonDel.

1.1.9. Сообщения, определяемые пользователем

Сообщения очень удобны в тех случаях, когда нужно заставить окно выполнить какое-то действие. Поэтому Windows предоставляет возможность программисту создавать свои сообщения. Существуют три типа *пользовательских сообщений*:

- сообщения оконного класса;
- сообщения приложения;
- глобальные (строковые) сообщения.

Для каждого из них выделен отдельный диапазон номеров. Номера стандартных сообщений лежат в диапазоне от `0` до `WM_USER-1` (`WM_USER` — константа, для 32-разрядных версий Windows равная 1024).

Сообщения оконного класса имеют номера в диапазоне от `WM_USER` до `WM_APP-1` (`WM_APP` имеет значение 32 768). Программист может выбирать произвольные номера для своих сообщений в этом диапазоне. Каждое сообщение должно иметь смысл только для конкретного оконного класса. Для различных оконных классов можно определять сообщения, имеющие одинаковые номера. Система никак не следит за тем, чтобы сообщения, определенные для какого-либо оконного класса, посылались только окнам этого класса — программист должен сам об этом заботиться. В этом же диапазоне лежат сообщения, специфические для стандартных оконных классов `'BUTTON'`, `'EDIT'`, `'LISTBOX'`, `'COMBOBOX'` и т. п.

Использование сообщений из этого диапазона иллюстрируется примером ButtonDel.

Диапазон от `WM_APP` до 49 151 (для этого значения константа не предусмотрена) предназначен для сообщений приложения. Номера этих сообщений также выбираются программистом произвольно. Система гарантирует, что ни один из стандартных оконных классов не задействует сообщения из этого диапазона. Это позволяет выполнять их широковещательную в пределах приложения рассылку. Ни один из стандартных классов не откликнется на такое сообщение и не выполнит нежелательных действий.

Упоминавшиеся ранее внутренние сообщения VCL с префиксами `CM_` и `CN_` имеют номера в диапазоне от 45 056 до 49 151, т. е. используют часть диапазона сообщений приложения. Таким образом, при использовании VCL диапазон сообщений приложения сокращается до `WM_APP..45055`.

Сообщения оконного класса и приложения пригодны и для взаимодействия с другими приложениями, но при этом отправитель должен быть уверен, что адресат правильно его поймет. Широковещательная рассылка при этом исключена — реакция других приложений, которые также получат это сообщение, может быть непредсказуемой. Если все же необходимо рассылать широковещательные сообщения между приложениями, то следует воспользоваться глобальными сообщениями, для которых зарезервирован диапазон номеров от 49 152 до 65 535.

Глобальное сообщение обязано иметь имя (именно поэтому такие сообщения называются также строковыми), под которым оно регистрируется в системе с помощью функции `RegisterWindowMessage`. Эта функция возвращает уникальный номер регистрируемого сообщения. Если сообщение с таким именем регистрируется впервые, номер выбирается из числа еще не занятых. Если же сообщение с таким именем уже было зарегистрировано, то возвращается тот

же самый номер, который был присвоен ему при первой регистрации. Таким образом, разные программы, регистрирующие сообщения с одинаковыми именами, получат одинаковые номера и смогут понимать друг друга. Для прочих же окон это сообщение не будет иметь никакого смысла.

Создание и использование оконных сообщений демонстрируется примером NumBroadcast, содержащимся на прилагаемом компакт-диске.

Разумеется, существует вероятность, что два разных приложения выберут для своих глобальных сообщений одинаковые имена, и это приведет к проблемам при широковещательной рассылке этих сообщений. Но, если давать своим сообщениям осмысленные имена, а не что-то вроде `WM_MYMESSAGE1`, вероятность такого совпадения будет очень мала. В особо критических ситуациях можно в качестве имени сообщения использовать GUID, уникальность которого гарантируется.

Номера глобальных сообщений становятся известными только на этапе выполнения программы. Это означает, что для их обработки нельзя использовать методы с директивой `message`, вместо этого следует перекрывать методы `WndProc` или `DefaultHandler`.

1.1.10. Особые сообщения

Отправка и обработка некоторых сообщений производится не по общим правилам, а с различными исключениями. Приведенный далее список таких сообщений не претендует на полноту, но все-таки может дать представление о таких исключениях.

Сообщение `WM_COPYDATA` служит для передачи блока данных от одного процесса к другому. В 32-разрядных версиях Windows память, выделенная некоторому процессу, недоступна для всех остальных процессов. Поэтому просто передать указатель другому процессу нельзя, поскольку он не сможет получить доступ к этой области памяти. При передаче сообщения `WM_COPYDATA` система копирует указанный блок из адресного пространства отправителя в адресное пространство получателя, передает получателю указатель на этот блок, и при завершении обработки сообщения освобождает блок. Все это требует определенной синхронности действий, которой невозможно достичь при посылке сообщения, поэтому с помощью `WM_COPYDATA` можно только отправлять, но не посылать (т. е. можно использовать `SendMessage`, но не `PostMessage`).

Сообщение `WM_PAINT` предназначено для перерисовки клиентской области окна. Если изображение сложное, этот процесс занимает много времени, поэтому в Windows предусмотрены механизмы, минимизирующие количество перерисовок. Перерисовывать свое содержимое окно должно при получении

сообщения WM_PAINT. С каждым таким сообщением связан регион, нуждающийся в обновлении. Этот регион может совпадать с клиентской областью окна или быть ее частью. В последнем случае программа может ускорить перерисовку, рисуя не все окно, а только нуждающуюся в этом часть (VCL игнорирует возможность перерисовки только части окна, поэтому при работе с этой библиотекой окно всегда перерисовывается полностью). Послать сообщение WM_PAINT с помощью PostMessage окну нельзя, т. к. оно не ставится в очередь. Вместо этого можно пометить регион как нуждающийся в обновлении с помощью функций InvalidateRect и InvalidateRgn. Если на момент вызова этих функций регион, который необходимо обновить, не был пуст, новый регион объединяется со старым. Функции GetMessage и PeekMessage, если очередь сообщений пуста, а регион, требующий обновления, не пуст, возвращают сообщение WM_PAINT. Таким образом, перерисовка окна откладывается до того момента, когда все остальные сообщения будут обработаны. Отправить WM_PAINT с помощью SendMessage тоже нельзя. Если требуется немедленная перерисовка окна, следует вызвать функции UpdateWindow или RedrawWindow, которые не только отправляют сообщение окну, но и выполняют сопутствующие действия, связанные с регионом обновления.

Обработка сообщения WM_PAINT также имеет некоторые особенности. Обработчик должен получить контекст устройства окна *(см. разд. 1.1.11 данной главы)* с помощью функции BeginPaint и по окончании работы освободить его с помощью EndPaint. Эти функции должны вызываться только один раз при обработке сообщения. Соответственно, если сообщение обрабатывается поэтапно несколькими обработчиками, как это бывает при перехвате сообщений, получать и освобождать контекст устройства должен только первый из них, а остальные должны пользоваться тем контекстом, который он получил. Система не накладывает обязательных требований, которые могли бы решить проблему, но предлагает решение, которое используют все предопределенные системные классы. Когда сообщение WM_PAINT извлекается из очереди, его параметр wParam равен нулю. Если же обработчик получает сообщение с wParam <> 0, то он рассматривает значение этого параметра как дескриптор контекста устройства и использует его, вместо того чтобы получать дескриптор через BeginPaint. Первый в цепочке обработчиков должен передать вниз по цепочке сообщение с измененным параметром wParam. Компоненты VCL также пользуются этим решением. При перехвате сообщения WM_PAINT это нужно учитывать.

Примеры PanelMsg и Line, имеющиеся на прилагаемом компакт-диске, демонстрируют, как правильно перехватывать сообщение WM_PAINT.

Простые таймеры, создаваемые системой с помощью функции SetTimer, сообщают об истечении интервала посредством сообщения WM_TIMER. Проверка

того, истек ли интервал, осуществляется внутри функций GetMessage и PeekMessage. Таким образом, если эти функции долго не вызываются, сообщение WM_TIMER не ставится в очередь, даже если положенный срок истек. Если за время обработки других сообщений срок истек несколько раз, в очередь ставится только одно сообщение WM_TIMER. Если в очереди уже есть сообщение WM_TIMER, новое в очередь не добавляется, даже если срок истек. Таким образом, часть сообщений WM_TIMER теряется, т. е., например, если интервал таймера установить равным одной секунде, то за час будет получено не 3600 сообщений WM_TIMER, а меньшее число, и разница будет тем больше, чем интенсивнее программа загружает процессор.

Примечание

Класс TTimer инкапсулирует таймер, работающий через WM_TIMER. Сообщения получает невидимое окно, создающееся специально для этого. Поэтому событие OnTimer за час при секундном интервале также возникнет меньше, чем 3600 раз.

Некоторую специфику имеют и сообщения от клавиатуры. При обработке таких сообщений можно использовать функцию GetKeyState, которая возвращает состояние любой клавиши (нажата-отпущена) в момент возникновения данного события. Именно в момент возникновения, а не в момент вызова функции. Если функцию GetKeyState использовать при обработке не клавиатурного сообщения, оно вернет состояние клавиши на момент последнего извлеченного из очереди клавиатурного сообщения.

1.1.11. Графика в Windows API

Та часть Windows API, которая служит для работы с графикой, обычно называется *GDI* (Graphic Device Interface). Ключевое понятие в GDI — *контекст устройства* (Device Context, DC). Контекст устройства — это специфический объект, хранящий информацию о возможностях устройства, о способе работы с ним и о разрешенной для изменения области. В Delphi контекст устройства представлен классом TCanvas, свойство Handle которого содержит дескриптор контекста устройства. TCanvas универсален в том смысле, что с его помощью рисование в окне, на принтере или в метафайле выглядит одинаково. То же самое справедливо и для контекста устройства. Разница заключается только в том, как получить в разных случаях дескриптор контекста.

Большинство методов класса TCanvas являются "калькой" с соответствующих (в большинстве случаев одноименных) функций GDI. Но в некоторых случаях (прежде всего в методах вывода текста и рисования многоугольников) параметры методов TCanvas имеют более удобный тип, чем функции GDI. Например, метод TCanvas.Polygon требует в качестве параметра открытый мас-

сив элементов типа TPoint, а соответствующая функция GDI — указатель на область памяти, содержащую координаты точек, и число точек. Это означает, что до вызова функции следует выделить память, а потом — освободить ее. Еще нужен код, который заполнит эту область памяти требуемыми значениями. И ни в коем случае нельзя ошибаться в количестве элементов массива. Если зарезервировать память для одного числа точек, а при вызове функции указать другое, программа будет работать неправильно. Но для простых функций работа через GDI ничуть не сложнее, чем через TCanvas.

Для получения дескриптора контекста устройства существует много функций. Только для того, чтобы получить дескриптор контекста обычного окна, существуют четыре функции: BeginPaint, GetDC, GetWindowDC и GetDCEx. Первая из них возвращает контекст клиентской области окна при обработке сообщения WM_PAINT. Вторая дает контекст клиентской области окна, который можно использовать в любой момент времени, а не только при обработке WM_PAINT. Третья позволяет получить контекст всего окна, вместе с неклиентской частью. Последняя же дает возможность получить контекст определенной области клиентской части окна.

После того как дескриптор контекста получен, можно воспользоваться преимуществами класса TCanvas. Для этого необходимо создать экземпляр такого класса и присвоить его свойству Handle полученный дескриптор. Освобождение ресурсов нужно проводить в следующем порядке: сначала свойству Handle присваивается нулевое значение, затем уничтожается экземпляр класса TCanvas, после этого с помощью подходящей функции GDI освобождается контекст устройства. Пример такого использования класса TCanvas демонстрируется листингом 1.17.

Листинг 1.17. Использование класса TCanvas для работы с произвольным контекстом устройства

```
var
  DC: HDC;
  Canvas: TCanvas;
begin
  DC := GetDC (...); // Здесь возможны другие способы получения DC
  Canvas := TCanvas.Create;
  try
    Canvas.Handle := DC;
    // Здесь рисуем, используя Canvas
  finally
    Canvas.Free;
  end;
```

```
  // Освобождение объекта Canvas не означает освобождения контекста DC
  // DC необходимо удалить вручную
  ReleaseDC(DC);
end;
```

Использование класса TCanvas для рисования на контексте устройства, для которого имеется дескриптор, показано в примере PanelMsg на прилагающемся компакт-диске.

Разумеется, можно вызывать функции GDI при работе через TCanvas. Для этого им просто нужно передать в качестве дескриптора контекста значение свойства Canvas.Handle. Коротко перечислим те возможности GDI, которые разработчики VCL почему-то не сочли нужным включать в TCanvas: работа с регионами и траекториями; выравнивание текста по любому углу или по центру; установка собственной координатной системы; получение детальной информации об устройстве; использование геометрических перьев; вывод текста под углом к горизонтали; расширенные возможности вывода текста; ряд возможностей по рисованию нескольких кривых и многоугольников одной функцией; поддержка режимов заливки. Доступ ко всем этим возможностям может быть осуществлен только через API. Отметим также, что Windows NT/2000/XP поддерживает больше графических функций, чем 9*x*/ME. Функции, которые не поддерживаются в 9*x*/ME, также не имеют аналогов среди методов TCanvas, иначе программы, написанные с использованием данного класса, нельзя было бы запустить в этих версиях Windows.

GDI предоставляет очень интересные возможности по преобразованию координат, но только в Windows NT/2000/XP; в Windows 9*x*/ME они не поддерживаются. С помощью функции SetWorldTransform можно задать произвольную матрицу преобразования координат, и все дальнейшие графические операции будут работать в новой координатной системе. Матрица позволяет описать такие преобразования координат, как поворот, сдвиг начала координат и масштабирование, т. е. возможности открываются очень широкие. Также существует менее гибкая, но тоже полезная функция преобразования координат — SetMapMode, которая поддерживается во всех версиях Windows. С ее помощью можно установить такую систему координат, чтобы во всех функциях задавать координаты, например, в миллиметрах, а не пикселах. Это позволяет использовать один и тот же код для вывода на устройства с разными разрешениями.

Некоторые возможности GDI, которым нет аналогов в TCanvas, демонстрируются примером GDIDraw.

Для задания цвета в GDI предусмотрен тип COLORREF (в модуле Windows определен также его синоним для Delphi — TColorRef). Это 4-байтное беззнаковое целое, старший байт которого определяет формат представления цвета. Если

этот байт равен нулю (будем называть этот формат нулевым), первый, второй и третий байты представляют собой интенсивности красного, зеленого и синего цветов соответственно. Если старший байт равен 1, два младших байта хранят индекс цвета в текущей палитре устройства, третий байт не используется и должен быть равен нулю. Если старший байт равен 2, остальные байты, как и в нулевом формате, показывают интенсивность цветовых составляющих.

Тип `TColorRef` позволяет менять глубину каждого цветового канала от 0 до 255, обеспечивая кодирование 16 777 216 различных оттенков (это соответствует режиму TrueColor). Если цветовое разрешение устройства невелико, GDI подбирает ближайший возможный цвет из палитры. Если старший байт `TColorRef` равен нулю, цвет выбирается из текущей системной палитры (по умолчанию эта палитра содержит всего 20 цветов, поэтому результаты получаются далекими от совершенства). Если же старший байт равен 2, то GDI выбирает ближайший цвет из палитры устройства. В этом случае результаты получаются более приемлемыми. Если устройство имеет большую цветовую глубину и не задействует палитру, разницы между нулевым и вторым форматом `COLORREF` нет.

Примечание

Хотя режимы HighColor (32 768 или 65 536 цветов) не обладают достаточной цветовой глубиной, чтобы передать все возможные значения `TColorRef`, палитра в этих режимах не используется, и ближайший цвет выбирается не из палитры, а из всех цветов, которые способно отобразить устройство. Поэтому выбор нулевого формата в этих режимах дает хорошие результаты.

В Windows API определены макросы (а в модуле `Windows`, соответственно, одноименные функции) `RGB`, `PaletteIndex` и `PaletteRGB`. `RGB` принимает три параметра — интенсивности красного, зеленого и синего компонентов и строит из них значение типа `TColorRef` нулевого формата. `PaletteIndex` принимает в качестве параметра номер цвета в палитре и на его основе конструирует значение первого формата. Макрос `PaletteRGB` эквивалентен `RGB`, за исключением того, что устанавливает старший байт возвращаемого значения равным двум. Для извлечения интенсивностей отдельных цветовых компонентов из значения типа `TColorRef` можно воспользоваться функциями `GetRValue`, `GetGValue` и `GetBValue`.

В системе определены два специальных значения цвета: `CLR_NONE` (`$1FFFFFFF`) и `CLR_DEFAULT` (`$20000000`). Они используются только в списках рисунков (image lists) для задания фонового и накладываемого цветов при выводе рисунка. `CLR_NONE` задает отсутствие фонового или накладываемого цвета (в этом случае соответствующий визуальный эффект не применяется), `CLR_DEFAULT` — установка цвета, заданного для всего списка.

В VCL для передачи цвета предусмотрен тип TColor, определенный в модуле Graphics. Это 4-байтное число, множество значений которого является надмножеством значений типа TColorRef. К системным форматам 0, 1 и 2 добавлен формат 255. Если старший байт значения типа TColor равен 255, то младший байт интерпретируется как индекс системного цвета (второй и третий байт при этом не учитываются). Системные цвета — это цвета, используемые системой для рисования различных элементов интерфейса пользователя. Конкретные RGB-значения этих цветов зависят от версии Windows и от текущей цветовой схемы. RGB-значение системного цвета можно получить с помощью функции GetSysColor. 255-й формат TColor освобождает от необходимости явного вызова данной функции.

Для типа TColor определен ряд констант, облегчающих его использование. Некоторые из них соответствуют определенному RGB-цвету (clWhite, clBlack, clRed и т. п.), другие — определенному системному цвету (clWindow, clHighlight, clBtnFace и т. п.). Значения RGB-цветов определены в нулевом формате. Это не приведет к потере точности цветопередачи в режимах с палитрой, т. к. константы определены только для 16-ти основных цветов, которые обязательно присутствуют в системной палитре. Значениям CLR_NONE и CLR_DEFAULT соответствуют константы clNone и clDefault. Они служат (помимо списков рисунков) для задания прозрачного цвета в растровом изображении. Если этот цвет равен clNone, изображение считается непрозрачным, если clDefault, в качестве прозрачного цвета берется цвет левого нижнего пиксела.

Везде, где требуется значение типа TColor, можно подставлять TColorRef, т. е. всем свойствам и параметрам методов класса TCanvas, имеющим тип TColor, можно присваивать те значения TColorRef, которые сформированы функциями API. Обратное неверно: API-функции не умеют обращаться с 255-м форматом TColor. Преобразование из TColor в TColorRef осуществляется с помощью функции ColorToRGB. Значения нулевого, первого и второго формата, а также clNone и clDefault она оставляет без изменения, а значения 255-го формата приводит к нулевому с помощью функции GetSysColor. Эту функцию следует использовать при передаче значений типа TColor в функции GDI.

Применение кистей, перьев и шрифтов в GDI принципиально отличается от того, как это делается в VCL. Класс TCanvas имеет свойства Pen, Brush, и Font, изменение свойств которых приводит к выбору того или иного пера, кисти, шрифта. В GDI эти объекты самостоятельны, должны создаваться, получать свой дескриптор, "выбираться" в нужный контекст устройства с помощью функции SelectObject и уничтожаться после использования. Причем удалять можно только те объекты, которые не выбраны ни в одном контексте. Есть

также несколько стандартных объектов, которые не нужно ни создавать, ни удалять. Их дескрипторы можно получить с помощью функции `GetStockObject`. Для примера рассмотрим фрагмент программы, рисующей на контексте с дескриптором `DC` две линии: синюю и красную (листинг 1.18). В этом фрагменте используется то, что функция `SelectObject` возвращает дескриптор объекта, родственного выбираемому, который был выбран ранее. Так, при выборе нового пера она вернет дескриптор того пера, которое было выбрано до этого.

Листинг 1.18. Рисование разными перьями с использованием GDI

```
SelectObject(DC, CreatePen(PS_SOLID, 1, RGB(255, 0, 0)));
MoveToEx(DC, 100, 100, nil);
LineTo(DC, 200, 200);
DeleteObject(SelectObject(DC,
  CreatePen(PS_SOLID, 1, RGB(0, 0, 255))));
MoveToEx(DC, 200, 100, nil);
LineTo(DC, 100, 200);
DeleteObject(SelectObject(DC, GetStockObject(BLACK_PEN)));
```

Дескрипторы объектов GDI имеют смысл только в пределах того процесса, который их создал, передавать их между процессами нельзя. Тем не менее изредка можно встретить утверждения, что такая передача возможна. Источник этой ошибки в том, что дескрипторы объектов GDI можно было передавать между процессами в старых, 16-разрядных версиях Windows, так что все утверждения о возможности такой передачи просто основываются на устаревших сведениях.

Для хранения растровых изображений в Windows существуют три формата: *DDB*, *DIB* и *DIB-секция*. DDB — это Device Dependent Format, формат, определяемый графическим устройством, на которое идет вывод. DIB — это Device Independent Bitmap, формат, единый для всех устройств. Формат DIB — это устаревший формат, который не позволяет использовать графические функции GDI для модификации картинки, модифицировать изображение можно, только одним способом: вручную изменяя цвета отдельных пикселов. В 32-разрядных версиях появился еще один формат — DIB-секция. По сути дела это тот же самый DIB, но дополненный возможностями рисовать на нем с помощью GDI-функций. Все различия между этими тремя форматами можно прочитать в замечательной книге [1]; мы же здесь ограничимся только кратким их обзором.

Формат DDB поддерживается самой видеокартой (или другим устройством вывода), поэтому при операциях с таким изображением задействуется аппаратный ускоритель графики. DDB-изображение хранится в выгружаемом

системном пуле памяти (Windows NT/2000/XP) или в куче GDI (Windows 9*x*/ME). При этом размер DDB-растра не может превышать 16 Мбайт в Windows 9*x*/ME и 48 Мбайт в Windows NT/2000/XP. Формат DDB не переносим с одного устройства на другое, он должен использоваться только в рамках одного устройства. Прямой доступ к изображению и его модификация вручную невозможны, т. к. формат хранения изображения конкретным устройством непредсказуем. Модифицировать DDB можно только с помощью функций GDI. Цветовая глубина DDB-изображений определяется устройством.

DIB-секция может храниться в любой области памяти, ее размер ограничивается только размером доступной приложению памяти, функции GDI для рисования на таком изображении используют чисто программные алгоритмы, никак не задействуя аппаратный ускоритель. DIB-секция поддерживает различную цветовую глубину и прямой доступ к области памяти, в которой хранится изображение. DIB-секция переносима с одного устройства на другое. BMP-файлы хранят изображение как DIB.

Скорость работы с изображением в формате DIB-секции зависит только от производительности процессора, памяти и качества реализации графических алгоритмов системой (а они, надо сказать, реализованы в Windows очень неплохо). Скорость работы с изображением в формате DDB зависит еще и от драйвера и аппаратного ускорителя видеокарты. Во-первых, аппаратный ускоритель и драйвер могут поддерживать или не поддерживать рисование графических примитивов (в последнем случае эти примитивы рисует система; то, какие операции поддерживает драйвер, можно узнать с помощью функции `GetDeviceCaps`). До недавнего времени была характерной ситуация, когда рисование картинки на DDB-растре и вывод такого растра на экран были заметно (иногда — в два-три раза) быстрее, чем аналогичные операции с DIB-секцией. Однако сейчас разница стала существенно меньше, производительность системы в целом выросла сильнее, чем производительность двумерных аппаратных ускорителей (видимо, разработчики видеокарт больше не считают двумерную графику узким местом и поэтому сосредоточили свои усилия на развитии аппаратных ускорителей 3D-графики). На некоторых мощных компьютерах можно даже столкнуться с ситуацией, когда DDB-изображение отстает по скорости от DIB.

Класс `TBitmap` может хранить изображение как в виде DDB, так и в виде DIB-секции — это определяется значением свойства `PixelFormat`. Значение `pfDevice` означает использование DDB, остальные значения — DIB-секции с различной цветовой глубиной. По умолчанию `TBitmap` создает изображение с форматом `pfDevice`, но программист может изменить формат в любой момент. При этом создается новое изображение требуемого формата, старое копируется в него и уничтожается.

Со свойством PixelFormat тесно связано свойство HandleType, которое может принимать значения bmDIB и bmDDB. Изменение свойства PixelFormat приводит к изменению свойства HandleType, и наоборот.

Примечание

Если вы собираетесь распечатывать изображение, содержащееся в TBitmap, то вы должны установкой свойств PixelFormat или HandleType обеспечить, чтобы изображение хранилось в формате DIB. Попытка вывести DDB-изображение на принтер приводит к непредсказуемым результатам (чаще всего просто ничего не выводится) из-за того, что драйвер принтера не понимает формат, совместимый с видеокартой.

При загрузке изображения из файла, ресурса или потока класс TBitmap обычно создает изображение в формате DIB-секции, соответствующее источнику по цветовой глубине. Исключение составляют сжатые файлы (формат BMP поддерживает сжатие только для 16- и 256-цветных изображений) — в этом случае создается DDB. В файле Graphics определена глобальная переменная DDBsOnly, которая по умолчанию равна False. Если изменить ее значение на True, загружаемое изображение всегда будет иметь формат DDB.

Примечание

В справке сказано, что когда DDBsOnly = False, вновь создаваемые изображения по умолчанию хранятся в виде DIB-секции. На самом деле из-за ошибки в модуле Graphics (как минимум до 2007-й версии Delphi включительно) вновь созданное изображение всегда хранится как DDB, независимо от значения DDBsOnly.

Класс TBitmap имеет свойство ScanLine, через которое можно получить прямой доступ к массиву пикселов, составляющих изображение. В справке написано, что это свойство можно использовать только с DIB-изображениями. Но на самом деле DDB-изображения тоже позволяют использовать это свойство, хотя и с существенными ограничениями. Если изображение хранится в DDB-формате, при обращении к ScanLine создается его DIB-копия, и ScanLine возвращает указатель на массив этой копии. Поэтому, во-первых, ScanLine работает с DDB-изображениями очень медленно, а во-вторых, работает не с изображением, а с его копией, откуда вытекают следующие ограничения:

1. Копия создается на момент обращения к ScanLine, поэтому изменения, сделанные на изображении с помощью GDI-функций после этого, будут недоступными.
2. Каждое обращение к ScanLine создает новую копию изображения, а старая при этом уничтожается. Гарантии, что новая копия будет располагаться в той же области памяти, нет, поэтому указатель, полученный при предыдущем обращении к ScanLine, больше нельзя использовать.

3. Изменения, сделанные в массиве пикселов, затрагивают только копию изображения, но само изображение при этом не меняется. Поэтому в случае DDB свойство `ScanLine` дает возможность прочитать, но не изменить изображение.

Следует отметить, что `TBitmap` иногда создает DIB-секции, даже если свойства `HandleType` и `PixelFormat` явно указывают на использование DDB. Особенно часто это наблюдается для изображений большого размера. По всей видимости, это происходит, когда в системном пуле нет места для хранения DDB-изображения такого размера, и разработчики `TBitmap` решили, что в таком случае лучше создать DIB-изображение, чем не создавать никакого.

Пример BitmapSpeed на прилагаемом компакт-диске позволяет сравнить скорость выполнения различных операций с DDB- и DIB-изображениями.

1.1.12. ANSI и Unicode

Windows поддерживает две кодировки: ANSI и Unicode. В кодировке ANSI (American National Standard Institute) каждый символ кодируется однобайтным кодом. Коды от 0 до 127 совпадают с кодами ASCII, коды от 128 до 255 могут означать разные символы различных языков в зависимости от выбранной кодовой страницы. Кодовые страницы позволяют уместить многочисленные символы различных языков в однобайтный код, но при этом можно работать только с одной кодовой страницей, т. е. с одним языком. Неверно выбранная кодовая страница приводит к появлению непонятных символов (в Интернете их обычно называют "кракозябрами") вместо осмысленного текста.

В кодировке Unicode используется 2 байта на символ, что дает возможность закодировать 65 536 символов. Этого хватает для символов латиницы и кириллицы, греческого алфавита, китайских иероглифов, арабских и еврейских букв, а также многочисленных дополнительных (финансовых, математических и т. п.) символов. Кодовых страниц в Unicode нет.

Примечание

Кодовая страница русского языка в ANSI имеет номер 1251. Кодировка символов в ней отличается от принятой в DOS так называемой альтернативной кодировки. В целях совместимости для DOS-программ, а также для консольных приложений Windows использует альтернативную кодировку. Именно поэтому при выводе русского текста в консольных приложениях получаются те самые "кракозябры". Чтобы избежать этого, следует перекодировать символы из кодировки ANSI в DOS при выводе, и наоборот — при вводе. Это можно сделать с помощью функций `CharToOem` и `OemToChar`.

Windows NT/2000/XP поддерживает ANSI и Unicode в полном объеме. Это значит, что любая функция, работающая со строками, представлена в двух вариантах: для ANSI и для Unicode. Windows 9*x*/ME в полном объеме под-

держивает только ANSI. Unicode-варианты в этих системах есть у относительно небольшого числа функций. Каждая страница MSDN, посвященная функции, работающей со строками (или со структурами, содержащими строки), в нижней части содержит надпись, показывающую, реализован ли Unicode-вариант этой функции только для NT/2000/XP или для всех платформ.

Примечание

В качестве примера рассмотрим функции вывода текста на экран. Unicode-версию на всех платформах имеют только две из них: `TextOut` и `ExtTextOut`. Функции `DrawText` и `DrawTextEx` имеют Unicode-варианты только в Windows NT/2000/XP. Если же смотреть функции для работы с окнами, то среди них нет ни одной, которая имела бы Unicode-вариант в Windows 9*x*/ME.

Следует отметить, что с относительно недавнего времени Microsoft предоставляет расширение для Windows 9*x*/ME, которое позволяет добавить полную поддержку Unicode в эти системы. Это расширение называется MSLU (Microsoft Layer for Unicode), его можно скачать с официального сайта Microsoft.

Рассмотрим, как сосуществуют два варианта на примере функции `RegisterWindowMessage`. Согласно справке, она экспортируется библиотекой user32.dll. Однако если посмотреть список функций, экспортируемых этой библиотекой (это можно сделать, например, при помощи утилиты TDump.exe, входящей в состав Delphi), то там этой функции не будет, зато будут функции `RegisterWindowMessageA` и `RegisterWindowMessageW`. Первая из них — это ANSI-вариант функции, вторая — Unicode-вариант (буква W означает Wide — широкий; символы кодировки Unicode часто называются широкими из-за того, что на один символ приходится не один, а два байта).

Сначала разберемся с тем, как используются два варианта одной функции в Microsoft Visual C++. В стандартных заголовочных файлах учитывается наличие макроопределения `UNICODE`. Есть два символьных типа — `CHAR` для ANSI и `WCHAR` для Unicode. Если макрос `UNICODE` определен, тип `TCHAR` соответствует типу `WCHAR`, если не определен — типу `CHAR` (после этого производные от `TCHAR` типы, такие как `LPCTSTR`, автоматически начинают соответствовать кодировке, определяемой наличием или отсутствием определения `UNICODE`). В заголовочных файлах импортируются оба варианта функции, а также определяется макрос `RegisterWindowMessage`. Его смысл также зависит от макроса `UNICODE`: если он определен, `RegisterWindowMessage` эквивалентен `RegisterWindowMessageW`, если не определен — `RegisterWindowMessageA`. Все функции, поддерживающие два варианта кодировки, импортируются точно так же. Таким образом, вставляя или убирая макроопределение `UNICODE`, можно, не меняя ни единого символа в программе, компилировать ее ANSI- или Unicode-версию.

Разработчики Delphi не стали полностью копировать этот механизм, видимо, этому помешала существующая в Delphi раздельная компиляция модулей, из-за которой невозможно определением одного символа заставить все модули перекомпилироваться (тем более что часть из них может не иметь исходных кодов). Поэтому в Delphi нет типа, аналогичного TCHAR.

Рассмотрим, как та же функция RegisterWindowMessage импортируется модулем Windows (листинг 1.19).

Листинг 1.19. Импорт функции RegisterWindowMessage

```
interface
...
function RegisterWindowMessage(lpString: PChar): UINT; stdcall;
function RegisterWindowMessageA(lpString: PAnsiChar): UINT; stdcall;
function RegisterWindowMessageW(lpString: PWideChar): UINT; stdcall;
...
implementation
...
function RegisterWindowMessage;
  external user32 name 'RegisterWindowMessageA';
function RegisterWindowMessageA;
  external user32 name 'RegisterWindowMessageA';
function RegisterWindowMessageW;
  external user32 name 'RegisterWindowMessageW';
```

Видно, что функция RegisterWindowMessageA импортируется дважды: один раз под своим настоящим именем, а второй раз — под именем RegisterWindowMessage. Любое из этих имен подходит для вызова ANSI-варианта этой функции (напоминаю, что типы PChar и PAnsiChar эквивалентны). Чтобы вызвать Unicode-вариант функции, потребуется функция RegisterWindowMessageW.

Структуры, содержащие строковые данные, также имеют ANSI- и Unicode-вариант. Например, структура WNDCLASS в модуле Windows представлена типами TWndClassA (с синонимами WNDCLASSA и tagWNDCLASSA) и TWndClassW (с синонимами WNDCLASSW и tagWNDCLASSW). Тип TWndClass (и его синонимы WNDCLASS и tagWNDCLASS) эквивалентен типу TWndClassA. Соответственно, при вызове функций RegisterClassA и RegisterClassExA используется тип TWndClassA, при вызове RegisterClassW и RegisterClassExW — тип TWndClassW.

1.1.13. Строки в Windows API

Unicode в Delphi встречается редко, т. к. программы, использующие эту кодировку, не работают в Windows 9*x*/ME. Библиотека VCL также игнорирует

возможность работы с Unicode, ограничиваясь ANSI. Поэтому далее мы будем говорить только об ANSI. С кодировкой Unicode можно работать аналогично, заменив `PChar` на `PWideChar` и `string` на `WideString`.

Для работы со строками в Delphi наиболее распространен тип `AnsiString`, обычно называемый просто `string` (более детально отношения между этими типами рассмотрены в *главе 3*). Переменная типа `string` является указателем на строку, хранящуюся в динамической памяти. Этот указатель указывает на первый символ строки. По отрицательному смещению хранится число символов в строке и счетчик ссылок, который позволяет избежать ненужных копирований строки, реализуя так называемое *"копирование при необходимости"*. Если присвоить одной строковой переменной значение другой строковой переменной, то строка не копируется, а просто обе переменные начинают указывать на одну и ту же строку. Счетчик ссылок при этом увеличивается на единицу. Когда строка модифицируется, проверяется счетчик ссылок: если он не равен единице, то строка копируется, счетчик ссылок старой копии уменьшается на единицу, у новой копии счетчик ссылок будет равен единице, и переменная, которая меняется, будет указывать на новую копию. Таким образом, строка копируется только тогда, когда одна из ссылающихся на нее переменных начинает изменять эту строку, чтобы изменения не коснулись остальных переменных. При любых модификациях строки в ее конец автоматически добавляется нулевой символ (при подсчете длины строки с помощью функции `Length` он игнорируется). Но если присвоить строковой переменной пустую строку, то эта переменная станет нулевым указателем (`nil`), память для хранения одного символа `#0` выделена не будет. При выходе строковой переменной из области видимости (т. е., например, при завершении процедуры, в которой она является локальной переменной, или при уничтожении объекта, полем которого она является) она автоматически финализируется, т. е. счетчик ссылок уменьшается на единицу и, если он оказывается равен нулю, память, выделенная для строки, освобождается. (О внутреннем устройстве `AnsiString` см. также *разд. 3.3.*)

Механизм выделения и освобождения памяти и подсчета ссылок прозрачен для программы. От разработчика требуется только не вмешиваться в его работу с помощью низкоуровневых операций с указателями, чтобы менеджер памяти не запутался.

Примечание

В отличие от `string`, тип `WideString` не имеет счетчика ссылок, и любое присваивание переменных этого типа приводит к копированию строки. Это сделано в целях совместимости с системным типом BSTR, использующимся в COM/DCOM и OLE.

Функции Windows API не поддерживают тип `string`. Они работают со строками, оканчивающимися на `#0` (*нуль-терминированные строки*, null-terminated strings). Это означает, что строкой называется указатель на цепочку символов. Признаком конца такой цепочки является символ с кодом 0. Раньше для таких строк существовал термин *ASCIIZ*. ASCII — название кодировки, Z — zero. Сейчас кодировка ASCII в чистом виде не встречается, поэтому этот термин больше не применяется, хотя это те же самые по своей сути строки. Как уже говорилось, в Delphi ко всем строкам типа `string` неявно добавляется нулевой символ, не учитывающийся при подсчете числа символов. Это сделано для совместимости с нуль-терминированными строками. Однако эта совместимость ограничена.

Для работы с нуль-терминированными строками в Delphi обычно служит тип `PChar`. Формально это указатель на один символ типа `Char`, но подразумевается, что это только первый символ строки, и за ним следуют остальные символы. Где будут эти символы размещены и какими средствами для них будет выделена память, программист должен решить самостоятельно. Он же должен позаботиться о том, чтобы в конце цепочки символов стоял `#0`.

Строку, на которую указывает `PChar`, можно использовать везде, где требуется `string` — компилятор сам выполнит необходимые преобразования. Обратное неверно. Фактически, `string` — это указатель на начало строки, завершающейся нулем, т. е. тот самый указатель, который требуется при работе с `PChar`. Однако, как уже отмечалось, некорректные манипуляции с этим указателем могут привести к нежелательным эффектам, поэтому компилятор требует явного приведения переменных и выражений типа `string` к `PChar`. В свою очередь, программист должен ясно представлять, к каким последствиям это может привести.

Если посмотреть описание функций API, имеющих строковые параметры, в справке, можно заметить, что в некоторых случаях строковые параметры имеют тип `LPCTSTR` (как, например, у функции `SetWindowText`), а в некоторых — `LPTSTR` (`GetWindowText`). Ранее мы говорили, что появление префикса *`C`* после *`LP`* указывает на то, что это указатель на константу, т. е. то, на что указывает такой указатель, не может быть изменено. Тип `LPCTSTR` имеют те строковые параметры, содержимое которых функция только читает, но не модифицирует. С такими параметрами работать проще всего. Рассмотрим на примере функции `SetWindowText`, как можно работать с такими параметрами (листинг 1.20).

Листинг 1.20. Вызов функции с параметром типа LPCTSTR

```
{ Вариант 1 }
SetWindowText(Handle, 'Строка');
```

```
{ Вариант 2
  S — переменная типа string }
SetWindowText(PChar(S));

{ Вариант 3
  X — переменная типа Integer }
SetWindowText(PChar('Выполнено ' + IntToStr(X) + '%'));
```

В первом варианте компилятор размещает строковый литерал в сегменте кода, а в функцию передает указатель на эту область памяти. Поскольку функция не модифицирует строку, а только читает, передача такого указателя не вызывает проблем.

Во втором варианте функции передается указатель, хранящийся в переменной S. Такое приведение string к PChar безопасно, т. к. строка, на которую ссылается переменная S, не будет модифицироваться. Но здесь существует одна тонкость: конструкция PChar(S) — это не просто приведение типов, при ее использовании неявно вызывается функция _LStrToPChar. Как мы уже говорили, когда string хранит пустую строку, указатель просто имеет значение nil. Функция _LStrToPChar проверяет, пустая ли строка хранится в переменной, и, если не пустая, возвращает этот указатель, а если пустая, то возвращает не nil, а указатель на символ #0, который специально для этого размещен в сегменте кода. Поэтому даже если S содержит пустую строку, в функцию будет передан ненулевой указатель.

Вычисление строковых выражений требует перераспределения памяти, а это компилятор делает только с выражениями типа string. Поэтому результат выражения, приведенного в третьем варианте, также имеет тип string. Но его можно привести к PChar. Память для хранения результата выражения выделяется динамически, как и для обычных переменных типа string. Чтобы передать указатель на это выражение в функцию, следует привести его к PChar. В эпилог процедуры, вызывающей функцию SetWindowText или иную функцию с подобным аргументом, добавляется код, который освобождает динамически сформированную строку, поэтому утечек памяти не происходит.

Разумеется, существуют и другие способы формирования параметра типа LPCTSTR, кроме предложенных здесь. Можно, например, выделить память для нуль-терминированной строки с помощью StrNew или родственной ей функции из модуля SysUtils. Можно использовать массив типа Char. Можно выделять память какими-либо другими способами. Но предложенные здесь три варианта в большинстве случаев наиболее удобны.

Параметры типа LPTSTR применяются в тех случаях, когда функция может не только читать, но и модифицировать передаваемое ей значение. В большинстве случаев такие параметры чисто выходные, т. е. функция не интересуется, какое значение имел параметр при вызове, используя его только для возврата

значения. При возврате строкового значения всегда возникает проблема: где, кем и как будет выделена память, в которую будет записана строка? Функции Windows API, за очень редким исключением, решают эту проблему следующим образом: память должна выделить вызывающая программа, а в функцию передается указатель на этот заранее выделенный блок. Сама функция только копирует строку в этот блок.

Таким образом, перед программой встает задача узнать, какой объем памяти следует выделить под возвращаемую строку. Здесь API не предлагает универсального решения, разные функции по-разному решают эту проблему. Например, при получении заголовка окна с помощью `GetWindowText` размер этого заголовка можно узнать, вызвав предварительно `GetWindowTextLength`. Функции типа `GetCurrentDirectory` возвращают длину строки. Если при первом вызове этой функции памяти выделено недостаточно, можно увеличить буфер и вызвать функцию еще раз. И наконец, есть функции типа `SHGetSpecialFolderPath`, в описании которых написано, каков минимальный размер буфера, необходимый для гарантированной передачи полной строки этой функцией (это, разумеется, возможно только в том случае, когда размер возвращаемой строки имеет какое-то естественное ограничение). Следует также отметить, что большинство API-функций, возвращающих строки, в качестве одного из параметров принимают размер буфера, чтобы не скопировать больше байтов, чем буфер может принять.

Выделять буфер для получения строки можно многими способами. На практике удобнее всего статические массивы, тип `string` или динамическое выделение памяти для нуль-терминированных строк.

Статические массивы могут использоваться, если размер буфера известен на этапе компиляции. Массивы типа `Char` с начальным индексом 0 рассматриваются компилятором как нуль-терминированные строки, поэтому с ними удобно выполнять дальнейшие операции. Этот способ удобен тем, что не нужно заботиться о выделении и освобождении памяти, поэтому он часто применяется там, где формально длина строки на этапе неизвестна, но "исходя из здравого смысла" можно сделать вывод, что в подавляющем большинстве случаев эта длина не превысит некоторой величины, которая и берется в качестве размера массива.

Строки типа `string` также могут служить буфером для получения строковых значений от системы. Для этого нужно предварительно установить требуемую длину строки с помощью `SetLength`, а затем передать указатель на начало строки в функцию API. Здесь следует соблюдать осторожность: если длина строки окажется равной нулю, переменная типа `string` будет иметь значение `nil`, а система попытается записать по этому указателю пустую строку, состоящую из единственного символа `#0`. Это приведет к ошибке Access violation.

Третий способ — выделение памяти для буфера с помощью `StrAlloc` или аналогичной ей функции. Память, выделенную таким образом, следует обязательно освобождать с помощью `StrDispose`. При этом крайне желательно использовать конструкцию `try/finally`, чтобы возникновение исключений не привело к утечкам памяти.

Все три способа получения строковых данных от функций Windows API показаны в примере EnumWnd, находящемся на прилагаемом компакт-диске.

1.2. Примеры использования Windows API

В этом разделе разобраны простые примеры, находящиеся на компакт-диске. Все эти примеры уже упоминались ранее, и каждый из них иллюстрирует какую-то отдельно взятую возможность API. Более сложным обобщающим примерам, которые задействуют сразу несколько возможностей API и VCL, посвящен следующий, третий раздел данной главы.

1.2.1. Пример EnumWnd

Программа EnumWnd представляет собой простой пример использования функций `EnumWindows` и `EnumChildWindows`, а также функций обратного вызова, которые необходимы для работы этих двух функций. Программа ищет все окна, созданные на данный момент в системе, и отображает их в виде дерева: каждый узел дерева соответствует одному окну, дочерние узлы соответствуют дочерним окнам данного окна (рис. 1.8).

Программа EnumWnd является также примером того, как можно работать с параметрами типа `LPTSTR`, через которые функции Windows API возвращают программе строковые значения. В *разд. 1.1.13* были перечислены три способа создания буфера для работы с такими параметрами: выделение памяти в стеке в виде массива элементов типа `Char`, использование строк типа `string` и строк типа `PChar`. Все три способа реализованы в примере EnumWnd.

На главной и единственной форме программы EnumWnd размещены два компонента: `TreeWindow` типа `TTreeView` и кнопка `BtnBuild`. Обработчик нажатия кнопки выглядит очень лаконично (листинг 1.21).

Листинг 1.21. Обработчик нажатия кнопки `BtnBuild`

```
procedure TFormWindows.BtnBuildClick(Sender: TObject);
begin
  Screen.Cursor := crHourGlass;
```

```
  try
    TreeWindows.Items.Clear;
    EnumWindows(@EnumWindowsProc, 0);
  finally
    Screen.Cursor := crDefault;
  end;
end;
```

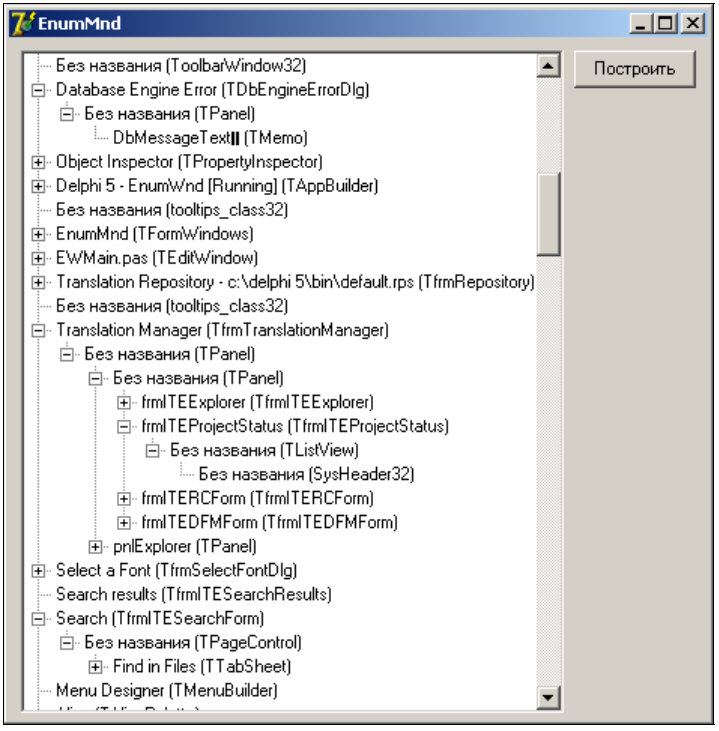


Рис. 1.8. Окно программы EnumWnd

Все, что делает этот обработчик, — это очищает компонент `TreeWindows` и вызывает `EnumWindows`, передавая ей функцию обратного вызова `EnumWindowsProc`, в которой и выполняется основная работа. Сразу отметим, что в этом примере мы будем использовать одну и ту же функцию обратного вызова как для `EnumWindows`, так и для `EnumWindowsProc`. Сама функция обратного вызова выглядит следующим образом (листинг 1.22).

Листинг 1.22. Функция обратного вызова `EnumWindowsProc` (первый вариант)

```
// Это функция обратного вызова, которая будет
// использоваться при вызове EnumWindows и EnumChildWindows.
```

```
// Тип второго параметра не совпадает с типом, который
// указан MSDN. Однако TTreeNode, как и любой класс,
// является указателем, поэтому может использоваться везде,
// где требуется нетипизированный указатель — на двоичном
// уровне между ними нет разницы. Указатель на функцию
// обратного вызова в EnumWindows и EnumChildWindows в
// модуле Windows.dcu объявлен как нетипизированный
// указатель, поэтому компилятор не контролирует
// соответствие реального прототипа заявленному.
function EnumWindowsProc(Wnd: HWND; ParentNode: TTreeNode): Bool;
  stdcall;
 // Система не предусматривает возможности узнать, какова
 // длина имени класса, поэтому при получении этого имени
 // приходится выделять буфер большой длины в надежде, что
 // имя класса не окажется еще длиннее. В данном примере
 // размер этого буфера определяется константой ClassNameLen.
 // Крайне маловероятно, что имя класса окажется длиннее,
 // чем 511 символов (512-й зарезервирован для завершающего
 // нулевого символа).
const
  ClassNameLen = 512;

var
  // Здесь будет храниться заголовок окна
  Text: string;
  TextLen: Integer;
  // Это — буфер для имени класса
  ClassName: array[0..ClassNameLen — 1] of Char;
  Node: TTreeNode;
  NodeName: string;

begin
  Result := True;
  // Функция EnumChildWindows перечисляет не только
  // непосредственно дочерние окна данного окна, но и
  // дочерние окна его дочерних окон и т. п. Но при
  // построении дерева на каждом шаге нам нужны только
  // прямые потомки, поэтому все окна, не являющиеся прямыми
  // потомками, мы здесь игнорируем.
  if Assigned(ParentNode) and
    (GetParent(Wnd) <> HWND(ParentNode.Data)) then
    Exit;
```

```
// Получаем длину заголовка окна. Вместо функций
// GetWindowText и GetWindowTextLength мы здесь
// используем сообщения WM_GETTEXT и WM_GETTEXTLENGTH,
// потому что функции, в отличие от сообщений, не
// умеют работать с элементами управления,
// принадлежащими окнам чужих процессов.
TextLen := SendMessage(Wnd, WM_GETTEXTLENGTH, 0, 0);
// Устанавливаем длину строковой переменной, которая
// будет служить буфером для заголовка окна.
// Использование SetLength гарантирует, что будет
// выделена специальная область памяти, на которую не
// будет других ссылок.
SetLength(Text, TextLen);
// Если заголовок окна — пустая строка, TextLen будет
// иметь значение 0, и указатель Text при выполнении
// SetLength получит значение nil. Но при обработке
// сообщения WM_GETTEXT оконная процедура в любом случае
// попытается записать строку по переданному адресу,
// даже если заголовок окна пустой — в этом случае в
// переданный буфер будет записан один символ -
// завершающий ноль. Но если будет передан nil, то
// попытка записать что-то в такой буфер приведет к
// Access violation, поэтому отправлять окну WM_GETTEXT
// можно только в том случае, если TextLen > 0.
if TextLen > 0 then
  SendMessage(Wnd, WM_GETTEXT, TextLen + 1, LParam(Text));
// Заголовок окна может быть очень длинным — например, в
// Memo заголовком считается весь текст, который там
// есть. Практика показывает, что существуют проблемы
// при добавлении в TTreeView узлов с очень длинными
// названиями: при попытке открыть такой узел программа,
// запущенная из Delphi, вылетает в отладчик (при
// запуске вне среды Delphi проблем не замечено). Чтобы
// этого не происходило, слишком длинные строки
// обрезаются.
if TextLen > 100 then
  Text := Copy(Text, 1, 100) + ' ...';
GetClassName(Wnd, ClassName, ClassNameLen);
ClassName[ClassNameLen — 1] := #0;
if Text = '' then
  NodeName := 'Без названия (' + ClassName + ')'
else
  NodeName := Text + ' (' + ClassName + ')';
Node := FormWindows.TreeWindows.Items.AddChild(ParentNode, NodeName);
```

```
  // Записываем в данные узла дескриптор соответствующего
  // ему окна, чтобы иметь возможность отбросить непрямые
  // потомки.
  Node.Data := Pointer(Wnd);
  // Вызываем EnumChildWindows, передавая функцию
  // EnumWindowsProc в качестве параметра, а указатель на
  // созданный узел — в качестве параметра этой функции.
  // При этом EnumWindowsProc будет вызываться из
  // EnumChildWindows, т. е. получается рекурсия.
  EnumChildWindows(Wnd, @EnumWindowsProc, LParam(Node));
end;
```

Как мы помним, первый параметр функции обратного вызова для EnumWindows содержит дескриптор найденного окна, а второй параметр может быть произвольным 4-байтным значением, которое система игнорирует, просто копируя сюда то значение, которое было передано при вызове EnumWindows или EnumChildWindows. Мы задействуем этот параметр для передачи ссылки на узел дерева, соответствующий родительскому окну. Также договоримся, что в свойство Data каждого узла будем записывать дескриптор связанного с ним окна. Для окон верхнего уровня ссылка будет иметь значение nil — это обеспечивается тем, что при вызове EnumWindows второй параметр равен нулю (см. листинг 1.21).

Работа функции начинается с проверки того, что родительским окном для данного окна действительно является то окно, чей дескриптор связан с узлом родительского окна. Эта проверка нужна потому, что функция EnumChildWindows перечисляет не только дочерние, но и "внучатые", "правнучатые" и т. д. окна. Нам здесь это не нужно, на каждом шаге нас интересуют только непосредственные "дети" окна, а до "внуков" мы доберемся, когда вызовем EnumChildWindows для дочерних окон, поэтому и отсеиваем лишнее.

Следующий шаг — получение заголовка окна. Для этого мы используем сообщение WM_GETTEXT (разница между этим сообщением и функцией GetWindiowText обсуждается в *разд. 1.3.1*). Буфером является переменная Text типа string. Сначала с помощью сообщения WM_GETTEXTLENGTH мы узнаем длину заголовка окна, а затем выделяем под строку Text требуемое количество памяти с помощью SetLength. После этого можно получить строку с помощью сообщения WM_GETTEXT. Второй параметр этого сообщения — адрес буфера, в который будет помещена строка. Так как переменная типа string и есть указатель на буфер строки (это детально обсуждается в *разд. 3.3*), достаточно просто привести переменную Text к типу LParam и передать получившееся значение.

Примечание

Строго говоря, у нас здесь нигде нет параметра типа LPTSTR, однако при работе с параметрами этого типа можно действовать точно так же: выделить для строки типа string нужное количество памяти и передать эту переменную, приведенную к типу LPTSTR, в качестве параметра.

Далее получаем название класса окна. Для этого мы используем статический массив ClassName, т. е. размер буфера определяется на этапе компиляции. С одной стороны, это неправильно, потому что ограничений на длину имени класса не существует (по крайней мере, в документации они не упомянуты), а мы уже говорили, что такой метод следует применять только тогда, когда существует известное на этапе компиляции ограничение длины. Но с другой стороны, когда речь идет об имени класса, не существует ничего подобного сообщению WM_GETTEXTLENGTH, т. е. API не дает возможности получить длину имени класса, что делает бессмысленными все манипуляции с размером буфера во время работы программы. Поэтому мы определяем размер буфера еще на этапе компиляции, исходя из того, что слишком уж длинные имена классов встречаются редко. При вызове функции с параметром типа LPTSTR можно просто передавать массив без приведения типа, т. к. LPTSTR — это PChar, а массивы символов Char, индексирующиеся с нуля, компилятор полагает совместимыми с этим типом и все необходимые преобразования делает неявно.

И, хотя мы и взяли размер буфера с хорошим запасом, нельзя исключать ситуации, когда имя класса окажется длиннее, чем буфер. Ничего страшного при этом не произойдет, т. к. мы передаем в функцию размер буфера специально для того, чтобы она не пыталась что-то записать за пределами буфера. Но в этом случае завершающий строку символ #0 не попадет в буфер, и при попытке дальше работать с этой строкой какая-нибудь другая функция может, не найдя конца строки в пределах буфера, попытаться поискать этот конец за его пределами, что приведет к непредсказуемым результатам. Поэтому на всякий случай записываем #0 в последний символ буфера. Если имя класса оказалось длиннее буфера, это обрежет строку по границе буфера, а если короче, то это ничему не повредит, т. к. признак конца строки будет в буфере где-то раньше, а все символы после него все равно игнорируются.

После этого остается только создать новый элемент в дереве, а чтобы заполнить его дочерние элементы — вызвать EnumChildWindows для получения списка дочерних окон. Так как в EnumChildWindows передается та же функция обратного вызова, получается рекурсия, которая останавливается тогда, когда функция доходит до окна, не имеющего дочерних окон.

Ранее мы говорили, что программа EnumWnd демонстрирует три метода получения строки через параметр типа LPTSTR, но пока мы увидели только два

(действительно, трудно показать три различных метода на примере получения двух строк). Чтобы показать третий вариант — организацию буфера через строки типа PChar — перепишем функцию EnumWindowsProc (листинг 1.23). В исходном коде программы EnumWnd этот вариант присутствует в виде комментария. Можно убрать этот комментарий, а закомментировать, наоборот, первый вариант, чтобы попробовать, как работает получение строки с помощью PChar.

Листинг 1.23. Функция обратного вызова EnumWindowsProc (второй вариант)

```
// Ниже приведен другой вариант функции
// EnumWindowsProc, который отличается от предыдущего тем,
// что буфер для получения заголовка окна организуется
// вручную с помощью переменной типа PChar, а не string. По
// своим функциональным возможностям оба варианта равноценны.
function EnumWindowsProc(Wnd: HWND; ParentNode: TTreeNode): Bool;
  stdcall;
const
  ClassNameLen = 512;
var
  TextLen: Integer;
  Text: PChar;
  ClassName: array[0..ClassNameLen – 1] of Char;
  Node: TTreeNode;
  NodeName: string;
begin
  Result := True;
  if Assigned(ParentNode) and
    (GetParent(Wnd) <> HWND(ParentNode.Data)) then
    Exit;
  // Здесь, в отличие от предыдущего варианта, к длине,
  // получаемой через WM_GETTEXTLENGTH, добавляется
  // единица, потому что нужно вручную учесть добавочный
  // байт для завершающего нуля.
  TextLen := SendMessage(Wnd, WM_GETTEXTLENGTH, 0, 0) + 1;
  // Выделяем требуемое количество памяти. Так как
  // компилятор не освободит эту память автоматически,
  // необходимо использовать блок try/finally, иначе будут
  // утечки памяти при исключениях.
  Text := StrAlloc(TextLen);
  try
    // Так как для буфера даже при пустом заголовке будет
    // выделен хотя бы один байт, здесь можно отправлять
```

```
    // WM_GETTEXT, не проверяя длину строки, как это было
    // в предыдущем варианте — буфер всегда будет
    // корректным.
    SendMessage(Wnd, WM_GETTEXT, TextLen, LParam(Text));
    // Обрезаем слишком длинную строку. Модифицировать
    // PChar сложнее, чем string. Вставка нуля в середину
    // строки приводит к тому, что все API-функции будут
    // игнорировать "хвост", но на работу StrDispose это не
    // повлияет, т. к. функция StrAlloc (а также прочие
    // функции выделения памяти для нуль-терминированных
    // строк модуля SysUtils) сохраняет размер выделенной
    // памяти рядом с самой строкой, и StrDispose
    // ориентируется именно на этот размер, а не на
    // завершающий ноль.
    if TextLen > 104 then
    begin
      (Text + 104)^ := #0;
      (Text + 103)^ := '.';
      (Text + 102)^ := '.';
      (Text + 101)^ := '.';
      (Text + 100)^ := ' ';
    end;
    GetClassName(Wnd, ClassName, ClassNameLen);
    if Text^ = #0 then
      NodeName := 'Без названия (' + ClassName + ')'
    else
      NodeName := Text + ' (' + ClassName + ')';
    Node := FormWindows.TreeWindows.Items.AddChild(ParentNode, NodeName);
    Node.Data := Pointer(Wnd);
    EnumChildWindows(Wnd, @EnumWindowsProc, LParam(Node));
  finally
    // Вручную освобождаем память, выделенную для буфера
    StrDispose(Text);
  end;
end;
```

Второй вариант функции `EnumWindowsProc` отличается от первого только тем, что для организации буфера для получения имени окна вместо переменной типа `string` используется переменная типа `PChar`. Соответственно, все манипуляции с динамической памятью теперь выполняются вручную, а просто отсечь конец слишком длинной строки и прибавить к результату другую строку (многоточие) мы не можем, приходится модифицировать строку по-

символьно. Тем не менее видно, что и с помощью типа `PChar` задача создания буфера для строки, возвращаемой API-функцией, достаточно легко решается.

1.2.2. Пример Line

Пример Line представляет собой невизуальный компонент `TLine`, который перехватывает оконные сообщения своего владельца (владельца в терминах VCL, разумеется, раз речь идет о неоконном компоненте). Компонент `TLine` рисует на своем владельце линию из точки (`StartX`, `StartY`) в точку (`EndX`, `EndY`) цветом `Color`. Пользователь может перемещать концы линии мышью. Достаточно разместить компонент `TLine` на форме, и на ней появится линия, которую пользователь может перемещать как во время проектирования формы, так и во время выполнения программы. Можно также разместить на форме, например, панель, и сделать ее владельцем компонента `TLine` — тогда линия будет рисоваться на панели. Но это можно сделать только во время исполнения программы, потому что владельцем всех компонентов, созданных во время проектирования формы, становится сама форма.

Чтобы установить компонент, нужно выполнить следующие действия:

1. Переписать с компакт-диска файлы Line.pas и Line.dcr в папку, где вы храните компоненты. Если такой папки еще нет, самое время создать ее. Где именно она будет расположена, значения не имеет, выбирайте любое удобное для вас место. Главное — это прописать эту папку в путях, где Delphi ищет компоненты. Чтобы сделать это в Delphi 7 и более ранних версиях, откройте меню **Tools\Environment Options**, в появившемся диалоговом окне выберите закладку **Library** и добавьте свою папку в поле **Library path**. В BDS 2006 и выше откройте меню **Tools\Options**, в появившемся диалоговом окне в дереве в левой части выберите пункт **Environment Options\Delphi Options\Library — Win32** и добавьте папку в поле **Library path**.
2. Создайте новый пакет (меню **File\New\Other**, в открывшемся окне выбрать **Package**). После этого в Delphi 7 и более ранних версиях откроется небольшое окно пакета. В BDS 2006 и более поздних версиях окно не откроется, но пакет появится в группе проектов (по умолчанию это окно **Project Manager** в правом верхнем углу главного окна). Сохраните пакет в ту же папку, где находится Line.pas, под любым именем, кроме Line (иначе будет конфликт имен).
3. Добавьте в пакет файл Line.pas. В BDS 2006 для этого необходимо с помощью правой кнопки мыши вызвать контекстное меню пакета в окне **Project Manager** и выбрать там пункт **Add**. В Delphi 7 и более ранних версиях в окне пакета нужно нажать кнопку **Add**.

4. Установите компонент. В BDS 2006 и выше для этого следует выбрать пункт **Install** в контекстном меню проекта, а в Delphi 7 и более ранних версиях — нажать кнопку **Install** в окне пакета. После этого в палитре компонентов у вас появится вкладка **Delphi Kingdom Samples**, а в ней — компонент `TLine`.

Если вы не хотите помещать компонент `TLine` в палитру компонентов (или у вас Turbo Delphi Explorer, и вы просто не имеете такой возможности), можно воспользоваться проектом LineSample, который во время выполнения создает два экземпляра `TLine`, владельцем одного из которых является форма, другого — панель.

Перехват сообщения владельца осуществляется путем изменения его свойства `WindowProc` — записи в него указателя на свой обработчик сообщений. Здесь можно применить один хитрый прием. Компонент `TLine` не имеет своей оконной процедуры, т. к., будучи прямым наследником класса `TComponent`, окном не является. Но метод `Dispatch` у него есть, поскольку он объявлен в классе `TObject`. В классе `TComponent` и в его предках метод `Dispatch` никогда не вызывается. Если мы напишем обработчик сообщений таким образом, что он будет передавать сообщения методу `Dispatch`, то сможем в нашем компоненте создавать свои методы для обработки сообщений, в которые метод `Dispatch` при необходимости будет передавать сообщения для обработки. Необработанные сообщения при этом будут передаваться в метод `DefaultHandler`, который у класса `TComponent` ничего не делает. Если мы перекроем `DefaultHandler` так, чтобы он вызывал оригинальный обработчик сообщений родителя, то все необработанные сообщения пойдут туда. Более того, вызов `inherited` из методов-обработчиков сообщений тоже будет приводить к вызову оригинального обработчика родителя, т. к. в данном случае `inherited` при отсутствии унаследованного обработчика приводит к вызову `DefaultHandler`. В листинге 1.24 показано объявление класса `TLine` и код его методов, относящихся к перехвату сообщений.

Листинг 1.24. Базовая часть класса `TLine`

```
type
  TLine = class(TComponent)
  private
    // FCoords хранит координаты линии. Начало линии
    // находится в точке (FCoords[0], FCoords[1]),
    // конец — в (FCoords[2], FCoords[3]).
    FCoords: array[0..3] of Integer;
    // Конструктор класса написан так, что владельцем TLine
    // может стать только TWinControl или его наследник.
```

```
    // Но свойство Owner имеет тип TComponent, поэтому при
    // использовании свойств и методов класса TWinControl
    // Owner придется каждый раз приводить к типу
    // TWinControl. Чтобы избежать приведений типа,
    // используется поле FWinOwner. Оно указывает на тот же
    // объект, что и Owner, но имеет тип TWinControl.
    FWinOwner: TWinControl;
    // Здесь хранится адрес обработчика сообщений, бывший до
    // перехвата.
    FOldProc: TWndMethod;
    // Цвет линии
    FColor: TColor;
    // Состояние линии. Если FStartMoving = True, в данный
    // момент пользователь перемещает начало линии, если
    // FEndMoving = True — ее конец.
    FStartMoving, FEndMoving: Boolean;
    // Если FDrawLine = False, линия не рисуется. Это
    // используется, когда нужно стереть линию.
    FDrawLine: Boolean;
    procedure WMPaint(var Msg: TWMPaint);
      message WM_PAINT;
    procedure WMLButtonDown(var Msg: TWMLButtonDown);
      message WM_LBUTTONDOWN;
    procedure WMLButtonUp(var Msg: TWMLButtonUp);
      message WM_LBUTTONUP;
    procedure WMMouseMove(var Msg: TWMMouseMove);
      message WM_MOUSEMOVE;
    procedure SetColor(Value: TColor);
    procedure SetCoord(Index, Value: Integer);
  protected
    // Этот метод будет новым обработчиком сообщений
    // владельца
    procedure HookOwnerMessage(var Msg: TMessage);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure DefaultHandler(var Msg); override;
  published
    property Color: TColor
      read FColor write SetColor default clWindowText;
    property StartX: Integer index 0
      read FCoords[0] write SetCoord default 10;
    property StartY: Integer index 1
      read FCoords[1] write SetCoord default 10;
```

```
    property EndX: Integer index 2
      read FCoords[2] write SetCoord default 50;
    property EndY: Integer index 3
      read FCoords[3] write SetCoord default 50;
  end;

...

constructor TLine.Create(AOwner: TComponent);
begin
  if not Assigned(AOwner) then
    raise EWrongOwner.Create(
      'Должен быть назначен владелец компонента TLine');
  if not (AOwner is TWinControl) then
    raise EWrongOwner.Create(
      'Владелец компонента TLine должен быть наследником TWinControl');
  FWinOwner := AOwner as TWinControl;
  inherited;
  FCoords[0] := 10;
  FCoords[1] := 10;
  FCoords[2] := 50;
  FCoords[3] := 50;
  FColor := clWindowText;
  FStartMoving := False;
  FEndMoving := False;
  FDrawLine := True;
  // Запоминаем старый обработчик сообщений владельца и
  // назначаем новый.
  FOldProc := FWinOwner.WindowProc;
  FWinOwner.WindowProc := HookOwnerMessage;
  FWinOwner.Refresh;
end;

destructor TLine.Destroy;
begin
  // Восстанавливаем старый обработчик сообщений владельца.
  FWinOwner.WindowProc := FOldProc;
  FWinOwner.Refresh;
  inherited;
end;

procedure TLine.HookOwnerMessage(var Msg: TMessage);
begin
```

```
  // Единственное, что делает перехватчик сообщений -
  // передает их методу Dispatch. Было бы оптимальнее
  // назначить обработчиком сообщений сам метод Dispatch,
  // но формально он имеет прототип, несовместимый с
  // типом TWndMethod, поэтому компилятор не разрешает
  // подобное присваивание. Фактически же Dispatch
  // совместим с TWndMethod, поэтому, используя хакерские
  // методы, можно было бы назначить обработчиком его и
  // обойтись без метода HookOwnerMessage. Но хакерские
  // методы — вещь небезопасная, они допустимы только
  // тогда, когда других средств решения задачи нет.
  Dispatch(Msg);
end;

procedure TLine.DefaultHandler(var Msg);
begin
  FOldProc(TMessage(Msg));
end;
```

Собственно рисование линии на поверхности владельца обеспечивает метод `WMPaint` (листинг 1.25).

Листинг 1.25. Метод `WMPaint`

```
procedure TLine.WMPaint(var Msg: TWMPaint);
var
  NeedDC: Boolean;
  PS: TPaintStruct;
  Pen: HPEN;
begin
  if FDrawLine then
  begin
    // Проверка, был ли DC получен предыдущим обработчиком
    NeedDC := Msg.DC = 0;
    if NeedDC then
      Msg.DC := BeginPaint (FWinOwner.Handle, PS);
    inherited;
    Pen := CreatePen(PS_SOLID, 1, ColorToRGB(FColor));
    SelectObject(Msg.DC, Pen);
    MoveToEx(Msg.DC, FCoords[0], FCoords[1], nil);
    LineTo(Msg.DC, FCoords[2], FCoords[3]);
    SelectObject(Msg.DC, GetStockObject(BLACK_PEN));
    DeleteObject(Pen);
```

```
    if NeedDC then
      EndPaint(FWinOwner.Handle, PS);
  end
  else
    inherited;
end;
```

Поскольку рисуется простая линия, мы не будем здесь создавать экземпляр TCanvas и привязывать его к контексту устройства, обойдемся вызовом функций GDI. Особенности работы с контекстом устройства при перехвате сообщения WM_PAINT описаны в *разд. 1.2.4*.

Чтобы пользователь мог перемещать концы линии, нужно перехватывать и обрабатывать сообщения, связанные с перемещением мыши и нажатием и отпусканием ее левой кнопки (листинг 1.26).

Листинг 1.26. Обработка сообщений мыши

```
procedure TLine.WMLButtonDown(var Msg: TWMLButtonDown);
var
  DC: HDC;
  OldMode: Integer;
begin
  if PTInRect(Rect(FCoords[0] – 3, FCoords[1] – 3,
    FCoords[0] + 4, FCoords[1] + 4),
    Point(Msg.XPos, Msg.YPos)) then
  begin
    FStartMoving := True;
    FDrawLine := False;
    FWinOwner.Refresh;
    FDrawLine := True;
    DC := GetDC (FWinOwner.Handle);
    OldMode := SetROP2(DC, R2_NOT);
    SelectObject(DC, GetStockObject(BLACK_PEN));
    MoveToEx(DC, FCoords[0], FCoords[1], nil);
    LineTo(DC, FCoords[2], FCoords[3]);
    SetROP2(DC, OldMode);
    ReleaseDC(FWinOwner.Handle, DC);
    SetCapture(FWinOwner.Handle);
    Msg.Result := 0;
  end
  else if PTInRect(Rect(FCoords[2] – 3, FCoords[3] – 3,
    FCoords[2] + 4, FCoords[3] + 4), Point(Msg.XPos, Msg.YPos)) then
```

```
  begin
    FEndMoving := True;
    FDrawLine := False;
    FWinOwner.Refresh;
    FDrawLine := True;
    DC := GetDC (FWinOwner.Handle);
    OldMode := SetROP2(DC, R2_NOT);
    SelectObject(DC, GetStockObject(BLACK_PEN));
    MoveToEx(DC, FCoords[0], FCoords[1], nil);
    LineTo(DC, FCoords[2], FCoords[3]);
    SetROP2(DC, OldMode);
    ReleaseDC(FWinOwner.Handle, DC);
    SetCapture(FWinOwner.Handle);
    Msg.Result := 0;
  end
  else
    inherited;
end;

procedure TLine.WMLButtonUp(var Msg: TWMLButtonUp);
begin
  if FStartMoving then
  begin
    FStartMoving := False;
    ReleaseCapture;
    FWinOwner.Refresh;
    Msg.Result := 0;
  end
  else if FEndMoving then
  begin
    FEndMoving := False;
    ReleaseCapture;
    FWinOwner.Refresh;
    Msg.Result := 0;
  end
  else
    inherited;
end;

procedure TLine.WMMouseMove(var Msg: TWMMouseMove);
var
  DC: HDC;
  OldMode: Integer;
```

```
begin
  if FStartMoving then
  begin
    DC := GetDC (FWinOwner.Handle);
    OldMode := SetROP2(DC, R2_NOT);
    SelectObject(DC, GetStockObject(BLACK_PEN));
    MoveToEx(DC, FCoords[0], FCoords[1], nil);
    LineTo(DC, FCoords[2], FCoords[3]);
    FCoords[0] := Msg.XPos;
    FCoords[1] := Msg.YPos;
    MoveToEx(DC, FCoords[0], FCoords[1], nil);
    LineTo(DC, FCoords[2], FCoords[3]);
    SetROP2(DC, OldMode);
    ReleaseDC(FWinOwner.Handle, DC);
    Msg.Result := 0;
  end
  else if FEndMoving then
  begin
    DC := GetDC (FWinOwner.Handle);
    OldMode := SetROP2(DC, R2_NOT);
    SelectObject(DC, GetStockObject(BLACK_PEN));
    MoveToEx(DC, FCoords[0], FCoords[1], nil);
    LineTo(DC, FCoords[2], FCoords[3]);
    FCoords[2] := Msg.XPos;
    FCoords[3] := Msg.YPos;
    MoveToEx(DC, FCoords[0], FCoords[1], nil);
    LineTo(DC, FCoords[2], FCoords[3]);
    SetROP2(DC, OldMode);
    ReleaseDC(FWinOwner.Handle, DC);
    Msg.Result := 0;
  end
  else
    inherited;
end;
```

Здесь реализован инверсный способ создания "резиновой" линии, когда при рисовании линии все составляющие ее пикселы инвертируются, а при стирании инвертируются еще раз. Этот способ подробно описан в *разд. 1.3.4.2*.

Перехват сообщений родителя — дело относительно простое, гораздо хуже обстоят дела с удалением компонента, перехватившего сообщения родителя. Пока такой компонент один, проблем не возникает, но когда их несколько, приходится обращаться с ними очень аккуратно. Рассмотрим, например, такой код (листинг 1.27).

Листинг 1.27. Пример кода, вызывающего ошибку

```
Line1 := TLine.Create(Form1);
Line2 := TLine.Create(Form2);
...
Line1.Free;
...
Line2.Free;
```

Проанализируем, что происходит при выполнении этого кода. Для простоты предположим, что других компонентов, перехватывающих сообщения, здесь нет, и перед выполнением этого кода `Form1.WindowProc` ссылается на `Form1.WndProc`, т. е. на собственный обработчик сообщений формы. При создании объекта `Line1` он перехватывает обработчик, и `Form1.WindowProc` начинает ссылаться на `Line1.HookOwnerMessage`, а ссылка на `Form1.WndProc` сохраняется в `Line1.FOldProc`. Объект `Line2` также перехватывает обработчик сообщений, и после его создания `Form1.WindowProc` будет ссылаться на `Line2.HookOwnerMessage`, а `Line2.FOldProc` — на `Line1.HookOwnerMessage`.

Теперь удалим `Line1`. При удалении объект восстановит ссылку на тот обработчик сообщений, который был установлен на момент его создания, т. е. `Form1.WindowProc` вновь станет указывать на `Form1.WndProc`. Соответственно, компонент `Line2` потеряет способность реагировать на сообщения владельца. Поле `Line2.FOldProc` при этом останется без изменений.

Но самое неприятное начнется при удалении объекта `Line2`. Он тоже восстановит ссылку на обработчик, который был назначен на момент его создания, т. е. запишет в свойство `Form1.WindowProc` ссылку на `Line1.HookOwnerMessage`. Но поскольку объекта `Line1` уже не существует, это будет ссылка в никуда, и обработка первого же сообщения, пришедшего форме, даст ошибку Access violation.

Примечание

Аналогичная проблема возникнет и в режиме проектирования, если на форму положить два компонента `TLine`, удалить первый, а затем — второй. В этом случае ошибки возникнут в самой среде Delphi, и ее придется перезапускать. Вообще говоря, компоненты, перехватывающие сообщения владельца, должны делать это только во время выполнения программы, чтобы не "уронить" среду. Здесь мы для наглядности опустили соответствующие проверки.

Проблема не возникает, если удалять объекты в порядке, обратном порядку их создания. Но в общем случае это не может быть решением проблемы, т. к. объекты должны создаваться и удаляться в том порядке, который требуется логикой работы программы. Соответственно, единственное решение — все

перехватывающие сообщения владельца компоненты должны знать друг о друге и уведомлять друг друга о своем создании и удалении. Но и этот способ не дает полной гарантии. Пока один разработчик пишет компонент или библиотеку компонентов, он может обеспечить взаимодействие всех экземпляров компонентов в программе. Но если в одной программе будут использованы две такие библиотеки от разных разработчиков, они так же будут конфликтовать друг с другом, и универсального решения проблемы, судя по всему, не существует. Пользователю библиотек остается только соблюдать порядок удаления компонентов. Но, с другой стороны, есть ряд задач, в которых без перехвата сообщений владельца не обойтись, поэтому иногда приходится идти на это.

1.2.3. Пример CoordLabel

CoordLabel — это пример визуального компонента, перехватывающего сообщения своего родителя. Компонент `TCoordLabel` отслеживает нажатие левой кнопки мыши на своем родителе и отображает координаты точки, в которой произошло нажатие. Для перехвата сообщений родителя используется тот же способ через свойство `WindowProc`, что и в предыдущем примере, но т. к. теперь перехватываются сообщения родителя, а не владельца, появляются некоторые нюансы.

Установка компонента `TCoordLabel` полностью аналогична установке компонента `TLine` из предыдущего раздела. На прилагаемом компакт-диске находится также проект LineCoordSample для того, чтобы работу компонента можно было увидеть без установки в палитру компонентов. На форме проекта LineCoordSample находится панель, кнопка **Переместить** и компонент `TLineCoordSample`, который по нажатию кнопки меняет родителя с формы на панель и обратно.

Код компонента `TCoordLabel` приведен в листинге 1.28.

Листинг 1.28. Компонент `TCoordLabel`

```
type
  TCoordLabel = class(TLabel)
  private
    // Здесь хранится адрес обработчика
    // сообщений, бывший до перехвата.
    FOldProc: TWndMethod;
  protected
    procedure SetParent(AParent: TWinControl); override;
    // Этот метод будет новым обработчиком
    // сообщений владельца
```

```
    procedure HookParentMessage(var Msg: TMessage);
  end;

...

procedure TCoordLabel.SetParent(AParent: TWinControl);
begin
  if Assigned(Parent) and Assigned(FOldProc) then
    Parent.WindowProc := FOldProc;
  inherited;
  if Assigned(Parent) then
  begin
    FOldProc := Parent.WindowProc;
    Parent.WindowProc := HookParentMessage;
  end;
end;

procedure TCoordLabel.HookParentMessage(var Msg: TMessage);
begin
  if Msg.Msg = WM_LBUTTONDOWN then
    Caption := '(' + IntToStr(Msg.LParamLo) + ', ' +
      IntToStr(Msg.LParamHi) + ')';
  FOldProc(Msg);
end;
```

Класс `TLabel`, предок `TCoordLabel`, является визуальным компонентом и сам может получать и обрабатывать сообщения, поэтому метод `Dispatch` у него уже "занят". Соответственно, мы не можем диспетчеризовать с его помощью перехваченные сообщения и должны обрабатывать их внутри метода `HookParentMessage`.

Сам перехват осуществляется не в конструкторе, т. к. на момент вызова конструктора родитель компонента еще неизвестен. Он устанавливается позже, через свойство `Parent`, которое приводит к вызову виртуального метода `SetParent`. Мы перекрываем этот метод и выполняем в нем как восстановление обработчика старого родителя, так и перехват сообщений нового. Это позволяет компоненту менять родителя во время работы программы.

Писать отдельно деструктор для восстановления оригинального обработчика родителя в данном случае нужды нет, поскольку деструктор, унаследованный от `TControl`, содержит вызов метода `SetParent` с параметром `nil`. Так как мы уже перекрыли `SetParent`, это приведет к восстановлению оригинального обработчика, т. е. к тому, что нам нужно.

Если на форму, содержащую TCoordLabel, поместить другие компоненты, можно заметить, что TCoordLabel отлавливает нажатия мыши, сделанные на неоконных компонентах, но игнорирует те, которые сделаны на оконных. Это происходит потому, что неоконные компоненты получают сообщения через оконную процедуру родителя (которая перехвачена), а оконные имеют свою оконную процедуру, никак не связанную с оконной процедурой родителя.

И, разумеется, компонент TCoordLabel имеет те же проблемы с восстановлением оригинального обработчика, что и TLine, если на одном родителе расположены несколько компонентов. Соответственно, применять TCoordLabel необходимо аккуратно, с учетом возможных последствий.

1.2.4. Пример PanelMsg

Программа PanelMsg показывает, как можно перехватить оконные сообщения, поступающие компоненту, лежащему на форме. В данном случае этим компонентом будет TPanel. Для перехвата сообщений используется свойство WindowProc панели.

Мы будем обрабатывать два сообщения, приходящих с панели: WM_RBUTTONDBLCLK и WM_PAINT. Таким образом, наша панель получит возможность реагировать на двойной щелчок правой кнопки мыши, а также рисовать что-то на своей поверхности. С помощью одной только библиотеки VCL это сделать нельзя.

Примечание

Для рисования на поверхности панели, вообще говоря, существует более простой и правильный способ: нужно положить на панель компонент TPaintBox, растянуть его на всю область панели и рисовать в его событии OnPaint. Мы здесь используем более сложный способ перехвата сообщения WM_PAINT только в учебных целях.

При перехвате сообщения WM_PAINT любого компонента, на котором расположены неоконные визуальные компоненты, может возникнуть проблема с перерисовкой этих компонентов. Чтобы продемонстрировать способ решения этих проблем, разместим на панели компонент TLabel, который заодно будет показывать пользователю реакцию на двойной щелчок правой кнопкой мыши. В результате получается окно, показанное на рис. 1.9. При двойном щелчке правой кнопкой мыши на панели надпись **Сделайте двойной щелчок правой кнопкой** перемещается в то место, где находится курсор.

Чтобы перехватить оконную процедуру панели, следует написать метод, который ее подменит, а адрес старого метода сохранить в предназначенном для этого поле. Сам перехват будем осуществлять в обработчике события OnCreate формы (листинг 1.29).

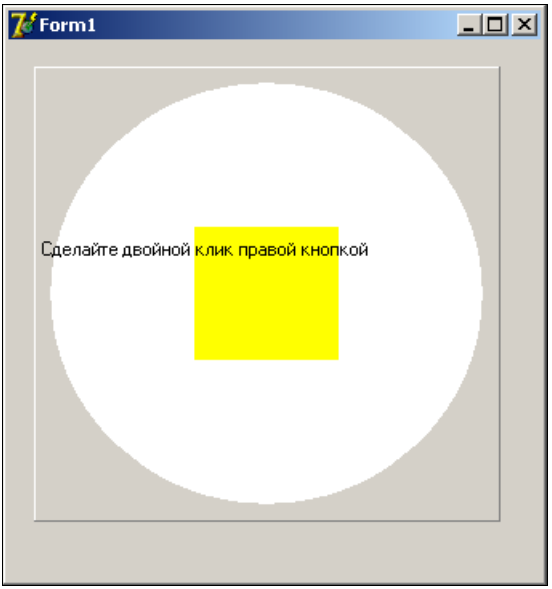


Рис. 1.9. Окно программы PanelMsg

Листинг 1.29. Перехват обработчика сообщений панели

```
type
  TForm1 = class(TForm)
    Panel: TPanel;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
  private
    // Здесь будет храниться исходный обработчик сообщений
    // панели
    FOldPanelWndProc: TWndMethod;
    // Этот метод формы будет перехватывать сообщения,
    // предназначенные панели
    procedure NewPanelWndProc(var Msg: TMessage);
  end;

...

procedure TForm1.FormCreate(Sender: TObject);
begin
  FOldPanelWndProc := Panel.WindowProc;
  Panel.WindowProc := NewPanelWndProc;
end;
```

Сам перехватчик выглядит так, как показано в листинге 1.30.

Листинг 1.30. Метод-перехватчик сообщений панели

```
procedure TForm1.NewPanelWndProc(var Msg: TMessage);
var
  NeedDC: Boolean;
  PS: TPaintStruct;
  PanelCanvas: TCanvas;
begin
  if Msg.Msg = WM_RBUTTONDBLCLK then
  begin
    Label1.Left := Msg.LParamLo;
    Label1.Top := Msg.LParamHi;
    Msg.Result := 0;
  end
  else if Msg.Msg = WM_PAINT then
  begin
    // Проверяем, был ли запрошен контекст устройства
    // обработчиком, стоящим раньше по цепочке, и если не
    // был, то запрашиваем его.
    NeedDC := Msg.WParam = 0;
    if NeedDC then
      Msg.WParam := BeginPaint (Panel.Handle, PS);
    // Вызываем старый обработчик WM_PAINT. Его нужно
    // вызывать обязательно до того, как мы начнем рисовать
    // на поверхности что-то свое, т. к. в противном случае
    // это что-то будет закрашено стандартным обработчиком.
    FOldPanelWndProc(Msg);
    // При использовании графических функций API самое
    // неудобное — это вручную создавать и уничтожать кисти,
    // карандаши и т. п. Поэтому здесь создается экземпляр
    // класса TCanvas для рисования на контексте устройства
    // с дескриптором, полученным при вызове BeginPaint.
    PanelCanvas := TCanvas.Create;
    try
      PanelCanvas.Handle := Msg.WParam;
      PanelCanvas.Pen.Style := psClear;
      PanelCanvas.Brush.Style := bsSolid;
      PanelCanvas.Brush.Color := clWhite;
      PanelCanvas.Ellipse(10, 10, Panel.Width — 10, Panel.Height — 10);
      PanelCanvas.Brush.Color := clYellow;
      PanelCanvas.Rectangle(100, 100,
        Panel.Width — 100, Panel.Height — 100);
```

```
    finally
      PanelCanvas.Free;
    end;
    // В данном случае панель содержит визуальный неоконный
    // компонент TLabel. Отрисовка неоконных компонентов
    // происходит при обработке WM_PAINT родительского
    // компонента, т. е. здесь она была выполнена при вызове
    // стандартного обработчика. Таким образом, сделанный
    // рисунок закрасил не только фон панели, но и
    // неоконные компоненты. Чтобы компоненты были поверх
    // рисунка, их приходится перерисовывать еще раз,
    // вызывая protected-метод PaintControls. Это не очень
    // эффективно, т. к. получается, что компоненты рисуются
    // дважды: в стандартном обработчике и здесь. Но
    // другого способа решить проблему, видимо, нет. Если
    // бы на панели лежали только оконные компоненты,
    // вызывать PaintControls не понадобилось, поскольку то, что
    // мы рисуем на панели, не может затереть поверхность
    // лежащих на этой панели других окон.
    TFakePanel(Panel).PaintControls(Msg.WParam, nil);
    // Если мы получали контекст устройства, мы же должны
    // освободить его.
    if NeedDC then
    begin
      EndPaint(Panel.Handle, PS);
      Msg.WParam := 0;
    end;
  end
  else
    FOldPanelWndProc(Msg);
end;
```

Так как в наш обработчик поступают все сообщения, передающиеся в оконную процедуру панели, начинается он с проверки того, какое сообщение пришло. Сначала реализуем реакцию на `WM_RBUTTONDBLCLK`: просто перемещаем метку `Label1` на то место, где пользователь щелкнул мышью. Затем обнуляем результат, давая понять системе, что сообщение полностью обработано. Вызов унаследованного обработчика в данном случае не выполняем, потому что никакая унаследованная реакция на данное событие нам не нужна.

Обработка сообщения `WM_PAINT` сложнее. Сначала необходимо разобраться с контекстом устройства, на котором будет производиться рисование. Вообще говоря, обработчик `WM_PAINT` должен получать этот контекст с помощью функции `BeginPaint`. Но если до написанного нами кода сообщение `WM_PAINT`

уже начало обрабатываться, то контекст устройства уже получен, а вызывать BeginPaint два раза нельзя. В этом случае контекст устройства передается через параметр сообщения WParam. Соответственно, обработка сообщения WM_PAINT начинается с того, что мы проверяем, равен ли нулю параметр WParam, и если равен, то получаем контекст устройства, а если не равен, используем то, что передано.

Унаследованный обработчик закрашивает всю панель целиком, поэтому его нужно вызывать до того, как мы нарисуем что-то свое, иначе он просто закрасит то, что мы нарисовали. Так что следующий шаг — это вызов стандартного обработчика сообщений панели, указатель на который мы сохранили в поле FOldPanelWndProc. Только после этого можно что-то рисовать.

Примечание

Перекрывая обработку сообщения WM_PAINT, мы лишаем код VCL возможности полностью контролировать процесс перерисовки. В частности, это означает, что значение свойства DoubleBuffered будет игнорироваться, двойной буферизации не будет. Поэтому еще раз напоминаем, что программа PanelMsg — это учебный пример, помогающий разобраться с механизмами взаимодействия VCL и Windows API, но не являющийся образцом для подражания. Если в реальной жизни потребуется рисовать что-то непосредственно на панели, нужно порождать от класса TPanel наследника и перекрывать в нем метод Paint.

Теперь можно нарисовать что-то свое. Здесь мы рисуем большой белый круг, а на его фоне — желтый прямоугольник. Для этого используем класс TCanvas способом, который был продемонстрирован в листинге 1.17 *(см. разд. 1.1.11)*.

Если бы мы остановились на этом, то увидели бы интересную картину: нарисованные фигуры лежат поверх текста метки Label1. Объяснение этому очень простое: метка является неоконным визуальным компонентом и рисуется на поверхности своего родительского компонента при обработке его сообщения WM_PAINT. А поскольку стандартный обработчик у нас вызывается до того, как рисуются круг и прямоугольник, любой неоконный компонент будет перекрыт ими. К оконным компонентам это, разумеется, не относится, они лежат над родительской панелью, и то, что мы рисуем на этой панели, не может оказаться над ними.

Мы не можем вставить свой код между рисованием непосредственно поверхности панели и рисованием компонентов на ней. Поэтому после отработки нашего кода приходится рисовать неоконные компоненты еще раз. Проще всего это сделать, вызвав метод PaintControls, который и используется стандартным обработчиком. Конечно, получится, что неоконные компоненты рисуются дважды: в стандартном обработчике и в нашем, и это не очень хорошо. Но повторим еще раз, что программа PanelMsg — не образец для подражания, а что-то вроде зонда для исследования особенностей работы VCL.

Вызов метода PaintControls затруднен тем, что он объявлен в разделе protected, а потому не может быть вызван из метода NewPanelWndProc, который относится к классу формы. Чтобы обойти это ограничение, нужно породить наследника от TPanel — TFakePanel. Этот наследник ничего не добавляет к классу TPanel и ничего не переопределяет в нем. Но раз он объявлен в нашем модуле, все его protected-члены, в том числе и унаследованный метод PaintControls, становятся доступными в нем. После этого мы можем привести поле, содержащее ссылку на панель, к этому типу и вызвать PaintControls. Так как структуры типов TPanel и TFakePanel идентичны, это приведет к вызову нужного метода.

Для завершения обработки сообщения WM_PAINT осталось только вызвать EndPaint, разумеется, только в том случае, если BeginPaint вызывали мы сами.

И последнее, что мы должны сделать, — это передать все остальные сообщения стандартному обработчику. После этого программа PanelMsg готова.

1.2.5. Пример NumBroadcast

Программа NumBroadcast демонстрирует широковещательную рассылку глобальных сообщений. Окно программы показано на рис. 1.10.

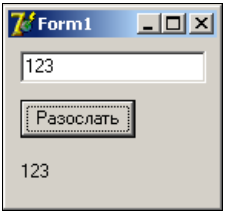


Рис. 1.10. Окно программы NumBroadcast

Для того чтобы увидеть, как работает программа, нужно запустить несколько ее экземпляров. После ввода числа и нажатия кнопки **Разослать** любому из экземпляров программы число под кнопкой меняется во всех экземплярах. Чтобы добиться такого эффекта, программа NumBroadcast регистрирует глобальное сообщение с помощью функции RegisterWindowMessage, а в оконной процедуре предусмотрена реакция на это сообщение (число передается через параметр WParam). Код программы приведен в листинге 1.31.

Листинг 1.31. Модуль главного окна программы NumBroadcast

```
unit NBMain;

interface
```

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

type
  TForm1 = class(TForm)
    EditNumber: TEdit;
    BtnBroadcast: TButton;
    LabelNumber: TLabel;
    procedure BtnBroadcastClick(Sender: TObject);
  private
    // Здесь будет храниться номер, присвоенный системой
    // глобальному сообщению
    FSendNumberMessage: Cardinal;
  protected
    // Так как номер сообщения станет известным только при
    // выполнении программы, объявить обработчик сообщения
    // с помощью директивы message нельзя. Приходится
    // перекрывать метод WndProc и обрабатывать сообщение в
    // нем. Можно было бы вместо WndProc перекрыть метод
    // DefaultHandler, но при этом зарегистрированное
    // сообщение обрабатывалось бы медленнее, потому что
    // сначала выполнялся бы метод WndProc, затем Dispatch
    // пытался бы найти подходящий обработчик среди методов
    // объекта, и лишь затем дело доходило бы до перекрытого
    // DefaultHandler. Но, с другой стороны, при перекрытии
    // WndProc обработка всех сообщений начинается со
    // сравнения их номера с FSendNumberMessage и вызова
    // унаследованного WndProc, если это другое сообщение.
    // А до DefaultHandler многие сообщения не дойдут, т. к.
    // будут обработаны ранее, и накладные расходы на
    // сравнение и вызов унаследованного метода будут меньше.
    procedure WndProc(var Msg: TMessage); override;
  public
    constructor Create(AOwner: TComponent); override;
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
```

```
constructor TForm1.Create(AOwner: TComponent);
 begin
  // Регистрируем глобальное сообщение с именем
  // WM_DelphiKingdom_APISample_SendNumber. Имя достаточно
  // длинное и осмысленное, поэтому можно надеяться, что
  // никакое другое приложение не зарегистрирует сообщение с
  // таким же именем. Регистрация сообщения выполняется до
  // вызова унаследованного конструктора, т. к. при
  // выполнении этого конструктора окно получит ряд
  // сообщений, и метод WndProc будет несколько раз вызван.
  // Если вызвать унаследованный конструктор до вызова
  // RegisterWindowMessage, то поле FSendNumberMessage
  // будет иметь присвоенное ему по умолчанию значение 0,
  // а это — код сообщения WM_NULL. Таким образом, если в
  // это время окно получит сообщение WM_NULL, оно будет
  // неправильно обработано. Конечно, вероятность получения
  // WM_NULL во время выполнения унаследованного
  // конструктора крайне мала, но лучше подстраховаться и
  // сделать так, чтобы поле FSendNumberMessage на момент
  // первого вызова WndProc уже имело правильное значение.
  FSendNumberMessage := RegisterWindowMessage (
    'WM_DelphiKingdom_APISample_SendNumber');
  inherited;
  // Здесь мы меняем стиль окна поля ввода, добавляя в него
  // ES_NUMBER. Стиль ES_NUMBER запрещает полю ввода
  // вводить какие-либо символы, кроме цифр. Это уменьшает
  // риск ошибки ввода в тех случаях, когда требуется целое
  // неотрицательное число.
  SetWindowLong(EditNumber.Handle, GWL_STYLE,
    GetWindowLong(EditNumber.Handle, GWL_STYLE) or ES_NUMBER);
 end;

procedure TForm1.BtnBroadcastClick(Sender: TObject);
var
  Num: Integer;
  Recipients: DWORD;
begin
  try
    Num := StrToInt(EditNumber.Text);
    // Для широковещательной рассылки сообщения служит
    // функция BroadcastSystemMessage. В литературе обычно
    // советуют использовать более простую функцию
    // PostMessage, указывая в качестве адресата
```

```
    // HWND_BROADCAST. Однако PostMessage рассылает
    // сообщения только окнам верхнего уровня, не имеющим
    // владельца (в терминах системы). Но главная форма
    // приложения имеет владельца — это невидимое окно
    // приложения, реализуемое объектом TApplication.
    // Поэтому такое широковещательное сообщение главная
    // форма приложения не получит — его получит только
    // невидимое окно приложения (это сообщение можно
    // будет перехватить, назначив обработчик
    // Application.OnMessage — вручную или с помощью
    // компонента TApplicationEvents). Чтобы главная форма
    // тоже попала в список окон, получающих
    // широковещательное сообщение, используется функция
    // BroadcastSystemMessage.
    Recipients := BSM_APPLICATIONS;
    BroadcastSystemMessage(BSF_POSTMESSAGE, @Recipients,
      FSendNumberMessage, Num, 0);
  except
    on EConvertError do
    begin
      Application.MessageBox(
        'Введенное значение не является числом', 'Ошибка',
        MB_OK or MB_ICONSTOP);
    end;
  end;
end;

procedure TForm1.WndProc(var Msg: TMessage);
begin
  if Msg.Msg = FSendNumberMessage then
    LabelNumber.Caption := IntToStr(Msg.WParam)
  else
    inherited;
end;

end.
```

Как уже отмечалось ранее, для обработки глобального сообщения нельзя использовать методы с директивой `message`, т. к. номер сообщения на этапе компиляции еще не известен. Здесь для обработки глобального сообщения мы перекрываем метод `WndProc`. Соответственно, все оконные сообщения, в том числе и те, которые окно получает при создании, будет обрабатывать перекрытый метод `WndProc`. Это значит, что поле `FSendNumberMessage`, которое

задействовано в этом методе, должно быть правильно инициализировано раньше, чем окно получит первое сообщение. Поэтому вызов функции RegisterWindowMessage выполнять, например, в обработчике события OnCreate формы уже поздно. Его необходимо выполнить в конструкторе формы, причем до того, как будет вызван унаследованный конструктор.

Примечание

Существует другой способ решения этой проблемы: метод WndProc должен проверять значение поля FSendNumberMessage, и, если оно равно нулю, сразу переходить к вызову унаследованного метода. В этом случае инициализировать FSendNumberMessage можно позже.

Нажатие на кнопку BtnBroadcast приводит к широковещательной отправке сообщения. Отправить широковещательное сообщение можно двумя способами: функцией PostMessage с адресатом HWND_BROADCAST вместо дескриптора окна и с помощью функции BroadcastSystemMessage. Первый вариант позволяет отправить сообщения только окнам верхнего уровня, не имеющим владельца в терминах системы. Таким окном в VCL-приложении является только невидимое окно приложения, создаваемое объектом Application. Главная форма имеет владельца в терминах системы — то самое невидимое окно приложения. Поэтому широковещательное сообщение, посланное с помощью PostMessage, главная форма не получит, это сообщение пришлось бы ловить с помощью события Application.OnMessage. Мы здесь применяем другой способ — отправляем сообщение с помощью функции BroadcastSystemMessage, которая позволяет указывать тип окон, которым мы хотим отправить сообщения. В частности, здесь мы выбираем тип BSM_APPLICATION, чтобы сообщение посылалось всем окнам верхнего уровня, в том числе и тем, которые имеют владельца. При таком способе отправки главная форма получит это широковещательное сообщение, поэтому его обработку можно реализовать в главной форме.

1.2.6. Пример ButtonDel

Программа ButtonDel демонстрирует, как можно удалить кнопку в обработчике нажатия этой кнопки. Очень распространенная ошибка — попытка написать код, один из примеров которого приведен в листинге 1.32.

Листинг 1.32. Неправильный вариант удаления кнопки в обработчике ее нажатия

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Free;
end;
```

Рассмотрим, что произойдет в случае выполнения этого кода. Когда пользователь нажимает на кнопку, форма получает сообщение WM_COMMAND. При его обработке форма выясняет, что источником сообщения является объект Button1 и передает этому объекту сообщение CN_COMMAND. Button1, получив его, вызывает метод Click, который проверяет, назначен ли обработчик OnClick, и, если назначен, вызывает его. Таким образом, после завершения Button1Click управление снова вернется в метод Click объекта Button1, из него — в метод CNCommand, из него — в Dispatch, оттуда — в WndProc, а оттуда — в MainWndProc. А из MainWndProc управление будет передано в оконную процедуру, сформированную компонентом с помощью MakeObjectInstance. В деструкторе Button1 эта оконная процедура будет уже удалена. Таким образом, управление получат последовательно пять методов уже не существующего объекта и одна несуществующая процедура. Это может привести к самым разным неприятным эффектам, но, скорее всего, — к ошибке Access violation (обращение к памяти, которую программа не имеет права использовать). Поэтому приведенный в листинге 1.32 код будет неработоспособным. В классе TCustomForm для безопасного удаления формы существует метод Release, который откладывает уничтожение объекта до того момента, когда это будет безопасно, но остальные компоненты подобного метода не имеют.

Примечание

Метод TCustomForm.Release на проверку тоже оказывается не совсем безопасным — подробнее об этом написано в *разд. 3.4.3*.

Очевидно, что для безопасного удаления кнопки эту операцию следует отложить до того момента, когда все методы удаляемой кнопки уже закончат свою работу. Вставить требуемый код в обработчик WM_COMMAND формы достаточно сложно, поэтому мы будем использовать другой способ: пусть обработчик кнопки посылает форме сообщение, в обработчике которого она будет удалять кнопку. Здесь важно, что сообщение посылается, а не отправляется, т. е. ставится в очередь, из которой извлекается уже после того, как будет полностью обработано сообщение WM_COMMAND. В этом случае методы удаляемой кнопки не будут вызваны, и удаление пройдет без неприятных последствий.

Как раз для подобных случаев и предусмотрена возможность определять свои сообщения, т. к. ни одно из стандартных для наших целей не подходит. Свое сообщение мы будем посылать только одному окну, без широковещания, поэтому для него вполне подходит диапазон сообщений класса. Номер сообщения становится известным на этапе компиляции, поэтому для обработки этого сообщения мы можем применить самый удобный способ — написать метод-обработчик с помощью директивы message. С учетом всего этого код выглядит следующим образом (листинг 1.33).

Листинг 1.33. Модуль главной формы программы ButtonDel

```
unit BDMain;

interface

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

// Определяем свое сообщение. Константа, добавляемая к
// WM_USER, может иметь произвольное значение в диапазоне
// от 0 до 31743.
const
  WM_DELETEBUTTON = WM_USER + 1;

type
  TForm1 = class(TForm)
    BtnDeleteSelf: TButton;
    procedure BtnDeleteSelfClick(Sender: TObject);
  private
    // Определяем метод — обработчик события WM_DELETEBUTTON.
    // Ему будет передано управление через Dispatch.
    procedure WMDeleteButton(var Msg: TMessage); message WM_DELETEBUTTON;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.BtnDeleteSelfClick(Sender: TObject);
begin
  // Помещаем сообщение WM_DELETEBUTTON в очередь формы.
  // Указатель на объект, который нужно удалить, помещаем
  // в LParam. В 32-разрядных версиях Windows указатель
  // можно помещать как в wParam, так и в lParam, но по
  // традиции, берущей начало в 16-разрядных версиях,
  // указатель обычно передают через lParam.
  PostMessage(Handle, WM_DELETEBUTTON, 0, LParam(BtnDeleteSelf));
```

```
  // Здесь принципиально использование PostMessage, а не
  // SendMessage. SendMessage в данном случае привел бы к
  // немедленному вызову оконной процедуры, и метод
  // WMDeleteButton был бы вызван до завершения работы
  // BtnDeleteSelfClick. Это привело бы к тому же
  // результату, что и прямой вызов BtnDeleteSelf.Free.
end;

procedure TForm1.WMDeleteButton(var Msg: TMessage);
begin
  // Просто удаляем объект, указатель на который передан
  // через lParam.
  TObject(Msg.LParam).Free;
end;

end.
```

Приведенный здесь способ хорошо работает в такой простой ситуации, но в более сложных случаях может не дать результата. Рассмотрим, например, ситуацию, когда на форме лежат две кнопки: `Button1` и `Button2`. Обработчик нажатия `Button1` содержит длительную операцию, и поэтому в нем вызывается `Application.ProcessMessages`. Обработчик нажатия `Button2` содержит строку `Button1.Free`. Если после запуска программы сразу нажать `Button2`, проблем не возникнет и объект `Button1` будет благополучно удален. Но если сначала нажать `Button1`, а затем — `Button2`, возникнет ошибка. Это произойдет потому, что нажатие `Button2` будет в данном случае обработано локальной петлей сообщения, и после обработки управление вернется `Button1Click`, а оттуда — в методы уже не существующего объекта `Button1`. Посылка в `Button2Click` сообщения форме здесь не поможет, потому что это сообщение также будет извлечено и обработано локальной петлей. Общего решения таких проблем, видимо, не существует. В сложных случаях можно посоветовать не удалять объект, а просто прятать его (`Visible := False`) — видимый результат для пользователя будет тот же самый.

1.2.7. Пример GDIDraw

Программа GDIDraw демонстрирует некоторые возможности GDI, которые не поддерживаются классом `TCanvas`. Выбраны только те возможности, которые поддерживаются не только в Windows NT/2000/XP, но и в 9*x*/ME. Окно программы показано на рис. 1.11.

В своей работе программа использует рисунок из стандартных картинок Delphi, предполагая, что эти картинки установлены в папку "C:\Program

Files\Common Files\Borland Shared\Images". Если у вас эти картинки установлены в другую папку, или по каким-то причинам вы хотите выбрать другой рисунок, измените обработчик события `OnCreate` формы так, чтобы он загружал рисунок из нужного вам файла. Загруженный рисунок сохраняется в поле `FBitmap` формы.

Рис. 1.11. Окно программы GDIDraw

Основная работа выполняется в обработчике события `OnPaint` формы. Мы здесь будем разбирать этот обработчик не целиком, а по частям в соответствии с тем, что каждая часть рисует. Начнем с надписи **Delphi Kingdom** в левом верхнем углу окна (листинг 1.34).

Листинг 1.34. Вывод надписи *Delphi Kingdom*

```
var
  R: TRect;
...
// Формируем регион, использующийся для отсечения.
// Формируем его только при первом вызове метода, а при
// дальнейших используем созданный ранее. Поле FRgn
// содержит дескриптор этого региона
```

```
if FRgn = 0 then
begin
  Canvas.Font.Name := 'Times New Roman';
  Canvas.Font.Style := [fsBold];
  Canvas.Font.Height := 69;
  // Начинаем рисование траектории. Все вызовы
  // графических функций, находящиеся между BeginPath
  // и EndPath, не будут приводить к выводу на экран.
  // Вместо этого информация о том, что рисуется, будет
  // сохраняться в специальном объекте GDI — траектории.
  BeginPath(Canvas.Handle);
  R := Rect(10, 10, 10 + FBitmap.Width, 10 + FBitmap.Height);
  // Если не установить с помощью SetBkMode прозрачный
  // фон, в траекторию попадут не только контуры букв,
  // но и контуры содержащих их прямоугольных знакомест.
  SetBkMode(Canvas.Handle, TRANSPARENT);
  // Выводим текст "Delphi Kingdom", выравнивая его по
  // центру по вертикали и горизонтали.
  DrawText(Canvas.Handle, 'Delphi'#13#10'Kingdom', -1, R,
    DT_CENTER or DT_VCENTER);
  EndPath(Canvas.Handle);
  // Превращаем траекторию в регион. В результате вызова
  // этой функции получится регион, контуры которого
  // совпадают с контурами надписи "Delphi Kingdom",
  // сделанной в указанных координатах выбранным шрифтом.
  FRgn := PathToRegion(Canvas.Handle);
end;
// Устанавливаем регион отсечения. Все, что не будет
// попадать в выбранный регион, при выводе будет
// игнорироваться.
SelectClipRgn(Canvas.Handle, FRgn);
// Выводим изображение. Все, что не попадает в область
// региона, отсекается. Таким образом, получаем надпись
// "Delphi Kingdom", подсвеченную выбранным изображением.
Canvas.Draw(10, 10, FBitmap);
// Отменяем отсечение по региону
SelectClipRgn(Canvas.Handle, 0);
```

Если присмотреться к надписи, видно, что внутренняя часть контуров букв содержит тот самый рисунок, который был загружен в обработчике `OnCreate` (как будто мы нарисовали этот рисунок через трафарет, имеющий форму надписи). По сути, так оно и есть, только называется это не трафарет, а *регион отсечения*. Регион — это специальный объект, который хранит об-

ласть произвольной формы. Способы применения регионов различны (см. *разд. 1.3.3*), и один из них — это использование региона для отсечения графического вывода. Если установить регион отсечения для контекста устройства, то, что бы мы ни выводили потом в данный контекст, все, что лежит за пределами региона отсечения, игнорируется.

Соответственно, чтобы сделать такую надпись, нужно создать регион, совпадающий по форме с этой надписью. В GDI есть целый ряд функций для создания регионов различной формы, но вот для создания региона в форме букв функции нет. Зато GDI поддерживает другие объекты — траектории. Строго говоря, это не совсем объекты: траектория не имеет дескриптора (по крайней мере, API не предоставляет этот дескриптор программам), и в каждом контексте устройства может быть только одна траектория. Создание траектории начинается с вызова функции `BeginPath`, заканчивается вызовом функции `EndPath`. Графические функции, вызванные между `BeginPath` и `EndPath`, не выводят ничего в контекст устройства, а то, что должно быть выведено, вместо этого запоминается в траектории (которая представляет собой совокупность замкнутых кривых). С траекторией можно выполнить много полезных операций (см., например, *разд. 1.3.4*). В нашем случае между вызовами `BeginPath` и `EndPath` мы вызываем `DrawText`, формируя таким образом траекторию, состоящую из контуров букв. Затем с помощью функции `PathToRegion` мы создаем регион, границы которого совпадают с контурами траектории, т. е., в данном случае, регион, совпадающий по форме с надписью.

Примечание

На самом деле не все графические функции, вызванные между `BeginPath` и `EndPath`, добавляют контуры к траектории. Это зависит от версии операционной системы. Подробнее этот вопрос обсуждается в *разд. 1.3.4*.

В ходе работы программы регион не меняется, так что нет нужды создавать его каждый раз при обработке события `OnPaint`. Он создается только один раз, и его дескриптор сохраняется в поле `FRgn` формы для дальнейшего использования.

Все, что осталось сделать, — это установить регион отсечения с помощью функции `SelectClipRgn`, отобразить рисунок и убрать регион отсечения, чтобы не мешал в дальнейшем.

Теперь рассмотрим, как рисуются звезды в правом верхнем углу окна (листинг 1.35).

Листинг 1.35. Рисование звезд

```
var
  I: Integer;
```

```
  Star: array[0..4] of TPoint;
...
// Следующая группа команд рисует две звезды справа от
// надписи. Эти звезды демонстрируют использование двух
// режимов заливки: WINDING и ALTERNATE. Для простых
// фигур эти режимы дают одинаковые результаты, разница
// возникает только при закрашивании сложных фигур,
// имеющих самопересечения.
Canvas.Pen.Style := psSolid;
Canvas.Pen.Width := 1;
Canvas.Pen.Color := clRed;
Canvas.Brush.Style := bsSolid;
Canvas.Brush.Color := clRed;
// Вычисляем координаты вершин звезды. Они помещаются
// в массив Star в следующем порядке (если первой
// считать верхнюю вершину и нумеровать остальные по
// часовой стрелке от нее): 1-3-5-2-4
for I := 0 to 4 do
begin
  Star[I].X := Round(380 + 90 * Sin(0.8 * I * Pi));
  Star[I].Y := Round(100 – 90 * Cos(0.8 * I * Pi));
end;
// Устанавливаем режим заливки WINDING. При
// использовании этого режима закрашивается все
// содержимое многоугольника независимо от того,
// как именно он нарисован.
SetPolyFillMode(Canvas.Handle, WINDING);
Canvas.Polygon(Star);
// Сдвигаем координаты звезды, чтобы нарисовать ее
// правее с другим режимом заливки.
for I := 0 to 4 do
  Inc(Star[I].X, 200);
// Устанавливаем режим заливки ALTERNATE. При
// использовании этого режима заполняются горизонтальные
// линии, лежащие между нечетной и четной сторонами
// многоугольника. В результате пятиугольник в центре
// звезды оказывается незаполненным.
SetPolyFillMode(Canvas.Handle, ALTERNATE);
Canvas.Polygon(Star);
```

Самое интересное здесь то, что обе звезды рисуются практически одинаково, меняется только *режим заливки*. Сначала с помощью простейшей тригонометрии вычисляются координаты вершин звезды, помещаются в массив `Star`,

и эта звезда рисуется с режимом заливки WINDING. При этом закрашиваются все точки, для которых выполняется условие, что луч, выпущенный из этой точки, пересекает контур многоугольника нечетное число раз, т. е. всю внутренность контура. Затем координаты вершин звезды смещаются вправо, и такая же звезда рисуется точно так же, но уже с режимом заливки ALTERNATE. В этом режиме закрашиваются только те точки, которые оказались между четной и нечетной сторонами многоугольника, и пятиугольник внутри звезды остается незакрашенным. Обратите внимание, что звезду мы здесь рисуем с помощью класса TCanvas, и только режимы заливки переключаем API-функциями.

Следующий шаг — это рисование черной прямоугольной рамки на фоне пересекающихся зеленых линий. Линии рисуются до рамки для того, чтобы показать, что центр рамки действительно остается прозрачным, а не заливается цветом фона. Сама рамка рисуется вызовом одной функции PolyPolygon, позволяющей за один раз нарисовать фигуру, ограниченную несколькими замкнутыми многоугольными контурами (листинг 1.36).

Листинг 1.36. Рисование рамки с использованием PolyPolygon

```
const
  Pts: array[0..7] of TPoint = ((X: 40; Y: 230), (X: 130; Y: 230),
                                (X: 130; Y: 320), (X: 40; Y: 320),
                                (X: 60; Y: 250), (X: 60; Y: 300),
                                (X: 110; Y: 300), (X: 110; Y: 250));
  Cnt: array[0..1] of Integer = (4, 4);
...
// Следующая группа команд рисует прямоугольную рамку
Canvas.Pen.Color := clLime;
Canvas.Pen.Width := 3;
// Эти линии рисуются для того, чтобы показать, что
// центр рамки остается прозрачным.
Canvas.MoveTo(30, 220);
Canvas.LineTo(140, 330);
Canvas.MoveTo(140, 220);
Canvas.LineTo(30, 330);
Canvas.Pen.Color := clBlack;
Canvas.Brush.Color := clBlack;
// Функция PolyPolygon позволяет нарисовать несколько
// многоугольников одной командой. Второй параметр
// задает координаты всех многоугольников, третий
// параметр задает массив, содержащий число вершин
// каждого из многоугольников. В нашем случае массив
```

```
// Cnt имеет значение (4, 4). Это значит, что первые
// четыре элемента массива Pts задают координаты первого
// многоугольника, следующие четыре — второго. Отметим,
// что указатели на массивы приходится передавать не
// очень простым способом: сначала нужно получить
// указатель на массив с помощью оператора @, а потом
// этот указатель разыменовать. Формальные параметры,
// определяющие указатели на массив, при импорте функции
// PolyPolygon в модуле Windows.dcu объявлены как
// нетипизированные параметры-переменные, поэтому
// компилятор не разрещает просто передать Pts и Cnt в
// качестве фактических параметров — он запрещает
// использовать константы там, где требуются переменные.
// Это не совсем корректно, т. к. локальная
// типизированная константа — это на самом деле не
// константа, а глобальная переменная с локальной
// областью видимости. Тем не менее компилятор имеет
// такую особенность, которую приходится учитывать.
// В данном примере функция PolyPolygon используется для
// рисования двух квадратов, один из которых целиком
// лежит внутри другого. При этом содержимое внутреннего
// квадрата остается незаполненным. Обратите внимание,
// что квадраты рисуются в разных направлениях: внешний
// по часовой стрелке, внутренний — против. Если
// установлен режим заполнения ALTERNATE, это никак не
// влияет на результат, но если установить режим WINDING,
// внутренний квадрат не будет закрашен только в том
// случае, если квадраты рисуются в противоположных
// направлениях.
PolyPolygon(Canvas.Handle, (@Pts)^, (@Cnt)^, 2);
```

Вся хитрость в этом коде — как передать параметры в функцию `PolyPolygon`. Ее второй параметр — это указатель на массив элементов `TPoint`, содержащий координаты вершин всех контуров в массиве: сначала все вершины первого контура в нужном порядке, затем — все вершины второго контура и т. д. Третий параметр — это указатель на массив, содержащий число точек в каждом контуре: первый элемент массива содержит число точек в первом контуре, второй — во втором и т. д. Общее число контуров определяется четвертым, последним параметром функции `PolyPolygon`. Число элементов во втором массиве должно быть равно значению четвертого параметра, а число элементов в первом массиве — сумме значений элементов второго массива. За выполнением этих требований должен следить сам программист, если он ошибется, функция может обратиться к памяти, лежащей за пределами массивов, и последствия будут непредсказуемыми.

В оригинале параметры-массивы функции PolyPolygon объявлены как указатели на типы элементов массива. В модуле Windows при импорте этой функции, как это часто бывает в подобных случаях, эти параметры стали нетипизированными параметрами-переменными. В нашем случае массивы объявлены как локальные типизированные константы. По сути, в этом случае они являются глобальными переменными с локальной областью видимости, т. е., как обычные глобальные переменные, хранятся в сегменте данных и существуют на протяжении всего времени работы программы, но компилятор разрешает использовать их только внутри той процедуры, в которой они объявлены. И, несмотря на то, что по сути такие "константы" являются переменными, компилятор их рассматривает как константы и запрещает подставлять там, где требуются параметры-переменные. Поэтому приходится "обманывать" компилятор, получая указатель на эти константы, а затем разыменовывая его. Если бы наши массивы хранились в обычных переменных, нужды прибегать к такому приему не было бы.

Нетрудно убедиться, что первые четыре элемента массива Pts содержат координаты вершин внешнего квадрата рамки, последние четыре — внутреннего квадрата. Массив Cnt, соответственно, содержит два элемента, оба из которых имеют значение 4. Это означает, что в нашей фигуре два замкнутых контура, и оба содержат по четыре вершины. Порядок следования вершин во внешнем квадрате — по часовой стрелке, во внутреннем — против. Это имеет значение, если выбран режим заливки WINDING, тогда направления обхода контуров должны быть противоположными, иначе отверстие тоже окажется закрашенным. Для режима заливки ALTERNATE направление обхода контуров не имеет значения.

Далее программа GDIDraw демонстрирует работу функции InvertRect, которая инвертирует цвета в заданной прямоугольной области контекста устройства. Для того чтобы это было нагляднее, мы снова выведем на форму загруженный в OnCreate рисунок (только на этот раз без региона отсечения) и инвертируем область, частично пересекающуюся с областью рисунка (листинг 1.37).

Листинг 1.37. Пример использования функции InvertRect

```
// Следующая группа команд выводит рисунок и инвертирует
// его часть
Canvas.Draw(300, 220, FBitmap);
// Функция InverRect делает указанный прямоугольник
// "негативом".
InvertRect(Canvas.Handle, Rect(320, 240, 620, 340));
```

Еще одна забавная функция GDI, которая почему-то не нашла отражения в классе TCanvas — это GrayString. Она предназначена для вывода "серого" текста, т. е. текста, который по яркости находится посредине между черным и белым. Обычно для этого просто устанавливается цвет RGB(128, 128, 128), но некоторые черно-белые устройства не поддерживают полутона (это касается, прежде всего, старых моделей принтеров) — именно на них и ориентирована функция GrayString. Она позволяет рисовать серый текст произвольным образом с помощью функции обратного вызова, но эту функцию можно не указывать, и тогда рисование осуществляется функцией TextOut. Но при этом текст выводится через промежуточную растровую картинку в памяти, что обеспечивает полупрозрачность текста, т. к. закрашиваются не все пикселы, а только половина в шахматном порядке. На черно-белых принтерах с большим разрешением это действительно выглядит как серый текст, на экране же можно получать "полупрозрачные" надписи. Пример использования функции GrayString приведен в листинге 1.38.

Листинг 1.38. Пример использования функции GrayString

```
// Следующая группа команд выводит "полупрозрачную"
// надпись "Windows API"
Canvas.Brush.Color := clBlue;
// Функция GrayString при выводе текста закрашивает
// заданной кистью не все пикселы подряд, а в шахматном
// порядке, оставляя остальные нетронутыми. Это создает
// эффект полупрозрачности.
Canvas.Font.Name := 'Times New Roman';
Canvas.Font.Style := [fsBold];
Canvas.Font.Height := 69;
GrayString(Canvas.Handle, Canvas.Brush.Handle, nil,
  LPARAM(PChar('Windows API')), 0, 20, 350, 0, 0);
```

Обратите внимание на второй параметр — через него передается дескриптор кисти, с помощью которой будет осуществляться закраска пикселов в выводимой строке. Функция GrayString игнорирует ту кисть, которая выбрана в контексте устройства и использует свою. Здесь для простоты мы передаем ей кисть контекста устройства, но, в принципе, это могла бы быть любая другая кисть. Третий параметр — это указатель на функцию обратного вызова. В нашем случае он равен nil, что указывает на использование функции TextOut. Четвертый параметр имеет тип LPARAM и содержит произвольные данные, которые передаются функции обратного вызова. В случае использования TextOut это интерпретируется как указатель на строку, которую нужно вывести, поэтому здесь приходится возиться с приведением типов. Пятый

параметр содержит длину выводимой строки. Это очень характерно для функций GDI, предназначенных для вывода текста: конец строки в них определяется не обязательно по завершающему символу #0, можно вывести только часть строки, явно задав нужное число символов. Но этот же параметр можно сделать равным нулю (как в нашем случае), и тогда длина строки определяется обычным образом — по символу #0. Прочие параметры функции определяют координаты выводимой строки.

Последняя часть примера посвящена вопросу, который долгое время был очень популярен в форумах: как вывести текст, расположенный наклонно (в программе примером такого текста является надпись **Sample**, выведенная под углом 60 градусов). Это связано с тем, что только в BDS 2006 у класса TFont появилось свойство Orientation, позволяющее задавать направление текста (в справке BDS 2006 информация об этом свойстве отсутствует, она появляется только в справке Delphi 2007, но это свойство, тем не менее, есть и в BDS 2006, а также в Turbo Delphi). В более ранних версиях текст под углом можно было вывести только с помощью функций GDI, вручную создавая шрифт (листинг 1.9).

Листинг 1.39. Вывод текста под углом средствами GDI

```
// Следующая группа функций выводит надпись "Sample",
// повернутую на угол 60 градусов.
Canvas.Brush.Style := bsClear;
// При создании логического шрифта для контекста
// устройства следует в обязательном порядке указывать
// угол поворота. Однако класс TFont игнорирует такую
// возможность, поэтому шрифт нужно создавать вручную.
// Чтобы выбрать шрифт в контексте устройства, легче
// всего присвоить его дескриптор свойству
// Canvas.Font.Handle. Параметры fdwItalic, fdwUnderline
// и fdwStrikeOut, согласно справке, могут принимать
// значения True или False, но имеют тип DWORD. Для
// C/C++ это не имеет значения — целый и логический типы
// в этих языках совместимы. Но в Delphi приходится
// использовать 0 и 1 вместо True и False. Угол поворота
// шрифта задается в десятых долях градуса, т. е.
// значение 600 означает 60 градусов.
Canvas.Font.Handle := CreateFont(60, 0, 600, 600, FW_NORMAL, 0, 0, 0,
  ANSI_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
  DEFAULT_PITCH, 'Times New Roman');
Canvas.TextOut(140, 320, 'Sample');
// Эта строка нужна для того, чтобы пример работал
// корректно в BDS 2006 и выше. В этой версии у класса
```

```
// TFont появилось свойство Orientation, задающее
// направление текста, и этот класс научился определять
// и сохранять это направление даже в том случае, если
// оно было задано функцией GDI, а не через свойство
// Orientation. Чтобы этого не происходило, нужно снова
// придать шрифту горизонтальное направление. В версиях
// Delphi, более ранних, чем BDS 2006, эта строка
// не нужна: при изменении шрифта через класс TFont
// направление текста и так станет горизонтальным.
Canvas.Font.Handle := CreateFont(60, 0, 0, 0, FW_NORMAL, 0, 0, 0,
  ANSI_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
  DEFAULT_PITCH, 'Times New Roman');
```

Новый шрифт создается функцией `CreateFont`. Если бы мы программировали без VCL, то полученный в результате вызова этой функции дескриптор шрифта необходимо было бы выбрать в контексте устройства (функция `SelectObject`) и вывести надпись. Затем в устройстве следовало бы выбрать другой шрифт, а созданный ранее удалить. Но т. к. VCL мы все же используем, можно поступить проще: присвоить созданный дескриптор свойству `Canvas.Font.Handle`, а все остальное сделают классы `TCanvas` и `TFont`.

Примечание

Вообще говоря, при использовании GDI нет нужды каждый раз заново создавать шрифт или любой другой объект, когда они понадобятся. Создать их можно один раз, а затем указать в программе сохраненный дескриптор везде, где это необходимо.

Функция `CreateFont` имеет 14 параметров, определяющих свойства создаваемого шрифта. Мы не будем перечислять их все, отметим только, что мы здесь создаем шрифт на основе гарнитуры Times New Roman, имеющий размер 60, обычный (т. е. не жирный и не наклонный). О точных значениях всех параметров рекомендуем посмотреть в MSDN.

Самые интересные для нас параметры — это третий (`nEscapement`) и четвертый (`nOrientation`), которые и определяют угол наклона шрифта. Они задаются в десятых долях градуса, т. е., чтобы получить нужное значение параметра, следует требуемое число градусов умножить на 10 (в нашем примере оба эти параметра равны 600, что означает 60 градусов). Параметр `nEscapement` задает угол поворота базовой линии текста относительно горизонтальной оси. Параметр `nOrientation` задает угол поворота отдельных букв относительно своего нормального положения. По умолчанию в контексте устройства включен режим `GM_COMPATIBLE`, при котором эти два значения должны совпадать, т. е. угол поворота надписи в целом и угол поворота отдельной буквы всегда совпадают. В Windows NT/2000/XP с помощью функ-

ции `SetGraphicsMode` можно установить для контекста устройства режим `GM_ADVANCED`, при котором, в частности, параметры `nOrientation` и `nEscapement` могут принимать различные значения (в Windows 9*x*/ME тоже есть функция `SetGraphicsMode`, но установить режим `GM_ADVANCED` она не позволяет).

Когда мы присваиваем значение свойству `TFont.Handle`, все прочие свойства объекта `TFont` меняют свои значения в соответствии с тем, какой шрифт установлен. Так как в Delphi до 7-й версии свойство `TFont.Orientation` отсутствует, направление шрифта, установленное нами, в этом классе не запоминается, и поэтому при дальнейшем изменении шрифта с помощью свойств `Canvas.Font.Name`, `Canvas.Font.Size` и т. п. мы снова получим горизонтальный шрифт. Другое дело — BDS 2006 и выше. В этих версиях направление шрифта тоже запоминается, и поэтому дальнейшие манипуляции со свойствами `Canvas.Font` будут снова давать наклонный шрифт, пока мы явно не присвоим значение 0 свойству `Canvas.Font.Orientation`. В нашем случае это означает, что при повторном вызове события `OnPaint` при вызове функции `GrayString` будет выведен наклонный текст, если не принять дополнительные меры. Как мы уже сказали, проблема легко решается присваиванием нуля свойству `Canvas.Font.Orientation`, но, т. к. наши примеры должны работать во всех версиях Delphi, начиная с пятой, этот вариант нам не подходит. Поэтому мы здесь вновь вручную создаем шрифт, на этот раз не важно, какой именно, главное, чтобы его параметры `nOrientation` и `nEscapement` были равны нулю. В Delphi до 7-й версии программа GDIDraw будет корректно работать и без второго вызова функции `CreateFont`.

Отметим, что во всех версиях до Delphi 2007 как минимум класс `TFont` имеет свойство `Orientation`, но не имеет свойства `Escapement`. Это означает, что если вы хотите вывести надпись, у которой угол наклона букв и угол наклона базовой линии будут разными, вам все-таки придется самостоятельно вызывать функцию `CreateFont`.

1.2.8. Пример BitmapSpeed

Программа BitmapSpeed предназначена для сравнения скорости работы с растровыми изображениями в формате DDB и DIB через класс `TBitmap`. Тестируются три операции: рисование прямых линий, вывод растра на экран и работа со свойством `ScanLine`. Окно программы показано на рис. 1.12.

Одна отдельно взятая операция выполняется настолько быстро, что измерить время ее выполнения можно только с большой погрешностью. Чтобы уменьшить погрешность, нужно повторить операцию много раз и измерить общее время. Все три теста выполняются методом `DoTest`, показанном в листинге 1.40.

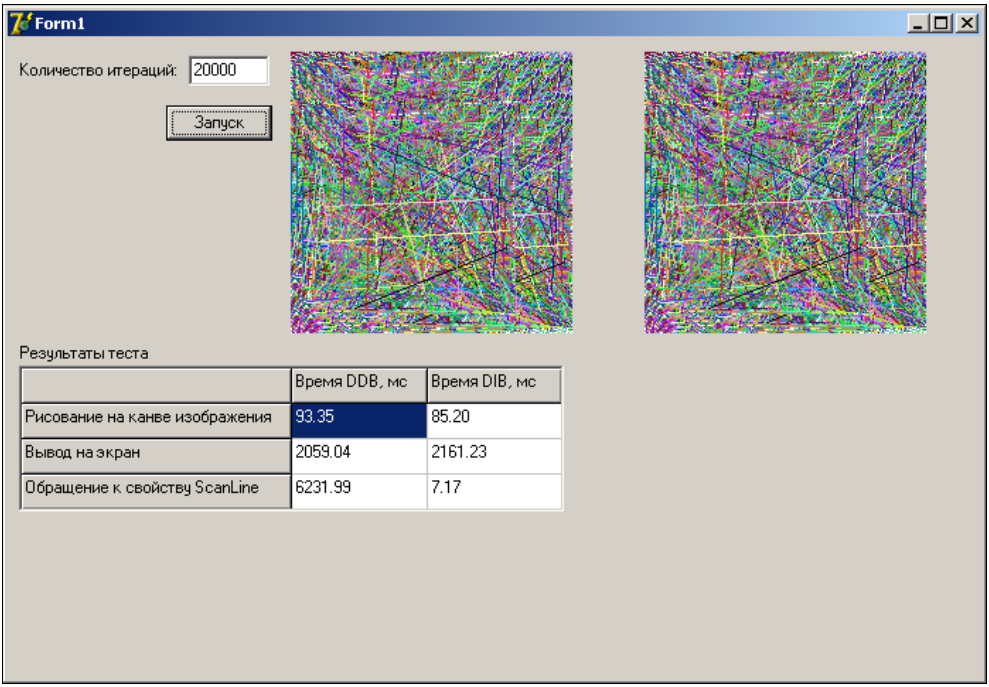


Рис. 1.12. Окно программы BitmapSpeed после завершения теста

Листинг 1.40. Метод DoTest, выполняющий тесты скорости

```
procedure TForm1.DoTest(Cnt, XOfs, ColNum: Integer;
  PixelFormat: TPixelFormat);
{ Cnt        — число повторов операции при тестах
  XOfs       — X-координата области, в которой будет
               выполняться вывод изображения во втором тесте
  ColNum     — номер колонки в GridResults, в которую будут
               выводиться результаты
  PixelFormat — формат изображения }
var
  Pict: TBitmap;
  I: Integer;
  P: Pointer;
  Freq, StartTime, EndTime: Int64;
begin
  // Узнаем частоту условного счетчика тактов
  QueryPerformanceFrequency(Freq);
  // Создаем изображение
  Pict := TBitmap.Create;
```

```
try
  Pict.PixelFormat := PixelFormat;
  Pict.Width := PictSize;
  Pict.Height := PictSize;
  Pict.Canvas.Pen.Width := 0;
  // Вывод линий на картинку
  // Выводится Cnt линий со случайными координатами
  QueryPerformanceCounter(StartTime);
  for I := 1 to Cnt do
  begin
    Pict.Canvas.Pen.Color :=
      RGB(Random(256), Random(256), Random(256));
    Pict.Canvas.MoveTo(Random(PictSize), Random(PictSize));
    Pict.Canvas.LineTo(Random(PictSize), Random(PictSize));
  end;
  QueryPerformanceCounter(EndTime);
  GridResults.Cells[ColNum, 1] :=
    FloatToStrF((EndTime – StartTime) / Freq * 1000, ffFixed, 10, 2);
  // Вызываем Application.ProcessMessages, чтобы GridResults
  // перерисовался в соответствии с новым значением ячейки
  Application.ProcessMessages;
  // Второй тест — вывод рисунка на экран
  QueryPerformanceCounter(StartTime);
  // Повторяем вывод рисунка на экран Cnt раз
  // Чтобы пользователь мог видеть, когда вывод
  // заканчивается, каждый раз прибавляем к координатам
  // случайную величину
  for I := 1 to Cnt do
    Canvas.Draw(XOfs + Random(50), 10 + Random(50), Pict);
  QueryPerformanceCounter(EndTime);
  GridResults.Cells[ColNum, 2] :=
    FloatToStrF((EndTime – StartTime) / Freq * 1000, ffFixed, 10, 2);
  Application.ProcessMessages;
  // Третий тест — доступ к свойству ScanLine
  QueryPerformanceCounter(StartTime);
  // Обращаемся к случайной строке свойства ScanLine
  // Cnt раз
  for I := 1 to Cnt do
    P := Pict.ScanLine[Random(PictSize)];
  QueryPerformanceCounter(EndTime);
  GridResults.Cells[ColNum, 3] :=
    FloatToStrF((EndTime – StartTime) / Freq * 1000, ffFixed, 10, 2);
  Application.ProcessMessages;
```

```
  finally
    Pict.Free;
  end;
end;
```

Для измерения скорости работы будем использовать счетчик производительности — это высокопроизводительный счетчик, поддерживаемый системой для измерения производительности. Текущее значение счетчика можно узнать с помощью функции `QueryPerformanceCounter`, число тактов счетчика в секунду — с помощью функции `QueryPerformanceFrequency`. Этот счетчик позволяет получить более точные результаты, чем традиционно применяющаяся для таких целей функция `GetTickCount`. Теоретически, счетчик производительности может не поддерживаться аппаратной частью (в этом случае функция `QueryPerformanceFrequency` вернет нулевую частоту), однако все современные компьютеры такой счетчик поддерживают, поэтому его можно применять без опасений.

В зависимости от параметра `PixelFormat` метод `DoTest` создает DDB- или DIB-изображение и тестирует скорость исполнения операций с ним. В первом тесте `Cnt` раз рисуется линия случайного цвета со случайными координатами — так проверяется скорость рисования на картинке. Разумеется, это весьма односторонний тест, т. к. при рисовании других примитивов будет, скорее всего, иное соотношение скоростей для DIB и DDB. Но общее представление о соотношении скоростей он все же дает.

Во втором тесте полученное изображение `Cnt` раз выводится на экран. Если бы оно выводилось всегда в одном и том же месте, пользователь не видел бы процесс вывода на экран, т. к. каждый следующий раз картинка рисовалась бы точно в том же месте, что и в предыдущий, и общее изображение не менялось бы. Чтобы этого не происходило, изображение выводится со случайным смещением относительно базовых координат, и пользователь может наблюдать за процессом. Кроме того, координаты определяются также параметром `XOfs` — это сделано для того, чтобы при тестировании DDB- и DIB-изображений рисунки выводились в разных частях окна и не накладывались друг на друга.

На некоторых компьютерах в этом тесте с DDB-изображением наблюдается интересный эффект: время, измеренное программой, заметно меньше, чем время, когда картинка меняется на экране (например, пользователь ясно видит, что тест выполняется в течение примерно трех секунд, в то время как программа дает значение около одной секунды). Это связано со способностью некоторых видеокарт буферизовать переданные им команды и выполнять их асинхронно, т. е. вызов функции завершается очень быстро, программа продолжает работать дальше, а видеокарта параллельно ей выполняет

команду. Если вы столкнетесь с такой ситуацией, можете провести небольшой эксперимент: вставить вызов функции Beep сразу после окончания второго теста. Вы услышите звуковой сигнал раньше, чем изображение закончит меняться.

Третий тест самый простой: Cnt раз значение свойства ScanLine присваивается переменной P. Так как значение P потом нигде не используется, компилятор выдает соответствующую подсказку, но в данном случае ее можно игнорировать.

Таким образом, метод DoTest нужно вызвать два раза: для DDB-изображения и для DIB это делает обработчик нажатия кнопки BtnStart (листинг 1.41).

Листинг 1.41. Обработчик нажатия кнопки BtnStart

```
procedure TForm1.BtnStartClick(Sender: TObject);
var
  IterCnt, RandomStart: Integer;
begin
  IterCnt := StrToInt(EditIter.Text);
  GridResults.Cells[1, 1] := '';
  GridResults.Cells[1, 2] := '';
  GridResults.Cells[1, 3] := '';
  GridResults.Cells[2, 1] := '';
  GridResults.Cells[2, 2] := '';
  GridResults.Cells[2, 3] := '';
  // Чтобы новый текст ячеек отобразился в GridResults,
  // нужно, чтобы было извлечено из очереди и обработано
  // сообщение WM_PAINT. Чтобы сделать это немедленно,
  // вызываем Application.ProcessMessages.
  Application.ProcessMessages;
  RandomStart := Random(MaxInt);
  Screen.Cursor := crHourGlass;
  // Точное измерение времени выполнения кода в Windows
  // невозможно, потому что это многозадачная система, и
  // часть измеренного времени может быть потрачена на
  // выполнение кода других процессов. Чтобы максимально
  // уменьшить погрешность измерения, нужно установить
  // наивысший приоритет процессу и его главной нити -
  // тогда вероятность переключения процессора на
  // выполнение другой задачи будет минимальна. Столь
  // высокий приоритет приводит к тому, что во время
  // выполнения теста система перестает реагировать на
  // перемещение нити. Поэтому необходимо использовать блок
```

```
  // try/finally, чтобы даже при возникновении исключения
  // приоритет процесса и нити был снижен до нормального
  // уровня.
  SetThreadPriority(GetCurrentThread, THREAD_PRIORITY_TIME_CRITICAL);
  SetPriorityClass(GetCurrentProcess, REALTIME_PRIORITY_CLASS);
  try
    // В тестах активно используются псевдослучайные числа.
    // Чтобы сравнение было корректно, нужно, чтобы
    // последовательности чисел в экспериментах с DIB и DDB
    // были одинаковыми. Каждое следующее псевдослучайное
    // число генерируется на основе значения глобальной
    // переменной модуля System RandSeed. Значение RandSeed
    // при этом обновляется по определенному закону. Таким
    // образом, если установить определенное значение
    // RandSeed, то последовательность псевдослучайных чисел
    // будет строго детерминирована. Это свойство генератора
    // случайных чисел используется, чтобы в обоих
    // экспериментах были одинаковые последовательности.
    RandSeed := RandomStart;
    DoTest(IterCnt, 200, 1, pfDevice);
    RandSeed := RandomStart;
    DoTest(IterCnt, 450, 2, pf24bit);
  finally
    Screen.Cursor := crDefault;
    SetThreadPriority(GetCurrentThread, THREAD_PRIORITY_NORMAL);
    SetPriorityClass(GetCurrentProcess, NORMAL_PRIORITY_CLASS);
  end;
end;
```

Все три теста используют случайные числа. Чтобы условия были одинаковыми, нужно обеспечить идентичность последовательностей случайных чисел при тестировании DDB- и DIB-изображений. К счастью, этого легко добиться, установив перед тестированием одинаковые значения переменной `RandSeed` модуля `System`, которая и определяет последующее случайное число. Начальное значение `RandSeed` также выбирается случайным образом, а т. к. в обработчике события `OnCreate` формы есть вызов `Randomize`, при каждом запуске будет сгенерирована новая последовательность случайных чисел. Это одна из причин того, что результаты тестов будут меняться от запуска к запуску.

Вторая причина заключается в том, что Windows — это система с вытесняющей многозадачностью, и ни одна программа не может даже на короткое время захватить процессор для монопольного использования. Пока выполняются тесты, Windows может время от времени переключаться на выполнение

других операций, внося тем самым погрешность в результаты измерений времени выполнения тестов. Чтобы уменьшить эту погрешность до минимума, перед выполнением тестов мы назначаем своему процессу и его главной нити максимальный приоритет, чтобы минимизировать число ситуаций, когда система может отобрать квант времени у теста. Тем не менее полностью исключить такую возможность нельзя, поэтому результаты имеют некоторую степень условности.

Что касается самих результатов, то они, конечно, сильно зависят от конфигурации компьютера. По первым двум тестам время выполнения для DDB-растра может быть как в два-три раза меньше, чем для DIB, так и несколько превышать его. В третьем тесте DIB-растр, разумеется, существенно опережает по скорости DDB, хотя отношение и здесь зависит от компьютера.

Также наблюдается некоторая зависимость от версии Delphi, под которой откомпилирован проект. Например, первый тест и для DIB, и для DDB выполняется несколько быстрее под Delphi 2007, чем под Delphi 5, а вот третий тест под Delphi 2007 выполняется несколько медленнее.

1.3. Обобщающие примеры

Рассмотрев основы работы с функциями API, мы переходим к обобщающим примерам — программам, использующим разные средства API для создания простого законченного примера.

1.3.1. Обобщающий пример 1 — Информация о процессах

Первым обобщающим примером станет программа для получения информации о процессах системы и об окнах, которые они открывают. На компакт-диске, прилагаемом к книге, эта программа называется ProcInfo. Окно программы ProcInfo показано на рис. 1.13.

1.3.1.1. Получение списка процессов

Исторически сложилось так, что существует два способа получить список процессов: с помощью функций *Tool Help* и посредством функций *PSAPI*. Эти две группы функций использовались в разных линиях Windows: функции Tool Help появились в Windows 95, функции PSAPI — в Windows NT 4. Windows 2000/XP также поддерживают функции Tool Help, в то время как Windows 98/ME не поддерживают PSAPI. Поэтому мы выберем функции Tool Help, что даст нашему примеру возможность работать во всех версиях Windows, кроме NT 4 (впрочем, в Windows 95 пример тоже не будет работать,

но по другой причине: из-за функций `GetWindowInfo` и `RealGetWindowClass`, отсутствующих в этой версии). Функции Tool Help объявлены в модуле `TlHelp32`.

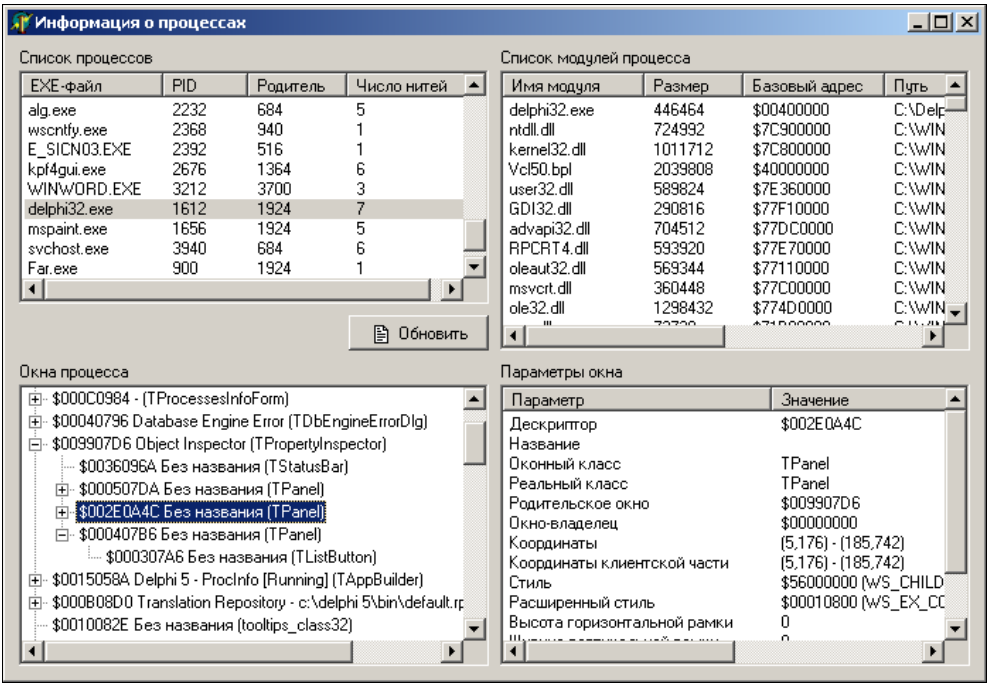


Рис. 1.13. Окно программы ProcInfo

Для получения списка процессов необходимо сделать "снимок" состояния системы с помощью функции `CreateToolhelp32Snapshot`. Эта функция создает специальный объект, который может хранить информацию о процессах, модулях, нитях и кучах, созданных в системе. Этот объект называется снимком потому, что информация, хранящаяся в нем, актуальна на момент вызова функции `CreateToolhelp32Snapshot`; дальнейшие изменения списка процессов, модулей и т. п. не приводят к изменению снимка. Доступ к снимку, как и к большинству объектов системы, осуществляется через его дескриптор. В данном случае функция `CreateToolhelp32Snapshot` вызывается с параметром `TH32CS_SNAPPROCESS` для получения списка процессов.

Навигация по списку процессов, сохраненных в снимке, осуществляется с помощью функций `Process32First` и `Process32Next`. Они позволяют получить ряд параметров процесса, главный среди которых — *идентификатор процесса* (Process Identifier, PID). Это уникальный идентификатор процесса, с помощью которого можно отличать один процесс от другого.

Примечание

Не следует путать идентификатор процесса и дескриптор объекта процесса, который используется, например, в функции SetPriorityClass. Объект процесса — это специальный объект, связанный с процессом, но не тождественный ему. В частности, объект процесса может продолжать существовать уже после того, как сам процесс завершит работу (это позволяет, например, корректно синхронизироваться с уже завершенным процессом при помощи функции WaitForSingleObject). Через объект процесса можно управлять многими свойствами процесса. Поучить дескриптор объекта процесса по идентификатору процесса можно с помощью функции OpenProcess.

Код для получения списка процессов показан в листинге 1.42.

Листинг 1.42. Получение списка процессов с помощью Tool Help

```
procedure TProcessesInfoForm.FillProcessList;
var
  SnapProc: THandle;
  ProcEntry: TProcessEntry32;
  Item: TListItem;
begin
  ClearAll;
  // Создаем снимок, в котором сохраняем все процессы, а
  // затем в цикле получаем информацию о каждом из этих
  // процессов, перенося ее в ListProcesses
  SnapProc := CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if SnapProc <> INVALID_HANDLE_VALUE then
  try
    ProcEntry.dwSize := SizeOf(TProcessEntry32);
    if Process32First(SnapProc,ProcEntry) then
      repeat
        Item := ListProcesses.Items.Add;
        Item.Caption := ProcEntry.szExeFile;
        Item.SubItems.Add(IntToStr(ProcEntry.th32ProcessID));
        Item.SubItems.Add(IntToStr(ProcEntry.th32ParentProcessID));
        Item.SubItems.Add(IntToStr(ProcEntry.cntThreads));
        // Сохраняем PID в поле Data соответствующего
        // элемента списка. Вообще, поле Data имеет тип
        // Pointer, а PID — это целое число, но т. к. оба этих
        // типа 32-битные, их можно приводить друг к другу
        Item.Data := Pointer(ProcEntry.th32ProcessID);
      until not Process32Next(SnapProc, ProcEntry);
  finally
    CloseHandle(SnapProc);
  end
```

```
  else
  begin
    ListProcesses.Visible := False;
    LabelProcessError.Caption :=
      'Невозможно получить список процессов:'#13#10'Ошибка №' +
      IntToStr(GetLastError);
  end;
end;
```

Для получения списка модулей данного процесса также используется снимок. Функция `CreateToolhelp32Snapshot` вызывается с параметром `TH32CS_SNAPMODULE`, в качестве второго параметра ей передается PID процесса, модули которого требуется получить. Навигация по снимку модулей осуществляется с помощью функций `Module32First` и `Module32Next`. В остальном код получения списка модулей совпадает с кодом, приведенным в листинге 1.42.

1.3.1.2. Получение списка и свойств окон

Список окон, созданных процессом, формируется с помощью функции `EnumWindows`, которая позволяет получить список всех окон верхнего уровня (т. е. расположенных непосредственно на рабочем столе). Для каждого из этих окон с помощью функции `GetWindowThreadProcessID` определяется идентификатор процесса. Окна, не принадлежащие выбранному процессу, отсеиваются.

Для каждого из окон верхнего уровня, принадлежащих процессу, с помощью функции `EnumChildWindows` ищутся дочерние окна, а для каждого из найденных таким образом дочерних окон — его дочерние окна. Здесь следует учесть, что `EnumChildWindows` возвращает не только дочерние окна заданного окна, но и все окна, которыми владеют эти дочерние окна. Чтобы в дереве окон не было дублирования, при построении очередного уровня дерева окон отбрасываются все окна, непосредственным родителем которых не является данное окно. Код, выполняющий построение дерева, приведен в листинге 1.43.

Листинг 1.43. Получение всех окон, относящихся к данному процессу

```
function EnumWindowsProc(Wnd: HWnd; ParentNode: TTreeNode): BOOL;
  stdcall;
var
  Text: string;
  TextLen: Integer;
  ClassName: array[0..ClassNameLen – 1] of Char;
```

```
  Node: TTreeNode;
  NodeName: string;
begin
  Result := True;
  // Функция EnumChildWindows возвращает список
  // не только прямых потомков окна, но и потомков его
  // потомков, поэтому необходимо отсеять все те окна,
  // которые не являются прямыми потомками данного
  if Assigned(ParentNode) and
    (THandle(ParentNode.Data) <> GetAncestor(Wnd, GA_PARENT)) then
    Exit;
  TextLen := GetWindowTextLength(Wnd);
  SetLength(Text, TextLen);
  if TextLen > 0 then
    GetWindowText(Wnd, PChar(Text), TextLen + 1);
  if TextLen > 100 then
    Text := Copy(Text, 1, 100) + ' ...';
  GetClassName(Wnd, ClassName, ClassNameLen);
  ClassName[ClassNameLen – 1] := #0;
  if Text = '' then
    NodeName := 'Без названия (' + ClassName + ')'
  else
    NodeName := Text + ' (' + ClassName + ')';
  NodeName := '$' + IntToHex(Wnd, 8) + ' ' + NodeName;
  Node := ProcessesInfoForm.TreeWindows.Items.AddChild(ParentNode,
    NodeName);
  Node.Data := Pointer(Wnd);
  EnumChildWindows(Wnd, @EnumWindowsProc, LParam(Node));
end;

function EnumTopWindowsProc(Wnd: HWnd; PIDNeeded: Cardinal): BOOL;
  stdcall;
var
  Text: string;
  TextLen: Integer;
  ClassName: array[0..ClassNameLen – 1] of Char;
  Node: TTreeNode;
  NodeName: string;
  WndPID: Cardinal;
begin
  Result := True;
  // Здесь отсеиваются окна, которые не принадлежат
  // выбранному процессу
```

```
  GetWindowThreadProcessID(Wnd, @WndPID);
  if WndPID = PIDNeeded then
  begin
    TextLen := GetWindowTextLength(Wnd);
    SetLength(Text, TextLen);
    if TextLen > 0 then
      GetWindowText(Wnd, PChar(Text), TextLen + 1);
    if TextLen > 100 then
      Text := Copy(Text, 1, 100) + ' ...';
    GetClassName(Wnd, ClassName, ClassNameLen);
    ClassName[ClassNameLen – 1] := #0;
    if Text = '' then
      NodeName := 'Без названия (' + ClassName + ')'
    else
      NodeName := Text + ' (' + ClassName + ')';
    NodeName := '$' + IntToHex(Wnd, 8) + ' ' + NodeName;
    Node := ProcessesInfoForm.TreeWindows.Items.AddChild(nil,
      NodeName);
    Node.Data := Pointer(Wnd);
    EnumChildWindows(Wnd, @EnumWindowsProc, LParam(Node));
  end;
end;

procedure TProcessesInfoForm.FillWindowList(PID: Cardinal);
begin
  if PID = 0 then
    Exit;
  EnumWindows(@EnumTopWindowsProc, PID);
end;
```

В отличие от примера `EnumWnd` из *разд. 1.2.1* здесь для функций `EnumWindows` и `EnumChildWindows` предусмотрены разные процедуры обратного вызова. Связано это с тем, что окна верхнего уровня необходимо фильтровать по принадлежности к выбранному процессу, и параметр функции обратного вызова служит для передачи PID этого процесса. А их дочерние окна фильтровать по процессам не обязательно (мы предполагаем, что если некоторое окно верхнего уровня принадлежит данному процессу, то и все его дочерние окна также принадлежат этому процессу), зато нужно передавать указатель на элемент дерева, соответствующий родительскому окну, чтобы процедура знала, где размещать новый элемент. Таким образом, смысл параметра меняется, поэтому требуется новая процедура обратного вызова. (В принципе, можно было бы проверять, есть ли у найденного окна родитель, и в зависимости от этого трактовать параметр так или иначе, используя приведение типов. Но

это сильно усложняет код, поэтому в учебном примере мы не будем использовать такой способ.)

Примечание

Как уже было сказано ранее, мы полагаем, что если некоторое окно верхнего уровня принадлежит данному процессу, то и все его дочерние окна также принадлежат этому процессу. В общем случае это неверно: функция `CreateWindow(Ex)` позволяет при создании нового окна использовать в качестве родительского окно другого процесса. Поэтому наш код ошибочно отнесет подобные окна к тому процессу, к которому относятся их родители, а не к тому, который их реально создал. Здесь мы пренебрегаем такой возможностью, потому что для ее учета не нужны дополнительные знания API, необходимо просто запрограммировать более сложный алгоритм отсева. В учебном примере, посвященном API, реализация такого алгоритма была бы неоправданным усложнением, но в реальных программах эту возможность следует учесть.

Для получения названия окна в приведенном коде используется функция `GetWindowText`. Эта функция безопасна при работе с зависшими приложениями, поскольку она гарантирует, что вызвавшее ее приложение не зависнет само, пытаясь получить ответ от зависшего приложения. Но `GetWindowText` не всегда может получить названия элементов управления, расположенных в окнах чужих приложений (точнее, в MSDN написано, что она вообще не может получать названия элементов управления в чужих окнах, но практика показывает, что нередко `GetWindowText` все же делает это). Существует альтернативный способ получения названия окна — отправка ему сообщения `WM_GETTEXT`. При этом ограничений на работу с чужими элементами управления нет, но и гарантии, что из-за отсутствия ответа от чужого приложения программа не зависнет, тоже нет.

Использование `WM_GETTEXT` показано в другой части программы — при заполнении списка параметров окна, отображающегося в правом нижнем углу формы. Чтобы программа не зависала, вместо `SendMessage` для отправки `WM_GETTEXT` применяется `SendMessageTimeout`. Код получения имени показан в листинге 1.44.

Листинг 1.44. Получение заголовков "чужих" окон

```
if SendMessageTimeout(Wnd, WM_GETTEXTLENGTH, 0, 0,
  SMTO_NORMAL or SMTO_ABORTIFHUNG, 5000, TextLen) = 0 then
begin
  LastError := GetLastError;
  if LastError = 0 then
    Text := 'Приложение не отвечает'
  else
    Text := 'Ошибка при получении длины заголовка: ' +
      IntToStr(LastError);
end
```

```
else
begin
  SetLength(Text, TextLen);
  if TextLen > 0 then
    if SendMessageTimeout(Wnd, WM_GETTEXT, TextLen + 1, LParam(Text),
      SMTO_NORMAL or SMTO_ABORTIFHUNG, 5000, TextLen) = 0 then
    begin
      LastError := GetLastError;
      if LastError = 0 then
        Text := 'Приложение не отвечает'
      else
        Text := 'Ошибка при получении заголовка: ' +
          IntToStr(LastError);
    end;
end;
```

Для каждого окна программа выводит имя оконного класса и реальный класс окна. В большинстве случаев эти два класса совпадают. Они различаются только для тех окон, чьи классы "унаследованы" от стандартных классов, таких как EDIT, COMBOBOX и т. п.

Вообще, наследования оконных классов в Windows нет. Но существует один нехитрый прием, который позволяет имитировать наследование. Оконная процедура обычно не обрабатывает все сообщения, а передает часть их в одну из стандартных оконных процедур (DefWindowProc, DefFrameProc и т. п.). Программа может с помощью функции GetClassInfo узнать адрес оконной процедуры, назначенной стандартному классу, и использовать ее вместо стандартной оконной процедуры. Так как большая часть свойств окна определяется тем, как и какие сообщения оно обрабатывает, использование оконной процедуры другого класса позволяет почти полностью унаследовать свойства этого класса. (В VCL для наследования оконных классов существует метод TWinControl.CreateSubClass.) Функция RealGetWindowClass позволяет узнать имя класса-предка, если такой имеется. Соответствующая часть кода примера приведена в листинге 1.45.

Листинг 1.45. Получение реального класса окна

```
GetClassName(Wnd, ClassName, ClassNameLen);
ClassName[ClassNameLen – 1] := #0;
ListParams.Items[2].SubItems[0] := ClassName;
RealGetWindowClass(Wnd, ClassName, ClassNameLen);
ClassName[ClassNameLen – 1] := #0;
ListParams.Items[3].SubItems[0] := ClassName;
```

У окна, если оно имеет стиль WS_CHILD, должно быть родительское окно. Если такого стиля нет, то окно располагается непосредственно на рабочем столе. Кроме того, такое окно может (но не обязано) иметь владельца. Получить дескриптор родительского окна можно с помощью функции GetParent. Владельца — с помощью функции GetWindow с параметром GW_OWNER.

Примечание

Кроме GetParent существует функция GetAncestor, которая также возвращает дескриптор родительского окна, если она вызвана с параметром GA_PARENT. Разница между этими функциями заключается в том, что для окон верхнего уровня (т. е. расположенных непосредственно на рабочем столе) GetParent возвращает 0, а GetAncestor — дескриптор рабочего стола (этот дескриптор можно получить через функцию GetDesktopWindow).

Значительную часть кода программы составляет анализ того, какие флаги присутствуют в стиле окна. В этом нет ничего сложного, но он громоздкий из-за большого числа флагов. Следует также учесть, что для стандартных классов одни и те же числовые значения могут иметь разный смысл. Так, например, константы ES_NOHIDESEL и BS_LEFT имеют одинаковые значения. Поэтому при расшифровке стиля следует также учитывать класс окна. Приводить здесь этот код мы не будем по причине его тривиальности. Его можно посмотреть в примере на компакт-диске.

1.3.2. Обобщающий пример 2 — Ассоциированные файлы и предотвращение запуска второй копии приложения

Расширения файлов могут быть связаны *(ассоциированы)* с определенной программой. Такие ассоциации помогают системе выбрать программу для выполнения различных действий с файлом из Проводника. Так, например, если на компьютере установлен Microsoft Office, двойной щелчок в Проводнике на файле с расширением xls приведет к запуску Microsoft Excel и открытию файла в нем. Это происходит потому, что расширение xls ассоциировано с приложением Microsoft Excel.

Примечание

Добиться аналогичного эффекта в своей программе можно используя функцию ShellExecute (стандартная системная функция, в Delphi импортируется в модуле ShellAPI). Эта функция запускает файл, имя которого передано ей как параметр. Если это исполняемый файл, он запускается непосредственно, если нет — функция ищет ассоциированное с расширением файла приложение и открывает файл в нем.

Пример, который мы здесь рассмотрим (программа DKSView), умеет ассоциировать файлы с расширением dks с собой, а также проверять, не были ли они ассоциированы с другим приложением. DKSView является MDI-приложением, т. е. может открывать одновременно несколько файлов. Если приложение уже запущено, а пользователь пытается открыть еще один dks-файл, желательно, чтобы он открывался не в новом экземпляре DKSView, а в новом окне уже имеющегося. Поэтому наш пример будет также уметь обнаруживать уже запущенный экземпляр программы и переадресовывать открытие файла ему.

1.3.2.1. Ассоциирование расширения с приложением

Файловые ассоциации прописываются в реестре, в разделе HKEY_CLASSES_ROOT. Чтобы связать расширение с приложением, необходимо выполнить следующие действия:

1. В корне раздела HKEY_CLASSES_ROOT нужно создать новый раздел, имя которого совпадает с расширением с точкой перед ним (в нашем случае это будет раздел с именем ".dks"). В качестве значения по умолчанию в этот раздел должна быть записана непустая строка, которая будет идентифицировать соответствующий тип файла. Содержимое этой строки может быть произвольным и определяется разработчиком (в нашем случае эта строка имеет значение "DKS_View_File").
2. Далее в корне раздела HKEY_CLASSES_ROOT следует создать раздел, имя которого совпадает со значением ключа из предыдущего пункта (т. е. в нашем случае — с именем "DKS_View_File"). В качестве значения по умолчанию для этого ключа нужно поставить текстовое описание типа (это описание будет показываться пользователю в Проводнике в качестве типа файла).
3. В этом разделе создать подраздел Shell, в нем — подраздел Open, а в нем — подраздел Command, значением по умолчанию которого должна стать командная строка для запуска файла. Имя файла в ней заменяется на %1 (подробнее о командной строке чуть ниже).
4. Описанных действий достаточно, чтобы система знала, как правильно открывать файл из Проводника или с помощью `ShellExecute`. Однако правила хорошего тона требуют, чтобы с файлом была ассоциирована также иконка, которую будет отображать рядом с ним Проводник. Для этого в разделе, созданном во втором пункте, следует создать подраздел "DefaultIcon" и в качестве значения по умолчанию задать ему имя файла, содержащего иконку. Если это ico-файл, содержащий только одну иконку, к имени файла ничего добавлять не нужно. Если иконка содержится в файле, в котором может быть несколько иконок (например, в exe или dll),

после имени файла следует поставить запятую и номер требуемой иконки (иконки нумеруются, начиная с нуля).

Приведенный список — это самый минимальный набор действий, необходимых для ассоциирования расширения с приложением. Вернемся к третьему пункту. Имя подраздела "Open" задает команду, связанную с данным расширением, т. е. в данном случае — команду "Open". В разделе Shell можно сделать несколько аналогичных подразделов — в этом случае с файлом будет связано несколько команд. У функции `ShellExecute` есть параметр `lpOperation`, в котором задается имя требуемой команды. Пользователь Проводника может выбрать одну из возможных команд через контекстное меню, которое появляется при нажатии правой кнопки мыши над файлом. Существует возможность установить для этих пунктов меню более дружественные имена. Для этого нужно задать значение по умолчанию соответствующего подраздела. В этой строке допустим символ "&" для указания "горячей" клавиши, аналогично тому, как это делается, например, в компоненте `TButton`.

Если в `ShellExecute` команда не указана явно, используется команда по умолчанию (то же самое происходит при двойном щелчке на файле в Проводнике). Если не оговорено обратное, командой по умолчанию является команда "Open" или, если команды "Open" нет, первая команда в списке. При необходимости можно задать другую команду по умолчанию. Для этого нужно указать ее название в качестве значения по умолчанию раздела Shell.

В нашем примере будет две команды: Open (открыть для редактирования) и View (открыть для просмотра). Поэтому информация в реестр заносится так, как показано в листинге 1.46.

Листинг 1.46. Занесение в реестр информации, необходимой для ассоциирования файла с приложением

```
const
  FileExt = '.dks';
  FileDescr = 'DKS_View_File';
  FileTitle = 'Delphi Kingdom Sample file';
  OpenCommand = '&Открыть';
  ViewCommand = '&Просмотреть';

// Занесение в реестр информации об ассоциации
// расширения dks с программой
procedure TDKSViewMainForm.SetAssociation(Reg: TRegistry);
begin
  Reg.OpenKey('\' + FileExt, True);
  Reg.WriteString('', FileDescr);
```

```
  Reg.OpenKey('\' + FileDescr, True);
  Reg.WriteString('', FileTitle);
  Reg.OpenKey('Shell', True);
  Reg.OpenKey('Open', True);
  Reg.WriteString('', OpenCommand);
  Reg.OpenKey('command', True);
  Reg.WriteString('', '"' + ParamStr(0) + '" "%1"');
  Reg.OpenKey('\' + FileDescr, True);
  Reg.OpenKey('Shell', True);
  Reg.OpenKey('View', True);
  Reg.WriteString('', ViewCommand);
  Reg.OpenKey('command', True);
  Reg.WriteString('', '"' + ParamStr(0) + '" "%1" /v');
  Reg.OpenKey('\' + FileDescr, True);
  Reg.OpenKey('DefaultIcon', True);
  Reg.WriteString('', ParamStr(0) + ',0');
end;
```

1.3.2.2. Командная строка

Командная строка досталась Windows по наследству от DOS. Там основным средством общения пользователя с системой был ввод команд с клавиатуры. Команда запуска приложения выглядела так:

```
<Имя приложения> <Строка параметров>
```

Строка параметров — это произвольная строка, которая без изменений передавалась программе. От имени программы она отделялась пробелом (пробелы в именах файлов и директорий в DOS не допускались). Разработчик конкретного приложения мог, в принципе, интерпретировать эту строку как угодно, но общепринятым стал способ, когда строка разбивалась на отдельные параметры, которые разделялись пробелами. Вид и смысл параметров зависел от конкретной программы. В качестве параметров нередко передавались имена файлов, с которыми должна была работать программа.

В Windows мало что изменилось — функции `CreateProcess` и `ShellExecute`, запускающие приложение, по-прежнему используют понятие командной строки. Разве что теперь максимальная длина строки стала существенно больше, и командную строку можно получить в кодировке Unicode. Но, как и раньше, разделителем параметров считается пробел. Однако теперь пробел может присутствовать и в имени файла, как в имени самой программы, так и в именах файлов, передаваемых в качестве параметров. Чтобы отличать такой пробел от пробела-разделителя, параметры, содержащие пробелы, заключа-

ются в двойные кавычки. Если имя программы содержит пробелы, они тоже заключаются в двойные кавычки. И, конечно же, если в кавычки окажется заключенным параметр, в котором нет пробелов, хуже от этого не будет.

Для работы с параметрами командной строки в Delphi существуют две стандартные функции: `ParamCount` и `ParamStr`. Функция `ParamCount` возвращает количество параметров, переданных в командной строке, `ParamStr` — параметр с заданным порядковым номером. Параметры нумеруются начиная с единицы, нулевым параметром считается имя самой программы (при подсчетах с помощью `ParamCount` этот "параметр" не учитывается). Эти функции осуществляют разбор командной строки по описанным ранее правилам: разделитель — пробел, за исключением заключенных в кавычки. Кавычки, в которые заключен параметр, функция `ParamStr` не возвращает.

Ассоциированный файл запускается с помощью механизма командной строки. В реестр записывается командная строка (вместе с именем приложения), в которой имя открываемого файла заменяется на `%1`. Когда пользователь запускает ассоциированный файл (или он запускается приложением через `ShellExecute`), система извлекает из реестра соответствующую командную строку, вместо `%1` подставляет реальное имя файла и пытается выполнить получившуюся команду. Отметим, что если имя файла содержит пробелы, в кавычки оно автоматически не заключается, поэтому о кавычках приходится заботиться самостоятельно, заключая в них `%1`. Таким образом, в реестр в качестве командной строки должно записываться следующее:

```
<Имя программы> "%1"
```

Если существуют разные варианты запуска одного файла (т. е. как в нашем случае — open и view), они могут различаться дополнительными параметрами. В частности, в нашем примере для открытия для редактирования не будут требоваться дополнительные параметры, для открытия для просмотра в качестве второго параметра должен передаваться ключ `/v`, т. е. в реестр для этой команды будет записана такая строка:

```
<Имя программы> "%1" /v
```

Программа должна анализировать переданные ей параметры и открывать соответствующий файл в требуемом режиме. В нашем случае этот код выглядит очень просто (листинг 1.47).

Листинг 1.47. Анализ командной строки

```
procedure TDKSViewMainForm.FormShow(Sender: TObject);
var
  OpenForView: Boolean;
```

```
begin
  // Проверяем наличие ключа "/v" в качестве второго параметра
  OpenForView := (ParamCount > 1) and
    (CompareText(ParamStr(2), '/v') = 0);
  if ParamCount > 0 then
    OpenFile(ParamStr(1), OpenForView);
  ...
end;
```

В более сложных случаях (например, при большем числе команд для ассоциированного файла) анализ командной строки будет сложнее, но его принципы останутся теми же.

1.3.2.3. Поиск уже запущенной копии приложения

Во многих случаях желательно не давать пользователю возможности запустить второй экземпляр вашего приложения. В 16-разрядных версиях Windows все приложения выполнялись в одной виртуальной машине, и каждому из них через переменную `HPrevInstance` передавался дескриптор предыдущей копии. По значению `HPrevInstance` программа легко могла найти свой предыдущий экземпляр или определить, что других экземпляров нет, если `HPervInstance` равна нулю. В 32-разрядных версиях эта переменная для совместимости оставлена, но всегда равна нулю, т. к. предыдущая копия работает в своей виртуальной машине, и ее дескриптор не имеет смысла. Альтернативного механизма обнаружения уже запущенной копии система не предоставляет, приходится выкручиваться своими силами.

Для обнаружения уже запущенного приложения многие авторы предлагают использовать именованные системные объекты (мьютексы, семафоры, атомы и т. п.). При запуске программа пытается создать такой объект с определенным именем. Если оказывается, что такой объект уже создан, программа "понимает", что она — вторая копия, и завершается. Недостаток такого подхода — с его помощью можно установить только сам факт наличия предыдущей копии, но не более того. В нашем случае задача шире: при запуске второго экземпляра приложения должен активизироваться первый, а если второму экземпляру была передана непустая командная строка, первый должен получить эту строку и выполнить соответствующее действие, поэтому описанный способ нам не подходит.

Для решения задачи нам подойдут *почтовые ящики* (mailslots). Это специальные системные объекты для односторонней передачи сообщений между приложениями (ничего общего с электронной почтой эти почтовые ящики не имеют). Под сообщением здесь понимаются не сообщения Windows, а произвольный набор данных (здесь больше подходит скорее термин "дейтаграмма", а не "сообщение"). Каждый почтовый ящик имеет уникальное имя. Ал-

горитм отслеживания повторного запуска с помощью почтового ящика следующий. Сначала программа пытается создать почтовый ящик как сервер. Если оказывается, что такой ящик уже существует, то она подключается к нему как клиент и передает содержимое своей командной строки и завершает работу. Сервером в таком случае становится экземпляр приложения, запустившийся первым, — он-то и создаст почтовый ящик. Остальным экземплярам останется только передать ему данные.

Примечание

В случае аварийного завершения программы система сама закроет все открытые ею дескрипторы, поэтому даже если первая копия будет снята системой и не сможет корректно закрыть дескриптор почтового ящика, ящик будет уничтожен и не помешает пользователю запустить новую копию программы.

Почтовый ящик лучше создать как можно раньше, поэтому мы будем его создавать не в методе формы, а в основном коде проекта, который обычно программист не исправляет. В результате код в dpr-файле проекта будет выглядеть так, как показано в листинге 1.48.

Листинг 1.48. Создание почтового ящика в главном файле проекта

```
const
  MailslotName =
    '\\.\mailslot\DekphiKingdomSample_Viewer_FileCommand';
  EventName = 'DelphiKingdomSample_Viewer_Command_Event';

var
  ClientMailslotHandle: THandle;
  Letter: string;
  OpenForView: Boolean;
  BytesWritten: DWORD;

begin
  // Пытаемся создать почтовый ящик
  ServerMailslotHandle := CreateMailSlot(MailslotName, 0,
    MAILSLOT_WAIT_FOREVER, nil);
  if ServerMailslotHandle = INVALID_HANDLE_VALUE then
  begin
    if GetLastError = ERROR_ALREADY_EXISTS then
    begin
      // Если такой ящик уже есть, подключаемся к нему как клиент
      ClientMailslotHandle := CreateFile(MailslotName, GENERIC_WRITE,
        FILE_SHARE_READ, nil, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, 0);
```

```
      // В зависимости от того, какие переданы параметры, формируем
      // строку для передачи предыдущему экземпляру. Первый символ
      // строки — команда:
      // e — открыть файл для редактирования
      // v — открыть файл для просмотра
      // s — просто активизировать предыдущий экземпляр
      // Для команд e и v к строке, начиная со 2-го символа,
      // добавляется имя файла
      if ParamCount > 0 then
      begin
        OpenForView := (ParamCount > 1) and
          (CompareText(ParamStr(2), '/v') = 0);
        if OpenForView then
          Letter := 'v' + ParamStr(1)
        else
          Letter := 'e' + ParamStr(1);
      end
      else
        Letter := 's';
      // Отправляем команду в почтовый ящик
      WriteFile(ClientMailslotHandle, Letter[1], Length(Letter),
        BytesWritten, nil);
      // Сигнализируем об отправке данных через специальное событие
      CommandEvent := OpenEvent(EVENT_MODIFY_STATE, False, EventName);
      SetEvent(CommandEvent);
      // Закрываем все дескрипторы
      CloseHandle(CommandEvent);
      CloseHandle(ClientMailslotHandle);
    end;
  end
  else
  begin
    // Создаем событие для сигнализирования о поступлении данных
    CommandEvent := CreateEvent(nil, False, False, EventName);
    // Выполняем обычный для VCL-приложений цикл
    Application.Initialize;
    Application.CreateForm(TDKSViewMainForm, DKSViewMainForm);
    Application.Run;
    // Закрываем все дескрипторы
    CloseHandle(ServerMailslotHandle);
    CloseHandle(CommandEvent);
  end;
end.
```

Теперь осталось "научить" первую копию приложения обнаруживать момент, когда в почтовом ящике оказываются сообщения, и забирать их оттуда. Было бы идеально, если при поступлении данных главная форма получала бы какое-то сообщение, но готового такого механизма, к сожалению, не существует. Из положения можно выйти, задействовав события.

Примечание

События — это объекты синхронизации, использующиеся в системе. Событие может быть взведено и сброшено. С помощью функции `WaitForSingleObject` можно перевести нить в состояние ожидания до тех пор, пока указанное событие не будет взведено. Подробное рассмотрение объектов синхронизации выходит за рамки нашей книги; они детально описаны, например, в [2].

В принципе, при использовании перекрытого ввода-вывода система может сама взводить указанное программой событие при получении данных почтовым ящиком, но перекрытый ввод-вывод имеет ограниченную поддержку в Windows 9*x*/ME и на почтовые ящики не распространяется. Чтобы приложение могло работать не только в Windows NT/2000/XP, мы не будем применять перекрытый ввод-вывод.

События относятся к именованным объектам, поэтому с их помощью можно синхронизировать разные процессы. В нашем случае первая копия приложения с помощью `CreateEvent` создает событие, а последующие копии с помощью `OpenEvent` получают дескриптор этого события и взводят его, чтобы послать сигнал о появлении данных в почтовом ящике. Для обнаружения этого момента в первой копии приложения создается отдельная нить, которая ожидает событие и, дождавшись, посылает главной форме сообщение (эта нить практически не требует процессорного времени, потому что почти все время находится в режиме ожидания, т. е. квант времени планировщик задач ей не выделяет; по крайней мере, проверка наличия данных в главной нити по таймеру отняла бы больше ресурсов). Это сообщение определяется пользователем и берется из диапазона `WM_USER`, т. к. его широковещательной рассылки не будет. При получении этого сообщения форма выполняет код, приведенный в листинге 1.49.

Листинг 1.49. Реакция формы на поступление данных в почтовый ящик

```
// Реакция на получение команд от других экземпляров приложения
proedure TDKSViewMainForm.WMCommandArrived(var Message: TMessage);
var
  Letter: string;
begin
  // Переводим приложение на передний план
  GoToForeground;
```

```
  // Пока есть команды, читаем их и выполняем
  Letter := ReadStringFromMailslot;
  while Letter <> '' do
  begin
    // Анализируем и выполняем команду.
    // Команда "s" не требует никаких действий, кроме перевода
    // приложения на передний план, поэтому здесь мы ее не учитываем
    case Letter[1] of
      'e': OpenFile(Copy(Letter, 2, MaxInt), False);
      'v': OpenFile(Copy(Letter, 2, MaxInt), True);
    end;
    Letter := ReadStringFromMailslot;
  end;
end;

// Чтение очередного сообщения из почтового ящика
function TDksViewMainForm.ReadStringFromMailslot: string;
var
  MessageSize: DWORD;
begin
  // Получаем размер следующего сообщения в почтовом ящике
  GetMailslotInfo(ServerMailslotHandle, nil, MessageSize, nil, nil);
  // Если сообщения нет, возвращаем пустую строку
  if MessageSize = MAILSLOT_NO_MESSAGE then
  begin
    Result := '';
    Exit;
  end;
  // Выделяем для сообщения буфер и читаем его в этот буфер
  SetLength(Result, MessageSize);
  ReadFile(ServerMailslotHandle, Result[1], MessageSize,
    MessageSize, nil);
end;
```

Примечание

Так как события являются именованными объектами, второй экземпляр приложения мог бы обнаруживать наличие первого не по почтовому ящику, а по событию. Более того, если бы нам требовалось не передавать данные первому экземпляру, а только активизировать его, можно было бы вообще обойтись одним только событием.

1.3.2.4. Перевод приложения на передний план

Первая копия приложения, получив команду от другой копии, должна вывести себя на передний план. Казалось бы, все просто: с помощью функции

SetForegroundWindow мы можем вывести туда любое окно. Однако так было только до Windows 95 и NT 4. В более поздних версиях введены ограничения, и теперь программа не может вывести себя на передний план по собственному усмотрению. Функция SetForegroundWindow просто заставит мигать соответствующую кнопку на панели задач.

Тем не менее, если программа свернута, команда Application.Restore не только восстанавливает окно, но и выводит его на передний план, что нам и требуется. Ну а если программа не свернута, то "выливаем из чайника воду и тем самым сводим задачу к предыдущей": сначала сворачиваем приложение с помощью Application.Minimize, а потом разворачиваем его. Цели мы добились — главное окно на переднем плане.

Дело портит только то, что изменение состояния окна сопровождается анимацией: видно, как главное окно сначала сворачивается, а потом разворачивается. Чтобы убрать этот неприятный эффект, можно на время сворачивания/разворачивания окна запретить анимацию, а потом восстановить ее. С учетом этого метод GoToForeground выглядит так, как показано в листинге 1.50.

Листинг 1.50. Перевод приложения на передний план

```
// Перевод приложения на передний план
procedure TDKSViewMainForm.GoToForeground;
var
  Info: TAnimationInfo;
  Animation: Boolean;
begin
  // Проверяем, включена ли анимация для окон
  Info.cbSize := SizeOf(TAnimationInfo);
  Animation := SystemParametersInfo(SPI_GETANIMATION,
    SizeOf(Info), @Info, 0) and (Info.iMinAnimate <> 0);
  // Если включена, отключаем, чтобы не было ненужного мерцания
  if Animation then
  begin
    Info.iMinAnimate := 0;
    SystemParametersInfo(SPI_SETANIMATION, SizeOf(Info), @Info, 0);
  end;
  // Если приложение не минимизировано, минимизируем
  if not IsIconic(Application.Handle) then
    Application.Minimize;
  // Восстанавливаем приложение. При этом оно автоматически выводится
  // на передний план
  Application.Restore;
```

```
  // Если анимация окон была включена, снова включаем ее
  if Animation then
  begin
    Info.iMinAnimate := 1;
    SystemParametersInfo(SPI_SETANIMATION, SizeOf(Info), @Info, 0);
  end;
end;
```

Теперь у нас сделано все, что нужно: приложение умеет ассоциировать расширение с двумя командами; проверять, не ассоциировано ли расширение с другим приложением, и если да, предлагать пользователю установить эту ассоциацию; запрещать запуск второй копии приложения, переводя вместо этого на передний план первую копию; передавать параметры второй копии первой, чтобы она могла выполнить требуемые действия.

1.3.3. Обобщающий пример 3 — "Дырявое" окно

В этом примере мы создадим "дырявое" окно. Те, кто уже знаком с функцией `SetWindowRgn`, знает, что сделать "дырку" в окне или придать ему какую-либо другую необычную форму не так уж и сложно. Но мы здесь пойдем дальше: у дырки в нашем окне будет рамка, и пользователь сможет изменять размеры и положение дырки так же, как он может изменять положение и размеры окна. Как это выглядит, показано на рис. 1.14.

Рассмотрим те средства, которые нам понадобятся для реализации этого.

1.3.3.1. Сообщение *WM_NCHCHITTEST*

Каждое окно в Windows делится на две области: *клиентскую* и *неклиентскую*. Клиентской называется та область, в которой отображается содержимое окна. Неклиентская область — это различные служебные области окна: рамка, заголовок, полосы прокрутки, главное меню и т. п. Положение клиентской части окна относительно неклиентской определяет само окно при обработке сообщения `WM_NCCALCRECT`. Многие окна (особенно различные элементы управления) вообще не имеют неклиентской части.

Некоторые сообщения для клиентской части окна имеют аналоги для неклиентской. Например, перерисовка клиентской области осуществляется с помощью сообщения `WM_PAINT`, а неклиентской — `WM_NCPAINT`. Нажатие левой кнопки мыши над клиентской частью окна генерирует сообщение `WM_LBUTTONDOWN`, а над неклиентской — `WM_NCLBUTTONDOWN` и т. п.

Неклиентская область неоднородна: в нее входит заголовок, кнопки сокрытия, разворачивания и закрытия окна, иконка системного меню, главное меню, вертикальная и горизонтальная полосы прокрутки и рамка. Рамка тоже

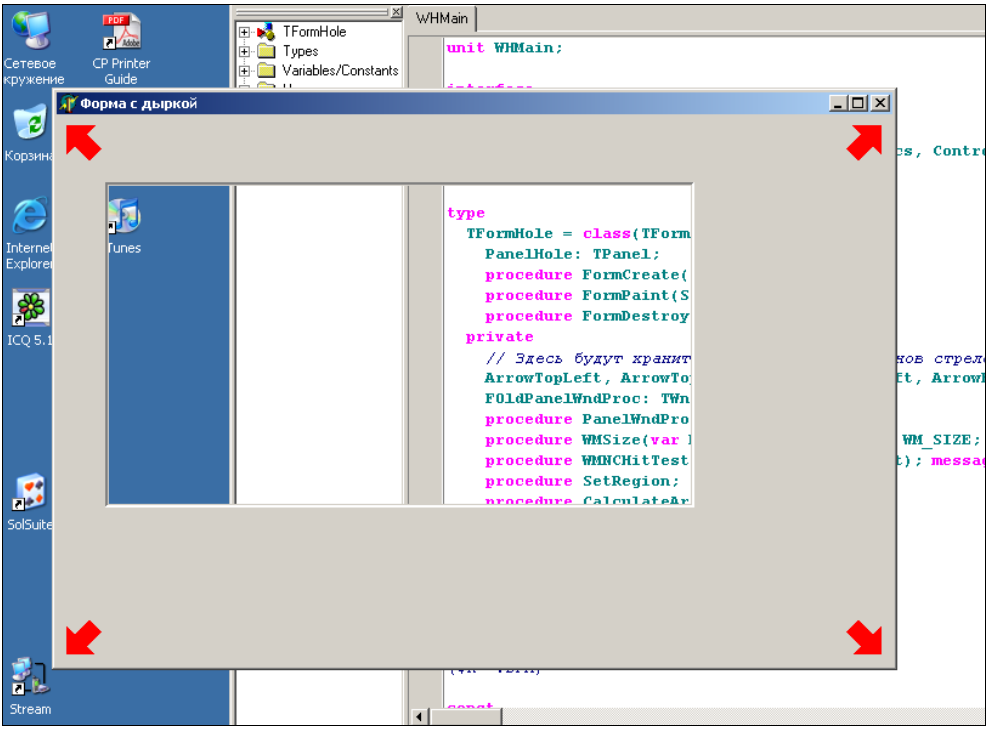


Рис. 1.14. "Дырявое" окно

неоднородна — она имеет левую, правую, верхнюю и нижнюю границы и четыре угла. Сообщение `WM_NCCALCSIZE` позволяет выяснить, какая область окна является неклиентской, но не позволяет узнать, где какая часть неклиентской области находится. Эта задача решается с помощью другого сообщения — `WM_NCHITTEST`. В качестве входных параметров `WM_NCHITTEST` получает координаты точки, а результат кодирует, к какой части окна относится эта точка (например, `HTCLIENT` означает, что точка принадлежит к клиентской части окна, `HTCAPTION` — к заголовку, `HTLEFT` — к левой границе рамки, меняющей размер, и т. п.).

При любых событиях от мыши система начинает с того, что посылает окну сообщение `WM_NCHITTEST` с координатами положения мыши. Получив результат, система решает, что делать дальше. В частности, при нажатии левой кнопки мыши окну посылается `WM_NCHITTEST`. Затем, если результатом был `HTCLIENT`, посылается сообщение `WM_LBUTTONDOWN`, в противном случае — `WM_NCLBUTTONDOWN`. При каждом перемещении мыши окно также получает `WM_NCHITTEST` — это позволяет системе постоянно отслеживать, над какой частью окна находится курсор, при необходимости меняя его вид (как, например, при прохождении курсора над рамкой).

Что будет, если подменить обработчик WM_NCHITTEST? Например, так, чтобы при попадании точки в клиентскую часть окна он возвращал не HTCLIENT, а HTCAPTION? Это приведет к тому, что любые события от мыши над клиентской областью будут восприниматься так же, как над заголовком. Например, можно будет взять окно за клиентскую часть и переместить его, а двойной щелчок на ней приведет к разворачиванию окна. Однако это полностью блокирует нормальную реакцию на мышь, потому что вместо клиентских "мышиных" сообщений окно будет получать неклиентские.

С практической точки зрения окно, которое можно таскать за любую точку, обычно не очень интересно (особенно это касается приложений, разработанных с помощью VCL: на мышь перестанет правильно реагировать не только само окно, но и расположенные на нем неоконные элементы управления). Однако обработчик WM_NCHITTEST можно сделать более интеллектуальным и получить довольно интересные эффекты. Например, положив на форму панель и переопределив у панели обработчик WM_NCHITTEST таким образом, чтобы при нахождении мыши около границ панели возвращался результат, соответствующий различным частям рамки с изменяемым размером, можно получить панель, размеры которой пользователь программы сможет изменять: система будет реагировать на эту область панели как на обычную рамку, которую можно взять и потянуть. (Пример такой панели можно увидеть в статье "Компонент, который меняет свои размеры в режиме run-time аналогично тому, как это происходит в design-time" **http://www.delphikingdom.com /asp/viewitem.asp?catalogid=22**.) Фантазия может подсказать и многие другие способы получения интересных эффектов с помощью WM_NCHITTEST.

1.3.3.2. Регионы

Регионы — это особые графические объекты, представляющие собой области произвольной формы. Ограничений на форму региона нет, они даже не обязаны быть связными. Существует ряд функций для создания регионов простых форм (CreateRectRgn, CreateEllipticRgn, CreatePolygonRgn и т. п.), а также функция CombineRgn для объединения регионов различными способами. Все это вместе позволяет получать регионы любых форм.

Область применения регионов достаточно широка. Ранее мы уже видели, как с помощью регионов можно ограничить область вывода графики. Здесь же мы будем с помощью функции SetWindowRgn изменять форму окна, придавая ему форму заданного региона.

1.3.3.3. Сообщения *WM_SIZE* и *WM_SIZING*

События WM_SIZE и WM_SIZING позволяют окну реагировать на перемещение его пользователем. В "классическом" варианте, когда пользователь начинает тянуть рамку окна, на экране рисуется "резиновый" прямоугольник, соответ-

ствующая сторона или угол которого движется за курсором мыши. Окно получает сообщение WM_SIZING при каждом изменении размера этого прямоугольника. Параметр lParam при этом содержит указатель на структуру TRect с новыми координатами прямоугольника. Окно может не только прочитать эти координаты, но и изменить их, блокировав тем самым нежелательные изменения размера. На этом, в частности, основано использование свойства Constraints: если размер окна при перемещении становится меньше или больше заданного, при обработке сообщения WM_SIZING размер увеличивается или уменьшается до необходимого. Параметр wParam содержит информацию о том, за какую сторону или угол тянет пользователь, чтобы программа знала, координаты какого из углов прямоугольника нужно смещать, если возникнет такая необходимость.

После того как пользователь закончит изменять размеры окна и отпустит кнопку мыши, окно получает сообщение WM_SIZE. При получении этого сообщения окно должно перерисовать себя с учетом новых размеров. (Окно получает сообщение WM_SIZE после изменения его размеров по любой причине, а не только из-за действий пользователя.)

Описанный "классический" вариант в чистом виде существует только в Windows 95. Во всех более поздних версиях по умолчанию включена опция отображения содержимого окна при перетаскивании и изменении размеров (начиная с Windows XP эта опция не только включается по умолчанию, но и не отключается средствами пользовательского интерфейса). В таком режиме при изменении размеров окна вместо прямоугольника "резиновым" становится само окно, и любое перемещение мыши при изменении размеров приводит к перерисовке окна. В этом режиме окно получает сообщение WM_SIZE каждый раз после сообщения WM_SIZING, а не только при завершении изменения размеров. Но в целом логика этих сообщений остается прежней, просто с точки зрения программы это выглядит так, как будто пользователь изменяет размеры окна "по чуть-чуть".

1.3.3.4. А теперь — все вместе

Комбинация описанных достаточно простых вещей позволяет построить окно с дыркой, имеющей изменяемые размеры.

Для начала объявим несколько констант, которые нам потребуются при вычислении размеров дырки и т. п. (листинг 1.51).

Листинг 1.51. Константы примера WndHole

```
const
  // Минимальное расстояние от края дырки до края окна.
  HoleDistance = 40;
```

```
  // Зона чувствительности рамки панели — насколько пикселов
  // может отстоять курсор вглубь от края панели, чтобы его
  // положение расценивалось как попадание в рамку.
  BorderMouseSensivity = 3;
  // Зона чувствительности угла рамки панели — насколько пикселов
  // может отстоять курсор от угла панели, чтобы его
  // положение расценивалось как попадание в угол рамки.
  CornerMouseSensivity = 15;
  // Толщина рамки дырки, использующаяся при вычислении региона
  HoleBorder = 3;
  // Минимальная ширина и высота дырки
  MinHoleSize = 10;
  // Смещение стрелки относительно соответствующего угла
  ArrowOffset = 8;
```

Теперь приступаем к созданию программы. На форму "кладем" панель. С помощью функции SetWindowRgn устанавливаем такую форму окна, чтобы от панели была видна только рамка, а на всю внутреннюю часть панели пришлась дырка. Рамку выбираем такую, чтобы панель выглядела утопленной, так края дырки будут выглядеть естественней. Для расчета региона используется метод SetRegion (листинг 1.52), он вызывается всегда, когда нужно изменить регион окна.

Листинг 1.52. Метод SetRegion, устанавливающий регион окна

```
procedure TFormHole.SetRegion;
var
  Rgn1, Rgn2: HRGN;
  R, R2: TRect;
begin
  // Создаем регион, соответствующий прямоугольнику окна
  Rgn1 := CreateRectRgn(0, 0, Width, Height);
  // Нам потребуются координаты панели относительно левого
  // верхнего угла окна (а не относительно левого верхнего
  // угла клиентской области, как это задается свойствами
  // Left и Top). Функций для получения смещения клиентской
  // области относительно левого верхнего угла окна нет.
  // Придется воспользоваться сообщением WM_NCCalcRect
  R2 := Rect(Left, Top, Left + Width, Top + Height);
  Perform(WM_NCCALCSIZE, 0, LParam(@R2));
  // Переводим координаты полученного прямоугольника из
  // экранных в координаты относительно левого верхнего
  // угла окна
  OffsetRect(R2, -Left, -Top);
```

```
  // Получаем координаты панели относительно левого
  // верхнего угла клиентской области и пересчитываем их
  // в координаты относительно верхнего левого угла окна
  R := Rect(0, 0, PanelHole.Width, PanelHole.Height);
  OffsetRect(R, PanelHole.Left + R2.Left, PanelHole.Top + R2.Top);
  // Уменьшаем прямоугольник на величину рамки и создаем
  // соответствующий регион
  InflateRect(R, -HoleBorder, -HoleBorder);
  Rgn2 := CreateRectRgnIndirect(R);
  // Вычитаем один прямоугольный регион из другого, получая
  // прямоугольник с дыркой
  CombineRgn(Rgn1, Rgn1, Rgn2, RGN_DIFF);
  // Уничтожаем вспомогательный регион
  DeleteObject(Rgn2);
  // Назначаем регион с дыркой окну
  SetWindowRgn(Handle, Rgn1, True);
  // Обратите внимание, что регион, назначенный окну, нигде
  // не уничтожается. После выполнения функции SetWindowRgn
  // регион переходит в собственность системы, и она сама
  // уничтожит его при необходимости
end;
```

Сообщения, поступающие с панели, перехватываются через ее свойство `WindowProc` (подробно эта технология описана в первой части данной главы, здесь мы ее касаться не будем). Сообщение `WM_NCHITTEST` будем обрабатывать так, чтобы при попадании мыши на рамку панели возвращались такие значения, чтобы за эту рамку можно было тянуть. В обработчике сообщения `WM_SIZE` панели изменяем регион так, чтобы он соответствовал новому размеру панели. Все, дырка с изменяемыми размерами готова.

Теперь нужно научить "дырку" менять размеры при изменении размеров окна, если окно стало слишком маленьким, чтобы вместить в себя дырку.

Осталось только немного "навести красоту". "Красота" заключается в том, чтобы пользователь не мог уменьшить размеры дырки до нуля и увеличить так, чтобы она вышла за пределы окна, а также уменьшить окно так, чтобы дырка оказалась за пределами окна. Первая из этих задач решается просто: добавляется обработчик сообщения `WM_SIZING` для дырки таким образом, чтобы ее размеры не могли стать меньше, чем `MinHoleSize` на `MinHoleSize` пикселов, а границы нельзя было придвинуть к границам окна ближе, чем на `HoleDistance` пикселов. Вторая задача решается еще проще: в обработчике `WM_SIZE` дырки меняем свойство `Constraints` формы таким образом, чтобы пользователь не мог слишком сильно уменьшить окно. Теперь окно с дыркой ведет себя корректно при любых действиях пользователя с дыркой. Полу-

чившийся в результате код обработчика сообщений панели приведен в листинге 1.53.

Листинг 1.53. Обработчик сообщений панели, образующей "дырку"

```
procedure TFormHole.PanelWndProc(var Msg: TMessage);
var
  Pt: TPoint;
  R: TRect;
begin
 FOldPanelWndProc(Msg);
 if Msg.Msg = WM_NCHITTEST then
 begin
   // Вся хитрость обработки сообщения WM_NCHITTEST
   // заключается в правильном переводе экранных координат
   // в клиентские и в несколько муторной проверке попадания
   // мыши на сторону рамки или в ее угол. Дело упрощается
   // тем, что у панели нет неклиентской части, поэтому
   // верхний левый угол окна и верхний левый угол клиентской
   // части совпадают.
   Pt := PanelHole.ScreenToClient(Point(Msg.LParamLo, Msg.LParamHi));
   if Pt.X < BorderMouseSensivity then
     if Pt.Y < CornerMouseSensivity then
       Msg.Result := HTTOPLEFT
     else if Pt.Y >= PanelHole.Height – CornerMouseSensivity then
       Msg.Result := HTBOTTOMLEFT
     else
       Msg.Result := HTLEFT
   else if Pt.X >= PanelHole.Width – BorderMouseSensivity then
     if Pt.Y < CornerMouseSensivity then
       Msg.Result := HTTOPRIGHT
     else if Pt.Y >= PanelHole.Height – CornerMouseSensivity then
       Msg.Result := HTBOTTOMRIGHT
     else
      Msg.Result := HTRIGHT
   else if Pt.Y < BorderMouseSensivity then
     if Pt.X < CornerMouseSensivity then
       Msg.Result := HTTOPLEFT
     else if Pt.X >= PanelHole.Width – CornerMouseSensivity then
       Msg.Result := HTTOPRIGHT
     else
       Msg.Result := HTTOP
```

```
  else if Pt.Y >= PanelHole.Height – BorderMouseSensivity then
    if Pt.X < CornerMouseSensivity then
      Msg.Result := HTBOTTOMLEFT
    else if Pt.X >= PanelHole.Width – CornerMouseSensivity then
      Msg.Result := HTBOTTOMRIGHT
   else
      Msg.Result := HTBOTTOM;
end
else if Msg.Msg = WM_SIZE then
begin
  // Пересчитываем регион
  SetRegion;
  // Устанавливаем новые ограничения для размеров окна,
  // учитывающие новое положение дырки
  Constraints.MinWidth := Width – ClientWidth + PanelHole.Left +
    MinHoleSize + HoleDistance;
  Constraints.MinHeight := Height – ClientHeight + PanelHole.Top +
    MinHoleSize + HoleDistance;
end
else if Msg.Msg = WM_SIZING then
begin
  // Копируем переданный прямоугольник в переменную R,
  // одновременно пересчитывая координаты из экранных
  // в клиентские
  R.TopLeft := ScreenToClient(PRect(Msg.LParam)^.TopLeft);
  R.BottomRight := ScreenToClient(PRect(Msg.LParam)^.BottomRight);
  // Если ширина слишком мала, проверяем, за какую
  // сторону тянет пользователь. Если за левую -
  // корректируем координаты левой стороны, если за
  // правую – ее координаты
  if R.Right – R.Left < MinHoleSize then
    if Msg.WParam in [WMSZ_BOTTOMLEFT, WMSZ_LEFT, WMSZ_TOPLEFT] then
      R.Left := R.Right – MinHoleSize
    else
      R.Right := R.Left + MinHoleSize;
  // Аналогично действуем, если слишком мала высота
  if R.Bottom – R.Top < MinHoleSize then
    if Msg.WParam in [WMSZ_TOP, WMSZ_TOPLEFT, WMSZ_TOPRIGHT] then
      R.Top := R.Bottom – MinHoleSize
    else
      R.Bottom := R.Top + MinHoleSize;
  // Сдвигаем стороны, слишком близко подошедшие
  // к границам окна
```

```
    if R.Left < HoleDistance then
      R.Left := HoleDistance;
    if R.Top < HoleDistance then
      R.Top := HoleDistance;
    if R.Right > ClientWidth – HoleDistance then
      R.Right := ClientWidth – HoleDistance;
    if R.Bottom > ClientHeight – HoleDistance then
      R.Bottom := ClientHeight – HoleDistance;
    // Копируем прямоугольник R, переводя его координаты
    // обратно в экранные
    PRect(Msg.LParam)^.TopLeft := ClientToScreen(R.TopLeft);
    PRect(Msg.LParam)^.BottomRight := ClientToScreen(R.BottomRight);
  end;
end;
```

Остается еще одна ситуация, когда границы "дырки" могут подойти к границам окна слишком близко: когда пользователь меняет не границы "дырки", а границы самого окна. Чтобы этого не случилось, нужно отслеживать изменения размеров окна и соответствующим образом менять размеры дырки — для этого нам потребуется изменить размеры панели и пересчитать регион. Пересчет региона необходим и в случае увеличения размеров окна: если его не пересчитать, получится, что часть окна не будет попадать в регион и будет отрезана. Все перечисленные действия выполняются в обработчике сообщения WM_SIZE окна (листинг 1.54).

Листинг 1.54. Обработчик сообщения WM_SIZE главного окна

```
procedure TFormHole.WMSize(var Msg: TWMSize);
begin
  inherited;
  // При уменьшении размеров окна уменьшаем размер дырки,
  // если границы окна подошли слишком близко к ней
  if PanelHole.Left + PanelHole.Width >
    ClientWidth – HoleDistance then
    PanelHole.Width := ClientWidth – HoleDistance – PanelHole.Left;
  if PanelHole.Top + PanelHole.Height >
    ClientHeight – HoleDistance then
    PanelHole.Height := ClientHeight – HoleDistance – PanelHole.Top;
  // На случай увеличения окна пересчитываем его регион,
  // иначе та часть, которая добавилась, окажется за его
  // пределами и будет отрезана
  SetRegion;
```

```
  // Пересчитываем координаты стрелок
  CalculateArrows;
  Invalidate;
end;
```

Напоследок добавим к программе один "бантик": красные стрелки по углам формы, за которые можно потянуть, чтобы изменить ее размер. Каждая стрелка будет представляться отдельным регионом. Координаты регионов должны быть привязаны к соответствующим углам окна, поэтому при изменении размеров окна эти координаты придется пересчитывать (в листинге 1.54 можно найти соответствующий вызов). Все стрелки выглядят одинаково, но являются зеркальным отражением друг друга. Чтобы рассчитывать координаты стрелок, мы зададим координаты одной нарисованной заранее стрелки в виде константы, а потом будем рассчитывать на основе этих данных координаты реальных стрелок, отражая их от нужной плоскости и добавляя требуемое смещение (листинг 1.55).

Листинг 1.55. Расчет координат стрелок

```
// Координаты верхней левой стрелки, отсчитанные от точки
// (0,0). Для получения координат реальных стрелок эти точки
// будут смещаться и отражаться
const
  ArrowTemplate: TArrowCoords =
    ((X:0; Y:0), (X:24; Y:0), (X:17; Y:7), (X:29; Y:19), (X:19; Y:29),
     (X:7; Y:17), (X:0; Y:24));

procedure TFormHole.CalculateArrows;
var
  Arrow: TArrowCoords;
  I: Integer;
begin
  // Вычисление региона левой верхней стрелки
  // Координаты просто смещаются на постоянную величину
  for I := 0 to High(Arrow) do
  begin
    Arrow[I].X := ArrowTemplate[I].X + ArrowOffset;
    Arrow[I].Y := ArrowTemplate[I].Y + ArrowOffset;
  end;
  // При необходимости уничтожаем старый регион
  if ArrowTopLeft <> 0 then
    DeleteObject(ArrowTopLeft);
```

```
  ArrowTopLeft :=
    CreatePolygonRgn(Arrow[0], Length(Arrow), WINDING);
  // Вычисление региона правой верхней стрелки
  // Координаты по X отражаются и смещаются
  // на постоянную величину относительно правого края окна
  for I := 0 to High(Arrow) do
  begin
    Arrow[I].X := ClientWidth – ArrowOffset – 1 – ArrowTemplate[I].X;
    Arrow[I].Y := ArrowTemplate[I].Y + ArrowOffset;
  end;
  if ArrowTopRight <> 0 then
    DeleteObject(ArrowTopRight);
  ArrowTopRight :=
    CreatePolygonRgn(Arrow[0], Length(Arrow), WINDING);
  // Вычисление региона левой нижней стрелки
  // Координаты по Y отражаются и смещаются
  // на постоянную величину относительно нижнего края окна
  for I := 0 to High(Arrow) do
  begin
    Arrow[I].X := ArrowTemplate[I].X + ArrowOffset;
    Arrow[I].Y := ClientHeight – ArrowOffset – 1 – ArrowTemplate[I].Y;
  end;
  if ArrowBottomLeft <> 0 then
    DeleteObject(ArrowBottomLeft);
  ArrowBottomLeft :=
    CreatePolygonRgn(Arrow[0], Length(Arrow), WINDING);
  // Вычисление региона правой нижней стрелки
  // Координаты по обеим осям отражаются и смещаются
  // на постоянную величину относительно правого нижнего угла окна
  for I := 0 to High(Arrow) do
  begin
    Arrow[I].X := ClientWidth – ArrowOffset – 1 – ArrowTemplate[I].X;
    Arrow[I].Y := ClientHeight – ArrowOffset – 1 – ArrowTemplate[I].Y;
  end;
  if ArrowBottomRight <> 0 then
    DeleteObject(ArrowBottomRight);
  ArrowBottomRight :=
    CreatePolygonRgn(Arrow[0], Length(Arrow), WINDING);
end;
```

Следующий шаг — рисование стрелки на форме. Делается это очень просто (листинг 1.56).

Листинг 1.56. Рисование стрелок на форме

```
procedure TFormHole.FormPaint(Sender: TObject);
begin
  // Закрашиваем регионы стрелок
  Canvas.Brush.Style := bsSolid;
  Canvas.Brush.Color := clRed;
  FillRgn(Canvas.Handle, ArrowTopLeft, Canvas.Brush.Handle);
  FillRgn(Canvas.Handle, ArrowTopRight, Canvas.Brush.Handle);
  FillRgn(Canvas.Handle, ArrowBottomLeft, Canvas.Brush.Handle);
  FillRgn(Canvas.Handle, ArrowBottomRight, Canvas.Brush.Handle);
end;
```

Остался последний шаг — объяснить системе, что пользователь может, ухватив за стрелки, изменять размеры формы. Очевидно, что делается это через обработчик `WM_NCHITTEST`. Вопрос только в том, как узнать, когда координаты мыши попадают внутрь нарисованной стрелки, поскольку стрелка является объектом сложной формы, вычислить это не очень просто. Данная задача также решается с помощью регионов: попадание координат курсора в регион каждой из стрелок отслеживается с помощью стандартной функции `PtInRegion` (листинг 1.57).

Листинг 1.57. Обработчик `WM_NCHITTEST` формы

```
procedure TFormHole.WMNCHitTest(var Msg: TWMNCHitTest);
var
  Pt: TPoint;
begin
  // Чтобы правильно обрабатывать стандартную неклиентскую область,
  // вызываем унаследованный обработчик
  inherited;
  // Не забываем, что параметры WM_NCHITTEST дают экранные,
  // а не клиентские координаты
  Pt := ScreenToClient(Point(Msg.XPos, Msg.YPos));
  // Проверяем координаты на попадание в регионы стрелок
  if PtInRegion(ArrowTopLeft, Pt.X, Pt.Y) then
    Msg.Result := HTTOPLEFT
  else if PtInRegion(ArrowTopRight, Pt.X, Pt.Y) then
    Msg.Result := HTTOPRIGHT
  else if PtInRegion(ArrowBottomLeft, Pt.X, Pt.Y) then
    Msg.Result := HTBOTTOMLEFT
```

```
  else if PtInRegion(ArrowBottomRight, Pt.X, Pt.Y) then
    Msg.Result := HTBOTTOMRIGHT;
end;
```

Вот и все. С помощью нескольких нехитрых приемов мы получили окно, которое имеет такой необычный вид (см. рис. 1.14).

1.3.4. Обобщающий пример 4 — Линии нестандартного стиля

GDI позволяет рисовать линии разных стилей, но бывают ситуации, когда стандартных возможностей по изменению стиля линий не хватает. В этом разделе мы покажем, как рисовать линии произвольного стиля (начнем с прямых, потом перейдем к кривым Безье), а также сделаем "резиновую" линию, которую пользователь может тянуть мышью.

1.3.4.1. Получение координат точек прямой

Рисование нестандартных линий выполняется следующим образом: вычисляются координаты всех пикселов, составляющих данную прямую, а потом каждый из них (а при необходимости — и какая-либо его окрестность) раскрашиваются нужным цветом. Следовательно, возникает вопрос об определении координат пикселов.

Существует ряд алгоритмов вычисления этих координат. Наиболее известный из них — *алгоритм Брезенхэма* (Bresengham), который заключается в равномерном разбрасывании "ступенек" разной длины вдоль линии. В Windows используется алгоритм *GIQ* (Grid Intersection Quantization). Каждый пиксел окружается воображаемым ромбом из четырех пикселов. Если прямая имеет общие точки с этим ромбом, то пиксел рисуется.

Самостоятельно реализовывать один из таких алгоритмов нет необходимости — в Windows существует функция `LineDDA`, которая возвращает вызвавшей ее программе координаты линии. Эта функция в качестве параметра принимает координаты начала и конца линии, а также указатель на функцию, которой будут передаваться координаты пикселов. Данная функция должна быть реализована в программе. За время выполнения `LineDDA` эта функция будет вызвана столько раз, сколько пикселов содержит линия (как обычно в Windows, последний пиксел не считается принадлежащим прямой). Каждый раз при вызове ей будут передаваться координаты очередного пиксела, причем пикселы будут упорядочены от начала к концу прямой.

В примере Lines (рис. 1.15) с помощью `LineDDA` рисуется пять различных типов линий. Рассмотрим на примере самого сложного из реализуемых программой типов линии ("Зеленая елочка"), как это делается (листинг 1.58).

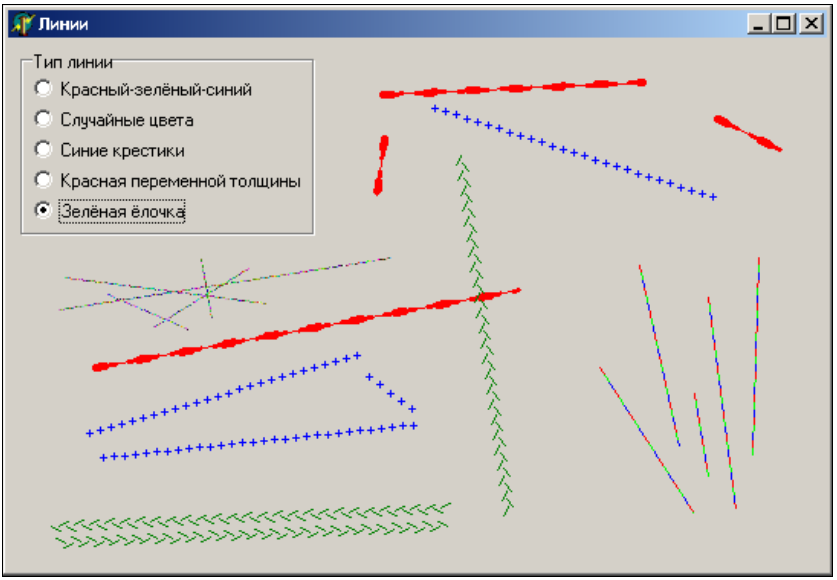


Рис. 1.15. Окно программы Lines

Листинг 1.58. Рисование линии сложного стиля

```
// Константы для типа "Зеленая елочка"
const
  // Угол отклонения "иголки" от направления линии
  FirNeedleAngle = 30;
  // Длина иголки
  FirNeedleLength = 8;

var
  Counter: Integer; // Счетчик точек линии
  // Вспомогательные переменные для построения "елочки"
  DX1, DY1, DX2, DY2: Integer;

// Линия в виде "елочки"
procedure LineDrawFir(X, Y: Integer; Canvas: TCanvas); stdcall;
begin
  with Canvas do
    case Counter mod 10 of
      0: begin
           MoveTo(X, Y);
           LineTo(X + DX1, Y + DY1);
         end;
```

```
      5: begin
           MoveTo(X, Y);
           LineTo(X + DX2, Y + DY2);
         end;
    end;
  Inc(Counter);
end;

procedure TLinesForm.Line(X1, Y1, X2, Y2: Integer);
var
  Angle: Extended;
begin
  case RGroupLine.ItemIndex of
    ...
    4: begin
         Counter := 0;
         Angle := ArcTan2(Y2 – Y1, X2 – X1);
         DX1 := Round(FirNeedleLength *
           Cos(Angle + Pi / 180 * FirNeedleAngle));
         DY1 := Round(FirNeedleLength *
           Sin(Angle + Pi / 180 * FirNeedleAngle));
         DX2 := Round(FirNeedleLength *
           Cos(Angle – Pi / 180 * FirNeedleAngle));
         DY2 := Round(FirNeedleLength *
           Sin(Angle – Pi / 180 * FirNeedleAngle));
         LineDDA(X1, Y1, X2, Y2, @LineDrawFir, Integer(Canvas));
       end;
  end;
end;
```

Каждая "иголка" — это линия длиной `FirNeedleLength` пикселов, отклоняющаяся от направления прямой на угол `FirNeedleAngle` градусов. "Иголки" отклоняются попеременно то в одну, то в другую сторону от прямой. В процедуре `Line` сначала рассчитываются смещения координат конца "иголки" относительно начала и результаты помещаются в глобальные переменные `DX1`, `DY1`, `DX2`, `DY2`. Переменная `Counter` служит для определения номера точки. Перед вызовом `LineDDA` она инициализируется нулем. Затем вызывается функция `LineDDA`, в качестве одного из параметров которой передается указатель на функцию обратного вызова `LineDrawFir`. В результате этого функция `LineDrawFir` будет вызвана последовательно для каждого из пикселов, составляющих линию, начиная с (X1, Y1). `LineDrawFir` ведет подсчет пикселов, каждый раз увеличивая `Counter` на единицу. Если остаток от деления номера

точки на 10 равен 0, рисуется "иголка", отклоняющаяся в положительном направлении, если 5 — в отрицательном. В остальных случаях не рисуется ничего. Так получается "елочка".

1.3.4.2. "Резиновая" линия и растровые операции

Теперь нужно дать пользователю возможность рисовать линии. Для этого мы используем стандартную "резиновую" линию: пользователь нажимает левую кнопку мыши и, удерживая ее, передвигает мышь. До тех пор, пока кнопка удерживается, за курсором тянется линия. Как только пользователь отпускает кнопку, линия "впечатывается" в рисунок.

Сама по себе реализация "резиновой" линии очень проста: при наступлении события `OnMouseDown` запоминаются координаты начала линии и взводится флаг, показывающий, что включен режим рисования "резиновой" линии. Также запоминаются координаты конца отрезка, который на данный момент совпадает с началом. В обработчике `OnMouseMove`, если включен режим рисования "резиновой" линии, стирается линия со старыми координатами конца и рисуется с новыми. При наступлении `OnMouseUp` программа выходит из режима рисования "резиновой" линии, рисуя окончательный ее вариант с текущими координатами конца.

Самое сложное в этой последовательности действий — стереть нарисованную ранее линию. Если бы у нас был однородный фон, можно было бы просто нарисовать старую линию еще раз цветом фона — это выглядело бы как ее стирание. Но поскольку фон не однородный, а составлен из нарисованных ранее линий, этот способ мы применить не можем.

Для решения этой задачи мы здесь рассмотрим самый простой метод — инверсное рисование (более сложный метод будет рассмотрен чуть позже). При этом каждая точка, принадлежащая линии, закрашивается не каким-либо фиксированным цветом, а инвертируется (т. е. к текущему цвету точки применяется операция `not`). Для стирания линии просто рисуем ее еще раз: двойная инверсия восстанавливает предыдущий цвет точек (`not not X = X` для любого `X`).

При рисовании пером и кистью GDI позволяет использовать различные *растровые операции*, которые определяют результирующий цвет каждого пиксела в зависимости от цвета фона и пера или кисти. По умолчанию применяется операция `R2_COPYPEN`, в которой цвет фона игнорируется, а результирующий цвет пиксела совпадает с цветом пера или кисти. Изменить растровую операцию можно с помощью функции `SetROP2` (двойка в названии функции показывает, что устанавливаемая растровая операция имеет два аргумента — цвет рисования и цвет фона; при выводе растровых рисунков могут применяться растровые операции с тремя аргументами — см. функцию

BitBlt). Нас будет интересовать операция R2_NOT, которая инвертирует фоновый цвет, игнорируя цвет пера или кисти.

Примечание

Растровая операция влияет на все, что рисуется с помощью пера и кисти, т. е. на рисование границ фигур и их заливку. Кроме того, растровая операция влияет также на результат работы функции SetPixel (и, соответственно, изменение цвета с помощью Canvas.Pixels[X, Y]), т. к. эта операция выполняется с помощью кистей.

Код, рисующий "резиновую" линию, приведен в листинге 1.59.

Листинг 1.59. Рисование "резиновой" линии инверсным методом

```
procedure TLinesForm.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    OldX := X;
    OldY := Y;
    BegX := X;
    BegY := Y;
    LineDrawing := True;
  end;
end;

procedure TLinesForm.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  if LineDrawing and ((X <> OldX) or (Y <> OldY)) then
    with Canvas do
    begin
      SetROP2(Handle, R2_NOT);
      Line(BegX, BegY, OldX, OldY); //    Стираем старую линию.
      Line(BegX, BegY, X, Y);       //    Рисуем новую.
      OldX := X;
      OldY := Y;
    end;
end;

procedure TLinesForm.FormMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
```

```
begin
  if (Button = mbLeft) and LineDrawing then
  begin
    case RGroupLine.ItemIndex of
      2: Canvas.Pen.Color := clBlue;
      3: begin
           Canvas.Brush.Color := clRed;
           Canvas.Pen.Color := clRed;
         end;
      4: Canvas.Pen.Color := clGreen;
    end;
    Line(BegX, BegY, X, Y);
    LineDrawing := False;
  end;
end;
```

Обратите внимание, что резиновая линия следует за мышью даже тогда, когда мышь выходит за пределы формы, т. е. форма получает сообщения мыши, когда курсор находится за ее пределами. Это становится возможным благодаря захвату мыши окном. Любое окно в Windows может захватить мышь для монопольного использования, и тогда все сообщения от мыши будет получать это окно, независимо от того, где находится курсор. В VCL любой визуальный компонент, у которого установлен стиль `csCaptureMouse` (а у формы он по умолчанию установлен) автоматически захватывает мышь при нажатии левой кнопки и освобождает при ее отпускании, поэтому мы получаем требуемый нам эффект автоматически.

1.3.4.3. Кривые Безье

Сделаем следующий шаг — научимся рисовать произвольным стилем не только прямые, но и кривые. Проще всего это сделать с так называемыми *кривыми Безье* — они, во-первых, поддерживаются системой Windows, а во-вторых, ими можно аппроксимировать многие другие кривые (в частности, в Windows NT/2000/XP все кривые — окружности, эллипсы, дуги — аппроксимируются кривыми Безье).

Теорию кривых Безье разработал П. де Кастело в 1959 году и, независимо от него, П. Безье в 1962 году. Для построения кривой Безье N-го порядка необходимо $N+1$ точек, две из которых определяют концы кривой, а остальные $N-1$ называются опорными. В компьютерной графике наибольшее распространение получили квадратичные кривые Безье, строящиеся по трем точкам, и кубические кривые Безье, строящиеся по четырем точкам. Квадратичные кривые Безье используются, например, в шрифтах TrueType при определении

контуров символов. Windows API позволяет строить только кубические кривые Безье.

Кубические кривые Безье задаются следующей формулой:

$$P(t) = A(1-t)^3 + 3Bt(1-t)^2 + 3Ct^2(1-t) + Dt^3, \qquad (1)$$

где A — начало кривой, D — ее конец, а B и C — первая и вторая опорные точки. Прямая AB касательная к кривой в точке A, прямая CD — в точке D. Параметр t изменяется от 0 до 1. При $t = 0$ $P(t) = A$, при $t = 1$ $P(t) = D$.

Одним из важнейших свойств кривой Безье является ее делимость. Если кривую разделить на две кривых в точке $t = 0,5$, каждая из полученных кривых также будет являться кривой Безье. На этом свойстве основывается алгоритм рисования кривых Безье: если кривая может быть достаточно точно аппроксимирована прямой, рисуется отрезок прямой, если нет — она разбивается на две кривых Безье, к каждой из которых вновь применяется этот алгоритм.

Для рисования кривых Безье служат функции `PolyBezier`, `PolyBezierTo` и `PolyDraw`.

В некоторых случаях удобно строить кривую Безье не по опорным точкам, а по точкам, через которые она должна пройти. Пусть кривая начинается в точке A, при $t = 1/3$ проходит через точку Bў, при $t = 2/3$ — через точку Cў, и заканчивается в точке D. Подставляя эти точки в уравнение (1), получаем систему, связывающую Bў и Cў с B и C. Решая систему, получаем

$$B = \frac{-5A + 18B\text{ў} - 9C\text{ў} + 2D}{6}, \quad C = \frac{2A - 9B\text{ў} + 18C\text{ў} - 5D}{6}. \qquad (2)$$

Из этих уравнений, в частности, следует, что для любых четырех точек плоскости существует, и притом единственная, кривая Безье, которая начинается в первой точке, проходит при $t = 1/3$ через вторую точку, при $t = 2/3$ — через третью и завершается в четвертой точке. Аналогичным образом можно вычислить опорные точки для кривой, которая должна проходить через заданные точки при других значениях t.

1.3.4.4. Траектории

API Windows реализует поддержку специфических объектов, называемых *траекториями* (path). Траектория представляет собой запись движения пера и включает один или несколько замкнутых контуров. Каждый контур состоит из отрезков прямых и кривых Безье. Для построения траектории в Windows NT/2000/XP могут быть задействованы все графические функции рисования прямых, кривых и замкнутых контуров, а также функции вывода

текста (в этом случае замкнутые контуры будут совпадать с контурами символов). В Windows 9*x*/Me могут быть использованы только функции рисования прямых, ломаных, многоугольников (за исключением `PolyDraw` и `Rectangle`), кривых Безье и функций вывода текста. Функции рисования эллипсов, окружностей и эллиптических дуг не могут быть использованы для создания траектории в Windows 9*x*/Me, т. к. в этих системах эллиптические кривые рисуются специальным алгоритмом, а не аппроксимируются кривыми Безье. Для создания траектории предусмотрены функции `BeginPath` и `EndPath`. Все вызовы графических функций, расположенные между `BeginPath` и `EndPath`, вместо вывода в контекст устройства будут создавать в нем траекторию.

После того как траектория построена, ее можно отобразить или преобразовать. Мы не будем здесь перечислять все возможные операции с траекториями, остановимся только на преобразовании траектории в ломаную. Как уже отмечалось, все контуры траектории представляют собой набор отрезков прямых и кривых Безье. С другой стороны, при построении кривой Безье она аппроксимируется ломаной. Следовательно, вся траектория может быть аппроксимирована набором отрезков прямой. Функция `FlattenPath` преобразует кривые Безье, входящие в состав траектории, в ломаные линии. Таким образом, после вызова этой функции траектория будет состоять из отрезков прямой.

Отметим также некоторые другие преобразования траектории, полезные для создания графических редакторов и подобных им программ. Функция `PathToRegion` позволяет преобразовать траекторию в регион. Это может понадобиться, в частности, при определении того обстоятельства, попадает ли курсор мыши в область объекта, представляемого сложной фигурой. Функция `WidenPath` превращает каждый контур траектории в два контура — внутренний и внешний. Расстояние между ними определяется толщиной текущего пера. Таким образом, траектория как бы утолщается. После преобразования утолщенной траектории в регион можно определять, попадает ли курсор мыши на кривую с учетом погрешности, определяемой толщиной пера.

Получить информацию о точках текущей траектории можно с помощью функции `GetPath`. Для каждой точки траектории эта функция возвращает координаты и тип точки (начальная линии, замыкающая точка отрезка, точка кривой Безье, конец контура).

Таким образом, создав траекторию из кривой Безье (`BeginPath`/`PolyBezier`/`EndPath`), мы можем преобразовать эту траекторию в ломаную (`FlattenPath`), а затем получить координаты узлов этой ломаной (`GetPath`). А каждое звено этой ломаной мы можем нарисовать произвольным стилем, используя `LineDDA`. Таким образом, задача построения кривой Безье сведена к уже решенной задаче построения отрезка.

В листинге 1.60 реализован метод DrawCurve, выполняющий указанные действия. Здесь FCurve — это поле формы типа TCurve, в котором хранятся координаты четырех точек, образующих кривую.

Листинг 1.60. Работа с траекторией на основе кривой Безье

```
type
  // Тип TCurve хранит координаты кривой в следующем порядке: начало,
  // первую промежуточную точку, вторую промежуточную точку, конец
  TCurve = array[0..3] of TPoint;

// Функция обратного вызова для LineDDA
procedure LineDrawFunc(X, Y: Integer; Canvas: TCanvas); stdcall;
begin
  case CurveForm.RGroupType.ItemIndex of
    // Разноцветные шарики
    0: if CurveForm.FCounter mod 10 = 0 then
       begin
         Canvas.Pen.Style := psSolid;
         Canvas.Pen.Width := 1;
         Canvas.Brush.Style := bsSolid;
         if CurveForm.FCounter mod 15 = 0 then
           Canvas.Pen.Color := clBlue
         else if CurveForm.FCounter mod 15 = 5 then
           Canvas.Pen.Color := clLime
         else
           Canvas.Pen.Color := clRed;
         Canvas.Brush.Color := Canvas.Pen.Color;
         Canvas.Ellipse(X – 2, Y – 2, X + 3, Y + 3);
       end;
    // Поперечные полосы
    1: if CurveForm.FCounter mod 5 = 0 then
       begin
         Canvas.Pen.Style := psSolid;
         Canvas.Pen.Width := 1;
         Canvas.Pen.Color := clBlue;
         Canvas.MoveTo(X – CurveForm.FDX, Y – CurveForm.FDY);
         Canvas.LineTo(X + CurveForm.FDX, Y + CurveForm.FDY);
       end;
    // Плакатное перо
    2: begin
         Canvas.Pen.Style := psSolid;
```

```
        // Предположим, некоторая точка прямой имеет координаты (X,Y),
        // а соседняя с ней — координаты (X+1,Y-1). Тогда при проведении
        // через эти точки наклонной линии одинарной ширины между ними
        // останутся незаполненные точки, как на шахматной доске.
        // Поэтому потребовалось увеличить толщину пера
        Canvas.Pen.Width := 2;
        Canvas.Pen.Color := clBlack;
        Canvas.MoveTo(X — 5, Y — 5);
        Canvas.LineTo(X + 6, Y + 6);
      end;
    // Цепочка
    3: begin
        case CurveForm.FCounter mod 15 of
          0: begin
               Canvas.Pen.Style := psSolid;
               Canvas.Pen.Width := 1;
               Canvas.Pen.Color := clBlack;
               Canvas.Brush.Style := bsClear;
               Canvas.Ellipse(X — 5, Y — 5, X + 6, Y + 6);
             end;
          2..13: Canvas.Pixels[X, Y] := clBlack;
        end;
      end;
  end;
  Inc(CurveForm.FCounter);
end;

procedure TCurveForm.DrawCurve(Canvas: TCanvas);
var
  LCurve: TCurve;
  I, Size: Integer;
  PtBuf: array of TPoint;
  TpBuf: array of Byte;
  L: Extended;
begin
  // LCurve хранит координаты начала и конца кривой и ее
  // опорных точек. Если включен режим рисования по опорным
  // точкам, LCurve совпадает с FCurve, если включен режим
  // рисования по точкам кривой, опорные точки LCurve[1]
  // и LCurve[2] рассчитываются по приведенным в книге
  // формулам на основании точек FCurve
  LCurve := FCurve;
```

```
if RGroupDrawMethod.ItemIndex = 1 then
begin
  LCurve[1].X := Round((-5 * FCurve[0].X + 18 * FCurve[1].X – 9 *
    FCurve[2].X + 2 * FCurve[3].X) / 6);
  LCurve[1].Y := Round((-5 * FCurve[0].Y + 18 * FCurve[1].Y – 9 *
    FCurve[2].Y + 2 * FCurve[3].Y) / 6);
  LCurve[2].X := Round((2 * FCurve[0].X – 9 * FCurve[1].X + 18 *
    FCurve[2].X – 5 * FCurve[3].X) / 6);
  LCurve[2].Y := Round((2 * FCurve[0].Y – 9 * FCurve[1].Y + 18 *
    FCurve[2].Y – 5 * FCurve[3].Y) / 6);
end;
// Создаем траекторию на основе кривой
BeginPath(Canvas.Handle);
Canvas.PolyBezier(LCurve);
EndPath(Canvas.Handle);
// Аппроксимируем траекторию отрезками прямых
FlattenPath(Canvas.Handle);
// Получаем число точек траектории. Так как сами точки никуда
// пока не копируются, в качестве фиктивного буфера можно указать
// любую переменную. В данном случае – переменную I
Size := GetPath(Canvas.Handle, I, I, 0);
// Выделяем память для хранения координат и типов точек траектории
SetLength(PtBuf, Size);
SetLength(TpBuf, Size);
// Получаем координаты и типы точек. Типы точек нас в данном случае
// не интересуют: у первой точки будет тип PT_MOVETO,
// а у остальных – PT_LINETO. Появление PT_MOVETO у других точек
// невозможно, т. к. траектория содержит только один замкнутый
// контур, состоящий из кривой и соединяющей ее концы прямой.
// Появление точек типа PT_BEZIERTO также исключено, т. к. после
// вызова FlattenPath контур содержит только отрезки прямых.
// Поэтому значения, записанные в TpBuf, будут в дальнейшем
// игнорироваться
GetPath(Canvas.Handle, PtBuf[0], TpBuf[0], Size);
FCounter := 0;
// Рисуем по отдельности каждый из отрезков, составляющих контур
for I := 1 to Size – 1 do
begin
  // Вычисляем длину отрезка
  L := Sqrt(Sqr(PtBuf[I – 1].X – PtBuf[I].X) +
    Sqr(PtBuf[I – 1].Y – PtBuf[I].Y));
  // Практика показала, что аппроксимированный контур может
  // содержать отрезки нулевой длины – видимо, это издержки
```

```
    // алгоритма аппроксимации. Так как в дальнейшем нам придется
    // делить на L, такие отрезки мы просто игнорируем, т. к.
    // на экране они все равно никак не отображаются
    if L > 0 then
    begin
      // Переменные FDX и FDY используются только при рисовании
      // линии типа "поперечные полосы". Если бы линии этого
      // типа не было, то FDX, FDY, а так же L можно было бы
      // не рассчитывать
      FDX := Round(4 * (PtBuf[I – 1].Y – PtBuf[I].Y) / L);
      FDY := Round(4 * (PtBuf[I].X – PtBuf[I – 1].X) / L);
      LineDDA(PtBuf[I – 1].X, PtBuf[I – 1].Y, PtBuf[I].X, PtBuf[I].Y,
        @LineDrawFunc, Integer(Canvas));
    end;
  end;
end;
```

1.3.4.5. Интерактивная кривая

Описанная технология создания "резиновой" линии не годится для рисования кривой Безье, т. к. пользователь должен задать координаты не двух точек, а четырех. Удобнее всего это сделать следующим образом: сначала нарисовать "резиновую" прямую, задав тем самым начало и конец кривой, а потом дать пользователю возможность перемещать опорные или промежуточные точки кривой до тех пор, пока она не будет завершена. При этом логично дать возможность перемещать и концы линии, а также менять ее стиль, т. е. свободно манипулировать незавершенной кривой. Для ее завершения будет использоваться кнопка **Завершить** (рис. 1.16).

Чтобы кривая была более дружественной для пользователя, мы не будем применять здесь растровые операции, а попытаемся нарисовать незавершенную кривую без искажения цветов. Для этого нужно хранить картинку с завершенными кривыми, и при выводе нового положения незавершенной кривой сначала выводить эту картинку, а потом поверх нее — незавершенную кривую в новом положении. Так как фон в нашем случае состоит только из нарисованных ранее кривых, то можно было бы просто хранить список, содержащий координаты и стиль каждой кривой, и при перерисовке фона сначала заливать всю форму фоновым цветом, а потом рисовать на ней каждую из этих кривых заново. Но рисование одной кривой — достаточно медленная операция, т. к. на основе кривой нужно создать траекторию, аппроксимировать ее отрезками и нарисовать каждый из них по отдельности с помощью `LineDDA`. При большом количестве кривых эта реакция на перемещение мыши будет занимать слишком много времени. Поэтому мы выберем другой метод: будет создан растр, содержащий все завершенные кривые, и при перерисовке

формы этот растр будет просто копироваться на нее. Так как операции с растрами выполняются очень быстро, мерцания фона не будет. Чтобы незавершенная кривая также не мерцала, будет установлен режим двойной буферизации.

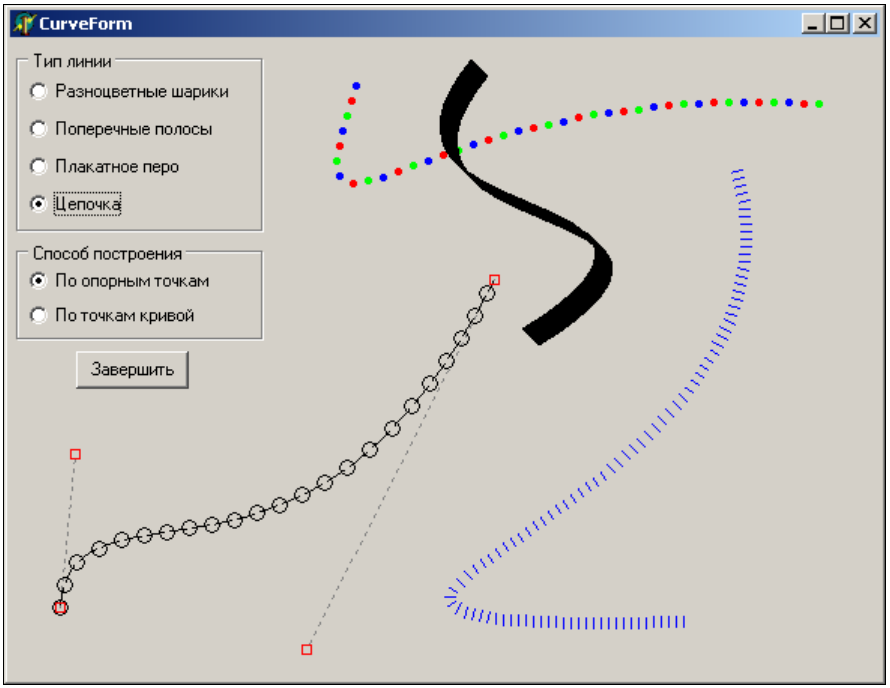


Рис. 1.16. Окно программы Bezier. Красные квадратики — области, за которые можно перемещать концы и опорные точки незавершенной кривой

Когда пользователь нажимает кнопку мыши, программа проверяет, есть ли незавершенная кривая. Если таких кривых нет, начинается создание новой кривой. До тех пор, пока пользователь не отпустит кнопку мыши, рисуется резиновая прямая. Эта прямая становится заготовкой для новой незавершенной кривой.

Если в момент нажатия кнопки мыши незавершенная кривая уже существует, координаты мыши сравниваются с координатами опорных и концевых точек и, если они оказываются достаточно близки к одной из них, дальнейшее перемещение мыши (при удерживании кнопки) приводит к перемещению соответствующей точки и перерисовке кривой в новом положении.

Изменение типа линии и/или способа построения отражается на незавершенной кривой — она немедленно перерисовывается в соответствии с новыми параметрами.

При нажатии кнопки **Завершить** незавершенная кривая рисуется уже не на самой форме, а на растре, содержащем фон. После этого кривая перестает существовать как кривая и становится набором пикселов на фоновой картинке, а программа вновь переходит в режим, когда нажатие кнопки мыши интерпретируется как создание новой кривой.

Реализацию интерактивной кривой в данном случае иллюстрирует листинг 1.61.

Листинг 1.61. Реализация интерактивной кривой

```
const
  // Чтобы перемещать точку кривой, пользователь должен попасть мышью
  // в некоторую ее окрестность. Константа RectSize задает размер этой
  // окрестности
  RectSize = 3;

type
  // Тип TDragPoint показывает, какую точку перемещает пользователь:
  //  ptNone — пользователь пытается тянуть несуществующую точку
  //  ptFirst — пользователь перемещает вторую точку "резиновой" прямой
  //  ptBegin — пользователь перемещает начало кривой
  //  ptInter1, ptInter2 — пользователь перемещает промежуточные точки
  //  ptEnd — пользователь перемещает конец кривой
  TDragPoint = (dpNone, dpFirst, dpBegin, dpInter1, dpInter2, dpEnd);

  TCurveForm = class(TForm)
    BtnEnd: TButton;
    RGroupType: TRadioGroup;
    RGroupDrawMethod: TRadioGroup;
    procedure FormCreate(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseMove(Sender: TObject; Shift: TShiftState;
      X, Y: Integer);
    procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormPaint(Sender: TObject);
    procedure BtnEndClick(Sender: TObject);
    procedure RGroupTypeClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    // Если FNewLine = True, незавершенных кривых нет, и при нажатии на
    // кнопку мыши начинает рисоваться новая кривая.
```

```
    // Если FNewLine = False, есть незавершенная кривая, и нажатия мыши
    // интерпретируются как попытки ее редактирования
    FNewLine: Boolean;
    // Поле FDragPoint указывает, какую точку перемещает пользователь
    FDragPoint: TDragPoint;
    // Поле FCurve хранит координаты незавершенной кривой
    FCurve: TCurve;
    // FBack — фоновый рисунок с завершенными кривыми
    FBack: TBitmap;
    // FCounter — счетчик точек, использующийся при рисовании отрезков
    // с помощью LineDDA
    FCounter: Integer;
    // FDX, FDY — смещения относительно координаты точки кривой для
    // рисования поперечной полосы
    FDX, FDY: Integer;
    // Функция PtNearPt возвращает True, если точка с координатами
    // (X1, Y1) удалена от точки Pt по каждой из координат не более
    // чем на RectSize
    function PtNearPt(X1, Y1: Integer; const Pt: TPoint): Boolean;
    // Процедура DrawCurve рисует кривую по координатам FCurve вида,
    // задаваемого RadioGroup.ItemIndex
    procedure DrawCurve(Canvas: TCanvas);
  end;

...

procedure TCurveForm.FormCreate(Sender: TObject);
begin
  FNewLine := True;
  FDragPoint := dpNone;
  FBack := TBitmap.Create;
  FBack.Canvas.Brush.Color := Color;
  // Устанавливаем размер фонового рисунка равным размеру развернутого
  // на весь рабочий стол окна
  FBack.Width := GetSystemMetrics(SM_CXFULLSCREEN);
  FBack.Height := GetSystemMetrics(SM_CYFULLSCREEN);
  // Включаем режим двойной буферизации, чтобы незавершенная кривая
  // не мерцала
  DoubleBuffered := True;
end;

procedure TCurveForm.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

```
begin
  if Button = mbLeft then
  begin
    // Если незавершенных кривых нет, начинаем рисование новой кривой
    if FNewLine then
    begin
      FDragPoint := dpFirst;
      FCurve[0].X := X;
      FCurve[0].Y := Y;
      FCurve[3] := FCurve[0];
    end
    else
    begin
      // Если есть незавершенная кривая, определяем, в какую точку попал
      // курсор мыши. Строго говоря, необходимо также запоминать,
      // насколько отстоят координаты курсора мыши от координат
      // контрольной точки, чтобы при первом перемещении не было скачка.
      // Но т. к. окрестность точки очень мала, этот прыжок практически
      // незаметен, и в данном случае этим можно пренебречь, чтобы
      // не усложнять программу
      if PtNearPt(X, Y, FCurve[0]) then
        FDragPoint := dpBegin
      else if PtNearPt(X, Y, FCurve[1]) then
        FDragPoint := dpInter1
      else if PtNearPt(X, Y, FCurve[2]) then
        FDragPoint := dpInter2
      else if PtNearPt(X, Y, FCurve[3]) then
       FDragPoint := dpEnd
      else
       FDragPoint := dpNone;
    end;
  end;
end;

procedure TCurveForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  if ssLeft in Shift then
  begin
    case FDragPoint of
      dpFirst, dpEnd:
        begin
          FCurve[3].X := X;
```

```
          FCurve[3].Y := Y;
          Refresh;
        end;
      dpBegin:
        begin
          FCurve[0].X := X;
          FCurve[0].Y := Y;
          Refresh;
        end;
      dpInter1:
        begin
          FCurve[1].X := X;
          FCurve[1].Y := Y;
          Refresh;
        end;
      dpInter2:
        begin
          FCurve[2].X := X;
          FCurve[2].Y := Y;
          Refresh;
        end;
    end;
  end;
end;

procedure TCurveForm.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  // Если кнопка отпущена при отсутствии незавершенной кривой, значит,
  // пользователь закончил рисование резиновой прямой, на основе которой
  // нужно делать новую кривую
  if (Button = mbLeft) and (FDragPoint = dpFirst) then
  begin
    FNewLine := False;
    FDragPoint := dpNone;
    // Промежуточные точки равномерно распределяем по прямой
    FCurve[1].X := FCurve[0].X +
      Round((FCurve[3].X – FCurve[0].X) / 3);
    FCurve[1].Y := FCurve[0].Y +
      Round((FCurve[3].Y – FCurve[0].Y) / 3);
    FCurve[2].X := FCurve[0].X +
      Round(2 * (FCurve[3].X – FCurve[0].X) / 3);
    FCurve[2].Y := FCurve[0].Y +
      Round(2 * (FCurve[3].Y – FCurve[0].Y) / 3);
```

```
    Refresh;
  end;
end;

procedure TCurveForm.FormPaint(Sender: TObject);
var
  I: Integer;
  L: Extended;
begin
  // Сначала выводим фон
  Canvas.Draw(0, 0, FBack);
  if FNewLine then
  begin
    // Если программа находится в режиме рисования резиновой прямой,
    // рисуем прямую от точки FCurve[0] до FCurve[3]. Значения FCurve[1]
    // и FCurve[2] на данном этапе игнорируются
    if FDragPoint = dpFirst then
    begin
      FCounter := 0;
      L := Sqrt(Sqr(FCurve[0].X – FCurve[3].X) +
        Sqr(FCurve[0].Y – FCurve[3].Y));
      if L > 0 then
      begin
        FDX := Round(4 * (FCurve[0].Y – FCurve[3].Y) / L);
        FDY := Round(4 * (FCurve[3].X – FCurve[0].X) / L);
        LineDDA(FCurve[0].X, FCurve[0].Y, FCurve[3].X, FCurve[3].Y,
          @LineDrawFunc, Integer(Canvas));
      end;
    end;
  end
  else
  begin
    // Если есть незавершенная кривая и установлен режим рисования
    // по опорным точкам, выводим отрезки, показывающие касательные
    // к кривой в ее начале и конце
    if RGroupDrawMethod.ItemIndex = 0 then
    begin
      Canvas.Pen.Style := psDot;
      Canvas.Pen.Width := 1;
      Canvas.Pen.Color := clDkGray;
      Canvas.MoveTo(FCurve[0].X, FCurve[0].Y);
      Canvas.LineTo(FCurve[1].X, FCurve[1].Y);
      Canvas.MoveTo(FCurve[3].X, FCurve[3].Y);
      Canvas.LineTo(FCurve[2].X, FCurve[2].Y);
    end;
```

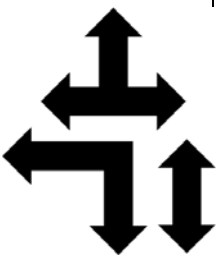
```
    // Рисуем саму кривую
    DrawCurve(Canvas);
    // Рисуем красные квадраты, показывающие точки, которые пользователь
    // может перемещать
    Canvas.Pen.Style := psSolid;
    Canvas.Pen.Width := 1;
    Canvas.Pen.Color := clRed;
    Canvas.Brush.Style := bsClear;
    for I := 0 to 3 do
      Canvas.Rectangle(FCurve[I].X – RectSize, FCurve[I].Y – RectSize,
        FCurve[I].X + RectSize + 1, FCurve[I].Y + RectSize + 1);
  end;
end;

// Функция PtNearPt возвращает True, если точка с координатами (X1, Y1)
// удалена от точки Pt по каждой из координат не более чем на RectSize
function TCurveForm.PtNearPt(X1, Y1: Integer; const Pt: TPoint): Boolean;
begin
  Result := (X1 >= Pt.X – RectSize) and
    (X1 <= Pt.X + RectSize) and
    (Y1 >= Pt.Y – RectSize) and
    (Y1 <= Pt.Y + RectSize);
end;

procedure TCurveForm.BtnEndClick(Sender: TObject);
begin
  if not FNewLine then
  begin
    DrawCurve(FBack.Canvas);
    FNewLine := True;
    Refresh;
  end;
end;
```

Размеры фонового растра устанавливаются равными размеру развернутого на весь экран окна. Таким образом, если уменьшить окно, то те завершенные кривые, которые окажутся за его пределами, не пропадут — они вновь будут видимы, если увеличить размеры окна. Однако в программе не предусмотрено, что система позволяет пользователю менять разрешение экрана. Это можно учесть, добавив реакцию на сообщение `WM_DISPLAYCHANGE` и меняя в нем размеры фонового рисунка.

ГЛАВА 2



Использование сокетов Delphi

Так как большинство современных компьютеров объединены в сети, то и задачи программирования передачи и получения данных по сети возникают часто. Существует множество высокоуровневых средств обмена, но иногда их бывает недостаточно, и тогда приходится использовать самые низкоуровневые средства сетевого программирования — сокеты. Однако существует целый ряд причин, по которым овладеть этой технологией непросто. Во-первых, она сложна сама по себе из-за большого количества возможностей. Во-вторых, существующие библиотеки отягощены совместимостью со старыми версиями, которые, с современной точки зрения, не всегда развивались по правильному пути. В-третьих, у программистов на Delphi возникают дополнительные трудности, связанные как с тем, что сама реализация библиотеки сокетов ориентирована на язык C, так и с тем, что стандартный модуль `WinSock` почему-то "застрял" на первой версии этой библиотеки, в то время как начиная с Windows NT 4 существует более удобная вторая.

Несмотря на традиционно высокий интерес к сокетам, литературы, в которой бы эта библиотека была детально описана, очень мало. Если не считать книг, где сокеты упоминаются обзорно, автору известна только одна книга, посвященная сокетам в Windows [3]. Но она не имеет никакого отношения к Delphi и не учитывает специфику этого средства разработки. Естественно, что из-за недостатка информации у начинающих программистов часто возникают вопросы, связанные с использованием сокетов в Delphi. В данной главе мы попытаемся ответить на эти вопросы.

Изложение материала ориентировано на человека, который не только не сталкивался с сокетами раньше, но и вообще имеет смутные представления об устройстве и функционировании компьютерных сетей. Поэтому мы будем обзорно рассматривать и общетеоретические темы, которые необходимо знать, чтобы эффективно программировать сетевое взаимодействие, так что

никакая предварительная подготовка в этих вопросах от читателя не требуется.

Глава разбита на две части. Первая посвящена *стандартным сокетам*, а вторая — *сокетам Windows*. Термины достаточно условны и нуждаются в дополнительном пояснении. Строго говоря, стандартными называются *сокеты Беркли* (Berkley sockets), разработанные в университете Беркли для системы Unix. Как это ни парадоксально звучит, но сокеты Беркли появились до появления компьютерных сетей. Изначально они предназначались для взаимодействия между процессами в системе и только позже были приспособлены для TCP/IP. Работа с сокетами Беркли сделана максимально похожей на работу с файлами в Unix. В частности, для отправки и получения данных используются те же функции, что и для файлового ввода-вывода.

Сокеты в Windows не полностью совместимы с сокетами Беркли (например, для них предусмотрены специальные функции отправки и получения данных, переопределены некоторые типы данных и т. п.). Но возможности работы с сокетами в Windows можно разделить на две части: то, что укладывается в идеологию сокетов Беркли, хотя и реализовано несколько иначе, и то, что является специфичным для Windows. Ту часть реализации сокетов Windows, которая по функциональности соответствует сокетам Беркли, мы будем называть стандартными сокетами, а сокетами Windows (Windows sockets) — специфичные для Windows расширения.

Ориентироваться мы будем только на два протокола передачи данных — TCP и UDP. Хотя библиотека сокетов поддерживает и другие протоколы, но эти два, во-первых, применяются наиболее часто, а во-вторых, именно на них во многом ориентированы стандартные сокеты. Поэтому здесь мы не будем касаться особенностей работы функций библиотеки сокетов, которые проявляются только для протоколов, отличных от TCP и UDP. Объем информации, с которой придется познакомиться, и без того большой, а с другими протоколами легче ознакомиться потом, изучив TCP и UDP.

2.1. Стандартные сокеты

Сначала рассмотрим классические методы работы с сокетами, которые не учитывают ни существования окон и оконных сообщений, ни возможности распараллеливания работы программы на несколько нитей. Это, впрочем, не означает, что программа, использующая эти методы, должна быть безоконной и однонитевой, оконные и многонитевые программы есть среди примеров этого раздела. Просто приспосабливать стандартные сокеты к окнам и распараллеливанию приходится вручную, а не за счет средств самой библиотеки. Тем не менее из-за своей простоты стандартные сокеты нередко оказываются более удобными, чем сокеты Windows, даже в оконных приложениях.

2.1.1. Соглашения об именах

Первые библиотеки сокетов писались на языке C. В этом языке идентификаторы чувствительны к регистру символов, т. е., например, `SOCKET`, `Socket` и `socket` — это разные идентификаторы. Исторически сложилось, что имена встроенных в C типов данных пишутся в нижнем регистре, имена определенных в программе типов, макроопределений и констант — в верхнем, а имена функций — в смешанном (прописные буквы выделяют начала слов, например, `GetWindowText`). Разработчики библиотеки сокетов несколько отошли от этих правил: имена всех стандартных сокетных функций пишутся в нижнем регистре. И хотя мы и программируем на Паскале, нечувствительном к регистру символов, все же будем придерживаться этой традиции, пусть это не очень удобно, зато не будет расхождений с другими источниками.

Чувствительность C к регистру символов создает некоторые проблемы при переносе библиотек, написанных на этом языке, в Delphi. Это связано с тем, что разные объекты могут иметь имена, различающиеся только регистром символов. В частности, есть тип `SOCKET` и функция `socket`. Сохранить эти имена в Delphi невозможно. Чтобы избежать этой проблемы, разработчики Delphi при переносе библиотек к имени типа добавляют букву "T", причем независимо от того, существуют ли у этого типа одноименные функции или нет. Так, типу `SOCKET` из C в Delphi соответствует `TSocket`. Имена функций остаются без изменений.

Ранее был упомянут термин *"макроопределение"*. Он может быть непонятен тем, кто не работал с языками C и C++, потому что в Delphi макроопределения отсутствуют. Нормальная последовательность трансляции программы в Delphi следующая: сначала компилятор создает объектный код, в котором вместо реальных адресов функций, переменных и т. п. стоят ссылки на них (на этапе компиляции эти адреса еще не известны). Затем компоновщик размещает объекты в памяти и заменяет ссылки реальными адресами. Так получается готовая к исполнению программа. В C/C++ трансляция включает в себя еще один этап: перед компиляцией текст программы модифицируется препроцессором, и компилятор получает уже несколько видоизмененный текст. Макроопределения, или просто *макросы*, — это директивы препроцессору, предписывающие ему, как именно нужно менять текст программы. Макрос задает подмену: везде, где в программе встречается имя макроса, препроцессор изменяет его на тот текст, который задан при определении этого макроса. Определяются макросы с помощью директивы препроцессору `#define`.

В простейшем случае макросы используются для определения констант. Например, директива `#define SOMECONST 10` вынуждает препроцессор заменять

`SOMECONST` на 10. Для компилятора эта директива ничего не значит, идентификатора `SOMECONST` для него не существует. Он получит уже измененный препроцессором текст, в котором вместо `SOMECONST` будет 10. Допускается также создавать параметризованные макросы, которые изменяют текст программы по более сложным правилам.

Макросы позволяют в некоторых случаях существенно сократить программу и повысить ее читабельность. Тем не менее они считаются устаревшим средством, т. к. их использование может привести к существенным проблемам (обсуждение этого выходит за рамки данной книги). В современных языках от макросов отказываются. В частности, в C++ макросы поддерживаются в полном объеме, но использовать их не рекомендуется, т. к. есть более безопасные инструменты, решающие типичные для макросов задачи. В C# и Java макросы отсутствуют. Тем не менее в заголовочных файлах для системных библиотек Windows (в том числе и библиотеки сокетов) макросы широко применяются, т. к. требуется обеспечить совместимость с языком C. При портировании таких файлов в Delphi макросы без параметров заменяются константами, а макросы с параметрами — функциями (иногда один макрос приходится заменять несколькими функциями для разных типов данных).

2.1.2. Общие сведения о сокетах

Сокетом (от англ. *socket* — гнездо, розетка) называется специальный объект, создаваемый для отправки и получения данных через сеть. Отметим, что под термином "объект" в данном случае подразумевается не объект в терминах объектно-ориентированного программирования, а некоторая сущность, внутренняя структура которой скрыта от нас, и мы можем оперировать с ней только как с единым и неделимым (атомарным) объектом. Этот объект создается внутри библиотеки сокетов, а программист, работающий с данной библиотекой, получает уникальный номер (дескриптор) этого сокета. Конкретное значение этого дескриптора не несет для программиста никакой полезной информации и может быть использовано только для того, чтобы при вызове функции из библиотеки сокетов указать, с каким сокетом требуется выполнить операцию. В этом смысле тип `TSocket` полностью аналогичен дескрипторам окон, графических объектов и т. п., с которыми мы встречались в предыдущей главе.

Чтобы две программы могли общаться друг с другом через сеть, каждая из них должна создать сокет. Каждый сокет обладает двумя основными характеристиками: *протоколом* и *адресом*, к которым он привязан. Протокол задается при создании сокета и не может быть изменен впоследствии. Адрес сокета задается позже, но обязательно до того, как через сокет пойдут данные. В некоторых случаях привязка сокета к адресу может быть неявной.

Формат адреса сокета определяется конкретным протоколом. В частности, для протоколов TCP и UDP адрес состоит из IP-адреса сетевого интерфейса и номера порта.

Каждый сокет имеет два *буфера*: для входящих и для исходящих данных. При отправке данных они сначала помещаются в буфер исходящих, и лишь затем отправляются в фоновом режиме. Программа в это время продолжает свою работу. При получении данных сокет помещает их в буфер для входящих, откуда они затем могут извлекаться программой.

Сеть может связывать разные аппаратные платформы, поэтому требуется согласование форматов передаваемых данных, в частности форматов целых чисел. Двухбайтные целые числа хранятся в памяти в двух последовательно расположенных байтах. При этом возможны два варианта: в первом байте хранится младший байт числа, а во втором — старший, и наоборот. Способ хранения определяется аппаратной частью платформы. Процессоры Intel используют первый вариант, т. е. первым хранится младший байт, а другие процессоры (например, Motorola) — второй вариант. То же касается и четырехбайтных чисел: процессоры Intel хранят их, начиная с младшего байта, а некоторые другие процессоры — начиная со старшего. Сетевой формат представления таких чисел совпадает с форматом процессора Motorola, т. е. на платформах с процессором Intel необходимо переставлять байты при преобразовании чисел в сетевой формат.

Библиотека сокетов разрабатывалась для ОС Unix, в которой традиционно высоко ценилась переносимость между платформами, поэтому она содержит функции, позволяющие не задумываться о порядке байтов в числах: `ntohs`, `ntohl`, `htons` и `htonl`. Первая буква в названии этих функций показывает, в каком формате дано исходное число (n — Network — сетевой формат, h — Host — формат платформы), четвертая буква — формат результата, последняя буква — разрядность (s — Short — двухбайтное число, l — Long — четырехбайтное число). Например, функция `htons` принимает в качестве параметра число типа `u_short` (Word) в формате платформы и возвращает то же число в сетевом формате. Реализация этих функций для каждой платформы своя: где-то они переставляют байты, где-то они возвращают в точности то число, которое было им передано. Благодаря этим функциям программы становятся переносимыми. Хотя для программиста на Delphi вопросы переносимости не столь актуальны, приходится прибегать к этим функциям хотя бы потому, что байты переставлять нужно, а никакого более удобного способа для этого не существует.

2.1.3. Сетевые протоколы. Семиуровневая модель OSI

Сетевым протоколом называется набор соглашений, следование которым позволяет обеим сторонам одинаково интерпретировать принимаемые и отправляемые данные. Сетевой протокол можно сравнить с языком: два человека понимают друг друга тогда, когда говорят на одном языке. Причем если люди, говорящие на похожих, но немного разных языках, все же могут понимать друг друга, то компьютеры для нормального обмена данными должны поддерживать в точности одинаковый протокол.

Для установления взаимодействия между компьютерами должен быть согласован целый ряд вопросов, начиная от напряжения в проводах и заканчивая форматом пакетов. Реализуются эти соглашения на разных уровнях, поэтому логичнее иметь не один протокол, описывающий все и вся, а набор протоколов, каждый из которых охватывает только вопросы одного уровня. Организация *Open Software Interconnection* (OSI) предложила разделить все вопросы, требующие согласования, на семь уровней. Это разделение известно *как семиуровневая модель OSI*.

Семейство протоколов, реализующих различные уровни, называется *стеком протоколов*. Стеки протоколов не всегда точно следуют модели OSI, некоторые протоколы решают вопросы, связанные сразу с несколькими уровнями.

Первый уровень в модели OSI называется *физическим*. На нем согласовываются физические, электрические и оптические параметры сети: напряжение и форма импульсов, кодирующих 0 и 1, какой разъем используется и т. п.

Второй уровень носит название *канального*. На этом уровне решаются вопросы конфигурации сети (шина, звезда, кольцо и т. п.), приема и передачи кадров, допустимости и методов разрешения коллизий (ситуаций, когда сразу два компьютера пытаются передать данные).

Третий уровень — *сетевой*. Здесь определяется, как адресуются компьютеры. Большинство сетей используют широковещательный способ передачи: пакет, переданный одним компьютером, получают все остальные. Протокол сетевого уровня описывает критерии, на основании которых каждый компьютер может выбирать из сети только те пакеты, которые предназначены ему, и игнорировать все остальные. На этом же уровне определяется, как пакеты проходят через маршрутизатор.

Четвертый уровень называется *транспортным*. На этом уровне единый физический поток данных разбивается на независимые логические потоки. Это позволяет нескольким программам независимо друг от друга использовать сеть, не опасаясь, что их данные смешаются. Кроме того, на транспортном

уровне решаются вопросы, связанные с подтверждением доставки пакета и упорядочиванием пакетов.

Пятый уровень известен как *уровень сессии*. Он определяет процедуру установления, завершения связи и ее восстановления после разрыва. Расписывается последовательность действий каждой стороны и пакеты, которые они должны друг другу отправить для инициализации и завершения связи. Определяются соглашения о том, как единый поток разбивается на логические пакеты.

Шестой уровень называется *уровнем представлений*. На этом уровне определяется то, в каком формате данные передаются по сети. Под этим подразумевается, в первую очередь, внутренняя структура пакета, а также способ представления данных разных типов. Например, для двух- и четырехбайтных целых чисел должен быть согласован порядок байтов, для логических величин — какие значения соответствуют `True`, какие — `False`, для строк — кодировка и способ задания конца строки и т. п.

Седьмой уровень называется *уровнем приложений*. Соглашения этого уровня позволяют работать с ресурсами (файлами, принтерами и т. д.) удаленного компьютера как с локальными, осуществлять удаленный вызов процедур и т. п.

Чтобы получить данные через сеть, должны быть реализованы все уровни, за исключением, может быть, седьмого. Для каждого уровня должен быть определен свой протокол. В идеале механизмы взаимодействия между протоколами разных уровней должны иметь столь высокую степень абстракции, чтобы один протокол на любом из уровней можно было заменить любым другим протоколом того же уровня, не внося каких-либо изменений в выше- и нижележащие уровни.

2.1.4. Стек TCP/IP

Физический и канальный уровни полностью реализуются сетевой картой или модемом (или другим устройством, выполняющим ту же функцию) и ее драйвером. Здесь действительно достигнута настолько полная абстракция, что программист обычно не задумывается о том, какая используется сеть. Поэтому мы также не будем останавливаться на этих двух уровнях.

В реальной жизни не все протоколы, особенно старые, соответствуют модели OSI. Существует такое понятие, как стек протоколов — набор протоколов разных уровней, которые совместимы друг с другом. Эти уровни не всегда точно соответствуют тем, которые предлагает модель OSI, но определенное разделение задач на уровни в них присутствует. Здесь мы сосредоточимся на стеке протоколов, который называется *TCP/IP* (нередко можно услышать словосочетание "протокол TCP/IP" , что не совсем корректно: TCP/IP не про-

токол, а стек протоколов). Название этот стек получил по наименованию двух самых известных своих протоколов: TCP и IP.

Протокол сетевого уровня *IP* расшифровывается как Internet Protocol. Это название иногда ошибочно переводят как "протокол Интернета" или "протокол для Интернета". На самом деле, когда разрабатывался этот протокол, никакого Интернета еще и в помине не было, поэтому правильный перевод — межсетевой протокол. История появления этого протокола связана с особенностями работы сети Ethernet. Эта сеть строится по принципу шины, когда все компьютеры подключены, грубо говоря, к одному проводу. Если хотя бы два компьютера попытаются одновременно передавать данные по общей шине, возникнет неразбериха, поэтому все шинные сети строятся по принципу "один говорит — все слушают". Очевидно, что требуется какая-то защита от так называемых коллизий (ситуаций, когда два узла одновременно пытаются передавать данные).

Разные сети решают проблему коллизий по-разному. В промышленных сетях, например, обычно имеется маркер — специальный индикатор, который показывает, какому узлу разрешено сейчас передавать данные. Узел, называемый мастером, следит за тем, чтобы маркер вовремя передавался от одного узла к другому. Маркер исключает возможность возникновения коллизий. Ethernet же является одноранговой сетью, в которой нет мастера, поэтому в ней реализован другой подход: коллизии допускаются, но существует механизм их разрешения, заключающийся в том, что, во-первых, узел не начинает передачу данных, если видит, что другой узел уже что-то передает, а во-вторых, если два узла одновременно пытаются начать передачу, то оба прекращают попытку и повторяют ее через случайный промежуток времени. У кого этот промежуток окажется меньше, тот и захватит сеть (или за этот промежуток времени сеть будет захвачена кем-то еще).

При большом числе компьютеров, сидящих на одной шине, коллизии становятся слишком частыми, и производительность сети резко падает. Для борьбы с этим служат специальные устройства — *маршрутизаторы* — специализированные узлы, подключенные одновременно к нескольким сетям. Пока остальные узлы каждой из этих сетей взаимодействуют только между собой, маршрутизатор никак себя не проявляет, и эти сети существуют независимо друг от друга. Но если компьютер из одной сети посылает пакет компьютеру другой сети, этот пакет получает маршрутизатор и переправляет его в ту сеть, в которой находится адресат, или в которой находится другой маршрутизатор, способный передать этот пакет адресату.

На канальном уровне существует адресация узлов, основанная на так называемом *MAC-адресе* сетевой карты (MAC — это сокращение Media Access Control). Этот адрес является уникальным номером карты, присвоенной ей производителем. Очевидно неудобство такого способа адресации, т. к. по

MAC-адресу невозможно определить положение компьютера в сети, т. е. выяснить, куда направлять пакет. Кроме того, при замене сетевой карты меняется адрес компьютера, что также не всегда удобно. Поэтому на сетевом уровне определяется собственный способ адресации, не связанный с аппаратными особенностями узла. Отсюда следует, что маршрутизатор должен понимать протокол сетевого уровня, чтобы принимать решение о передаче пакета из одной сети в другую, а протокол, в свою очередь, должен учитывать наличие маршрутизаторов в сети и предоставлять им необходимую информацию. IP был одним из первых протоколов сетевого уровня, который решал такую задачу, и с его помощью стала возможной передача пакетов между сетями. Поэтому он и получил название межсетевого протокола. Впрочем, название прижилось: в некоторых статьях MSDN сетевой уровень (network layer) называется межсетевым уровнем (internet layer). В протоколе IP, в частности, вводится важный параметр для каждого пакета: максимальное число маршрутизаторов, которое он может пройти, прежде чем попадет к адресату (этот параметр носит не совсем удачное название *TTL* — Time To Live, время жизни). Это позволяет защититься от бесконечного блуждания пакетов по сети.

Примечание

Здесь следует заметить, что сеть Ethernet ушла далеко вперед по сравнению с моментом создания протокола IP и теперь организована сложнее, поэтому не следует думать, что в предыдущих абзацах изложены все принципы работы этой сети (это выходит за рамки данной книги). Тем не менее протокол IP по-прежнему используется, а компьютеры по-прежнему видят в сети не только свои, но и чужие пакеты. На этом основана работа так называемых *снифферов* — программ, позволяющих одному компьютеру читать пакеты, пересылаемые между двумя другими компьютерами.

Для адресации компьютера протокол IP использует уникальное четырехбайтное число, называемое *IP-адресом*. Впрочем, более распространена форма записи этого числа в виде четырех однобайтных значений. Система назначения этих адресов довольно сложна и призвана оптимизировать работу маршрутизаторов, обеспечить прохождение широковещательных пакетов только внутри определенной части сети и т. п. Мы здесь не будем подробно останавливаться на этом, потому что в правильно настроенной сети программисту не нужно знать всех этих тонкостей: достаточно помнить, что каждый узел имеет уникальный IP-адрес, для которого принята запись в виде четырех цифровых полей, разделенных точками, например, 192.168.200.217. Также следует знать, что адреса из диапазона 127.0.0.1—127.255.255.255 задают так называемый *локальный узел*: через эти адреса могут связываться программы, работающие на одном компьютере. Таким образом, обеспечивается прозрачность местонахождения адресата. Кроме того, один компьютер может иметь несколько IP-адресов, которые могут использоваться для одного и того же или разных сетевых интерфейсов.

Кроме IP, в стеке TCP/IP существует еще несколько протоколов — ICMP, IGMP и ARP, — решающих задачи сетевого уровня. Эти протоколы не являются полноценными и не могут заменить IP. Они служат только для решения некоторых частных задач.

Протокол *ICMP* (Internet Control Message Protocol — протокол межсетевых управляющих сообщений) обеспечивает диагностику связи на сетевом уровне. Многим знакома утилита ping, позволяющая проверить связь с удаленным узлом. В основе ее работы лежат специальные запросы и ответы, определяемые в рамках протокола ICMP. Кроме того, этот же протокол определяет сообщения, которые получает узел, отправивший IP-пакет, если этот пакет по каким-то причинам не доставлен.

Протокол называется надежным (reliable), если он гарантирует, что пакет будет либо доставлен, либо отправивший его узел получит уведомление о том, что доставка невозможна. Кроме того, надежный протокол должен гарантировать, что пакеты доставляются в том же порядке, в каком они отправлены, и дублирования сообщений не происходит. Протокол IP в чистом виде не является надежным протоколом, т. к. в нем вообще не предусмотрены средства уведомления узла о проблемах с доставкой пакета. Добавление ICMP также не делает IP надежным, т. к. ICMP-пакет является частным случаем IP-пакета, и также может не дойти до адресата, поэтому возможны ситуации, когда пакет не доставлен, а отправитель об этом не подозревает.

Протокол *IGMP* (Internet Group Management Protocol — протокол управления межсетевыми группами) предназначен для управления группами узлов, которые имеют один групповой IP-адрес. Отправку пакета по такому адресу можно рассматривать как нечто среднее между адресной и широковещательной рассылкой, т. к. такой пакет будет получен сразу всеми узлами, входящими в группу.

Протокол *ARP* (Address Resolution Protocol — протокол разрешения адресов) необходим для установления соответствия между IP- и MAC-адресами. Каждый узел имеет таблицу соответствия. Исходящий пакет содержит два адреса узла: MAC-адрес для канального уровня и IP-адрес для сетевого. Отправляя пакет, узел находит в своей таблице MAC-адрес, соответствующий IP-адресу получателя, и добавляет его к пакету. Если в таблице такой адрес не найден, отправляется широковещательное сообщение, формат которого определяется протоколом ARP. Получив такое сообщение, узел, чей IP-адрес соответствует искомому, отправляет ответ, в котором указывает свой MAC-адрес. Этот ответ также широковещательный, поэтому его получают все узлы, а не только отправивший запрос, и все узлы обновляют свои таблицы соответствия.

Строго говоря, обеспечение правильного функционирования протоколов сетевого уровня — задача администратора системы, а не программиста. В своей

работе программист чаще всего сталкивается с более высокоуровневыми протоколами и не интересуется деталями реализации сетевого уровня.

Протоколами транспортного уровня в стеке TCP/IP являются протоколы TCP и UDP. Строго говоря, они решают не только задачи транспортного уровня, но и небольшую часть задач уровня сессии. Тем не менее они традиционно называются транспортными. Эти протоколы мы рассмотрим детально в следующих разделах.

Уровни сессии, представлений и приложений в стеке TCP/IP не разделены: протоколы HTTP, FTP, SMTP и т. д., входящие в этот стек, решают задачи всех трех уровней. Мы здесь не будем рассматривать эти протоколы, потому что при использовании сокетов они в общем случае не нужны: программист сам определяет формат пакетов, отправляемых с помощью TCP или UDP.

Новички нередко думают, что фраза "программа поддерживает соединение через TCP/IP" полностью описывает то, как можно связаться с программой и получить данные. На самом деле необходимо знать формат пакетов, которые эта программа может принимать и отправлять, т. е. должны быть согласованы протоколы уровня сессии и представлений. Гибкость сокетов дает программисту возможность самостоятельно определить этот формат, т. е., по сути дела, придумать и реализовать собственный протокол поверх TCP или UDP. И без описания этого протокола организовать обмен данными с программой невозможно.

2.1.5. Протокол UDP

Протокол *UDP* (User Datagram Protocol — протокол пользовательских дейтаграмм) встречается реже, чем его "одноклассник" TCP, но он проще для понимания, поэтому мы начнем изучение транспортных протоколов с него. Коротко UDP можно описать как ненадежный протокол без соединения, основанный на дейтаграммах. Теперь рассмотрим каждую из этих характеристик подробнее.

UDP не имеет никаких дополнительных средств управления пакетами по сравнению с IP. Это значит, что пакеты, отправленные с помощью UDP, могут теряться, дублироваться и менять порядок следования. В сети без маршрутизаторов ничего этого с пакетами почти никогда не происходит, и UDP может условно считаться надежным протоколом. Сети с маршрутизаторами строятся, конечно же, таким образом, чтобы подобные случаи происходили как можно реже, но полностью исключить их, тем не менее, нельзя. Происходит это из-за того, что передача данных может идти несколькими путями через разные маршрутизаторы. Например, пакет может пропасть, если короткий путь к удаленному узлу временно недоступен, а в длинном приходится пройти больше маршрутизаторов, чем это разрешено. Дублироваться пакеты мо-

гут, если они ошибочно передаются двумя путями, а порядок следования может изменяться, если пакет, посланный первым, идет по более длинному пути, чем пакет, посланный вторым.

Все сказанное отнюдь не означает, что на основе UDP нельзя построить надежный обмен данными, просто заботу об этом должно взять на себя само приложение. Каждый исходящий пакет должен содержать порядковый номер, и в ответ на него должен приходить специальный пакет — квитанция, которая уведомляет отправителя, что пакет доставлен. При отсутствии квитанции пакет высылается повторно (для этого необходимо ввести тайм-ауты на получение квитанции). Принимающая сторона по номерам пакетов восстанавливает их исходный порядок.

UDP не поддерживает соединение. Это означает, что при использовании этого протокола можно в любой момент отправить данные по любому адресу без необходимости каких-либо предварительных действий, направленных на установление связи с адресатом. Это напоминает процесс отправки обычного письма: на нем пишется адрес, и оно опускается в почтовый ящик без каких-либо предварительных действий. Такой подход обеспечивает большую гибкость, но лишает систему возможности автоматической проверки исправности канала связи.

Дейтаграммами называются пакеты, которые передаются как единое целое. Каждый пакет, отправленный с помощью UDP, составляет одну дейтаграмму. Принятые дейтаграммы складываются в буфер принимающего сокета и могут быть получены только раздельно: за одну операцию чтения из буфера программа, использующая сокет, может получить только одну дейтаграмму. Если в буфере лежит несколько дейтаграмм, потребуется несколько операций чтения, чтобы прочитать все. Кроме того, одну дейтаграмму нельзя получить из буфера по частям: она должна быть прочитана целиком за одну операцию.

Чтобы данные, передаваемые разным сокетам, не перемешивались, каждый сокет должен получить уникальный в пределах узла номер от 0 до 65 535, называемый *номером порта*. При отправке дейтаграммы отправитель указывает IP-адрес и порт получателя, и система принимающей стороны находит сокет, привязанный к указанному порту, и помещает данные в его буфер. По сути дела, UDP является очень простой надстройкой над IP, все функции которой заключаются в том, что физический поток разделяется на несколько логических с помощью портов, и добавляется проверка целостности данных с помощью контрольной суммы (сам по себе протокол IP не гарантирует отсутствия искажений данных при передаче).

Максимальный размер одной дейтаграммы IP равен 65 535 байтам. Из них не менее 20 байтов занимает заголовок IP. Заголовок UDP имеет размер 8 байтов. Таким образом, максимальный размер одной дейтаграммы UDP составляет 65 507 байтов.

Типичная область применения UDP — программы, для которых потеря пакетов некритична. Например, некоторые сетевые 3D-игры в локальной сети используют UDP, т. к. очень часто посылают пакеты, информирующие о действиях игрока, и потеря одного пакета не приведет к существенным проблемам: следующий пакет доставит необходимые данные. Достоинства UDP — простота установления связи, возможность обмена данными с несколькими адресами через один сокет и отсутствие необходимости возобновлять соединение после разрыва связи. В некоторых задачах также очень удобно то, что дейтаграммы не смешиваются, и получатель всегда знает, какие данные были отправлены одной дейтаграммой, а какие — разными.

Еще одно преимущество UDP — возможность отправки широковещательных дейтаграмм. Для этого нужно указать широковещательный IP-адрес (обычно это 255.255.255.255, но в некоторых случаях допустимы адреса типа 192.168.100.225 для вещания в пределах сети 192.168.100.XX и т. п.), и такую дейтаграмму получат все сокеты в локальной сети, привязанные к заданному порту. Эту возможность нередко используют программы, которые заранее не знают, с какими компьютерами они должны связываться. Они посылают широковещательное сообщение и связываются со всеми узлами, которые распознали это сообщение и прислали на него соответствующий ответ. По умолчанию для широковещательных пакетов число маршрутизаторов, через которые они могут пройти (TTL), устанавливается равным нулю, поэтому такие пакеты не выходят за пределы подсети.

2.1.6. Протокол TCP

Протокол *TCP* (Transmission Control Protocol — протокол управления передачей) является надежным потоковым протоколом с соединением, т. е. полной противоположностью UDP. Единственное, что у этих протоколов общее, — это способ адресации: в TCP каждому сокету также назначается уникальный номер порта. Уникальность номера порта требуется только в пределах протокола: два сокета могут иметь одинаковые номера портов, если один из них работает через TCP, а другой — через UDP.

В TCP предусмотрены так называемые *хорошо известные* (well-known) порты, которые зарезервированы для нужд системы и не должны использоваться программами. Стандарт TCP определяет диапазон хорошо известных портов от 0 до 255, в Windows и в некоторых других системах этот диапазон расширен до 0—1023. Часть портов UDP тоже выделена для системных нужд, но зарезервированного диапазона в UDP нет. Кроме того, некоторые системные утилиты используют порты за пределами диапазона 0—1023. Полный список системных портов для TCP и UDP содержится в MSDN, в разделе Resource Kits/Windows 2000 Server Resource Kit/TCP/IP Core Networking/Appendixes/TCP and UDP Port Assignment.

Для отправки пакета с помощью TCP отправителю необходимо сначала установить соединение с получателем. После выполнения этого действия соединенные таким образом сокеты могут использоваться только для отправки сообщений друг другу. Если соединение разрывается (самой программой или из-за проблем в сети), эти сокеты уже непригодны для установления нового соединения: они должны быть уничтожены, а вместо них созданы новые сокеты.

Механизм соединения, принятый в TCP, подразумевает разделение ролей соединяемых сторон: одна из них пассивно ждет, когда кто-то установит с ней соединение, и называется *сервером*, другая самостоятельно устанавливает соединение и называется *клиентом*. Действия клиента по установке связи заключаются в следующем: создать сокет, привязать его к адресу и порту, вызвать функцию для установки соединения, передав ей адрес сервера. Если все эти операции выполнены успешно, то связь установлена, и можно начинать обмен данными. Действия сервера выглядят следующим образом: создать сокет, привязать его к адресу и порту, перевести в режим ожидания соединения и дождаться соединения. При соединении система создаст на стороне сервера специальный сокет, который будет связан с соединившимся клиентом, и обмениваться данными с подключившимся клиентом сервер будет через этот новый сокет. Старый сокет останется в режиме ожидания соединения, и другой клиент сможет к нему подключиться. Для каждого нового подключения будет создаваться новый сокет, обслуживающий только данное соединение, а исходный сокет будет по-прежнему ожидать соединения. Это позволяет нескольким клиентам одновременно соединяться с одним сервером, а серверу — не путаться в своих клиентах. Точное число клиентов, которые могут одновременно работать с сервером, в документации не приводится, но оно достаточно велико.

Установление такого соединения позволяет осуществлять дополнительный контроль прохождения пакетов. В рамках протокола TCP выполняется проверка доставки пакета, соблюдения очередности и отсутствия дублей. Механизмы обеспечения надежности достаточно сложны, и мы их здесь рассматривать не будем. Программисту для начала достаточно знать, что данные, переданные с помощью TCP, не теряются, не дублируются и доставляются в том порядке, в каком были отправлены. В противном случае отправитель получает сообщение об ошибке. Соединенные сокеты время от времени обмениваются между собой специальными пакетами, чтобы убедиться в наличии соединения.

Если из-за неполадок в сети произошел разрыв связи, при попытке отправить данные или прочитать их клиент получит отказ, а соединение будет разорвано. После этого клиент должен уничтожить сокет, создать новый и повторить подключение. Сервер также получает ошибку на сокете, обслуживающем

данное соединение, но существенно позже (эта задержка может достигать часа). При обнаружении ошибки сервер просто уничтожает сокет и ждет нового подключения от клиента. Возможна ситуация, когда клиент уже подключился заново и для него создан новый сокет, а старый сокет еще не закрыт. Это не является существенной проблемой — на старом сокете рано или поздно будет получена ошибка, и он будет закрыт. Тем не менее сервер может учитывать такую ситуацию и уничтожать старый сокет, не дожидаясь, пока на нем будет получена ошибка, если новое соединение с тем же клиентом уже установлено. На исходный сокет, находящийся в режиме ожидания подключения, физические разрывы связи никак не влияют, после восстановления связи никаких действий с ним проводить не нужно.

Если на клиентской стороне не удалось для нового сокета установить соединение с сервером с первого раза (из-за отсутствия связи или неработоспособности сервера), этот сокет не обязательно уничтожать: он может использоваться при последующих попытках установления связи неограниченное число раз, пока связь не будет установлена.

Протокол TCP называется потоковым потому, что он собирает входящие пакеты в один поток. В частности, если в буфере сокета лежат 30 байтов, принятые по сети, не существует возможности определить, были ли эти 30 байтов отправлены одним пакетом, 30 пакетами по 1 байту или еще как-либо. Гарантируется только то, что порядок байтов в буфере совпадает с тем порядком, в котором они были отправлены. Принимающая сторона также не ограничена в том, как она будет читать информацию из буфера: все сразу или по частям. Это существенно отличает TCP от UDP, в котором дейтаграммы не объединяются и не разбиваются на части.

Склеивание пакетов осуществляется не только принимающей, но и отправляющей стороной. Библиотека сокетов может придержать в выходном буфере то, что кладет программа, и потом отправить одним пакетом данные, которые программа складывала в буфер постепенно. И наоборот, данные большого объема могут быть отправлены по частям, поэтому возможна ситуация, когда принимающая сторона находит в своем буфере только часть сообщения, посланного своим визави. Это значит, что оставшаяся часть сообщения придет позже. Будут ли пакеты сливаться или разбиваться на части, зависит от пропускной способности и текущей загрузке сети, определяется это алгоритмом, который называется *алгоритм Нагля*.

TCP применяется там, где программа не хочет заботиться о проверке целостности данных. За отсутствие этой проверки приходится расплачиваться более сложной процедурой установления и восстановления связи. Если при использовании UDP сообщение не будет отправлено из-за проблем в сети или на удаленной стороне, никаких действий перед отправкой следующего сообщения выполнять не нужно и можно использовать тот же сокет. В случае же

TCP, как уже было сказано, необходимо сначала уничтожать старый сокет, затем создать новый и подключить его к серверу, и только потом можно будет снова отправлять сообщения. Другим недостатком TCP по сравнению с UDP является то, что через один TCP-сокет все пакеты отправляются только по одному адресу, в то время как через UDP-сокет разные пакеты могут быть отправлены по разным адресам. И наконец, TCP не позволяет рассылать широковещательные сообщения. Но, несмотря на эти неудобства, TCP применяется существенно чаще UDP, потому что автоматическая проверка целостности данных и гарантия их доставки является очень важным преимуществом. Кроме того, TCP гарантирует более высокую безопасность в Интернете (это связано с тем, что обеспечить безопасность при передаче данных легче при наличии соединения, а не в ситуации, когда пакеты могут передаваться от кого угодно кому угодно).

То, что TCP склеивает данные в один поток, не всегда удобно. Во многих случаях пакеты, приходящие по сети, обрабатываются отдельно, поэтому читать их из буфера желательно тоже по одному. Это просто сделать, если все пакеты имеют одинаковую длину. Но при различной длине пакетов принимающая сторона заранее не знает, сколько байтов нужно прочитать из буфера, чтобы получить ровно один пакет и ни байта больше. Чтобы обойти эту ситуацию, в пакете можно предусмотреть обязательный заголовок фиксированной длины, одно из полей которого хранит длину пакета. В этом случае принимающая сторона может читать пакет по частям: сначала заголовок известной длины, а потом тело пакета, размер которого стал известен благодаря заголовку. Другой способ разделения пакетов — вставка между ними заранее оговоренной последовательности байтов, которая не может появиться внутри пакета.

Но самое неудобное то, что пакеты не только склеиваются, но и разбиваются на части. Принимающая сторона может получить пакет меньшего размера, чем отправленный, если этот пакет был послан по частям, и на момент его чтения принимающей стороной еще не все части были получены. Тогда приходится повторять операцию чтения данных, пока не будет получено все, что нужно.

В отличие от UDP, в TCP данные, которые программа отправляет одной командой, могут разбиваться на части и отправляться несколькими IP-пакетами. Поэтому ограничение на длину данных, отправляемых за один раз, в TCP отсутствует (точнее, определяется доступными ресурсами системы). Количество данных, получаемое отправителем за одну операцию чтения, ограничено размером низкоуровнего буфера сокета и может быть разным в различных реализациях. Следует иметь в виду, что при переполнении буфера принимающей стороны протокол TCP предусматривает передачу отправляющей стороне сигнала, по которому она приостанавливает отправку, при-

чем этот сигнал прерывает всю передачу данных между этими двумя компьютерами с помощью TCP, т. е. это может повлиять и на другие программы. Поэтому желательно не допускать таких ситуаций, когда у принимающей стороны в буфере накапливается много данных.

2.1.7. Сетевые экраны

Сеть не только позволяет пересылать полезные данные, но и может служить путем проникновения вредоносных программ, несанкционированного доступа к данным и т. п. С этим, естественно, борются, и один из способов борьбы — *сетевые экраны* (они же брандмауэры, или firewalls). Мы здесь не будем детально знакомиться с ними, но затронем эту тему, потому что сетевые экраны могут повлиять на работоспособность наших примеров.

Сетевые экраны бывают аппаратными и программными. Их общий принцип действия заключается в проверке пакетов, идущих по сети, и блокировании тех из них, которые не удовлетворяют заданным критериям. Критерии могут быть различными и зависят от настройки конкретного сетевого экрана. Все пакеты делятся на входящие и исходящие. Для входящих UDP-сообщений обычно оставляют открытыми некоторые порты, а все сообщения, присланные на другие порты, отсекаются. Для исходящих сообщений тоже может быть задан набор портов, но обычно сетевые экраны осуществляют проверку по-другому: у них есть список приложений, которым разрешено отправлять исходящие UDP-сообщения, а пакеты, отправляемые другими приложениями, сетевой экран не пропускает.

Для протокола TCP настройки обычно задаются на уровне соединения, а не отдельного пакета. Составляется список портов, открытых для внешнего подключения. Если сервер использует порт не из этого набора, клиент не сможет к нему подключиться. Для исходящих подключений тоже составляется список программ, которым разрешено это делать, и, если клиент, отсутствующий в "белом" списке сетевого экрана, пытается подключиться к удаленному серверу, сетевой экран не допускает этого.

Примечание

Здесь описаны наиболее типичные способы локальной фильтрации пактов сетевым экраном. В каждом конкретном случае могут применяться другие правила.

При тестировании своих примеров или примеров из этой книги вы можете столкнуться с тем, что программы по непонятным причинам не могут обмениваться данными по сети, и это может объясняться работой сетевых экранов. Проконсультируйтесь у администратора сети насчет настроек сетевых экранов в вашей сети и согласуйте с ним возможность работы с теми или иными портами и запуска ваших приложений.

2.1.8. Создание сокета

До сих пор мы обсуждали только теоретические аспекты работы с сокетами. Далее будут рассматриваться конкретные функции, позволяющие осуществлять те или иные операции с сокетами. Эти функции экспортируются системной библиотекой wsock32.dll (а также библиотекой ws2_32.dll; взаимоотношение этих библиотек будет обсуждаться во втором разделе данной главы), для их использования в Delphi в раздел `uses` нужно добавить стандартный модуль `WinSock`. Полное формальное описание функций этого модуля здесь приводиться не будет (для этого есть MSDN), но для каждой функции будет дано описание, достаточно полное для понимания ее предназначения. Кроме того, мы будем также обращать внимание на некоторые моменты, которые в MSDN найти трудно. Тем не менее после знакомства с этим текстом настоятельно рекомендуется самостоятельно прочитать в MSDN описания всех упомянутых в нем функций.

Хотя ранее мы договорились, что будем обсуждать только стандартные сокеты, тем не менее, есть три функции, относящиеся к сокетам Windows, не познакомившись с которыми мы не сможем двигаться дальше. Это функции `WSAStartup`, `WSACleanup` и `WSAGetLastError` (префикс WSA означает Windows Sockets API и служит для именования большинства функций, относящихся к Windows-расширению библиотеки сокетов).

Функция `WSAStartup` предназначена для инициализации библиотеки сокетов. Эту функцию необходимо вызвать до вызова любой другой функции из этой библиотеки. Ее прототип имеет вид:

```
function WSAStartup(wVersionRequired: Word;
  var WSData: TWSAData): Integer;
```

Параметр `wVersionRequired` задает требуемую версию библиотеки сокетов. Младший байт задает основную версию, старший — дополнительную. Допустимы версии 1.0 ($0001), 1.1 ($0101), 2.0 ($0002) и 2.2 ($0202). Пока мы работаем со стандартными сокетами, принципиальной разницы между этими версиями нет, но версии 2.0 и выше пока лучше не использовать, т. к. модуль `WinSock` не рассчитан на их поддержку. Вопросы взаимоотношения библиотек и версий будут рассматриваться во втором разделе этой главы, а пока ограничимся версией 1.1.

Параметр `WSData` выходной, т. е. значение, которое имела переменная до вызова функции, игнорируется, а имеет смысл только то значение, которое эта переменная получит после вызова функции. Через этот параметр передается дополнительная информация о библиотеке сокетов. В большинстве случаев эти сведения не представляют никакого интереса, поэтому их можно игнорировать.

Нулевое значение, возвращаемое функцией, говорит об успешном завершении, в противном случае возвращается код ошибки. Обычно функция, завершившаяся с ошибкой, возвращает значение `SOCKET_ERROR`.

Функция `WSACleanup` завершает работу с библиотекой сокетов. Эта функция не имеет параметров и возвращает ноль в случае успешного завершения или код ошибки в противном случае.

Функцию `WSAStartup` достаточно вызвать один раз, даже в многонитевом приложении. В этом ее отличие от таких функций, как, например, `CoInitialize`, которая должна быть вызвана в каждой нити, использующей COM. Функцию можно вызывать повторно — в этом случае ее вызов не дает никакого эффекта, но для завершения работы с библиотекой сокетов функция `WSACleanup` должна быть вызвана столько же раз, сколько была вызвана `WSAStartup`.

Большинство функций библиотеки сокетов возвращают значение, позволяющее судить только об успешном или неуспешном завершении операции, но не дающее информации о том, какая именно ошибка произошла (если она произошла). Для получения сведений об ошибке служит функция `WSAGetLastError`, не имеющая параметров и возвращающая целочисленный код последней ошибки, произошедшей в библиотеке сокетов в данной нити. После неудачного завершения функции из библиотеки сокетов следует вызывать функцию `WSAGetLastError`, чтобы выяснить причину неудачи.

Забегая чуть вперед, отметим, что библиотека сокетов содержит стандартную функцию `getsockopt`, которая, кроме всего прочего, также позволяет получить информацию об ошибке. Однако она менее удобна, поэтому в тех случаях, когда не требуется совместимость с другими платформами, лучше прибегнуть к `WSAGetLastError`. К тому же, `getsockopt` возвращает ошибку, связанную с указанным сокетом, поэтому с ее помощью нельзя получить код ошибки, не связанной с конкретным сокетом.

Для создания сокета предусмотрена стандартная функция `socket` со следующим прототипом:

```
function socket(af, struct, protocol: Integer): TSocket;
```

Параметр `af` задает *семейство адресов* (address family). Этот параметр определяет, какой способ адресации (т. е., по сути дела, какой стек протоколов) будет использоваться для данного сокета. Для TCP/IP этот параметр должен быть равен `AF_INET`, для других стеков также есть соответствующие константы, которые можно посмотреть в файле WinSock.pas.

Параметр `struct` указывает тип сокета и может принимать одно из двух значений: `SOCK_STREAM` (для потоковых протоколов) и `SOCK_DGRAM` (для дейтаграммных протоколов).

Параметр protocol позволяет указать, какой именно протокол будет использоваться сокетом. Этот параметр можно оставить равным нулю — тогда будет выбран протокол по умолчанию, отвечающий условиям, заданным первыми двумя параметрами. Для стека TCP/IP потоковый протокол по умолчанию — TCP, дейтаграммный — UDP. В некоторых примерах можно увидеть, что значение третьего параметра равно IPPPROTO_IP. Значение этой константы равно 0, и ее использование только повышает читабельность кода, но приводит к тому же результату: будет выбран протокол по умолчанию. Если требуется протокол, отличный от протокола по умолчанию (например, в некоторых реализациях стека TCP/IP существует протокол *RDP* — Reliable Datagram Protocol, надежный дейтаграммный протокол), следует указать здесь соответствующую константу (для RDP это будет IPPROTO_RDP). Можно также явно задать TCP или UDP с помощью констант IPPROTO_TCP и IPPROTO_UDP соответственно.

Тип TSocket предназначен для хранения дескриптора сокета. Формально он совпадает с 32-битным беззнаковым целым типом, но об этом лучше не вспоминать, т. к. любые операции над значениями типа TSocket бессмысленны. Значение, возвращаемое функцией socket, следует сохранить в переменной соответствующего типа и затем использовать для идентификации сокета при вызове других функций. Если по каким-то причинам создание сокета невозможно, функция вернет значение INVALID_SOCKET. Причину ошибки можно узнать с помощью функции WSAGetLastError.

Сокет, созданный с помощью функции socket, не привязан ни к какому адресу. Привязка осуществляется с помощью функции bind, имеющей следующий прототип:

```
function bind(s: TSocket; var addr: TSockAddr;
  namelen: Integer): Integer;
```

Первый параметр этой функции — дескриптор сокета, который привязывается к адресу. Здесь, как и в остальных подобных случаях, требуется передать значение, которое вернула функция socket. Второй параметр содержит адрес, к которому требуется привязать сокет, а третий — длину структуры, содержащей адрес.

Функция bind предназначена для сокетов, реализующих разные протоколы из разных стеков, поэтому кодирование адреса в ней сделано достаточно универсальным. Впрочем, следует отметить, что разработчики модуля WinSock для Delphi выбрали не лучший способ перевода прототипа этой функции на Паскаль, поэтому универсальность в значительной мере утрачена. В оригинале прототип функции bind имеет следующий вид:

```
int bind(SOCKET s, const struct sockaddr FAR* name, int namelen);
```

Видно, что второй параметр — это указатель на структуру sockaddr. Однако компилятор C/C++ позволяет при вызове функции в качестве параметра передать указатель на любую другую структуру, если будет выполнено явное приведение типов. Для каждого семейства адресов предусмотрена своя структура, и в качестве фактического параметра передается указатель на эту структуру. Если бы авторы модуля WinSock описали второй параметр как параметр-значение типа указатель, можно было бы поступать точно так же. Однако они описали этот параметр как параметр-переменную. В результате на двоичном уровне ничего не изменилось: и там, и там в стек помещается указатель. Однако компилятор при вызове функции bind не допустит использование никакой другой структуры, кроме TSockAddr, а эта структура не универсальна и удобна, по сути дела, только при использовании стека TCP/IP. В других случаях наилучшим решением будет самостоятельно импортировать функцию bind из wsock32.dll с нужным прототипом. При этом придется импортировать и некоторые другие функции, работающие с адресами. Впрочем, мы здесь ограничиваемся только протоколами TCP и UDP, поэтому больше останавливаться на этом вопросе не будем.

Примечание

На самом деле существует способ передать в функцию bind с таким прототипом параметр addr любого типа, совместимого с этой функцией. Если A — некая переменная типа, отличающегося от TSockAddr, то передать в качестве параметра-переменной ее можно так: PSockAddr(@A)^. Однако подобные низкоуровневые операции программу не украшают.

В стандартной библиотеке сокетов (т. е. в заголовочных файлах для этой библиотеки) полагается, что адрес кодируется структурой sockaddr длиной 16 байтов, причем первые два байта этой структуры кодируют *семейство протоколов*, а смысл остальных зависит от этого семейства. В частности, для стека TCP/IP семейство протоколов задается константой PF_INET. (Ранее мы уже встречались с термином "семейство адресов" и константой AF_INET. В ранних версиях библиотеки сокетов семейства протоколов и семейства адресов были разными понятиями, но затем эти понятия слились в одно, и константы AF_XXX и PF_XXX стали взаимозаменяемыми.) Остальные 14 байтов структуры sockaddr занимает массив типа char (напомним, что тип char в C/C++ соответствует одновременно двум типам Delphi: Char и ShortInt). В принципе, в стандартной библиотеке сокетов предполагается, что структура, задающая адрес, всегда имеет длину 16 байтов, но на всякий случай предусмотрен третий параметр функции bind, который хранит длину структуры. В сокетах Windows длина структуры может быть любой (это зависит от протокола), так что этот параметр, в принципе, может пригодиться.

Ранее уже упоминалось, что неструктурированное представление адреса в виде массива из 14 байтов бывает неудобно, и поэтому для каждого семейст-

ва протоколов предусмотрена своя структура, учитывающая особенности адреса. В частности, для протоколов стека TCP/IP используется структура sockaddr_in, размер которой также составляет 16 байтов. Из них задействовано только восемь: два для кодирования семейства протоколов, четыре для IP-адреса и два — для порта. Оставшиеся 8 байтов должны содержать нули.

Можно было бы предположить, что типы TSockAddr и TSockAddrIn, описанные в модуле WinSock, соответствуют структурам sockaddr и sockaddr_in, однако это не так. На самом деле эти типы описаны следующим образом (листинг 2.1).

Листинг 2.1. Типы TSockAddr и TSockAddrIn

```
SunB = packed record
  s_b1, s_b2, s_b3, s_b4: u_char;
end;

SunW = packed record
  s_w1, s_w2: u_short;
end;

in_addr = record
  case Integer of
    0: (S_un_b: SunB);
    1: (S_un_w: SunW);
    2: (S_addr: u_long);
end;
TInAddr = in_addr;

sockaddr_in = record
  case Integer of
    0: (sin_family: u_short;
        sin_port: u_short;
        sin_addr: TInAddr;
        sin_zero: array[0..7] of Char);
    1: (sa_family: u_short;
        sa_data: array[0..13] of Char);
end;

TSockAddrIn = sockaddr_in;
TSockAddr = sockaddr_in;
```

Таким образом, типы TSockAddr и TSockAddrIn — это синонимы типа sockaddr_in (но не того sockaddr_in, который имеется в стандартной библио-

теке сокетов, а типа sockaddr_in, описанного в модуле WinSock). А тип sockaddr_in из WinSock является вариантной записью, и в случае 0 соответствует типу sockaddr_in из стандартной библиотеки сокетов, а в случае 1 — типу sockaddr из этой же библиотеки. Вот такая несколько запутанная ситуация, хотя на практике все выглядит не так страшно.

Примечание

Из названия типов можно сделать вывод, что тип u_short — это Word, а u_long — Cardinal. На самом деле u_short — это действительно Word, а вот u_long — это LongInt. Сложно сказать, почему выбран знаковый тип там, где предполагается беззнаковый. Видимо, это осталось в наследство от старых версий Delphi, которые не поддерживали тип Cardinal в полном объеме. Кстати, тип u_char — это Char, а не Byte.

Перейдем, наконец, к более практически важному вопросу: какими значениями следует заполнять переменную типа TSockAddr, чтобы при передаче ее в функцию bind сокет был привязан к нужному адресу. Так как мы ограничиваемся рассмотрением протоколов TCP и UDP, нас не интересует та часть вариантной записи sockaddr_in, которая соответствует случаю 1, т. е. мы будем рассматривать только те поля этой структуры, которые имеют префикс sin.

Поле sin_zero, очевидно, должно содержать массив нулей. Это то самое поле, которое не несет никакой смысловой нагрузки и служит только для увеличения размера структуры до стандартных 16 байтов. Поле sin_family должно иметь значение PF_INET. В поле sin_port записывается номер порта, к которому привязывается сокет. Номер порта должен быть записан в сетевом формате, т. е. здесь необходимо прибегнуть к функции htons, чтобы из привычной нам записи номера порта получить число в требуемом формате. Номер порта можно оставить нулевым, тогда система выберет для сокета свободный порт с номером от 1024 до 5000.

IP-адрес для привязки сокета задается полем sin_addr, которое имеет тип TInAddr. Этот тип сам является вариантной записью, которая отражает три способа задания IP-адреса: в виде 32-битного числа, в виде двух 16-битных чисел или в виде четырех 8-битных чисел. На практике чаще всего встречается формат в виде четырех 8-битных чисел, реже — в виде 32-битного числа. Задание адресов в виде двух 16-битных чисел или двух 8-битных и одного 16-битного числа относится к очень редко встречающейся экзотике.

Пусть у нас есть переменная Addr типа TSockAddr, и нам требуется в ее поле sin_addr записать адрес 192.168.200.217. Это можно сделать так, как показано в листинге 2.2.

Листинг 2.2. Прямое присваивание IP-адреса

```
Addr.sin_addr.S_un_b.s_b1 := 192;
Addr.sin_addr.S_un_b.s_b2 := 168;
Addr.sin_addr.S_un_b.s_b3 := 200;
Addr.sin_addr.S_un_b.s_b4 := 217;
```

Существует альтернатива такому присвоению четырех полей по отдельности — функция `inet_addr`. Эта функция в качестве входного параметра принимает строку, в которой записан IP-адрес, и возвращает этот IP-адрес в формате 32-битного числа. С использованием функции `inet_addr` приведенный в листинге 2.2 код можно переписать так:

```
Addr.sin_addr.S_addr := inet_addr('192.168.200.217');
```

Функция `inet_addr` выполняет простой парсинг строки и не проверяет, существует ли такой адрес на самом деле. Поля адреса можно задавать в десятичном, восьмеричном и шестнадцатеричном форматах. Восьмеричное поле должно начинаться с нуля, шестнадцатеричное — с "0x". Приведенный адрес можно записать в виде "0300.0250.0310.0331" (восьмеричный) или "0xC0.0xA8.0xC8.0xD9" (шестнадцатеричный). Допускается также смешанный формат записи, в котором разные поля заданы в разных системах исчисления. Функция `inet_addr` поддерживает также менее распространенные форматы записи IP-адреса в виде трех полей. Подробнее об этом можно прочитать в MSDN.

Примечание

Если строка, переданная функции `inet_addr`, не распознается как допустимый адрес, то функция возвращает значение `INADDR_NONE`. До пятой версии Delphi включительно эта константа имеет значение `$FFFFFFFF`, начиная с шестой версии — значение `-1`. Поле `S_addr` имеет тип `u_long`, который, как отмечалось, соответствует типу `LongInt`, т. е. является знаковым. Сравнение знакового числа с беззнаковым обрабатывается компилятором особым образом (подробнее об этом написано в *разд. 3.1.4*), поэтому, если просто сравнить `S_addr` и `INADDR_NONE` в старых версиях Delphi, получится неправильный результат. Перед сравнением константу `INADDR_NONE` следует привести к типу `u_long`, тогда операция выполнится правильно. В шестой и более поздних версиях Delphi приведение не обязательно, но оно не мешает, поэтому в целях совместимости со старыми версиями его тоже лучше выполнять.

В библиотеке сокетов предусмотрена константа `INADDR_ANY`, позволяющая не указывать явно адрес в программе, а оставить его выбор на усмотрение системы. Для этого полю `sin_addr.S_addr` следует присвоить значение `INADDR_ANY`. Если IP-адрес компьютеру не назначен, то при использовании этой константы сокет будет привязан к локальному адресу 127.0.0.1. Если

компьютеру назначен один IP-адрес, сокет будет привязан к этому адресу. Если компьютеру назначено несколько IP-адресов, то будет выбран один из них, причем сама привязка при этом отложится до установления соединения (в случае TCP) или до первой отправки данных через сокет (в случае UDP). Выбор конкретного адреса при этом зависит от того, какой адрес имеет удаленная сторона.

Итак, резюмируем все сказанное. Пусть у нас есть сокет S, который нужно привязать, например, к адресу 192.168.200.217 и порту 3320. Для этого следует выполнить код листинга 2.3.

Листинг 2.3. Привязка сокета к конкретному адресу

```
Addr.sin_family := PF_INET;
Addr.sin_addr.S_addr := inet_addr('192.168.200.217');
Addr.sin_port := htons(3320);
FillChar(Addr.sin_zero, SizeOf(Addr.sin_zero), 0);
if bind(S, Addr, SizeOf(Addr)) = SOCKET_ERROR then
begin
  // какая-то ошибка, анализируем с помощью WSAGetLastError
end;
```

`FillChar` — это стандартная процедура Паскаля, заполняющая некоторую область памяти заданным значением. В данном случае мы применяем ее для заполнения нулями поля `sin_zero`. Для этой же цели пригодна функция Windows API `ZeroMemory`. В примерах на C/C++ обычно используется функция `memset`.

Теперь рассмотрим другой случай: пусть выбор адреса и порта можно оставить на усмотрение системы. Тогда код будет выглядеть так, как показано в листинге 2.4.

Листинг 2.4. Привязка сокета к адресу, выбираемому системой

```
Addr.sin_family := PF_INET;
Addr.sin_addr.S_addr := INADDR_ANY;
Addr.sin_port := 0;
FillChar(Addr.sin_zero, SizeOf(Addr.sin_zero), 0);
if bind(S, Addr, SizeOf(Addr)) = SOCKET_ERROR then
begin
  // какая-то ошибка, анализируем с помощью WSAGetLastError
end;
```

В случае TCP сервер сам не является инициатором подключения, но может работать с любым подключившимся клиентом, какой бы у него ни был адрес.

Для сервера принципиально, какой порт он будет использовать — если порт не определен заранее, клиент не будет знать, куда подключаться. Поэтому номер порта является важным признаком для сервера. (Иногда, впрочем, встречаются серверы, порт которых заранее неизвестен, но в таких случаях всегда существует другой канал передачи данных, позволяющий клиенту до подключения узнать, какой порт задействован в данный момент сервером.) С другой стороны, клиенту обычно непринципиально, какой порт будет у его сокета, поэтому чаще всего серверу назначается фиксированный порт, а клиент оставляет выбор системе.

Протокол UDP не поддерживает соединение, но при его применении часто одно приложение тоже можно условно назвать сервером, а другое — клиентом. Сервер создает сокет и ждет, когда кто-нибудь что-нибудь пришлет и высылает что-то в ответ, а клиент сам отправляет что-то куда-то. Поэтому, как и в случае TCP, сервер должен использовать фиксированный порт, а клиент может выбирать любой свободный.

Если у компьютера только один IP-адрес, то выбор адреса для сокета и клиент, и сервер могут доверить системе. Если компьютер имеет несколько интерфейсов к одной сети, каждый со своим IP-адресом, то выбор конкретного адреса в большинстве случаев также непринципиален и может быть оставлен на усмотрение системы. Проблемы возникают, когда у компьютера несколько сетевых интерфейсов, каждый из которых включен в свою сеть. В этом случае выбор того или иного IP-адреса для сокета привязывает его к одной из сетей, и только к одной. Поэтому нужно принять меры для того, чтобы сокет оказался привязан к той сети, в которой находится его адресат.

Ранее мы уже говорили, что в системах с несколькими сетевыми картами привязка сокета к адресу в том случае, когда его выбор доверен системе, может осуществляться не во время выполнения функции `bind`, а позже, когда системе станет понятно, зачем используется этот сокет. Например, когда TCP-клиент осуществляет подключение к серверу, система по адресу этого сервера определяет, через какую карту должен идти обмен, и выбирает соответствующий адрес. То же самое происходит с UDP-клиентом: когда он отправляет первую дейтаграмму, система по адресу получателя определяет, к какой карте следует привязать сокет. Поэтому клиент и в данном случае может оставить выбор адреса на усмотрение системы. С серверами все несколько сложнее. Система привязывает сокет UDP-сервера к адресу, он ожидает получения пакета. В этот момент система не имеет никакой информации о том, с какими узлами будет вестись обмен через данный сокет, и может выбрать не тот адрес, который нужен. Поэтому сокеты UDP-серверов, работающих в подобных системах, должны явно привязываться к требуемому адресу. Сокеты TCP-серверов, находящиеся в режиме ожидания и имеющие адрес `INADDR_ANY`, допускают подключение к ним по любому сетевому интерфейсу, который имеется в системе. Сокет, который создается таким сервером

при подключении клиента, будет автоматически привязан к IP-адресу того сетевого интерфейса, через который осуществляется взаимодействие с подключившимся клиентом. Таким образом, сокеты, созданные для взаимодействия с разными клиентами, могут оказаться привязанными к разным адресам.

После успешного завершения функций `socket` и `bind` сокет создан и готов к работе. Дальнейшие действия с ним зависят от того, какой протокол он реализует и для какой роли предназначен. Мы разберем эти операции в разделах, посвященных соответствующим протоколам. Там же мы увидим, что в некоторых случаях можно обойтись без вызова функции `bind`, поскольку она будет неявно вызвана при вызове других функций библиотеки сокетов.

Когда сокет больше не нужен, следует освободить связанные с ним ресурсы. Это выполняется в два этапа: сначала сокет "выключается", а потом закрывается.

Для выключения сокета предусмотрена функция `shutdown`, имеющая следующий прототип:

```
function shutdown(s: TSocket; how: Integer): Integer;
```

Параметр `S` определяет сокет, который необходимо выключить, параметр `how` может принимать значения `SD_RECEIVE`, `SD_SEND` или `SD_BOTH`. Функция возвращает ноль при успешном выполнении и `SOCKET_ERROR` — в случае ошибки.

Вызов функции с параметром `SD_RECEIVE` запрещает чтение данных из входного буфера сокета. Однако на уровне протокола вызов этой функции игнорируется: дейтаграммы UDP и пакеты TCP, посланные данному сокету, продолжают помещаться в буфер, хотя программа уже не может их оттуда забрать.

При указании значения `SD_SEND` функция запрещает отправку данных через сокет. В случае протокола TCP при этом удаленный сокет получает специальный сигнал, предусмотренный данным протоколом, уведомляющий о том, что больше данные посылаться не будут. Если на момент вызова `shutdown` в буфере для исходящих остаются данные, сначала посылаются они, а потом только сигнал о завершении. Поскольку протокол UDP подобных сигналов не предусматривает, то в этом случае `shutdown` просто запрещает библиотеке сокетов использовать указанный сокет для отправки данных.

Параметр `SD_BOTH` позволяет одновременно запретить и прием, и передачу данных через сокет.

Примечание

Модуль `WinSock` до пятой версии Delphi включительно содержит ошибку: в нем не определены константы `SD_XXX`. Чтобы использовать их в своей программе, нужно объявить их так, как показано в листинге 2.5.

Листинг 2.5. Объявление констант SD_XXX для Delphi 5 и более ранних версий

```
const
  SD_RECEIVE = 0;
  SD_SEND = 1;
  SD_BOTH = 2;
```

Для освобождения ресурсов, связанных с сокетом, служит функция closesocket, которая освобождает память, выделенную для буферов, и порт. Ее единственный параметр задает сокет, который требуется закрыть, а возвращаемое значение — ноль или SOCKET_ERROR. После вызова этой функции соответствующий дескриптор сокета перестает иметь смысл, и использовать его больше нельзя.

По умолчанию функция closesocket немедленно возвращает управление вызвавшей ее программе, а процесс закрытия сокета начинает выполняться в фоновом режиме. Под закрытием подразумевается не только освобождение ресурсов, но и отправка данных, которые остались в выходном буфере сокета. Вопрос о том, как изменить поведение функции closesocket, будет обсуждаться в *разд. 2.1.17*. Если сокет закрывается одной нитью в тот момент, когда другая нить пытается выполнить какую-либо операцию с этим сокетом, то эта операция завершается с ошибкой.

Функция shutdown нужна в первую очередь для того, чтобы заранее сообщить партнеру по связи о намерении завершить связь, причем это имеет смысл только для протоколов, поддерживающих соединение. В случае UDP функцию shutdown вызывать практически бессмысленно, можно сразу вызывать closesocket. При использовании TCP удаленная сторона получает сигнал о выключении партнера, но стандартная библиотека сокетов не позволяет программе обнаружить его получение (такие функции есть в сокетах Windows, о чем мы будем говорить далее). Но этот сигнал может быть важен для внутрисистемных функций, реализующих сокеты. Windows-версия библиотеки сокетов относится к отсутствию данного сигнала достаточно либерально, поэтому вызов shutdown в том случае, когда и клиент, и сервер работают под управлением Windows, не обязателен. Но реализации TCP в других системах не всегда столь же снисходительно относятся к подобной небрежности. Результатом может стать долгое (до двух часов) "подвешенное" состояние сокета в той системе, когда с ним и работать уже нельзя, и информации об ошибке программа не получает. Поэтому в случае TCP лучше не пренебрегать вызовом shutdown, чтобы сокет на другой стороне не имел проблем.

MSDN рекомендует следующий порядок закрытия TCP-сокета. Во-первых, сервер не должен закрывать свой сокет по собственной инициативе, он может это делать только после того, как был закрыт связанный с ним клиентский

сокет. Клиент начинает закрытие сокета с вызова shutdown с параметром SD_SEND. Сервер после этого сначала получает все данные, которые оставались в буфере сокета клиента, а затем получает от клиента сигнал о завершении передачи. Тем не менее сокет клиента продолжает работать на прием, поэтому сервер при необходимости может на этом этапе послать клиенту какие-либо данные, если это необходимо. Затем сервер вызывает shutdown с параметром SD_SEND, и сразу после этого — closesocket. Клиент продолжает читать данные из входящего буфера сокета до тех пор, пока не будет получен сигнал о завершении передачи сервером. После этого клиент также вызывает closesocket. Такая последовательность гарантирует, что данные не будут потеряны, но, как мы уже обсуждали ранее, она не может быть реализована в рамках стандартных сокетов из-за невозможности получить сигнал о завершении передачи, посланный удаленной стороной. Поэтому на практике следует реализовывать упрощенный способ завершения связи: клиент вызывает shutdown с параметром SD_SEND или SD_BOTH, и сразу после этого — closesocket. Сервер при попытке выполнить операцию с сокетом получает ошибку, после которой также вызывает closesocket. Вызов shutdown на стороне сервера при этом не нужен, т. к. в этот момент соединение уже потеряно, и высылать данные из буфера вместе с сигналом завершения уже некуда.

2.1.9. Передача данных при использовании UDP

Мы наконец-то добрались до изучения того, ради чего сокеты и создавались: как передавать и получать с их помощью данные. По традиции начнем рассмотрение с более простого протокола UDP. Функции, которые рассматриваются в этом разделе, могут работать и с другими протоколами, и от этого их поведение может меняться. Мы здесь описываем только их поведение при использовании UDP.

Для передачи данных удаленному сокету предусмотрена функция sendto, описанная следующим образом:

```
function sendto(s: TSocket; var Buf; len, flags: Integer;
  var addrto: TSockAddr; tolen: Integer): Integer;
```

Первый параметр данной функции задает сокет, который служит для передачи данных. Здесь нужно указать значение, полученное ранее от функции socket. Параметр Buf задает буфер, в котором хранятся данные для отправки, а параметр len — размер этих данных в байтах. Параметр flags позволяет указать некоторые дополнительные опции, которых мы здесь касаться не будем, т. к. в большинстве случаев они не нужны. Пока следует запомнить, что параметр flags в функции sendto, а также в других функциях, где он встречается, должен быть равен нулю. Параметр addrto задает адрес (состоящий из

IP-адреса и порта) удаленного сокета, который должен получить эти данные. Значение параметра addrto должно формироваться по тем же правилам, что и значение аналогичного параметра функции bind, за исключением того, что IP-адрес и порт должны быть заданы явно (т. е. не допускаются значение INADDR_ANY и нулевой номера порта). Параметр tolen задает длину буфера, отведенного для адреса, и должен быть равен SizeOf(TSockAddr).

Один вызов функции sendto приводит к отправке одной дейтаграммы. Данные, переданные в sendto, никогда не разбиваются на несколько дейтаграмм, и данные, переданные последовательными вызовами sendto, никогда не объединяются в одну дейтаграмму.

Функцию sendto можно использовать с сокетами, не привязанными к адресу. В этом случае внутри библиотеки сокетов будет неявно вызвана функция bind для привязки сокета к адресу INADDR_ANY и нулевому порту (т. е. адрес и порт будут выбраны системой).

Если выходной буфер сокета имеет ненулевой размер, sendto помещает данные в этот буфер и сразу возвращает управление программе, а собственно отправка данных осуществляется библиотекой сокетов в фоновом режиме. Поэтому успешное завершение sendto гарантирует только то, что данные скопированы в буфер и что на момент их копирования не обнаружено никаких проблем, которые делали бы невозможной их отправку. Но такие проблемы могут возникнуть позже, поэтому даже в случае успешного завершения sendto отправитель не получает гарантии, что данные посланы.

Если в выходном буфере сокета не хватает места для новой порции данных, sendto не возвращает управление программе (т. е. блокирует ее) до тех пор, пока в буфере за счет фоновой отправки не появится достаточно места или не будет обнаружена ошибка.

Если размер выходного буфера сокета равен нулю, функция sendto копирует данные сразу в сеть, без промежуточной буферизации. Когда функция вернет управление программе, программа может быть уверена, что информация уже успешно передана в сеть. Однако даже в этом случае успешное завершение sendto не гарантирует доставку информации: дейтаграмма может потеряться по дороге.

В случае успешного завершения функция sendto возвращает количество байтов, скопированных в буфер (или переданных напрямую в сеть, если буфера нет). Для протокола UDP это значение может быть равно только значению параметра len, хотя для некоторых других протоколов (например, TCP) возможны ситуации, когда в буфер сокета копируется только часть данных, переданных программой, и тогда sendto возвращает значение в диапазоне от 1 до len. Если при выполнении sendto возникает ошибка, она возвращает значение SOCKET_ERROR (эта константа имеет отрицательное значение).

Для получения данных, присланных сокету, предназначена функция `recvfrom`, имеющая следующий прототип:

```
function recvfrom(s: TSocket; var Buf; len, flags: Integer;
  var from: TSockAddr; var fromlen: Integer): Integer;
```

Параметр `s` задает сокет, из входного буфера которого будут извлекаться данные, `Buf` — буфер, в который эти данные будут копироваться, а `len` — размер этого буфера. Параметр `flags` задает дополнительные опции и в большинстве случаев должен быть равен нулю. Параметр `from` выходной: в него помещается адрес, с которого была послана дейтаграмма. Параметр `fromlen` задает размер в байтах буфера для адреса отправителя. При вызове функции значение переменной, подставляемой в качестве фактического параметра, должно быть равно `SizeOf(TSockAddr)`. Функция меняет это значение на ту длину, которая реально потребовалась для хранения адреса отправителя (в случае UDP это значение также будет равно `SizeOf(TSockAddr)`).

В оригинале параметры `from` и `fromlen` передаются как указатели, и программа может использовать вместо них нулевые указатели, если ее не интересует адрес отправителя. Разработчики модуля `WinSock` заменили указатели параметрами-переменными, что в большинстве случаев удобнее. Но для передачи нулевых указателей приходится в качестве фактических параметров подставлять неуклюжие конструкции `PSockAddr(nil)^` и `PInteger(nil)^`.

Функция `recvfrom` всегда читает только одну дейтаграмму, даже если размер переданного ей буфера достаточен для чтения нескольких дейтаграмм. Если на момент вызова `recvfrom` дейтаграммы во входном буфере сокета отсутствуют, функция будет ждать, пока они там появятся, и до этого момента не вернет управление вызвавшей ее программе. Если в буфере находится несколько дейтаграмм, то они читаются в порядке очередности поступления в буфер. Напомним, что дейтаграммы могут поступать в буфер не в том порядке, в котором они были отправлены. Кроме того, в очень редких случаях буфер может содержать несколько копий одной дейтаграммы, каждую из которых нужно извлекать отдельно.

Значение, возвращаемое функцией `recvfrom`, равно длине прочитанной дейтаграммы. Это значение может быть равно нулю, т. к. UDP позволяет отправлять дейтаграммы нулевой длины (для этого при вызове `sendto` нужно задать параметр `len` равным нулю). Если обнаружена какая-то ошибка, возвращается значение `SOCKET_ERROR`.

Если размер буфера, определяемого параметром `Buf`, меньше, чем первая находящаяся во входном буфере сокета дейтаграмма, то копируется только часть дейтаграммы, помещающаяся в буфере, а `recvfrom` завершается с ошибкой (`WSAGetLastError` при этом вернет ошибку `WSAEMSGSSIZE`). Оставшаяся

часть дейтаграммы при этом безвозвратно теряется, при следующем вызове recvfrom будет прочитана следующая дейтаграмма. Этой проблемы легко избежать, т. к. длина дейтаграммы в UDP не может превышать 65 507 байтов. Достаточно подготовить буфер соответствующей длины, и в него гарантированно поместится любая дейтаграмма.

Другой способ избежать подобной проблемы — использовать флаг MSG_PEEK. В этом случае дейтаграмма не удаляется из входного буфера сокета, а значение, возвращаемое функцией recvfrom, равно длине дейтаграммы. При этом в буфер, заданный параметром Buf, копируется та часть дейтаграммы, которая в нем помещается. Программа может действовать следующим образом: вызвать recvfrom с флагом MSG_PEEK, выделить память, требуемую для хранения дейтаграммы, вызвать recvfrom без флага MSG_PEEK, чтобы прочитать дейтаграмму целиком и удалить ее из входного буфера сокета. Этот метод сложнее, а 65 507 байтов — не очень большая по нынешним меркам память, поэтому легче все-таки заранее приготовить буфер фиксированной длины.

Функция recvfrom непригодна для тех сокетов, которые еще не привязаны к адресу, поэтому перед вызовом этой функции должна быть вызвана либо функция bind, либо функция, которая осуществляет неявную привязку сокета к адресу (например, sendto).

Протокол UDP не поддерживает соединения в том смысле, в котором их поддерживает TCP, но библиотека сокетов позволяет частично имитировать такие соединения. Для этого служит функция connect, имеющая следующий прототип:

```
function connect(s: TSocket; var name: TSockAddr;
  namelen: Integer): Integer;
```

Параметр s задает сокет, который должен быть "соединен" с удаленным адресом. Адрес задается параметром name аналогично тому, как он задается в параметре addr функции sendto. Параметр namelen содержит длину структуры, описывающей адрес, и должен быть равен SizeOf(TSockAddr). Функция возвращает ноль при успешном завершении и SOCKET_ERROR — в случае ошибки.

Вызов функции connect в случае UDP устанавливает фильтр для входящих дейтаграмм. Дейтаграммы, адрес отправителя которых не совпадает с адресом, заданным в функции connect, игнорируются: новые дейтаграммы не помещаются во входной буфер сокета, а те, которые находились там на момент вызова connect, удаляются из него. Функция connect не проверяет, существует ли адрес, с которым сокет "соединяется", и может успешно завершиться, даже если узла с таким IP-адресом нет.

Программа может вызывать connect неограниченное число раз с разными адресами. Если параметр name задает IP-адрес INADDR_ANY и нулевой порт, то со-

кет "отсоединяется", т. е. все фильтры для него снимаются, и он ведет себя так же, как сокет, для которого не была вызвана функция `connect`. Для сокетов, не привязанных к адресу, `connect` неявно вызывает `bind`.

После вызова `connect` для отправки данных можно использовать функцию `send` со следующим прототипом:

```
function send(s: TSocket; var Buf; len, flags: Integer): Integer;
```

От функции `sendto` она отличается отсутствием параметров `addrto` и `tolen`. При использовании `send` дейтаграмма отправляется по адресу, заданному при вызове `connect`. В остальном эти функции ведут себя одинаково. Функция `sendto` при работе с "соединенным" сокетом ведет себя так же, как с несоединенным, т. е. отправляет дейтаграмму по адресу, определяемому параметром `addrlen`, а не по адресу, заданному при вызове `connect`.

Получение данных через "соединенные" сокеты может также осуществляться с помощью функции `recv`, имеющей следующий прототип:

```
function recv(s: TSocket; var Buf; len, flags: Integer): Integer;
```

От своего аналога `recvfrom` она отличается только отсутствием параметров `from` и `fromlen`, через которые передается адрес отправителя дейтаграммы.

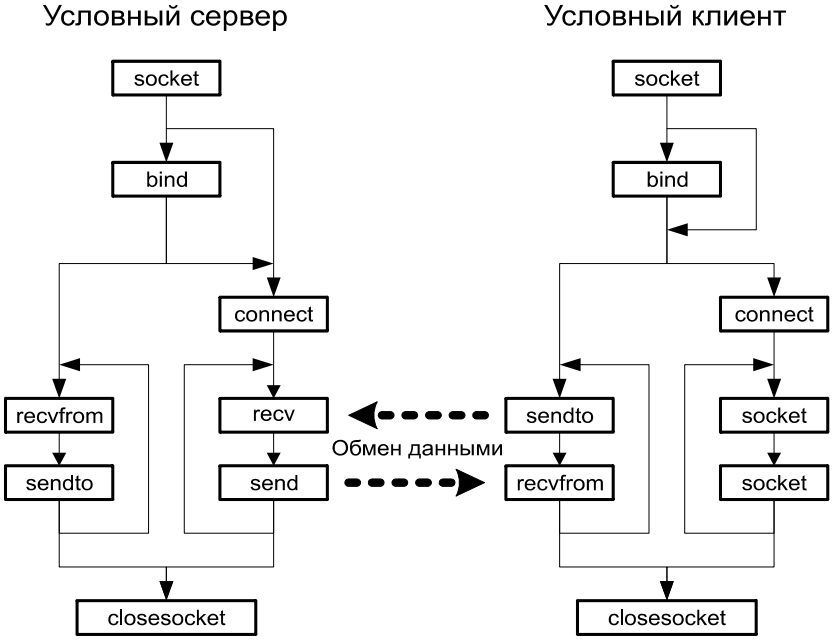


Рис. 2.1. Последовательность действий программы при обмене данными с помощью UDP

Строго говоря, функцию recv можно использовать и для несоединенных сокетов, но при этом программе остается неизвестным адрес отправителя. В случае же "соединенных" сокетов адрес отправителя заранее известен — это адрес, заданный в функции connect, а дейтаграммы всех других отправителей будут отбрасываться. Функция recvfrom также пригодна для "соединенных" сокетов, но адрес отравителя, который она возвращает, в данном случае может быть только тот, который определен в функции connect.

Таким образом, функция connect в случае протокола UDP позволяет, во-первых, выполнить фильтрацию входящих дейтаграмм по адресу средствами самой библиотеки сокетов, а во-вторых, использовать более лаконичные альтернативы recvfrom и sendto — recv и send.

Возможные последовательности действий программы для протокола UDP показаны на рис. 2.1.

2.1.10. Пример программы: простейший чат на UDP

Попробуем применить свои знания на практике и напишем простейший чат на основе протокола UDP. Пример этой программы находится на прилагаемом к книге компакт-диске и называется UDPChat, окно приложения показано на рис. 2.2.

Прежде чем писать программу, необходимо определиться с форматом передаваемых данных (т. е. договориться о протоколе уровня представлений). Так как мы пишем простейший пример, то и протокол у нас будет простейшим: дейтаграмма содержит текстовое сообщение, введенное пользователем, без завершающего нуля (он не нужен, т. к. размер строки определяется размером дейтаграммы) и без дополнительной служебной информации.

Для начала нам потребуется научиться сообщать пользователю об ошибках. Номер ошибки мало что дает даже опытному пользователю, поэтому сообщения должны быть дружественными, с внятным объяснением того, какая именно ошибка произошла. К счастью, мы избавлены от необходимости вручную писать текстовое сообщение для каждой из возможных ошибок, т. к. в системе уже есть функция FormatMessage, которая возвращает текстовое сообщение по коду ошибки (эта функция работает со всеми ошибками, а не только с ошибками сокетов). На основе FormatMessage мы создадим функцию GetErrorString (листинг 2.6), которая возвращает сообщение, соответствующее коду ошибки, возвращаемому функцией WSAGetLastError. Эта функция будет встречаться во всех наших примерах.

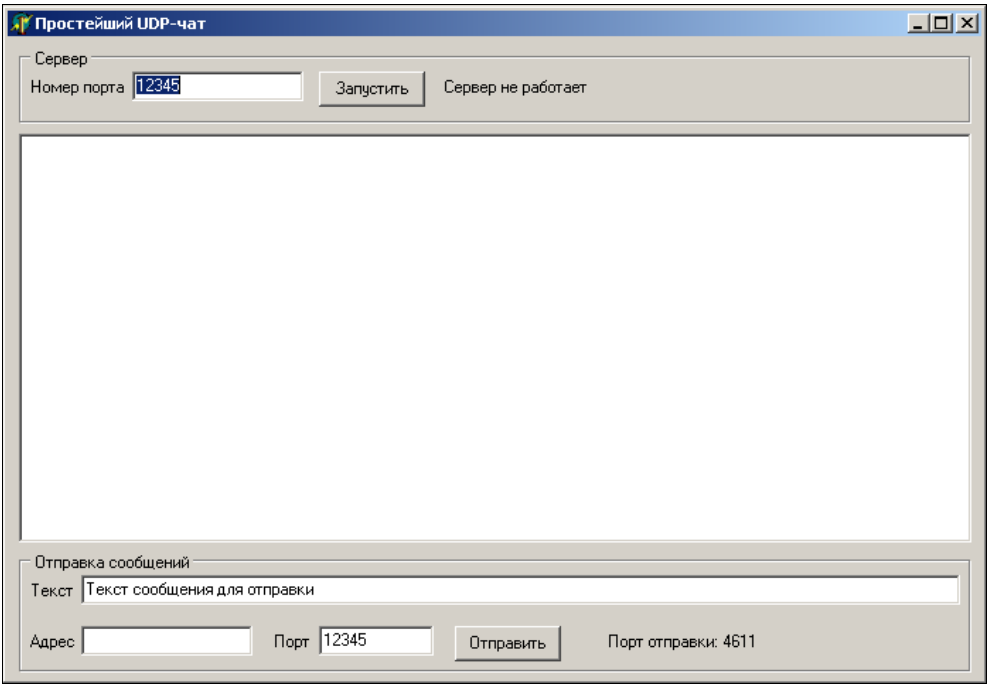


Рис. 2.2. Главное окно UDP-чата

Листинг 2.6. Функция GetErrorString, возвращающая описание ошибки

```
// Функция GetErrorString возвращает сообщение об ошибке,
// сформированное системой на основе значения, которое
// вернула функция WSAGetLastError. Для получения сообщения
// используется системная функция FormatMessage.
function GetErrorString: string;
var
  Buffer: array[0..2047] of Char;
begin
  FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, nil, WSAGetLastError, $400,
    @Buffer, SizeOf(Buffer), nil);
  Result := Buffer;
end;
```

Нам понадобится и принимать, и передавать данные. Как мы помним, функция `recvfrom` не возвращает управление вызвавшей ее нити до тех пор, пока не будут получены данные. Таким образом, если мы будем вызывать `recvfrom` в главной нити, то при отсутствии входящих дейтаграмм программа просто

повиснет, т. к. не сможет обрабатывать оконные сообщения. Поэтому все действия по приему сообщений мы должны вынести в отдельную нить. Задача этой нити очень проста: она в бесконечном цикле вызывает recvfrom и все полученные дейтаграммы передает в главное окно для отображения на экране.

Нить, читающая данные, создается обычным образом — порождением наследника от класса TThread. Мы не будем возлагать на эту нить задачу создания сокета, — пусть он создается в главной нити, а затем его дескриптор передается в дополнительную, которая сохраняет его в своем внутреннем поле FSocket. Код нити, читающей сообщения, показан в листинге 2.7.

Листинг 2.7. Код "читающей" нити

```
unit ReceiveThread;
{  В этом модуле реализуется дополнительная нить UDP-чата,
   отвечающая за прием сообщений.
}

interface

uses
  SysUtils, Classes, WinSock;

type
  TReceiveThread = class(TThread)
  private
    // Сообщение, которое нужно добавить в лог,
    // хранится в отдельном поле, т. к. метод, вызывающийся через
    // Synchronize, не может иметь параметров.
    FMessage: string;
    // Сокет, получающий сообщения
    FSocket: TSocket;
    // Вспомогательный метод для вызова через Synchronize
    procedure DoLogMessage;
  protected
    procedure Execute; override;
    // Вывод сообщения в лог главной формы
    procedure LogMessage(const Msg: string);
  public
    constructor Create(ServerSocket: TSocket);
  end;
```

```
implementation

uses
  ChatMainUnit;

{ TReceiveThread }

// Сокет, получающий сообщения, создается в главной нити,
// а сюда передается через параметр конструктора
constructor TReceiveThread.Create(ServerSocket: TSocket);
begin
  FSocket := ServerSocket;
  inherited Create(False);
end;

procedure TReceiveThread.Execute;
var
  // Буфер для получения сообщения.
  // Размер равен максимальному размеру UDP-дейтаграммы
  Buffer: array[0..65506] of Byte;
  // Адрес, с которого пришло сообщение
  RecvAddr: TSockAddr;
  RecvLen, AddrLen: Integer;
  Msg: string;
begin
  // Начинаем бесконечный цикл, на каждой итерации которого
  // читается одна дейтаграмма
  repeat
    AddrLen := SizeOf(RecvAddr);
    // Получаем дейтаграмму
    RecvLen := recvfrom(FSocket, Buffer, SizeOf(Buffer), 0,
      RecvAddr, AddrLen);
    // Так как UDP не поддерживает соединение, ошибку при вызове recvfrom
    // мы можем получить, только если случилось что-то совсем
    // экстраординарное. В этом случае завершаем работу нити.
    if RecvLen < 0 then
    begin
      LogMessage('Ошибка при получении сообщения: '+GetErrorString);
      // Перевод элементов управления главной формы
      // в состояние "Сервер не работает"
      Synchronize(ChatForm.OnStopServer);
      Break;
    end;
```

```
    // Устанавливаем нужный размер строки
    SetLength(Msg, RecvLen);
    // и копируем в нее дейтаграмму из буфера
    if RecvLen > 0 then
      Move(Buffer, Msg[1], RecvLen);
    LogMessage('Сообщение с адреса ' + inet_ntoa(RecvAddr.sin_addr) +
      ':' + IntToStr(ntohs(RecvAddr.sin_port)) + ': ' + Msg);
  until False;
  closesocket(FSocket);
end;

procedure TReceiveThread.LogMessage(const Msg: string);
begin
  FMessage := Msg;
  Synchronize(DoLogMessage);
end;

procedure TReceiveThread.DoLogMessage;
begin
  ChatForm.AddMessageToLog(FMessage);
end;

end.
```

Отправлять данные можно и из основной нити, поскольку функция `sendto` при наших объемах данных практически никогда не будет блокировать вызывающую ее нить (да и при больших объемах данных, как мы увидим в дальнейшем, этого практически никогда не бывает). Соответственно, нам нужно создать два сокета: один для отправки сообщений, другой для приема.

Сокет для отправки сообщений создаем сразу же при запуске приложения, при обработке события `OnCreate` главной (и единственной) формы. Дескриптор сокета хранится в поле `FSendSocket`. Пользователю не принципиально, какой порт займет этот сокет, поэтому мы доверяем его выбор системе (листинг 2.8).

Листинг 2.8. Инициализация программы UDPChat

```
procedure TChatForm.FormCreate(Sender: TObject);
var
  // Без этой переменной не удастся инициализировать библиотеку сокетов
  WSAData: TWSAData;
  // Адрес, к которому привязывается сокет для отправки сообщений
  Addr: TSockAddr;
  AddrLen: Integer;
```

```
begin
  // Инициализация библиотеки сокетов
  if WSAStartup($101, WSAData) <> 0 then
  begin
    MessageDlg('Ошибка при инициализации библиотеки WinSock',
      mtError, [mbOK], 0);
    Application.Terminate;
  end;
  // Перевод элементов управления в состояние "Сервер не работает"
  OnStopServer;
  // Создание сокета
  FSendSocket := socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if FSendSocket = INVALID_SOCKET then
  begin
    MessageDlg('Ошибка при создании отправляющего сокета:'#13#10 +
      GetErrorString, mtError, [mbOK], 0);
    Exit;
  end;
  // Формирование адреса, к которому будет привязан сокет
  // для отправки сообщений
  FillChar(Addr.sin_zero, SizeOf(Addr.sin_zero), 0);
  Addr.sin_family := AF_INET;
  // Пусть система сама выбирает для него IP-адрес и порт
  Addr.sin_addr.S_addr := INADDR_ANY;
  Addr.sin_port := 0;
  // Привязка сокета к адресу
  if bind(FSendSocket, Addr, SizeOf(Addr)) = SOCKET_ERROR then
  begin
    MessageDlg('Ошибка при привязке отправляющего сокета к адресу:'#13#10
      + GetErrorString, mtError, [mbOK], 0);
    Exit;
  end;
  // Узнаем, какой адрес система назначила сокету.
  // Это нужно для вывода информации для пользователя
  AddrLen := SizeOf(Addr);
  if getsockname(FSendSocket, Addr, AddrLen) = SOCKET_ERROR then
  begin
    MessageDlg('Ошибка при получении адреса отправляющего сокета:'#13#10
      + GetErrorString, mtError, [mbOK], 0);
    Exit;
  end;
  // Не забываем, что номер порта возвращается в сетевом формате,
  // и его нужно преобразовать к обычному функцией ntohs.
```

```
  LabelSendPort.Caption:= 'Порт отправки: ' +
    IntToStr(ntohs(Addr.sin_port));
end;
```

Сокет для получения сообщений создается при нажатии кнопки **Запустить** и привязывается к тому порту, который указал пользователь. В случае его успешного создания запускается нить, которой передается этот сокет, и все дальнейшие операции с ним выполняет эта нить. Нить вместе с этим сокетом мы будем условно называть сервером. Код обработчика нажатия кнопки **Запустить** показан в листинге 2.9.

Листинг 2.9. Обработчик нажатия кнопки *Запустить*

```
// Реакция на кнопку "Запустить"
procedure TChatForm.BtnStartServerClick(Sender: TObject);
var
  // Сокет для приема сообщений
  ServerSocket: TSocket;
  // Адрес, к которому привязывается сокет для приема сообщений
  ServerAddr: TSockAddr;
begin
  // Формирование адреса сокета для приема сообщений
  FillChar(ServerAddr.sin_zero, SizeOf(ServerAddr.sin_zero), 0);
  ServerAddr.sin_family := AF_INET;
  // IP-адрес может выбрать система, а порт назначаем тот,
  // который задан пользователем
  ServerAddr.sin_addr.S_addr := INADDR_ANY;
  try
    // Не забываем преобразовать номер порта к сетевому формату
    // с помощью функции htons
    ServerAddr.sin_port := htons(StrToInt(EditServerPort.Text));
    if ServerAddr.sin_port = 0 then
    begin
      MessageDlg('Номер порта должен находиться в диапазоне 1-65535',
        mtError, [mbOK], 0);
      Exit;
    end;
    // Создание сокета для получения сообщений
    ServerSocket := socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if ServerSocket = INVALID_SOCKET then
    begin
      MessageDlg('Ошибка при создании сокета:'#13#10 + GetErrorString,
        mtError, [mbOK], 0);
```

```
      Exit;
    end;
    // Привязка сокета к адресу
    if bind(ServerSocket, ServerAddr, SizeOf(ServerAddr)) =
      SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при привязке сокета к адресу:'#13#10 +
        GetErrorString, mtError, [mbOK], 0);
      closesocket(ServerSocket);
      Exit;
    end;
    // Создание нити, которая будет получать сообщения.
    // Сокет передается ей, и дальше она отвечает за него.
    TReceiveThread.Create(ServerSocket);
    // Перевод элементов управления в состояние "Сервер работает"
    LabelServerPort.Enabled := False;
    EditServerPort.Enabled := False;
    BtnStartServer.Enabled := False;
    LabelServerState.Caption := 'Сервер работает';
  except
    on EConvertError do
      // Это исключение может возникнуть только в одном месте -
      // при вызове StrToInt(EditServerPort.Text)
      MessageDlg('"' + EditServerPort.Text +
        '" не является целым числом', mtError, [mbOK], 0);
    on ERangeError do
      // Это исключение может возникнуть только в одном месте -
      // при присваивании значения номеру порта
      MessageDlg('Номер порта должен находиться в диапазоне 1-65535',
        mtError, [mbOK], 0);
  end;
end;
```

Для отправки сообщения пользователь должен нажать кнопку **Отправить**. При этом формируется адрес на основании введенных пользователем данных и вызывается функция sendto (листинг 2.10). Пользователь должен каким-то образом узнать, какой порт назначения выбран у адресата. Его IP-адрес тоже, разумеется, необходимо знать.

Листинг 2.10. Обработчик нажатия кнопки *Отправить*

```
// Реакция на кнопку "Отправить"
procedure TChatForm.BtnSendClick(Sender: TObject);
```

```
var
  // Адрес назначения
  SendAddr: TSockAddr;
  // Сообщение для отправки
  Msg: string;
  // Результат отправки
  SendRes: Integer;
begin
  // Формируем адрес назначения на основе того,
  // что пользователь ввел в соответствующие поля
  FillChar(SendAddr.sin_zero, SizeOf(SendAddr.sin_zero), 0);
  SendAddr.sin_family := AF_INET;
  SendAddr.sin_addr.S_addr := inet_addr(PChar(EditSendAddr.Text));
  // Для совместимости со старыми версиями Delphi приводим
  // константу INADDR_NONE к типу u_long
  if SendAddr.sin_addr.S_addr = u_long(INADDR_NONE) then
  begin
    MessageDlg('"' +EditSendAddr.Text + '" не является IP-адресом',
      mtError, [mbOK], 0);
    Exit;
  end;
  try
    SendAddr.sin_port := htons(StrToInt(EditSendPort.Text));
    // Получаем сообщение, которое ввел пользователь.
    // Дополнительная переменная понадобилась потому,
    // что нам потребуется ее передавать в качестве var-параметра,
    // а делать это со свойством EditMessage.Text нельзя.
    Msg := EditMessage.Text;
    if Length(Msg) = 0 then
      // Отправляем дейтаграмму нулевой длины -
      // протокол UDP разрешает такое
      SendRes := sendto(FSendSocket, Msg, 0, 0, SendAddr,
        SizeOf(SendAddr))
    else
      // Отправляем сообщение, содержащее строку
      SendRes := sendto(FSendSocket, Msg[1], Length(Msg), 0, SendAddr,
        SizeOf(SendAddr));
    if SendRes < 0 then
      MessageDlg('Ошибка при отправке сообщения:'#13#10 + GetErrorString,
        mtError, [mbOK], 0)
    else
      AddMessageToLog('Для ' + EditSendAddr.Text + ':' +
        EditSendPort.Text + ' отправлено сообщение: ' + Msg);
```

```
  except
    on EConvertError do
      // Это исключение может возникнуть только в одном месте -
      // при вызове IntToStr(EditSendPort.Text)
      MessageDlg('"' + EditSendPort.Text + '" не является целым числом',
        mtError, [mbOK], 0);
    on ERangeError do
      // Это исключение может возникнуть только в одном месте -
      // при присваивании значения номеру порта
      MessageDlg('Номер порта должен находиться в диапазоне 1-65535',
        mtError, [mbOK], 0);
  end;
end;
```

Заметим, что в нашем сервере есть один очень неудобный момент. Предположим, получено сообщение, и программа высветила следующую надпись: "Сообщение с адреса 192.168.200.211:2231: Привет!". Порт, который указан в этом сообщении — это порт того сокета, который используется на удаленной стороне для отправки сообщений. Для их получения там предназначен другой сокет и другой порт, поэтому цифра 2231 не несет никакой информации о том, на какой порт нужно отправлять ответ. В нашем примитивном чате соответствие между номерами портов для отправки и для получения сообщений пользователю приходится держать в голове. По сути дела, более-менее нормальная работа такого чата возможна только тогда, когда все пользователи используют один и тот же порт для сокета, принимающего сообщения (или когда компьютеры стоят рядом, и пользователи могут сообщить друг другу номера своих портов).

Не будем слишком строги к нашему первому примеру — его ценность в том, что он учит основам использования сокетов и протокола UDP. Проблему можно было бы решить, передавая в дейтаграмме не только сообщения, но и номер порта для ответа и реализовав в программе таблицу соответствия портов для отправки и приема сообщений известных адресатов. Однако это уже не относится к работе с сокетами, и потому мы не стали загромождать этим пример. Чуть позже мы научимся делать так, что функция `recvfrom` не будет блокировать нить, и тогда переделаем чат так, чтобы отправка и прием данных осуществлялись через один и тот же сокет.

Здесь возникает вопрос: нельзя ли с помощью `sendto` передавать данные через тот же сокет, который в другой нити используется в функции `recvfrom`? Документация по этому поводу упорно молчит. Если в нашем чате оставить только один сокет и задействовать его в обеих нитях, то все вроде как работает. Однако это тот случай, когда эксперимент не может служить доказательством, потому что у ошибок, связанных с неправильной синхронизацией ни-

тей, есть очень неприятная особенность: программа может миллион раз отработать правильно, а на миллион первый дать сбой. Поэтому сколько бы раз такой эксперимент ни завершился удачно, полной гарантии он все же не дает, так что приходится действовать осторожно и не использовать один сокет в разных нитях.

В заключение отметим, что наш чат допускает одновременное общение любого количества человек с любого числа адресов, но сообщения всегда передаются от одного человека к другому. Широковещательных и групповых сообщений у нас нет. Отметим также, что отправлять сообщения можно и не запуская "сервер".

Примечание

Для того чтобы протестировать работу чата, не обязательно иметь два компьютера, соединенных в сеть. Два или более экземпляра чата можно запустить и на одном компьютере, главное, чтобы у них у всех были разные порты для принимающего сокета. В качестве IP-адреса для отправки сообщений можно задавать адрес локального компьютера вида 127.0.0.N. Это же верно и для всех остальных примеров работы с сокетами.

2.1.11. Передача данных при использовании TCP

При программировании TCP и UDP применяются одни и те же функции, но их поведение при этом различно. Для передачи данных с помощью TCP необходимо сначала установить соединение, и после этого возможен обмен данными только с тем адресом, с которым это соединение установлено. Функция `sendto` может использоваться для TCP-сокетов, но ее параметры, задающие адрес получателя, игнорируются, а данные отправляются на тот адрес, с которым соединен сокет. Поэтому при отправке данных через TCP обычно прибегают к функции `send`, которая дает тот же результат. По тем же причинам обычно используется `recv`, а не `recvfrom`.

В TCP существует разделение ролей взаимодействующих сторон на клиент и сервер. Мы начнем изучение передачи данных в TCP с изучения действий клиента.

Для начала взаимодействия клиент должен соединиться с сервером с помощью функции `connect`. Мы уже знакомы с этой функцией, но в случае TCP она выполняет несколько иные действия. В данном случае она устанавливает реальное соединение, поэтому ее действия начинаются с проверки того, существует ли по указанному адресу серверный сокет, находящийся в режиме ожидания подключения. Функция `connect` завершается успешно только тогда, когда соединение установлено, и серверная сторона выполнила все необходимые для этого действия. При вызове `connect` в TCP предварительный явный вызов функции `bind` также не обязателен.

В отличие от UDP, сокет в TCP нельзя отсоединить или соединить с другим адресом, если он уже соединен. Для нового соединения необходим новый сокет.

Мы уже говорили, что TCP является надежным протоколом, т. е. в том случае, если пакет не доставлен, отправляющая сторона уведомляется об этом. Тем не менее успешное завершение `send`, как и в случае UDP, не является гарантией того, что пакет был отослан и дошел до получателя, а говорит только о том, что данные скопированы в выходной буфер сокета, и на момент копирования сокет был соединен. Если в дальнейшем библиотека сокетов не сможет отправить эти данные или не получит подтверждения об их доставке, соединение будет закрыто, и следующая операция с этим сокетом завершится с ошибкой.

Если выходной буфер сокета равен нулю, данные сразу копируются в сеть, но успешное завершение функции и в этом случае не гарантирует успешную доставку. Использовать нулевой выходной буфер для TCP-сокетов не рекомендуется, т. к. это снижает производительность при последовательной отправке данных небольшими порциями. При буферизации эти порции накапливаются в буфере, а потом отправляются одним большим пакетом, требующим одного подтверждения от клиента. Если же буферизация не осуществляется, то будет отправлено несколько мелких пакетов, каждый со своим заголовком и своим подтверждением от клиента, что приведет к снижению производительности.

Функция `recv` копирует пришедшие данные из входного буфера сокета в буфер, заданный параметром `Buf`, но не более `len` байтов. Скопированные данные удаляются из буфера сокета. При этом все полученные данные сливаются в один поток, поэтому получатель может самостоятельно выбирать, какой объем данных считывать за один раз. Если за один раз была скопирована только часть пришедшего пакета, оставшаяся часть не пропадает, а будет скопирована при следующем вызове `recv`. Функция `recv` возвращает число байтов, скопированных в буфер. Если на момент ее вызова входной буфер сокета пуст, она ждет, когда там что-то появится, затем копирует полученные данные и лишь после этого возвращает управление вызвавшей ее программе. Если `recv` возвращает 0, это значит, что удаленный сокет корректно завершил соединение. Если соединение завершено некорректно (например, из-за обрыва кабеля или сбоя удаленного компьютера), функция завершается с ошибкой (т. е. возвращает `SOCKET_ERROR`).

Теперь рассмотрим, какие действия при использовании TCP должен выполнить сервер. Как мы уже говорили, сервер должен перевести сокет в режим ожидания соединения. Это делается с помощью функции `listen`, имеющей следующий прототип:

```
function listen(s: TSocket; backlog: Integer): Integer;
```

Параметр s задает сокет, который переводится в режим ожидания подключения. Этот сокет должен быть привязан к адресу, т. е. функция bind должна быть вызвана для него явно. Для сокета, находящегося в режиме ожидания, создается очередь подключений. Размер этой очереди определяется параметром backlog, если он равен SOMAXCONN, очередь будет иметь максимально возможный размер. В MSDN отмечается, что узнать максимально допустимый размер очереди стандартными средствами нельзя. Функция возвращает ноль при успешном завершении и SOCKET_ERROR — в случае ошибки.

Когда клиент вызывает функцию connect, и по указанному в ней адресу имеется сокет, находящийся в режиме ожидания подключения, то информация о клиенте помещается в очередь подключений этого сокета. Успешное завершение connect говорит о том, что на стороне сервера подключение добавлено в очередь. Однако для того, чтобы соединение было действительно установлено, сервер должен выполнить еще некоторые действия: извлечь из очереди соединений информацию о соединении и создать сокет для его обслуживания. Эти операции выполняются с помощью функции accept, имеющей следующий прототип:

```
function accept(s: TSocket; addr: PSockAddr; addrlen: PInteger): TSocket;
```

Параметр s задает сокет, который находится в режиме ожидания соединения и из очереди которого извлекается информация о соединении. Выходной параметр addr позволяет получить адрес клиента, установившего соединение. Здесь должен быть передан указатель на буфер, в который этот адрес будет помещен. Параметр addrlen содержит указатель на переменную, в которой хранится длина этого буфера: до вызова функции эта переменная должна содержать фактическую длину буфера, задаваемого параметром addr, после вызова — количество байтов буфера, реально понадобившихся для хранения адреса клиента. Очевидно, что в случае TCP и входное, и выходное значение этой переменной должно быть равно SizeOf(TSockAddr). Эти параметры передаются как указатели, а не как параметры-переменные, что было бы более естественно для Delphi, потому что библиотека сокетов допускает для этих указателей нулевые значения, если сервер не интересует адрес клиента. В данном случае разработчики модуля WinSock сохранили полную функциональность, предоставляемую библиотекой.

В случае ошибки функция accept возвращает значение INVALID_SOCKET. При успешном завершении возвращается дескриптор сокета, созданного библиотекой сокетов и предназначенного для обслуживания данного соединения. Этот сокет уже привязан к адресу и соединен с сокетом клиента, установившего соединение, и его можно использовать в функциях recv и send без предварительного вызова каких-либо других функций. Уничтожается этот сокет обычным образом, с помощью closesocket.

Исходный сокет, определяемый параметром s, остается в режиме прослушивания. Если сервер поддерживает одновременное соединение с несколькими клиентами, то функция accept может быть вызвана многократно. Каждый раз при этом будет создаваться новый сокет, обслуживающий одно конкретное соединение: протокол TCP и библиотека сокетов гарантируют, что данные, посланные клиентами, попадут в буферы соответствующих сокетов и не будут перемешаны.

Для получения целостной картины кратко повторим все сказанное. Для установления соединения сервер должен, во-первых, создать сокет с помощью функции socket, а во-вторых, привязать его к адресу с помощью функции bind. Далее сокет должен быть переведен в режим ожидания с помощью функции listen, а потом с помощью функции accept создается новый сокет, обслуживающий соединение, установленное клиентом. После этого сервер может обмениваться данными с клиентом. Клиент же должен создать сокет, при необходимости привязки к конкретному порту вызвать bind, и затем вызвать connect для установления соединения. После успешного завершения этой функции клиент может обмениваться данными с сервером. Это иллюстрируют листинги 2.11 и 2.12.

Листинг 2.11. Код сервера

```
var
  S, AcceptedSock: TSocket;
  Addr: TSockAddr;
  Data: TWSAData;
  Len: Integer;
begin
  WSAStartup($101, Data);
  S := socket(AF_INET, SOCK_STREAM, 0);
  Addr.sin_family := PF_INET;
  Addr.sin_port := htons(3030);
  Addr.sin_addr.S_addr := INADDR_ANY;
  FillChar(Addr.sin_zero, SizeOf(Addr.sin_zero), 0);
  bind(S, Addr, SizeOf(TSockAddr));
  listen(S, SOMAXCONN);
  Len := SizeOf(TSockAddr);
  AcceptedSock := accept(S, @Addr, @Len);
  { Теперь Addr содержит адрес клиента, с которым установлено
    соединение, а AcceptedSock — дескриптор, обслуживающий это
    соединение. Допустимы следующие действия:
     send(AcceptedSock, ...) — отправить данные клиенту
     recv(AcceptedSock, ...) — получить данные от клиента
     accept(...) — установить соединение с новым клиентом }
```

Здесь сокет сервера привязывается к порту с номером 3030. В общем случае разработчик сервера сам должен выбрать порт из диапазона 1024—65 535.

Листинг 2.12. Код клиента

```
var
  S: TSocket;
  Addr: TSockAddr;
  Data: TWSAData;
begin
  WSAStartup($101, Data);
  S := socket(AF_INET, SOCK_STREAM, 0);
  Addr.sin_family := AF_INET;
  Addr.sin_port := htons(3030);
  Addr.sin_addr.S_addr := inet_addr(...);
  FillChar(Addr.sin_zero, SizeOf(Addr.sin_zero), 0);
  connect(S, Addr, SizeOf(TSockAddr));
  { Теперь соединение установлено. Допустимы следующие действия:
     send(S, ...) — отправить данные серверу
     recv(S, ...) — получить данные от сервера }
```

В приведенном коде для краткости опущены проверки результатов функций с целью обнаружения ошибок. При написании серьезных программ этим пренебрегать нельзя. Блок-схема действий клиента и сервера приведена на рис. 2.3.

Если на момент вызова функции accept очередь соединений пуста, то нить, вызвавшая ее, блокируется до тех пор, пока какой-либо клиент не подключится к серверу. С одной стороны, это удобно: сервер может не вызывать функцию accept в цикле до тех пор, пока она не завершится успехом, а вызвать ее один раз и ждать, когда подключится клиент. С другой стороны, это создает проблемы тем серверам, которые должны взаимодействовать с несколькими клиентами. Действительно, пусть функция accept успешно завершилась и в распоряжении программы оказались два сокета: находящийся в режиме ожидания новых подключений и созданный для обслуживания уже существующего подключения. Если вызвать accept, то программа не сможет продолжить работу до тех пор, пока не подключится еще один клиент, а это может произойти через очень длительный промежуток времени или вообще никогда не случится. Из-за этого программа не сможет обрабатывать вызовы уже подключившегося клиента. С другой стороны, если функцию accept не вызывать, сервер не сможет обнаружить подключение новых клиентов. Средства для решения этой проблемы есть как у стандартных сокетов, так и у сокетов Windows, и далее мы их рассмотрим. Но существует довольно попу-

лярный способ ее решения средствами не библиотеки сокетов, а операционной системы. Он заключается в использовании отдельной нити для обслуживания каждого из клиентов. Каждый раз, когда клиент подключается, функция `accept` передает управление программе, возвращая новый сокет. Здесь сервер может породить новую нить, которая предназначена исключительно для обмена данными с новым клиентом. Старая нить после этого снова вызывает `accept` для старого сокета, а новая — функции `recv` и `send` для нового сокета. Такой метод решает заодно и проблемы, связанные с тем, что функции `send` и `recv` также могут блокировать работу программы и помешать обмену данными с другими клиентами. В данном случае будет блокирована только одна нить, обменивающаяся данными с одним из клиентов, а остальные нити продолжат свою работу. Далее мы рассмотрим пример сервера, работающего по такой схеме.

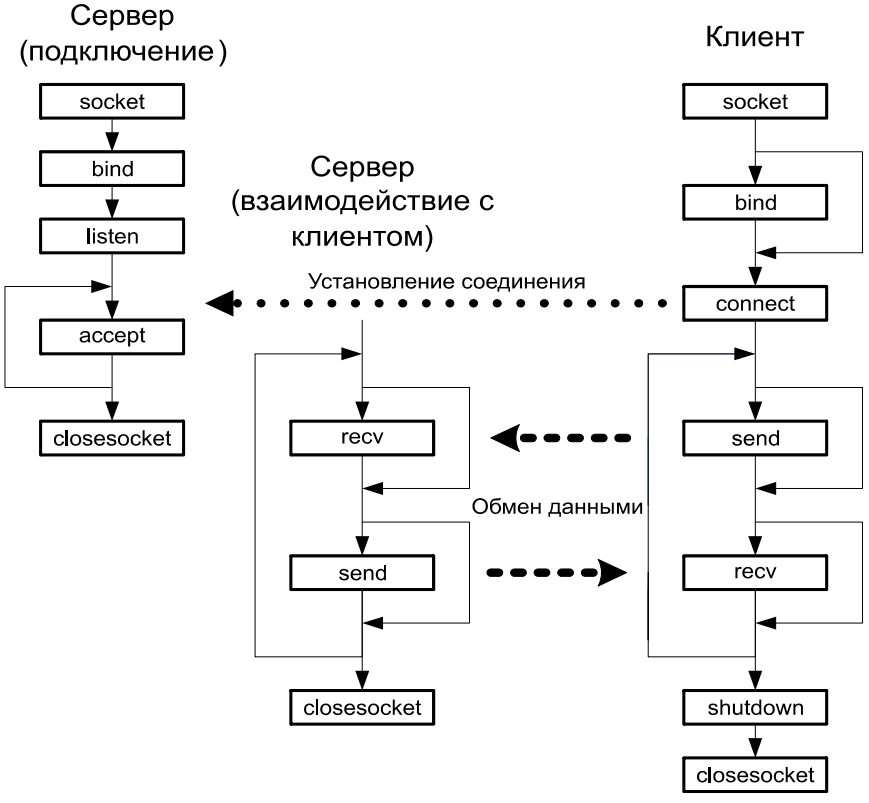


Рис. 2.3. Последовательность действий клиента и сервера при использовании TCP

То, что функция `recv` может возвратить только часть ожидаемого пакета, обычно вызывает трудности, поэтому здесь мы рассмотрим один из вариан-

тов написания функции (назовем ее ReadFromSocket), которая эти проблемы решает (листинг 2.13). Суть этой функции в том, что она вызывает recv до тех пор, пока не будет получено требуемое количество байтов или пока не возникнет ошибка. Тот код, который получает и анализирует приходящие данные, использует уже не recv, а ReadFromSocket, которая гарантирует, что будет возвращено столько байтов, сколько требуется.

Листинг 2.13. Функция ReadFromSocket, читающая из буфера сокета заданное количество байтов

```
// Функция читает Cnt байтов в буфер Buffer из сокета S
// Учитывается, что может потребоваться несколько операций чтения,
// прежде чем будет прочитано нужное число байтов.
// Возвращает:
//   1 — в случае успешного чтения
//   0 — в случае корректного закрытия соединения удаленной стороной
//  -1 — в случае ошибки чтения
function ReadFromSocket(S: TSocket; var Buffer;
  Cnt: Integer): Integer;
var
  Res, Total: Integer;
begin
  // Total содержит количество принятых байтов
  Total := 0;
  // Читаем байты в цикле до тех пор, пока не будет прочитано Cnt байтов
  repeat
    // На каждой итерации цикла нам нужно прочитать
    // не более чем Cnt — Total байтов, т. е. не более
    // чем нужное количество минус то, что уже прочитано
    // на предыдущих итерациях. Очередную порцию данных
    // помещаем в буфер со смещением Total.
    Res := recv(S, (PChar(@Buffer) + Total)^, Cnt — Total, 0);
    if Res = 0 then
    begin
      // Соединение закрыто удаленной стороной
      Result := 0;
      Exit;
    end;
    if Res < 0 then
    begin
      // Произошла ошибка при чтении
      Result := -1;
      Exit;
    end;
```

```
    Inc(Total, Res);
  until Total >= Cnt;
  Result := 1;
end;
```

Эта функция будет использоваться в дальнейшем в нескольких наших примерах.

2.1.12. Примеры передачи данных с помощью TCP

Теперь у нас достаточно знаний, чтобы написать TCP-клиент и TCP-сервер. Как и в случае с UDP, сначала нужно договориться о том, какими данными и в каком формате будут обмениваться наши программы. С протоколом, описанным здесь, нам предстоит работать на протяжении всей главы. По мере изучения новых возможностей библиотеки сокетов мы будем реализовывать новые варианты серверов и клиентов, но почти все они будут поддерживать один и тот же протокол, поэтому любой клиент сможет работать с любым сервером.

Наши сервер и клиент будут обмениваться строковыми сообщениями: клиент пошлет строку, сервер отправит ответ. Мы уже не можем, как в случае UDP, просто отправить строку, потому что при использовании TCP несколько строк могут быть отправлены одним пакетом, или наоборот, одна строка разбита на несколько пакетов. Соответственно, наш протокол должен позволять определить, где заканчивается одна строка и начинается другая.

Ранее мы уже упоминали три основных способа определения границ логического пакета в TCP: все пакеты могут иметь одинаковую длину, пакет может предваряться фиксированным заголовком, содержащим длину, между пакетами может вставляться заранее оговоренная последовательность байт. Первый способ самый легкий в реализации, но он накладывает существенные ограничения на передаваемые данные. В частности, нам он не подходит, потому что мы будем передавать строки произвольной длины. Второй и третий способы приемлемы для передачи строк, и чтобы проиллюстрировать как можно больше различных вариантов в наших примерах, мы будем использовать их оба. При передаче данных от клиента серверу мы будем перед строкой передавать четырехбайтное значение — длину строки, а при передаче данных от сервера клиенту длину строки мы передавать не будем, но к каждой строке будет добавляться символ `#0`, указывающий на завершение строки. Таким образом, получается, что строки, передаваемые клиентом, могут содержать символ `#0` в середине, а передаваемые сервером — нет.

Все серверы, которые мы напишем, будут возвращать клиенту присланную им строку, но слегка преобразованную. Во-первых, все символы `#0` будут в

ней заменены на подстроку "#0", во-вторых, все буквы превращены в заглавные, а в-третьих, добавлено имя сервера, который ответил.

Практическое знакомство с TCP мы начнем с написания простейшего сервера. На компакт-диске этот сервер находится в папке SimplestServer. Сразу отметим, что это чисто учебный пример, и брать его за основу при создании реальных серверов ни в коем случае нельзя. Чуть позже мы напишем другой сервер, который уже может служить образцом для подражания.

Наш простейший сервер будет использовать только одну нить. Как мы помним, сервер должен вызывать две функции, которые блокируют работу нити: accept и recv. Очевидно, что задействовать их обе сразу в одной нити не получится, именно поэтому наш сервер сможет работать только с одним клиентом одновременно. И чтобы не блокировать пользовательский интерфейс, наш сервер будет консольным приложением. В командной строке ему передается номер порта, к которому привязывается слушающий сокет.

Первое, что должен сделать сервер, — это создать сокет, привязать его к требуемому адресу и перевести в режим прослушивания. Этот код мало чем отличается от приведенного ранее примера создания сокета для UDP (см. листинг 2.8). Вся разница только в том, что вместо сокета типа SOCK_DGRAM создается сокет типа SOCK_STREAM, а в конце еще вызывается функция listen (листинг 2.14).

Листинг 2.14. Создание сокета в программе SimplestServer

```
var
  // Порт, который будет "слушать" сервер
  Port: Word;
  // "Слушающий" сокет
  MainSocket: TSocket;
  // Сокет, создающийся для обслуживания клиента
  ClientSocket: TSocket;
  // Адрес "слушающего" сокета
  MainSockAddr: TSockAddr;
  // Адрес подключившегося клиента
  ClientSockAddr: TSockAddr;
  // Размер адреса подключившегося клиента
  ClientSockAddrLen: Integer;
  // Без этой переменной не удастся инициализировать библиотеку сокетов
  WSAData: TWSAData;
  StrLen: Integer;
  Str: string;
```

```
begin
  try
    if ParamCount = 0 then
      // Если в командной строке порт не задан, назначаем его
      Port := 12345
    else
      // В противном случае анализируем командную строку и назначаем порт
      try
        Port := StrToInt(ParamStr(1));
        if Port = 0 then
          raise ESocketException.Create(
            'Номер порта должен находиться в диапазоне 1-65535');
      except
        on EConvertError do
          raise ESocketException.Create('Параметр "' + ParamStr(1) +
            '" не является целым числом');
        on ERangeError do
          raise ESocketException.Create(
            'Номер порта должен находиться в диапазоне 1-65535');
      end;
    // Инициализация библиотеки сокетов
    if WSAStartup($101, WSAData) <> 0 then
      raise ESocketException.Create(
        'Ошибка при инициализации библиотеки WinSock');
    // Создание сокета, который затем будет "слушать" порт
    MainSocket := socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if MainSocket = INVALID_SOCKET then
      raise ESocketException.Create(
        'Невозможно создать слушающий сокет: ' + GetErrorString);
    // Формирование адреса для "слушающего" сокета
    FillChar(MainSockAddr.sin_zero, SizeOf(MainSockAddr.sin_zero), 0);
    MainSockAddr.sin_family := AF_INET;
    // Выбор IP-адреса доверяем системе
    MainSockAddr.sin_addr.S_addr := INADDR_ANY;
    // Порт назначаем, не забывая перевести его номер в сетевой формат
    MainSockAddr.sin_port := htons(Port);
    // Привязка сокета к адресу
    if bind(MainSocket, MainSockAddr, SizeOf(MainSockAddr)) =
      SOCKET_ERROR then
      raise ESocketException.Create(
        'Невозможно привязать слушающий сокет к адресу: ' +
        GetErrorString);
```

```
    // Установка сокета в режим прослушивания
    if listen(MainSocket, SOMAXCONN) = SOCKET_ERROR then
      raise ESocketException.Create(
        'Невозможно установить сокет в режим прослушивания: ' +
        GetErrorString);
    WriteLn(OemString('Сервер успешно начал прослушивание порта '),
      Port);
    ...
    // Основная часть сервера приведена в листинге 2.15
    ...
  except
    on E: ESocketException do
      WriteLn(OemString(E.Message));
    on E: Exception do
      WriteLn(OemString('Неожиданное исключение ' + E.ClassName +
        ' с сообщением ' + E.Message));
  end;
end.
```

Основная часть кода сервера — это два цикла, один из которых вложен в другой (листинг 2.15). Перед внешним циклом сервер создает сокет и переводит его в режим прослушивания, и внешний цикл начинается с вызова функции accept. Завершение accept указывает на подключение клиента. После этого начинается внутренний цикл, который состоит из получения сообщений от клиента, преобразования строки и отправки ответа. Внутренний цикл завершается, когда соединение разрывается либо самим клиентом, либо из-за ошибки в сети. После этого управление вновь передается на начало внешнего цикла, т. е. на accept, и сервер может принять подключение другого клиента (или повторное подключение того же клиента).

Листинг 2.15. Основная часть сервера SimplestServer

```
// Начало цикла подключения и общения с клиентом
repeat
  ClientSockAddrLen := SizeOf(ClientSockAddr);
  // Принимаем подключившегося клиента. Для общения с ним создается новый
  // сокет, дескриптор которого помещается в ClientSocket.
  ClientSocket := accept(MainSocket, @ClientSockAddr,
    @ClientSockAddrLen);
  if ClientSocket = INVALID_SOCKET then
    raise ESocketException.Create(
      'Ошибка при ожидании подключения клиента: ' + GetErrorString);
```

```
// При выводе сообщения не забываем,
// что номер порта имеет сетевой формат
WriteLn(OemString('Зафиксировано подключение с адреса '),
  Ord(ClientSockAddr.sin_addr.S_un_b.s_b1), '.',
  Ord(ClientSockAddr.sin_addr.S_un_b.s_b2), '.',
  Ord(ClientSockAddr.sin_addr.S_un_b.s_b3), '.',
  Ord(ClientSockAddr.sin_addr.S_un_b.s_b4), ':',
  ntohs(ClientSockAddr.sin_port));
// Цикл общения с клиентом. Продолжается до тех пор,
// пока клиент не закроет соединение или пока
// не возникнет ошибка
repeat
  // Читаем длину присланной клиентом строки и помещаем ее в StrLen
  case ReadFromSocket(ClientSocket, StrLen, SizeOf(StrLen)) of
    0:
      begin
        WriteLn(OemString('Клиент закрыл соединение'));
        Break;
      end;
    -1:
      begin
        WriteLn(OemString('Ошибка при получении данных от клиента: ' +
          GetErrorString));
        Break;
      end;
  end;
  // Протокол не допускает строк нулевой длины
  if StrLen <= 0 then
  begin
    WriteLn(OemString('Неверная длина строки от клиента: '), StrLen);
    Break;
  end;
  // Установка длины строки в соответствии с полученным значением
  SetLength(Str, StrLen);
  // Чтение строки нужной длины
  case ReadFromSocket(ClientSocket, Str[1], StrLen) of
    0:
      begin
        WriteLn(OemString('Клиент закрыл соединение'));
        Break;
      end;
    -1:
      begin
```

```
          WriteLn(OemString('Ошибка при получении данных от клиента: ' +
            GetErrorString));
          Break;
        end;
    end;
    WriteLn(OemString('Получена строка: ' + Str));
    // Преобразование строки
    Str := AnsiUpperCase(StringReplace(Str, #0, '#0', [rfReplaceAll])) +
      ' (Simplest server)';
    // Отправка строки. Отправляется на один байт больше, чем
    // длина строки, чтобы завершающий символ #0 тоже попал в пакет
    if send(ClientSocket, Str[1], Length(Str) + 1, 0) < 0 then
    begin
      WriteLn(OemString('Ошибка при отправке данных клиенту: ' +
        GetErrorString));
      Break;
    end;
    WriteLn(OemString('Клиенту отправлен ответ: ' + Str));
  // Завершение цикла обмена с клиентом
  until False;
  // Сокет для связи с клиентом больше не нужен
  closesocket(ClientSocket);
until False;
```

Теперь перейдем к написанию клиента. Пример этого клиента находится на компакт-диске в папке SimpleClient, главное окно показано на рис. 2.4. Клиент должен вызывать только одну функцию, которая реально может блокировать вызвавшую ее нить, — функцию recv. Но по нашему протоколу сервер не посылает клиенту ничего по собственной инициативе, он только отвечает на сообщения клиента. Следовательно, клиент не должен быть всегда готов принять сообщение, он его принимает только после отправки своего. В простых случаях, когда сообщение имеет небольшой размер, а формирование ответа на сервере не требует длительной работы, мы можем считать, что попытка получения ответа от сервера сразу же после отправки ему сообщения в подавляющем большинстве случаев не будет блокировать работу клиента, а оставшееся незначительное количество случаев считаем форс-мажором и допускаем, что в такой ситуации блокирование будет допустимо. На практике заметить это блокирование можно будет только тогда, когда сервер не будет должным образом отвечать на сообщения или связь с ним будет потеряна. Для простого клиента с невысокими требованиями к надежности такое упрощение вполне допустимо и вполне может быть использовано на практике. А в дальнейшем мы познакомимся со средствами библиотеки сокетов, позволяющими писать программы, в которых работа с сокетами никогда не приводит к блокировке.

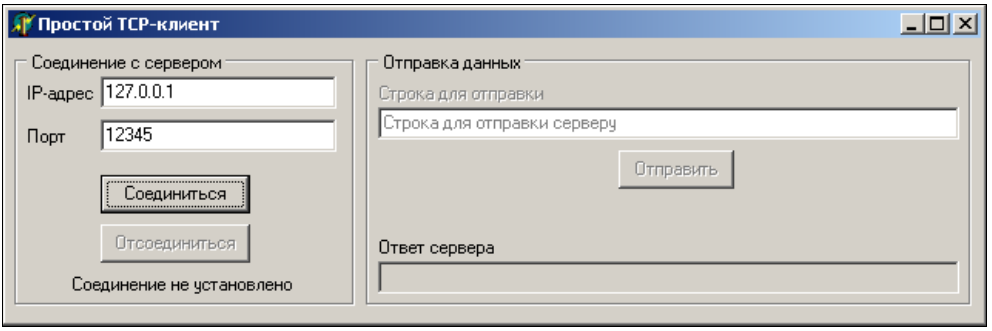


Рис. 2.4. Главное окно программы SimpleClient

Таким образом, наш клиент будет очень простым: по кнопке **Соединиться** он будет соединяться с сервером, по кнопке **Отправить** — отправлять серверу сообщение и дожидаться ответа. Третья кнопка, **Отсоединиться**, служит для корректного завершения работы с сервером. Рассмотрим эти действия подробнее.

При соединении с сервером клиент должен создать сокет и вызвать функцию connect. Здесь мы не можем создать сокет один раз и потом пользоваться им на протяжении всего времени работы клиента, т. к. после закрытия соединения (неважно, корректного или из-за ошибки) сокет больше нельзя использовать. Поэтому при установлении соединения каждый раз приходится создавать новый сокет. Обработчик нажатия кнопки **Соединиться** приведен в листинге 2.16.

Листинг 2.16. Обработчик нажатия кнопки *Соединиться*

```
procedure TSimpleClientForm.BtnConnectClick(Sender: TObject);
var
  // Адрес сервера
  ServerAddr: TSockAddr;
begin
  // Формируем адрес сервера, к которому нужно подключиться
  FillChar(ServerAddr.sin_zero, SizeOf(ServerAddr.sin_zero), 0);
  ServerAddr.sin_family := AF_INET;
  ServerAddr.sin_addr.S_addr := inet_addr(PChar(EditIPAddress.Text));
  // Для совместимости со старыми версиями Delphi приводим
  // константу INADDR_NONE к типу u_long
  if ServerAddr.sin_addr.S_addr = u_long(INADDR_NONE) then
  begin
    MessageDlg('Синтаксическая ошибка в IP-адресе', mtError, [mbOK], 0);
    Exit;
  end;
```

```
  try
    ServerAddr.sin_port := htons(StrToInt(EditPort.Text));
    // Создание сокета
    FSocket := socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if FSocket = INVALID_SOCKET then
    begin
      MessageDlg('Ошибка при создании сокета: '#13#10 + GetErrorString,
        mtError, [mbOK], 0);
      Exit;
    end;
    // Подключение к серверу
    if connect(FSocket, ServerAddr, SizeOf(ServerAddr)) < 0 then
    begin
      MessageDlg('Ошибка при установлении подключения: '#13#10 +
        GetErrorString, mtError, [mbOK], 0);
      // Так как сокет был успешно создан,
      // в случае ошибки его нужно удалить
      closesocket(FSocket);
      FSocket := 0;
      Exit;
    end;
    // Включаем режим "Соединение установлено"
    OnConnect;
  except
    on EConvertError do
      // Это исключение может возникнуть только в одном месте -
      // при вызове StrToInt(EditPort.Text)
      MessageDlg('"' + EditPort.Text + '" не является целым числом',
        mtError, [mbOK], 0);
    on ERangeError do
      // Это исключение может возникнуть только в одном месте -
      // при присваивании значения номеру порта
      MessageDlg('Номер порта должен находиться в диапазоне 1-65535',
        mtError, [mbOK], 0);
  end;
end;
```

Теперь посмотрим, как клиент реагирует на нажатие кнопки **Отправить** (листинг 2.17). Сама по себе отправка — вещь очень простая: нужно сформировать адрес получателя и вызвать функцию send. Несколько сложнее выполняется чтение данных, потому что, согласно нашему протоколу, клиент не знает, сколько байтов он должен прочитать, и читает до тех пор, пока не встретит символ #0.

Листинг 2.17. Обработчик нажатия кнопки *Отправить*

```
procedure TSimpleClientForm.BtnSendClick(Sender: TObject);
const
  // Данные из буфера сокета мы будем читать порциями.
  // Константа BufStep определяет размер порции
  BufStep = 10;
var
  Str: string;
  StrLen, BufStart, Portion: Integer;
  Buf: array of Char;
begin
  Str := EditStringToSend.Text;
  StrLen := Length(Str);
  if StrLen = 0 then
  begin
    MessageDlg('Протокол не допускает отправки пустых строк', mtError,
      [mbOK], 0);
    Exit;
  end;
  // Отправляем серверу длину строки
  if send(FSocket, StrLen, SizeOf(StrLen), 0) < 0 then
  begin
    MessageDlg('Ошибка при отправке данных серверу: '#13#10 +
      GetErrorString, mtError, [mbOK], 0);
    OnDisconnect;
    Exit;
  end;
  // Отправляем серверу строку
  if send(FSocket, Str[1], StrLen, 0) < 0 then
  begin
    MessageDlg('Ошибка при отправке данных серверу: '#13#10 +
      GetErrorString, mtError, [mbOK], 0);
    OnDisconnect;
    Exit;
  end;
  BufStart := 0;
  // Цикл получения ответа от сервера.
  // Завершается, когда получаем посылку, оканчивающуюся на #0
  repeat
    SetLength(Buf, Length(Buf) + BufStep);
    // Читаем очередную порцию ответа от сервера
    Portion := recv(FSocket, Buf[BufStart], BufStep, 0);
```

```
    if Portion <= 0 then
    begin
      MessageDlg('Ошибка при получении ответа от сервера: '#13#10 +
        GetErrorString, mtError, [mbOK], 0);
      OnDisconnect;
      Exit;
    end;
    // Если порция кончается на #0, ответ прочитан полностью, выходим из
    // цикла. Здесь мы использовали особенность нашего протокола, который
    // запрещает серверу присылать несколько строк подряд, следующая
    // строка будет выслана сервером только после нового запроса от
    // клиента. Если бы протокол допускал отправку сервером нескольких
    // ответов подряд, при чтении очередной порции данных могло бы
    // оказаться, что начало порции принадлежит одной строке, конец -
    // следующей, а признак конца строки нужно искать где-то в середине.
    if Buf[BufStart + Portion — 1] = #0 then
    begin
      EditReply.Text := PChar(@Buf[0]);
      Break;
    end;
    Inc(BufStart, BufStep);
  until False;
end;
```

Реакция на кнопку **Отсоединиться** совсем простая: нужно разорвать соединение и закрыть сокет (листинг 2.18).

Листинг 2.18. Реакция на нажатие кнопки *Отсоединиться*

```
procedure TSimpleClientForm.BtnDisconnectClick(Sender: TObject);
begin
  shutdown(FSocket, SD_BOTH);
  closesocket(FSocket);
  OnDisconnect;
end;
```

Откомпилируем наши примеры и посмотрим, что получилось. Пока у нас один клиент работает с одним сервером, все вполне предсказуемо: клиент передает сообщения, сервер на них отвечает. Попытаемся подключиться вторым клиентом, не отключая первый, и посмотрим, что будет. Само подключение с точки зрения клиента проходит нормально, хотя сервер находится в своем внутреннем цикле и не вызывает accept для второго клиента. Впрочем, как мы знаем, для успешного выполнения функции connect на стороне клиен-

та достаточно, чтобы сокет сервера находился в режиме прослушивания. Теперь попытаемся отправить что-то серверу со второго клиента. Сама отправка проходит успешно, но при попытке получить ответ клиент "зависает", т. к. функция recv блокирует нить до прихода данных, а данные не приходят, потому что сервер не обрабатывает сообщения от этого клиента. Отсоединим первый клиент от сервера, чтобы сервер вернулся к выполнению функции accept. Мы видим, что сервер немедленно обнаружил подключение второго клиента, а также то, что клиент прислал ему сообщение. Соответственно, сервер отвечает на это сообщение, и второй клиент "отвисает" — теперь с ним можно нормально работать.

Простейший сервер и эксперименты с ним, конечно, очень познавательны, но на практике хотелось бы иметь такой сервер, который может работать одновременно с несколькими клиентами. Чтобы добиться этого, сделаем так же, как при написании UDP-чата: вынесем в отдельные нити работу с блокирующими функциями (пример MultithreadedServer на компакт-диске). Нам понадобится одна нить для выполнения функции accept и по одной нити на работу с каждым подключившимся клиентом. Инициализация выполняется при нажатии кнопки **Запустить** (листинг 2.19). После инициализации библиотеки сокетов, создания сокета и перевода его в режим прослушивания она создает нить типа TListenThread, передает ей дескриптор сокета и больше с сокетами не работает — дальнейшая роль главной нити заключается только в обработке сообщений. Благодаря этому сервер может иметь нормальный пользовательский интерфейс.

Листинг 2.19. Инициализация многонитевого сервера

```
// Реакция на кнопку "Запустить" — запуск сервера
procedure TServerForm.BtnStartServerClick(Sender: TObject);
var
  // Сокет, который будет "слушать"
  ServerSocket: TSocket;
  // Адрес, к которому привязывается слушающий сокет
  ServerAddr: TSockAddr;
begin
  // Формируем адрес для привязки.
  FillChar(ServerAddr.sin_zero, SizeOf(ServerAddr.sin_zero), 0);
  ServerAddr.sin_family := AF_INET;
  ServerAddr.sin_addr.S_addr := INADDR_ANY;
  try
    ServerAddr.sin_port := htons(StrToInt(EditPortNumber.Text));
    if ServerAddr.sin_port = 0 then
```

```
    begin
      MessageDlg('Номер порта должен находиться в диапазоне 1-65535',
        mtError, [mbOK], 0);
      Exit;
    end;
    // Создание сокета
    ServerSocket := socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if ServerSocket = INVALID_SOCKET then
    begin
      MessageDlg('Ошибка при создании сокета:'#13#10 + GetErrorString,
        mtError, [mbOK], 0);
      Exit;
    end;
    // Привязка сокета к адресу
    if bind(ServerSocket, ServerAddr, SizeOf(ServerAddr)) =
      SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при привязке сокета к адресу:'#13#10 +
        GetErrorString, mtError, [mbOK], 0);
      closesocket(ServerSocket);
      Exit;
    end;
    // Перевод сокета в режим прослушивания
    if listen(ServerSocket, SOMAXCONN) = SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при переводе сокета в режим прослушивания:' +
        #13#10 + GetErrorString, mtError, [mbOK], 0);
      closesocket(ServerSocket);
      Exit;
    end;
    // Запуск нити, обслуживающей слушающий сокет
    TListenThread.Create(ServerSocket);
    // Перевод элементов управления в состояние "Сервер работает"
    LabelPortNumber.Enabled := False;
    EditPortNumber.Enabled := False;
    BtnStartServer.Enabled := False;
    LabelServerState.Caption := 'Сервер работает';
  except
    on EConvertError do
      // Это исключение может возникнуть только в одном месте -
      // при вызове StrToInt(EditPortNumber.Text)
      MessageDlg('"' + EditPortNumber.Text +
        '" не является целым числом', mtError, [mbOK], 0);
```

```
    on ERangeError do
      // Это исключение может возникнуть только в одном месте -
      // при присваивании значения номеру порта
      MessageDlg('Номер порта должен находиться в диапазоне 1-65535',
        mtError, [mbOK], 0);
  end;
end;
```

"Слушающая" нить `TListenThread` состоит из бесконечного ожидания подключения клиента. Каждый раз при подключении клиента библиотека сокетов создает новый сокет, и для работы с ним создается новая нить типа `TClientThread` (листинг 2.20).

Листинг 2.20. Код "слушающей" нити

```
procedure TListenThread.Execute;
var
  // Сокет, созданный для общения с подключившимся клиентом
  ClientSocket: TSocket;
  // Адрес подключившегося клиента
  ClientAddr: TSockAddr;
  ClientAddrLen: Integer;
begin
  // Начинаем бесконечный цикл
  repeat
    ClientAddrLen := SizeOf(ClientAddr);
    // Ожидаем подключения клиента
    ClientSocket := accept(FServerSocket, @ClientAddr, @ClientAddrLen);
    if ClientSocket = INVALID_SOCKET then
    begin
      // Ошибка в функции accept возникает только тогда, когда
      // происходит нечто экстраординарное. Продолжать работу
      // в этом случае бессмысленно.
      LogMessage('Ошибка при подключении клиента: ' + GetErrorString);
      Break;
    end;
    // Создаем новую нить для обслуживания подключившегося клиента
    // и передаем ей сокет, созданный для взаимодействия с ним.
    TClientThread.Create(ClientSocket, ClientAddr);
  until False;
  closesocket(FServerSocket);
  LogMessage('Сервер завершил работу');
  Synchronize(ServerForm.OnStopServer);
end;
```

Метод LogMessage, существующий у "слушающей" нити, эквивалентен тому, который приведен в листинге 2.7.

Код нити типа TClientThread, которая отвечает за взаимодействие с одним клиентом, приведен в листинге 2.21.

Листинг 2.21. Код нити, реализующей взаимодействие с клиентом

```
// Сокет для взаимодействия с клиентом создается в главной нити,
// а сюда передается через параметр конструктора. Для формирования
// заголовка сюда же передается адрес подключившегося клиента
constructor TClientThread.Create(ClientSocket: TSocket; const
ClientAddr:TSockAddr);
begin
  FSocket := ClientSocket;
  // Заголовок содержит адрес и номер порта клиента.
  // Этот заголовок будет добавляться ко всем сообщениям в лог
  // от данного клиента.
  FHeader := 'Сообщение от клиента ' + inet_ntoa(ClientAddr.sin_addr) +
    ':' + IntToStr(ntohs(ClientAddr.sin_port)) + ': ';
  inherited Create(False);
end;

procedure TClientThread.Execute;
var
  Str: string;
  StrLen: Integer;
begin
  LogMessage('Соединение установлено');
  // Начинаем цикл, из которого выходим только при закрытии
  // соединения клиентом или из-за ошибки в сети.
  repeat
    // Читаем длину присланной клиентом строки и помещаем ее в StrLen
    case ReadFromSocket(FSocket, StrLen, SizeOf(StrLen)) of
      0:
        begin
          LogMessage('Клиент закрыл соединение');
          Break;
        end;
      -1:
        begin
          LogMessage('Ошибка при получении данных от клиента: ' +
            GetErrorString);
```

```
          Break;
        end;
    end;
    // Протокол не допускает строк нулевой длины
    if StrLen <= 0 then
    begin
      LogMessage('Неверная длина строки от клиента: ' +
        IntToStr(StrLen));
      Break;
    end;
    // Установка длины строки в соответствии с полученным значением
    SetLength(Str, StrLen);
    // Чтение строки нужной длины
    case ReadFromSocket(FSocket, Str[1], StrLen) of
      0:
        begin
          LogMessage('Клиент закрыл соединение');
          Break;
        end;
      -1:
        begin
          LogMessage('Ошибка при получении данных от клиента: ' +
            GetErrorString);
          Break;
        end;
    end;
    LogMessage('Получена строка: ' + Str);
    // Преобразование строки
    Str := AnsiUpperCase(StringReplace(Str, #0, '#0', [rfReplaceAll])) +
      ' (Multithreaded server)';
    // Отправка строки. Отправляется на один байт больше, чем
    // длина строки, чтобы завершающий символ #0 тоже попал в пакет
    if send(FSocket, Str[1], Length(Str) + 1, 0) < 0 then
    begin
      LogMessage('Ошибка при отправке данных клиенту: ' +
        GetErrorString);
      Break;
    end;
    LogMessage('Клиенту отправлен ответ: ' + Str);
  until False;
  closesocket(FSocket);
end;
```

```
procedure TClientThread.LogMessage(const Msg: string);
begin
  FMessage := FHeader + Msg;
  Synchronize(DoLogMessage);
end;
```

Метод LogMessage здесь несколько видоизменен по сравнению с предыдущими примерами: к каждому сообщению он добавляет адрес клиента, чтобы пользователь мог видеть, с каким именно из одновременно подключившихся клиентов связано сообщение. Что же касается кода Execute, то видно, что он практически не отличается от кода внутреннего цикла простейшего сервера (см. листинг 2.15). Это неудивительно — сообщение здесь читается и обрабатывается единым образом. Вся разница только в том, что теперь у нас одновременно могут работать несколько таких нитей, обеспечивая одновременную работу сервера с несколькими клиентами.

Этот сервер уже можно использовать как образец для подражания. Нужно только помнить, что он тратит на каждого клиента относительно много ресурсов, и поэтому не подходит там, где могут подключаться сотни и более клиентов одновременно. Кроме того, этот сервер очень уязвим по отношению к DoS-атакам, поэтому подобный сервер целесообразен там, где число клиентов относительно невелико, а вероятность DoS-атак низка.

Примечание

DoS-атака (Denied of Service) — способ помешать функционированию сервера, заключающийся в загрузке его бесполезной работой. В простейшем случае — это просто одновременное подключение большого числа клиентов. У нас даже простое подключение большого числа клиентов приводит к большому расходу системных ресурсов, поэтому DoS-атакой можно добиться неработоспособности не только самого сервера, но и системы в целом. Полностью защититься от DoS-атаки невозможно, но можно снизить урон, наносимый ею. Об этом мы поговорим далее.

2.1.13. Определение готовности сокета

Так как многие функции библиотеки сокетов блокируют вызвавшую их нить, если соответствующая операция не может быть выполнена немедленно, часто бывает полезно заранее знать, готов ли сокет к немедленному (без блокирования) выполнению той или иной операции. Основным средством определения этого в библиотеке сокетов служит функция select:

```
function select(nfds: Integer; readfds, writefds, exceptfds: PFDSet;
  timeout: PTimeVal): LongInt;
```

Первый параметр этой функции оставлен только для совместимости со старыми версиями библиотеки сокетов; в существующих версиях он игнориру-

ется. Три следующих параметра содержат указатели на множества сокетов (эти множества описываются типом TFDSet), состояние которых должно проверяться. В данном случае понятие множества не имеет ничего общего с типом множество в Delphi. В оригинальной версии библиотеки сокетов, написанной на C, определены макросы, позволяющие очищать такие множества, добавлять и удалять сокеты и определять, входит ли тот или иной сокет в множество. В модуле WinSock эти макросы заменены одноименными процедурами и функциями (листинг 2.22).

Листинг 2.22. Функции для работы с типом TFDSet

```
// Удаляет сокет Socket из множества FDSet.
procedure FD_CLR(Socket: TSocket; var FDSet: TFDSet);

// Определяет, входит ли сокет Socket в множество FDSet.
function FD_ISSET(Socket: TSocket; var FDSet: TFDSet): Boolean;

// Добавляет сокет Socket в множество FDSet.
procedure FD_SET(Socket: TSocket; var FDSet: TFDSet);

// Инициализирует множество FDSet.
procedure FD_ZERO(var FDSet: TFDSet);
```

При создании переменной типа TFDSet в той области памяти, которую она занимает, могут находиться произвольные данные, являющиеся, по сути дела, "мусором". Из-за этого мусора функции FD_CLR, FD_ISSET и FD_SET не смогут работать корректно. Процедура FD_ZERO очищает мусор, создавая пустое множество. Вызов остальных функций FD_XXX до вызова FD_ZERO приведет к непредсказуемым результатам.

Мы намеренно не приводим здесь описание внутренней структуры типа TFDSet. С помощью функций FD_XXX можно выполнить все необходимые операции с множеством, не зная этой структуры. Отметим, что в Windows и в Unix внутреннее устройство этого типа существенно различается, но благодаря использованию этих функций код остается переносимым.

В Windows максимальное количество сокетов, которое может содержать в себе множество TFDSet, определяется значением константы FD_SETSIZE. По умолчанию ее значение равно 64. В C/C++ отсутствует раздельная компиляция модулей в том смысле, в котором она существует в Delphi, поэтому модуль в этих языках может поменять значение константы FD_SETSIZE перед включением заголовочного файла библиотеки сокетов, и это изменение приведет к изменению внутренней структуры типа TFDSet (точнее, типа FDSET — в C/C++ он называется так). К счастью, в Delphi модули надежно защищены

от подобного влияния друг на друга, поэтому как бы мы ни переопределяли константу FD_SETSIZE в своем модуле, на модуле WinSock это никак не отразится. В Delphi приходится прибегать к другому способу изменения количества сокетов в множестве: для этого следует определить свой тип, эквивалентный по структуре TFDSet, но резервирующий иное количество памяти для хранения сокетов (структуру TFDSet можно узнать из исходного кода модуля WinSock). В функцию select можно передавать указатели на структуры нового типа, необходимо только приведение типов указателей. А вот существующие функции FD_XXX, к сожалению, не смогут работать с новой структурой, потому что компилятор требует строгого соответствия типов для параметров-переменных. Но, опять же, при необходимости очень легко создать аналоги этих функций для своей структуры.

Примечание

На первый взгляд может показаться, что Delphi в данном случае хуже, чем C/C++. Но достаточно хотя бы раз столкнуться с ошибкой, вызванной взаимным влиянием макроопределений в модулях C/C++, чтобы понять, что уж лучше написать несколько лишних строк кода, лишь бы никогда больше не иметь таких проблем.

Последний параметр функции select содержит указатель на структуру TTimeVal, которая описывается следующим образом:

```
TTimeVal = record
  tv_sec: Longint;
  tv_usec: Longint;
end;
```

Эта структура служит для задания времени ожидания. Поле tv_sec содержит число полных секунд в этом интервале, поле tv_usec — число микросекунд. Так, чтобы задать интервал ожидания, равный 1,5 с, нужно присвоить полю tv_sec значение 1, а полю tv_usec — значение 500 000. Параметр timeout функции select должен содержать указатель на заполненную подобным образом структуру, определяющую, сколько времени функция будет ожидать, пока хотя бы один из сокетов не будет готов к требуемой операции. Если этот указатель равен nil, ожидание будет бесконечным.

Мы потратили достаточно много времени, выясняя структуру параметров функции select. Теперь, наконец-то, можно перейти к описанию того, зачем она нужна и какой смысл несет каждый из ее параметров.

Функция select позволяет дождаться, когда хотя бы один из сокетов, переданный в одном из множеств, будет готов к выполнению той или иной операции. Какой именно операции, определяется тем, в какое из трех множеств входит сокет. Для сокетов, входящих в множество readfds, готовность озна-

чает, что функции recv или recvfrom будут выполнены без блокирования. В случае UDP это означает, что во входном буфере сокета есть данные, которые можно прочитать. При использовании TCP функции recv и recvfrom могут быть выполнены без задержки еще в двух случаях: когда партнер закрыл соединение (в этом случае функции вернут 0), а также когда соединение некорректно разорвано (в этом случае функции вернут SOCKET_ERROR). Кроме того, если сокет, включенный в множество readfds, находится в состоянии ожидания соединения (в которое он переведен с помощью функции listen), то для него состояние готовности означает, что очередь соединений не пуста и функция accept будет выполнена без задержек.

Для сокетов, входящих в множество writefds, готовность означает, что сокет соединен, а в его выходном буфере есть свободное место. (До сих пор мы обсуждали только блокирующие сокеты, для которых успешное завершение функции connect автоматически означает, что сокет соединен. Далее мы познакомимся с неблокирующими сокетами, для которых нужно вызвать функцию select, чтобы понять, установлено ли соединение.) Наличие свободного места в буфере не гарантирует того, что функции send или sendto не будут блокировать вызвавшую их нить, т. к. программа может попытаться передать больший объем информации, чем размер свободного места в буфере на момент вызова функции. В этом случае функции send и sendto вернут управление вызвавшей их нити только после того, как часть данных будет отправлена, и в буфере сокета освободится достаточно места.

Следует отметить, что большинство протоколов обмена устроено таким образом, что при их реализации проблема переполнения выходного буфера практически никогда не возникает. Чаще всего клиент и сервер обмениваются небольшими пакетами, причем сервер посылает клиенту только ответы на его запросы, а клиент не посылает новый запрос до тех пор, пока не получит ответ на предыдущий. В этом случае гарантируется, что пакеты будут уходить из выходного буфера быстрее (или, по крайней мере, не медленнее), чем программа будет их туда помещать. Поэтому заботиться о том, чтобы в выходном буфере было место, приходится достаточно редко.

И наконец, последнее множество exceptfds. Для сокетов, входящих в это множество, состояние готовности означает либо неудачу попытки соединения для неблокирующего сокета, либо получение *высокоприоритетных данных* (out-of-band data). В этой книге мы не будем детально рассматривать отправку и получение высокоприоритетных данных. Те, кому это понадобится, легко разберутся с этим вопросом по MSDN.

Функция select возвращает общее количество сокетов, находящихся в состоянии готовности. Если функция завершила работу по тайм-ауту, возвращается 0. Множества readfds, writefds и exceptfds модифицируются функци-

ей: в них остаются только те сокеты, которые находятся в состоянии готовности. При вызове функции любые два из этих трех указателей могут быть равны nil, если программу не интересует готовность сокетов по соответствующим критериям. Один и тот же сокет может входить в несколько множеств.

В листинге 2.23 приведен пример кода TCP-сервера, взаимодействующего с несколькими клиентами в рамках одной нити и работающего по простой схеме "запрос-ответ".

Листинг 2.23. Пример сервера, использующего select

```
var
  Sockets: array of TSocket;
  Addr: TSockAddr;
  Data: TWSAData;
  Len, I, J: Integer;
  FDSet: TFDSet;
begin
  WSAStartup($101, Data);
  SetLength(Sockets, 1);
  Sockets[0] := socket(AF_INET, SOCK_STREAM, 0);
  Addr.sin_family := AF_INET;
  Addr.sin_port := htons(5514);
  Addr.sin_addr.S_addr := INADDR_ANY;
  FillChar(Addr.sin_zero, SizeOf(Addr.sin_zero), 0);
  bind(Sockets[0], Addr, SizeOf(TSockAddr));
  listen(Sockets[0], SOMAXCONN);
  while True do
  begin
    // 1. Формирование множества сокетов
    FD_ZERO(FDSet);
    for I := 0 to High(Sockets) do
      FD_SET(Sockets[I], FDSet);
    // 2. Проверка готовности сокетов
    select(0, @FDSet, nil, nil, nil);
    // 3. Чтение запросов клиентов тех сокетов, которые готовы к этому
    I := 1;
    while I <= High(Sockets) do
    begin
      if FD_ISSET(Sockets[I], FDSet) then
        if recv(Sockets[I], ...) <= 0 then
        begin
          // Связь разорвана, нужно закрыть сокет
          // и удалить его из массива
```

```
          closesocket(Sockets[I]);
          for J := I to High(Sockets) – 1 do
            Sockets[J] := Sockets[J + 1];
          Dec(I);
          SetLength(Sockets, Length(Sockets) – 1);
        end
        else
        begin
          // Получены данные от клиента, нужно ответить
          send(Sockets[I], ...);
        end;
      Inc(I);
    end;
    // 4. Проверка подключения нового клиента
    if FD_ISSET(Sockets[0], FDSet) then
    begin
      // Подключился новый клиент
      SetLength(Sockets, Length(Sockets) + 1);
      Len := SizeOf(TSockAddr);
      Sockets[High(Sockets)] := accept(Sockets[0], @Addr, @Len)
    end;
  end;
end;
```

Как и в предыдущих примерах, код для краткости не содержит проверок успешности завершения функций. Еще раз напоминаем, что в реальном коде такие проверки необходимы.

Теперь разберем программу по шагам. Создание сокета, привязка к адресу и перевод в режим ожидания подключений вам уже знакомы, поэтому мы на них останавливаться не будем. Отметим только, что вместо переменной типа `TSocket` мы формируем динамический массив этого типа, длина которого сначала устанавливается равной одному элементу, и этот единственный элемент и содержит дескриптор созданного сокета. В дальнейшем мы будем добавлять в этот массив сокеты, создающиеся в результате выполнения функции `accept`. После перевода сокета в режим ожидания подключения начинается бесконечный цикл, состоящий из четырех шагов.

На первом шаге цикла создается множество сокетов, в которое добавляются все сокеты, содержащиеся в массиве. В этом месте в примере пропущена важная проверка того, что сокетов в массиве не больше 64-х. Если их будет больше, то попытки добавить лишние сокеты в множество будут проигнорированы функцией `FD_SET` и, соответственно, эти сокеты выпадут из дальней-

шего рассмотрения, т. е. даже если клиент что-то пришлет, сервер этого не увидит. Решить проблему можно тремя способами. Самый простой — это отказывать в подключении лишним клиентам. Для этого сразу после вызова accept нужно вызывать для нового сокета closesocket. Второй способ — это увеличение количества сокетов в множестве, как это было описано ранее. В этом случае все равно остается та же проблема, хотя если сделать число сокетов в множестве достаточно большим, она практически исчезает. И наконец, можно разделить сокеты на несколько порций, для каждой из которых вызывать select отдельно. Это потребует усложнения примера, потому что сейчас в функции select мы используем бесконечное ожидание. При разбиении сокетов на порции это может привести к тому, что из-за отсутствия готовых сокетов в первой порции программа не сможет перейти к проверке второй порции, в которой готовые сокеты, может быть, есть. Пример разделения сокетов на порции будет рассмотрен в следующем разделе.

При создании множества оно сначала очищается, а потом в него в цикле добавляются сокеты. Для любителей кратких решений есть существенно более быстрый способ формирования множества, при котором не потребуются ни циклы, ни FD_ZERO, ни FD_SET:

```
Move((PChar(Sockets) – 4)^, FDSet,
  Length(Sockets) * SizeOf(TSocket) + SizeOf(Integer));
```

Почему такая конструкция будет работать, предлагаем разобраться самостоятельно, изучив по справке Delphi, как хранятся в памяти динамические массивы, а по MSDN — структуру типа FDSET. Тем же, кто по каким-то причинам не захочет разбираться, настоятельно рекомендуем никогда и ни при каких обстоятельствах не использовать такую конструкцию, потому что в неумелых руках она превращается в мину замедленного действия, из-за которой ошибки могут появиться в самых неожиданных местах программы.

Второй шаг — это собственно выполнение ожидания готовности сокетов с помощью функции select. Готовность к записи и к чтению высокоприоритетной информации нас в данном случае не интересует, поэтому мы ограничиваемся заданием множества readfds. В нашем простом примере не должно выполняться никаких действий, если ни один сокет не готов, поэтому последний параметр тоже равен nil, что означает ожидание, не ограниченное тайм-аутом.

Третий шаг выполняется только после функции select, т. е. тогда, когда хотя бы один из сокетов находится в состоянии готовности. На этом шаге мы проверяем сокеты, созданные для взаимодействия с клиентами на предыдущих итерациях цикла с помощью функции accept. Эти сокеты располагаются в массиве сокетов, начиная с элемента с индексом 1. Программа в цикле про-

сматривает все сокеты и, если они находятся в состоянии готовности, выполняет операцию чтения.

На первый взгляд может показаться странным, почему для перебора элементов массива выбран цикл `while`, а не `for`. Но в дальнейшем мы увидим, что размер массива во время выполнения цикла может изменяться. Особенность же цикла `for` заключается в том, что его границы вычисляются один раз и запоминаются в отдельных ячейках памяти, и дальнейшее изменение значений выражений, задающих эти границы, не изменяет эти границы. В нашем примере это приведет к тому, что в случае уменьшения массива цикл `for` не остановится на реальной уменьшившейся длине, а продолжит выполнение по уже не существующим элементам, что приведет к трудно предсказуемым последствиям. Поэтому в данном случае предпочтительнее цикл `while`, в котором условие продолжения цикла заново вычисляется при каждой его итерации.

Напомним, что функция `select` модифицирует переданные ей множества таким образом, что в них остаются лишь сокеты, находящиеся в состоянии готовности. Поэтому чтобы проверить, готов ли конкретный сокет, достаточно с помощью функции `FD_ISSET` проверить, входит ли он в множество `FDSet`. Если входит, то вызываем для него функцию `recv`. Если эта функция возвращает положительное значение, значит, данные в буфере есть, программа их читает и отвечает. Если функция возвращает 0 или –1 (`SOCKET_ERROR`), значит, соединение закрыто или разорвано, и данный сокет больше не может быть использован. Поэтому мы должны освободить связанные с ним ресурсы (`closesocket`) и убрать его из массива сокетов (как раз на этом шаге размер массива уменьшается). При удалении оставшиеся сокеты смещаются на одну позицию влево, поэтому переменную цикла необходимо уменьшить на единицу, иначе следующий сокет будет пропущен.

И наконец, на четвертом шаге мы проверяем состояние готовности исходного сокета, который хранится в нулевом элементе массива. Так как этот сокет находится в режиме ожидания соединения, для него состояние готовности означает, что в очереди соединений появились клиенты, и необходимо вызвать функцию `accept`, чтобы создать сокеты для взаимодействия с этими клиентами.

Хотя приведенный пример вполне работоспособен, следует отметить, что это только один из возможных вариантов организации сервера. Так что лучше не относиться к нему как к догме, потому что именно в вашем случае может оказаться предпочтительнее какой-либо другой вариант. Ценность этого примера заключается в том, что он иллюстрирует работу функции `select`, а не в том, что он дает готовое решение на все случаи жизни.

2.1.14. Примеры использования функции *select*

Рассмотрим два практических примера использования функции select для получения информации о готовности сокета. Оба примера станут развитием рассмотренных ранее.

Сначала модифицируем UDP-чат *(см. разд. 2.1.10)* таким образом, чтобы он использовал один сокет и для отправки, и для получения сообщений (пример SelectChat на компакт-диске). Вторая нить нам теперь не понадобится, все будет делать главная форма. Процедуры создания сокета и отправки сообщений изменений не претерпели, главное дополнение — это то, что на форме появился таймер, в обработчике события OnTimer которого мы будем проверять с помощью select, пришло ли сообщение для сокета (листинг 2.24). С помощью таких простейших модификаций мы получили чат, который работает без распараллеливания и использует всего один сокет. Работать с таким чатом стало намного проще, потому что теперь ответ нужно посылать на тот же порт, с которого пришло сообщение, а не запоминать, какой порт для отправки какому из экземпляров чата соответствует.

Примечание

Несмотря на эти изменения, новая версия UDP-чата может обмениваться сообщениями со старой, т. к. протокол обмена остался неизменным.

Листинг 2.24. Проверка готовности сокетов при обработке сообщения от таймера

```
// Реакция на таймер. С периодичностью, заданной таймером,
// проверяем, не пришли ли сообщения, и если пришли,
// получаем их.
procedure TChatForm.TimerChatTimer(Sender: TObject);
var
  // Множество сокетов для функции select.
  // Будет содержать только один сокет FSocket.
  SocketSet: TFDSet;
  // Тайм-аут для функции select
  Timeout: TTimeVal;
  // Буфер для получения сообщения.
  // Размер равен максимальному размеру UDP-дейтаграммы
  Buffer: array[0..65506] of Byte;
  Msg: string;
  // Адрес, с которого пришло сообщение
  RecvAddr: TSockAddr;
  RecvLen, AddrLen: Integer;
```

```
begin
  // Инициализируем множество сокетов,
  // т. е. очищаем его от случайного мусора
  FD_ZERO(SocketSet);
  // Добавляем в это множество сокет FSocket
  FD_SET(FSocket, SocketSet);
  // Устанавливаем тайм-аут равным нулю, чтобы
  // функция select ничего не ждала, а возвращала
  // готовность сокетов на момент вызова.
  Timeout.tv_sec := 0;
  Timeout.tv_usec := 0;
  // Проверяем готовность сокета для чтения
  if select(0, @SocketSet, nil, nil, @Timeout) = SOCKET_ERROR then
  begin
    AddMessageToLog('Ошибка при проверке готовности сокета: ' +
      GetErrorString);
    Exit;
  end;
  // Проверяем, оставила ли функция select сокет в множестве.
  // Если оставила, значит, во входном буфере сокета есть данные.
  if FD_ISSET(FSocket, SocketSet) then
  begin
    AddrLen := SizeOf(RecvAddr);
    // Получаем дейтаграмму
    RecvLen := recvfrom(FSocket, Buffer, SizeOf(Buffer), 0,
      RecvAddr, AddrLen);
    // Так как UDP не поддерживает соединение, ошибку при вызове recvfrom
    // мы можем получить, только если случилось что-то совсем
    // экстраординарное.
    if RecvLen < 0 then
    begin
      AddMessageToLog('Ошибка при получении сообщения: ' +
        GetErrorString);
      Exit;
    end;
    // Устанавливаем нужный размер строки
    SetLength(Msg, RecvLen);
    // и копируем в нее дейтаграмму из буфера
    if RecvLen > 0 then
      Move(Buffer, Msg[1], RecvLen);
    AddMessageToLog('Сообщение с адреса ' + inet_ntoa(RecvAddr.sin_addr)
      + ':' + IntToStr(ntohs(RecvAddr.sin_port)) + ': ' + Msg);
  end;
end;
```

Обратите внимание, что в обработчике события от таймера читается только одно сообщение, хотя за время, прошедшее с предыдущего вызова этого обработчика, в принципе, могло прийти несколько сообщений. Если запустить два экземпляра чата на одном компьютере, и с одного из них послать несколько сообщений подряд другому (добиться этого можно, несколько раз быстро нажав на кнопку **Отправить**), то адресат получит сообщения последовательно, с полусекундной задержкой между ними. Было бы достаточно просто организовать в обработчике сообщения таймера цикл до тех пор, пока функция `select` не покажет, что сокет не готов к чтению, и извлечь за один раз сразу все сообщения, которые накопились в буфере сокета. Этого не сделано, чтобы уменьшить уязвимость чата по отношению к действиям потенциального злоумышленника. Имеется в виду та разновидность DoS-атаки, когда злоумышленник посылает большой поток сообщений, чтобы парализовать работу чата. Работа в этом случае, конечно же, будет парализована независимо от того, будет ли в обработчике события таймера извлекаться одно сообщение или все сразу — все равно чат будет замусорен бессмысленными сообщениями. Но в первом случае между показом сообщений будут интервалы, и пользователь хотя бы сможет корректно закрыть программу. Во втором же случае, если злоумышленник посылает сообщения достаточно быстро, цикл может оказаться бесконечным, обработка других оконных сообщений прекратится, и пользователь вынужден будет снять задачу средствами системы. Таким образом, извлечение только одного сообщения за один раз снижает ущерб от атаки. (Разумеется, вряд ли кто-то всерьез захочет атаковать наш учебный пример, но эту возможность следует учитывать при разработке более серьезных приложений.)

Перейдем к следующему примеру использования `select` — TCP-серверу, который может работать одновременно с неограниченным числом клиентов (пример находится на компакт-диске в папке SelectServer). Этот сервер будет усовершенствованной версией нашего простейшего сервера *(см. разд. 2.1.12)* и тоже будет консольным приложением (функция `select`, как мы видели на примере UDP-чата, позволяет создавать приложения с графическим интерфейсом пользователя, так что реализация сервера в качестве консольного приложения — это не необходимость, а свободный выбор для иллюстрации различных способов применения функции `select`).

Примечание

Разумеется, ни один сервер не может работать с неограниченным числом клиентов. Здесь и далее под словом "неограниченный" подразумевается то, что количество клиентов сервера ограничивается только ресурсами системы, а не самой реализацией сервера.

Инициализация сокета и установка его в режим прослушивания в новом сервере ничем не отличается от простейшего, изменения начинаются только с

цикла. Теперь цикл только один (вложенные циклы в нем есть, но они выполняют чисто техническую роль). Начинается цикл с того, что с помощью функции select определяется готовность к чтению слушающего сокета. Если слушающий сокет готов к чтению, то в данном случае это означает, что есть клиенты, которые уже подключились к серверу, но еще не были обработаны функцией accept. Если такие клиенты есть, то сервер принимает подключение, причем только одно за одну итерацию цикла. Для каждого подключившегося клиента сервер создает экземпляр записи TConnection, которая описана в листинге 2.25.

Листинг 2.25. Описание типа TConnection

```
// Запись TConnection хранит информацию о подключившемся клиенте.
// Поле ClientAddr содержит строковое представление адреса клиента.
// Поле ClientSocket содержит сокет, созданный функцией accept
// для взаимодействия с данным клиентом.
// Поле Deleted — служебное. Если оно равно False, значит,
// соединение с данным клиентом по каким-то причинам потеряно,
// и сервер должен освободить ресурсы, выделенные для этого клиента.
PConnection = ^TConnection;
TConnection = record
  ClientAddr: string;
  ClientSocket: TSocket;
  Deleted: Boolean;
end;
```

Поле ClientAddr хранит строковое представление адреса клиента в виде "X.X.X.X:Port" — это поле используется только при выводе сообщений, связанных с данным клиентом. Поле ClientSocket содержит сокет, созданный для связи с данным клиентом. Поле Deleted необходимо для того, чтобы упростить удаление записей для тех клиентов, соединение с которыми уже потеряно. Список соединений хранится в глобальной переменной FConnections типа TList. Потеря соединения обнаруживается при попытке чтения или отправки данных через сокет. Если в одном цикле делать и попытки чтения, и удаление ненужных записей, этот цикл усложняется, и в нем легко сделать ошибку в индексах. Чтобы избежать этого, в "читающем" цикле те записи, для которых потеряно соединение, просто помечаются как удаленные с помощью свойства Deleted. Затем другой цикл удаляет все записи, помеченные для удаления.

После проверки новых подключений начинается проверка получения сообщений от тех клиентов, которые уже подключены. Для этого перебираются сокеты из списка подключений и для каждого вызывается select. Чтобы по-

высить производительность, сокеты проверяются не по одному, а группами. Как уже было сказано, множество типа TFDSet может содержать не более чем FD_SETSIZE сокетов, а в нашем списке их может оказаться больше. Приходится разбивать сокеты на группы размером по FD_SETSIZE и для каждой группы вызывать select отдельно.

Для тех сокетов, которые готовы к чтению, вызывается процедура ProcessSocketMessage. Ее код практически полностью совпадает с кодом одной итерации внутреннего цикла примера SimplestServer (см. листинг 2.15), т. е. процедура сначала читает размер строки, затем — саму строку, после этого формирует ответ и отправляет его клиенту. Реализуя эту функцию таким образом, мы пошли на некоторый риск блокировки, потому что функция select информирует только о том, что во входном буфере сокета есть хоть что-то, но вовсе не гарантирует, что там лежит уже все сообщение целиком. Наша же функция реализована таким образом, что она завершается либо после прочтения сообщения целиком, либо после обнаружения ошибки. Тем не менее в простых случаях можно пойти на такой риск, потому что, во-первых, короткие сообщения редко разбиваются на части, а во-вторых, если даже такое произойдет, оставшаяся часть сообщения, скорее всего, догонит первую достаточно быстро, и блокировка долгой не будет, так что риск при нормальной работе сети и клиента не очень велик.

Примечание

Эта ситуация отличается от использования select для UDP-сокетов. С ними такой проблемы не возникает, т. к. дейтаграмма никогда не приходит по частям, и если функция select показала готовность сокета, значит, уже получено все сообщение целиком.

Завершается основной цикл сервера удалением всех ресурсов, связанных с закрытыми соединениями. После небольшой паузы, сделанной для того, чтобы сервер не нагружал процессор непрерывно, управление передается на начало цикла (листинг 2.26).

Листинг 2.26. Основная часть сервера SelectServer

```
// Тайм-аут для функции select, хотя и передается через указатель,
// является для нее входным параметром, который не изменяется.
// Так как у нас везде будет использоваться один и тот же нулевой
// тайм-аут, можем один раз задать значение переменной Timeout
// и в дальнейшем всегда им пользоваться.
Timeout.tv_sec := 0;
Timeout.tv_usec := 0;
// Начало цикла подключения и общения с клиентами
repeat
```

```
// Сначала проверяем, готов ли слушающий сокет.
// Если он готов, это означает, что есть подключившийся,
// но не обработанный функцией accept клиент
FD_ZERO(SockSet);
FD_SET(MainSocket, SockSet);
if select(0, @SockSet, nil, nil, @Timeout) = SOCKET_ERROR then
  raise ESocketException.Create(
    'Ошибка при проверке готовности слушающего сокета: ' +
    GetErrorString);
// Если функция select оставила MainSocket в множестве, значит,
// зафиксировано подключение клиента, и функция accept не приведет
// к блокированию нити.
if FD_ISSET(MainSocket, SockSet) then
begin
  ClientSockAddrLen := SizeOf(ClientSockAddr);
  // Принимаем подключившегося клиента. Для общения с ним создается
  // новый сокет, дескриптор которого помещается в ClientSocket.
  ClientSocket := accept(MainSocket, @ClientSockAddr,
    @ClientSockAddrLen);
  if ClientSocket = INVALID_SOCKET then
    raise ESocketException.Create(
      'Ошибка при ожидании подключения клиента: ' + GetErrorString);
  // Создаем в динамической памяти новый экземпляр TConnection
  // и заполняем его данными, соответствующими подключившемуся клиенту
  New(NewConnection);
  NewConnection.ClientSocket := ClientSocket;
  NewConnection.ClientAddr := Format('%u.%u.%u.%u:%u', [
    Ord(ClientSockAddr.sin_addr.S_un_b.s_b1),
    Ord(ClientSockAddr.sin_addr.S_un_b.s_b2),
    Ord(ClientSockAddr.sin_addr.S_un_b.s_b3),
    Ord(ClientSockAddr.sin_addr.S_un_b.s_b4),
    ntohs(ClientSockAddr.sin_port)]);
  NewConnection.Deleted := False;
  // Добавляем соединение в список
  Connections.Add(NewConnection);
  WriteLn(OemString('Зафиксировано подключение с адреса ' +
    NewConnection.ClientAddr));
end;
// Теперь проверяем готовность всех сокетов подключившихся клиентов.
// Так как множество SockSet не может содержать более чем FD_SETSIZE
// элементов, а размер списка Connections мы нигде не ограничиваем,
// приходится разбивать Connections на "куски" размером не более
// FD_SETSIZE и обрабатывать этот список по частям.
```

```
  //  Поэтому у нас появляется два цикла — внешний, который повторяется
  // столько раз, сколько у нас будет кусков, и внутренний, который
  // повторяется столько раз, сколько элементов в одном куске.
  for J := 0 to Ceil(Connections.Count, FD_SETSIZE) — 1 do
  begin
    FD_ZERO(SockSet);
    for I := FD_SETSIZE * J to Min(FD_SETSIZE * (J + 1) — 1,
      Connections.Count — 1) do
      FD_SET(PConnection(Connections[I])^.ClientSocket, SockSet);
    if select(0, @SockSet, nil, nil, @Timeout) = SOCKET_ERROR then
      raise ESocketException.Create(
        'Ошибка при проверке готовности сокетов: ' + GetErrorString);
    // Проверяем, какие сокеты функция select оставила в множестве,
    // и вызываем для них ProcessSocketMessage. В этом есть некоторый
    // риск, т. к. для того, чтобы select оставила сокет в множестве,
    // достаточно, чтобы он получил хотя бы один байт от клиента,
    // а не все сообщение. Поэтому может возникнуть такая ситуация,
    // когда сервер получил только часть сообщения, но уже пытается
    // прочитать сообщение целиком. Это приведет к блокированию нити,
    // но вероятность блокирования на долгое время мы оцениваем как
    // крайне низкую, т. к. оставшаяся часть сообщения, скорее всего,
    // придет достаточно быстро, и поэтому идем на такой риск.
    for I := FD_SETSIZE * J to Min(FD_SETSIZE * (J + 1) — 1,
      Connections.Count — 1) do
      if FD_ISSET(PConnection(Connections[I])^.ClientSocket, SockSet)
        then
        ProcessSocketMessage(PConnection(Connections[I])^);
  end;
  // Проверяем поле Deleted у всех соединений. Те, у которых
  // оно равно True, закрываем: закрываем сокет, освобождаем память,
  // удаляем указатель из списка. Цикл идет с конца списка к началу,
  // потому что в ходе работы цикла верхняя граница списка
  // может меняться, и цикл for снизу вверх мог бы привести
  // к появлению индексов вне диапазона.
  for I := Connections.Count — 1 downto 0 do
    if PConnection(Connections[I])^.Deleted then
    begin
      closesocket(PConnection(Connections[I])^.ClientSocket);
      Dispose(PConnection(Connections[I]));
      Connections.Delete(I);
    end;
  Sleep(100);
until False;
```

Функции Ceil и Min, которые встречаются здесь, можно было бы заменить одноименными функциями из модуля Math. Но этот модуль входит не во все варианты поставки Delphi, и чтобы пример можно было откомпилировать в любом варианте поставки Delphi, мы описали их здесь самостоятельно (листинг 2.27).

Листинг 2.27. Функции Ceil и Math

```
// Функция Ceil возвращает наименьшее целое число X, удовлетворяющее
// неравенству X >= A / B
function Ceil(A, B: Integer): Integer;
begin
  Result := A div B;
  if A mod B <> 0 then
    Inc(Result);
end;

// Функция Min возвращает меньшее из двух чисел
function Min(A, B: Integer): Integer;
begin
  if A < B then
    Result := A
  else
    Result := B;
end;
```

Получившийся сервер более устойчив к DoS-атакам, чем написанный ранее многонитевой сервер. Так как он обходится одной нитью, планировщик задач не перегружается при большом числе подключившихся клиентов. DoS-атака заставляет расходовать только ресурсы библиотеки сокетов и процессорное время, причем вредный эффект последнего легко уменьшить, установив процессу сервера низкий приоритет.

Однако сервер имеет другую уязвимость, связанную с возможным отступлением от протокола обмена клиентом (случайным или злонамеренным). Если клиент, например, пришлет всего один байт и на этом остановится, не разрывая связь с сервером, то при попытке получить сообщение от такого клиента сервер окажется заблокированным, т. к. будет ожидать как минимум четырех байтов (длина строки). Это полностью парализует работу сервера, потому что его единственная нить окажется заблокированной, и обрабатывать сообщения от других клиентов он не сможет.

Примечание

Многонитевой сервер в этом отношении надежнее: некорректное сообщение от клиента заблокирует только ту нить, которая взаимодействует с этим клиентом, никак не влияя на остальные нити, работающие с другими клиентами.

Сделать сервер более устойчивым к некорректным действиям клиента можно, если каждый раз читать ровно столько байтов, сколько пришло. Это усложнит сервер, т. к. придется между "сеансами связи с клиентом" помнить, сколько байтов было прочитано в прошлый раз. Однако это поможет полностью избежать блокировок при операциях чтения, что существенно повысит надежность сервера. В этом разделе мы не будем рассматривать соответствующий пример, а реализуем эту возможность в следующем сервере, использующем неблокирующие сокеты. В сервере на основе `select` это делается совершенно аналогично.

2.1.15. Неблокирующий режим

Ранее мы столкнулись с функциями, которые могут надолго приостановить работу вызвавшей их нити, если действие не может быть выполнено немедленно. Это функции `accept`, `recv`, `recvfrom`, `send`, `sendto` и `connect` (в дальнейшем в этом разделе мы не будем упоминать функции `recvfrom` и `sendto`, потому что они в смысле блокирования эквивалентны функциям `recv` и `send` соответственно, и все, что будет здесь сказано о `recv` и `send`, применимо к `recvfrom` и `sendto`). Такое поведение не всегда удобно вызывающей программе, поэтому в библиотеке сокетов предусмотрен особый режим работы сокетов — неблокирующий. Этот режим может быть установлен или отменен для каждого сокета индивидуально с помощью функции `ioctlsocket`, имеющей следующий прототип:

```
function ioctlsocket(s: TSocket; cmd: DWORD;
  var arg: u_long): Integer;
```

Данная функция предназначена для выполнения нескольких логически мало связанных между собой действий. Возможно, у разработчиков первых версий библиотеки сокетов были причины экономить на количестве функций, потому что мы и дальше увидим, что иногда непохожие операции выполняются одной функцией. Но вернемся к `ioctlsocket`. Ее параметр `cmd` определяет действие, которое выполняет функция, а также смысл параметра `arg`. Допустимы три значения параметра `cmd`: `SIOCATMARK`, `FIONREAD` и `FIONBIO`. При задании `SIOCATMARK` параметр `arg` рассматривается как выходной: в нем возвращается ноль, если во входном буфере сокета имеются высокоприоритетные данные, и ненулевое значение, если таких данных нет (как уже было оговорено, мы в этой книге не будем касаться передачи высокоприоритетных данных).

При `cmd`, равном `FIONREAD`, в параметре `arg` возвращается размер данных, находящихся во входном буфере сокета, в байтах. При использовании TCP это число равно максимальному количеству информации, которое можно получить на данный момент за один вызов `recv`. Для UDP это значение равно суммарному размеру всех находящихся в буфере дейтаграмм (напомним, что прочитать несколько дейтаграмм за один вызов `recv` нельзя). Функция `ioctlsocket` с параметром `FIONREAD` может использоваться для проверки наличия данных с целью избежать вызова `recv` тогда, когда это может привести к блокированию, или для организации вызова `recv` в цикле до тех пор, пока из буфера не будет извлечена вся информация.

При задании аргумента `FIONBIO` параметр `arg` рассматривается как входной. Если его значение равно нулю, сокет будет переведен в блокирующий режим, если не равно нулю — в неблокирующий. Таким образом, чтобы перевести некоторый сокет `S` в неблокирующий режим, нужно выполнить следующие действия (листинг 2.28).

Листинг 2.28. Перевод сокета в неблокирующий режим

```
var
  S: TSocket;
  Arg: u_long;
begin
  ...
  Arg := 1;
  ioctlsocket(S, FIONBIO, Arg);
```

Пока программа использует только стандартные сокеты (а не сокеты Windows), сокет может быть переведен в неблокирующий или обратно в блокирующий режим в любой момент. Неблокирующим может быть сделан любой сокет (серверный или клиентский) независимо от протокола.

Функция `ioctlsocket` возвращает нулевое значение в случае успеха и ненулевое — при ошибке. В примере, как всегда, проверка результата для краткости опущена.

Итак, по умолчанию сокет работает в блокирующем режиме. С особенностями работы функций `accept`, `connect`, `recv` и `send` в этом режиме мы уже познакомились. Теперь рассмотрим то, как они ведут себя в неблокирующем режиме. Для этого сначала вспомним, когда эти функции блокируют вызвавшую их нить.

- `accept` — блокирует нить, если на момент ее вызова очередь подключений пуста.

- connect — в случае TCP блокирует сокет практически всегда, потому что требуется время на установление связи с удаленным сокетом. Без блокирования вызов connect выполняется только в том случае, если какая-либо ошибка не дает возможности приступить к операции установления связи. Также без блокирования функция connect выполняется при использовании UDP, потому что в данном случае она только устанавливает фильтр для адресов.
- recv — блокирует нить, если на момент вызова входной буфер сокета пуст.
- send — блокирует нить, если в выходном буфере сокета недостаточно места, чтобы скопировать туда переданную информацию.

Если условия, при которых эти функции выполняются без блокирования, выполнены, то их поведение в блокирующем и неблокирующем режимах идентично. Если же выполнение операции без блокирования невозможно, функции возвращают результат, указывающий на ошибку. Чтобы понять, произошла ли ошибка из-за необходимости блокирования или из-за чего-либо еще, программа должна вызвать функцию WSAGetLastError. Если она вернет WSAEWOULDBLOCK, значит, никакой ошибки не было, но выполнение операции без блокирования невозможно. Закрывать сокет и создавать новый после WSAEWOULDBLOCK, разумеется, не нужно, т. к. ошибки не было, и связь (в случае TCP) осталась неразорванной.

Следует отметить, что при нулевом выходном буфере сокета (т. е. когда функция send передает данные напрямую в сеть) и большом объеме информации функция send может выполняться достаточно долго, т. к. эти данные отправляются по частям, и на каждую часть в рамках протокола TCP получаются подтверждения. Но эта задержка не считается блокированием, и в данном случае send будет одинаково вести себя с блокирующими и неблокирующими сокетами, т. е. вернет управление программе лишь после того, как все данные окажутся в сети.

Для функций accept, recv и send WSAEWOULDBLOCK означает, что операцию следует повторить через некоторое время, и, может быть, в следующий раз она не потребует блокирования и будет выполнена. Функция connect в этом случае начинает фоновую работу по установлению соединения. О завершении этой работы можно судить по готовности сокета, которая проверяется с помощью функции select. Листинг 2.29 иллюстрирует это.

Листинг 2.29. Установление связи при использовании неблокирующего сокета

```
var
  S: TScoket;
```

```
  Block: u_long;
  SetW, SetE: TFDSet;
begin
  S := socket(AF_INET, SOCK_STREAM, 0);
  ...
  Block := 1;
  ioctlsocket(S, FIONBIO, Block);
  connect(S, ...);
  if WSAGetLastError <> WSAEWOULDBLOCK then
  begin
    // Произошла ошибка
    raise ...
  end;
  FD_ZERO(SetW);
  FD_SET(S, SetW);
  FD_ZERO(SetE);
  FD_SET(S, SetE);
  select(0, nil, @SetW, @SetE, nil);
  if FD_ISSET(S, SetW) then
    // Connect выполнен успешно
  else if FD_ISSET(S, SetE) then
    // Соединиться не удалось
  else
    // Произошла еще какая-то ошибка
```

Напомним, что сокет, входящий в множество SetW, будет считаться готовым, если он соединен, а в его выходном буфере есть место. Сокет, входящий в множество SetE, будет считаться готовым, если попытка соединения не удалась. До тех пор, пока попытка соединения не завершилась (успехом или неудачей), ни одно из этих условий готовности не будет выполнено. Таким образом, в данном случае select завершит работу только после того, как будет выполнена попытка соединения, и о результатах этой попытки можно будет судить по тому, в какое из множеств входит сокет.

Из приведенного примера не видно, какие преимущества дает неблокирующий сокет по сравнению с блокирующим. Казалось бы, проще вызвать connect в блокирующем режиме, дождаться результата и лишь потом переводить сокет в неблокирующий режим. Во многих случаях это действительно может оказаться удобнее. Преимущества соединения в неблокирующем режиме связаны с тем, что между вызовами connect и select программа может выполнить какую-либо полезную работу, а в случае блокирующего сокета программа будет вынуждена сначала дождаться завершения работы функции connect и лишь потом сделать что-то еще.

Функция send для неблокирующего сокета также имеет некоторые специфические черты поведения. Они проявляются, когда свободное место в выходном буфере есть, но его недостаточно для хранения данных, которые программа пытается отправить с помощью этой функции. В этом случае функция send, согласно документации, может скопировать в выходной буфер такой объем данных, для которого хватает места. При этом она вернет значение, равное этому объему (оно будет меньше, чем значение параметра len, заданного программой). Оставшиеся данные программа должна отправить позже, вызвав еще раз функцию send. Такое поведение функции send возможно только при использовании TCP. В случае UDP дейтаграмма никогда не разделяется на части, и если в выходном буфере не хватает места для всей дейтаграммы, то функция send возвращает ошибку, а WSAGetLastError — WSAEWOULDBLOCK.

Сразу отметим, что, хотя спецификация допускает частичное копирование функцией send данных в буфер сокета, на практике такое поведение наблюдать пока не удалось: все эксперименты показали, что функция send всегда либо копирует данные целиком, расширяя при необходимости буфер, либо дает ошибку WSAEWOULDBLOCK. Далее этот вопрос будет обсуждаться подробнее. Тем не менее при написании программ следует учитывать возможность частичного копирования, т. к. оно может появиться в тех условиях или в тех реализациях библиотеки сокетов, которые в наших экспериментах не были проверены.

2.1.16. Сервер на неблокирующих сокетах

В этом разделе мы создадим сервер, основанный на неблокирующих сокетах. Это будет наш первый сервер, не использующий функцию ReadFromSocket (см. листинг 2.13). Этот сервер (пример NonBlockingServer на компакт-диске) состоит из одной нити, которая никогда не будет блокироваться сокетными операциями, т. к. все сокеты используют неблокирующий режим. На форме находится таймер, по сигналам которого сервер выполняет попытки чтения данных с сокетов всех подключившихся клиентов. Если данных нет, функция recv немедленно завершается с ошибкой WSAEWOULDBLOCK, и сервер переходит к попытке чтения из следующего сокета.

Запуск сервера (листинг 2.30) мало чем отличается от запуска многонитевого сервера (см. листинг 2.19). Практически вся разница заключается в том, что вместо запуска "слушающей" нити сокет переводится в неблокирующий режим и включается таймер.

Листинг 2.30. Инициализация сервера на неблокирующих сокетах

```
// Реакция на кнопку "Запустить" — запуск сервера
procedure TServerForm.BtnStartServerClick(Sender: TObject);
```

```
var
  // Адрес, к которому привязывается слушающий сокет
  ServerAddr: TSockAddr;
  NonBlockingArg: u_long;
begin
  // Формируем адрес для привязки.
  FillChar(ServerAddr.sin_zero, SizeOf(ServerAddr.sin_zero), 0);
  ServerAddr.sin_family := AF_INET;
  ServerAddr.sin_addr.S_addr := INADDR_ANY;
  try
    ServerAddr.sin_port := htons(StrToInt(EditPortNumber.Text));
    if ServerAddr.sin_port = 0 then
    begin
      MessageDlg('Номер порта должен находиться в диапазоне 1-65535',
        mtError, [mbOK], 0);
      Exit;
    end;
    // Создание сокета
    FServerSocket := socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if FServerSocket = INVALID_SOCKET then
    begin
      MessageDlg('Ошибка при создании сокета:'#13#10 + GetErrorString,
        mtError, [mbOK], 0);
      Exit;
    end;
    // Привязка сокета к адресу
    if bind(FServerSocket, ServerAddr, SizeOf(ServerAddr)) =
      SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при привязке сокета к адресу:'#13#10 +
        GetErrorString, mtError, [mbOK], 0);
      closesocket(FServerSocket);
      Exit;
    end;
    // Перевод сокета в режим прослушивания
    if listen(FServerSocket, SOMAXCONN) = SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при переводе сокета в режим прослушивания:' +
        #13#10 + GetErrorString, mtError, [mbOK], 0);
      closesocket(FServerSocket);
      Exit;
    end;
    // Перевод сокета в неблокирующий режим
    NonBlockingArg := 1;
```

```
    if ioctlsocket(FServerSocket, FIONBIO, NonBlockingArg) =
      SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при переводе сокета в неблокирующий режим:' +
        #13#10 + GetErrorString, mtError, [mbOK], 0);
      closesocket(FServerSocket);
      Exit;
    end;
    // Перевод элементов управления в состояние "Сервер работает"
    LabelPortNumber.Enabled := False;
    EditPortNumber.Enabled := False;
    BtnStartServer.Enabled := False;
    TimerRead.Interval := TimerInterval;
    LabelServerState.Caption := 'Сервер работает';
  except
    on EConvertError do
      // Это исключение может возникнуть только в одном месте -
      // при вызове StrToInt(EditPortNumber.Text)
      MessageDlg('"' + EditPortNumber.Text +
        '" не является целым числом', mtError, [mbOK], 0);
    on ERangeError do
      // Это исключение может возникнуть только в одном месте -
      // при присваивании значения номеру порта
      MessageDlg('Номер порта должен находиться в диапазоне 1-65535',
        mtError, [mbOK], 0);
  end;
end;
```

Так как протокол TCP допускает разбиение посылки на произвольное число пакетов, возможна ситуация, когда на момент срабатывания таймера в буфере сокета будет только часть того, что отправил клиент. Так как мы договорились не блокировать нить, то ждать, пока придет остальное, мы не будем. Вместо этого будем запоминать то, что пришло, а при следующем срабатывании таймера, если пришло еще что-то, добавлять это к предыдущим данным, и так до тех пор, пока не придет все, что мы ожидаем получить от клиента. Так как посылка может разорваться в любом месте, наш код должен быть к этому готов.

Взаимодействие сервера с клиентом состоит из трех этапов. На первом этапе сервер получает от клиента четырехбайтное значение — длину строки. На втором этапе сервер получает от клиента саму строку, размер которой уже известен из величины, полученной на первом этапе. На третьем этапе сервер отправляет ответ клиенту, состоящий из строки, завершающейся нулем. Чтобы при очередном "тике" таймера сервер мог продолжить общение с клиен-

том, прерванное в произвольном месте, необходимо запоминать, на каком этапе было прервано взаимодействие в предыдущий раз, сколько байтов на данном этапе уже прочитано или отправлено и сколько еще осталось прочитать или отправить. Для хранения этих данных мы будем использовать типы `TTransportPhase` и `TConnection` (листинг 2.31).

Листинг 2.31. Типы `TTransportPhase` и `TConnection`

```
type
  // Этап взаимодействия с клиентом:
  // tpReceiveLength — сервер ожидает от клиента длину строки
  // tpReceiveString — сервер ожидает от клиента строку
  // tpSendString — сервер посылает клиенту строку
  TTransportPhase = (tpReceiveLength, tpReceiveString, tpSendString);

  // Информация о соединении с клиентом:
  // ClientSocket — сокет, созданный для взаимодействия с клиентом
  // ClientAddr — строковое представление адреса клиента
  // MsgSize — длина строки, получаемая от клиента
  // Msg — строка, получаемая от клиента или отправляемая ему
  // Phase — этап взаимодействия с данным клиентом
  // Offset — количество байтов, уже полученных от клиента
  //   или отправленных ему на данном этапе
  // BytesLeft — сколько байтов осталось получить от клиента
  //   или отправить ему на данном этапе
  PConnection = ^TConnection;
  TConnection = record
                  ClientSocket: TSocket;
                  ClientAddr: string;
                  MsgSize: Integer;
                  Msg: string;
                  Phase: TTransportPhase;
                  Offset: Integer;
                  BytesLeft: Integer;
                end;
```

Для каждого подключившегося клиента создается отдельный экземпляр записи `TConnection`, в котором хранится информация как о самом подключении, так и о том, на каком этапе находится взаимодействие с данным клиентом.

Проверка подключения клиентов и взаимодействие с подключившимися ранее реализуется, как уже было сказано, при обработке события таймера. Код обработчика приведен в листинге 2.32.

Листинг 2.32. Обработчик события таймера

```
// Обработка сообщения от таймера
// В ходе обработки проверяется наличие вновь подключившихся клиентов,
// а также осуществляется обмен данными с клиентами
procedure TServerForm.TimerReadTimer(Sender: TObject);
var
  // Сокет, который создается для вновь подключившегося клиента
  ClientSocket: TSocket;
  // Адрес подключившегося клиента
  ClientAddr: TSockAddr;
  // Длина адреса
  AddrLen: Integer;
  // Вспомогательная переменная для создания нового подключения
  NewConnection: PConnection;
  I: Integer;
begin
  AddrLen := SizeOf(TSockAddr);
  // Проверяем наличие подключений. Так как сокет неблокирующий,
  // accept не будет блокировать нить даже в случае отсутствия
  // подключений.
  ClientSocket := accept(FServerSocket, @ClientAddr, @AddrLen);
  if ClientSocket = INVALID_SOCKET then
  begin
    // Если произошедшая ошибка — WSAEWOULDBLOCK, это просто означает,
    // что на данный момент подключений нет, а вообще все в порядке,
    // поэтому ошибку WSAEWOULDBLOCK мы просто игнорируем. Прочие же
    // ошибки могут произойти только в случае серьезных проблем,
    // которые требуют остановки сервера.
    if WSAGetLastError <> WSAEWOULDBLOCK then
    begin
      MessageDlg('Ошибка при подключении клиента:#13#10' +
        GetErrorString + #13#10'Сервер будет остановлен',
        mtError, [mbOK], 0);
      ClearConnections;
      closesocket(FServerSocket);
      OnStopServer;
    end;
  end
  else
  begin
    // Создаем запись для нового подключения и заполняем ее
    New(NewConnection);
```

```
    NewConnection.ClientSocket := ClientSocket;
    NewConnection.ClientAddr := Format('%u.%u.%u.%u:%u', [
          Ord(ClientAddr.sin_addr.S_un_b.s_b1),
          Ord(ClientAddr.sin_addr.S_un_b.s_b2),
          Ord(ClientAddr.sin_addr.S_un_b.s_b3),
          Ord(ClientAddr.sin_addr.S_un_b.s_b4),
          ntohs(ClientAddr.sin_port)]);
    NewConnection.Phase := tpReceiveLength;
    NewConnection.Offset := 0;
    NewConnection.BytesLeft := SizeOf(Integer);
    // Добавляем запись нового соединения в список
    FConnections.Add(NewConnection);
    AddMessageToLog('Зафиксировано подключение с адреса ' +
      NewConnection.ClientAddr);
  end;
  // Обрабатываем все существующие подключения.
  // Цикл идет от конца списка к началу потому, что в ходе
  // обработки соединение может быть удалено из списка.
  for I := FConnections.Count – 1 downto 0 do
    ProcessConnection(I);
end;
```

Обратите внимание, что сокет, созданный функцией accept, нигде не переводится в неблокирующий режим. Это связано с тем, что такой сокет наследует свойства слушающего сокета, поэтому он в данном случае сразу создается неблокирующим.

Собственно взаимодействие сервера с клиентом вынесено в метод ProcessConnection (листинг 2.33), который осуществляет чтение данных от клиента и отправку данных в соответствии с этапом, на котором остановилось взаимодействие. При реализации этого метода необходимо просто аккуратно следить за тем, куда и сколько данных нужно передать.

Листинг 2.33. Метод ProcessConnection

```
// Обработка клиента. Index задает индекс записи в списке
procedure TServerForm.ProcessConnection(Index: Integer);
var
  // Вспомогательная переменная, чтобы не приводить каждый раз
  // FConnections[Index] к PConnection
  Connection: PConnection;
  // Результат вызова recv и send
  Res: Integer;
```

```
  // Вспомогательная процедура, освобождающая ресурсы, связанные
  // с клиентом и удаляющая запись подключения из списка
  procedure RemoveConnection;
  begin
    closesocket(Connection.ClientSocket);
    Dispose(Connection);
    FConnections.Delete(Index);
  end;

begin
  Connection := PConnection(FConnections[Index]);
  // Проверяем, на каком этапе находится взаимодействие с клиентом.
  // Используется оператор if, а не case, потому, что в случае case
  // выполняется только одна альтернатива, а в нашем случае в ходе
  // выполнения этапа он может завершиться, и взаимодействие
  // перейдет к следующему этапу. Использование if позволяет выполнить
  // все три этапа, если это возможно, а не один из них.
  if Connection.Phase = tpReceiveLength then
  begin
    // Этап получения от клиента длины строки. При выполнении этого
    // этапа сервер получает от клиента длину строки и размещает ее
    // в поле Connection.MsgSize. Здесь приходится учитывать, что
    // теоретически даже такая маленькая (4 байта) посылка может
    // быть разбита на несколько пакетов, поэтому за один раз этот
    // этап не будет завершен, и второй раз его придется продолжать,
    // загружая оставшиеся байты. Connection.Offset — количество
    // уже прочитанных на данном этапе байтов — одновременно является
    // смещением, начиная с которого заполняется буфер.
    Res := recv(Connection.ClientSocket, (PChar(@Connection.MsgSize) +
      Connection.Offset)^, Connection.BytesLeft, 0);
    if Res > 0 then
    begin
      // Если Res > 0, это означает, что получено Res байтов.
      // Соответственно, увеличиваем на Res количество прочитанных
      // на данном этапе байтов и на такую же величину уменьшаем
      // количество оставшихся.
      Inc(Connection.Offset, Res);
      Dec(Connection.BytesLeft, Res);
      // Если количество оставшихся байтов равно нулю, можно переходить
      // к следующему этапу.
      if Connection.BytesLeft = 0 then
      begin
        // Проверяем корректность принятой длины строки
        if Connection.MsgSize <= 0 then
```

```
      begin
        AddMessageToLog('Неверная длина строки от клиента ' +
          Connection.ClientAddr +': ' + IntToStr(Connection.MsgSize));
        RemoveConnection;
        Exit;
      end;
      // Следующий этап — это чтение самой строки
      Connection.Phase := tpReceiveString;
      // Пока на этом этапе не прочитано ни одного байта
      Connection.Offset := 0;
      // Осталось прочитать Connection.MsgSize байтов
      Connection.BytesLeft := Connection.MsgSize;
      // Сразу выделяем память под строку
      SetLength(Connection.Msg, Connection.MsgSize);
    end;
  end
  else if Res = 0 then
  begin
    AddMessageToLog('Клиент ' + Connection.ClientAddr +
      ' закрыл соединение');
    RemoveConnection;
    Exit;
  end
  else
    // Ошибку WSAEWOULDBLOCK игнорируем, т. к. она говорит
    // только о том, что входной буфер сокета пуст, но в целом
    // все в порядке
    if WSAGetLastError <> WSAEWOULDBLOCK then
    begin
      AddMessageToLog('Ошибка при получении данных от клиента ' +
        Connection.ClientAddr + ': ' + GetErrorString);
      RemoveConnection;
      Exit;
    end;
end;
if Connection.Phase = tpReceiveString then
begin
  // Следующий этап — чтение строки. Он практически не отличается
  // по реализации от этапа чтения длины строки, за исключением
  // того, что теперь буфером, куда помещаются полученные от клиента
  // данные, служит не Connection.MsgSize, а Connection.Msg.
  Res := recv(Connection.ClientSocket,
    Connection.Msg[Connection.Offset + 1], Connection.BytesLeft, 0);
```

```
if Res > 0 then
begin
  Inc(Connection.Offset, Res);
  Dec(Connection.BytesLeft, Res);
  // Если количество оставшихся байтов равно нулю, можно переходить
  // к следующему этапу.
  if Connection.BytesLeft = 0 then
  begin
    AddMessageToLog('От клиента ' + Connection.ClientAddr +
      ' получена строка: ' + Connection.Msg);
    // Преобразуем строку. В отличие от предыдущих примеров, здесь
    // мы явно добавляем к строке #0. Это связано с тем, что при
    // отправке, которая тоже может быть выполнена не за один раз,
    // мы указываем индекс того символа строки, начиная с которого
    // нужно отправлять данные. И (хотя теоретически вероятность
    // этого очень мала) может возникнуть ситуация, когда за
    // один раз будут отправлены все символы строки, кроме
    // завершающего #0, и тогда при следующей отправке начинать
    // придется с него. Если мы будем использовать тот #0, который
    // добавляется к концу строки автоматически, то в этом случае
    // индекс выйдет за пределы диапазона. Поэтому мы вручную
    // добавляем еще один #0 к строке, чтобы он стал законной
    // ее частью.
    Connection.Msg := AnsiUpperCase(StringReplace(Conection.Msg, #0,
      '#0', [rfReplaceAll])) + ' (Non-blocking server)'#0;
    // Следующий этап — отправка строки клиенту
    Connection.Phase := tpSendString;
    // Отправлено на этом этапе 0 байт
    Connection.Offset := 0;
    // Осталось отправить Length(Connection.Msg) байт.
    // Единицу к длине строки, в отличие от предыдущих примеров,
    // не добавляем, т. к. там эта единица нужна была для того,
    // чтобы учесть добавляемый к строке автоматически символ #0.
    // Здесь мы еще один #0 добавили к строке явно, поэтому
    // он уже учтен в функции Length.
    Connection.BytesLeft := Length(Connection.Msg);
  end;
end
else if Res = 0 then
begin
  AddMessageToLog('Клиент ' + Connection.ClientAddr +
    ' закрыл соединение');
  RemoveConnection;
  Exit;
end
```

```
    else
      // Как обычно, "ошибку" WSAEWOULDBLOCK просто игнорируем
      if WSAGetLastError <> WSAEWOULDBLOCK then
      begin
        AddMessageToLog('Ошибка при получении данных от клиента ' +
          Connection.ClientAddr + ': ' + GetErrorString);
        RemoveConnection;
        Exit;
      end;
  end;
  if Connection.Phase = tpSendString then
  begin
    // Следующий этап — отправка строки. Код примерно такой же,
    // как и в предыдущем этапе, но вместо recv используется send.
    // Кроме того, отсутствует проверка на Res = 0, т. к. при
    // использовании TCP send никогда не возвращает 0.
    Res := send(Connection.ClientSocket,
      Connection.Msg[Connection.Offset + 1], Connection.BytesLeft, 0);
    if Res > 0 then
    begin
      Inc(Connection.Offset, Res);
      Dec(Connection.BytesLeft, Res);
      // Если Connections.BytesLeft = 0, значит, строка отправлена
      // полностью.
      if Connection.BytesLeft = 0 then
      begin
        AddMessageToLog('Клиенту ' + Connection.ClientAddr +
          ' отправлена строка: ' + Connection.Msg);
        // Очищаем строку, просто чтобы сэкономить память
        Connection.Msg := '';
        // Следующий этап — снова получение длины строки от клиента
        Connection.Phase := tpReceiveLength;
        // Получено — 0 байт
        Connection.Offset := 0;
        // Осталось прочитать столько, сколько занимает целое число
        Connection.BytesLeft := SizeOf(Integer);
      end;
    end
    else
      if WSAGetLastError <> WSAEWOULDBLOCK then
      begin
        AddMessageToLog('Ошибка при отправке данных клиенту ' +
          Connection.ClientAddr + ': ' + GetErrorString);
```

```
        RemoveConnection;
        Exit;
      end;
  end;
end;
```

В итоге мы получили сервер, достаточно устойчивый как к подключению множества клиентов, так и к нарушению протокола со стороны клиента.

Для самостоятельной работы рекомендуем подумать о том, как можно сделать UDP-чат на неблокирующих сокетах. На самом деле он мало чем будет отличаться от рассмотренного чата на основе `select`. Просто при использовании `select` проверка возможности неблокирующего чтения из сокета проверяется предварительным вызовом этой функции, а в случае неблокирующих сокетов сначала вызывается `recvfrom`, а потом проверяется, было что-то прочитано, или же операция не может быть выполнена потому, что блокировки запрещены. Во всем остальном использование `select` и неблокирующих сокетов очень похоже, причем не только в данном случае, но и вообще.

2.1.17. Параметры сокета

Каждый сокет обладает рядом параметров (опций), которые влияют на его работу. Существуют параметры уровня сокета, которые относятся к сокету как к объекту безотносительно используемого протокола и его уровня. Впрочем, некоторые параметры уровня сокета применимы не ко всем протоколам.

Здесь мы не будем рассматривать все параметры сокета, а ограничимся лишь изложением методов доступа к ним и познакомимся с некоторыми самыми интересными параметрами.

Для получения текущего значения параметров сокета предусмотрена функция `getsockopt`, для изменения — `setsockopt`. Прототипы этих функций выглядят следующим образом:

```
function getsockopt(s: TSocket; level, optname: Integer;
  optval: PChar; var optlen: Integer): Integer;

function setsockopt(s: TSocket; level, optname: Integer;
  optval: PChar; optlen: Integer): Integer;
```

Параметры у функций почти одинаковы. Первый задает сокет, параметры которого следует узнать или изменить. Второй указывает, параметр какого уровня следует узнать или изменить. Третий задает сам параметр сокета. Параметр `optval` содержит указатель на буфер, в котором хранится значение параметра, а `optlen` — размер этого буфера (разные параметры имеют раз-

личные типы и поэтому размер буфера может быть разным). Функция getsockopt сохраняет значение параметра в буфере, заданном указателем optval. Длина буфера передается через параметр optlen, и через него же возвращается размер, реально понадобившийся для хранения параметра. У функции setsockopt параметр optval содержит указатель на буфер, хранящий новое значение параметра сокета, а optlen — размер этого буфера.

Чаще всего параметры сокета имеют целый или логический тип. В обоих случаях параметр optval должен содержать указатель на значение типа Integer. Для логического типа любое ненулевое значение интерпретируется как True, нулевое — как False.

Два достаточно важных параметра сокета — размеры входного и выходного буфера. Это параметры уровня сокета (SOL_SOCKET), их номера задаются константами SO_RCVBUF и SO_SNDBUF. Например, чтобы получить размер входного буфера сокета, нужно выполнить код листинга 2.34.

Листинг 2.34. Получение размера входного буфера сокета

```
var
  Val, Len: Integer;
  S: TSocket;
begin
  ...
  Len := SizeOf(Integer);
  getsockopt(S, SOL_SOCKET, SO_RCVBUF, @Val, Len);
```

После выполнения этого кода размер буфера будет содержаться в переменной Val.

Немного поэкспериментировав, можно обнаружить, что размер входного и выходного буфера равен 8192 байтам как для TCP, так и для UDP. Тем не менее это не мешает отправлять и получать дейтаграммы большего размера (для UDP), а также накапливать в буфере больший объем информации (для TCP). При получении данных это достигается за счет использования более низкоуровневых буферов, чем буфер самого сокета. Можно даже установить входной буфер сокета равным нулю — тогда все поступившие данные будут храниться в низкоуровневых буферах. Однако делать так не рекомендуется, т. к. при этом снижается производительность.

Как уже говорилось, если буфер для исходящих имеет нулевой размер, то функции send и sendto независимо от режима работы сокета отправляют данные непосредственно в сеть. Если же размер этого буфера не равен нулю, при необходимости он может увеличиваться.

В MSDN описаны следующие правила роста буфера:

1. Если объем данных в буфере меньше, чем это задано параметром SO_SNDBUF, то новые данные копируются в буфер полностью. Буфер при необходимости увеличивается.
2. Если объем данных в буфере достиг или превысил SO_SNDBUF, но в буфере находятся данные, переданные в результате только одного вызова send, последующий вызов приводит к увеличению буфера до размера, необходимого, чтобы принять эти данные целиком.
3. Если объем данных в буфере достиг или превысил SO_SNDBUF, и эти данные оказались в буфере в результате нескольких вызовов send, то буфер не расширяется. Блокирующий сокет при этом ждет, когда за счет отправки данных в буфере появится место, неблокирующий завершает операцию с ошибкой WSAEWOULDBLOCK.

Следует отметить, что увеличение размера буфера носит временный характер.

Заметим также, что в ходе наших экспериментов второе правило воспроизвести не удалось. Если предел, заданный параметром SO_SNDBUF, был достигнут, не удавалось поместить новые данные в буфер независимо от того, были ли имеющиеся данные положены туда одним вызовом send или несколькими. Впрочем, это могут быть детали реализации, которые различны в разных версиях системы.

Ранее мы упоминали, что UDP допускает широковещательную рассылку (рассылку по адресу 255.255.255.255 и т. п.). Но по умолчанию такая рассылка запрещена. Чтобы разрешить широковещательную рассылку, нужно установить в True параметр SO_BROADCAST, относящийся к уровню сокета (SOL_SOCKET). Таким образом, вызов функции setsockopt для разрешения широковещательной рассылки будет выглядеть так, как показано в листинге 2.35.

Листинг 2.35. Включение возможности широковещательной рассылки

```
var
  EnBroad: Integer;
begin
  EnBroad := 1;
  setsockopt(S, SOL_SOCKET, SO_BROADCAST, PChar(@EnBroad),
    SizeOf(Integer));
```

Для запрета широковещательной рассылки через сокет используется тот же код, за исключением того, что переменной EnBroad следует присвоить ноль.

Последний параметр сокета, который мы рассмотрим, называется SO_LINGER. Он управляет поведением функции closesocket. Напомним, что по умолчанию эта функция не блокирует вызвавшую ее нить, а закрывает сокет в фоновом режиме. Параметр SO_LINGER имеет тип TLinger, представляющий собой следующую структуру:

```
TLinger = record
  l_onoff: u_short;
  l_linger: u_short;
end;
```

Поле l_onoff этой структуры показывает, будет ли использоваться фоновый режим закрытия сокета. Нулевое значение показывает, что закрытие выполняется в фоновом режиме, как это установлено по умолчанию (в этом случае поле l_linger игнорируется). Ненулевое значение показывает, что функция closesocket не вернет управление вызвавшей ее нити, пока сокет не будет закрыт. В этом случае возможны два варианта: мягкое и грубое закрытие. Мягкое закрытие предусматривает, что перед закрытием сокета все данные, находящиеся в его выходном буфере, будут переданы партнеру. При грубом закрытии данные партнеру не передаются. Поле l_linger задает время (в секундах), которое дается на передачу данных партнеру. Если за отведенное время данные, находящиеся в выходном буфере сокета, не были отправлены, сокет будет закрыт грубо. Если поле l_linger будет равно нулю (при ненулевом l_onoff), сокет всегда будет закрываться грубо. Неблокирующие сокеты рекомендуется закрывать с нулевым временем ожидания или в фоновом режиме. При мягком закрытии неблокирующего сокета не в фоновом режиме, если остались непереданные данные, вызов closesocket завершится с ошибкой WSAEWOULDBLOCK, и сокет не будет закрыт. Придется вызывать функцию closesocket несколько раз до тех пор, пока она не завершится успешно.

Остальные параметры сокета детально описаны в MSDN.

2.1.18. Итоги первого раздела

Мы рассмотрели основные принципы работы со стандартными сокетами. Хотя многое осталось за кадром, того, что здесь было написано, достаточно, чтобы начать создавать разнообразные приложения с использованием сокетов. Для самостоятельного изучения рекомендуется сделать следующее:

- для каждой из упоминавшихся здесь функций выяснить, какие ошибки может возвращать WSAGetLastError в случае неуспешного завершения и что каждая из этих ошибок означает;

- посмотреть, какие еще параметры (опции) есть у сокета;
- самостоятельно разобраться с не упомянутыми здесь функциями `getsockname`, `gethostbyaddr` и `getaddrbyhost`.

Из приведенных примеров видно, что стандартные сокеты достаточно легко интегрируются с пользовательским интерфейсом, однако приложение, использующее их, вынуждено самостоятельно опрашивать сокеты с определенной периодичностью (например, по таймеру). Это не совпадает с принятой в Windows схемой событийного управления программой, основанной на принципе "пусть мне скажут, когда что-то произойдет, и я отреагирую". Именно поэтому стандартные сокеты были расширены и появились сокеты Windows, с которыми мы познакомимся далее.

2.2. Сокеты Windows

В предыдущих разделах мы рассмотрели те методы работы с сокетами, которые восходят еще к сокетам Беркли. Разработчики библиотеки сокетов для Windows добавили в нее также поддержку новых методов, упрощающих работу с сокетами для приложений, имеющих традиционную для Windows событийно-ориентированную модель. В Windows можно использовать асинхронные сокеты и перекрытый ввод-вывод. Далее мы рассмотрим эти расширения, а также ряд новых функций, пришедших на смену "морально устаревшим" функциям из стандартных сокетов.

Материал здесь, как и ранее, не претендует на полноту, а предназначен лишь для знакомства с наиболее часто употребляемыми возможностями библиотеки сокетов. По-прежнему рассматриваются только протоколы TCP и UDP. Не будут затронуты такие вопросы, как поддержка качества обслуживания, пространства имен, простые сокеты (`RAW_SOCK`) и SPI (Service Provider Interface). Тем, кто захочет самостоятельно разобраться с данными вопросами, рекомендуем книгу [3].

2.2.1. Версии Windows Sockets

При рассмотрении функции `WSAStartup` уже упоминалось, что существуют разные версии библиотеки сокетов, которые заметно различаются по функциональности. К сожалению, полный перечень существующих на сегодняшний день версий Windows Sockets и их особенностей в документации в явном виде не приводится, но, изучая разрозненную информацию, можно сделать некоторые выводы, которые приведены в табл. 2.1.

В дальнейшем, если не оговорено иное, под WinSock 1 мы будем подразумевать версию 1.1, под WinSock 2 — версию 2.2.

Таблица 2.1. *Версии Windows Sockets*

Версия	Комментарий
1.0	Упоминается только вскользь. Видимо, настолько старая версия, что ее поддержка в чистом виде в современных системах отсутствует
1.1	Основная подверсия первой версии библиотеки. По умолчанию входила во все версии Windows до Windows 95 включительно. Ориентирована на 16-разрядные системы с корпоративной многозадачностью
2.0	В чистом виде никуда не ставилась. Ориентирована на 32-разрядные системы с вытесняющей многозадачностью. Исключены некоторые устаревшие функции
2.2	Основная подверсия второй версии библиотеки. По умолчанию входит в состав Windows 98/NT 4/2000, а также, видимо, в более поздних версий. Для Windows 95 существует возможность обновления Windows Sockets до этой версии

WinSock 1 в 16-разрядных версиях Windows реализуется библиотекой WinSock.dll, в 32-разрядных — WSock32.dll. WinSock 2 реализуется библиотекой WS2_32.dll, и, кроме того, часть функций вынесена в отдельную библиотеку MSWSock.dll. При этом для сохранения совместимости WS2_32.dll содержит даже те устаревшие функции, которые формально исключены из спецификации WinSock 2. В тех системах, в которых установлена библиотека WinSock 2, WSock32.dll не реализует самостоятельно практически ни одной функции, а просто импортирует их из WS2_32.dll и MSWSock.dll. WSock32.dll требуется только для обратной совместимости, в новых программах необходимости в этой библиотеке нет.

Как это ни удивительно, но в Delphi даже 2007-й версии (не говоря уже о более ранних) отсутствует поддержка WinSock 2. Стандартный модуль `WinSock` импортирует функции только из WSock32.dll, поэтому программисту доступны только функции WinSock 1. Разумеется, импортировать функции WinSock 2 самостоятельно не составит труда. Более того, в Интернете можно найти уже готовые модули, импортирующие их (например, на сайте Алекса Коншина **http://home.earthlink.net/~akonshin/delphi_ru.htm**). Тем не менее, чтобы избежать разночтений, мы не будем использовать какой-либо готовый модуль для импорта и примем следующее соглашение: если прототип функции приведен только на Паскале, значит, эта функция есть в модуле `WinSock`. Если же прототип приведен и на C/C++, и на Паскале, значит, функция в `WinSock` не описана. В этом случае прототип функции на C/C++ берется из MSDN, а перевод на Паскаль — импровизация автора книги. В некоторых случаях возможны несколько вариантов перевода, поэтому не стоит рассматривать приведенный здесь перевод как истину в последней инстанции. Тем,

кто будет самостоятельно импортировать функции из WS2_32.dll, следует помнить, что они имеют модель вызова `stdcall` (при описании прототипов функций мы для краткости будем опускать эту директиву).

Примечание

С Delphi поставляется библиотека Indy (Internet Direct), в состав которой входит модуль `IdWinSock2`, импортирующий почти все функции WinSock 2 из системных библиотек. Импорт в нем динамический, над каждой функцией сделана обертка, которая при первом вызове проверяет, была ли уже загружена функция из библиотеки, и при необходимости загружает ее. Чтобы реализовать это, имена всех функций изменены, а вызов идет через процедурные переменные, имена которых совпадают с оригинальными именами соответствующих функций.

WinSock 2 предлагает разработчику *Service Provider Interface* (SPI), с помощью которого можно добавлять в систему поддержку своих протоколов. Устаревшими объявлены функции, имеющие привязку к конкретным протоколам (например, уже знакомая нам функция `inet_addr`, которая имеет смысл только при использовании протокола IP). Добавлены новые функции, которые призваны унифицировать операции с разными протоколами. Фактически, если работать с WinSock 2, то программа может быть написана так, что сможет использовать даже те протоколы, которые не существовали на момент ее разработки. Кроме того, добавлена возможность связи асинхронных сокетов с событиями вместо оконных сообщений, а также поддержка перекрытого ввода-вывода (в WinSock 1 он поддерживался только в линии NT и не в полном объеме). Добавлена поддержка *качества обслуживания* (Quality of Service, QoS — резервирование части пропускной способности сети для нужд конкретного соединения), поддержка портов завершения, многоадресной рассылки и регистрации имен. Большинство этих нововведений требуются для пользовательских программ относительно редко (или вообще не нужны), поэтому мы не будем заострять на них внимание. Далее будут рассмотрены асинхронные сокеты (связанные как с сообщениями, так и с событиями), перекрытый ввод-вывод, методы универсализации работы с протоколами и многоадресная рассылка.

2.2.2. Устаревшие функции WinSock 1

В этом разделе мы познакомимся с теми устаревшими функциями, которые не стоит применять в 32-разрядных программах. Рассмотрим мы их, разумеется, очень обзорно, только для того, чтобы после прочтения книги вас не смущали упоминания этих функций и связанных с ними ошибок, которые иногда встречаются в MSDN.

В 16-разрядных версиях Windows реализована так называемая корпоративная многозадачность: каждая программа время от времени должна добровольно

возвращать управление операционной системе, чтобы та могла передать управление другой программе. Если какая-то программа при этом поведет себя некорректно и не вернет управление системе, то все остальные приложения не смогут продолжать работу. Другой недостаток такой модели — в ней невозможно распараллеливание работы в рамках одного процесса, т. е. создание нитей.

При такой модели многозадачности использование блокирующих сокетов может привести к остановке всей системы, если не будут приняты дополнительные меры. В Windows проблема решается следующим образом: библиотека сокетов во время ожидания периодически вызывает заранее указанную функцию. В 16-разрядных версиях Windows эта функция по умолчанию извлекает сообщение из системной очереди и передает его соответствующему приложению. Таким образом, остальные приложения не прекращают работу во время блокирующего вызова.

В очереди могут находиться сообщения и для того приложения, которое выполняет блокирующий вызов. В этом случае будет снова вызвана оконная процедура, инициировавшая блокирующую операцию. Это напоминает рекурсию, при которой процедура вызывает сама себя: в памяти компьютера будут одновременно две активации этой процедуры. Упрощенно это выглядит так: оконная процедура вызывает блокирующую функцию (например, `accept`), а та, в свою очередь, снова вызывает ту же самую оконную процедуру. При этом вторая активация не может выполнять никаких операций с сокетами: они будут завершены с ошибкой `WSAEINPROGRESS`. Эта ошибка не фатальная, она указывает, что в данный момент выполняется блокирующая операция, и программа должна подождать ее завершения и лишь потом пытаться работать с сокетами (т. е. не раньше, чем первая активация оконной процедуры вновь получит управление). Существует специальная функция `WSAIsBlocking`, которая возвращает `True`, если в данный момент выполняется блокирующая операция и работа с сокетами невозможна.

Вторая активация процедуры может прервать блокирующий вызов с помощью функции `WSACancelBlockingCall`. При этом первая активация получит ошибку `WSAECANCELLED`.

Программа может устанавливать свою процедуру, которая будет вызываться во время выполнения блокирующей операции. Для этого предусмотрены функции `WSASetBlockingHook` и `WSAUnhookBlockingHook`.

Данная модель неудобна, поэтому разработчики WinSock 1 рекомендуют модель асинхронных сокетов, более приспособленную к особенностям Windows.

В 32-разрядных версиях WinSock такая модель работы поддерживается в полном объеме, за исключением того, что по умолчанию при блокирующем

вызове не вызывается никакая функция. Поэтому если не использовать WSASetBlockingHook, то в 32-разрядном приложении невозможно получить ситуацию, когда операция с сокетом не будет выполнена из-за того, что в этот момент уже выполняется другая операция, и второй активации оконной процедуры из-за блокирующего вызова тоже не будет создано. Отметим, что разные нити могут одновременно выполнять блокирующие операции с сокетами, и это не приведет к появлению ошибки WSAEINPROGRESS.

Все перечисленные функции формально исключены из спецификации WinSock 2, хотя фактически они присутствуют в библиотеке WS2_32.dll и при необходимости могут быть задействованы (это, правда, осложняется тем, что в новых версиях MSDN отсутствует их описание). Тем не менее причин ориентироваться на эту неудобную модель в 32-разрядных версиях Windows, видимо, нет. Описание этих функций мы здесь привели только для того, чтобы упоминания об ошибках WSAEINPROGRESS и WSAECANCELLED, которые иногда встречаются в MSDN, не смущали вас.

2.2.3. Информация о протоколе

Ранее мы уже видели, что передача данных через сокет осуществляется одними и теми же функциями независимо от протокола. Но при этом программа должна учитывать, является ли протокол потоковым, дейтаграммным или иным. Кроме того, информация о протоколе требуется для создания сокета и для распределения ролей между клиентом и сервером при установлении соединения. Чтобы работать с любым протоколом, программа должна иметь возможность получить всю эту информацию и выполнить на основе ее те или иные действия. Могут также понадобиться такие сведения, как максимальное число сокетов, поддерживаемых провайдером протокола, допустимый диапазон адресов, максимальный размер сообщений для дейтаграммных протоколов и т. д. Для хранения полного описания протокола и его провайдера в WinSock 2 предусмотрена структура WSAPROTOCOL_INFO. Она не описана в модуле WinSock, т. к. в WinSock 1 ее нет. Тем, кто захочет использовать эту структуру, придется самостоятельно добавлять ее описание в программу. Листинг 2.36 показывает, как выглядит эта структура.

Листинг 2.36. Тип WSAPROTOCOL_INFO

```
// ***** Описание на C++ *****
typedef struct _WSAPROTOCOLCHAIN {
  int ChainLen;
  DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, *LPWSAPROTOCOLCHAIN;
```

```
typedef struct _WSAPROTOCOL_INFO {
  DWORD dwServiceFlags1;
  DWORD dwServiceFlags2;
  DWORD dwServiceFlags3;
  DWORD dwServiceFlags4;
  DWORD dwProviderFlags;
  GUID ProviderId;
  DWORD dwCatalogEntryId;
  WSAPROTOCOLCHAIN ProtocolChain;
  int iVersion;
  int iAddressFamily;
  int iMaxSockAddr;
  int iMinSockAddr;
  int iSocketType;
  int iProtocol;
  int iProtocolMaxOffset;
  int iNetworkByteOrder;
  int iSecurityScheme;
  DWORD dwMessageSize;
  DWORD dwProviderReserved;
  TCHAR szProtocol[WSAPROTOCOL_LEN + 1];
} WSAPROTOCOL_INFO, *LPWSAPROTOCOL_INFO;

// ***** Описание на Delphi *****
TWSAProtocolChain = packed record
  ChainLen: Integer;
  ChainEntries: array[0..MAX_PROTOCOL_CHAIN – 1] of DWORD;
end;

// Структура на C++ содержит тип TCHAR, который, как мы
// говорили в главе 1, может означать как Char,
// так и WideChar, т. е. структура должна иметь
// два варианта описания: TWSAProtocolInfoA для
// однобайтной кодировки и TWSAProtocolInfo для
// двухбайтной. Соответственно, все функции,
// использующие эту структуру, реализованы
// в системных библиотеках в двух вариантах.
// Здесь мы приводим только ANSI-вариант.
PWSAProtocolInfo = ^TWSAProtocolInfo;
TWSAProtocolInfo = packed record
  dwServiceFlags1: DWORD;
  dwServiceFlags2: DWORD;
  dwServiceFlags3: DWORD;
```

```
  dwServiceFlags4: DWORD;
  dwProviderFlags: DWORD;
  ProviderId: GUID;
  dwCatalogEntryId: DWORD;
  ProtocolChain: TWSAProtocolChain;
  iVersion: Integer;
  iAddressFamily: Integer;
  iMaxSockAddr: Integer;
  iMinSockAddr: Integer;
  iSocketType: Integer;
  iProtocol: Integer;
  iProtocolMaxOffset: Integer;
  iNetworkByteOrder: Integer;
  iSecurityScheme: Integer;
  dwMessageSize: DWORD;
  dwProviderReserved: DWORD;
  szProtocol: array[0..WSAPROTOCOL_LEN] of Char;
end;
```

Расшифровка полей типа `TWSAProtocolInfo` есть в MSDN, мы здесь не будем ее приводить.

Сама функция `WSAEnumProtocols`, которая позволяет получить список всех протоколов, провайдеры которых установлены на компьютере, приведена в листинге 2.37.

Листинг 2.37. Функция `WSAEnumProtocols`

```
// ***** Описание на C++ *****
int WSAEnumProtocols(
  LPINT lpiProtocols,
  LPWSAPROTOCOL_INFO lpProtocolBuffer,
  LPDWORD lpdwBufferLength);

// ***** Описание на Delphi *****
function WSAEnumProtocols(
  lpiProtocols: PInteger;
  lpProtocolBuffer: PWSAProtocolInfo;
  var BufferLength: DWORD): Integer;
```

Примечание

В старых версиях MSDN в описании этой функции есть небольшая опечатка: тип параметра `lpdwBufferLength` назван `ILPDWORD` вместо `LPDWORD`.

Библиотека WS2_32.dll придерживается тех же правил насчет ANSI- и Unicode-вариантов функций, что и другие системные библиотеки *(см. разд. 1.1.12)*, поэтому в ней нет функции с именем `WSAEnumProtocols`, а есть `WSAEnumProtocolsA` и `WSAEnumProtocolsW`. Эти функции работают с разными вариантами структуры `WSAPROTOCOL_INFO`, которые различаются типом элементов в последнем массиве — `CHAR` или `WCHAR`.

Параметр `lpiProtocols` указывает на первый элемент массива, содержащего список протоколов, информацию о которых нужно получить. Если этот указатель равен `nil`, то возвращается информация обо всех доступных протоколах. Параметр `lpProtocolBuffer` содержит указатель на начало массива структур типа `TWSAProtocolInfo`. Программа должна заранее выделить память под этот массив. Параметр `BufferLength` при вызове должен содержать размер буфера `lpProtocolBuffer` в байтах (именно размер в байтах, а не количество элементов). После завершения функции сюда помещается минимальный размер буфера, необходимый для размещения информации обо всех запрошенных протоколах. Если это значение больше переданного, функция завершается с ошибкой.

Если параметр `lpiProtocols` не равен нулю, он должен содержать указатель на массив, завершающийся нулем. Следовательно, если количество протоколов, запрашиваемых программой, равно N, этот массив должен состоять из $N+1$ элементов, и первые N элементов должны содержать номера протоколов, а последний элемент — ноль.

В системе может быть установлено несколько провайдеров для одного протокола. В этом случае информация о каждом провайдере будет помещена в отдельный элемент массива. Из-за этого число задействованных элементов в массиве `lpProtocolBuffer` может превышать количество протоколов, определяемых параметром `lpiProtocols`.

К сожалению, полную информацию о том, каким протоколам какие номера соответствуют, в документации найти не удалось. Можно только сказать, что для получения информации о протоколе TCP в массив `lpiProtocols` необходимо поместить константу `IPPROTO_TCP`, о протоколе UDP — константу `IPPROTO_UDP`.

Возвращаемое функцией значение равно числу протоколов, информация о которых помещена в массив, если функция выполнена успешно, и `SOCKET_ERROR`, если при ее выполнении возникла ошибка. Конкретная ошибка определяется стандартным методом, с помощью `WSAGetLastError`. Если массив `lpProtocolBuffer` слишком мал для хранения всей требуемой информации, функция завершается с ошибкой `WSAENOBUFS`.

WinSock 1 содержит аналогичную по возможности функцию `EnumProtocols`, возвращающую массив структур `PROTOCOL_INFO`. Эта структура содержит

меньше информации о протоколе, чем WSAPROTOCOL_INFO и, в отличие от последней, не используется никакими другими функциями WinSock. Несмотря на то, что функция EnumProtocols и структура PROTOCOL_INFO описаны в первой версии WinSock, модуль WinSock их не импортирует, при необходимости их нужно импортировать самостоятельно. Но функция EnumProtocols считается устаревшей, использовать ее в новых приложениях не рекомендуется, поэтому практически всегда, за исключением редких случаев, требующих совместимости с WinSock 1, лучше выбрать более современную функцию WSAEnumProtocols.

2.2.4. Новые функции

В этом разделе мы рассмотрим некоторые новые функции, появившиеся в WinSock 2. Большинство из них позволяет выполнять действия, уже знакомые нам из предыдущих разделов, но предоставляет бóльшие возможности, чем стандартные сокетные функции.

Для создания сокета предназначена функция WSASocket со следующим прототипом (листинг 2.38).

Листинг 2.38. Функция WSASocket

```
// ***** Описание на C++ *****
SOCKET WSASocket(int af, int SockType, int protocol,
  LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g, DWORD dwFlags);

// ***** Описание на Delphi *****
function WSASocket(AF, SockType, Protocol: Integer;
  lpProtocolInfo: PWSAProtocolInfo; g: TGroup;
  dwFlags: DWORD): TSocket;
```

Первые три параметра совпадают с тремя параметрами функции socket. Параметр lpProtocolInfo указывает на структуру TWSAProtocolInfo, содержащую информацию о протоколе, для которого создается сокет. Если этот указатель равен nil, функция создает сокет на основании первых трех параметров так же, как это делает функция socket. С другой стороны, если этот параметр не равен nil, то структура, на которую он указывает, содержит всю информацию, необходимую для создания сокета, поэтому первые три параметра должны быть равны константе FROM_PROTOCOL_INFO (–1). Параметр g зарезервирован для использования в будущем и должен быть равен нулю (тип TGroup совпадает с DWORD). Последний параметр dwFlags определяет, какие дополнительные возможности имеет создаваемый сокет. Вызов функции socket эквивалентен вызову функции WSASocket с флагом WSA_FLAG_OVERLAPPED, который показывает, что данный сокет можно использовать для перекрытого ввода-

вывода *(см. разд. 2.2.9)*. Остальные флаги нужны при многоадресной рассылке (не все из них допустимы для протоколов TCP и UDP). Эти флаги мы рассмотрим в *разд. 2.2.11*.

В случае TCP и UDP функция `WSASocket` дает следующие преимущества по сравнению с функцией `socket`. Во-первых, через параметр `lpProtocolInfo` появляется возможность явно указать провайдера, который будет выбран программой. Во-вторых, если программа не использует перекрытый ввод-вывод, можно создавать сокеты без флага `WSA_FLAG_OVERLAPPED`, экономя при этом некоторое незначительное количество ресурсов. Кроме того, как это будет обсуждаться далее, с помощью `WSASocket` две разных программы могут работать с одним и тем же сокетом.

Функция `WSAConnect` — это более мощный аналог `connect`. Ее прототип приведен в листинге 2.39.

Листинг 2.39. Функция `WSAConnect` и связанные с ней типы

```
// ***** Описание на C++ *****
int WSAConnect(
  SOCKET s,
  const struct sockaddr FAR *name,
  int namelen,
  LPWSABUF lpCallerData,
  LPWSABUF lpCalleeData,
  LPQOS lpSQOS,
  LPQOS lpGQOS);

typedef struct __WSABUF {
  u_long len;
  char FAR *buf;} WSABUF, FAR * LPWSABUF;

// ***** Описание на Delphi *****
function WSAConnect(
  S: TSocket;
  var Name: TSockAddr;
  NameLen: Integer;
  lpCallerData, lpCalleeData: PWSABuf;
  lpSQOS, lpGQOS: PQOS);

PWSABuf = ^TWSABuf;
TWSABuf = packed record
  Len: Cardinal;
  Buf: PChar;
end;
```

Функция WSAConnect устанавливает соединение со стороны клиента. Ее первые три параметра совпадают с параметрами функции connect. Параметры lpCallerData и lpCalleeData служат для передачи данных от клиента серверу и от сервера клиенту при установлении соединения. Они оба являются указателями на структуру TWSABuf тип TWSABuf, которая содержит размер буфера Len и указатель на буфер Buf. Протоколы стека TCP/IP не поддерживают передачу данных при соединении, поэтому для TCP и UDP lpCallerData и lpCalleeData должны быть равны nil. Параметры lpSQOS и lpGQOS — это указатели на структуры, с помощью которых программа передает свои требования к качеству обслуживания, причем параметр lpGQOS связан с не поддерживаемым в настоящий момент групповым качеством и всегда должен быть равен nil. Параметр lpSQOS также должен быть равен nil, если программа не предъявляет требований к качеству обслуживания. Так как рассмотрение качества обслуживания выходит за рамки данной книги, мы не приводим здесь определение структуры SQOS, которое при необходимости легко найти в MSDN.

Между функциями connect и WSAConnect существует небольшое различие при работе с сокетами, не поддерживающими соединение. Как вы знаете из *разд. 2.1.9*, функция connect может использоваться с такими сокетами для задания адреса отправки по умолчанию и автоматической фильтрации входящих пакетов. Для того чтобы отменить такое "соединение", нужно при вызове функции connect указать адрес INADDR_ANY и нулевой порт. В случае WSAConnect для отмены "соединения" требуется, чтобы все без исключения поля структуры Name, включая sin_family, были нулевыми. Это сделано для того, чтобы обеспечить независимость от протокола: при любом протоколе для разрыва "соединения" должно устанавливаться одно и то же значение Name.

Если программа не предъявляет требований к качеству обслуживания, то для протоколов TCP и UDP функция WSAConnect не предоставляет никаких преимуществ по сравнению с connect.

Функция accept из стандартной библиотеки сокетов позволяет серверу извлечь из очереди соединений информацию о подключившемся клиенте и создать сокет для его обслуживания. Эти действия выполняются безусловно, для любых подключившихся клиентов. Если сервер допускает подключение не любых клиентов, а только тех, которые отвечают некоторым условиям (для протокола TCP эти условия могут заключаться в том, какие IP-адреса и какие порты допустимо использовать клиентам), сразу после установления соединения его приходится разрывать, если клиент не удовлетворяет этим условиям. Для упрощения этой операции в WinSock 2 предусмотрена функция WSAAccept, прототип которой приведен в листинге 2.40.

Листинг 2.40. Функция WSAAccept

```
// ***** Описание на C++ *****
SOCKET WSAAccept(
  SOCKET s,
  struct sockaddr FAR *addr,
  LPINT addrlen,
  LPCONDITIONPROC lpfnCondition,
  DWORD dwCallbackData);

// ***** Описание на Delphi *****
function WSAAccept(
  S: TSocket;
  Addr: PSockAddr;
  AddrLen: PInteger;
  lpfnCondition: TConditionProc;
  dwCallbackData: DWORD): TSocket;
```

По сравнению с уже известной нам функцией accept функция WSAAccept имеет два новых параметра: lpfnCondition и dwCallbackData. lpfnCondition является указателем на функцию обратного вызова. Эта функция объявляется и реализуется программой. WSAAccept вызывает ее внутри себя и в зависимости от ее результата принимает или отклоняет соединение. Параметр dwCallbackData не имеет смысла для самой функции WSAAccept и передается без изменений в функцию обратного вызова. Тип TConditionProc должен быть объявлен следующим образом (листинг 2.41).

Листинг 2.41. Тип TConditionProc

```
// ***** Описание на C++ *****
typedef (int*) (
  LPWSABUF lpCallerId,
  LPWSABUF lpCallerData,
  LPQOS lpSQOS,
  LPQOS lpGQOS,
  LPWSABUF lpCalleeId,
  LPWSABUF lpCalleeData,
  GROUP FAR *g,
  DWORD dwCallbackData) LPCONDITIONPROC;

// ***** Описание на Delphi *****
TConditionProc = function(
  lpCallerId, lpCallerData: PWSABuf;
```

```
lpSQOS, lpGQOS: PQOS;
lpCalleeID, lpCalleeData: PWSABuf;
g: PGroup;
dwCallbackData: DWORD): Integer; stdcall;
```

Параметр lpCallerId указывает на буфер, в котором хранится адрес подключившегося клиента. При работе со стеком TCP/IP lpCallerId^.Len будет равен SizeOf(TSockAddr), а lpCallerId^.Buf будет указывать на структуру TSockAddr, содержащую адрес клиента. Параметр lpCallerData определяет буфер, в котором хранятся данные, переданные клиентом при соединении. Как уже отмечалось, протоколы стека TCP/IP не поддерживают передачу данных при соединении, поэтому для них этот параметр будет равен nil.

Параметры lpSQOS и lpGQOS задают требуемое клиентом качество обслуживания для сокета и для группы соответственно. Так как группы сокетов в текущей реализации WinSock не поддерживаются, параметр lpGQOS будет равен nil. Параметр lpSQOS тоже будет равен nil, если клиент не задал качество обслуживания при соединении.

Параметр lpCalleeId содержит адрес интерфейса, принявшего соединение (поля структуры при этом используются так же, как у параметра lpCallerId). Ранее уже обсуждалось, что сокет, привязанный к адресу INADDR_ANY, прослушивает все сетевые интерфейсы, имеющиеся на компьютере, но каждое подключение, созданное с его помощью, использует конкретный интерфейс. Параметр lpCalleeId содержит адрес, привязанный к конкретному соединению. Параметр lpCalleeData указывает на буфер, в который сервер может поместить данные для отправки клиенту. Этот параметр также не имеет смысла для протокола TCP, не поддерживающего отправку данных при соединении.

Параметр g выходной, он позволяет управлять присоединением создаваемого функцией WSAAccept сокета к группе. Параметр, как и все, связанное с группами, зарезервирован для использования в будущем.

Примечание

Если вы пользуетесь старой версией MSDN, то можете не обнаружить там описания параметра g — оно там отсутствует. Видимо, просто по ошибке.

И наконец, через параметр dwCallbackData в функцию обратного вызова передается значение параметра dwCallbackData, переданное в функцию WSAAccept. Программист должен сам решить, как ему интерпретировать это значение.

Функция должна вернуть CF_ACCEPT (0), если соединение принимается, CF_REJECT (1), если оно отклоняется, и CF_DEFER (2), если решение о разрешении или запрете соединения откладывается. Если функция обратного вызова

вернула CF_REJECT, то WSAAccept завершается с ошибкой WSAECONNREFUSED, если CF_DEFER — то с ошибкой WSATRY_AGAIN (в последнем случае соединение остается в очереди, и информация о нем вновь будет передана в функцию обратного вызова при следующем вызове WSAAccept). Обе эти ошибки не фатальные, сокет остается в режиме ожидания соединения и может принимать подключения от новых клиентов.

Ранее уже обсуждалось, что функция connect на стороне клиента считается успешно завершенной тогда, когда соединение встало в очередь, а не тогда, когда оно реально принято сервером через функцию accept. По умолчанию для клиента, соединение с которым сервер отклонил, нет разницы, вызвал ли сервер функцию WSAAccept и сразу отклонил соединение, или установил его с помощью accept, а потом разорвал. В обоих случаях клиент сначала получит информацию об успешном соединении с сервером, а потом это соединение будет разорвано. Но при использовании WSAAccept можно установить такой режим работы, когда сначала выполняется функция, заданная параметром lpCondition, и лишь потом клиенту отправляется разрешение или запрет на подключение. Включается этот режим установкой параметра слушающего сокета SO_CONDITIONAL_ACCEPT, что иллюстрирует листинг 2.42.

Листинг 2.42. Включение режима ожидания реального подключения

```
var
  Cond: BOOL;
begin
  Cond := True;
  setsockopt(S, SOL_SOCKET, SO_CONDITIONAL_ACCEPT,
    PChar(@Cond), SizeOf(Cond));
```

Этот режим снижает нагрузку на сеть и повышает устойчивость сервера против DoS-атак, заключающихся в многократном подключении-отключении посторонних клиентов, поэтому в серьезных серверах рекомендуется использовать эту возможность.

Из сказанного следует, что при использовании протокола TCP функция WSAAccept по сравнению с accept дает два принципиальных преимущества: позволяет управлять качеством обслуживания и запрещать подключение нежелательных клиентов.

Некоторые протоколы поддерживают передачу информации не только при установлении связи, но и при ее завершении. Для таких протоколов в WinSock2 предусмотрены функции WSASendDisconnect и WSARecvDisconnect. Так как протокол TCP не поддерживает передачу данных при закрытии со-

единения, для него эти функции не дают никаких преимуществ по сравнению с вызовом функции shutdown, поэтому мы не будем их здесь рассматривать.

Далее мы рассмотрим несколько новых функций, унифицирующих работу с различными протоколами.

Функция inet_addr, как это уже упоминалось, жестко связана с протоколом IP и не имеет смысла для других протоколов. WinSock 2 предлагает вместо нее функцию WSAStringToAddress, имеющую следующий прототип (листинг 2.43).

Листинг 2.43. Функция WSAStringToAddress

```
// ***** Описание на C++ *****
INT WSAStringToAddress(
  LPTSTR AddressString,
  INT AddressFamily,
  LPWSAPROTOCOL_INFO lpProtocolInfo,
  LPSOCKADDR lpAddress,
  LPINT lpAddressLength);

// ***** Описание на Delphi *****
function WSAStringToAddress(
  AddressString: PChar;
  AddressFamily: Integer;
  lpProtocolInfo: PWSAProtocolInfo;
  var Address:TSockAddr;
  var AddressLength:Integer):Integer;
```

Данная функция преобразует строку, задающую адрес сокета, в адрес, хранящийся в структуре TSockAddr. Параметр AddressString указывает на строку, хранящую адрес, параметр AddressFamily — на семейство адресов, для которого осуществляется трансляция. Если есть необходимость выбрать конкретный провайдер для протокола, в функцию может быть передан параметр lpProtocolInfo, в котором указан идентификатор провайдера. Если же программу устраивает провайдер по умолчанию, параметр lpProtocolInfo должен быть равен nil. Адрес возвращается через параметр Address. Параметр AddressLength при вызове функции должен содержать размер буфера, переданного через Address, а на выходе содержит реально использованное число байтов в буфере.

Функция возвращает 0 в случае успешного выполнения и SOCKET_ERROR — при ошибке.

Допустимый формат строки определяется протоколом (некоторые протоколы вообще не поддерживают текстовую запись адреса, и для них функция WSAStringToAddress неприменима). Для семейства AF_INET, к которому относятся TCP и UDP, адрес может задаваться в виде "IP1.IP2.IP3.IP4:Port" или "IP1.IP2.IP3.IP4", где IPn — *n*-й компонент IP-адреса, записанного в виде 4-байтных полей, Port — номер порта. Если порт явно не указан, устанавливается нулевой номер порта.

Таким образом, чтобы в структуре TSockAddr оказался, например, адрес 192.168.100.217 и порт с номером 5000, необходимо выполнить следующий код (листинг 2.44).

Листинг 2.44. Пример использования функции WSAStringToAddress

```
var
  Addr: TSockAddr;
  AddrLen: Integer;
begin
  AddrLen := SizeOf(Addr);
  WSAStringToAddress('192.168.100.217:5000', AF_INET, nil, Addr,
    AddrLen);
```

Существует также функция WSAAddressToString, обратная к WSAStringToAddress. Ее прототип приведен в листинге 2.45.

Листинг 2.45. Функция WSAAddressToString

```
// ***** Описание на C++ *****
INT WSAAddressToString(
  LPSOCKADDR lpsaAddress,
  DWORD dwAddressLength,
  LPWSAPROTOCOL_INFO lpProtocolInfo,
  LPTSTR lpszAddressString,
  LPDWORD lpdwAddressStringLength);

// ***** Описание на Delphi *****
function WSAAddressToString(
  var Address: TSockAddr;
  dwAddressLength: DWORD;
  lpProtocolInfo: PWSAProtocolInfo;
  lpszAddressString: PChar;
  var AddressStringLength: DWORD): Integer;
```

Как нетрудно догадаться по названию функции, она преобразует адрес, заданный структурой TSockAddr, в строку. Адрес задается параметром Address,

параметр dwAddressLength определяет длину буфера Address. Необязательный параметр lpProtocolInfo содержит указатель на структуру TWSAProtocolInfo, с помощью которой можно определить, какой именно провайдер должен выполнить преобразование. Параметр lpszAddressString содержит указатель на буфер, заранее выделенный программой, в который будет помещена строка. Параметр AddressStringLength на входе должен содержать размер буфера, заданного параметром lpszAddressString, а на выходе содержит длину получившейся строки.

Функция возвращает ноль в случае успеха и SOCKET_ERROR — при ошибке.

Ранее мы уже обсуждали различные форматы представления целых чисел, а также то, что формат, предусмотренный сетевым протоколом, может не совпадать с форматом, используемым узлом. Напомним, что для преобразования из сетевого формата в формат узла предназначены функции htons, ntohs, htonl и ntohl, привязанные к протоколам стека TCP/IP (другие протоколы могут иметь другой формат представления чисел). WinSock 2 предлагает аналоги этих функций WSAHtons, WSANtohs, WSAHtonl и WSANtohl, которые учитывают особенности конкретного протокола. Мы здесь рассмотрим только функцию WSANtohl, преобразующую 32-битное целое из сетевого формата в формат узла. Остальные три функции работают аналогично. Листинг 2.46 содержит прототип функции WSANtohl.

Листинг 2.46. Функция WSANtohl

```
// ***** Описание на C++ *****
int WSANtohl(SOCKET s, u_long netlong, u_long FAR *lphostlong);

// ***** Описание на Delphi *****
function WSANtohl(S: TSocket; NetLong: Cardianl;
  var HostLong: Cardinal): Integer;
```

Параметр S задает сокет, для которого осуществляется преобразование. Так как сокет всегда связан с конкретным протоколом, этого параметра достаточно, чтобы библиотека могла определить, по какому закону преобразовывать число из сетевого формата в формат хоста. Число в сетевом формате задается параметром NetLong, результат преобразования помещается в параметр HostLong. Функция возвращает ноль в случае успешного выполнения операции и SOCKET_ERROR — при ошибке.

Если программа работает только с протоколами стека TCP/IP, старые варианты функций удобнее новых, потому что возвращают непосредственно результат преобразования, который можно использовать в выражениях. При работе с новыми функциями для получения результата следует заводить отдельную

переменную, поэтому эти функции целесообразны тогда, когда программа должна единым образом работать с разными протоколами.

Последняя функция, которую мы здесь рассмотрим, не имеет прямых аналогов среди старых функций. Называется она WSADuplicateSocket и служит для копирования дескриптора сокета в другой процесс. Прототип функции WSADuplicateSocket приведен в листинге 2.47.

Листинг 2.47. Функция WSADuplicateSocket

```
// ***** Описание на C++ *****
int WSADuplicateSocket(
  SOCKET s,
  DWORD dwProcessId,
  LPWSAPROTOCOL_INFO lpProtocolInfo);

// ***** Описание на Delphi *****
function WSADuplicateSocket(
  S: TSocket;
  dwProcessID: DWORD;
  var ProtocolInfo: TWSAProtocolInfo): Integer;
```

Параметр S задает сокет, дескриптор которого нужно скопировать, параметр dwProcessID — идентификатор процесса, для которого предназначена копия. Функция помещает в структуру ProtocolInfo информацию, необходимую для создания копии дескриптора другим процессом. Затем эта структура должна быть каким-то образом передана другому процессу, который передаст ее в функцию WSASocket и получит свою копию дескриптора для работы с данным сокетом.

Функция WSADuplicateSocket возвращает ноль при успешном завершении и SOCKET_ERROR — в случае ошибки.

Как мы помним, сокет является объектом, внутренняя структура которого остается скрытой от использующей его программы. Программа манипулирует только дескриптором сокета — некоторым уникальным идентификатором этого объекта. Функция WSADuplicateSocket позволяет другой программе получить новый дескриптор для уже существующего сокета. Старый и новый дескриптор становятся равноправными. Чтобы освободить сокет, нужно закрыть все его дескрипторы с помощью функции closesocket.

Если во входной буфер сокета поступают данные, их получит та программа, которая первой вызовет соответствующую функцию чтения, поэтому совместное использование одного сокета разными программами требует синхронизации их работы. MSDN рекомендует такую схему работы, при которой одна программа только создает сокет и устанавливает соединение, а затем переда-

ет сокет другой программе, которая реализует через него ввод-вывод. Первая программа при этом закрывает свой дескриптор. Такой алгоритм работы позволяет полностью исключить проблемы, возникающие при совместном доступе разных программ к одному сокету.

Отметим, что функция `WSADuplicateSocket` может быть полезна только для копирования дескрипторов между разными процессами. Разные нити одного процесса не нуждаются в этой функции, т. к., находясь в одном адресном пространстве, они могут работать с одним и тем же дескриптором.

2.2.5. Асинхронный режим, основанный на сообщениях

Все операции с сокетами, которые мы рассматривали раньше, являлись синхронными. Программа, использующая такие сокеты, должна сама время от времени проверять тем или иным способом, пришли ли данные, установлена ли связь и т. п. Асинхронные сокеты позволяют программе получать уведомления о событиях, происходящих с сокетом: поступлении данных, освобождении места в буфере, закрытии и т. п. Такой способ работы лучше подходит для событийно-ориентированных программ, типичных для Windows.

Поддержка асинхронных сокетов впервые появилась в WinSock 1 и была основана на сообщениях, которые обрабатывались оконными процедурами. В WinSock 2 этот асинхронный режим остался без изменений. Программист указывает, какое сообщение какому окну должно приходить при возникновении события на интересующем его сокете.

Асинхронный режим с уведомлением через сообщения устанавливается функцией `WSAAsyncSelect`, имеющей следующий прототип:

```
function WSAAsyncSelect(S: TSocket; HWindow: HWND;
  wMsg: u_int; lEvent: LongInt): Integer;
```

Параметр `S` определяет сокет, для которого устанавливается асинхронный режим работы. Параметр `HWindow` — дескриптор окна, которому будут приходить сообщения, `wMsg` — сообщение, а `lEvent` задает события, которые вызывают отправку сообщения. Для этого параметра определены константы, комбинация которых задает интересующие программу события. Мы не будем рассматривать здесь все возможные события, остановимся только на самых главных (табл. 2.2).

Каждый последующий вызов `WSAAsyncSelect` для одного и того же сокета отменяет предыдущий вызов. Таким образом, в результате выполнения следующего кода форма будет получать только сообщения, показывающие готовность сокета к чтению, а готовность к записи не приведет к отправке сообщения (листинг 2.48).

Таблица 2.2. *Асинхронные события сокета*

Событие	Комментарий
FD_READ	Сокет готов к чтению
FD_WRITE	Сокет готов к записи
FD_ACCEPT	В очереди сокета есть подключения (применимо только для сокетов, находящихся в режиме ожидания подключения)
FD_CONNECT	Соединение установлено (применимо только для сокетов, для которых вызвана функция connect или аналогичная ей)
FD_CLOSE	Соединение закрыто

Листинг 2.48. Последовательный вызов функции WSAAsyncSelect

```
WSAAsyncSelect(S, Form1.Handle, WM_USER, FD_WRITE);
// Второй вызов отменит результаты первого
WSAAsyncSelect(S, Form1.Handle, WM_USER, FD_READ);
// Теперь окно не будет получать уведомления о возможности записи
```

WSAAsyncSelect связывает с сообщением именно сокет, а не его дескриптор. Это означает, что если две программы используют один сокет (копия дескриптора которого была создана с помощью функции WSADuplicateSocket), и первая программа вызывает WSAAsyncSelect со своим дескриптором, а затем вторая — со своим, то вызов WSAAsyncSelect, сделанный во второй программе, отменит вызов, сделанный в первой.

Для того чтобы получать сообщения при готовности сокета как к чтению, так и к записи, нужно выполнить следующий код:

```
WSAAsyncSelect(S, Form1.Handle, WM_USER, FD_READ or FD_WRITE);
```

При необходимости с помощью or можно комбинировать и большее число констант.

Из сказанного следует, что нельзя связать с разными событиями одного и того же сокета разные сообщения (или отправлять сообщения разным окнам), т. к. при одном вызове WSAAsyncSelect можно передать только один дескриптор окна и один номер сообщения, а следующий вызов этой функции, с другим дескриптором и/или номером, отменит предыдущий.

Функция WSAAsyncSelect переводит сокет в неблокирующий режим. Если необходимо использовать асинхронный сокет в блокирующем режиме, после вызова WSAAsyncSelect требуется перевести его в этот режим вручную.

Сообщение, которое связывается с асинхронным сокетом, может быть любым. Обычно его номер выбирают от WM_USER и выше, чтобы исключить путаницу со стандартными сообщениями.

При получении сообщения его параметр wParam содержит дескриптор сокета, на котором произошло событие. Младшее слово lParam содержит произошедшее событие (одну из констант FD_XXX), а старшее слово — код ошибки, если она произошла. Для выделения кода события и кода ошибки из lParam в библиотеке WinSock предусмотрены макросы WSAGETSELECTEVENT и WSAGETSELECTERROR соответственно. В модуле WinSock они заменены функциями WSAGetSelectEvent и WSAGetSelectError. Одно сообщение может информировать только об одном событии на сокете. Если произошло несколько событий, в очередь окна будет добавлено несколько сообщений.

Сокет, созданный при вызове функции accept, наследует режим того сокета, который принял соединения. Таким образом, если сокет, находящийся в режиме ожидания подключения, является асинхронным, то и сокет, порожденный функцией accept, будет асинхронным, и тот же набор его событий будет связан с тем же сообщением, что и у исходного сокета.

Рассмотрим подробнее каждое из перечисленных событий.

Событие FD_READ возникает, когда во входной буфер сокета поступают данные (если на момент вызова WSAAsyncSelect, разрешающего такие события, в буфере сокета уже есть данные, то событие также возникает). Как только соответствующее сообщение помещается в очередь окна, дальнейшая генерация таких сообщений для этого сокета блокируется, т. е. получение новых данных не будет приводить к появлению новых сообщений (при этом сообщения, связанные с другими событиями этого сокета или с событием FD_READ других сокетов, будут по-прежнему помещаться при необходимости в очередь окна). Генерация сообщений снова разрешается после того, как будет вызвана функция для чтения данных из буфера сокета (это может быть функция recv, recvfrom, WSARecv или WSARecvFrom; мы в дальнейшем будем говорить только о функции recv, потому что остальные ведут себя в этом отношении аналогично).

Если после вызова recv в буфере асинхронного сокета остались данные, в очередь окна снова помещается это же сообщение. Благодаря этому программа может обрабатывать большие массивы по частям. Действительно, пусть в буфер сокета приходят данные, которые программа хочет забирать оттуда по частям. Приход этих данных вызывает событие FD_READ, сообщение о котором помещается в очередь. Когда программа начинает обрабатывать это сообщение, она вызывает recv и читает часть данных из буфера. Так как данные в буфере еще есть, снова генерируется сообщение о событии FD_READ, которое

ставится в конец очереди. Через некоторое время программа снова начинает обрабатывать это сообщение. Если и на этот раз данные будут прочитаны не полностью, в очередь снова будет добавлено такое же сообщение. И так будет продолжаться до тех пор, пока не будут прочитаны все полученные данные.

Описанная схема, в принципе, достаточно удобна, но следует учитывать, что в некоторых случаях она может давать ложные срабатывания, т. е. при обработке сообщения о событии `FD_READ` функция `recv` завершится с ошибкой `WSAEWOULDBLOCK`, показывающей, что входной буфер сокета пуст.

Если программа читает данные из буфера не только при обработке `FD_READ`, может возникнуть следующая ситуация: в буфер сокета поступают данные. Сообщение о событии `FD_READ` помещается в очередь. Программа в это время обрабатывает какое-то другое сообщение, при обработке которого также читаются данные. В результате все данные извлекаются из буфера, и он остается пустым. Когда очередь доходит до обработки `FD_READ`, читать из буфера уже нечего.

Другой вариант ложного срабатывания возможен, если программа при обработке `FD_READ` читает данные из буфера по частям, вызывая `recv` несколько раз. Каждый вызов `recv`, за исключением последнего, приводит к тому, что в очередь ставится новое сообщение о событии `FD_READ`. Чтобы избежать появления пустых сообщений в подобных случаях, MSDN рекомендует перед началом чтения отключить для данного сокета реакцию на поступление данных, вызвав для него `WSAAsyncSelect` без `FD_READ`, а перед последним вызовом `recv` — снова включить.

И наконец, следует помнить, что сообщение о событии `FD_READ` можно получить и после того, как с помощью `WSAAsyncSelect` сокет будет переведен в синхронный режим. Это может случиться в том случае, когда на момент вызова `WSAAsyncSelect` в очереди еще остались необработанные сообщения о событиях на данном сокете. Впрочем, это касается не только `FD_READ`, а вообще любого события.

Событие `FD_WRITE` информирует программу о том, что в выходном буфере сокета есть место для данных. Вообще говоря, оно там есть практически всегда, если только программа не отправляет постоянно большие объемы данных. Следовательно, механизм генерации этого сообщения должен быть таким, чтобы не забивать очередь программы постоянными сообщениями о том, что в буфере есть место, а посылать эти сообщения только тогда, когда программа действительно нуждается в такой информации.

При использовании TCP первый раз сообщение, уведомляющее о событии `FD_WRITE`, присылается сразу после успешного завершения операции подклю-

чения к серверу с помощью connect, если речь идет о клиенте, или сразу после создания сокета функцией accept или ее аналогом в случае сервера. В случае UDP это событие возникает после привязки сокета к адресу явным или неявным вызовом функции bind. Если на момент вызова WSAAsyncSelect описанные действия уже выполнены, событие FD_WRITE также генерируется.

В следующий раз событие может возникнуть только в том случае, если функция send (или sendto) не смогла положить данные в буфер из-за нехватки места в нем (в этом случае функция вернет значение, меньшее, чем размер переданных данных, или завершится с ошибкой WSAEWOULDBLOCK). Как только в выходном буфере сокета снова появится свободное место, возникнет событие FD_WRITE, показывающее, что программа может продолжить отправку данных. Если же программа отправляет данные не очень большими порциями и относительно редко, не переполняя буфер, то второй раз событие FD_WRITE не возникнет никогда.

Событие FD_ACCEPT во многом похоже на FD_READ, за исключением того, что оно возникает не при получении данных, а при подключении клиента. После постановки сообщения о событии FD_ACCEPT в очередь новые сообщения о FD_ACCEPT для данного сокета в очередь не ставятся, пока не будет вызвана функция accept или WSAAccept. При вызове одной из этих функций сообщение о событии вновь помещается в очередь окна, если в очереди подключений после вызова функции остаются подключения.

Событие FD_CONNECT возникает при установлении соединения для сокетов, поддерживающих соединение. Для клиентских сокетов оно возникает после завершения процедуры установления связи, начатой с помощью функции connect, для серверных — после создания нового сокета с помощью функции accept (событие возникает именно на новом сокете, а не на том, который находится в режиме ожидания подключения). В MSDN написано, что оно должно возникать также и после выполнения connect для сокетов, не поддерживающих соединение, однако для UDP практика это не подтверждает. Событие FD_CONNECT также возникает, если при попытке установить соединение произошла ошибка (например, оказался недоступен указанный сетевой адрес). Поэтому при получении этого события необходимо анализировать старшее слово параметра lParam, чтобы понять, удалось ли установить соединение.

Событие FD_CLOSE возникает только для сокетов, поддерживающих соединение, при разрыве такого соединения нормальным образом или в результате ошибки связи. Если удаленная сторона для завершения соединения использует функцию shutdown, то FD_CLOSE возникает после вызова этой функции с параметром SD_SEND. При этом соединение закрыто еще не полностью, удаленная сторона еще может получать данные, поэтому при обработке FD_CLOSE

можно попытаться отправить те данные, которые в этом нуждаются. Однако гарантии, что вызов функции отправки не завершится неудачей, нет, т. к. удаленная сторона может закрывать сокет сразу, не прибегая к shutdown.

Рекомендуемая последовательность действий при завершении связи такова. Сначала клиент завершает отправку данных через сокет, вызывая функцию shutdown с параметром SD_SEND. Сервер при этом получает событие FD_CLOSE. Сервер отсылает данные клиенту (при этом клиент получает одно или несколько событий FD_READ), а затем также завершает отправку данных с помощью shutdown с параметром SD_SEND. Клиент при этом получает событие FD_CLOSE, в ответ на которое закрывает сокет с помощью closesocket. Сервер, в свою очередь, сразу после вызова shutdown также вызывает closesocket.

В листинге 2.49 приведен пример кода сервера, использующего асинхронные сокеты. Сервер работает в режиме запрос-ответ, т. е. посылает какие-то данные клиенту только в ответ на его запросы. Константа WM_SOCKETEVENT, определенная в коде для сообщений, связанных с сокетом, может, в принципе, иметь и другие значения.

Листинг 2.49. Пример простого сервера на асинхронных сокетах

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, WinSock;

const
  WM_SOCKETEVENT = WM_USER + 1;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    ServSock: TSocket;
    procedure WMSocketEvent(var Msg: TMessage); message WM_SOCKETEVENT;
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
var
  Data: TWSAData;
  Addr: TSockAddr;
begin
  WSAStartup($101, Data);
  // Обычная последовательность действий по созданию сокета,
  // привязке его к адресу и установлению на прослушивание
  ServSock := socket(AF_INET, SOCK_STREAM, 0);
  Addr.sin_family := AF_INET;
  Addr.sin_addr.S_addr := INADDR_ANY;
  Addr.sin_port := htons(3320);
  FillChar(Addr.sin_zero, SizeOf(Addr.sin_zero), 0);
  bind(ServSock, Addr, SizeOf(Addr));
  listen(ServSock, SOMAXCONN);
  // Перевод сокета в асинхронный режим. Кроме события FD_ACCEPT
  // указаны также события FD_READ и FD_CLOSE, которые никогда не
  // возникают на сокете, установленном в режим прослушивания.
  // Это сделано потому, что сокеты, созданные с помощью функции
  // accept, наследуют асинхронный режим, установленный для
  // слушающего сокета. Таким образом, не придется вызывать
  // функцию WSAAsyncSelect для этих сокетов — для них сразу
  // будет назначен обработчик событий FD_READ и FD_CLOSE.
  WSAAsyncSelect(ServSock, Handle,
    WM_SOCKETEVENT, FD_READ or FD_ACCEPT or FD_CLOSE);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  closesocket(ServSock);
  WSACleanup;
end;

procedure TForm1.WMSocketEvent(var Msg: TMessage);
var Sock: TSocket;
    SockError: Integer;
begin
  Sock := TSocket(Msg.WParam);
  SockError := WSAGetSelectError(Msg.lParam);
```

```
  if SockError <> 0 then
  begin
    // Здесь должен быть анализ ошибки
    closesocket(Sock);
    Exit;
  end;
  case WSAGetSelectEvent(Msg.lParam) of
    FD_READ:
      begin
        // Пришел запрос от клиента. Необходимо прочитать данные,
        // сформировать ответ и отправить его.
      end;
    FD_ACCEPT:
      begin
        // Просто вызываем функцию accept. Ее результат нигде не
        // сохраняется, потому что вновь созданный сокет автоматически
        // начинает работать в асинхронном режиме, и его дескриптор
        // при необходимости будет передан через Msg.wParam при
        // возникновении события
        accept(Sock, nil, nil);
      end;
    FD_CLOSE:
      begin
        // Получив от клиента сигнал завершения, сервер, в принципе,
        // может попытаться отправить ему данные. После этого сервер
        // также должен закрыть соединение со своей стороны
        shutdown(Sock, SD_SEND);
        closesocket(Sock);
      end;
  end;
end;

end.
```

Преимущество такого сервера по сравнению с сервером, основанным на функции select, заключается в том, что он не должен постоянно проверять наличие полученных данных — когда данные поступят, он без дополнительных усилий получит уведомление об этом. Кроме того, этот сервер не имеет проблем, связанных с количеством сокетов в множестве типа TFDSet. Впрочем, последнее несущественно, т. к. при таком количестве клиентов сервер обычно реализует другие, более производительные способы взаимодействия с клиентами.

2.2.6. Пример сервера, основанного на сообщениях

В этом разделе мы напишем сервер, использующий асинхронные сокеты и их сообщения (пример AsyncSelectServer на компакт-диске). Этот сервер будет во многом похож на сервер на основе неблокирующих сокетов *(см. разд. 2.1.16)*, только он не станет проверять по таймеру наличие данных в буфере и возможность отправки данных, а будет выполнять это тогда, когда поступят соответствующие сообщения.

Такая схема работы требует более осторожного подхода. По сигналу от таймера мы сами проверяем, на каком этапе в данный момент находится обмен данными с клиентом. Если, например, идет этап отправки данных, то проверять входной буфер сокета не нужно, можно оставить это до тех пор, пока не наступит этап чтения данных. При использовании сообщений приходится учитывать, что сообщение о поступлении данных в буфер сокета может прийти в любой момент, в том числе и тогда, когда обмен с клиентом находится на этапе отправки строки. По протоколу сервер не должен читать сообщение в этот момент, необходимо сначала закончить отправку, поэтому приходится данное уведомление игнорировать. Но второго уведомления система не пришлет, соответственно, после окончания отправки данных сервер должен сам вспомнить, что было уведомление, и перейти к операции чтения.

Примечание

Вообще говоря, ситуация, когда сервер не отправит данные за один раз, и их отправка растянется на несколько итераций петли сообщений, настолько редка, что при разработке сервера, к которому не предъявляются повышенные требования по надежности, ее можно было бы вообще не учитывать. Соответственно, возможность получения сервером нового уведомления о поступлении данных до того, как на старое сообщение будет дан ответ, возможна только тогда, когда клиент не соблюдает принятый протокол и посылает несколько сообщений подряд, не дожидаясь ответа. Наш пример призван продемонстрировать наиболее надежный к подобным действиям клиента сервер, поэтому мы его напишем "по всем правилам".

Как обычно, работа сервера начинается с инициализации слушающего сокета, выполняющейся при нажатии кнопки **Запустить** (листинг 2.50).

Листинг 2.50. Инициализация сервера, основанного на сообщениях

```
procedure TServerForm.BtnStartServerClick(Sender: TObject);
var
  // Адрес, к которому привязывается слушающий сокет
  ServerAddr: TSockAddr;
```

```
begin
  // Формируем адрес для привязки.
  FillChar(ServerAddr.sin_zero, SizeOf(ServerAddr.sin_zero), 0);
  ServerAddr.sin_family := AF_INET;
  ServerAddr.sin_addr.S_addr := INADDR_ANY;
  try
    ServerAddr.sin_port := htons(StrToInt(EditPortNumber.Text));
    if ServerAddr.sin_port = 0 then
    begin
      MessageDlg('Номер порта должен находиться в диапазоне 1-65535',
        mtError, [mbOK], 0);
      Exit;
    end;
    // Создание сокета
    FServerSocket := socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if FServerSocket = INVALID_SOCKET then
    begin
      MessageDlg('Ошибка при создании сокета:'#13#10 + GetErrorString,
        mtError, [mbOK], 0);
      Exit;
    end;
    // Привязка сокета к адресу
    if bind(FServerSocket, ServerAddr, SizeOf(ServerAddr)) =
      SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при привязке сокета к адресу:'#13#10 +
        GetErrorString, mtError, [mbOK], 0);
      closesocket(FServerSocket);
      Exit;
    end;
    // Перевод сокета в режим прослушивания
    if listen(FServerSocket, SOMAXCONN) = SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при переводе сокета в режим прослушивания:' +
        #13#10 + GetErrorString, mtError, [mbOK], 0);
      closesocket(FServerSocket);
      Exit;
    end;
    // Связь слушающего сокета с событием FD_ACCEPT
    if WSAAsyncSelect(FServerSocket, Handle,
      WM_ACCEPTMESSAGE, FD_ACCEPT) = SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при установке асинхронного режима ' +
        'слушающего сокета:'#13#10 + GetErrorString, mtError, [mbOK], 0);
```

```
      closesocket(FServerSocket);
      Exit;
    end;
    // Перевод элементов управления в состояние "Сервер работает"
    LabelPortNumber.Enabled := False;
    EditPortNumber.Enabled := False;
    BtnStartServer.Enabled := False;
    LabelServerState.Caption := 'Сервер работает';
  except
    on EConvertError do
      // Это исключение может возникнуть только в одном месте -
      // при вызове StrToInt(EditPortNumber.Text)
      MessageDlg('"' + EditPortNumber.Text +
        '" не является целым числом', mtError, [mbOK], 0);
    on ERangeError do
      // Это исключение может возникнуть только в одном месте -
      // при присваивании значения номеру порта
      MessageDlg('Номер порта должен находиться в диапазоне 1-65535',
        mtError, [mbOK], 0);
  end;
end;
```

Этот код мало чем отличается от того, что мы уже видели (сравните, например, с листингами 2.19 и 2.30). Единственное существенное отличие здесь — вызов функции `WSAAsyncSelect` после перевода сокета в режим прослушивания. Этот вызов связывает событие `FD_ACCEPT` с сообщением `WM_ACCEPTMESSAGE`.

Сообщение `WM_ACCEPTMESSAGE` нестандартное, мы должны сами определить его. Использовать это сообщение сервер будет только для определения момента подключения нового клиента, определять момент прихода данных мы будем с помощью другого сообщения — `WM_SOCKETMESSAGE`, которое тоже нужно определить. И, чтобы легче было писать обработчики для этих сообщений, объявим тип `TWMSocketMessage`, "совместимый" с типом `TMessage` (листинг 2.51).

Листинг 2.51. Сообщения, связанные с сокетами, и тип `TWMSocketMessage`

```
const
  WM_ACCEPTMESSAGE = WM_USER + 1;
  WM_SOCKETMESSAGE = WM_USER + 2;

type
  TWMSocketMessage = packed record
```

```
    Msg: Cardinal;
    Socket: TSocket;
    SockEvent: Word;
    SockError: Word;
  end;
```

Прежде чем реализовывать реакцию на эти сообщения, нужно позаботиться об обработке ошибок. Функция GetErrorString (см. листинг 2.6), столько времени служившая нам верой и правдой, нуждается в некоторых изменениях. Это связано с тем, что теперь код ошибки может быть получен не только в результате вызова функции WSAGetLastError, но и через параметр SockError сообщения. Новый вариант функции GetErrorString иллюстрирует листинг 2.52.

Листинг 2.52. Новый вариант функции GetErrorString

```
// Функция GetErrorString возвращает сообщение об ошибке,
// сформированное системой на основе значения, которое
// передано в качестве параметра. Если это значение
// равно нулю (по умолчанию), функция сама определяет
// код ошибки, используя функцию WSAGetLastError.
// Для получения сообщения используется системная функция
// FormatMessage.
function GetErrorString(Error: Integer = 0): string;
var
  Buffer: array[0..2047] of Char;
begin
  if Error = 0 then
    Error := WSAGetLastError;
  FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, nil, Error, $400,
    @Buffer, SizeOf(Buffer), nil);
  Result := Buffer;
end;
```

Сам обработчик сообщения WM_ACCEPTMESSAGE приведен в листинге 2.53.

Листинг 2.53. Обработчик сообщения WM_ACCEPTMESSAGE

```
procedure TServerForm.WMAcceptMessage(var Msg: TWMSocketMessage);
var
  NewConnection: PConnection;
  // Сокет, который создается для вновь подключившегося клиента
  ClientSocket: TSocket;
```

```
  // Адрес подключившегося клиента
  ClientAddr: TSockAddr;
  // Длина адреса
  AddrLen: Integer;
begin
  // Страхуемся от "тупой" ошибки
  if Msg.Socket <> FServerSocket then
    raise ESocketError.Create(
      'Внутренняя ошибка сервера — неверный серверный сокет');
  // Обрабатываем ошибку на сокете, если она есть.
  if Msg.SockError <> 0 then
  begin
    MessageDlg('Ошибка при подключении клиента:#13#10' +
      GetErrorString(Msg.SockError) +
      #13#10'Сервер будет остановлен', mtError, [mbOK], 0);
    ClearConnections;
    closesocket(FServerSocket);
    OnStopServer;
    Exit;
  end;
  // Страхуемся от еще одной "тупой" ошибки
  if Msg.SockEvent <> FD_ACCEPT then
    raise ESocketError.Create(
      'Внутренняя ошибка сервера — неверное событие на сокете');
  AddrLen := SizeOf(TSockAddr);
  ClientSocket := accept(FServerSocket, @ClientAddr, @AddrLen);
  if ClientSocket = INVALID_SOCKET then
  begin
    // Если произошедшая ошибка — WSAEWOULDBLOCK, это просто означает,
    // что на данный момент подключений нет, а вообще все в порядке,
    // поэтому ошибку WSAEWOULDBLOCK мы просто игнорируем. Прочие же
    // ошибки могут произойти только в случае серьезных проблем,
    // которые требуют остановки сервера.
    if WSAGetLastError <> WSAEWOULDBLOCK then
    begin
      MessageDlg('Ошибка при подключении клиента:#13#10' +
        GetErrorString + #13#10'Сервер будет остановлен',
        mtError, [mbOK], 0);
      ClearConnections;
      closesocket(FServerSocket);
      OnStopServer;
    end;
  end
```

```
  else
  begin
    // Связываем сообщение с новым сокетом
    if WSAAsyncSelect(ClientSocket, Handle, WM_SOCKETMESSAGE,
      FD_READ or FD_WRITE or FD_CLOSE) = SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при установке асинхронного режима ' +
        'подключившегося сокета:'#13#10 +
        GetErrorString, mtError, [mbOK], 0);
      closesocket(ClientSocket);
      Exit;
    end;
    // Создаем запись для нового подключения и заполняем ее
    New(NewConnection);
    NewConnection.ClientSocket := ClientSocket;
    NewConnection.ClientAddr := Format('%u.%u.%u.%u:%u', [
          Ord(ClientAddr.sin_addr.S_un_b.s_b1),
          Ord(ClientAddr.sin_addr.S_un_b.s_b2),
          Ord(ClientAddr.sin_addr.S_un_b.s_b3),
          Ord(ClientAddr.sin_addr.S_un_b.s_b4),
          ntohs(ClientAddr.sin_port)]);
    NewConnection.Phase := tpReceiveLength;
    NewConnection.Offset := 0;
    NewConnection.BytesLeft := SizeOf(Integer);
    NewConnection.SendRead := False;
    // Добавляем запись нового соединения в список
    FConnections.Add(NewConnection);
    AddMessageToLog('Зафиксировано подключение с адреса ' +
      NewConnection.ClientAddr);
  end;
end;
```

Для каждого подключившегося клиента создается запись типа `TConnection`, указатель на которую добавляется в список `FConnections` — здесь полная аналогия с сервером на неблокирующих сокетах. Отличие заключается в том, что в типе `TConnection` по сравнению с тем сервером (см. листинг 2.31) добавилось поле `SendRead` логического типа. Оно равно `True`, если возникло событие `FD_READ` в то время, как сервер находится на этапе отправки данных.

Каждый сокет, созданный функцией `accept`, связывается с сообщением `WM_SOCKETMESSAGE`. Обработчик этого сообщения приведен в листинге 2.54.

Листинг 2.54. Обработчик сообщения WM_SOCKETMESSAGE

```
// Метод GetConnectionBySocket находит в списке FConnections
// запись, соответствующую данному сокету
function TServerForm.GetConnectionBySocket(S: TSocket): PConnection;
var
  I: Integer;
begin
  for I := 0 to FConnections.Count – 1 do
    if PConnection(FConnections[I]).ClientSocket = S then
    begin
      Result := FConnections[I];
      Exit;
    end;
  Result := nil;
end;

procedure TServerForm.WMSocketMessage(var Msg: TWMSocketMessage);
var
  Connection: PConnection;
  Res: Integer;

  // Вспомогательная процедура, освобождающая ресурсы, связанные
  // с клиентом и удаляющая запись подключения из списка
  procedure RemoveConnection;
  begin
    closesocket(Connection.ClientSocket);
    FConnections.Remove(Connection);
    Dispose(Connection);
  end;

begin
  // Ищем соединение по сокету
  Connection := GetConnectionBySocket(Msg.Socket);
  if Connection = nil then
  begin
    AddMessageToLog(
      'Внутренняя ошибка сервера – не найдено соединение для сокета');
    Exit;
  end;
  // Проверяем, были ли ошибки при взаимодействии
  if Msg.SockError <> 0 then
  begin
```

```
    AddMessageToLog('Ошибка при взаимодействии с клиентом ' +
      Connection.ClientAddr + ': ' + GetErrorString(Msg.SockError));
    RemoveConnection;
    Exit;
  end;
  // Анализируем, какое событие произошло
  case Msg.SockEvent of
    FD_READ:
      begin
        // Проверяем, на каком этапе находится взаимодействие с клиентом.
        if Connection.Phase = tpReceiveLength then
        begin
          // Этап получения от клиента длины строки. При выполнении этого
          // этапа сервер получает от клиента длину строки и размещает ее
          // в поле Connection.MsgSize. Здесь приходится учитывать, что
          // теоретически даже такая маленькая (4 байта) посылка может
          // быть разбита на несколько пакетов, поэтому за один раз этот
          // этап не будет завершен, и второй раз его придется
          // продолжать, загружая оставшиеся байты. Connection.Offset –
          // количество уже прочитанных на данном этапе байтов –
          // одновременно является смещением, начиная с которого
          // заполняется буфер.
          Res := recv(Connection.ClientSocket,
            (PChar(@Connection.MsgSize) + Connection.Offset)^,
            Connection.BytesLeft, 0);
          if Res > 0 then
          begin
            // Если Res > 0, это означает, что получено Res байтов.
            // Соответственно, увеличиваем на Res количество прочитанных
            // на данном этапе байтов и на такую же величину уменьшаем
            // количество оставшихся.
            Inc(Connection.Offset, Res);
            Dec(Connection.BytesLeft, Res);
            // Если количество оставшихся байтов равно нулю, можно
            // переходить к следующему этапу.
            if Connection.BytesLeft = 0 then
            begin
              // Проверяем корректность принятой длины строки
              if Connection.MsgSize <= 0 then
              begin
                AddMessageToLog('Неверная длина строки от клиента ' +
                  Connection.ClientAddr + ': ' +
                  IntToStr(Connection.MsgSize));
```

```
          RemoveConnection;
          Exit;
        end;
        // Следующий этап — это чтение самой строки
        Connection.Phase := tpReceiveString;
        // Пока на этом этапе не прочитано ни одного байта
        Connection.Offset := 0;
        // Осталось прочитать Connection.MsgSize байтов
        Connection.BytesLeft := Connection.MsgSize;
        // Сразу выделяем память под строку
        SetLength(Connection.Msg, Connection.MsgSize);
      end;
    end
    else if Res = 0 then
    begin
      AddMessageToLog('Клиент ' + Connection.ClientAddr +
        ' закрыл соединение');
      RemoveConnection;
      Exit;
    end
    else
      // Ошибку WSAEWOULDBLOCK игнорируем, т. к. она говорит
      // только о том, что входной буфер сокета пуст, но в целом
      // все в порядке — такое вполне возможно при ложных
      // срабатываниях сообщения
      if WSAGetLastError <> WSAEWOULDBLOCK then
      begin
        AddMessageToLog('Ошибка при получении данных от клиента ' +
          Connection.ClientAddr + ': ' + GetErrorString);
        RemoveConnection;
        Exit;
      end;
  end
  else if Connection.Phase = tpReceiveString then
  begin
    // Следующий этап — чтение строки. Он практически не отличается
    // по реализации от этапа чтения длины строки, за исключением
    // того, что теперь буфером, куда помещаются полученные от
    // клиента данные, служит не Connection.MsgSize,
    // а Connection.Msg.
    Res := recv(Connection.ClientSocket,
      Connection.Msg[Connection.Offset + 1],
      Connection.BytesLeft, 0);
```

```
if Res > 0 then
begin
  Inc(Connection.Offset, Res);
  Dec(Connection.BytesLeft, Res);
  // Если количество оставшихся байтов равно нулю, можно
  // переходить к следующему этапу.
  if Connection.BytesLeft = 0 then
  begin
    AddMessageToLog('От клиента ' + Connection.ClientAddr +
      ' получена строка: ' + Connection.Msg);
    // Преобразуем строку. В отличие от предыдущих примеров,
    // здесь мы явно добавляем к строке #0. Это связано с тем,
    // что при отправке, которая тоже может быть выполнена не
    // за один раз, мы указываем индекс того символа строки,
    // начиная с которого нужно отправлять данные. И (хотя
    // теоретически вероятность этого очень мала) может
    // возникнуть ситуация, когда за один раз будут отправлены
    // все символы строки, кроме завершающего #0, и тогда при
    // следующей отправке начинать придется с него. Если мы
    // будем использовать тот #0, который добавляется к концу
    // строки автоматически, то в этом случае индекс выйдет за
    // пределы диапазона. Поэтому мы вручную добавляем еще один
    // #0 к строке, чтобы он стал законной ее частью.
    Connection.Msg :=
      AnsiUpperCase(StringReplace(Connection.Msg, #0, '#0',
      [rfReplaceAll])) + ' (AsyncSelect server)'#0;
    // Следующий этап — отправка строки клиенту
    Connection.Phase := tpSendString;
    // Отправлено на этом этапе 0 байт
    Connection.Offset := 0;
    // Осталось отправить Length(Connection.Msg) байтов.
    // Единицу к длине строки, в отличие от предыдущих
    // примеров, не добавляем, т. к. там эта единица нужна была
    // для того, чтобы учесть добавляемый к строке
    // автоматически символ #0. Здесь мы еще один #0 добавили
    // к строке явно, поэтому он уже учтен в функции Length.
    Connection.BytesLeft := Length(Connection.Msg);
    // Ставим в очередь сообщение с событием FW_WRITE.
    // Его получение заставит сервер отправить данные
    PostMessage(Handle, WM_SOCKETMESSAGE, Msg.Socket,
      FD_WRITE);
  end;
end
```

```
        else if Res = 0 then
        begin
          AddMessageToLog('Клиент ' + Connection.ClientAddr +
            ' закрыл соединение');
          RemoveConnection;
          Exit;
        end
        else
          // Как обычно, "ошибку" WSAEWOULDBLOCK просто игнорируем
          if WSAGetLastError <> WSAEWOULDBLOCK then
          begin
            AddMessageToLog('Ошибка при получении данных от клиента ' +
              Connection.ClientAddr + ': ' + GetErrorString);
            RemoveConnection;
            Exit;
          end;
      end
      else if Connection.Phase = tpSendString then
        // Если сервер находится на этапе отправки данных,
        // а событие FD_READ все же произошло, отмечаем это
        Connection.SendRead := True;
    end;
  FD_WRITE:
    begin
      if Connection.Phase = tpSendString then
      begin
        // При наступлении события FD_WRITE проверяем, находится ли
        // сервер на этапе отправки данных, и если да, отправляем их
        Res := send(Connection.ClientSocket,
          Connection.Msg[Connection.Offset + 1],
          Connection.BytesLeft, 0);
        if Res > 0 then
        begin
          Inc(Connection.Offset, Res);
          Dec(Connection.BytesLeft, Res);
          // Если Connections.BytesLeft = 0, значит, строка отправлена
          // полностью.
          if Connection.BytesLeft = 0 then
          begin
            AddMessageToLog('Клиенту ' + Connection.ClientAddr +
              ' отправлена строка: ' + Connection.Msg);
            // Очищаем строку, просто чтобы сэкономить память
            Connection.Msg := '';
```

```
            // Следующий этап — снова получение длины строки от клиента
            Connection.Phase := tpReceiveLength;
            // Получено — 0 байт
            Connection.Offset := 0;
            // Осталось прочитать столько, сколько занимает целое число
            Connection.BytesLeft := SizeOf(Integer);
            // Если были промежуточные события FD_READ, вызываем их
            // снова искусственно
            if Connection.SendRead then
            begin
              PostMessage(Handle, WM_SOCKETMESSAGE, Msg.Socket,
                FD_READ);
              Connection.SendRead := False;
            end;
          end;
        end
        else
          if WSAGetLastError <> WSAEWOULDBLOCK then
          begin
            AddMessageToLog('Ошибка при отправке данных клиенту ' +
              Connection.ClientAddr + ': ' + GetErrorString);
            RemoveConnection;
            Exit;
          end;
      end;
    end;
  FD_CLOSE:
    begin
      // Клиент вызвал функцию shutdown. Закрываем соединение.
      AddMessageToLog('Клиент ' + Connection.ClientAddr +
        ' закрыл соединение');
      shutdown(Connection.ClientSocket, SD_BOTH);
      RemoveConnection;
    end;
  else
    begin
      AddMessageToLog('Неверное событие при обмене с клиентом ' +
        Connection.ClientAddr);
      RemoveConnection;
    end;
  end;
end;
```

В этом примере можно найти много общего с кодом из листинга 2.32 — получение и отправка данных в зависимости от этапа выполняется практически одинаково, различаются только условия, при которых эти участки кода выполняются. Обратите внимание, что теперь проверка того, какой этап чтения выполняется, сделана взаимоисключающей, т. е. при обработке одного сообщения не может быть прочитана и длина строки, и сама строка. Это сделано, чтобы убрать ложные срабатывания. Рассмотрим два возможных варианта. Первый вариант — когда во входном буфере сокета оказывается сразу длина и строка (или ее часть). После того как будет прочитана длина, сообщение `WM_SOCKETMESSAGE` с параметром `FD_READ` вновь будет помещено в очередь, поскольку функция `recv` помещает это сообщение в очередь, если после ее вызова во входном буфере сокета остались данные. Если мы немедленно перейдем ко второму этапу, то прочитаем из буфера сокета все оставшиеся там данные, но сообщение в очереди все равно останется, что даст нам ложное срабатывание, когда петля сообщений извлечет и диспетчеризует это сообщение. Таким образом, выполнение сразу двух этапов при обработке одного сообщения не даст выигрыша в производительности, т. к. все равно придется извлекать и обрабатывать два сообщения.

Второй вариант — когда на момент обработки события `FD_READ` во входном буфере находится только длина строки. В этом случае функция `recv` не будет помещать в очередь второе сообщение `WM_SOCKETMESSAGE`, т. к. данных в буфере после ее выполнения не останется, но и попытка выполнить этап чтения строки окажется бесполезной работой, т. к. строка еще не получена. В любом случае этап чтения строки будет выполнен только при обработке следующего сообщения `WM_SOCKETMESSAGE`, когда от клиента будут получены новые данные.

Получается, что при обоих вариантах попытка выполнить за один раз сразу два этапа не дает никаких преимуществ в быстродействии, но зато повышает вероятность ложных срабатываний события `FD_READ` в то время, когда сервер находится на этапе отправки данных клиенту. А ложные срабатывания на этом этапе вредны тем, что сервер принимает их за поступление данных, которое нужно запомнить, чтобы обработать после того, как ответ будет отправлен клиенту. В принципе, эти ложные срабатывания в итоге не приводят ни к чему плохому, кроме незначительного увеличения нагрузки на процессор, но раз от них нет пользы, и мы можем избавиться от них совсем небольшой ценой, лучше это сделать.

Отправка данных клиенту выполняется при обработке события `FD_WRITE`. Это событие генерируется библиотекой сокета в двух случаях: при начале работы сокета и когда возможность отправки данных восстановлена после отказа из-за нехватки места в буфере. Пока речь не идет об обмене сообщениями размером в десятки мегабайтов, ситуация с нехваткой места в выходном буфере

сокета крайне маловероятна, т. е. библиотека сокетов будет генерировать это событие лишь один раз для каждого клиента. Но никто не мешает нам помещать соответствующее сообщение в очередь вручную, что мы и делаем при обработке события `FD_READ` после завершения этапа получения строки, т. е. когда сервер согласно протоколу должен отправить ответ. Таким образом, один и тот же участок кода используется для отправки данных как тогда, когда сервер видит в этом необходимость, так и тогда, когда их вновь можно отправлять после переполнения буфера.

При обработке события `FD_WRITE` в очередь сообщений также помещается сообщение `WM_SOCKETMESSAGE`, если было зафиксировано получение события `FD_READ` на этапе отправки данных. В принципе, это может дать ложное срабатывание `FD_READ` в двух случаях: когда исходное событие `FD_READ` было ложным и когда событие `FD_READ` уже присутствует в очереди на момент вызова `PostMessage`. Но, как мы уже отметили ранее, никаких неприятных последствий, кроме незначительного увеличения нагрузки на процессор, ложные срабатывания не приносят, так что с ними можно смириться.

В итоге у нас получился сервер, который, как и сервер на неблокирующих сокетах, никогда не блокируется и устойчив к нарушению клиентом протокола. По сравнению с сервером на неблокирующих сокетах сервер на асинхронных событиях имеет два преимущества. Во-первых, немного снижена нагрузка на процессор, т. к. попытка чтения данных из сокета выполняется не периодически, а только когда это необходимо. Во-вторых, сообщения клиента обрабатываются несколько быстрее, т. к. сообщение помещается в очередь сразу при получении данных, и, если сервер не занят ничем другим, он сразу приступает к его обработке, а не ждет, пока истечет период опроса.

2.2.7. Асинхронный режим, основанный на событиях

Асинхронный режим, основанный на событиях, появился во второй версии Windows Sockets. В его основе лежат события — специальные объекты, служащие для синхронизации работы нитей.

Существуют события, поддерживаемые на уровне системы. Они создаются с помощью функции `CreateEvent`. Каждое событие может находиться в сброшенном или взведенном состоянии. Нить с помощью функций `WaitForSingleObject` и `WaitForMultipleObjects` может дожидаться, пока одно или несколько событий не окажутся во взведенном состоянии. В режиме ожидания нить не требует процессорного времени. Другая нить может установить событие с помощью функции `SetEvent`, в результате чего первая нить выйдет из состояния ожидания и продолжит свою работу. Подробно о системных событиях и прочих объектах синхронизации написано в [2].

Аналогичные объекты определены и в Windows Sockets. Сокетные события отличаются от стандартных системных событий прежде всего тем, что они могут быть связаны с событиями FD_XXX, происходящими на сокете, и взводиться при наступлении этих событий.

Так как сокетные события поддерживаются только в WinSock 2, модуль WinSock не содержит объявлений типов и функций, требуемых для их поддержки. Поэтому их придется объявлять самостоятельно. Прежде всего, должен быть объявлен тип дескриптора событий, который в MSDN называется WSAEVENT. В Delphi он может быть объявлен следующим образом:

```
PWSAEvent = ^TWSAEvent;
TWSAEvent = THandle;
```

Событие создается с помощью функции WSACreateEvent, прототип которой приведен в листинге 2.55.

Листинг 2.55. Функция WSACreateEvent

```
// ***** Описание на C++ *****
WSAEVENT WSACreateEvent(void);

// ***** Описание на Delphi *****
function WSACreateEvent: TWSAEvent;
```

Событие, созданное этой функцией, находится в сброшенном состоянии, при ожидании автоматически не сбрасывается, не имеет имени и обладает стандартными атрибутами безопасности. В MSDN отмечено, что сокетное событие на самом деле является простым системным событием, и его можно создавать с помощью стандартной функции CreateEvent, управляя значениями всех перечисленных параметров.

Функция создает событие и возвращает его дескриптор. Если произошла ошибка, функция возвращает значение WSA_INVALID_EVENT (0).

Для ручного взведения и сброса события предназначены функции WSASetEvent и WSAResetEvent соответственно, прототипы которых приведены в листинге 2.56.

Листинг 2.56. Функции для управления событиями

```
// ***** Описание на C++ *****
BOOL WSASetEvent(WSAEVENT hEvent);
BOOL WSAResetEvent(WSAEVENT hEvent);
```

```
// ***** Описание на Delphi *****
function WSASetEvent(hEvent: TWSAEvent): BOOL;
function WSAResetEvent(hEvent: TWSAEvent): BOOL;
```

Функции возвращают True, если операция прошла успешно, и False — в противном случае.

После завершения работы с событием оно уничтожается с помощью функции WSACloseEvent (листинг 2.57).

Листинг 2.57. Функция WSACloseEvent

```
// ***** Описание на C++ *****
BOOL WSACloseEvent(WSAEVENT hEvent);

// ***** Описание на Delphi *****
function WSACloseEvent(hEvent: TWSAEvent): BOOL;
```

Функция уничтожает событие и освобождает связанные с ним ресурсы. Дескриптор, переданный в качестве параметра, становится недействительным.

Для ожидания взведения событий служит функция WSAWaitForMultipleEvents (листинг 2.58).

Листинг 2.58. Функция WSAWaitForMultipleEvents

```
// ***** Описание на C++ *****
DWORD WSAWaitForMultipleEvents(
  DWORD cEvents,
  const WSAEVENT FAR *lphEvents,
  BOOL fWaitAll,
  DWORD dwTimeout,
  BOOL fAlertable);

// ***** Описание на Delphi *****
function WSAWaitForMultipleEvents(
  cEvents: DWORD;
  lphEvents: PWSAEvent;
  fWaitAll: BOOL;
  dwTimeout: DWORD;
  fAlertable: BOOL): DWORD;
```

Дескрипторы событий, взведения которых ожидает нить, должны храниться в массиве, размер которого передается через параметр cEvents, а указатель —

через параметр lphEvents. Параметр fWaitAll определяет, что является условием окончания ожидания: если он равен True, ожидание завершается, когда все события из переданного массива оказываются во взведенном состоянии, если False — когда оказывается взведенным хотя бы одно из них. Параметр dwTimeout определяет тайм-аут ожидания в миллисекундах. В WinSock 2 определена константа WSA_INFINITE (совпадающая по значению со стандартной константой INFINITE), которая задает бесконечное ожидание. Параметр fAlertable нужен при перекрытом вводе-выводе; мы рассмотрим его позже в *разд. 2.2.9*. Если перекрытый ввод-вывод не используется, fAlertable должен быть равен False.

Существует ограничение на число событий, которое можно ожидать с помощью данной функции. Максимальное число событий определяется константой WSA_MAXIMIUM_WAIT_EVENTS, которая в данной реализации равна 64.

Результат, возвращаемый функцией, позволяет определить, по каким причинам закончилось ожидание. Если ожидалось взведение всех событий (fWaitAll = True), и оно произошло, функция возвращает WSA_WAIT_EVENT_0 (0). Если ожидалось взведение хотя бы одного из событий, возвращается WSA_WAIT_EVENT_0 + Index, где Index — индекс взведенного события в массиве lphEvents (отсчет индексов начинается с нуля). Если ожидание завершилось по тайм-ауту, возвращается значение WSA_WAIT_TIMEOUT (258). И наконец, если произошла какая-либо ошибка, функция возвращает WSA_WAIT_FAILED ($FFFFFFFF).

Существует еще одно значение, которое может возвратить функция WSAWaitForMultipleEvents: WAIT_IO_COMPLETION (это константа из стандартной части Windows API, она объявлена в модуле Windows). Смысл этого результата и условия, при которых он может быть возвращен, мы рассмотрим в *разд. 2.2.9*.

Функции, которые мы рассматривали до сих пор, являются аналогами системных функций для стандартных событий. Теперь мы переходим к рассмотрению тех функций, которые отличают сокетные события от стандартных. Главная из них — WSAEventSelect, позволяющая привязать события, создаваемые с помощью WSACreateEvent, к тем событиям, которые происходят на сокете. Прототип этой функции приведен в листинге 2.59.

Листинг 2.59. Функция WSAEventSelect

```
// ***** Описание на C++ *****
int WSAEventSelect(SOCKET s, WSAEVENT hEventObject,
  long lNetworkEvents);
```

```
// ***** Описание на Delphi *****
function WSAEventSelect(S: TSocket; hEventObject: TWSAEvent;
  lNetworkEvents: LongInt): Integer;
```

Эта функция очень похожа на функцию WSAAsyncSelect, за исключением того, что события FD_XXX привязываются не к оконным сообщениям, а к сокетным событиям. Параметр S определяет сокет, события которого отслеживаются, параметр hEventObject — событие, которое должно взводиться при наступлении отслеживаемых событий, lNetworkEvents — комбинация констант FD_XXX, определяющая, с какими событиями на сокете связывается событие hSocketEvent.

Функция WSAEventSelect возвращает ноль, если операция прошла успешно, и SOCKET_ERROR при возникновении ошибки.

Событие, связанное с сокетом функцией WSAEventSelect, взводится при тех же условиях, при которых в очередь окна помещается сообщение при использовании WSAAsyncSelect. Так, например, функция recv взводит событие, если после ее вызова в буфере сокета еще остаются данные. Но, с другой стороны, функция recv не сбрасывает событие, если данных в буфере сокета нет. А поскольку сокетные события не сбрасываются автоматически функцией WSAWaitForMultipleEvents, программа всегда должна сбрасывать события сама. Так, при обработке FD_READ наиболее типична ситуация, когда сначала сбрасывается событие, а потом вызывается функция recv, которая при необходимости снова взводит событие. Здесь мы снова имеем проблему ложных срабатываний в тех случаях, когда данные извлекаются из буфера по частям с помощью нескольких вызовов recv, но в данном случае проблему решить легче: не нужно отменять регистрацию событий, достаточно просто сбросить событие непосредственно перед последним вызовом recv.

В принципе, события FD_XXX разных сокетов можно привязать к одному сокетному событию, но этой возможностью обычно не пользуются, т. к. в WinSock 2 отсутствуют средства, позволяющие определить, событие на каком из сокетов привело к взведению сокетного события. Поэтому приходится для каждого сокета создавать отдельное событие.

Как и в случае с WSAAsyncSelect, при вызове WSAEventSelect сокет переводится в неблокирующий режим. Повторный вызов WSAEventSelect для данного сокета отменяет результаты предыдущего вызова (т. е. невозможно связать разные события FD_XXX одного сокета с разными сокетными событиями). Сокет, созданный в результате вызова accept или WSAAccept, наследует связь с сокетными событиями, установленную для слушающего сокета.

Существует весьма важное различие между использованием оконных сообщений и сокетных событий для оповещения о том, что происходит на сокете.

Предположим, с помощью функции WSAAsyncSelect события FD_READ, FD_WRITE и FD_CONNECT связаны с некоторым оконным сообщением. Пусть происходит событие FD_CONNECT. В очередь окна помещается соответствующее сообщение. Затем, до того, как предыдущее сообщение будет обработано, происходит FD_WRITE. В очередь окна помещается еще одно сообщение, которое информирует об этом. И наконец, при возникновении FD_READ в очередь будет помещено третье сообщение. Затем оконная процедура получит их по очереди и обработает.

Теперь рассмотрим ситуацию, когда те же события связаны с сокетным событием. Когда происходит FD_CONNECT, сокетное событие взводится. Теперь, если FD_WRITE и FD_READ произойдут до того, как сокетное событие будет сброшено, оно уже не изменит своего состояния. Таким образом, программа, работающая с асинхронными сокетами, основанными на событиях, должна, во-первых, учитывать, что взведенное событие может означать несколько событий FD_XXX, а во-вторых, иметь возможность узнать, какие именно события произошли с момента последней проверки. Для получения этой информации предусмотрена функция WSAEnumNetworkEvents, прототип которой приведен в листинге 2.60.

Листинг 2.60. Функция WSAEnumNetworkEvents

```
// ***** Описание на C++ *****
int WSAEnumNetworkEvents(
  SOCKET s,
  WSAEVENT hEventObject,
  LPWSANETWORKEVENTS lpNetworkEvents);

// ***** Описание на Delphi *****
function WSAEnumNetworkEvents(
  S: TSocket;
  hEventObject: TWSAEvent;
  var NetworkEvents: TWSANetworkEvents): Integer;
```

Функция WSAEnumNetworkEvents через параметр NetworkEvents возвращает информацию о том, какие события произошли на сокете S с момента последнего вызова этой функции для данного сокета (или с момента запуска программы, если функция вызывается в первый раз). Параметр hEventObject необязательный, он определяет сокетное событие, которое нужно сбросить. Использование этого параметра позволяет обойтись без явного вызова функции WSAResetEvent для сброса события. Как и большинство функций WinSock, функция WSAEnumNetworkEvents возвращает ноль в случае успеха и ненулевое значение при возникновении ошибки.

Запись TWSANetworkEvents содержит информацию о произошедших событиях и об ошибках (листинг 2.61).

Листинг 2.61. Тип TWSANetworkEvents

```
// ***** Описание на C++ *****
typedef struct _WSANETWORKEVENTS {
  long lNetworkEvents;
  int iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, *LPWSANETWORKEVENTS;

// ***** Описание на Delphi *****
TWSANetworkEvents = packed record
  lNetworkEvents: LongInt;
  iErrorCode: array[0..FD_MAX_EVENTS – 1] of Integer;
end;
```

Константа FD_MAX_EVENTS определяет количество разных типов событий и в данной реализации равна 10.

Значения констант FD_XXX представляют собой степени двойки, поэтому их можно объединять операцией арифметического ИЛИ без потери информации. Поле lNetworkEvents является таким объединением всех констант, задающих события, которые происходили на сокете. Другими словами, если результат операции (lNetworkEvents and FD_XXX) не равен нулю, значит, событие FD_XXX происходило на сокете.

Массив iErrorCode содержит информацию об ошибках, которыми сопровождались события FD_XXX. Для каждого события FD_XXX определена соответствующая константа FD_XXX_BIT (т. е. константы FD_READ_BIT, FD_WRITE_BIT и т. д.). Элемент массива с индексом FD_XXX_BIT содержит информацию об ошибке, связанной с событием FD_XXX. Если операция прошла успешно, этот элемент содержит ноль, в противном случае — код ошибки, которую в аналогичной ситуации вернула бы функция WSAGetLastError после выполнения соответствующей операции на синхронном сокете.

Таким образом, программа, использующая асинхронный режим, основанный на событиях, должна выполнить следующие действия. Во-первых, создать сокет и установить соединение. Во-вторых, привязать события FD_XXX к сокетному событию. В-третьих, организовать цикл, начинающийся с вызова WSAWaitForMultipleEvents, в котором с помощью WSAEnumNetworkEvents определять, какое событие произошло, и обрабатывать его. При возникновении ошибки на сокете цикл должен завершаться.

Сокетные события могут взводиться не только в результате событий на сокете, но и вручную, с помощью функции WSASetEvent. Это дает нити, вызвавшей

функцию WSAWaitForMultipleEvents, возможность выходить из состояния ожидания не только при возникновении событий на сокете, но и по сигналам от других нитей. Типичная область применения этой возможности — для тех случаев, когда программа может как отвечать на запросы от удаленного партнера, так и отправлять ему что-то по собственной инициативе. В этом случае могут использоваться два сокетных события: одно связывается с событием FD_READ для оповещения о поступлении данных, а второе не связывается ни с одним из событий FD_XXX, а устанавливается другой нитью тогда, когда необходимо отправить сообщение. Нить, работающая с сокетом, ожидает взведения одного из этих событий и в зависимости от того, какое из них взведено, читает или отправляет данные.

В листинге 2.62 приведен пример кода такой нити. Она задействует три сокетных события: одно для уведомления о событиях на сокете, второе — для уведомления о необходимости отправить данные, третье — для уведомления о необходимости завершиться. В данном примере мы предполагаем, что, во-первых, сокет создан и подключен до создания нити и передается ей в качестве параметра, а во-вторых, три сокетных события хранятся в глобальном массиве SockEvents: array[0..2] of TWSAEvent, причем нулевой элемент этого массива содержит событие, связываемое с событиями FD_XXX, первый элемент — событие отправки данных, второй — событие завершения нити. Прототип функции, образующей нить, совместим с функцией BeginThread из модуля SysUtils.

Листинг 2.62. Схема нити, использующей события асинхронного сокета

```
function ProcessSockEvents(Parameter: Pointer): Integer;
var
  S: TSocket;
  NetworkEvents: TWSANetworkEvents;
begin
  // Так как типы TSocket и Pointer занимают по 4 байта, такое
  // приведение типов вполне возможно, хотя и некрасиво
  S := TSocket(Parameter);
  // Связываем событие SockEvents[0] с FD_READ и FD_CLOSE
  WSAEventSelect(S, SockEvents[0], FD_READ or FD_CLOSE);
  while True do
  begin
    case WSAWaitForMultipleEvents(3, @SockEvents[0], True,
           WSA_INFINITE, False) of
      WSA_WAIT_EVENT_0:
        begin
          WSAEnumNetworkEvents(S, SockEvents[0], NetworkEvents);
```

```
        if NetworkEvents.lNetworkEvents and FD_READ > 0 then
          if NetworkEvents.iErrorCode[FD_READ_BIT] = 0 then
          begin
            // Пришли данные, которые нужно прочитать
          end
          else
          begin
            // Произошла ошибка. Нужно сообщить о ней и завершить нить
            closesocket(S);
            Exit;
          end;
        if NetworkEvents.lNetworkEvents and FD_CLOSE > 0 then
        begin
          // Связь разорвана
          if NetworkEvents.iErrorCode[FD_CLOSE_BIT] = 0 then
          begin
            // Связь закрыта корректно
          end
          else
          begin
            // Связь разорвана в результате сбоя сети
          end;
          // В любом случае нужно закрыть сокет и завершить нить
          closesocket(S);
          Exit;
        end;
      end;
    WSA_WAIT_EVENT_0 + 1:
      begin
        // Получен сигнал о необходимости отправить данные
        // Здесь должен быть код отправки данных
        // После отправки событие нужно сбросить вручную
        ResetEvent(SockEvents[1]);
      end;
    WSA_WAIT_EVENT_0 + 2:
      begin
        // Получен сигнал о необходимости завершения работы нити
        closesocket;
        ResetEvents(SockEvents[2]);
        Exit;
      end
  end;
 end;
end;
```

Как и во всех предыдущих примерах, здесь для краткости не проверяются результаты, возвращаемые функциями и не отлавливаются возникающие ошибки. Кроме того, отсутствует процедура завершения связи с вызовом `shutdown`.

Данный пример может рассматриваться как фрагмент кода простого сервера. В отдельной нити такого сервера выполняется цикл, состоящий из вызова `accept` и создания новой нити для обслуживания полученного таким образом сокета. Затем другие нити при необходимости могут давать таким нитям команды (необходимо только предусмотреть для каждой нити, обслуживающей сокет, свою копию массива `SockEvents`). Благодаря этому каждый клиент будет обслуживаться независимо.

К недостаткам такого сервера следует отнести его низкую устойчивость против DoS-атак, при которых к серверу подключается очень большое число клиентов. Если сервер будет создавать отдельную нить для обслуживания каждого подключения, количество нитей очень быстро станет слишком большим, и вся система окажется неработоспособной, т. к. бóльшая часть процессорного времени будет тратиться на переключение между нитями. Более защищенным является вариант, при котором сервер заранее создает некоторое разумное количество нитей (пул нитей) и обработку запроса или выполнение команды поручает любой свободной нити из этого пула. Если ни одной свободной нити в пуле нет, задание ставится в очередь. По мере освобождения нитей задания извлекаются из очереди и выполняются. При DoS-атаках такой сервер также не справляется с поступающими заданиями, но это не приводит к краху всей системы. Но сервер с пулом нитей реализуется сложнее (обычно — через порты завершения, которые мы здесь не рассматриваем). Тем не менее простой для реализации сервер без пула нитей тоже может оказаться полезным, если вероятность DoS-атак низка (например, в изолированных технологических подсетях).

Приведенный пример может рассматриваться также как заготовка для клиента. В этом случае целесообразнее передавать в функцию `ProcessSockEvents` не готовый сокет, а только адрес сервера, к которому необходимо подключиться. Создание сокета и установление связи с сервером при этом выполняет сама нить перед началом цикла ожидания событий. Такой подход очень удобен для независимой работы с несколькими однотипными серверами.

2.2.8. Пример использования сокетов с событиями

К достоинствам асинхронного режима, основанного на сообщениях, относится то, что нить, обслуживающая сокет, может выходить из состояния ожидания не только при получении данных сокетом, но и по иным сигналам. Протокол обмена, который мы до сих пор использовали, не позволяет продемон-

стрировать это достоинство в полном объеме. Поэтому, прежде чем создавать пример, мы несколько изменим протокол. Формат пакетов оставим прежним, но изменим условия, при которых эти пакеты могут быть посланы. Теперь клиент имеет право посылать пакеты, не дожидаясь от сервера ответа на предыдущий пакет, а сервер имеет право посылать пакеты клиенту не только в ответ на его запросы, но и по собственной инициативе. В нашей реализации он будет посылать клиентам строку с текущим временем.

Сервер на асинхронных событиях (пример EventSelectServer на компакт-диске) имеет много общего с рассмотренным ранее многонитевым сервером (пример MultithreadedServer, см. *разд. 2.1.12*). В нем также есть нить для обработки подключений клиентов и по одной нити на каждого клиента, а главная нить только создает слушающий сокет и запускает обслуживающую его нить.

Еще одним важным отличием нашего сервера от всех предыдущих примеров серверов станет то, что пользователь сможет его остановить в любой момент. Подобную функциональность было бы несложно добавить и к таким серверам, как SelectServer, NonBlockingServer и AsyncSelectServer, которые работают в одной нити. Но остановить нити в многонитевом сервере можно было только одним способом: уничтожив сокеты из главной нити — в этом случае все работающие с этими сокетами нити завершились бы с ошибками. Очевидно, что это порочный подход, не позволяющий корректно завершить работу с клиентами. Режим с использованием событий позволяет предусмотреть реакцию нити на внешний сигнал об отключении. Отключаться сервер будет по нажатию кнопки **Остановить**.

В листинге 2.63 приведен код нити, взаимодействующей с клиентом (код методов `LogMessage` и `DoLogMessage` опущен, т. к. он идентичен приведенному в листингах 2.20 и 2.7 соответственно).

Листинг 2.63. Нить, взаимодействующая с клиентами

```
unit ClientThread;
{ Нить, обслуживающая одного клиента.
  Выполняет цикл, выход из которого возможен по внешнему сигналу
  или при возникновении ошибки на сокете. Умеет отправлять
  клиенту сообщения по внешнему сигналу.
}

interface

uses
  Windows, Classes, WinSock, Winsock2_Events, ShutdownConst, SysUtils,
  SyncObjs;
```

```
type
  TClientThread = class(TThread)
  private
    // Сообщение, которое нужно добавить в лог,
    // хранится в отдельном поле, т. к. метод, вызывающийся через
    // Synchronize, не может иметь параметров.
    FMessage: string;
    // Префикс для всех сообщений лога, связанных с данным клиентом
    FHeader: string;
    // Сокет для взаимодействия с клиентом
    FSocket: TSocket;
    // События нити
    //  FEvents[0] используется для остановки нити
    //  FEvents[1] используется для отправки сообщения
    //  FEvents[2] связывается с событиями FD_READ, FD_WRITE и FD_CLOSE
    FEvents: array[0..2] of TWSAEvent;
    // Критическая секция для доступа к буферу исходящих
    FSendBufSection: TCriticalSection;
    // Буфер исходящих
    FSendBuf: string;
    // Вспомогательный метод для вызова через Synchronize
    procedure DoLogMessage;
    // Функция, проверяющая, завершила ли нить работу
    function GetFinished: Boolean;
  protected
    procedure Execute; override;
    // Вывод сообщения в лог главной формы
    procedure LogMessage(const Msg: string);
    // Отправка клиенту данных из буфера исходящих
    function DoSendBuf: Boolean;
  public
    constructor Create(ClientSocket: TSocket;
      const ClientAddr: TSockAddr);
    destructor Destroy; override;
    // Добавление строки в буфер исходящих
    procedure SendString(const S: string);
    // Остановка нити извне
    procedure StopThread;
    property Finished: Boolean read GetFinished;
  end;

  ESocketError = class(Exception);

implementation
```

```
uses
  MainServerUnit;

{ TClientThread }

// Сокет для взаимодействия с клиентом создается в главной нити,
// а сюда передается через параметр конструктора. Для формирования
// заголовка сюда же передается адрес подключившегося клиента
constructor TClientThread.Create(ClientSocket: TSocket;
  const ClientAddr: TSockAddr);
begin
  FSocket := ClientSocket;
  // Заголовок содержит адрес и номер порта клиента.
  // Этот заголовок будет добавляться ко всем сообщениям в лог
  // от данного клиента.
  FHeader := 'Сообщение от клиента ' + inet_ntoa(ClientAddr.sin_addr) +
    ':' + IntToStr(ntohs(ClientAddr.sin_port)) + ': ';
  // Создаем события и привязываем первое из них к сокету
  FEvents[0] := WSACreateEvent;
  if FEvents[0] = WSA_INVALID_EVENT then
    raise ESocketError.Create(FHeader + 'Ошибка при создании события: ' +
      GetErrorString);
  FEvents[1] := WSACreateEvent;
  if FEvents[1] = WSA_INVALID_EVENT then
    raise ESocketError.Create(FHeader + 'Ошибка при создании события: ' +
      GetErrorString);
  FEvents[2] := WSACreateEvent;
  if FEvents[2] = WSA_INVALID_EVENT then
    raise ESocketError.Create(FHeader + 'Ошибка при создании события: ' +
      GetErrorString);
  if WSAEventSelect(FSocket, FEvents[2], FD_READ or FD_WRITE or FD_CLOSE)
    = SOCKET_ERROR then
    raise ESocketError.Create(FHeader +
      'Ошибка при привязывании сокета к событию: ' + GetErrorString);
  FSendBufSection := TCriticalSection.Create;
  // Объект этой нити не должен удаляться сам
  FreeOnTerminate := False;
  inherited Create(False);
end;

destructor TClientThread.Destroy;
begin
  FSendBufSection.Free;
```

```
  WSACloseEvent(FEvents[0]);
  WSACloseEvent(FEvents[1]);
  WSACloseEvent(FEvents[2]);
  inherited;
end;

// Функция добавляет строку в буфер для отправки
procedure TClientThread.SendString(const S: string);
begin
  FSendBufSection.Enter;
  try
    FSendBuf := FSendBuf + S + #0;
  finally
    FSendBufSection.Leave;
  end;
  LogMessage('Сообщение "' + S + '" поставлено в очередь для отправки');
  // Взводим событие, которое говорит, что нужно отправлять данные
  WSASetEvent(FEvents[1]);
end;

// Отправка всех данных, накопленных в буфере
// Функция возвращает False, если произошла ошибка,
// и True, если все в порядке
function TClientThread.DoSendBuf: Boolean;
var
  SendRes: Integer;
begin
  FSendBufSection.Enter;
  try
    // Если отправлять нечего, выходим
    if FSendBuf = '' then
    begin
      Result := True;
      Exit;
    end;
    // Пытаемся отправить все, что есть в буфере
    SendRes := send(FSocket, FSendBuf[1], Length(FSendBuf), 0);
    if SendRes > 0 then
    begin
      // Удаляем из буфера ту часть, которая отправилась клиенту
      Delete(FSendBuf, 1, SendRes);
      Result := True;
    end
```

```
    else
    begin
      Result := WSAGetLastError = WSAEWOULDBLOCK;
      if not Result then
        LogMessage('Ошибка при отправке данных: ' + GetErrorString);
    end;
  finally
    FSendBufSection.Leave;
  end;
end;

procedure TClientThread.Execute;
const
  // Размер буфера для приема сообщений
  RecvBufSize = 4096;
var
  // Буфер для приема сообщений
  RecvBuf: array[0..RecvBufSize – 1] of Byte;
  RecvRes: Integer;
  NetEvents: TWSANetworkEvents;
  // Полученная строка
  Str: string;
  // Длина полученной строки
  StrLen: Integer;
  // Если ReadLength = True, идет чтение длины строки,
  // если False – самой строки
  ReadLength: Boolean;
  // Смещение от начала приемника
  Offset: Integer;
  // Число байтов, оставшихся при получении длины строки или самой строки
  BytesLeft: Integer;
  P: Integer;
  L: Integer;
  LoopExit: Boolean;
  WaitRes: Cardinal;
begin
  LogMessage('Соединение установлено');
  ReadLength := True;
  Offset := 0;
  BytesLeft := SizeOf(Integer);
  repeat
    WaitRes :=
      WSAWaitForMultipleEvents(3, @FEvents, False, WSA_INFINITE, False);
```

```
case  WaitRes of
  WSA_WAIT_EVENT_0:
    begin
      // Закрываем соединение с клиентом и останавливаем нить
      LogMessage('Получен сигнал об остановке нити');
      shutdown(FSocket, SD_BOTH);
      Break;
    end;
  WSA_WAIT_EVENT_0 + 1:
    begin
      // Сбрасываем событие и отправляем данные
      WSAResetEvent(FEvents[1]);
      if not DoSendBuf then
        Break;
    end;
  WSA_WAIT_EVENT_0 + 2:
    begin
      // Произошло событие, связанное с сокетом.
      // Проверяем, какое именно, и заодно сбрасываем его
      if WSAEnumNetworkEvents(FSocket, FEvents[2], NetEvents) =
        SOCKET_ERROR then
      begin
        LogMessage('Ошибка при получении списка событий: ' +
          GetErrorString);
        Break;
      end;
      if NetEvents.lNetworkEvents and FD_READ <> 0 then
      begin
        if NetEvents.iErrorCode[FD_READ_BIT] <> 0 then
        begin
          LogMessage('Ошибка в событии FD_READ: ' +
            GetErrorString(NetEvents.iErrorCode[FD_READ_BIT]));
          Break;
        end;
        // В буфере сокета есть данные.
        // Копируем данные из буфера сокета в свой буфер RecvBuf
        RecvRes := recv(FSocket, RecvBuf, SizeOf(RecvBuf), 0);
        if RecvRes > 0 then
        begin
          P := 0;
          // Эта переменная нужна потому, что здесь появляется
          // вложенный цикл, при возникновении ошибки в котором нужно
          // выйти и из внешнего цикла тоже. Так как в Delphi нет
```

```
// конструкции типа Break(2) в Аде, приходится прибегать
// к таким способам: если нужен выход из внешнего цикла,
// во внутреннем цикле выполняется LoopExit := True,
// а после выполнения внутреннего цикла проверяется
// значение этой переменной и при необходимости выполняется
// выход и из главного цикла.
LoopExit := False;
// В этом цикле мы извлекаем данные из буфера
// и раскидываем их по приемникам — Str и StrLen
while P < RecvRes do
begin
  // Определяем, сколько байтов нам хотелось бы скопировать
  L := BytesLeft;
  // Если в буфере нет такого количества,
  // довольствуемся тем, что есть
  if P + L > RecvRes then
    L := RecvRes — P;
  // Копируем в соответствующий приемник
  if ReadLength then
    Move(RecvBuf[P], (PChar(@StrLen) + Offset)^, L)
  else
    Move(RecvBuf[P], Str[Offset + 1], L);
  Dec(BytesLeft, L);
  // Если прочитали все, что хотели,
  // переходим к следующему этапу
  if BytesLeft = 0 then
  begin
    ReadLength := not ReadLength;
    Offset := 0;
    // Если закончено чтение строки, нужно вывести ее в лог
    if ReadLength then
    begin
      LogMessage('Получена строка: ' + Str);
      BytesLeft := SizeOf(Integer);
      // Формируем ответ и записываем его в буфер
      Str := AnsiUpperCase(StringReplace(Str, #0, '#0',
        [rfReplaceAll]))  + ' (AsyncEvent server)';
      SendString(Str);
      Str := '';
    end
    else
    begin
      if StrLen <= 0 then
```

```
            begin
              LogMessage('Неверная длина строки от клиента: ' +
                IntToStr(StrLen));
              LoopExit := True;
              Break;
            end;
            BytesLeft := StrLen;
            SetLength(Str, StrLen);
          end;
        end
        else
          Inc(Offset, L);
        Inc(P, L);
      end;
      // Проверяем, был ли аварийный выход из внутреннего цикла,
      // и если был, выходим и из внешнего, завершая работу
      // с клиентом
      if LoopExit then
        Break;
    end
    else if RecvRes = 0 then
    begin
      LogMessage('Клиент закрыл соединение');
      Break;
    end
    else
    begin
      if WSAGetLastError <> WSAEWOULDBLOCK then
      begin
        LogMessage('Ошибка при получении данных от клиента: ' +
          GetErrorString);
      end;
    end;
  end;
  // Сокет готов к передаче данных
  if NetEvents.lNetworkEvents and FD_WRITE <> 0 then
  begin
    if NetEvents.iErrorCode[FD_WRITE_BIT] <> 0 then
    begin
      LogMessage('Ошибка в событии FD_WRITE: ' +
        GetErrorString(NetEvents.iErrorCode[FD_WRITE_BIT]));
      Break;
    end;
```

```
            // Отправляем то, что лежит в буфере
            if not DoSendBuf then
              Break;
          end;
          if NetEvents.lNetworkEvents and FD_CLOSE <> 0 then
          begin
            // Клиент закрыл соединение
            if NetEvents.iErrorCode[FD_CLOSE_BIT] <> 0 then
            begin
              LogMessage('Ошибка в событии FD_CLOSE: ' +
                GetErrorString(NetEvents.iErrorCode[FD_CLOSE_BIT]));
              Break;
            end;
            LogMessage('Клиент закрыл соединение');
            shutdown(FSocket, SD_BOTH);
            Break;
          end;
        end;
      WSA_WAIT_FAILED:
        begin
          LogMessage('Ошибка при ожидании сообщения: ' + GetErrorString);
          Break;
        end;
      else
        begin
          LogMessage(
            'Внутренняя ошибка сервера — неверный результат ожидания ' +
            IntToStr(WaitRes));
          Break;
        end;
    end;
  until False;
  closesocket(FSocket);
  LogMessage('Нить остановлена');
end;

// Функция возвращает True, если нить завершилась
function TClientThread.GetFinished: Boolean;
begin
  // Ждем окончания работы нити с нулевым тайм-аутом.
  // Если нить завершена, вернется WAIT_OBJECT_0.
  // Если еще работает, вернется WAIT_TIMEOUT.
  Result := WaitForSingleObject(Handle, 0) = WAIT_OBJECT_0;
end;
```

```
// Метод для остановки нити извне.
// Взводим соответствующее событие, а остальное сделаем
// при обработке события
procedure TClientThread.StopThread;
begin
  WSASetEvent(FEvents[0]);
end;

end.
```

Модуль WinSock2_Events, появившийся в списке uses, содержит объявления констант, типов и функций из WinSock 2, которые понадобятся в программе. Модуль ShutdownConst содержит объявления констант для функции shutdown, которые отсутствуют в модуле WinSock Delphi 5 и более ранних версиях — этот модуль нам понадобился, чтобы программу можно было откомпилировать в Delphi 5.

Нить использует три события, дескрипторы которых хранятся в массиве FEvents. Событие FEvents[0] служит для уведомления нити о том, что необходимо завершиться, FEvents[1] — для уведомления о том, что нужно отправить данные, FEvenst[2] связывается с событиями на сокете. Такой порядок выбран не случайно. Если взведено несколько событий, функция WSAWaitForMultipleEvents вернет результат, соответствующий событию с самым младшим из взведенных событий индексом. Соответственно, чем ближе к началу массива, тем более высокий приоритет у события. Событие, связанное с сокетом, имеет наинизший приоритет для того, чтобы повысить устойчивость сервера к DoS-атакам. Если приоритет этого события был бы выше, чем события остановки нити, то в случае закидывания сервера огромным количеством сообщений от клиента, событие FD_READ было бы всегда взведено, и сервер все время тратил бы на обработку этого события, игнорируя сигнал об остановке нити. Соответственно, сигнал об остановке должен иметь самый высокий приоритет, чтобы остановке нити ничего не могло помешать. Тем, как отправляются сообщения, сервер управляет сам, поэтому не приходится ожидать проблем, связанных с тратой излишних ресурсов на обработку сигнала отправки. Соответственно, этому событию присваивается приоритет, промежуточный между событием остановки нити и событием сокета.

Так как клиент по новому протоколу перед отправкой сообщения не обязан ждать, пока сервер ответит на предыдущее, возможны ситуации, когда ответ на следующее сообщение сервер должен готовить уже тогда, когда предыдущее еще не отправлено. Кроме того, сервер может отправить сообщение по собственной инициативе, и этот момент тоже может наступить тогда, когда

предыдущее сообщение еще не отправлено. Таким образом, мы вынуждены формировать очередь сообщений в том или ином виде. Так как протокол TCP, с одной стороны, может объединять несколько пакетов в один, а с другой, не обязан отправлять отдельную строку за один раз, проще всего не делать очередь из отдельных строк, а заранее объединять их в одном буфере и затем пытаться отправить все содержимое буфера. Таким буфером в нашем случае является поле `FSendBuf`, метод `SendString` добавляет строку в этот буфер, а `DoSendBuf` отправляет данные из этого буфера. Если все данные отправить за один раз не удалось, отправленные данные удаляются из буфера, а оставшиеся будут отправлены при следующем вызове `SendBuf`. Все операции с буфером `FSendBuf` выполняются внутри критической секции, т. к. функция `SendString` может вызываться из других нитей. К каждой строке добавляется символ `#0`, который, согласно протоколу, является для клиента разделителем строк в потоке.

Сигналом к отправке данных является событие `FEvents[1]`. Метод `SendString`, помещая данные в буфер, взводит это событие. Если все содержимое буфера за один раз отправить не удастся, то через некоторое время возникнет событие `FD_WRITE`, означающее готовность сокета к приему новых данных. Это событие привязано у нас к `FEvents[2]`, поэтому при наступлении `FEvents[2]` тоже возможна отправка данных.

Для приема данных здесь также используется буфер. Прямой необходимости в этом нет — можно было, как и раньше, помещать данные непосредственно в переменную, хранящую длину строки, а затем и в саму строку. Сделано это в учебных целях, чтобы показать, как можно работать с подобным буфером. Буфер имеет фиксированный размер. Сначала мы читаем из сокета в этот буфер столько, сколько сможем, а потом начинаем разбирать полученное точно так же, как и раньше, копируя данные то в целочисленную, то в строковую переменную. Когда строковая переменная полностью заполняется, строка считается принятой, для пользователя выводится ответ на нее, а в буфер для отправки добавляется ответная строка. Достоинством такого способа является то, что, с одной стороны, за время обработки одного события сервер может прочитать несколько запросов от клиента (если буфер достаточно велик), но, с другой стороны, это не приводит к зацикливанию, если сообщения поступают непрерывно. Другими словами, разработчик здесь сам определяет, какой максимальный объем данных можно получить от сокета за один раз. Иногда это бывает полезно.

Теперь рассмотрим нить, обслуживающую слушающий сокет. Код этой нити приведен в листинге 2.64.

Листинг 2.64. Код нити, обслуживающей слушающий сокет

```
unit ListenThread;
{ Нить, следящая за подключением клиента к слушающему сокету.
  При обнаружении подключения она создает новую нить для работы
  с подключившимся клиентом, а сама продолжает обслуживать
  "слушающий" сокет.
}

interface

uses
  SysUtils, Classes, WinSock, WinSock2_Events;

type
  TListenThread = class(TThread)
  private
    // Сообщение, которое нужно добавить в лог.
    // Хранится в отдельном поле, т. к. метод, вызывающийся
    // через Synchronize, не может иметь параметров.
    FMessage: string;
    // Сокет, находящийся в режиме прослушивания
    FServerSocket: TSocket;
    // События нити
    //  FEvents[0] используется для остановки нити
    //  FEvents[1] связывается с событием FD_ACCEPT
    FEvents: array[0..1] of TWSAEvent;
    // Список нитей клиентов
    FClientThreads: TList;
    // Если True, сервер посылает клиенту сообщения
    // по собственной инициативе
    FServerMsg: Boolean;
    // Вспомогательный метод для вызова через Synchronize
    procedure DoLogMessage;
  protected
    procedure Execute; override;
    // Вывод сообщения в лог главной формы
    procedure LogMessage(const Msg: string);
  public
    constructor Create(ServerSocket: TSocket; ServerMsg: Boolean);
    destructor Destroy; override;
    // Вызывается извне для остановки сервера
    procedure StopServer;
  end;
```

```
implementation

uses
  MainServerUnit, ClientThread;

{ TListenThread }

// "Слушающий" сокет создается в главной нити,
// а сюда передается через параметр конструктора
constructor TListenThread.Create(ServerSocket: TSocket;
  ServerMsg: Boolean);
begin
  FServerSocket := ServerSocket;
  FServerMsg := ServerMsg;
  // Создаем события
  FEvents[0] := WSACreateEvent;
  if FEvents[0] = WSA_INVALID_EVENT then
    raise ESocketError.Create('Ошибка при создании события для сервера: '
      + GetErrorString);
  FEvents[1] := WSACreateEvent;
  if FEvents[1] = WSA_INVALID_EVENT then
    raise ESocketError.Create('Ошибка при создании события для сервера: '
      + GetErrorString);
  if WSAEventSelect(FServerSocket, FEvents[1], FD_ACCEPT) =
    SOCKET_ERROR then
    raise ESocketError.Create(
      'Ошибка при привязывании серверного сокета к событию: ' +
      GetErrorString);
  FClientThreads := TList.Create;
  inherited Create(False);
end;

destructor TListenThread.Destroy;
begin
  // Убираем за собой
  FClientThreads.Free;
  WSACloseEvent(FEvents[0]);
  WSACloseEvent(FEvents[1]);
  inherited;
end;

procedure TListenThread.Execute;
var
  // Сокет, созданный для общения с подключившимся клиентом
  ClientSocket: TSocket;
```

```
  // Адрес подключившегося клиента
  ClientAddr: TSockAddr;
  ClientAddrLen: Integer;
  NetEvents: TWSANetworkEvents;
  I: Integer;
  WaitRes: Cardinal;
begin
  LogMessage('Сервер начал работу');
  // Начинаем бесконечный цикл
  repeat
    // Ожидание события с 15-секундным тайм-аутом
    WaitRes :=
      WSAWaitForMultipleEvents(2, @FEvents, False, 15000, False);
    case WaitRes of
      WSA_WAIT_EVENT_0:
        // Событие FEvents[0] взведено — это означает, что
        // сервер должен остановиться.
        begin
          LogMessage('Сервер получил сигнал завершения работы');
          // Просто выходим из цикла, остальное сделает код после цикла
          Break;
        end;
      WSA_WAIT_EVENT_0 + 1:
        // Событие FEvents[1] взведено.
        // Это должно означать наступление события FD_ACCEPT.
        begin
          // Проверяем, почему событие взведено,
          // и заодно сбрасываем его
          if WSAEnumNetworkEvents(FServerSocket, FEvents[1], NetEvents) =
            SOCKET_ERROR then
          begin
            LogMessage('Ошибка при получении списка событий: ' +
              GetErrorString);
            Break;
          end;
          // Защита от "тупой" ошибки — проверка того,
          // что наступило нужное событие
          if NetEvents.lNetworkEvents and FD_ACCEPT = 0 then
          begin
            LogMessage(
              'Внутренняя ошибка сервера — неизвестное событие');
            Break;
          end;
```

```
      // Проверка, не было ли ошибок
      if NetEvents.iErrorCode[FD_ACCEPT_BIT] <> 0 then
      begin
        LogMessage('Ошибка при подключении клиента: ' +
          GetErrorString(NetEvents.iErrorCode[FD_ACCEPT_BIT]));
        Break;
      end;
      ClientAddrLen := SizeOf(ClientAddr);
      // Проверяем наличие подключения
      ClientSocket := accept(FServerSocket, @ClientAddr,
        @ClientAddrLen);
      if ClientSocket = INVALID_SOCKET then
      begin
        // Ошибка в функции accept возникает только тогда, когда
        // происходит нечто экстраординарное. Продолжать работу
        // в этом случае бессмысленно. Единственное возможное
        // в нашем случае исключение — ошибка WSAEWOULDBLOCK,
        // которая может возникнуть, если срабатывание события
        // было ложным, и подключение от клиента отсутствует.
        if WSAGetLastError <> WSAEWOULDBLOCK then
        begin
          LogMessage('Ошибка при подключении клиента: ' +
            GetErrorString);
          Break;
        end;
      end;
      // Создаем новую нить для обслуживания подключившегося клиента
      // и передаем ей сокет, созданный для взаимодействия с ним.
      // Указатель на нить сохраняем в списке
      FClientThreads.Add(
        TClientThread.Create(ClientSocket, ClientAddr));
    end;
  WSA_WAIT_TIMEOUT:
    // Ожидание завершено по тайм-ауту
    begin
      // Проверяем, есть ли клиентские нити, завершившие работу.
      // Если есть такие нити, удаляем их из списка
      // и освобождаем объекты.
      for I := FClientThreads.Count — 1 downto 0 do
        if TClientThread(FClientThreads[I]).Finished then
        begin
          TClientThread(FClientThreads[I]).Free;
          FClientThreads.Delete(I);
        end;
```

```
          // Если разрешены сообщения от сервера, отправляем
          // всем клиентам сообщение с текущим временем
          if FServerMsg then
            for I := 0 to FClientThreads.Count – 1 do
              TClientThread(FClientThreads[I]).SendString(
                'Время на сервере ' + TimeToStr(Now));
        end;
      WSA_WAIT_FAILED:
        // При ожидании возникла ошибка. Это может означать
        // только какой-то серьезный сбой в библиотеке сокетов.
        begin
          LogMessage('Ошибка при ожидании события сервера: ' +
            GetErrorString);
          Break;
        end;
      else
        // Неожиданный результат при ожидании
        begin
          LogMessage(
            'Внутренняя ошибка сервера – неожиданный результат ожидания '
            + IntToStr(WaitRes));
          Break;
        end;
    end;
  until False;
  // Останавливаем и уничтожаем все нити клиентов
  for I := 0 to FClientThreads.Count – 1 do
  begin
    TClientThread(FClientThreads[I]).StopThread;
    TClientThread(FClientThreads[I]).WaitFor;
    TClientThread(FClientThreads[I]).Free;
  end;
  closesocket(FServerSocket);
  LogMessage('Сервер завершил работу');
  Synchronize(ServerForm.OnStopServer);
end;

// Завершение работы сервера. Просто взводим соответствующее
// событие, а остальное делает код в методе Execute.
procedure TListenThread.StopServer;
begin
  WSASetEvent(FEvents[0]);
end;

end.
```

Нить `TListenThread` реализует сразу несколько функций. Во-первых, она обнаруживает подключение клиентов и создает нити для их обслуживания. Во-вторых, уничтожает объекты завершившихся нитей. В-третьих, она с определенной периодичностью ставит в очередь на отправку всем клиентам сообщение с текущим временем сервера. И в-четвертых, управляет уничтожением клиентских нитей при завершении работы сервера.

Здесь следует пояснить, почему выбран такой способ управления временем жизни объектов клиентских нитей. Очевидно, что нужно иметь список всех нитей, чтобы обеспечить возможность останавливать их и ставить в очередь сообщения для отправки клиентам (этот список реализован переменной `FClientThreads`). Если бы объект `TClientThread` автоматически удалялся при завершении работы нити, в его деструкторе пришлось бы предусмотреть и удаление ссылки на объект из списка, а это значит, что к списку пришлось бы обращаться из разных нитей. Соответственно, потребовалось бы синхронизировать обращение к списку, и здесь мы бы столкнулись с одной неприятной проблемой. Когда нить `TListenThread` получает команду завершиться, она должна завершить все клиентские нити. Для этого она должна использовать их список для отправки сигнала и ожидания их завершения. И получилась бы взаимная блокировка, потому что нить `TListenThread` ждала бы завершения клиентских нитей, а они не могли бы завершиться, потому что им требовался бы список, захваченный нитью `TListenThread`. Избежать этого можно с помощью асинхронных сообщений, но в нашем случае реализация этого механизма затруднительна (хотя и возможна). Для простоты был выбран другой вариант: клиентские нити сами свои объекты не удаляют, а к списку имеет доступ только нить `TListenThread`, которая время от времени проходит по списку и удаляет объекты всех завершившихся нитей. В этом случае клиентские нити не используют синхронизацию при завершении, и нить `TListenThread` может дожидаться их.

Нить `TListenThread` использует два события: `FEvents[0]` для получения сигнала о необходимости закрытия и `FEvents[1]` для получения уведомлений о возникновении события `FD_ACCEPT` на слушающем сокете (т. е. о подключении клиента). Порядок следования событий в массиве определяется теми же соображениями, что и в случае клиентской нити: сигнал остановки нити должен иметь более высокий приоритет, чтобы в случае DoS-атаки нить могла быть остановлена.

И поиск завершившихся нитей, и отправка сообщений с текущим временем клиентам осуществляется в том случае, если при ожидании события произошел тайм-аут (который в нашем случае равен 15 с). Подключение клиента — событие достаточно редкое, поэтому такое решение выглядит вполне оправ-

данным. Для тех экзотических случаев, когда клиенты часто подключаются и отключаются, можно предусмотреть еще одно событие у нити `TListenThread`, при наступлении которого она будет проверять список клиентов. Клиентская нить при своем завершении будет взводить это событие. Что же касается отправки сообщений клиентам, то в обработчик тайм-аута этот код помещен в демонстрационных целях. В реальной программе инициировать отправку сообщений клиентам будет, скорее всего, другой код, например, код главной нити по команде пользователя.

Несмотря на изменение протокола, новый сервер был бы вполне совместим со старым клиентом SimpleClient *(см. разд. 2.1.11)*, если бы не отправлял сообщения по своей инициативе. Действительно, прочие изменения в протоколе разрешают клиенту отправлять новые сообщения до получения ответа сервера, но не обязывают его делать это. В класс `TClientThread` добавлено логическое поле `FServerMsg`. Если оно равно `False`, то сервер не посылает клиентам сообщений по собственной инициативе, т. е. работает в режиме совместимости со старым клиентом. Поле `FServerMsg` инициализируется в соответствии с параметром, переданным в конструктор, т. е. в соответствии с состоянием галочки **Сообщения от сервера**, расположенной на главной форме. Если перед запуском сервера она снята, сервер не будет сам посылать сообщения, и старый клиент сможет обмениваться данными с ним.

Запуск сервера практически не отличается от запуска сервера MultithreadedServer (см. листинг 2.19), только теперь объект, созданный конструктором, запоминается классом главной формы, чтобы потом можно было сервер остановить. Остановка осуществляется методом `StopServer` (листинг 2.65).

Листинг 2.65. Метод StopServer

```
// Остановка сервера
procedure TServerForm.StopServer;
begin
  // Запрещаем кнопку, чтобы пользователь не мог нажать ее
  // еще раз, пока сервер не остановится.
  BtnStopServer.Enabled := False;
  // Ожидаем завершения слушающей нити. Так как вывод сообщений
  // эта нить осуществляет через Synchronize, выполняемый главной
  // нитью в петле сообщений, вызов метода WaitFor мог бы привести
  // к взаимной блокировке: главная нить ждала бы, когда завершится
  // нить TListenThread, а та, в свою очередь — когда главная нить
  // выполнит Synchronize. Чтобы этого не происходило, организуется
  // ожидание с локальной петлей сообщений.
```

```
  if Assigned(FListenThread) then
  begin
    FListenThread.StopServer;
    while Assigned(FListenThread) do
    begin
      Application.ProcessMessages;
      Sleep(10);
    end;
  end;
end;
```

Данный метод вызывается в обработчике нажатия кнопки **Остановить** и при завершении приложения. Сервер можно многократно останавливать и запускать вновь, не завершая приложение.

Чтобы увидеть все возможности сервера, потребуется новый клиент. На компакт-диске он называется EventSelectClient, но "EventSelect" в данном случае означает только то, что клиент является парным к серверу EventSelectServer. Сам клиент функцию `WSAEventSelect` не использует, поскольку она неудобна, когда нужно работать только с одним сокетом. Поэтому клиент работает в асинхронном режиме, основанном на сообщениях, т. е. посредством функции `WSAAsyncSelect`.

Клиент может получать от сервера сообщения двух видов: те, которые сервер посылает в ответ на запросы клиента, и те, которые он посылает по собственной инициативе. Но различить эти сообщения клиент не может: например, если клиент отправляет запрос серверу, а потом получает от него сообщение, он не может определить, то ли это сервер ответил на его запрос, то ли именно в этот момент сервер сам отправил клиенту свое сообщение. Соответственно, сообщения обоих типов читает один и тот же код.

Примечание

В принципе, протокол мог бы быть определен таким образом, что ответы на запросы клиента и сообщения, посылаемые сервером по собственной инициативе, имели бы разный формат, по которому их можно было бы различить и читать по-разному. Но даже при этом форматы нельзя различить, пока сообщение не будет прочитано хотя бы частично, так что начало чтения будет выполняться единообразно в любом случае.

Подключение клиента к серверу выполняется точно так же, как в листинге 2.16, за исключением того, что после выполнения функции `connect` сокет переводится в асинхронный режим, и его события `FD_READ` и `FD_CLOSE` связываются с сообщением `WM_SOCKETMESSAGE`. Обработчик этого сообщения приведен в листинге 2.66.

Листинг 2.66. Получение данных клиентом

```
procedure TESClientForm.WMSocketMessage(var Msg: TWMSocketMessage);
const
  // Размер буфера для получения данных
  RecvBufSize = 4096;
var
  // Буфер для получения данных
  RecvBuf: array[0..RecvBufSize – 1] of Byte;
  RecvRes: Integer;
  P: Integer;
begin
  // Защита от "тупой" ошибки
  if Msg.Socket <> FSocket then
  begin
    MessageDlg('Внутренняя ошибка программы – неверный сокет',
      mtError, [mbOK], 0);
    Exit;
  end;
  if Msg.SockError <> 0 then
  begin
    MessageDlg('Ошибка при взаимодействии с сервером'#13#10 +
      GetErrorString(Msg.SockError), mtError, [mbOK], 0);
    OnDisconnect;
    Exit;
  end;
  case Msg.SockEvent of
    FD_READ:
      // Получено сообщение от сервера
      begin
        // Читаем столько, сколько можем
        RecvRes := recv(FSocket, RecvBuf, RecvBufSize, 0);
        if RecvRes > 0 then
        begin
          // Увеличиваем строку на размер прочитанных данных
          P := Length(FRecvStr);
          SetLength(FRecvStr, P + RecvRes);
          // Копируем в строку полученные данные
          Move(RecvBuf, FRecvStr[P + 1], RecvRes);
          // В строке может оказаться несколько строк от сервера,
          // причем последняя может прийти не целиком.
          // Ищем в строке символы #0, которые, согласно протоколу,
          // являются разделителями строк.
          P := Pos(#0, FRecvStr);
```

```
          while P > 0 do
          begin
            AddMessageToRecvMemo('Сообщение от сервера: ' +
              Copy(FRecvStr, 1, P – 1));
            // Удаляем из строкового буфера выведенную строку
            Delete(FRecvStr, 1, P);
            P := Pos(#0, FRecvStr);
          end;
        end
        else if RecvRes = 0 then
        begin
          MessageDlg('Сервер закрыл соединение'#13#10 +
            GetErrorString, mtError, [mbOK], 0);
          OnDisconnect;
        end
        else
        begin
          if WSAGetLastError <> WSAEWOULDBLOCK then
          begin
            MessageDlg('Ошибка при получении данных от клиента'#13#10 +
              GetErrorString, mtError, [mbOK], 0);
            OnDisconnect;
          end;
        end;
      end;
    FD_CLOSE:
      begin
        MessageDlg('Сервер закрыл соединение', mtError, [mbOK], 0);
        shutdown(FSocket, SD_BOTH);
        OnDisconnect;
      end;
    else
      begin
        MessageDlg('Внутренняя ошибка программы – неизвестное событие ' +
          IntToStr(Msg.SockEvent), mtError, [mbOK], 0);
        OnDisconnect;
      end;
  end;
end;
```

Здесь мы используем новый способ чтения данных. Он во многом похож на тот, который применен в сервере. Функция recv вызывается один раз за один вызов обработчика значений и передает данные в буфер фиксированного

размера RecvBuf. Затем в буфере ищутся границы отдельных строк (символы #0), строки, полученные целиком, выводятся. Если строка получена частично (а такое может случиться не только из-за того, что она передана по частям, но и из-за того, что в буфере просто не хватило место для приема ее целиком), ее начало следует сохранить в отдельном буфере, чтобы добавить к тому, что будет прочитано при следующем событии FD_READ. Этот буфер реализуется полем FRecvStr типа string. После чтения к содержимому этой строки добавляется содержимое буфера RecvBuf, а затем из строки выделяются все подстроки, заканчивающиеся на #0. То, что остается в строке FRecvStr после этого, — это начало строки, прочитанной частично. Оно будет учтено при обработке следующего события FD_READ.

Примечание

Описанный алгоритм разбора буфера прост, но неэффективен с точки зрения нагрузки на процессор и использования динамической памяти, особенно в тех случаях, когда в буфере RecvBuf оказывается сразу несколько строк. Это связано с тем, что при добавлении содержимого RecvBuf к FRecvStr и последующем поочередном удалении строк из FRecvStr происходит многократное перераспределение памяти, выделенной для строки. Алгоритм можно оптимизировать: все строки, которые поместились в RecvBuf целиком, выделять непосредственно из этого буфера, не помещая в FRecvStr, а помещать туда только то, что действительно нужно сохранить между обработкой разных событий FD_READ. Реализацию такого алгоритма рекомендуем выполнить в качестве самостоятельного упражнения.

При отправке данных вероятность того, что функция send не сможет быть выполнена сразу, достаточно мала. Кроме того, как мы уже говорили, блокировка клиента при отправке данных часто бывает вполне приемлема из-за редкости и непродолжительности. Таким образом, блокирующий режим из-за своей простоты наиболее удобен при отправке данных серверу клиентом. Но мы не можем перевести сокет, работающий в асинхронном режиме, в блокирующий режим на время отправки, зато можем этот режим имитировать. Занимается этим метод SendString (листинг 2.67).

Листинг 2.67. Метод SendString, имитирующий блокирующий режим отправки

```
// Отправка строки серверу. Функция имитирует блокирующий
// режим работы сокета: если за один раз не удается отправить
// данные, попытка отправить их продолжается до тех пор,
// пока все данные не будут отправлены или пока не возникнет ошибка.
procedure TESClientForm.SendString(const S: string);
var
  SendRes: Integer;
```

```
  // Буфер, куда помещается отправляемое сообщение
  SendBuf: array of Byte;
  // Сколько байтов уже отправлено
  BytesSent: Integer;
begin
  if Length(S) > 0 then
  begin
    // Отправляемое сообщение состоит из длины строки и самой строки.
    // Выделяем для буфера память, достаточную для хранения
    // и того и другого.
    SetLength(SendBuf, SizeOf(Integer) + Length(S));
    // Копируем в буфер длину строки
    PInteger(@SendBuf[0])^ := Length(S);
    // А затем — саму строку
    Move(S[1], SendBuf[SizeOf(Integer)], Length(S));
    BytesSent := 0;
    // Повторяем попытку отправить до тех пор, пока все содержимое
    // буфера не будет отправлено серверу.
    while BytesSent < Length(SendBuf) do
    begin
      SendRes := send(FSocket, SendBuf[BytesSent],
        Length(SendBuf) — BytesSent, 0);
      if SendRes > 0 then
        Inc(BytesSent, SendRes)
      else
        if WSAGetLastError = WSAEWOULDBLOCK then
          Sleep(10)
        else
        begin
          MessageDlg('Ошибка при отправке данных серверу'#13#10 +
            GetErrorString, mtError, [mbOK], 0);
          OnDisconnect;
          Exit;
        end;
    end;
  end;
end;
```

Имитация блокирующего режима осуществляется очень просто: если сообщение не удалось отправить сразу, после небольшой паузы производится попытка отправить то, что еще не отправлено, и так до тех пор, пока не будет отправлено все или пока не возникнет ошибка. В программе SimpleClient мы отправляли длину строки и саму строку разными вызовами send. Теперь, из-

за того, что функция send может отправить только часть переданных ей данных, это становится неудобным из-за громоздкости многочисленных проверок. Поэтому мы создаем один буфер, куда заносим и длину строки, и саму строку, и затем передаем его как единое целое.

Примечание

Далее мы познакомимся с функцией WSASend, которая позволяет отправлять данные, находящиеся не в одном, а в нескольких разных местах. Если бы мы использовали ее, можно было бы не объединять самостоятельно длину строки и саму строку в специальном буфере, а просто передать два указателя на длину и на строку.

Чтобы продемонстрировать возможности сервера по приему нескольких слившихся запросов, клиент должен отправлять ему несколько строк сразу, поэтому на главной форме клиента мы заменяем однострочное поле ввода на многострочное (т. е. TEdit на TMemo). При нажатии кнопки **Отправить** клиент отправляет серверу все непустые строки из этого поля ввода.

Других существенных отличий от SimpleClient программа EventSelectClient не имеет. Получившийся пример работает не только с сервером EventSelectServer, но и с любым сервером, написанным нами ранее. Действительно, ни один из этих серверов не требует, чтобы на момент получения запроса от клиента в буфере сокета ничего не было, кроме этого запроса. Поэтому то, что EventSelectClient может отправлять несколько сообщений сразу, не помешает им работать: просто, в отличие от EventSelectServer, они будут обрабатывать эти запросы строго по одному, а не получать из сокета сразу несколько штук.

2.2.9. Перекрытый ввод-вывод

Прежде чем переходить к рассмотрению перекрытого ввода-вывода, вспомним, какие модели ввода-вывода нам уже известны. Появление разных моделей связано с тем, что операции ввода-вывода не всегда могут быть выполнены немедленно.

Самая простая модель ввода-вывода — блокирующая. В блокирующем режиме, если операция не может быть выполнена немедленно, работа нити приостанавливается до тех пор, пока не возникнут условия для выполнения операции. В неблокирующей модели ввода-вывода операция, которая не может быть выполнена немедленно, завершается с ошибкой. И наконец, в асинхронной модели ввода-вывода предусмотрена система уведомлений о том, что операция может быть выполнена немедленно.

При использовании перекрытого ввода-вывода операция, которая не может быть выполнена немедленно, формально завершается ошибкой — в этом за-

ключается сходство перекрытого ввода-вывода и неблокирующего режима. Однако, в отличие от неблокирующего режима, при перекрытом вводе-выводе библиотека сокетов начинает выполнять операцию в фоновом режиме, и после ее завершения начавшая операцию программа получает уведомление об успешно выполненной операции или о возникшей при ее выполнении фатальной ошибке. Несколько операций ввода-вывода могут одновременно выполняться в фоновом режиме, как бы перекрывая работу инициировавшей их нити и друг друга. Именно поэтому данная модель получила название модели перекрытого ввода-вывода.

Перекрытый ввод-вывод существовал и в спецификации WinSock 1, но реализовывался только для линии NT. Специальных функций для перекрытого ввода-вывода в WinSock 1 не было, требовались функции `ReadFile` и `WriteFile`, в которые вместо дескриптора файла подставлялся дескриптор сокета. В WinSock 2 появилась полноценная поддержка перекрытого ввода-вывода для всех версий Windows, а в спецификацию добавились новые функции для его реализации, избавившие от необходимости использования функций файлового ввода-вывода. Здесь мы будем рассматривать перекрытый ввод-вывод только в спецификации WinSock 2, т. к. старый вариант из-за своих ограничений уже не имеет практического смысла.

Существуют два варианта уведомления о завершении операции перекрытого ввода-вывода: через событие и через процедуру завершения. Кроме того, программа может не дожидаться уведомления, а проверять состояние запроса перекрытого ввода-вывода с помощью функции `WSAGetOverlappedResult` (ее мы рассмотрим позже).

Чтобы сокет мог использоваться в операциях перекрытого ввода-вывода, при его создании должен быть установлен флаг `WSA_FLAG_OVERLAPPED` (функция `socket` неявно устанавливает этот флаг). Для выполнения операций перекрытого ввода-вывода сокет не нужно переводить в какой-либо особый режим, достаточно обычные функции `send` и `recv` заменить на `WSARecv` и `WSASend`. Сначала мы рассмотрим функцию `WSARecv`, прототип которой приведен в листинге 2.68.

Листинг 2.68. Функция `WSARecv`

```
// ***** Описание на C++ *****
int WSARecv(
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesRecvd,
  LPDWORD lpFlags,
```

```
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);

// ***** Описание на Delphi *****
function WSARecv(
  S: TSocket;
  lpBuffers: PWSABuf;
  dwBufferCount: DWORD;
  var NumberOfBytesRecvd: DWORD;
  var Flags: DWORD;
  lpOverlapped: PWSAOverlapped;
  lpCompletionRoutine: TWSAOverlappedCompletionRoutine): Integer;
```

Перекрытым вводом-выводом управляют два последних параметра функции, но WSARecv обладает и другими дополнительными по сравнению с функцией recv возможностями, не связанными с перекрытым вводом-выводом. Если оба этих параметра равны nil, или сокет создан без указания флага WSA_FLAG_OVERLAPPED, функция работает в обычном блокирующем или неблокирующем режиме, который установлен для сокета. При этом ее поведение отличается от поведения функции recv только тремя незначительными аспектами: во-первых, вместо одного буфера ей можно передать несколько, заполняемых последовательно. Во-вторых, флаги передаются ей не как значение, а как параметр-переменная, и при некоторых условиях функция WSARecv может их изменять (при использовании TCP и UDP флаги никогда не меняются, поэтому мы не будем рассматривать здесь эту возможность). В-третьих, при успешном завершении функция WSARecv возвращает ноль, а не число прочитанных байтов (последнее возвращается через параметр lpNumberOfBytesRecvd).

Буферы, в которые нужно поместить данные, передаются функции WSARecv через параметр lpBuffers. Он содержит указатель на начало массива структур TWSABuf, а параметр dwBufferCount — число элементов в этом массиве. Ранее мы знакомились со структурой TWSABuf (см. листинг 2.39): она содержит указатель на начало буфера и его размер. Соответственно, массив таких структур определяет набор буферов. При чтении данных заполнение буферов начинается с первого буфера в массиве lpBuffers, затем, если в нем не хватает места, заполняется второй буфер и т. д. Функция не переходит к следующему буферу, пока не заполнит предыдущий до последнего байта. Таким образом, данные, получаемые с помощью функции WSARecv, могут быть помещены в несколько несвязных областей памяти, что иногда бывает удобно, если принимаемые сообщения имеют строго определенный формат с фиксированными размерами компонентов пакета: в этом случае можно каждый компонент поместить в свой независимый буфер.

Теперь переходим непосредственно к рассмотрению перекрытого ввода-вывода на основе событий. Для реализации этого режима при вызове функции WSARecv параметр lpCompletionRoutine должен быть равен nil, а через параметр lpOverlapped передается указатель на запись TWSAOverlapped, которая определена следующим образом (листинг 2.69).

Листинг 2.69. Тип TWSAOverlapped

```
// ***** Описание на C++ *****
typedef struct _WSAOVERLAPPED {
  DWORD Internal;
  DWORD InternalHigh;
  DWORD Offset;
  DWORD OffsetHigh;
  WSAEVENT hEvent;
} WSAOVERLAPPED, *LPWSAOVERLAPPED;

// ***** Описание на Delphi *****
PWSAOverlapped = ^TWSAOverlapped;
TWSAOverlapped = packed record
  Internal, InternalHigh, Offet, OffsetHigh: DWORD;
  hEvent: TWSAEvent;
end;
```

Поля Internal, InternalHigh, Offset и OffsetHigh предназначены для внутреннего использования системой, программа не должна выполнять никаких действий с ними. Поле hEvent задает событие, которое будет взведено при завершении операции перекрытого ввода-вывода. Если на момент вызова функции WSARecv данные в буфере сокета отсутствуют, она вернет значение SOCKET_ERROR, а функция WSAGetLastError — WSA_IO_PENDING (997). Это значит, что операция начала выполняться в фоновом режиме. В этом случае функция WSARecv не изменяет значения параметров NumberOfBytesRecvd и Flag. Поля структуры TWSAOverlapped при этом также модифицируются, и эта структура должна быть сохранена программой в неприкосновенности до окончания операции перекрытого ввода-вывода. После окончания операции будет взведено событие, указанное в поле hEvent параметра lpOverlapped. При необходимости программа может дождаться этого взведения с помощью функции WSAWaitForMultipleEvents.

Как только запрос будет выполнен, в буферах, переданных через параметр lpBuffers, оказываются принятые данные. Но знания одного только факта, что запрос выполнен, недостаточно, чтобы этими данными воспользоваться, потому что, во-первых, неизвестен размер этих данных, а во-вторых, неиз-

вестно, успешно ли завершена операция перекрытого ввода-вывода. Для получения недостающей информации служит функция WSAGetOverlappedResult, прототип которой приведен в листинге 2.70.

Листинг 2.70. Функция WSAGetOverlappedResult

```
// ***** Описание на C++ *****
BOOL WSAGetOverlappedResult(
  SOCKET s,
  LPWSAOVERLAPPED lpOverlapped,
  LPDWORD lpcbTransfer,
  BOOL fWait,
  LPDWORD lpdwFlags);

// ***** Описание на Delphi *****
function WSAGetOverlappedResult(
  S: TSocket;
  lpOverlapped: PWSAOverlapped;
  var cbTransfer: DWORD;
  fWait: BOOL;
  var Flags: DWORD): BOOL;
```

Параметры S и lpOverlapped функции WSAGetOverlappedResult определяют сокет и операцию перекрытого ввода-вывода, информацию о которой требуется получить. Их значения должны совпадать со значениями соответствующих параметров, переданных функции WSARecv. Через параметр cbTransfer возвращается число полученных байтов, а через параметр Flags — флаги (напомним, что в случае TCP и UDP флаги не модифицируются, и выходное значение параметра Flags будет равно входному значению параметра Flags функции WSARecv).

Допускается вызов функции WSAGetOverlappedResult до того, как операция перекрытого ввода-вывода будет завершена. В этом случае поведение функции зависит от параметра fWait. Если он равен True, функция переводит нить в состояние ожидания до тех пор, пока операция не будет завершена. Если он равен False, функция завершается немедленно с ошибкой WSA_IO_INCOMPLETE (996).

Функция WSAGetOverlappedResult возвращает True, если операция перекрытого ввода-вывода успешно завершена, и False, если произошли какие-то ошибки. Ошибка может возникнуть в одном из трех случаев:

1. Операция перекрытого ввода-вывода еще не завершена, а параметр fWait равен False.

2. Операция перекрытого ввода-вывода завершилась с ошибкой (например, из-за разрыва связи).
3. Параметры, переданные функции `WSAGetOverlappedResult`, имеют некорректные значения.

Точную причину, по которой функция вернула `False`, можно установить стандартным образом — по коду ошибки, возвращаемому функцией `WSAGetLastError`.

В принципе, программа может вообще не использовать события для отслеживания завершения операции ввода-вывода, а вызывать вместо этого время от времени функцию `WSAGetOverlappedResul` в удобные для себя моменты. Тогда при вызове функции `WSARecv` можно указать нулевое значение события `hEvent`. Но следует иметь в виду, что при вызове функции `WSAGetOverlappedResult` с параметром `fWait`, равным `True`, указанное событие служит для ожидания завершения операции, и если событие не задано, возникнет ошибка. Таким образом, если событие не используется, функция `WSAGetOverlappedResult` не может вызываться в режиме ожидания.

Отдельно рассмотрим ситуацию, когда на момент вызова функции `WSARecv` с ненулевым параметром `lpOverlapped` во входном буфере сокета есть данные. В этом случае функция отработает так же, как и в неперекрытом режиме, т. е. изменит значения параметров `NumberOfBytesRecvd` и `Flags` и вернет ноль, свидетельствующий об успешном выполнении функции. Но при этом событие будет взведено, а в структуру `lpOverlapped` будет внесена вся необходимая информация. Благодаря этому последующие вызовы функций `WSAWaitForMultipleEvents` и `WSAGetOverlappedResult` будут выполняться корректно, т. е. таким образом, как если бы функция `WSARecv` завершилась с ошибкой `WSA_IO_PENDING`, и сразу после этого в буфер сокета поступили данные. Это позволяет выполнить обработку результатов операций перекрытого ввода-вывода с помощью одного и того же кода независимо от того, были ли в буфере сокета данные на момент начала операции или нет.

Новая операция перекрытого ввода-вывода может быть начата до того, как закончится предыдущая. Это удобно при работе с несколькими сокетами: можно выполнять операции с ними параллельно в фоновом режиме, получая уведомления о завершении каждой из операций.

В MSDN не написано явно, что будет, если вызвать для сокета функцию `WSARecv` повторно, до того как будет завершена предыдущая операция перекрытого чтения (но запрета на такие действия тоже нет). Эксперименты показывают, что в этом случае операции перекрытого чтения встают в очередь, т. е. первый полученный сокетом пакет приводит к завершению операции, начатой первой, второй пакет — к завершению операции, начатой второй,

и т. д. Но поскольку это явно не документировано, лучше не полагаться на то, что такой порядок будет всегда соблюдаться.

В качестве примера реализации перекрытого ввода-вывода рассмотрим ситуацию, когда программа начинает операцию чтения данных из сокета, время от времени проверяя статус операции (листинг 2.71). События в этом примере не используются, проверка осуществляется с помощью функции `WSAGetOverlappedResult`.

Листинг 2.71. Перекрытый ввод-вывод с использованием функции `WSAGetOverlappedResult`

```
var
  S: TSocket;
  Overlapped: TWSAOverlapped;
  BufPtr: TWSABuf;
  RecvBuf: array[1..100] of Char;
  Cnt, Flags: Cardinal;
begin
  // Инициализация WinSock, создание сокета S, привязка его к адресу
  ......
  // Подготовка структуры, задающей буфер
  BufPtr.Buf := @RBuf;
  BufPtr.Len := SizeOf(RBuf);
  // Подготовка структуры TWSAOverlapped
  // Поля Internal, InternalHigh, Offset, OffsetHigh программа
  // не устанавливает
  Overlapped.hEvent := 0;
  Flags := 0;
  // Начало операции перекрытого получения данных
  WSARecv(S, @BufPtr, 1, Cnt, Flags, @Overlapped, nil);
  while True do
  begin
    if WSAGetOverlappedResult(S, @Overlapped, Cnt, False, Flags) then
    begin
      // Данные получены, находятся в RecvBuf, обрабатываем
      ......
      // Выходим из цикла
      Break;
    end
    else if WSAGetLastError <> WSA_IO_INCOMPLETE then
    begin
      // Произошла ошибка, анализируем ее
      ......
```

```
      // Выходим из цикла
      Break;
    end
    else
    begin
      // Операция чтения не завершена
      // Занимаемся другими действиями
    end;
  end;
```

Теперь перейдем к рассмотрению перекрытого ввода-вывода на основе процедур завершения. Для этого при вызове функции WSARecv нужно задать указатель на процедуру завершения, описанную в программе. Процедура завершения должна иметь прототип, приведенный в листинге 2.72.

Листинг 2.72. Прототип процедуры завершения

```
// ***** Описание на C++ *****
void CALLBACK CompletionROUTINE(
  DWORD dwError,
  DWORD cbTransferred,
  LPWSAOVERLAPPED lpOverlapped,
  DWORD dwFlags);

// ***** Описание на Delphi *****
TWSAOverlappedCompletionRoutine = procedure(
  dwError: DWORD;
  cdTransferred: DWORD;
  lpOverlapped: PWSAOverlapped;
  dwFlags: DWORD); stdcall;
```

При использовании процедур завершения в функцию WSARecv также нужно передавать указатель на запись TWSAOverlapped через параметр lpOverlapped, но значение поля hEvent этой структуры игнорируется. Вместо взведения события при завершении операции будет вызвана процедура, указанная в качестве параметра функции WSARecv. Указатель на структуру, заданный при вызове WSARecv, передается в процедуру завершения через параметр lpOverlapped. Смысл остальных параметров очевиден: dwError — это код ошибки (или ноль, если операция завершена успешно), cbTransferred — число полученных байтов (само полученное сообщение копируется в буферы, указанные при вызове функции WSARecv), а dwFlags — флаги.

Процедура завершения всегда выполняется в той нити, которая инициировала начало операции перекрытого ввода-вывода. Но система не может прерывать

нить для выполнения процедуры завершения в любой удобный ей момент — нить должна перейти в состояние ожидания. В это состояние ее можно перевести, например, с помощью функции SleepEx, имеющей следующий прототип:

```
function SleepEx(dwMilliseconds: DWORD; bAlertable: BOOL): DWORD;
```

Функция SleepEx является частью стандартного API системы и импортируется модулем Windows. Она переводит нить в состояние ожидания. Параметр dwMilliseconds задает время ожидания в миллисекундах (или значение INFINITE для бесконечного ожидания). Параметр bAlertable указывает, допустимо ли прерывание состояния ожидания для выполнения процедуры завершения. Если bAlertable равен False, функция SleepEx ведет себя так же, как функция Sleep, т. е. просто приостанавливает работу нити на заданное время. Если bAlertable равен True, нить может быть выведена системой из состояния ожидания раньше, чем истечет заданное время, если возникнет необходимость выполнить процедуру завершения. О причине завершения ожидания программа может судить по результату, возвращаемому функцией SleepEx: ноль в случае завершения по тайм-ауту и WAIT_IO_COMPLETION в случае завершения из-за выполнения процедуры завершения (в последнем случае сначала выполняется процедура завершения, а потом только происходит возврат из функции SleepEx). Если завершились несколько операций перекрытого ввода-вывода, в результате выполнения SleepEx будут вызваны процедуры завершения для всех этих операций.

Существует также возможность ожидать выполнения процедуры завершения одновременно с ожиданием взведения событий с помощью функции WSAWaitForMultuipleEvents. Напомним, что у этой функции также есть параметр fAlertable. Если задать его равным True, то при необходимости выполнения процедуры завершения функция WSAWaitForMultipleEvents, подобно функции SleepEx, выполняет эту процедуру и возвращает WAIT_IO_COMPLETION.

Если программа выполняет одновременно несколько операций перекрытого ввода-вывода, возникает вопрос, как при вызове процедуры завершения определить, какая из них завершилась. Для каждой такой операции должен быть создан уникальный экземпляр записи TWSAOverlapped. Процедура завершения получает указатель на тот экземпляр, который использовался для начала завершившейся операции. Можно сравнить указатель с теми, которые были заданы при запуске операций перекрытого ввода-вывода, и определить, какая из них завершилась. Это не всегда бывает удобно из-за необходимости где-то хранить список указателей, заданных при начале операций перекрытого ввода-вывода. Существуют еще два варианта решения этой проблемы. Первый заключается в создании своей процедуры завершения для каждой из выполняющихся параллельно операций. Этот способ приводит к получению

громоздкого кода и может быть неудобен, если число одновременно выполняющихся операций заранее неизвестно. Он целесообразен только при одновременном выполнении разнородных операций, требующих разных алгоритмов при обработке их завершения. Другой вариант предлагается в MSDN. Так как при работе через процедуры завершения значение поля `hEvent` структуры `TWSAOverlapped` игнорируется системой, программа может записать туда любое 32-битное значение и с его помощью определить, какая из операций завершена. В строго типизированном языке, каким является Delphi, подобное смешение типа дескриптора и целого выглядит весьма непривлекательно, но, к сожалению, это лучшее из того, что нам предлагают разработчики WinSock API.

Механизм процедур завершения допускает определение статуса операции с помощью функции `WSAGetOverlappedResult`, но ее параметр `fWait` обязательно должен быть равен `False`, потому что события, необходимые для выполнения ожидания, не взводятся, и попытка дождаться окончания операции может привести к блокировке работы нити.

В процедуре завершения допускается вызывать функции, начинающие новую операцию перекрытого ввода-вывода, в том числе и такую же операцию, которая только что завершена. Эта возможность используется в примере, приведенном в листинге 2.73. Пример иллюстрирует работу клиента, который подключается к серверу и получает от него данные в режиме перекрытого ввода-вывода, выполняя параллельно какие-то другие действия.

Листинг 2.73. Перекрытый ввод-вывод с использованием процедуры завершения

```
var
  S: TSocket;
  Overlapped: TWSAOverlapped;
  BufPtr: TWSABuf;
  RecvBuf: array[1..100] of Char;
  Cnt, Flags: Cardinal;
  Connected: Boolean;

procedure GetData(Err, Cnt: DWORD; OvPtr: PWSAOverlapped;
  Flags: DWORD); stdcall;
begin
  if Err <> 0 then
  begin
    // Произошла ошибка. Соединение нужно устанавливать заново
    closesocket(S);
    Connected := False;
  end
```

```
  else
  begin
    // Получены данные, обрабатываем
    ......
    // Запускаем новую операцию перекрытого чтения
    Flags := 0;
    WSARecv(S, @BufPtr, 1, Cnt, Flags, OvPtr, GetData)l
  end;
end;

procedure ProcessConnection;
begin
  // Устанавливаем начальное состояние — сокет не соединен
  Connected := False;
  // Задаем буфер
  BufPtr.Buf := @RecvBuf;
  BufPtr.Len := SizeOf(RecvBuf);
  while True do
  begin
    if not Connected then
    begin
      Connected := True;
      // Создаем и подключаем сокет
      S := socket(AF_INET, SOCK_STREAM, 0);
      connect(S, ...);
      // Запускаем первую для данного сокета операцию чтения
      Flags := 0;
      WSARecv(S, @BufPtr, 1, Cnt, Flags, @Overlapped, GetData);
    end;
    // Позволяем системе выполнить процедуру завершения,
    // если это необходимо
    SleepEx(0, True);
    // Выполняем какие-либо дополнительные действия
    ......
  end;
 end;
```

Основная процедура здесь — `ProcessConnection`. Эта процедура в бесконечном цикле устанавливает соединение, если оно не установлено, дает системе выполнить процедуру завершения, если это требуется, и выполняет какие-либо иные действия, не связанные с получением данных от сокета. Процедура завершения `GetData` получает и обрабатывает данные, а если произошла ошибка, закрывает сокет и сбрасывает флаг `Connected`, что служит для про-

цедуры ProcessConnection сигналом о необходимости установить соединение заново.

Из этого примера хорошо видны достоинства и недостатки процедур завершения. Получение и обработка данных выносится в отдельную процедуру, и это, с одной стороны, позволяет разгрузить основную процедуру, но, с другой стороны, заставляет прибегнуть к глобальным переменным для буфера и сокета.

Для протоколов, не поддерживающих соединение, существует другая функция для перекрытого получения данных — WSARecvFrom. Из названия очевидно, что она позволяет узнать адрес отправителя. Прототип функции WSARecvFrom приведен в листинге 2.74.

Листинг 2.74. Функция WSARecvFrom

```
// ***** Описание на C++ *****
int WSARecvFrom(
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesRecvd,
  LPDWORD lpFlags,
  struct sockaddr FAR *lpFrom,
  LPINT lpFromlen,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);

// ***** Описание на Delphi *****
function WSARecvFrom(
  S: TSocket;
  lpBuffers: PWSABuf;
  dwBufferCount: DWORD;
  var NumberOfBytesRecvd: DWORD;
  var Flags: DWORD;
  lpFrom: PSockAddr;
  lpFromLen: PInteger;
  lpOverlapped: PWSAOverlapped;
  lpCompletionRoutine: TWSAOverlappedCompletionRoutine): Integer;
```

Параметры lpFrom и lpFromLen этой функции, служащие для получения адреса отправителя, эквивалентны соответствующим параметрам функции recvfrom, с которой мы уже хорошо знакомы. В остальном WSARecvFrom ведет себя так же, как WSARecv, поэтому мы не будем останавливаться на ней.

Для отправки данных в режиме перекрытого ввода-вывода существуют функции WSASend и WSASendTo, имеющие следующие прототипы (листинг 2.75).

Листинг 2.75. Функции WSASend и WSASendTo

```
// ***** Описание на C++ *****
int WSASend(
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesSent,
  DWORD dwFlags,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);

int WSASendTo(
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesSent,
  DWORD dwFlags,
  const struct sockaddr FAR *lpTo,
  int iToLen,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);

// ***** Описание на Delphi *****
function WSASend(
  S: TSocket;
  lpBuffers: PWSABuf;
  dwBufferCount: DWORD;
  var NumberOfBytesRecvd: DWORD;
  Flags: DWORD;
  lpOverlapped: PWSAOverlapped;
  lpCompletionRoutine: TWSAOverlappedCompletionRoutine): Integer;

function WSASendTo(
  S: TSocket;
  lpBuffers: PWSABuf;
  dwBufferCount: DWORD;
  var NumberOfBytesRecvd: DWORD;
  Flags: DWORD;
  var AddrTo: TSockAddr;
```

```
  ToLen: Integer;
  lpOverlapped: PWSAOverlapped;
  lpCompletionRoutine: TWSAOverlappedCompletionRoutine): Integer;
```

Если вы разобрались с функциями WSARecv, send и sendto, то смысл параметров функций WSASend и WSASendTo должен быть вам очевиден, поэтому подробно разбирать мы их не будем. Но отметим, что флаги передаются по значению, и функции не могут изменять их.

Потребность в перекрытом вводе-выводе при отправке данных возникает достаточно редко. Но функции WSASend/WSASendTo могут оказаться удобными при подготовке многокомпонентных пакетов, которые, например, имеют фиксированный заголовок и финальную часть. Для таких пакетов можно один раз подготовить буферы с заголовком и с финальной частью и, пользуясь возможностью отправки данных из несвязных буферов, при отправке каждого пакета менять только его среднюю часть.

2.2.10. Сервер, использующий перекрытый ввод-вывод

В этом разделе мы рассмотрим создание сервера на основе перекрытого ввода-вывода на основе процедур завершения (пример кода с использованием событий есть в MSDN в описании функций WSARecv и WSASend). Перекрытый ввод-вывод лучше подходит для обмена в режиме "запрос-ответ", поэтому мы вновь вернемся к первоначальному протоколу, который не предусматривает отправку сервером сообщений по собственному усмотрению. На компакт-диске этот пример называется OverlappedServer.

Как обычно, для каждого соединения создается экземпляр записи TConnection, которая на этот раз выглядит так, как показано в листинге 2.76.

Листинг 2.76. Тип TConnection

```
// Информация о соединении с клиентом:
// ClientSocket — сокет, созданный для взаимодействия с клиентом
// ClientAddr — строковое представление адреса клиента
// MsgSize — длина строки, получаемая от клиента
// Msg — строка, получаемая от клиента или отправляемая ему
// Offset — количество байтов, уже полученных от клиента
//   или отправленных ему на данном этапе
// BytesLeft — сколько байтов осталось получить от клиента
//   или отправить ему на данном этапе
// Overlapped — структура для выполнения перекрытой операции
```

```
PConnection = ^TConnection;
TConnection = record
                ClientSocket: TSocket;
                ClientAddr: string;
                MsgSize: Integer;
                Msg: string;
                Offset: Integer;
                BytesLeft: Integer;
                Overlapped: TWSAOverlapped;
              end;
```

Основное отличие этого варианта типа TConnection от того, что применялся ранее в примерах NonBlockingServer и AsyncSelectServer (см. *разд. 2.1.16* и *2.2.6*, а также листинг 2.31) — это отсутствие поля Phase, которое хранит этап взаимодействия с клиентом. Разумеется, в программе OverlappedServer взаимодействие с клиентом также разбивается на три этапа, но реализуется другой способ для того, чтобы различать этапы — для каждого этапа создается своя процедура завершения.

Примечание

Использование одной процедуры завершения для всех трех этапов и распознавание в ней этапов с помощью поля Phase в случае перекрытого ввода-вывода также возможно. Рекомендуем написать такой вариант сервера в качестве самостоятельного упражнения.

Поле Overlapped содержит структуру TWSAOverlapped, которой программа непосредственно не пользуется, она только передает указатель на эту структуру в функции WSARecv и WSASend. Напомним, что одновременно может выполняться несколько операций перекрытого ввода-вывода, но у каждой из этих операций должен быть свой экземпляр TWSAOverlapped. Так как в нашем случае с одним клиентом в каждый момент времени может выполняться не более одной операции, мы создаем по одному экземпляру TWSAOverlapped на каждого клиента.

Функция для перекрытого подключения клиентов существует — это AcceptEx, с которой мы познакомимся в *разд. 2.2.12*. Но она неудобна при работе совместно с WSARecv и WSASend, особенно в таком строго типизированном языке, как Delphi. Поэтому подключение клиентов мы будем отслеживать с помощью уже опробованной технологии асинхронных сокетов на сообщениях. Код запуска сервера OverlappedServer выглядит идентично коду запуска AsyncSelectServer (см. листинг 2.30): точно так же создается сокет, ставится в режим прослушивания, а затем его событие FD_ACCEPT привязывается к сообщению WM_ACCEPTMESSAGE.

Сам обработчик WM_ACCEPTMESSAGE выглядит теперь следующим образом (листинг 2.77).

Листинг 2.77. Обработчик сообщения WM_ACCEPTMESSAGE

```
procedure TServerForm.WMAcceptMessage(var Msg: TWMSocketMessage);
var
  NewConnection: PConnection;
  // Сокет, который создается для вновь подключившегося клиента
  ClientSocket: TSocket;
  // Адрес подключившегося клиента
  ClientAddr: TSockAddr;
  // Длина адреса
  AddrLen: Integer;
  // Аргумент для перевода сокета в неблокирующий режим
  Arg: u_long;
  // Буфер для операции перекрытого чтения
  Buf: TWSABuf;
  NumBytes, Flags: DWORD;
begin
  // Страхуемся от "тупой" ошибки
  if Msg.Socket <> FServerSocket then
    raise ESocketError.Create(
      'Внутренняя ошибка сервера — неверный серверный сокет');
  // Обрабатываем ошибку на сокете, если она есть
  if Msg.SockError <> 0 then
  begin
    MessageDlg('Ошибка при подключении клиента:#13#10' +
      GetErrorString(Msg.SockError) +
      #13#10'Сервер будет остановлен', mtError, [mbOK], 0);
    ClearConnections;
    closesocket(FServerSocket);
    OnStopServer;
    Exit;
  end;
  // Страхуемся от еще одной "тупой" ошибки
  if Msg.SockEvent <> FD_ACCEPT then
    raise ESocketError.Create(
      'Внутренняя ошибка сервера — неверное событие на сокете');
  AddrLen := SizeOf(TSockAddr);
  ClientSocket := accept(FServerSocket, @ClientAddr, @AddrLen);
  if ClientSocket = INVALID_SOCKET then
  begin
```

```
    // Если произошедшая ошибка — WSAEWOULDBLOCK, это просто означает,
    // что на данный момент подключений нет, а вообще все в порядке,
    // поэтому ошибку WSAEWOULDBLOCK мы просто игнорируем. Прочие же
    // ошибки могут произойти только в случае серьезных проблем,
    // которые требуют остановки сервера.
    if WSAGetLastError <> WSAEWOULDBLOCK then
    begin
      MessageDlg('Ошибка при подключении клиента:#13#10' +
        GetErrorString + #13#10'Сервер будет остановлен',
        mtError, [mbOK], 0);
      ClearConnections;
      closesocket(FServerSocket);
      OnStopServer;
    end;
  end
  else
  begin
    // Новый сокет наследует свойства слушающего сокета.
    // В частности, он работает в асинхронном режиме,
    // и его событие FD_ACCEPT связано с сообщением WM_ACCEPTMESSAGE.
    // Так как нам это совершенно не нужно, отменяем асинхронный
    // режим и делаем сокет блокирующим.
    if WSAAsyncSelect(ClientSocket, Handle, 0, 0) = SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при отмене асинхронного режима ' +
        'подключившегося сокета:'#13#10 + GetErrorString,
        mtError, [mbOK], 0);
      closesocket(ClientSocket);
      Exit;
    end;
    Arg := 0;
    if ioctlsocket(ClientSocket, FIONBIO, Arg) = SOCKET_ERROR then
    begin
      MessageDlg('Ошибка при переводе подключившегося сокета ' +
        'в блокирующий режим:'#13#10 + GetErrorString,
        mtError, [mbOK], 0);
      closesocket(ClientSocket);
      Exit;
    end;
    // Создаем запись для нового подключения и заполняем ее
    New(NewConnection);
    NewConnection.ClientSocket := ClientSocket;
    NewConnection.ClientAddr := Format('%u.%u.%u.%u:%u', [
          Ord(ClientAddr.sin_addr.S_un_b.s_b1),
```

```
        Ord(ClientAddr.sin_addr.S_un_b.s_b2),
        Ord(ClientAddr.sin_addr.S_un_b.s_b3),
        Ord(ClientAddr.sin_addr.S_un_b.s_b4),
        ntohs(ClientAddr.sin_port)]);
    NewConnection.Offset := 0;
    NewConnection.BytesLeft := SizeOf(Integer);
    NewConnection.Overlapped.hEvent := 0;
    // Добавляем запись нового соединения в список
    FConnections.Add(NewConnection);
    AddMessageToLog('Зафиксировано подключение с адреса ' +
      NewConnection.ClientAddr);
    // Начинаем перекрытый обмен с сокетом.
    // Начинаем, естественно, с чтения длины строки.
    // В качестве принимающего буфера используем NewConnection.MsgSize
    Buf.Len := NewConnection.BytesLeft;
    Buf.Buf := @NewConnection.MsgSize;
    Flags := 0;
    if WSARecv(NewConnection.ClientSocket, @Buf, 1, NumBytes, Flags,
      @NewConnection.Overlapped, ReadLenCompleted) = SOCKET_ERROR then
    begin
      if WSAGetLastError <> WSA_IO_PENDING then
      begin
        AddMessageToLog('Клиент ' + NewConnection.ClientAddr +
          ' — ошибка при чтении длины строки: ' + GetErrorString);
        RemoveConnection(NewConnection);
      end;
    end;
  end;
end;
```

После того как сокет для взаимодействия с подключившимся клиентом создан, следует отменить для него асинхронный режим, унаследованный от слушающего сокета, т. к. при перекрытом вводе-выводе этот режим не нужен. Затем, после создания экземпляра `TConnection` и добавления его в список, запускается первая операция перекрытого чтения с помощью функции `WSARecv`. Об окончании этой операции будет сигнализировать вызов функции `ReadLenCompleted`, которая передана в `WSARecv` в качестве параметра.

Как мы уже говорили ранее, в программе OverlappedServer есть три разных функции завершения: `ReadLenCompleted`, `ReadMsgCompleted` и `SendMsgCompleted`. Последовательность работы с ними такая: сначала для чтения длины строки вызывается `WSARecv`, в качестве буфера передается `Connection.MsgSize`, в качестве функции завершения — `ReadLenCompleted` (это мы уже видели в листинге 2.77). Когда вызывается `ReadLenCompleted`, это значит, что операция чтения

уже завершена и прочитанная длина находится в Connection.MsgSize. Поэтому в функции ReadLenCompleted выделяем нужный размер для строки Connection.Msg и запускаем следующую операцию перекрытого чтения — с буфером Connection.Msg и функцией завершения ReadMsgCompleted. В этой функции полученная строка показывается пользователю, формируется ответ, и запускается следующая операция перекрытого ввода-вывода — отправка строки клиенту. В качестве буфера в функцию WSASend передается Connection.Msg, а в качестве функции завершения — SendMsgCompleted. В функции SendMsgCompleted вновь вызывается WSARecv с буфером Connection.MsgSize и функцией завершения ReadLenCompleted, и таким образом сервер возвращается к первому этапу взаимодействия с клиентом.

Описанную простую последовательность действий портит то, что из-за возможной отправки данных по частям можно столкнуться с ситуацией, когда функция завершения вызвана для уведомления о том, что получена или отправлена часть данных. Чтобы получить остальную их часть, необходимо вновь вызвать функцию чтения или записи с той же функцией завершения, а указатель на буфер должен при этом указывать на оставшуюся незаполненной часть переменной, в которую помещаются данные. С учетом этого, а также необходимости обработки ошибок, функции завершения выглядят так, как показано в листинге 2.78.

Листинг 2.78. Функции завершения

```
// Функция ReadLenCompleted используется в качестве функции завершения
// для перекрытого чтения длины строки
procedure ReadLenCompleted(dwError: DWORD; cdTransferred: DWORD;
  lpOverlapped: PWSAOverlapped; dwFlags: DWORD); stdcall;
var
  // Указатель на соединение
  Connection: PConnection;
  // Указатель на буфер
  Buf: TWSABuf;
  // Параметры для WSARecv
  NumBytes, Flags: DWORD;
begin
  // Для идентификации операции в функцию передается указатель
  // на запись TWSAOverlapped. Ищем по этому указателю
  // подходящее соединение в списке FConnections.
  Connection := ServerForm.GetConnectionByOverlapped(lpOverlapped);
  if Connection = nil then
  begin
    ServerForm.AddMessageToLog(
      'Внутренняя ошибка программы — не найдено соединение');
```

```
    Exit;
  end;
  // Проверяем, что не было ошибки
  if dwError <> 0 then
  begin
    ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
      ' — ошибка при чтении длины строки: ' + GetErrorString(dwError));
    ServerForm.RemoveConnection(Connection);
    Exit;
  end;
  // Уменьшаем число оставшихся к чтению байтов
  // на размер полученных данных
  Dec(Connection.BytesLeft, cdTransferred);
  if Connection.BytesLeft < 0 then
  begin
    // Страховка от "тупой" ошибки
    ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
      ' — внутренняя ошибка программы: получено больше байтов, ' +
      'чем ожидалось');
    ServerForm.RemoveConnection(Connection);
  end
  else if Connection.BytesLeft = 0 then
  begin
    // Длина строки прочитана целиком
    if Connection.MsgSize <= 0 then
    begin
      ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
        ' — получена неверная длина строки ' +
        IntToStr(Connection.MsgSize));
      ServerForm.RemoveConnection(Connection);;
      Exit;
    end;
    // Делаем строку нужной длины
    SetLength(Connection.Msg, Connection.MsgSize);
    // Данные пока не прочитаны, поэтому смещение — ноль,
    // осталось прочитать полную длину.
    Connection.Offset := 0;
    Connection.BytesLeft := Connection.MsgSize;
    // Заносим размер буфера и указатель на него в Buf.
    // Данные будут складываться в строку,
    // на которую ссылается Connection.Msg.
    Buf.Len := Connection.MsgSize;
    Buf.Buf := Pointer(Connection.Msg);
```

```
    // Вызываем WSARecv для чтения самой строки
    Flags := 0;
    if WSARecv(Connection.ClientSocket, @Buf, 1, NumBytes, Flags,
      @Connection.Overlapped, ReadMsgCompleted) = SOCKET_ERROR then
    begin
      if WSAGetLastError <> WSA_IO_PENDING then
      begin
        ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
          ' — ошибка при чтении строки: ' + GetErrorString(dwError));
        ServerForm.RemoveConnection(Connection);
      end;
    end;
  end
  else
  begin
    // Connection.BytesLeft < 0 — длина строки
    // прочитана не до конца.
    // Увеличиваем смещение на число прочитанных байтов
    Inc(Connection.Offset, cdTransferred);
    // Формируем буфер для чтения оставшейся части длины
    Buf.Len := Connection.BytesLeft;
    Buf.Buf := PChar(@Connection.MsgSize) + Connection.Offset;
    // Вызываем WSARecv для чтения оставшейся части длины строки
    Flags := 0;
    if WSARecv(Connection.ClientSocket, @Buf, 1, NumBytes, Flags,
      @Connection.Overlapped, ReadMsgCompleted) = SOCKET_ERROR then
    begin
      if WSAGetLastError <> WSA_IO_PENDING then
      begin
        ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
          ' — ошибка при чтении длины строки: ' +
          GetErrorString(dwError));
        ServerForm.RemoveConnection(Connection);
      end;
    end;
  end;
end;

// Функция ReadMsgCompleted используется в качестве функции завершения
// для перекрытого чтения строки.
// Она во многом аналогична функции ReadLenCompleted
procedure ReadMsgCompleted(dwError: DWORD; cdTransferred: DWORD;
  lpOverlapped: PWSAOverlapped; dwFlags: DWORD); stdcall;
```

```
var
  Connection: PConnection;
  Buf: TWSABuf;
  NumBytes, Flags: DWORD;
begin
  Connection := ServerForm.GetConnectionByOverlapped(lpOverlapped);
  if Connection = nil then
  begin
    ServerForm.AddMessageToLog(
      'Внутренняя ошибка программы — не найдено соединение');
    Exit;
  end;
  if dwError <> 0 then
  begin
    ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
      ' — ошибка при чтении строки: ' + GetErrorString(dwError));
    ServerForm.RemoveConnection(Connection);
    Exit;
  end;
  Dec(Connection.BytesLeft, cdTransferred);
  if Connection.BytesLeft < 0 then
  begin
    ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
      ' — внутренняя ошибка программы: получено больше байтов, ' +
      'чем ожидалось');
    ServerForm.RemoveConnection(Connection);
  end
  else if Connection.BytesLeft = 0 then
  begin
    // Строка получена целиком. Выводим ее на экран.
    ServerForm.AddMessageToLog('От клиента ' + Connection.ClientAddr +
      ' получена строка: ' + Connection.Msg);
    // Формируем ответ
    Connection.Msg := AnsiUpperCase(StringReplace(Connection.Msg, #0,
      '#0', [rfReplaceAll]))  + ' (Overlapped server)'#0;
    // Смещение — ноль, осталось отправить полную длину
    Connection.Offset := 0;
    Connection.BytesLeft := Length(Connection.Msg);
    // Формируем буфер из строки Connection.Msg
    Buf.Len := Connection.BytesLeft;
    Buf.Buf := Pointer(Connection.Msg);
    // Отправляем строку
    if WSASend(Connection.ClientSocket, @Buf, 1, NumBytes, 0,
      @Connection.Overlapped, SendMsgCompleted) = SOCKET_ERROR then
```

```
    begin
      if WSAGetLastError <> WSA_IO_PENDING then
      begin
        ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
          ' — ошибка при отправке строки: ' + GetErrorString);
        ServerForm.RemoveConnection(Connection);
      end;
    end;
  end
  else
  begin
    // Connection.BytesLeft < 0 — строка прочитана частично
    Inc(Connection.Offset, cdTransferred);
    // Формируем буфер из непрочитанного остатка строки
    Buf.Len := Connection.BytesLeft;
    Buf.Buf := PChar(Connection.Msg) + Connection.Offset;
    // Читаем остаток строки
    Flags := 0;
    if WSARecv(Connection.ClientSocket, @Buf, 1, NumBytes, Flags,
      @Connection.Overlapped, ReadMsgCompleted) = SOCKET_ERROR then
    begin
      if WSAGetLastError <> WSA_IO_PENDING then
      begin
        ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
          ' — ошибка при чтении строки: ' + GetErrorString);
        ServerForm.RemoveConnection(Connection);
      end;
    end;
  end;
end;

// Функция SendMsgCompleted используется в качестве функции завершения
// для перекрытой отправки строки.
// Во многом она аналогична функции ReadLenCompleted
procedure SendMsgCompleted(dwError: DWORD; cdTransferred: DWORD;
  lpOverlapped: PWSAOverlapped; dwFlags: DWORD); stdcall;
var
  Connection: PConnection;
  Buf: TWSABuf;
  NumBytes, Flags: DWORD;
begin
  Connection := ServerForm.GetConnectionByOverlapped(lpOverlapped);
```

```
if Connection = nil then
begin
  ServerForm.AddMessageToLog(
    'Внутренняя ошибка программы — не найдено соединение');
  Exit;
end;
if dwError <> 0 then
begin
  ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
    ' — ошибка при отправке строки: ' + GetErrorString(dwError));
  ServerForm.RemoveConnection(Connection);
  Exit;
end;
Dec(Connection.BytesLeft, cdTransferred);
if Connection.BytesLeft < 0 then
begin
  ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
    ' — внутренняя ошибка программы: отправлено больше байтов, ' +
    'чем ожидалось');
  ServerForm.RemoveConnection(Connection);
end
else if Connection.BytesLeft = 0 then
begin
  // Строка отправлена целиком. Выводим сообщение об этом.
  ServerForm.AddMessageToLog('Клиенту ' + Connection.ClientAddr +
    ' отправлена строка: ' + Connection.Msg);
  // Очищаем строку, чтобы зря не занимала память
  Connection.Msg := '';
  // Теперь будем снова читать длину строки
  Connection.Offset := 0;
  Connection.BytesLeft := SizeOf(Integer);
  // Читать будем в Connection.MsgSize
  Buf.Len := Connection.BytesLeft;
  Buf.Buf := @Connection.MsgSize;
  Flags := 0;
  if WSARecv(Connection.ClientSocket, @Buf, 1, NumBytes, Flags,
    @Connection.Overlapped, ReadLenCompleted) = SOCKET_ERROR then
  begin
    if WSAGetLastError <> WSA_IO_PENDING then
    begin
      ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
        ' — ошибка при чтении длины строки: ' + GetErrorString);
```

```
        ServerForm.RemoveConnection(Connection);
      end;
    end;
  end
  else
  begin
    // Строка отправлена не полностью
    Inc(Connection.Offset, cdTransferred);
    // Формируем буфер из остатка строки
    Buf.Len := Connection.BytesLeft;
    Buf.Buf := PChar(Connection.Msg) + Connection.Offset;
    if WSASend(Connection.ClientSocket, @Buf, 1, NumBytes, 0,
      @Connection.Overlapped, SendMsgCompleted) = SOCKET_ERROR then
    begin
      if WSAGetLastError <> WSA_IO_PENDING then
      begin
        ServerForm.AddMessageToLog('Клиент ' + Connection.ClientAddr +
          ' — ошибка при отправке строки: ' + GetErrorString);
        ServerForm.RemoveConnection(Connection);
      end;
    end;
  end;
end;
```

Чтобы это все заработало, остался последний штрих: нить нужно время от времени переводить в состояние ожидания. Мы будем это делать, вызывая `SleepEx` с нулевым тайм-аутом по сигналам от таймера.

В получившемся сервере трудно увидеть все преимущества перекрытого ввода-вывода. Это и неудивительно, потому что его главное достоинство — высокая производительность при большом количестве подключений. Перекрытый ввод-вывод ориентирован на создание серверов, интенсивно взаимодействующих с многими клиентами, а на таком маленьком сервере, как OverlappedServer, он выглядит несколько тяжеловесно, хотя и позволяет получить вполне работоспособный вариант.

2.2.11. Многоадресная рассылка

При описании стека протоколов TCP/IP мы упоминали протокол IGMP — дополнение к протоколу IP, позволяющее назначать нескольким узлам групповые адреса. С помощью этого протокола можно группе сокетов назначить один IP-адрес, и тогда все пакеты, отправленные на этот адрес, будут получать все сокеты, входящие в группу. Заметим, что не следует путать группы

сокетов в терминах IGMP, и группы сокетов в терминах WinSock (поддержка групп сокетов в WinSock пока отсутствует, существуют только зарезервированные для этого параметры в некоторых функциях).

Мы уже говорили, что сетевая карта получает все IP-пакеты, которые проходят через ее подсеть, но выбирает из них только те, которые соответствуют назначенному ей MAC- и IP-адресу. Существуют два режима работы сетевых карт. В первом выборка пакетов осуществляется аппаратными средствами карты, во втором — программными средствами драйвера. Аппаратная выборка осуществляется быстрее и не загружает центральный процессор, но ее возможности ограничены. В частности, у некоторых старых карт отсутствует аппаратная поддержка IGMP, поэтому они не могут получать пакеты, отправленные на групповой адрес, без переключения в режим программной выборки. Более современные сетевые карты способны запоминать несколько (обычно 16 или 32) групповых адресов, и, пока количество групповых адресов не превышает этот предел, могут осуществлять аппаратную выборку пакетов с учетом групповых адресов.

Windows 95 и NT 4 используют сетевые карты в режиме программной выборки пакетов. Windows 98 и 2000 и выше по умолчанию устанавливают сетевые карты в режим аппаратной выборки пакетов. При этом Windows 2000 может переключать карту в режим программной выборки, если число групповых адресов, с которых компьютер должен принимать пакеты, превышает ее аппаратные возможности. Windows 98 такой возможностью не обладает, поэтому программа, выполняемая в этой среде, может столкнуться с ситуацией, когда сокет не сможет присоединиться к групповому адресу из-за нехватки аппаратных ресурсов сетевой карты (программа при этом получит ошибку `WSAENOBUFS`).

WinSock предоставляет достаточно широкие возможности по управлению многоадресной рассылкой, но для их использования необходимо, чтобы выбранный сетевой протокол поддерживал все эти возможности. Поддержка многоадресной рассылки протоколом IP достаточно скудна по сравнению, например, с протоколами, применяющимися в сетях ATM. Здесь мы будем рассматривать только те возможности WinSock по поддержке многоадресной рассылки, которые совместимы с протоколом IP.

Протокол TCP не поддерживает многоадресную рассылку, поэтому все, что далее будет сказано, относится только к протоколу UDP. Отметим также, что при многоадресной рассылке через границы подсетей маршрутизаторы должны поддерживать передачу многоадресных пакетов.

Глава "Многоадресная рассылка" в [3], к сожалению, содержит множество неточностей. Далее мы будем обращать внимание на эти неточности, чтобы облегчить чтение этой книги.

Многоадресная рассылка в IP является одноранговой и в плоскости управления, и в плоскости данных (в [3] вместо "одноранговая" употребляется слово "немаршрутизируемая" — видимо, переводчик просто перепутал слова non-rooted и non-routed). Это значит, что все сокеты, участвующие в ней, равноправны. Каждый сокет без каких-либо ограничений может подключиться к многоадресной группе и получать все сообщения, отправленные на групповой адрес. При этом послать сообщение на групповой адрес может любой сокет, в том числе и не входящий в группу. Для групповых адресов протокол IP задействует диапазон от 224.0.0.0 до 239.255.255.255. Часть из этих адресов зарезервирована для стандартных служб, поэтому своим группам лучше назначать адреса, начиная с 225.0.0.0. Кроме того, весь диапазон от 224.0.0.0 до 224.0.0.255 зарезервирован для групповых сообщений, управляющих маршрутизаторами, поэтому сообщения, отправленные на эти адреса, никогда не передаются в соседние подсети.

Есть два варианта осуществления многоадресной рассылки с использованием IP средствами WinSock. Первый реализуется средствами WinSock 1 и жестко привязан к протоколу IP. Второй вариант подразумевает работу с WinSock 2 и осуществляется универсальными, не привязанными к конкретному протоколу средствами.

Если рассылка будет осуществляться средствами WinSock 1, то сокет, участвующий в ней, создается обычным образом — с помощью функции `WSASocket` со стандартным набором флагов или с помощью функции `socket` с обычными параметрами, задаваемыми при создании UDP-сокета. Если же используется WinSock 2, то сокет должен быть создан с указанием его роли в плоскостях управления и данных. Так как многоадресная рассылка в IP является одноранговой, все сокеты, участвующие в ней, могут быть только "листьями", поэтому сокет для рассылки должен создаваться функцией `WSASocket` с указанием флагов `WSA_FLAG_MULTIPOINT_C_LEAF` (4) и `WSA_FLAG_MULTIPOINT_D_LEAF` (16). В [3] на странице 313 написано, что для рассылки средствами WinSock 2 можно создавать сокет функцией `socket` — это неверно. Впрочем, на странице 328 все-таки сказано, что указанные флаги задавать обязательно.

Далее сокет, который планируется добавить в группу, привязывается к любому локальному порту обычным способом — с помощью функции `bind`. Этот шаг ничем не отличается от привязки к адресу обычного сокета, не использующего групповой адрес.

Затем выполняется собственно добавление сокета в группу. В WinSock 1 для этого потребуется функция `setsockopt` с параметром `IP_ADD_MEMBERSHIP`, в качестве уровня следует указать `IPPROTO_IP`. При этом через параметр `optval` передается указатель на запись `ip_mreq`, описанную так, как показано в листинге 2.79.

Листинг 2.79. Тип TIPMreq

```
// ***** Описание на C++ *****
struct ip_mreq {
  struct in_addr imr_multiaddr;
  struct in_addr imr_interface;
}

// ***** Описание на Delphi *****
TIPMreq = packed record
  IMR_MultiAddr: TSockAddr;
  IMR_Interface: TSockAddr
end;
```

Поле IMR_MultiAddr задает групповой адрес, к которому присоединяется сокет. У этой структуры должны быть заполнены поля sin_family (значением AF_INET) и sin_addr. Номер порта здесь указывать не нужно, значение этого поля игнорируется. Поле IMR_Interface определяет адрес сетевого интерфейса, через который будет вестись прием многоадресной рассылки. Если программу устраивает интерфейс, выбираемый системой по умолчанию, значение поля IMR_Interface.sin_addr должно быть INADDR_ANY (на компьютерах с одним сетевым интерфейсом обычно используется именно это значение). Но если у компьютера несколько сетевых интерфейсов, которые связывают его с разными сетями, интерфейс для получения групповых пакетов, выбираемый системой по умолчанию, может быть связан не с той сетью, из которой они реально ожидаются. В этом случае программа может явно указать IP-адрес того интерфейса, через который данный сокет должен принимать групповые пакеты. Как и в поле IMR_MultiAddr, в поле IMR_Interface задействованы только поля sin_family и sin_addr, а остальные поля игнорируются.

Для прекращения членства сокета в группе служит та же функция setsockopt, но с параметром IP_DROP_MEMBERSHIP. Через параметр optval при этом также передается структура ip_mreq, значимые поля которой должны быть заполнены так же, как и при добавлении данного сокета в данную группу.

Несмотря на то, что структура ip_mreq относится к WinSock 1, в модуле WinSock ее описание отсутствует. Константы IP_ADD_MEMBERSHIP и IP_DROP_MEMBERSHIP в этом модуле объявлены, но работать с ними следует с осторожностью, потому что они должны иметь разные значения в WinSock 1 и WinSock 2. В WinSock 1 они должны иметь значения 5 и 6 соответственно, а в WinSock 2 — 12 и 13. Из-за этого нужно внимательно следить, чтобы значения соответствовали той библиотеке, из которой импортируется функция setsockopt: 5 и 6 — для WSock32.dll и 12 и 13 — для WS2_32.dll.

В WinSock 2 для присоединения сокета к группе объявлена функция WSAJoinLeaf, прототип которой приведен в листинге 2.80.

Листинг 2.80. Функция WSAJoinLeaf

```
// ***** Описание на C++ *****
SOCKET WSAJoinLeaf(
  SOCKET s,
  const struct sockaddr FAR *name,
  int namelen,
  LPWSABUF lpCallerData,
  LPWSABUF lpCalleeData,
  LPQOS lpSQOS,
  LPQOS lpGQOS,
  DWORD dwFlags);

// ***** Описание на Delphi *****
function WSAJoinLeaf(
  S: TSocket;
  var Name:TSockAddr;
  NameLen: Integer;
  lpCallerData, lpCalleeData: PWSABuf;
  lpSQOS, lpGQOS: PQOS;
  dwFlags: DWORD): TSocket;
```

Параметры lpCallerData и lpCalleeData задают буферы, в которые помещаются данные, передаваемые и получаемые при присоединении к группе. Протокол IP не поддерживает передачу таких данных, поэтому при его использовании эти параметры должны быть равны nil. Параметры lpSQOS и lpGQOS относятся к качеству обслуживания, которое мы здесь не рассматриваем, поэтому их мы тоже полагаем равными nil.

Параметр S определяет сокет, который присоединяется к группе, Name — адрес группы, NameLen — размер буфера с адресом. Параметр dfFlags определяет, будет ли сокет служить для отправки данных (JL_SENDER_ONLY, 1), для получения данных (JL_RECEIVER_ONLY, 2) или и для отправки, и для получения (JL_BOTH, 4).

Функция возвращает сокет, который создан для взаимодействия с группой. В протоколах типа ATM подключение к группе похоже на установление связи в TCP, и функция WSAJoinLeaf, подобно функции accept, создает новый сокет, подключенный к группе. В случае UDP новый сокет не создается, и функция WSAJoinLeaf возвращает значение переданного ей параметра S.

Номер порта в параметре Name игнорируется. Для получения групповых сообщений используется тот интерфейс, который система назначает для этого по умолчанию.

Чтобы прекратить членство сокета в группе, в которую он был добавлен с помощью WSAJoinLeaf, нужно закрыть его посредством функции closesocket.

Если сокет, для которого вызывается функция WSAJoinLeaf, находится в асинхронном режиме, то при успешном присоединении сокета к группе возникнет событие FD_CONNECT (в [3] написано, что в одноранговых плоскостях управления FD_CONNECT не возникает — это не соответствует действительности). Но в случае ненадежного протокола UDP возникновение этого события говорит лишь о том, что было отправлено IGMP-сообщение, извещающее о включении сокета в группу (это сообщение должны получить все маршрутизаторы сети, чтобы потом правильно передавать групповые сообщения в другие подсети). Однако FD_CONNECT не гарантирует, что это сообщение успешно принято всеми маршрутизаторами.

UPD-сокет, присоединившийся к многоадресной группе, не должен "подключаться" к какому-либо адресу с помощью функции connect или WSAConnect. Соответственно, для отправки данных такой сокет может использовать только sendto и WSASendTo. Сокет, присоединившийся к группе, может отправлять данные на любой адрес, но если используется поддержка качества обслуживания, она работает только при отправке данных на групповой адрес сокета.

Отправка данных на групповой адрес не требует присоединения к группе, причем для сокета, отправляющего данные, нет никакой разницы между отправкой данных на обычный адрес и на групповой. И в том и в другом случае используется функция sendto или WSASendTo (или send/WSASend с предварительным вызовом connect). Никаких дополнительных действий для отправки данных на групповой адрес выполнять не требуется. Порт при этом также указывается. Как мы уже видели, номер порта при добавлении сокета в группу не указывается, но сам сокет перед этим должен быть привязан к какому-либо порту. При отправке группового сообщения его получат только те сокеты, входящие в группу, чей порт привязки совпадает с портом, указанным в адресе назначения сообщения.

Если сокет, отправляющий сообщение на групповой адрес, сам является членом этой группы, он, в зависимости от настроек, может получать или не получать свое сообщение. Это определяется его параметром IP_MULTICAST_LOOP, имеющим тип BOOL. По умолчанию этот параметр равен True — это значит, что сокет будет получать свои собственные сообщения. С помощью функции setsockopt можно изменить значение этого параметра на False, и тогда сокет не будет принимать свои сообщения.

Параметром IP_MULTICAST_LOOP следует пользоваться осторожно, т. к. он не поддерживается в Windows NT 4 и требует Windows 2000 или выше. В Windows 9*x*/ME он тоже не поддерживается (хотя упоминания об этом в MSDN нет).

В *разд. 2.1.4* мы говорили, что каждый IP-пакет в своем заголовке имеет целочисленный параметр TTL (Time To Live). Его значение определяет, сколько маршрутизаторов может пройти данный пакет. По умолчанию групповые пакеты имеют TTL, равный 1, т. е. могут распространяться только в пределах непосредственно примыкающих подсетей. Целочисленный параметр сокета IP_MULTICAST_TTL позволяет программе изменить это значение.

У функции WSAJoinLeaf не предусмотрены параметры для задания адреса сетевого интерфейса, через который следует получать групповые сообщения, поэтому всегда используется интерфейс, выбираемый системой для этих целей по умолчанию. Выбрать интерфейс, который система будет назначать по умолчанию, можно с помощью параметра сокета IP_MULTICAST_IF. Этот параметр имеет тип TSockAddr, причем значимыми полями структуры в данном случае являются sin_family и sin_addr, а значение поля sin_port игнорируется.

Значения констант IP_MULTICAST_IF, IP_MULTICAST_TTL и IP_MULTICAST_LOOP также зависят от версии WinSock. В WinSock 1 они должны быть равны 2, 3 и 4, а в WinSock 2 — 9, 10 и 11 соответственно.

2.2.12. Дополнительные функции

В этом разделе мы рассмотрим некоторые функции, относящиеся в WinSock к дополнительным. В WinSock 1 эти функции вместе со всеми остальными экспортируются библиотекой WSock32.dll, а в WinSock 2 они вынесены в отдельную библиотеку MSWSock.dll (в эту же библиотеку вынесены некоторые устаревшие функции типа EnumProtocols).

Начнем мы знакомство с этими функциями с функции WSARecvEx (которая, кстати, является расширенной версией функции recv, а отнюдь не WSARecv, как это можно заключить из ее названия), имеющей следующий прототип:

```
function WSARecvEx(s: TSocket; var buf; len: Integer;
  var flags: Integer): Integer;
```

Видно, что она отличается от обычной функции recv только тем, что флаги передаются через параметр-переменную вместо значения. В функции WSARecvEx этот параметр не только входной, но и выходной; функция может модифицировать его. Ранее мы познакомились с функцией WSARecv, которая также может модифицировать переданные ей флаги, но условия, при которых эти две функции модифицируют флаги, различаются.

При использовании TCP (а также любого другого потокового протокола) флаги не изменяются функцией, и результат работы `WSARecvEx` эквивалентен результату работы `recv`.

Как мы уже не раз говорили, дейтаграмма UDP должна быть прочитана из буфера сокета целиком. Если в буфере, переданном функции `recv` или `recvfrom`, недостаточно места для получения дейтаграммы, эти функции завершаются с ошибкой. При этом в буфер помещается та часть дейтаграммы, которая может в нем поместиться, а оставшаяся часть дейтаграммы теряется. Функция `WSARecvEx` отличается от `recv` только тем, что в случае, когда размер буфера меньше размера дейтаграммы, она завершается без ошибки (возвращая при этом размер прочитанной части дейтаграммы, т. е. размер буфера) и добавляет флаг `MSG_PARTIAL` к параметру `flags`. Остаток дейтаграммы при этом также теряется. Таким образом, `WSARecvEx` дает альтернативный способ проверки того, что дейтаграмма не поместилась в буфер, и в некоторых случаях этот способ может оказаться удобным.

Если при вызове функции `WSARecvEx` флаг `MSG_PARTIAL` установлен программой, но дейтаграмма поместилась в буфер целиком, функция сбрасывает этот флаг.

В описании функции `WSARecvEx` в MSDN можно прочитать, что если дейтаграмма прочитана частично, то следующий вызов функции позволит прочитать оставшуюся часть дейтаграммы. Это не относится к протоколу UDP и справедливо только по отношению к протоколам типа SPX, в которых одна дейтаграмма может разбиваться на несколько сетевых пакетов и потому возможна ситуация, когда в буфере сокета окажется только часть дейтаграммы. В UDP, напомним, дейтаграмма всегда посылается одним IP-пакетом и помещается в буфер сразу целиком.

Функция `WSARecvEx` не позволяет программе определить, с какого адреса прислана дейтаграмма, а аналога функции `recvfrom` с такими же возможностями в WinSock нет.

Мы уже упоминали о том, что в WinSock 1 существует перекрытый ввод-вывод, но только для систем линии NT. Также в WinSock 1 определена функция `AcceptEx`, которая является более мощным эквивалентом функции `accept`, и позволяет принимать входящие соединения в режиме перекрытого ввода-вывода. В WinSock 1 эта функция не поддерживается в Windows 95, в WinSock 2 она доступна во всех системах. Листинг 2.81 содержит ее прототип.

Листинг 2.81. Функция `AcceptEx`

```
function AcceptEx(
  sListenSocket, sAcceptSocket: TSocket;
```

```
  lpOutputBuffer: Pointer;
  dwReceiveDataLength: DWORD;
  dwLocalAddressLength: DWORD;
  dwRemoteAddressLength: DWORD;
  var lpdwBytesReceived: DWORD;
  lpOverlapped:POverlapped): BOOL;
```

Функция AcceptEx позволяет принять новое подключение со стороны клиента и сразу же получить от него первую порцию данных. Функция работает только в режиме перекрытого ввода-вывода.

Параметр sListenSocket определяет сокет, который должен находиться в режиме ожидания подключения. Параметр sAcceptSocket — сокет, через который будет осуществляться связь с подключившимся клиентом. Напомним, что функции accept и WSAAccept сами создают новый сокет. При использовании же AcceptEx программа должна заранее создать сокет и, не привязывая его к адресу, передать в качестве параметра sAcceptSocket.

Параметр lpOutputBufer задает указатель на буфер, в который будут помещены, во-первых, данные, присланные клиентом, а во-вторых, адреса подключившегося клиента и адрес, к которому привязывается сокет sAcceptSocket. Параметр dwReceiveDataLength задает число байтов в буфере, зарезервированных для данных, присланных клиентом, dwLocalAddressLength — для адреса привязки сокета sAcceptSocket, dwRemoteAddressLength — адреса подключившегося клиента. Если параметр dwReceiveDataLength равен нулю, функция не ждет, пока клиент пришлет данные, и считает операцию завершившейся сразу после подключения клиента, как функция accept. Для адресов нужно резервировать как минимум на 16 байтов больше места, чем реально требуется. Так как размер структуры TSockAddr составляет 16 байтов, на каждый из адресов требуется зарезервировать как минимум 32 байта.

Параметр lpdwBytesReceived используется функцией, чтобы вернуть количество байтов, присланных клиентом.

Параметр lpOverlapped указывает на запись TOverlapped, определенную в модуле Windows следующим образом (листинг 2.82).

Листинг 2.82. Тип TOverlapped

```
POverlapped = ^TOverlapped;
_OVERLAPPED = record
  Internal: DWORD;
  InternalHigh: DWORD;
  Offset: DWORD;
```

```
  OffsetHigh: DWORD;
  hEvent: THandle;
end;
TOverlapped =_OVERLAPPED;
```

Структура TOverlapped используется, в основном, для перекрытого ввода-вывода в файловых операциях. Видно, что она отличается от уже знакомой нам структуры TWSAOverlapped (см. листинг 2.69) только типом параметра hEvent — THandle вместо TWSAEvent. Впрочем, ранее мы уже обсуждали, что TWSAEvent — это синоним THandle, так что можно сказать, что эти структуры идентичны (но компилятор подходит к этому вопросу формально и считает их разными).

Параметр lpOverlapped функции AcceptEx не может быть равным nil, а его поле hEvent должно указывать на корректное событие. Процедуры завершения не предусмотрены. Если на момент вызова функции клиент уже подключился и прислал первую порцию данных (или место для данных в буфере не зарезервировано), AcceptEx возвращает True. Если же клиент еще не подключился, или подключился, но не прислал данные, функция AcceptEx возвращает False, а WSAGetLastError — ERROR_IO_PENDING. Параметр lpdwBytesReceived в этом случае остается без изменений.

Проконтролировать состояние операции можно с помощью функции GetOverlappedResult, которая является аналогом известной нам функции WSAGetOverlappedResult, за исключением того, что использует запись TOverlapped вместо TWSAOverlapped и не предусматривает передачу флагов. С ее помощью можно узнать, завершилась ли операция, а также дождаться ее завершения и узнать, сколько байтов прислано клиентом (функция AcceptEx не ждет, пока клиент заполнит весь буфер, предназначенный для него — для завершения операции подключения достаточно первого пакета).

Если к серверу подключаются некорректно работающие клиенты, которые не присылают данные после подключения, операция может не завершаться очень долго, что будет мешать подключению новых клиентов. MSDN рекомендует при ожидании время от времени с помощью функции getsockopt для сокета sAcceptSocket узнавать значение целочисленного параметра SO_CONNECT_TIME уровня SOL_SOCKET. Этот параметр показывает время в секундах, прошедшее с момента подключения клиента к данному сокету (или –1, если подключения не было). Если подключившийся клиент слишком долго не присылает данных, сокет sAcceptSocket следует закрыть, что приведет к завершению операции, начатой AcceptEx, с ошибкой. После этого можно снова вызывать AcceptEx для приема новых клиентов.

Функция AcceptEx реализована таким образом, чтобы обеспечивать максимальную скорость подключения. Ранее мы говорили, что сокеты, созданные

функциями accept и WSAAccept, наследуют параметры слушающего сокета (например, свойства асинхронного режима). Для повышения производительности сокет sAcceptSocket по умолчанию не получает свойств сокета sListenSocket. Но он может унаследовать их после завершения операции с помощью следующей установки параметра сокета SO_UPDATE_ACCEPT_CONTEXT:

```
setsockopt(sAcceptSocket, SOL_SOCKET, SO_UPDATE_ACCEPT_CONTEXT,
  PChar(@sListenSocket), SizeOf(sListenSocket));
```

На сокет sAcceptedSocket после его подключения к клиенту накладываются ограничения: он может использоваться не во всех функциях WinSock, а только в следующих: send, WSASend, recv, WSARecv, ReadFile, WriteFile, TransmitFile, closesocket и setsockopt, причем в последней — только для установки параметра SO_UPDATE_ACCEPT_CONTEXT.

В WinSock не документируется, в какую именно часть буфера помещаются адрес клиента и принявшего его сокета. Вместо этого предоставляется функция GetAcceptExSockAddrs, прототип которой приведен в листинге 2.83.

Листинг 2.83. Функция GetAcceptExSockAddrs

```
procedure GetAcceptExSockAddrs(
  lpOutputBuffer: Pointer;
  dwReceiveDataLength: DWORD;
  dwLocalAddressLength: DWORD;
  dwRemoteAddressLength: DWORD;
  var LocalSockaddr: PSockAddr;
  var LocalSockaddrLength: Integer;
  var RemoteSockaddr: PSockAddr;
  var RemoteSockaddrLength: Integer);
```

Примечание

В Delphi до 7-й версии включительно модуль WinSock содержит ошибку — параметры LocalSockaddr и RemoteSockaddr функции GetAcceptExSockAddr имеют в нем тип TSockAddr вместо PSockAddr. Из-за этой ошибки функцию GetAcceptExSockAddrs в этих версиях Delphi необходимо самостоятельно импортировать. Следует заметить, что во многих модулях для WinSock 2 от независимых разработчиков объявление этой функции скопировано из стандартного модуля вместе с ошибкой.

Первые четыре параметра функции GetAcceptExSockAddrs определяют буфер, в котором в результате вызова AcceptEx оказались данные от клиента и адреса, и размеры частей буфера, зарезервированных для данных и для адресов. Значения этих параметров должны совпадать со значениями аналогичных

параметров в соответствующем вызове AcceptEx. Через параметр LocalSockaddrs возвращается указатель на то место в буфере, в котором хранится адрес привязки сокета sAcceptSocket, а через параметр LocalSockaddrsLength — длина адреса (16 в случае TCP). Адрес клиента и его длина возвращаются через параметры RemoteSockaddrs и RemoteSockaddrsLength. Следует особенно подчеркнуть, что указатели LocalSockaddrs и RemoteSockaddrs указывают именно на соответствующие части буфера: память для них специально не выделяется и, следовательно, не должна освобождаться, а свою актуальность они теряют при освобождении буфера.

Последняя из дополнительных функций, TransmitFile, служит для передачи файлов по сети. Ее прототип приведен в листинге 2.84.

Листинг 2.84. Функция TransmitFile

```
function TransmitFile(
  hSocket: TSocket;
  hFile: THandle;
  nNumberOfBytesToWrite, nNumberOfBytesPerSend: DWORD;
  lpOverlapped: POverlapped;
  lpTransmitBuffers: PTransmitFileBuffers;
  dwReserved: DWORD): BOOL;
```

Функция TransmitFile отправляет содержимое указанного файла через указанный сокет. При этом допускаются только протоколы, поддерживающие соединение, т. е. использовать данную функцию с UDP-сокетом нельзя. Сокет задается параметром hSocket, файл — параметром hFile. Дескриптор файла обычно получается с помощью функции стандартного API CreateFile. Файл рекомендуется открывать с флагом FILE_FLAG_SEQUENTIAL_SCAN, т. к. это повышает производительность.

Параметр nNumberOfBytesToWrite определяет, сколько байтов должно быть передано (позволяя, тем самым, передавать не весь файл, а только его часть). Если этот параметр равен нулю, передается весь файл.

Функция TransmitFile кладет данные из файла в буфер сокета по частям. Параметр nNumberOfBytesPerSend определяет размер одной порции данных. Он может быть равен нулю — в этом случае система сама определяет размер порции. Этот параметр критичен только в случае дейтаграммных протоколов, потому что при этом размер порции определяет размер дейтаграммы. Для TCP данные, хранящиеся в буфере, передаются в сеть целиком или по частям в зависимости от загрузки сети, готовности принимающей стороны и т. п., а то, какими порциями они попали в буфер, на размер пакета почти не влияет.

Поэтому для TCP-сокета параметр nNumberOfBytesPerSend лучше установить равным нулю.

Параметр lpOverlapped указывает на запись TOverlapped, использующуюся для перекрытого ввода-вывода. Эту структуру мы обсуждали при описании функции AcceptEx. В отличие от AcceptEx, в TarnsmitFile этот параметр может быть равным nil, и тогда операция передачи файла не будет перекрытой.

Если параметр lpOverlapped равен nil, передача файла начинается с той позиции, на которую указывает файловый указатель (для только что открытого файла этот указатель указывает на его начало, а переместить его можно, например, с помощью функции SetFilePointer; также он перемещается при чтении файла с помощью ReadFile). Если же параметр lpOverlapped задан, то передача файла начинается с позиции, заданной значениями полей Offset и OffsetHigh, которые должны содержать соответственно младшую и старшую часть 64-битного смещения стартовой позиции от начала файла.

Параметр lpTransmitBuffers является указателем на запись TtransmitFileBuffers, объявленную так, как показано в листинге 2.85.

Листинг 2.85. Тип TTransmitFileBuffers

```
PTransmitFileBuffers = ^TTransmitFileBuffers;
_TRANSMIT_FILE_BUFFERS = record
  Head: Pointer;
  HeadLength: DWORD;
  Tail: Pointer;
  TailLength: DWORD;
end;
TTransmitFileBuffers = _TRANSMIT_FILE_BUFFERS;
```

С ее помощью можно указывать буферы, содержащие данные, которые должны быть отправлены перед передачей самого файла и после него. Поле Head содержит указатель на буфер, содержащий данные, предназначенные для отправки перед файлом, HeadLength — размер этих данных. Аналогично Tail и TailLength определяют начало и длину буфера с данными, которые передаются после передачи файла. Если передача дополнительных данных не нужна, параметр lpTransmitBuffer может быть равен nil.

Допускается и обратная ситуация: параметр hFile может быть равен нулю, и тогда передаются только данные, определяемые параметром lpTransmitBuffer.

Последний параметр функции TransmitFile в модуле WinSock имеет имя Reserved. В WinSock 1 он и в самом деле был зарезервирован и не имел смысла, но в WinSock 2 через него передаются флаги, управляющие операцией

передачи файла. Мы не будем приводить здесь полный список возможных флагов (он есть в MSDN), а ограничимся лишь самыми важными. Указание флага `TF_USE_DEFAULT_WORKER` или `TF_USE_SYSTEM_THREAD` позволяет повысить производительность при передаче больших файлов, а `TF_USE_KERNEL_APC` — при передаче маленьких файлов. Вообще, при работе с функцией `TransmitFile` чтение файла и передачу данных в сеть осуществляет ядро операционной системы, что приводит к повышению быстродействия по сравнению с использованием `ReadFile` и `send` самой программой.

Функция `TransmitFile` реализована по-разному в серверных версиях Windows NT/2000 и в остальных системах: в серверных версиях она оптимизирована по быстродействию, а в остальных — по количеству необходимых ресурсов.

Данные, переданные функцией `TransmitFile`, удаленная сторона должна принимать обычным образом, с помощью функций `recv`/`WSARecv`.

2.3. Итоги главы

На этом мы заканчиваем рассмотрение WinSock. Многие возможности этого стандарта остались не рассмотренными и даже не упомянутыми. Но для этого существуют книги, подобные [3]. Нашей же основной задачей было последовательное знакомство с базовыми возможностями WinSock API и способам их применения в Delphi.

Следует отметить, что в Delphi не обязательно напрямую использовать WinSock API, чтобы работать с сокетами, т. к. VCL содержит компоненты для этого. Прежде всего это `TServerSocket` и `TClientSocket`, использующие асинхронные сокеты, основанные на оконных сообщениях. Начиная с Delphi 7, к ним добавились компоненты `TTCPServer`, `TTCPClient` и `TUDPSocket`, использующие блокирующие или неблокирующие сокеты. Кроме того, с Delphi поставляется библиотека Indy, которая тоже содержит компоненты для работы с сокетами. Но практика показывает, что освоить эти компоненты без знания особенностей WinSock API очень сложно, так что даже если вы никогда не будете вызывать функции WinSock API явно, а ограничитесь компонентами, информация, изложенная в этой главе, вам все равно пригодится.

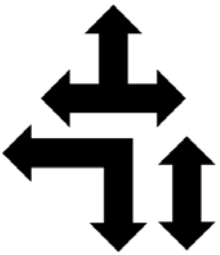
Примечание

Начиная с Delphi 7, компоненты `TClientSocket` и `TServerSocket` в поставке присутствуют, но в палитру компонентов по умолчанию не устанавливаются. Чтобы работать с этими компонентами, их нужно установить самостоятельно. Для этого в меню **Component** следует выбрать пункт **Install Packages**, в открывшемся диалоговом окне нажать кнопку **Add** и добавить нужный пакет. Этот

пакет находится в папке $(DELPHI)/Bin, а название его зависит от версии Delphi. Для Delphi 7 это будет dclsockets70.bpl, для BDS 2005 — dclsockets90.bpl, для BDS 2006, Turbo Delphi и Delphi 2007 — dclsockets100.bpl.

Настоятельно рекомендуем прочитать книгу [3]. Несмотря на незначительные недостатки, она является наиболее полным из изданных на данный момент на русском языке руководством по использованию сокетов в Windows. В крайнем случае рекомендуем хотя бы посмотреть ее содержание в Интернете, чтобы представлять себе, сколько различных возможностей WinSock API остались здесь не упомянутыми.

ГЛАВА 3



"Подводные камни"

Данная глава посвящена "подводным камням" — ситуациям, в которых ошибки или неожиданное поведение программы наиболее вероятны. Другими словами, подводные камни — это то, на чем раз за разом спотыкаются многие начинающие программисты. Не претендуя на описание всех подобных случаев, мы, тем не менее, разберем несколько достаточно характерных примеров. Более полный список можно посмотреть в разделе "Подводные камни" сайта "Королевство Delphi" *(см. приложение 1)*.

Подводные камни можно классифицировать по причинам, вызывающим повышенную вероятность ошибок, следующим образом:

- Аппаратные "камни" — проблемы, вызванные некорректной работой аппаратуры. Наиболее известная из таких проблем — неправильная работа операции деления в блоке FPU первых версий процессора Pentium (в настройках компилятора Delphi можно увидеть опцию **Pentium-safe FDIV** — при ее включении генерируется более медленный, но правильно работающий на (очень) старых процессорах код для вещественного деления). Но подобные проблемы, к счастью, редки, поэтому мы не будем рассматривать их здесь.
- Системные "камни" — проблемы, вызванные тем, что системные функции, которые использует программа, работают не так, как описано в документации, или же у этих функций обнаруживаются особенности работы, вообще не упомянутые в документации.
- "Камни" компилятора — проблемы, вызванные ошибками компилятора Delphi.
- "Камни" VCL — ошибки, содержащиеся в библиотеке VCL. Ранее мы уже упоминали о некоторых из них. Далее мы рассмотрим еще несколько имеющихся в ней ошибок.

- И последний класс "камней" — ошибки, связанные с тем, что программист — человек. Здесь объединены ситуации, когда документация дает исчерпывающее описание проблемы, аппаратура и программные средства работают безукоризненно, но все новые и новые поколения программистов совершают одни и те же ошибки, потому что ситуация кажется им слишком простой и очевидной, чтобы изучать документацию. (Заметим, что это не говорит плохо о таких программистах — человеческая психология имеет свои законы, столь же объективные, как и законы в естественных науках.) Но компьютер — лишь имитация реального мира, и нередко он не оправдывает наших интуитивных ожиданий.

Пункты приведенной классификации не являются взаимоисключающими: далее мы увидим, что некоторые ситуации попадают одновременно под несколько пунктов.

Данная глава посвящена детальному разбору некоторых из подобных ситуаций. Она состоит из четырех разделов. Первый раздел посвящен неочевидным проблемам при работе с целыми числами, второй — при работе с вещественными, в третьем описываются неочевидные моменты использования строк, а в четвертом собрана небольшая коллекция не связанных между собой "подводных камней", с которыми пришлось столкнуться автору книги. Всем ситуациям дано подробное объяснение, чтобы читатель не только запомнил, как делать нельзя, но и понял, почему.

Описание каждого из подводных камней будет сопровождаться примером, который можно найти на прилагаемом компакт-диске. Все примеры (за исключением специально оговоренных случаев) построены следующим образом: на главную (и единственную) форму программы помещаются компоненты `Button1: TButton` и `Label1: TLabel`. Событию `Button1.OnClick` назначается код, демонстрирующий проблему, результат работы кода отображается в `Label1.Caption`. В тексте книги приводится только код этого обработчика.

3.1. Неочевидные особенности целых чисел

Аппаратная реализация целочисленной арифметики достаточно очевидна и в большинстве случаев не приносит неожиданностей. К тому же возможные проблемы в том или ином виде упомянуты во многих книгах по Delphi, поэтому даже начинающий программист обычно готов к ним. В этом разделе мы компактно изложим эти проблемы и объясним причины их появления.

3.1.1. Аппаратное представление целых чисел

Delphi относится к языкам, в которых целые типы данных максимально приближены к аппаратной реализации целых чисел процессором. Это позволяет

выполнять операции с целочисленными данными максимально быстро, но заставляет программиста учитывать аппаратные ограничения.

Примечание

Такая реализация целых чисел может также приводить к проблемам при переносе языка на другую аппаратную платформу, но для Delphi это, видимо, не очень актуально.

Целые числа могут быть знаковыми и беззнаковыми. Сначала рассмотрим формат более простых беззнаковых чисел. Если у нас есть N двоичных разрядов для хранения такого числа, то мы можем представить любое число от 0 до $2^N - 1$. В Delphi беззнаковые целые представлены фундаментальными типами `Byte` ($N = 8$, диапазон 0..255), `Word` ($N = 16$, диапазон 0..65 535) и `LongWord` ($N = 32$, диапазон 0..4 294 967 295).

Примечание

Фундаментальными называются те типы данных, разрядность которых не зависит от аппаратной платформы. Кроме них существуют еще *общие* (generic) типы, разрядность которых определяется разрядностью платформы. В Delphi это типы `Integer` (знаковое целое) и `Cardinal` (беззнаковое целое). В имеющейся реализации они имеют 32 разряда, но при переходе на 64-разрядные компиляторы следует ожидать, что эти типы также станут 64-разрядными. В частности, в 16-разрядном Turbo Pascal тип `Integer` был 16-разрядным (а типа `Cardinal` там не было).

Знаковые числа устроены несколько сложнее. Старший из N бит, отводящихся на такое число, служит для хранения знака (этот бит называется знаковым). Если этот бит равен нулю, число считается положительным, а оставшиеся N–1 разрядов используются для хранения числа так же, как в случае беззнакового целого (эти разряды мы будем называть беззнаковой частью). В этом случае знаковое число ничем не отличается от беззнакового.

Отрицательные значения кодируются несколько сложнее. Когда все разряды (включая знаковый бит) равны единице, это соответствует значению –1. Рассмотрим это на примере однобайтного знакового числа. Числу –1 в данном случае соответствует комбинация 1 1111111 (знаковый бит мы будем отделять от остальных пробелом), т. е. беззнаковая часть числа содержит максимально возможное значение — 127. Числу –2 соответствует комбинация 1 1111110, т. е. в беззнаковой части содержится 126. В общем случае отрицательное число, хранящееся в N разрядах, равно $X - 2^{N-1}$, где X — положительное число, хранящееся в беззнаковой части. Таким образом, N разрядов позволяют представить знаковое целое в диапазоне $-2^{N-1}..2^{N-1} - 1$, причем значению -2^{N-1} соответствует ситуация, когда все биты, кроме знакового, равны нулю.

Такая на первый взгляд не очень удобная система позволяет унифицировать операции для знаковых и беззнаковых чисел. Для примера рассмотрим число 11111110. Если его рассматривать как беззнаковое, оно равно 254, если как знаковое, то –2. Вычитая из него, например, 3, мы должны получить 251 и –5 соответственно. Как нетрудно убедиться, в беззнаковой форме 251 — это 11111011. И число –5 в знаковой форме — это тоже 11111011, т. е. результирующее состояние разрядов зависит только от начального состояния этих разрядов и вычитаемого числа и не зависит от того, знаковое или беззнаковое число представляют эти разряды. И это утверждение справедливо не только для выбранных чисел, но и вообще для любых чисел, если ни они, ни результат операции не выходят за пределы допустимого диапазона. То же самое верно для операции сложения. Поэтому в системе команд процессора нет отдельно команд знакового и беззнакового сложения и вычитания — форматы чисел таковы, что можно обойтись одной парой команд (для умножения и деления это неверно, поэтому существуют отдельно команды знакового и беззнакового умножения и деления).

Ранее мы специально оговорили, что такое удобное правило действует только до тех пор, пока аргументы и результат остаются в рамках допустимого диапазона. Рассмотрим, что произойдет, если мы выйдем за его пределы. Пусть в беззнаковой записи нам нужно из 130 вычесть 10. 130 — это 10000010, после вычитания получим 01111000 (120). Но если попытаться интерпретировать эти двоичные значения как знаковые числа, получится, что из –126 мы вычитаем 10 и получаем 120. Такими парадоксальными результатами приходится расплачиваться за унификацию операций со знаковыми и беззнаковыми числами.

Рассмотрим другой пример: из пяти (в двоичном представлении 00000101) вычесть десять (00001010). Здесь уместно вспомнить вычитание в столбик, которое изучается в школе: если в разряде уменьшаемого стоит цифра, бóльшая, чем в соответствующем разряде вычитаемого, то из старшего разряда уменьшаемого приходится занимать единицу. То же самое и здесь: чтобы вычесть бóльшее число из меньшего, как бы занимается единица из несуществующего девятого разряда. Это можно представить так: из числа (1)00000101 вычитается (0)00001010 и получается (0)11111011 (несуществующий девятый разряд показан в скобках; после получения результата мы про него снова забываем). Если интерпретировать полученный результат как знаковое целое, то он равен –5, т. е. именно тому, что и должно быть. Но с точки зрения беззнаковой арифметики получается, что 5 – 10 = 251.

Приведенные примеры демонстрировали ситуации, когда результат укладывался в один из диапазонов (знаковый или беззнаковый), но не укладывался в другой. Рассмотрим, что будет, если результат не попадает ни в тот, ни в другой диапазон. Пусть нам нужно сложить 10000000 и 10000000. При таком

сложении снова появляется несуществующий девятый разряд, но на этот раз из него единица не занимается, а в него переносится лишняя. Получается (1)00000000. Несуществующий разряд потом игнорируется. С точки зрения знаковой интерпретации получается, что 128 + 128 = 0. С точки зрения беззнаковой — что –128 + (–128) = 0, т. е. оба результата, как и можно было ожидать с самого начала, оказываются некорректными.

Знаковые целые представлены в Delphi типами `ShortInt` (N = 8, диапазон –128..127), `SmallInt` (N = 16, диапазон –32 768..32 767), `LongInt` (N = 32, диапазон –2 147 483 648..2 147 483 647) и `Int64` (N = 64, диапазон –9 223 372 036 854 775 808..9 223 372 036 854 775 807).

Примечание

32-разрядные процессоры не могут выполнять операции непосредственно с 64-разрядными числами, поэтому компилятор генерирует код, который обрабатывает это число по частям. Сначала операция сложения или вычитания выполняется над младшими 32-мя разрядами, а потом — над старшими 32-мя, причем, если в первой операции занималась единица из несуществующего (в рамках данной операции) 33-го разряда, или единица переносилась в него, при второй операции эта единица учитывается.

Далее приведены несколько примеров, иллюстрирующих сказанное.

3.1.2. Выход за пределы диапазона при присваивании

Начнем с рассмотрения простого примера (листинг 3.1, проект Assignment1 на компакт-диске).

Листинг 3.1. Неявное преобразование знакового числа в беззнаковое при присваивании

```
procedure TForm1.Button1Click(Sender: TObject);
var
  X: Byte;
  Y: ShortInt;
begin
  Y := -1;
  X := Y;
  Label1.Caption := IntToStr(X)
end;
```

При выполнении этого примера будет выведено значение 255. Здесь мы сталкиваемся с тем, что все разряды значения `Y` без дополнительных проверок копируются в `X`, но если `Y` интерпретируется как знаковое число, то `X` — как

беззнаковое, а числам 255 и –1 в восьмиразрядном представлении соответствует одна и та же комбинация битов.

Примечание

Промежуточная переменная `Y` понадобилась потому, что прямо присвоить переменной значение, выходящее за ее диапазон, компилятор не позволит — возникнет ошибка компиляции "Constant expression violates subrange bounds".

Строго говоря, в Delphi предусмотрена защита от подобного присваивания. Если включить опцию **Range checking** (включается в окне **Project/Options...** на закладке **Compiler** или директивой компилятора `{$R+}` или `{$RANGECHECKS ON}`), то при попытке присвоения `X := Y` возникнет исключение `ERangeError`. Но по умолчанию эта опция отключена (для повышения производительности — дополнительные проверки требуют процессорного времени), поэтому программа без сообщений об ошибке выполняет такое неправильное присваивание.

В следующем примере (листинг 3.2, проект Assignment2 на компакт-диске) мы рассмотрим присваивание числу такого значения, которое не укладывается ни в знаковый, ни в беззнаковый диапазон.

Листинг 3.2. Присваивание переменной значения, выходящего за рамки диапазона

```
procedure TForm1.Button1Click(Sender: TObject);
var
  X: Byte;
  Y: Word;
begin
  Y := 1618;
  X := Y;
  Label1.Caption := IntToStr(X)
end;
```

На экране появится число 82. Разберемся, почему это происходит. Число 1618 в двоичной записи равно 00000110 01010010. При присваивании этого значения переменной `X` старшие восемь битов "некуда девать", поэтому они просто игнорируются. В результате в `X` записывается число 01010010, т. е. 82.

Разумеется, при включенной опции **Range checking** и в этом случае произойдет исключение `ERangeError`.

Приведенные примеры показывают два основных источника неожиданностей, возникающих при присваивании значения целой переменной:

1. При смешении знаковых и беззнаковых чисел значение меняется из-за того, что старший бит интерпретируется то как знак числа, то как старший разряд.

2. При присваивании переменной значения, требующего большего числа разрядов, "лишние" разряды просто игнорируются.

Все проблемы при присваивании сводятся к одному из этих случаев или к их комбинации.

Все эти ситуации при выключенной опции **Range checking** приводят к ошибкам, которые бывает очень трудно обнаружить. Из-за этого рекомендуется включать эту опцию хотя бы на этапе отладки.

В некоторых случаях возможность присваивания значений, выходящих за пределы диапазона переменной, может быть необходимой (например, для реализации "хитрых" алгоритмов или при сопряжении сторонних библиотек, одна из которых использует знаковые типы, другая — беззнаковые). Чтобы исключение `ERangeError` не возникало, следует предусмотреть явное приведение типа. Например, следующий код работает без исключений при включенной опции **Range checking** (листинг 3.3).

Листинг 3.3. Явное приведение типа для подавления исключений

```
procedure TForm1.Button1Click(Sender: TObject);
var
  X: Byte;
  Y: ShortInt;
begin
  Y := -1;
  X := Byte(Y);
  Label1.Caption := IntToStr(X)
end;
```

В результате его выполнения переменная `X` получает значение 255.

3.1.3. Переполнение при арифметических операциях

Переполнением принято называть ситуацию, когда при операциях над переменной результат выходит за пределы ее диапазона. Рассмотрим следующий пример (листинг 3.4, проект Overflow1 на компакт-диске).

Листинг 3.4. Переполнение при вычитании

```
procedure TForm1.Button1Click(Sender: TObject);
var
  X: Byte;
```

```
begin
  X := 0;
  X := X – 1;
  Label1.Caption := IntToStr(X)
end;
```

Переменная X получит значение 255, поскольку при вычитании получается –1, что в беззнаковом формате соответствует 255. В принципе, этот пример практически эквивалентен примеру Assignment1, за исключением того, что значение –1 появляется в результате арифметических операций.

Немного изменим этот пример — заменим оператор вычитания функцией Dec (листинг 3.5, пример Overflow2 на компакт-диске).

Листинг 3.5. Переполнение при декременте

```
{$R+}

procedure TForm1.Button1Click(Sender: TObject);
var
  X: Byte;
begin
  X := 0;
  Dec(X);
  Label1.Caption := IntToStr(X);
end;
```

Результат получается тот же (X получает значение 255), но обратите внимание: несмотря на то, что опция **Range checking** включена, исключение не возникает. Этим пример Overflow2 отличается от Overflow1 — там исключение возникнет. Связано это с тем, что переполнение при использовании Dec и подобных ей функций контролируется другой опцией — **Overflow checking** (в коде программы включается директивой {$Q+} или {$OVERFLOWCHECKS ON}). Эта опция по умолчанию тоже отключена и ее также рекомендуется включать при отладке. При ее включении в данном примере возникнет исключение EIntOverflow.

3.1.4. Сравнение знакового и беззнакового числа

Посмотрим, что произойдет, если мы попытаемся сравнить знаковое и беззнаковое число (листинг 3.6, пример Compare1 на компакт-диске).

Листинг 3.6. Сравнение "одинаковых" знакового и беззнакового числа

```
procedure TForm1.Button1Click(Sender: TObject);
var
  X: Byte;
  Y: ShortInt;
begin
  Y := -1;
  X := Y;
  if X = Y then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

В окне появится надпись **Не равно**, хотя последовательность битов в переменных `X` и `Y` будет, как мы уже знаем, одинаковая. Надпись соответствует действительности — `X` (255) действительно не равно `Y` (–1). Разберемся, почему так происходит.

Те, кто успел самостоятельно откомпилировать пример Compare1, могли заметить предупреждение компилятора на строке со сравнением: "Comparing signed and unsigned types — widened both operands". Это предупреждение все объясняет: компилятор, зная, что совпадение наборов битов не гарантирует равенство знакового и беззнакового выражения, сначала "расширяет" типы выражений до того типа, чей диапазон целиком вмещает оба требуемых диапазона и лишь затем выполняет сравнение. Это обеспечивает правильный результат сравнения, но требует дополнительных ресурсов, поэтому компилятор выдает предупреждение.

Аналогичные действия компилятор выполнит при сравнении выражений типов `Word` и `SmallInt`, а также `LongInt` и `LongWord`. Тип `Int64` не имеет беззнакового аналога, поэтому операнды этого типа при сравнении компилятор не "расширяет".

Явное приведение типов позволяет избавиться от операций по расширению типа и ограничиться побитовым сравнением (листинг 3.7, пример Compare2 на компакт-диске).

Листинг 3.7. Явное приведение типов при сравнении

```
procedure TForm1.Button1Click(Sender: TObject);
var
  X: Byte;
  Y: ShortInt;
```

```
begin
  Y := -1;
  X := Y;
  if X = Byte(Y) then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

При компиляции такого кода не выдается никаких предупреждений, но и результат сравнения будет неверным: **Равно**.

Примечание

Операции >, <, >= и <= тоже работают по-разному для знаковых и беззнаковых чисел. Пусть, например, сравниваются числа 01000000 и 11000000. В беззнаковом формате это 64 и 192, поэтому первое число меньше второго. А в знаковом это 64 и –64, т. е. первое число больше. Из-за этого для операций сравнения для знаковых и беззнаковых чисел в системе команд процессора существуют разные команды. В литературе, чтобы подчеркнуть это, часто используются различные названия операций в зависимости от формата: для знаковых чисел — "больше" и "меньше", для беззнаковых — "выше" и "ниже".

3.1.5. Неявное преобразование в цикле *for*

Рассмотрим программу (пример ForRange на компакт-диске), на форме которой находятся кнопка и панель, причем кнопка (это важно!) — не на панели, а на форме, а на панели нет никаких компонентов. Обработчик нажатия на кнопку выглядит следующим образом (листинг 3.8).

Листинг 3.8. Обработчик нажатия кнопки

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Cardinal;
begin
  for I := 0 to Panel1.ControlCount – 1 do
    Panel1.Controls[I].Tag := 1;
end;
```

На первый взгляд кажется, что при нажатии на кнопку ничего не должно происходить, т. к. на панели никаких визуальных компонентов нет, и цикл `for` не должен выполниться ни разу. Тем не менее нажатие на кнопку вызывает исключение Access violation.

При нулевом количестве компонентов на панели выражение `Panel1.ControlCount – 1` должно давать значение –1. Но поскольку переменная цикла имеет тип `Cardinal`, эта комбинация битов интерпретируется как 4 294 967 295, верхняя граница оказывается больше нижней, и цикл начинает выполняться, обращаясь к несуществующим элементам управления. Отсюда и ошибка.

Ошибка исчезнет, если тип переменной `I` изменить на `Integer` — в этом случае верхняя граница цикла получит корректное значение –1, и цикл действительно не выполнится ни разу. Если на панели будет находиться хотя бы один компонент, ошибки тоже не будет, потому что верхняя граница цикла не выйдет из диапазона неотрицательных чисел.

Получается, что в ситуации, когда использование беззнакового типа кажется вполне безобидным (действительно, индексы списка не могут быть отрицательными), нас подстерегает "подводный камень", связанный с тем, что верхняя граница цикла может оказаться отрицательной и будет неявно приведена к большому положительному числу. Поэтому в цикле `for` предпочтительнее переменная знакового типа `Integer`, а если по каким-то причинам приходится использовать переменную типа `Cardinal`, то необходимо внимательно следить за тем, какие значения принимают границы типа.

Примечание

Строго говоря, аналогичная проблема может возникнуть и со знаковыми типами, если границы цикла `for` переходят допустимый диапазон этих чисел, просто циклы, переменная которых принимает значения вблизи границ диапазона типа `Integer`, встречаются гораздо реже.

3.2. Неочевидные особенности вещественных чисел

Если рассмотренные в предыдущих разделах особенности целых чисел могли быть неочевидными только начинающим, то вещественные числа могут преподнести сюрпризы даже достаточно опытным программистам, т. к. их поведение существенно дальше от интуитивных представлений, и эти неожиданности не ограничиваются выходом за пределы диапазона. Существующая литература по Delphi, в основном, считает этот вопрос несущественным и обходит его стороной, в результате чего программист, впервые столкнувшийся с одним из таких сюрпризов, впадает в недоумение и испытывает желание "попрыгать вокруг компьютера с бубном". Здесь мы попытаемся восполнить этот пробел и показать, что необъяснимые на первый взгляд явления на самом деле просты и предсказуемы, если известно, как реализуется вещественная арифметика компьютером.

3.2.1. Двоичные дроби

Для начала — немного математики. В школе мы проходим два вида дробей: простые и десятичные. Десятичные дроби, по сути дела, представляют собой разложение числа по степеням десяти. Так, запись 13,6704 означает число, равное $1 \cdot 10^1 + 3 \cdot 10^0 + + 6 \cdot 10^{-1} + 7 \cdot 10^{-2} + 0 \cdot 10^{-3} + 4 \cdot 10^{-4}$. Но внутреннее представление всех чисел в компьютере, в том числе и вещественных, не десятичное, а двоичное. Поэтому он использует двоичные дроби. Они во многом похожи на десятичные, но основанием степени у них служит двойка. Так, двоичная дробь 101.1101 — это $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4}$. В десятичном представлении это число равно 5,8125, в чем нетрудно убедиться с помощью любого калькулятора.

Теперь вспомним научный формат записи десятичного числа. Первым в этой записи идет знак числа (плюс или минус). Дальше идет так называемая *мантисса* (число от 1 до 10). Затем идет *экспонента* (степень десяти, на которую надо умножить мантиссу, чтобы получить нужное число). Итак, уже упоминавшееся число 13,6704 запишется в этом формате как $1{,}36704 \cdot 10^1$ (или 1.36704E1 по принятым в компьютере правилам). Если записываемое число меньше единицы, экспонента будет отрицательной. Аналогичная запись существует и в двоичной системе. Так, 101.1101 запишется в виде $1.011101 \cdot 10^{10}$ (везде использована двоичная форма записи, так что 10^{10} означает 2^2). Именно такое представление реализовано в компьютере. Двоичная точка в такой записи не остается на одном месте, а сдвигается на величину, указанную в экспоненте, поэтому такие числа называются *числами с плавающей точкой* (floating point numbers).

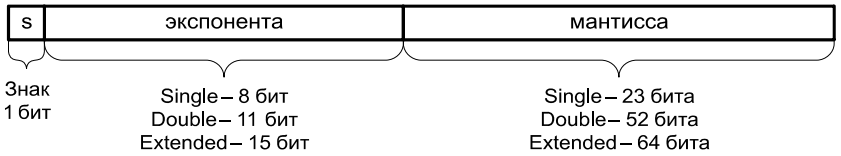
3.2.2. Вещественные типы Delphi

В Delphi существует четыре вещественных типа: `Single`, `Double`, `Extended` и `Real`. Их общий формат одинаков (рис. 3.1, *а*).

Знак — это всегда один бит. Он равен нулю для положительных чисел и единице для отрицательных. Что же касается размеров мантиссы и экспоненты, то именно в них и заключается различие между типами.

Прежде чем перейти к конкретным цифрам, рассмотрим подробнее тип `Real`, сделав для этого небольшой экскурс в историю. `Real` — это стандартный тип языка Паскаль, присутствовавший там изначально. Когда создавался Паскаль, процессоры еще не имели встроенной поддержки вещественных чисел, поэтому все операции с данными типа `Real` сводились к операциям с целыми числами. Соответственно, размер полей в типе `Real` был подобран так, чтобы оптимизировать эти операции.

а) Общий вид вещественного числа



б) Двоичное представление числа типа Single

Биты в числе

s	e7 e6 e5 e4 e3 e3 e1 e0	m22 m21 m20 m19 … m5 m4 m3 m2 m1 m0

Распределение битов по байтам

Номер бита	7	6	5	4	3	2	1	0
Смещение 0	m7	m6	m5	m4	m3	m2	m1	m0
Смещение 1	m15	m14	m13	m12	m11	m10	m9	m8
Смещение 2	e0	m22	m21	m20	m19	m18	m17	m16
Смещение 3	s	e7	e6	e5	e4	e3	e2	e1

Рис. 3.1. Хранение вещественного числа в памяти

Микропроцессор Intel 8086/88 и его улучшенные варианты — 80286 и 80386 — также не имели аппаратной поддержки вещественных чисел. Но у систем на базе этих процессоров была возможность подключения так называемого сопроцессора. Эта микросхема работала с памятью через шины основного процессора и обеспечивала аппаратную поддержку вещественных чисел. В системах средней руки гнездо сопроцессора обычно было пустым, т. к. это уменьшало цену (разумеется, вставить туда сопроцессор не было проблемой). Для каждого центрального процессора выпускались свои сопроцессоры, маркировавшиеся Intel 8087, 80287 и 80387 соответственно. Были даже сопроцессоры, выпускаемые другими фирмами. Они работали быстрее, чем сопроцессоры Intel, но появлялись на рынке позже. Тип вещественных чисел, поддерживаемый сопроцессорами, не совпадает с `Real`. Он определяется стандартом IEEE (Institute of Electrical and Electronics Engineers).

Чтобы обеспечить в своих системах поддержку типов IEEE, Borland вводит в Turbo Pascal типы `Single`, `Double` и `Extended`. `Extended` — это основной для сопроцессора тип, а `Single` и `Double` получаются из него очень простым усечением. Система команд сопроцессора допускает работу с этими типами: при

загрузке числа типа Single или Double во внутренний регистр сопроцессора последний конвертирует их в Extended. Напротив, при выгрузке чисел этих типов из регистра в память сопроцессор усекает их до нужного размера. Внутренние же операции всегда выполняются с данными типа Extended (впрочем, из этого правила есть исключение, на котором мы остановимся позже, после детального рассмотрения формата различных типов). Single и Double позволяют экономить память. Ни один из них также не совпадает с типом Real. В системах с сопроцессорами новые типы обрабатываются заметно (в 2—3 раза) быстрее, чем Real (это с учетом того, что тип Real после соответствующего преобразования также обрабатывался сопроцессором; если же сравнивать обработку типа Extended на машине с сопроцессором и Real на машине без сопроцессора, то там на отдельных операциях достигалась разница в скорости примерно в 100 раз). Чтобы программы с этими типами можно было выполнять и в системах без сопроцессора, была предусмотрена возможность подключать к ним программный эмулятор сопроцессора. Обработка этих типов эмулятором была медленнее, чем обработка Real.

Начиная с 486-й серии Intel берет курс на интеграцию процессора и сопроцессора в одной микросхеме. Процент брака в микросхемах слишком велик, поэтому Intel идет на хитрость: если у микросхемы брак только в сопроцессорной части, то на этом кристалле прожигаются перемычки, блокирующие сопроцессор, и микросхема продается как процессор 80486SX, не имеющий встроенного сопроцессора (в отличие от полноценной версии, которую назвали 80486DX). Бывали и обратные ситуации, когда сопроцессор поврежденный не имел, зато процессор был неработоспособен. Такие микросхемы превращали в "сопроцессор 80487". Но это уже из области экзотики, и, по имеющейся у нас информации, до России такой сопроцессор не дошел.

Процессор Pentium во всех своих вариантах имел встроенный блок вычислений с плавающей точкой (FPU — Floating Point Unit), и отдельный сопроцессор ему не требовался. Таким образом, с приходом этого процессора тип Real остался только для обратной совместимости, а на передний план вышли типы Single, Double и Extended. Начиная с Delphi 4, тип Real становится синонимом типа Double, а старый 6-байтный тип получает название Real48.

Здесь и далее под словом Real мы будем понимать старый 6-байтный тип.

Примечание

Существует директива компилятора {$REALCOMPATIBILITY ON/OFF}, при включении которой (по умолчанию она отключена) Real становится синонимом Real48, а не Double.

Размеры полей для различных вещественных типов указаны в табл. 3.1.

Таблица 3.1. *Размеры полей в вещественных типах*

Тип	Размер типа, байты	Размер мантиссы, биты	Размер экспоненты, биты
`Single`	4	23	8
`Double`	8	52	11
`Extended`	10	64	15
`Real`	6	40	7

Другие параметры вещественных типов, такие как диапазон и точность, можно найти в справке Delphi.

3.2.3. Внутренний формат вещественных чисел

Рассмотрим тип `Single`, т. к. он самый короткий и, следовательно, самый простой для понимания. Остальные типы отличаются от него только количественно. В дальнейшем числа в формате `Single` мы будем записывать как s eeeeeeee mmmmmmmmmmmmmmmmmmmmmmmm, где s означает знаковый бит, e — бит экспоненты, m — бит мантиссы. Порядок хранения битов в типе Single показан на рис. 3.1, *б* (по принятым в процессорах Intel правилам байты в многобайтных значениях переставляются так, что младший байт идет первым, а старший — последним, и вещественных чисел это тоже касается). В мантиссе хранится двоичное число. Чтобы получить истинное значение мантиссы, к ней надо мысленно добавить слева единицу с точкой (т. е., например, мантисса 10100000000000000000 означает двоичную дробь 1.101). Таким образом, имея 23 двоичных разряда, мы записываем числа с точностью до 24-х двоичных разрядов.

Экспонента — по определению всегда целое число. Но способ записи экспоненты в вещественных числах не совпадает с рассмотренным ранее способом записи чисел со знаком. Ноль в этом представлении записывается как 01111111 (в обычном представлении это равно 127). Соответственно, 10000000 (128 в обычном представлении) означает единицу, а 01111110 (126) означает –1, и т. д. (т. е. из обычного беззнакового числа надо вычесть 127, и получится число, закодированное в экспоненте). Такая запись числа называется *нормализованной*.

Из описанных правил есть исключения. Так, если все биты экспоненты равны нулю (т. е. там стоит число –127), то к мантиссе перед ее началом надо добавлять не "1.", а "0." *(денормализованная запись)*. Это позволяет увеличить диапазон вещественных чисел. Если бы этого исключения не было, то минимально возможное положительное число типа `Single` было бы равно пример-

но $5{,}9 \cdot 10^{-39}$. А так появляется возможность использовать числа до $1{,}4 \cdot 10^{-45}$. Побочным эффектом этого является то, что числа, меньшие чем $1{,}17 \cdot 10^{-38}$, представляются с меньшей, чем 24 двоичных разряда, точностью.

Если все биты в экспоненте равны единице, а в мантиссе — нулю, то мы получаем комбинацию, известную как `INF` (от англ. *Infinity* — бесконечность). Эта комбинация используется тогда, когда результат вычислений превышает максимально допустимое форматом число. В зависимости от значения бита `s` бесконечность может быть положительной или отрицательной. Если же при такой экспоненте в мантиссе хоть один бит не равен нулю, такая комбинация называется `NAN` (Not A Number — не число). Попытки работы с комбинациями `NAN` или `INF` приводят к ошибке времени выполнения.

Для задания нуля все биты мантиссы и экспоненты должны быть равны нулю (формально это означает $0 \cdot 10^{-127}$). С учетом описанных правил, если хотя бы один бит экспоненты не будет равен нулю (т. е. экспонента будет больше –127), запись будет считаться нормализованной, и нулевая мантисса будет рассматриваться как единица. Поэтому никакие другие комбинации значений мантиссы и экспоненты не могут дать ноль.

Тип `Double` устроен точно так же, разница только в количестве разрядов и в том, какое значение экспоненты берется за ноль. Итак, мы имеем 11 разрядов для экспоненты. За ноль берется значение 1023.

Несколько иначе устроен `Extended`. Кроме количественных отличий добавляется еще и одно качественное: в мантиссе явно указывается первый разряд. Это означает, что мантисса 1010... интерпретируется как 1.01, а не как 1.101, как это было в типах `Single` и `Float`. Поэтому если 23-битная мантисса типа `Single` обеспечивает 24-знаковую точность, а 52-битная мантисса `Double` — 53-битную, то 64-битная мантисса `Extended` обеспечивает 64-х, а не 65-битную точность. Соответственно, при денормализованной форме записи первый разряд мантиссы явно содержит 0. За ноль экспоненты принимается значение 16 383.

Тип `Real`, как уже упоминалось, стоит особняком. Во-первых, в нем биты следуют в другом порядке, а во-вторых, нет денормализованной формы. Мы не будем касаться внутреннего устройства типа `Real`, т. к. эта информация уже перестала быть актуальной.

3.2.4. "Неполноценный" *Extended*

Ранее мы отметили, что FPU всегда выполняет все операции в формате `Extended`, оговорившись при этом, что есть исключение из этого правила. Здесь мы рассмотрим это исключение.

У FPU существует специальный двухбайтный регистр, называемый *управляющим словом*. Установка отдельных битов этого регистра диктует то или иное поведение при выполнении операций. Прежде всего, это связано с тем, какие исключения может возбуждать FPU. Другие биты этого регистра отвечают за то, как будут округляться числа, как FPU понимает бесконечность, — все это можно при необходимости узнать из документации Intel. Нас же будут интересовать только два бита из этого слова: восьмой и девятый. Именно они определяют, как будут обрабатываться числа внутри сопроцессора.

Если восьмой бит содержит единицу (так установлено по умолчанию), то десять байтов внутренних регистров сопроцессора будут задействованы полностью, и мы получим "полноценный" `Extended`. Если же этот бит равен нулю, то все определяется значением бита 9. Если он равен единице, то используются только 53 разряда мантиссы (остальные всегда равны нулю). Если же этот бит равен нулю — только 24 разряда мантиссы. Это увеличивает скорость вычислений, но уменьшает точность. Другими словами, точность работы сопроцессора может быть понижена до типа `Double` или даже `Single`. Но это касается только мантиссы, экспонента в любом случае будет содержать 15 бит, так что диапазон типа `Extended` сохраняется в любом случае.

Для работы с управляющим словом сопроцессора в модуле `System` описана переменная `Default8087CW` типа `Word` и процедура `Set8087CW(CW: Word)`. При запуске программы в переменную `Default8087CW` записывается то управляющее слово, которое установила система при запуске программы. Функция `Set8087CW` одновременно записывает новое значение в управляющее слово и в переменную `Default8087CW`.

Такое поведение этой функции не всегда удобно — иногда бывает нужно сохранить старое значение переменной `Default8087CW` (впрочем, это несложно сделать, заведя дополнительную переменную). С другой стороны, если значение управляющего слова изменить, не используя `Set8087CW` (а в дальнейшем мы увидим, что такие изменения могут происходить помимо нашей воли), то с помощью функции `Default8087CW` просто нет возможности узнать текущее значение управляющего слова. В Delphi 6 и выше появилась функция `Get8087CW`, позволяющая узнать значение именно контрольного слова, а не переменной `Default8087CW`. В более ранних версиях существовал единственный способ получить значение этого слова — встроенный в Delphi ассемблер.

Итак, установить значение управляющего слова можно с помощью команды `FLDCW`, прочитать — с помощью `FNSTCW`. Обе эти команды имеют один аргумент — переменную типа `Word`. Чтобы, например, установить 53-значную точность, не изменив при этом другие биты управляющего слова, нужно выполнить такую последовательность команд:

```
asm
  FNSTCW  MyCW
  AND     MyCW,0FC00h
  OR      MyCW,200h
  FLDCW   MyCW
end;
```

Начиная с Delphi 6, в модуле `Math` появилась еще одна функция, позволяющая устанавливать точность FPU без манипуляции с отдельными битами управляющего слова — `SetPrecisionMode`. В зависимости от значения аргумента (`pmSingle`, `pmDouble` или `pmExtended`) она устанавливает требуемую точность.

Современные сопроцессоры обрабатывают числа с такой скоростью, что при обычных вычислениях вряд ли может возникнуть необходимость в ускорении за счет точности — выигрыш будет ничтожен. Эта возможность необходима, в основном, в тех случаях, когда вычисления с плавающей точкой составляют значительную часть программы, а высокая точность не имеет принципиального значения (например, в 3D-играх). Однако забывать об этой особенности работы сопроцессора не следует, потому что она может преподнести один неприятный сюрприз, о котором чуть позже.

3.2.5. Бесконечные дроби

Из школы мы все помним, что не каждое число может быть записано конечной десятичной дробью. Бесконечные дроби бывают двух видов: периодичные и непериодичные. Примером непериодичной дроби является число π, периодичной — число 1/3 или любая другая простая дробь, не представимая в виде конечной десятичной дроби.

Примечание

Напомним, что периодичные дроби — это такие дроби, которые содержат бесконечно повторяющуюся последовательность цифр. Например, 1/9=0,11111..., 1/12=0,08333333..., 1/7=0,142857142857... Такие числа записывают со скобками — в них заключают повторяющуюся часть. Те же числа должны быть записаны так: 1/9=0,1(1), 1/12=0,08(3), 1/7=0,1(428571).

Вопрос о периодичности или непериодичности числа нас сейчас не интересует, нам достаточно знать, что не все числа можно представить в виде конечной десятичной дроби. При работе с такими числами мы всегда имеем не точное, а приближенное значение, поэтому ответ получается тоже приближенным. Это нужно учитывать в своих расчетах.

До сих пор мы говорили только о десятичных бесконечных дробях. Но двоичные дроби тоже могут быть бесконечными. Даже более того, любое число, выражаемое конечной двоичной дробью, может быть также выражено и деся-

тичной конечной дробью. Но существуют числа (например, 1/5), которые выражаются конечной десятичной дробью, но не могут быть выражены конечной двоичной дробью. Это и есть наиболее важное отличие аппаратной реализации вещественных чисел от наших интуитивных представлений.

Теперь у нас достаточно теоретических знаний, чтобы перейти к рассмотрению конкретных примеров — "подводных камней", приготовленных вещественными числами.

3.2.6. "Неправильное" значение

Самый первый "подводный камень", на котором спотыкаются новички — это то, что вещественная переменная может получить не совсем то значение, которое ей присвоено. Рассмотрим это на простом примере (листинг 3.9, пример WrongValue на компакт-диске).

Листинг 3.9. Пример присваивания "неправильного" вещественного значения

```
procedure TForm1.Button1Click(Sender: TObject);
var
  R: Single;
begin
  R := 0.1;
  Label1.Caption := FloatToStr(R);
end;
```

Что мы увидим, когда нажмем кнопку? Разумеется, не **0.1**, иначе не было бы смысла писать этот пример. Мы увидим **0.100000001490116**, т. е. расхождение в девятой значащей цифре. Из справки по Delphi мы знаем, что точность типа `Single` — 7—8 десятичных разрядов, так что нас, по крайней мере, никто не обманывает. В чем же причина? Просто число 0,1 не представимо в виде конечной двоичной дроби, оно равно 0,0(0011). И эта бесконечная двоичная дробь обрубается на 24-х знаках; мы получаем не 0,1, а некоторое приближенное число (какое именно — *см. выше*). А если мы присвоим переменной `R` не `0.1`, а `0.5`? Тогда мы получим на экране **0.5**, потому что 0,5 представляется в виде конечной двоичной дроби. Немного поэкспериментировав с различными числами, мы заметим, что точно представляются те числа, которые выражаются в виде $m/2n$, где m, n — некоторые целые числа (разумеется, n не должно превышать 24, а то нам не хватит точности типа `Single`). В качестве упражнения предлагаем доказать, что любое целое число, для записи которого хватает 24-х двоичных разрядов, может быть точно передано типом `Single`.

Примечание

Если в этом примере изменить тип переменной R с Single на Double или Extended, на экран будет выведено **0.1**. Но это не значит, что в переменную будет записано ровно 0,1 — это просто особенности работы функции FloatToStr, которая не учитывает столь малую разницу между 0,1 и переданным ей числом.

3.2.7. Сравнение

Теперь попробуем сравнить значение переменной и константы, которую мы ей присвоили (листинг 3.10, пример Compare1 на компакт-диске).

Листинг 3.10. Пример ошибки при сравнении вещественной переменной и константы

```
procedure TForm1.Button1Click(Sender: TObject);
var
  R: Single;
begin
  R := 0.1;
  if R = 0.1 then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

При нажатии кнопки мы увидим надпись **Не равно**. На первый взгляд это кажется абсурдом. Действительно, мы уже знаем, что переменная R получает значение 0.100000001490116 вместо 0.1. Но ведь "0.1" в правой части равенства тоже должно преобразоваться по тем же законам, т. к. работает аналогичный алгоритм. Тут самое время вспомнить, что FPU работает только с 10-байтным типом Extended, поэтому и левая, и правая часть равенства сначала преобразуется в этот тип, и лишь потом производится сравнение. То число, которое оказалось в переменной R вместо 0.1, хотя и выглядит страшно, но зато представляется в виде конечной двоичной дроби. Информация же о том, что это на самом деле должно означать "0.1", нигде не сохранилась. При преобразовании этого числа в Extended младшие, избыточные по сравнению с типом Single разряды мантиссы просто заполняются нулями, и мы снова получим то же самое число, только записанное в формате Extended. А "0.1" из правой части равенства преобразуется в Extended без промежуточного превращения в Single. Поэтому некоторые из младших разрядов мантиссы будут содержать единицы. Другими словами, мы получим хоть и не точное представление числа 0.1, но все же более близкое к истине, чем 0.100000001490116.

Из-за таких хитрых преобразований оказывается, что мы сравниваем два близких, но все же не равных числа. Отсюда — закономерный результат в виде надписи **Не равно**.

Тут уместна аналогия с десятичными дробями. Допустим, в одном случае мы делим 1 на три с точностью до трех знаков и получаем 0,333. Потом мы делим 1 на три с точностью то четырех знаков и получаем 0,3333. Теперь мы хотим сравнить эти два числа. Для этого приводим их к точности в четыре разряда. Получается, что мы сравниваем 0,3330 и 0,3333. Очевидно, что это разные числа.

Если попробовать заменить число 0,1 на 0,5, то мы увидим надпись **Равно**. Полагаем, что читатели уже догадались, почему, но все же приведем объяснение. Число 0,5 — это конечная двоичная дробь. При прямом приведении ее к типу `Extended` в младших разрядах оказываются нули. Точно такие же нули оказываются в этих разрядах при превращении числа 0,5 типа `Single` в тип `Extended`. Поэтому в результате мы сравниваем два равных числа. Это похоже на процесс деления 1 на 4 с точностью до трех и до четырех значащих цифр. В первом случае получили бы 0,250, во втором — 0,2500. Приведя оба значения к точности в четыре знака, получим сравнение 0,2500 и 0,2500. Очевидно, что эти числа равны.

3.2.8. Сравнение разных типов

Теперь попытаемся сравнить переменную не с константой, а с другой переменной (листинг 3.11, пример Compare2 на компакт-диске).

Листинг 3.11. Пример ошибки при сравнении переменных разных типов

```
procedure TForm1.Button1Click(Sender: TObject);
var
  R1: Single;
  R2: Double;
begin
  R1 := 0.1;
  R2 := 0.1;
  if R1 = R2 then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

Почему этот пример также выдает **Не равно**, понять проще, чем в предыдущем случае. При записи в `R1` бесконечная дробь обрывается на 24-х разрядах,

а при R2 — на 53-х. Таким образом, в дополнительных по сравнению с типом Single разрядах переменной R2 будут единицы. При дополнении значений нулями до 10-байтной точности мы получим разные числа, что и определяет результат сравнения. Это напоминает ситуацию, когда мы сравниваем 0,333 и 0,3333, приводя их к точности в пять знаков: числа 0,33300 и 0,33330 не равны.

Как и в предыдущем случае, замена 0,1 на 0,5 даст результат **Равно**.

3.2.9. Вычитание в цикле

Рассмотрим еще один пример, иллюстрирующий ситуацию, которая часто озадачивает начинающего программиста (листинг 3.12, пример Subtraction на компакт-диске).

Листинг 3.12. Накапливание ошибки при вычитании

```
procedure TForm1.Button1Click(Sender: TObject);
var
  R: Single;
  I: Integer;
begin
  R := 1;
  for I := 1 to 10 do
    R := R – 0.1;
  Label1.Caption := FloatToStr(R);
end;
```

В результате выполнения этого кода на экране появится –7.3015691270939E–8 вместо ожидаемого нуля. Объяснение этому достаточно очевидно, если вспомнить то, о чем мы говорили ранее. Число 0,1 не может быть передано точно ни в одном из вещественных типов, а при каждом вычислении происходит преобразование Single в Extended и обратно, причем последнее — с потерей точности. Эти потери приводят к тому, что мы получаем в результате не ноль, а "почти ноль".

3.2.10. Неожиданная потеря точности

Изменим в предыдущем примере тип переменной R с Single на Double. Значение, выводимое программой, станет **1.44327637948555E–16**. Вполне логичный и предсказуемый результат, т. к. тип Double точнее, чем Single, и, следовательно, все вычисления имеют меньшую погрешность, мы просто обязаны получить более точный результат. Хотя, разумеется, абсолютная точность (т. е. ноль) для нас остается недостижимым идеалом.

А теперь — вопрос на засыпку. Изменится ли результат, если мы заменим `Double` на более точный `Extended`? Ответ не такой однозначный, каким его хотелось бы видеть. В принципе, после такой замены вы должны получить –6.7762635780344E–20. Но в некоторых случаях от замены `Double` на `Extended` результат не изменится, и вы снова получите 1.44327637948555E–16. Это зависит от операционной системы и версии Delphi.

Все дело в использовании "неполноценного" `Extended`. При запуске программы любая система устанавливает такое управляющее слово FPU, чтобы `Extended` был полноценным. Но затем программа вызывает много разных функций Windows API. Какая-то (или какие-то) из этих многочисленных функций некорректно работают с управляющим словом, меняя его значение и не восстанавливая при выходе. Такая проблема встречается, в основном, в Windows 95 и старых версиях Windows 98. Также имеются сведения о том, что управляющее слово может "портиться" и в Windows NT, причем эффект наблюдался не сразу после установки системы, а лишь через некоторое время, после доустановки других программ. Проблема именно в некорректности поведения системных функций; значение управляющего слова, устанавливаемое системой при запуске программы, всегда одинаково.

Таким образом, приходим к неутешительному выводу: к тем проблемам с вещественными числами, которые обусловлены особенностями их аппаратной реализации, добавляются еще и ошибки операционной системы. Правда, радует то, что в последнее время эти ошибки встречаются крайне редко — видимо, новые версии системы от них избавлены. Тем не менее полностью исключать такую возможность нельзя, особенно если ваша программа будет запускаться на старой технике с устаревшими системами. Чтобы приведенный пример всегда выдавал правильное значение –6.7762635780344E–20, достаточно поставить в начале нашей процедуры `Set8087CW(Get8087CW or $0100)`, и программа в любой системе будет устанавливать сопроцессор в режим максимальной точности.

Примечание

В версиях Delphi по 5-ю включительно, где отсутствует функция `Get8087CW`, можно использовать такую конструкцию: `Set8087CW(Default8087CW)`. При этом следует учитывать, что она возвращает к начальному состоянию все флаги, а не только интересующий нас. Если это неприемлемо, управляющее слово придется изменять с помощью встроенного ассемблера.

Раз уж мы заговорили об управляющем слове, давайте немного поэкспериментируем с ним. Изменим первую строчку на `Set8087CW(Get8087CW and $FCFF or $0200)`. Тем самым мы переведем сопроцессор в режим 53-разрядной точности представления мантиссы. Теперь в любой системе мы увидим 1.44327637948555E–16, несмотря на ис-

пользование Extended. Если же мы изменим первую строчку на Set8087CW(Get8087CW and $FCFF), то будем работать в режиме 24-разрядной точности. Соответственно, в любой системе будет результат –7.3015691270939E–8.

Заметим, что при загрузке в 10-байтный регистр сопроцессора числа типа Extended в режиме пониженной точности "лишние" биты не обнуляются. Только результаты математических операций представляются с пониженной точностью. Кроме того, при сравнении двух чисел также учитываются все биты, независимо от точности. Поэтому код, приведенный в листинге 3.10, при выборе любой точности даст **Не равно**.

3.2.11. Борьба с потерей точности в VCL

В том, что описанная проблема с потерей точности встречается все реже, есть заслуга и разработчиков VCL. Зная, вызовы каких функций могут привести к изменению управляющего слова FPU, они перед этими вызовами запоминают управляющее слово, а затем восстанавливают. В более поздних версиях Delphi количество таких "оберток" больше, чем в ранних, поэтому чем новее версия Delphi, тем меньше шанс столкнуться с описанной проблемой. Здесь мы рассмотрим несколько примеров из исходного кода стандартных модулей Delphi 2007.

Для динамической загрузки DLL предназначена API-функция LoadLibrary. В модуле SysUtils для этой функции предлагается обертка, называющаяся SafeLoadLibrary (листинг 3.13).

Листинг 3.13. Функция SysUtils.SafeLoadLibrary

```
{ SafeLoadLibrary calls LoadLibrary, disabling normal Win32 error message
  popup dialogs if the requested file can't be loaded.  SafeLoadLibrary
  also preserves the current FPU control word (precision, exception
  masks) across the LoadLibrary call (in case the DLL you're loading
  hammers the FPU control word in its initialization,
  as many MS DLLs do)}

function SafeLoadLibrary(const Filename: string; ErrorMode: UINT):
  HMODULE;
var
  OldMode: UINT;
  FPUControlWord: Word;
begin
  OldMode := SetErrorMode(ErrorMode);
```

```
  try
    asm
      FNSTCW  FPUControlWord
    end;
    try
      Result := LoadLibrary(PChar(Filename));
    finally
      asm
        FNCLEX
        FLDCW FPUControlWord
      end;
    end;
  finally
    SetErrorMode(OldMode);
  end;
end;
```

Как видно из комментария, проблема в том, что многие системные библиотеки изменяют управляющее слово FPU при своей инициализации.

В функции `CreateADOObject` (внутренняя функция модуля `ADODB`) тоже сохраняется и восстанавливается управляющее слово (листинг 3.14).

Листинг 3.14. Функция `CreateADOObject` модуля `ADODB`

```
function CreateADOObject(const ClassID: TGUID): IUnknown;
var
  Status: HResult;
  FPUControlWord: Word;
begin
  asm
    FNSTCW  FPUControlWord
  end;
  Status := CoCreateInstance(ClassID, nil, CLSCTX_INPROC_SERVER or
    CLSCTX_LOCAL_SERVER, IUnknown, Result);
  asm
    FNCLEX
    FLDCW FPUControlWord
  end;
  if (Status = REGDB_E_CLASSNOTREG) then
    raise Exception.CreateRes(@SADOCreateError) else
    OleCheck(Status);
end;
```

Здесь восстанавливать управляющее слово приходится после вызова системной функции CoCreateInstance, создающей COM-объект. Но, судя по тому, что больше нигде при вызове CoCreateInstance такой код не используется, проблема не в самой функции, а в тех конкретных ADO-объектах, которые создаются здесь с ее помощью.

Аналогичную защиту можно обнаружить в модуле Dialogs, в методе TCommonDialog.TaskModalDialog. Комментарий к этой защите гласит: "Avoid FPU control word change in NETRAP.dll, NETAPI32.dll, etc".

В модуле Windows особым образом импортируются функции CreateWindow и CreateWindowEx, которые, видимо, тоже были замечены в некорректном обращении с управляющим словом FPU. Вот как, например, выглядит импорт функции CreateWindowEx (листинг 3.15).

Листинг 3.15. Импорт функции CreateWindowEx модулем Windows

```
function _CreateWindowEx(dwExStyle: DWORD; lpClassName: PChar;
  lpWindowName: PChar; dwStyle: DWORD; X, Y, nWidth, nHeight: Integer;
  hWndParent: HWND; hMenu: HMENU; hInstance: HINST; lpParam: Pointer):
  HWND; stdcall; external user32 name 'CreateWindowExA';

function CreateWindowEx(dwExStyle: DWORD; lpClassName: PChar;
  lpWindowName: PChar; dwStyle: DWORD; X, Y, nWidth, nHeight: Integer;
  hWndParent: HWND; hMenu: HMENU; hInstance: HINST; lpParam: Pointer):
  HWND;
var
  FPUCW: Word;
begin
  FPUCW := Get8087CW;
  Result := _CreateWindowEx(dwExStyle, lpClassName, lpWindowName,
    dwStyle, X, Y, nWidth, nHeight, hWndParent, hMenu, hInstance,
    lpParam);
  Set8087CW(FPUCW);
end;
```

Модуль Windows импортирует функцию CreateWindowExA из библиотеки user32.dll, но дает ей измененное название и не показывает ее в своем интерфейсе. Вместо этого он экспортирует другую функцию с названием CreateWindowEx (и аналогичную с названием CreateWindowExA), которая является оберткой над настоящей CreateWindowExA и обеспечивает сохранение значения управляющего слова FPU. Аналогичным способом импортируется и Unicode-вариант функции. Таким образом, стандартные библиотеки обеспечивают вызов безопасного варианта CreateWindowEx в любой программе.

Примечание

В модуле Windows можно обнаружить еще одну интересную деталь: функции CreateWindowA и CreateWindowW из библиотеки user32.dll этим модулем вообще не импортируются. Вместо этого одноименные обертки вызывают импортированные функции _CreateWindowExA и _CreateWindowExW, передавая им 0 в качестве значения параметра dwExStyle.

3.2.12. Машинное эпсилон

Когда мы имеем дело с вычислениями с ограниченной точностью, возникает такой парадокс. Пусть, например, мы считаем с точностью до трех значащих цифр. Прибавим к числу 1,00 число $1{,}00 \cdot 10^{-4}$. Если бы все было честно, мы получили бы 1,0001. Но у нас ограничена точность, поэтому мы вынуждены округлять до трех значащих цифр. В результате получается 1,00. Другими словами, к некоторому числу мы прибавляем другое число, большее нуля, а в результате из-за ограниченной точности мы получаем то же самое число. Наименьшее положительное число, которое при добавлении его к единице дает результат, не равный единице, называется *машинным эпсилон*.

Понятие машинного эпсилон у новичков нередко путается с понятием наименьшего числа, которое может быть записано в выбранном формате. Это неправильно. Машинное эпсилон определяется только размером мантиссы, а минимально возможное число оказывается существенно меньше из-за сдвига плавающей двоичной точки с помощью экспоненты.

Прежде чем искать машинное эпсилон программно, попытаемся найти его из теоретических соображений. Итак, мантисса типа Extended содержит 64 разряда. Чтобы закодировать единицу, старший бит мантиссы должен быть равен 1 (денормализованная запись), остальные биты — нулю. Очевидно, что при такой записи наименьшее из чисел, для которых выполняется условие $x > 1$, получается, когда самый младший бит мантиссы тоже будет равен единице, т. е. $x = 1.00...001$ (в двоичном представлении; между точкой и младшей единицей 62 нуля). Таким образом, машинное эпсилон равно $x - 1$, т. е. 0.00...001. В более привычной десятичной форме записи это будет 2^{-63}, т. е. примерно $1{,}084 \cdot 10^{-19}$.

Листинг 3.16 показывает, как можно найти это число (пример Epsilon на компакт-диске).

Листинг 3.16. Поиск машинного эпсилон

```
procedure TForm1.Button1Click(Sender: TObject);
var
  R: Extended;
  I: Integer;
```

```
begin
  R := 1;
  while 1 + R / 2 > 1 do
    R := R / 2;
  Label1.Caption := FloatToStr(R);
end;
```

Запустив этот код, мы получим на экране **1.0842021724855E–19** в полном соответствии с нашими теоретическими выкладками.

Примечание

В тех системах, где наблюдается описанная проблема с уменьшением точности, программа выдаст **2.22044604925031E–16**. Если вы увидели у себя это число, добавьте код, который переведет FPU в режим максимальной точности.

А теперь изменим тип переменной `R` с `Extended` на `Double`. Результат не изменится. На `Single` — опять не изменится. Но такое поведение лишь на первый взгляд может показаться странным. Давайте подробнее рассмотрим выражение `1 + R / 2 > 1`. Итак, все вычисления (в том числе и сравнение) сопроцессор выполняет с данными типа `Extended`. Последовательность действий такова: число `R` загружается в регистр сопроцессора, преобразуясь при этом к типу `Extended`. Дальше оно делится на 2, а затем к результату прибавляется 1, и все это в `Extended`, никакого обратного преобразования в `Single` или `Double` не происходит. Затем это число сравнивается с единицей. Очевидно, что результат сравнения не должен зависеть от исходного типа `R`, т. к. диапазона даже типа `Single` вполне хватает, чтобы разместить машинное эпсилон.

3.2.13. Методы решения проблем

Подведем итоги сказанному. Значения, которые мы получаем, могут отличаться от ожидаемых, даже если речь идет о простом присваивании. Во многих случаях (например, в научных расчетах) это несущественно, т. к. сам метод расчета дает еще бóльшую погрешность. Проблемы начинаются там, где мы хотим вывести число на экран или сравнить его с другим. Универсальных рецептов на все случаи жизни не существует, но во многих ситуациях помогают следующие советы:

- Если ваша задача — просто получить "красивое" представление числа на экране, то функцию `FloatToStr` заменяйте на ее более мощный аналог `FloatToStrF` или на функцию `Format` — они позволяют указать желаемое количество символов после точки.
- Сравнение вещественных чисел следует выполнять с учетом погрешности, т. е. вместо `if a = b ...` писать `if Abs(a – b) < Eps ...`, где `Eps` — неко-

торая величина, задающая допустимую погрешность (в модуле `Math`, начиная с Delphi 6, существует функция `SameValue`, с помощью которой это же условие можно записать как `if SameValue(a, b, Eps) ...`).

- Для денежных расчетов следует выбирать тип `Currency`, реализующий число с фиксированной, а не плавающей, десятичной точкой.

Отметим также, что не следует пытаться решить проблему неточного представления числа (0,100000001490116 вместо 0,1) с помощью функции `RoundTo`, поскольку эта функция не может обеспечить точность бóльшую, чем точность аппаратного представления вещественных чисел.

3.3. Тонкости работы со строками

В этом разделе мы рассмотрим некоторые тонкости работы со строками, которые позволяют лучше понять, какой код генерирует компилятор при некоторых, казалось бы, элементарных действиях. Не все приведенные здесь примеры работают не так, как можно было бы ожидать, так что этот материал немного выходит за рамки главы. Но "подводные камни" здесь мы тоже встретим.

3.3.1. Виды строк в Delphi

Для работы с кодировкой ANSI в Delphi существует три вида строк: `AnsiString`, `ShortString` и `PChar`. Различие между ними заключается в способе хранения строки, а также выделения и освобождения памяти для нее. Зарезервированное слово `string` по умолчанию означает тип `AnsiString`, но если после нее следует число в квадратных скобках, то это означает тип `ShortString`, а число — ограничение по длине. Кроме того, существует опция компилятора **Huge strings** (управляется также директивами компилятора `{$H+/-}` и `{$LONGSTRINGS ON/OFF}`), которая по умолчанию включена, но если ее выключить, то слово `string` станет эквивалентно `ShortString` или, что то же самое, `string[255]`. Эта опция введена для обратной совместимости с Turbo Pascal, в новых программах отключать ее нет нужды. Внутреннее устройство этих типов данных иллюстрирует рис. 3.2.

Наиболее просто устроен тип `ShortString`. Это массив символов с индексами от 0 до N, где N — число символов, указанное при объявлении переменной (в случае использования идентификатора `ShortString` N явно не указывается и равно 255). Нулевой элемент массива хранит текущую длину строки, которая может быть меньше или равна объявленной (эту длину мы будем далее обозначать M), элементы с индексами от 1 до M — это символы, составляющие строку. Значения элементов с индексами $M + 1..N$ не определены. Все

стандартные функции для работы со строками игнорируют эти символы. В памяти такая переменная всегда занимает $N + 1$ байтов.

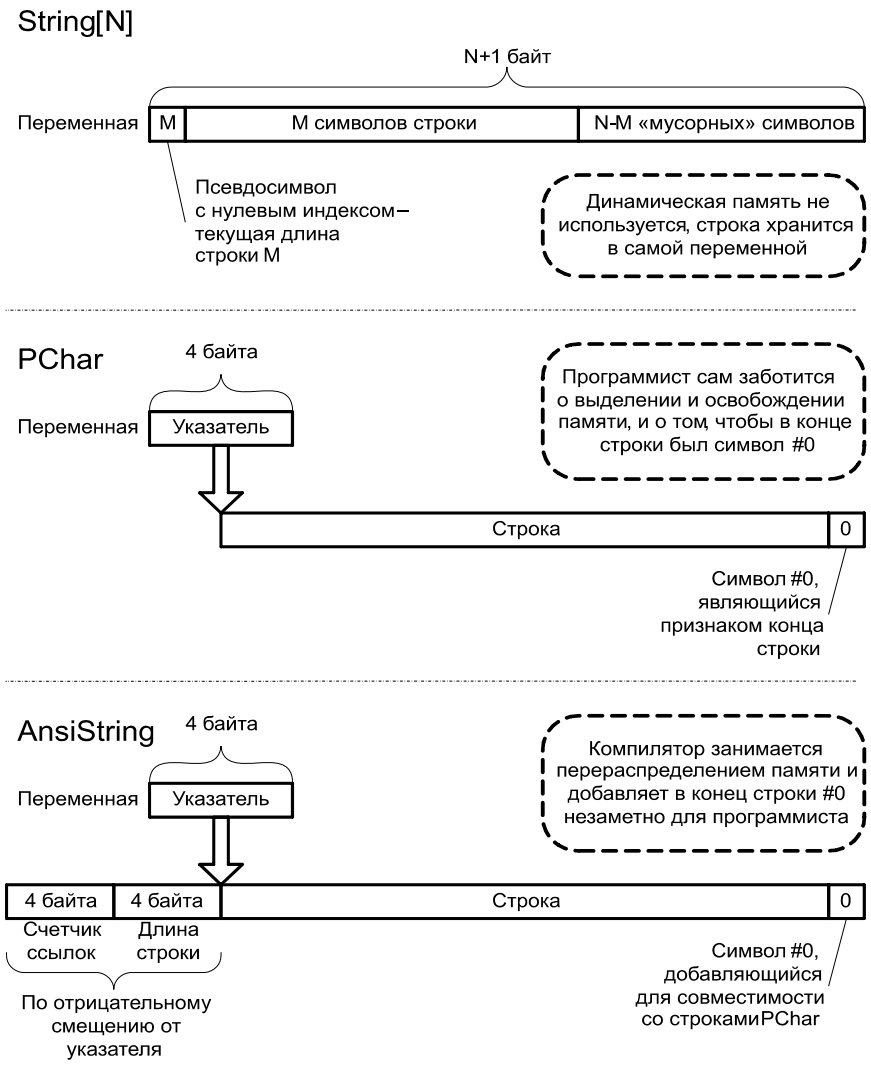


Рис. 3.2. Устройство различных строковых типов Delphi

Ограничения типа `ShortString` очевидны: на хранение длины отводится только один байт, поэтому такая строка не может содержать больше 255 символов. Кроме того, такой способ записи длины не совпадает с принятым в Windows, поэтому `ShortString` несовместим с системными строками.

В системе приняты так называемые нуль-терминированные строки: строка передается указателем на ее первый символ, длина строки отдельно нигде не хранится, признаком конца строки считается встретившийся в цепочке символов #0. Длина таких строк ограничена только доступной памятью и способом адресации (т. е. в Windows теоретически это 4 294 967 295 символов). Для работы с такими строками предусмотрен тип PChar. Переменная такого типа является указателем на начало строки. В литературе нередко можно встретить утверждение, что PChar = ^Char, однако это неверно: тип PChar встроен в компилятор и не выводится из других типов. Это позволяет выполнять с ним операции, недопустимые для других указателей. Во-первых, если P — переменная типа PChar, то допустимо обращение к отдельным символам строки с помощью конструкции P[N], где N — целочисленное выражение, определяющее номер символа (в отличие от типа ShortString, здесь символы нумеруются с 0, а не с 1). Во-вторых, к указателям типа PChar разрешено добавлять и вычитать целые числа, смещая указатель на соответствующее число байтов вверх или вниз (здесь речь идет только об операторах "+" и "–"; адресная арифметика с помощью процедур Inc и Dec доступна для любых типизированных указателей, а не только для PChar).

При работе с PChar программист целиком и полностью отвечает за выделение памяти для строки и за ее освобождение. Именно это и служит основным источником ошибок у новичков: они пытаются работать с такими строками так же, как и с AnsiString, надеясь, что операции с памятью будут выполнены автоматически. Это очень грубая ошибка, способная привести к самым непредсказуемым последствиям.

Хотя программист имеет полную свободу выбора в том, как именно выделять и освобождать память для нуль-терминированных строк, в большинстве случаев самыми удобными оказываются специально предназначенные для этого функции StrNew, StrDispose и т. п. Их преимущество заключается в том, что менеджер памяти выделяет чуть больше места, чем требуется для хранения строки, и в эту дополнительную память записывается, сколько байтов было выделено. Благодаря этому функция StrDispose удаляет ровно столько памяти, сколько было выделено, даже если в середину выделенного блока был записан символ #0, уменьшающий длину строки.

Компилятор также позволяет рассматривать статические массивы типа Char, начинающиеся с нулевого индекса, как нуль-терминированные строки. Такие массивы совместимы с типом PChar, что позволяет обойтись без использования динамической памяти при работе со строками.

Тип AnsiString объединяет достоинства типов ShortString и PChar: строки имеют фактически неограниченную длину, заботиться о выделении памяти для них не нужно, в их конец автоматически добавляется символ #0, что делает их совместимыми с системными строками (впрочем, эта совместимость

не абсолютная; как и когда можно использовать AnsiString в функциях API, мы рассматривали в *разд. 1.1.13*.).

Переменная типа AnsiString — это указатель на первый символ строки, как и в случае PChar. Разница в том, что перед этой строкой в память записывается дополнительная информация: длина строки и счетчик ссылок. Это позволяет компилятору генерировать код, автоматически выделяющий, перераспределяющий и освобождающий память, выделяемую для строки. Работа с памятью происходит совершенно прозрачно для программиста, в большинстве случаев со строками AnsiString можно работать, вообще не задумываясь об их внутреннем устройстве. Символы в таких строках нумеруются с единицы, чтобы облегчить перенос старых программ, использовавших строки типа ShortString.

Счетчик ссылок позволяет реализовать то, что называется copy-on-demand, *копирование по необходимости*. Если у нас есть две переменные S1, S2 типа AnsiString, присваивание вида S1 := S2 не приводит к копированию всей строки. Вместо этого в указатель S1 копируется значение указателя S2, а счетчик ссылок строки увеличивается на единицу. В дальнейшем, если одну из этих строк потребуется модифицировать, она сначала будет скопирована (а счетчик ссылок оригинала, естественно, уменьшен) и только потом изменена, чтобы это не затрагивало остальные переменные.

Далее мы рассмотрим, какие проблемы могут возникнуть при использовании строк разного вида.

3.3.2. Хранение строковых литералов

Литералами называются значения, записываемые в программе буквально. В частности, строковые литералы в Delphi — это последовательности символов, заключенных в кавычки или записанных в виде ANSI-кодов с использованием префикса #.

Когда в программе встречается строковый литерал, компилятор должен поместить его в какую-либо область памяти, чтобы это значение стало доступным программе. Компилятор Delphi размещает строковые литералы в сегменте кода, в участках, управление которым никогда не передается. В данном разделе мы рассмотрим, к каким последствиям это может привести.

Положим на форму пять кнопок и напишем следующие обработчики для нажатия на них (листинг 3.17, пример Constants на компакт-диске).

Листинг 3.17. Примеры работы со строковыми литералами

```
procedure TForm1.Button1Click(Sender: TObject);
var
  P: PChar;
```

```
begin
  P := 'Xest';
  P[0] := 'T'; { * }
  Label1.Caption := P;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  S: string;
  P: PChar;
begin
  S := 'Xest';
  P := PChar(S);
  P[0] := 'T'; { * }
  Label1.Caption := P;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  S: string;
begin
  S := 'Xest';
  S[1] := 'T';
  Label1.Caption := S;
end;

procedure TForm1.Button4Click(Sender: TObject);
var
  S: ShortString;
begin
  S := 'Xest';
  S[1] := 'T';
  Label1.Caption := S;
end;

procedure TForm1.Button5Click(Sender: TObject);
var
  S: ShortString;
  P: PChar;
begin
  S := 'Xest';
  P := @S[1];
  P[0] := 'T';
  Label1.Caption := P;
end;
```

В этом примере только нажатие на третью и четвертую кнопку приводит к появлению надписи **Test**. Первые два обработчика вызывают исключение Access violation в строках, отмеченных звездочками, а при нажатии пятой кнопки программа обычно работает без исключений (хотя в некоторых случаях оно все же может возникнуть), но к слову "Test" добавляется какой-то мусор. Разберемся, почему так происходит.

Встретив в первом обработчике литерал `'Xest'` и определив, что он относится к типу `PChar`, компилятор выделяет в подходящей области сегмента кода пять байтов (четыре значащих символа и один завершающий ноль), а в указатель `P` заносится адрес этого литерала. Сегмент кода доступен только для чтения, прав на его изменение система программе в целях безопасности не дает, поэтому попытка изменить то, что находится в этом сегменте, приводит к закономерному результату — выдаче сообщения "Access violation".

В обработчике второй кнопки происходит почти то же самое, с той лишь разницей, что для литерала выделяется на восемь байтов больше: т. к. в данном случае литерал имеет тип `AnsiString`, ему нужны еще 4 байта для хранения длины и 4 — для счетчика ссылок. В переменную `S` записывается указатель на этот литерал. Приводя эту переменную к типу `PChar`, мы, по сути, просто копируем этот указатель в переменную `P`, а дальше происходит то же самое — попытка изменить страницу памяти, доступную программе только для чтения с тем же самым результатом.

В третьем случае литерал, как и раньше, размещается в сегменте кода. Счетчик ссылок у таких литералов всегда равен –1 — это значение указывает менеджеру памяти, что это константа, которая не может быть изменена и память для которой не нужно освобождать. Поэтому при любой попытке изменить переменную, которой присвоен литерал, срабатывает механизм копирования по необходимости: для строки выделяется место в динамической памяти, затем значение литерала копируется в эту область, обновляется значение указателя `S`, а затем выполняется изменение копии, находящейся в динамической памяти. Так как эта память доступна и для чтения, и для записи, исключение не возникает, и все работает так, как и было задумано.

В четвертом случае литерал также хранится в сегменте кода, но работы с указателем уже нет. Этот литерал занимает там пять байтов: один байт на длину и четыре — на символы. Переменная `S` размещается в стеке, занимая там 256 байтов, а присваивание ей литерала — это копирование значения литерала из сегмента кода в область памяти, занятую переменной. Таким образом, в дальнейшем мы работаем не с константой в сегменте кода, а с ее копией в стеке, которую можно без проблем модифицировать.

В пятом случае мы получаем указатель на этот участок стека. Обратите внимание, что приведение типов в данном случае не работает: для записи в `P` адреса первого символа строки приходится использовать оператор получения

адреса @. Модификация строки проходит, как и в предыдущем случае, успешно, но при присваивании выражения типа PChar свойству типа AnsiString длина строки определяется по правилам, принятым для PChar, т. е. строка сканируется до обнаружения нулевого символа. Но поскольку ShortString "не отвечает" за то, что будет содержаться в неиспользуемых символах, там может остаться всякий мусор от предыдущего использования стека. Никакой гарантии, что сразу после последнего символа будет #0, нет. Отсюда и появление непонятных символов на экране.

Общий вывод таков: пока мы не вмешиваемся в работу компилятора с типами ShortString и AnsiString, получаем ожидаемый результат. Работа с этими же строками через PChar в обход стандартных механизмов приводит к появлению проблем. Кроме того, при работе со строками PChar необходимо четко представлять, где и как выделяется для них память, иначе можно получить неожиданную ошибку.

3.3.3. Приведение литералов к типу *PChar*

В *разд. 1.1.13* мы уже говорили, что когда у функции есть параметр типа PChar, и этот параметр не будет изменяться функцией, при вызове ей можно передавать строковый литерал (см. листинг 1.20). Компилятор размещает литерал в сегменте кода и передает функции указатель на эту память.

В примерах кода, приведенных на различных сайтах, можно нередко встретить такую ситуацию, когда литерал, передаваемый в качестве параметра типа PChar, явно приводится к этому типу. Разберемся, что это дает. Для этого положим на форму четыре кнопки и напишем в обработчиках их нажатия следующий код (листинг 3.18, пример PCharLit на компакт-диске).

Листинг 3.18. Приведение литералов к типу PChar

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Application.MessageBox('Text', nil, 0);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Application.MessageBox('A', nil, 0);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  Application.MessageBox(PChar('Text'), nil, 0);
end;
```

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  Application.MessageBox(PChar('A'), nil, 0);
end;
```

Метод `TApplication.MessageBox` по каким-то непонятным причинам имеет параметры типа `PChar` вместо `string`, и мы этим воспользуемся. При его вызове будет показано диалоговое окно с текстом, переданным в качестве первого параметра (в заголовке будет написано **Ошибка**, т. к. второй параметр у нас `nil`). Нажатие на первую и вторую кнопку не приводит ни к каким неожиданностям — мы видим на экране **Text** и **A** соответственно. Теперь перейдем к коду с явным приведением литерала к `PChar`. Нажатие на третью кнопку к сюрпризам не приведет, а вот нажатие на четвертую даст исключение Access violation.

Происходит это потому, что тип литерала зависит не только от его вида, но и от того, в каком контексте он упомянут. Например, в предыдущем разделе мы видели, что литерал `'Xest'` мог иметь тип `string` или `PChar` в зависимости от того, какой переменной он присваивался. Там, где явного приведения типов нет, тип литерала однозначно определяется по типу формального параметра, и в обработчиках нажатия первых двух кнопок компилятор создает правильные литералы `'Text'` и `'A'` типа `PChar`. Явное приведение литерала к типу `PChar` меняет контекст, в котором литерал упомянут, и компилятор может сделать неправильный вывод о его типе. В обработчике третьей кнопки компилятор правильно понимает, что литерал имеет тип `PChar` и генерирует код, полностью эквивалентный коду обработчика первой кнопки. А вот в случае приведения к типу `PChar` литерала `'A'` компилятор принимает этот литерал не за строковый, а за символьный (т. е. за литерал типа `Char`), состоящий из одного символа без всяких добавлений длины, символа `#0` и т. п. При приведении выражения типа `Char` к любому указателю (в том числе и к `PChar`) оно рассматривается как выражение любого порядкового типа, и его численное значение становится численным значением указателя. В нашем случае это символ с кодом 65 ($41 в шестнадцатеричной записи), поэтому в функцию передается указатель $00000041. Такой указатель указывает на ту область виртуальной памяти, которая никогда не отображается на физическую память, поэтому его использование приводит к ошибке Access violation.

Итак, мы увидели, что явное приведение литерала к типу `PChar` либо никак не отражается на генерируемом компилятором коде (в случае литералов из нескольких символов), либо приводит к генерированию заведомо некорректного кода (в случае односимвольных литералов). Если еще учесть, что приведение литералов к `PChar` загромождает код, легко сделать вывод, что приводить литералы к `PChar` не нужно, поскольку это потенциальный источник проблем и признак плохого оформления кода.

3.3.4. Сравнение строк

Для типов PChar и AnsiString, которые являются указателями, понятие равенства двух строк может толковаться двояко: либо как равенство указателей, либо как равенство содержимого памяти, на которую эти указатели указывают. Второй вариант предпочтительнее, т. к. он ближе к интуитивному понятию равенства строк. Для типа AnsiString реализован именно этот вариант, т. е. сравнивать такие строки можно, ни о чем не задумываясь. Более сложные ситуации мы проиллюстрируем примером Comparisons. В нем одиннадцать кнопок, и обработчик каждой из них иллюстрирует одну из возможных ситуаций.

Начнем со сравнения двух строк типа PChar (листинг 3.19).

Листинг 3.19. Сравнение строк типа PChar

```
procedure TForm1.Button1Click(Sender: TObject);
var
  P1, P2: PChar;
begin
  P1 := StrNew('Test');
  P2 := StrNew('Test');
  if P1 = P2 then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
  StrDispose(P1);
  StrDispose(P2);
end;
```

В данном примере мы увидим надпись **Не равно**. Это происходит потому, что в этом случае сравниваются указатели, а не содержимое строк, а указатели здесь будут разные. Попытка сравнить строки PChar с помощью оператора сравнения — весьма распространенная ошибка у начинающих. Для сравнения таких строк следует применять специальную функцию — StrComp.

Следующий пример, на первый взгляд, в плане сравнения ничем не отличается от только что рассмотренного (листинг 3.20).

Листинг 3.20. Сравнение строк типа PChar, заданных одинаковыми литералами

```
procedure TForm1.Button2Click(Sender: TObject);
var
  P1, P2: PChar;
```

```
begin
  P1 := 'Test';
  P2 := 'Test';
  if P1 = P2 then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

Разница только в том, что строки хранятся не в динамической памяти, а в сегменте кода. Тем не менее на экране появится надпись **Равно**. Это происходит, разумеется, не потому, что сравнивается содержимое строк, а потому, что в данном случае два указателя оказываются равными. Компилятор поступает достаточно интеллектуально: видя, что в разных местах указаны литералы с одинаковым значением, он выделяет для такого литерала место только один раз, а потом помещает в разные указатели один адрес. Поэтому сравнение дает правильный (с интуитивной точки зрения) результат.

Такое положение дел только запутывает ситуацию со сравнением `PChar`: написав подобный тест, человек может сделать вывод, что строки `PChar` сравниваются не по указателю, а по значению, и действовать под руководством этого заблуждения.

Раз уж мы столкнулись с такой особенностью компилятора, немного отвлечемся от сравнения строк и "копнем" этот вопрос немного глубже. В частности, выясним, распространяется ли "интеллект" компилятора на литералы типа `AnsiString` (листинг 3.21).

Листинг 3.21. Сравнение переменных типа `AnsiString` как указателей

```
procedure TForm1.Button3Click(Sender: TObject);
var
  S1, S2: string;
begin
  S1 := 'Test';
  S2 := 'Test';
  if Pointer(S1) = Pointer(S2) then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

В этом примере на экран будет выведено **Равно**. Как мы видим, указатели равны, т. е. и здесь компилятор проявил "интеллект".

Рассмотрим чуть более сложный случай (листинг 3.22).

Листинг 3.22. Сравнение переменных AnsiString и PChar как указателей

```
procedure TForm1.Button4Click(Sender: TObject);
var
  P: PChar;
  S: string;
begin
  S := 'Test';
  P := 'Test';
  if Pointer(S) = P then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

В этом случае указатели окажутся не равны. Действительно, с формальной точки зрения литерал типа AnsiString отличается от литерала типа PChar: в нем есть счетчик ссылок (равный –1) и длина. Однако если забыть о существовании этой добавки, эти два литерала одинаковы: четыре значащих символа и один #0, т. е. компилятор, в принципе, мог бы обойтись одним литералом. Тем не менее на это ему "интеллекта" уже не хватило.

Рассмотрим еще один пример сравнения строк по указателям (листинг 3.23).

Листинг 3.23. Сравнение глобальных переменных типа AnsiString как указателей

```
var
  GS1, GS2: string;

procedure TForm1.Button5Click(Sender: TObject);
begin
  GS1 := 'Test';
  GS2 := 'Test';
  if Pointer(GS1) = Pointer(GS2) then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

Этот пример отличается от приведенного в листинге 3.21 только тем, что теперь переменные глобальные, а не локальные. Однако этого достаточно, что-

бы результат оказался другим — на экране мы увидим надпись **Не равно**. Для глобальных переменных компилятор всегда создает уникальный литерал, на обнаружение одинаковых литералов ему "интеллекта" не хватает. Более того, если поставить точки останова в методах `Button3Click` и `Button4Click`, легко убедиться, что указатель, который будет помещен в переменную `S` в методе `Button4Click`, отличается от того, который будет помещен в переменные `S1` и `S2` в методе `Button3Click`, хотя литералы в обоих случаях одинаковые. Компилятор умеет обнаруживать равенство литералов типа `AnsiString` только в пределах одной функции.

Теперь посмотрим, что будет с глобальными переменными типа `PChar` при присваивании им одинакового литерала (листинг 3.24).

Листинг 3.24. Сравнение глобальных переменных типа PChar

```
var
  GP1, GP2: PChar;

procedure TForm1.Button6Click(Sender: TObject);
begin
  GP1 := 'Test';
  GP2 := 'Test';
  if GP1 = GP2 then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

После выполнения этого кода мы увидим надпись **Равно**, т. е. здесь компилятор смог обнаружить равенство литералов, несмотря на то, что переменные глобальные. Однако переменные типа `PChar`, которым присваиваются одинаковые литералы в разных функциях, как и переменные типа `AnsiString`, получат разные значения.

Но вернемся к сравнению строк. Как мы знаем, строки `AnsiString` сравниваются по значению, а `PChar` — по указателю. А что будет, если сравнить `AnsiString` с `PChar`? Ответ на этот вопрос дает листинг 3.25.

Листинг 3.25. Сравнение переменных типа AnsiString и PChar

```
procedure TForm1.Button7Click(Sender: TObject);
var
  P: PChar;
  S: string;
```

```
begin
  S := 'Test';
  P := 'Test';
  if S = P then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

Этот код выдаст **Равно**. Как мы знаем из предыдущих примеров (см. листинг 3.22), значения указателей не будут равны, следовательно, производится сравнение по содержанию, т. е. именно то, что и требуется. Если исследовать код, который генерирует компилятор, то можно увидеть, что сначала неявно создается строка `AnsiString`, в которую копируется содержимое строки `PChar`, а потом сравниваются две строки `AnsiString`. Сравниваются, естественно, по значению.

Для строк `ShortString` сравнение указателей невозможно, две таких строки всегда сравниваются по значению. Правила хранения литералов и сравнения с другими типами следующие:

1. Литералы типа `ShortString` размещаются в сегменте кода только один раз на одну функцию, сколько бы раз они ни повторялись в ее тексте.
2. При сравнении строк `ShortString` и `AnsiString` первая сначала конвертируется в тип `AnsiString`, а потом выполняется сравнение.
3. При сравнении строк `ShortString` и `PChar` строка `PChar` конвертируется в `ShortString`, затем эти строки сравниваются.

Последнее правило таит в себе "подводный камень", который иллюстрируется следующим примером (листинг 3.26).

Листинг 3.26. Ошибка при сравнении переменных типа `ShortString` и `PChar`

```
procedure TForm1.Button8Click(Sender: TObject);
var
  P: PChar;
  S: ShortString;
begin
  P := StrAlloc(300);
  FillChar(P^, 299, 'A');
  P[299] := #0;
  S[0] := #255;
  FillChar(S[1], 255, 'A');
```

```
  if S = P then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
  StrDispose(P);
end;
```

Здесь формируется строка типа PChar, состоящая из 299 символов "A". Затем формируется строка ShortString, состоящая из 255 символов "A". Очевидно, что эти строки не равны, потому что имеют разную длину. Тем не менее на экране появится надпись **Равно**.

Происходит это вот почему: строка PChar оказывается больше, чем максимально допустимый размер строки ShortString. Поэтому при конвертировании лишние символы просто отбрасываются. Получается строка длиной 255 символов, совпадающая со строкой ShortString, с которой мы ее сравниваем. Отсюда вывод: если строка ShortString содержит 255 символов, а строка PChar — более 255 символов, и ее первые 255 символов совпадают с символами строки ShortString, операция сравнения ошибочно даст положительный результат, хотя эти строки не равны.

Избежать этой ошибки поможет либо явное сравнение длины перед сравнением строк, либо приведение одной из сравниваемых строк к типу AnsiString (второй аргумент при этом также будет приведен к этому типу). Следующий пример (листинг 3.27) дает правильный результат **Не равно**.

Листинг 3.27. Правильное сравнение переменных типа ShortString и PChar

```
procedure TForm1.Button9Click(Sender: TObject);
var
  P: PChar;
  S: ShortString;
begin
  P := StrAlloc(300);
  FillChar(P^, 299, 'A');
  P[299] := #0;
  S[0] := #255;
  FillChar(S[1], 255, 'A');
  if string(S) = P then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
  StrDispose(P);
end;
```

Учтите, что конвертирование в AnsiString — операция дорогостоящая в смысле процессорного времени (в этом примере будут выделены, а потом освобождены два блока памяти), поэтому там, где нужна производительность, целесообразнее вручную сравнить длину, а еще лучше вообще по возможности избегать сравнения строк разных типов, т. к. без конвертирования это в любом случае не обходится.

Теперь зададимся глупым, на первый взгляд, вопросом: если мы приведем строку AnsiString к PChar, будут ли равны указатели? Проверим это (листинг 3.28).

Листинг 3.28. Равенство указателей после приведения AnsiString к PChar

```
procedure TForm1.Button10Click(Sender: TObject);
var
  S: string;
  P: PChar;
begin
  S := 'Test';
  P := PChar(S);
  if Pointer(S) = P then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

Вполне ожидаемый результат — **Равно**. Можно, например, перенести строку из сегмента кода в динамическую память с помощью UniqueString — результат не изменится. Однако выводы делать рано. Рассмотрим следующий пример (листинг 3.29).

Листинг 3.29. Сравнение указателей после приведения пустой строки к PChar

```
procedure TForm1.Button11Click(Sender: TObject);
var
  S: string;
  P: PChar;
begin
  S := '';
  P := PChar(S);
  if Pointer(S) = P then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

От предыдущего он отличается только тем, что строка S имеет пустое значение. Тем не менее на экране мы увидим **Не равно**. Связано это с тем, что приведение строки AnsiString к типу PChar на самом деле не является приведением типов. Это скрытый вызов функции _LStrToPChar, и сделано так для того, чтобы правильно обрабатывать пустые строки.

Значение '' (пустая строка) для строки AnsiString означает, что память для нее вообще не выделена, а указатель имеет значение nil. Для типа PChar пустая строка — это ненулевой указатель на символ #0. Нулевой указатель также может рассматриваться как пустая строка, но не всегда — иногда это рассматривается как отсутствие какого бы то ни было значения, даже пустого (аналог NULL в базах данных). Чтобы решить это противоречие, функция _LStrToPChar проверяет, пустая ли строка хранится в переменной, и, если не пустая, возвращает этот указатель, а если пустая, то возвращает не nil, а указатель на символ #0, который специально для этого размещен в сегменте кода. Таким образом, для пустой строки PChar(S) <> Pointer(S), потому что приведение строки AnsiString к указателю другого типа — это нормальное приведение типов без дополнительной обработки значения.

3.3.5. Побочное изменение

Из-за того, что две одинаковые строки AnsiString разделяют одну область памяти, на неожиданные эффекты можно натолкнуться, если модифицировать содержимое строки в обход стандартных механизмов. Следующий код (листинг 3.30, пример SideChange на компакт-диске) иллюстрирует такую ситуацию.

Листинг 3.30. Побочное изменение переменной S2 при изменении S1

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S1, S2: string;
  P: PChar;
begin
  S1 := 'Test';
  UniqueString(S1);
  S2 := S1;
  P := PChar(S1);
  P[0] := 'F';
  Label1.Caption := S2;
end;
```

В этом примере требует комментариев процедура UniqueString. Она обеспечивает то, что счетчик ссылок на строку будет равен единице, т. е. для этой

строки делается уникальная копия. Здесь это понадобилось для того, чтобы строка S1 хранилась в динамической памяти, а не в сегменте кода, иначе мы получили бы Access violation, как и во втором случае рассмотренного ранее примера Constants (см. листинг 2.17).

В результате работы этого примера на экран будет выведено не **Test**, а **Fest**, хотя значение S2, казалось бы, не должно меняться, потому что изменения, которые мы делаем, касаются только S1. Но более внимательный анализ подсказывает объяснение: после присваивания S2 := S1 счетчик ссылок строки становится равным двум, а сама строка разделяется двумя указателями: S1 и S2. Если бы мы попытались изменить непосредственно S1, то сначала была бы создана копия этой строки, а потом сделаны изменения в этой копии, а оригинал, на который указывала бы S2, остался без изменений. Но, использовав PChar, мы обошли механизм копирования, поэтому строка осталась в единственном экземпляре, и изменения затронули не только S1, но и S2.

В данном примере все достаточно очевидно, но в более сложных случаях разработчик программы может и не подозревать, что строка, с которой он работает, разделяется несколькими переменными. Справка Delphi советует сначала обеспечить уникальность копии строки с помощью UniqueString и только потом работать с ней через PChar, если в этом есть необходимость.

Рассмотрим еще один пример, практически не отличающийся от предыдущего (листинг 3.31).

Листинг 3.31. Отсутствие побочного изменения переменной S2 при изменении S1

```
procedure TForm1.Button2Click(Sender: TObject);
var
  S1, S2: string;
  P: PChar;
begin
  S1 := 'Test';
  UniqueString(S1);
  S2 := S1;
  P := @S1[1];
  P[0] := 'F';
  Label1.Caption := S2;
end;
```

В этом случае на экран будет выведено **Test**, т. е. побочного изменения переменной S2 не произойдет, хотя переменная S1 по-прежнему изменяется в обход стандартных механизмов Delphi.

Вся разница между двумя примерами заключается в том, как получается указатель на строку. В первом примере он является результатом приведения типа

строки к PChar, а во втором — операции взятия адреса первого символа строки. По идее, это должно приводить к одинаковому результату, однако компилятор, зная, что указатель получается, возможно, для того, чтобы с его помощью менять содержимое строки, вставляет сюда неявный вызов UniqueString. В результате этого для S1 выделяется в динамической памяти другая область, чем для S2, и манипуляции с содержимым S1 больше не затрагивают S2.

Неявный вызов UniqueString при обращении к символу строки по индексу выполняется всегда, когда у компилятора есть основания ожидать изменения строки. Это снижает производительность, т. к. многие вызовы UniqueString оказываются излишними. Например, если выполняется посимвольная модификация строки в цикле, UniqueString будет вызываться на каждой итерации цикла, хотя достаточно одного вызова — перед началом цикла. Поэтому в тех случаях, когда производительность критична, посимвольную модификацию строки лучше выполнять низкоуровневыми методами, обращаясь к символам через указатели и обеспечив уникальность строки самостоятельно. Что же касается скорости получения указателя, то тут наиболее быстрым является приведение переменной типа AnsiString к типу Pointer, т. к. это вообще не приводит к генерации дополнительного кода. Приведение к типу PChar работает медленнее потому, что выполняется неявный вызов функции _LStrToPChar, а получение адреса первого символа снижает производительность из-за неявного вызова UniqueString.

Примечание

Еще раз напомним, что низкоуровневые операции с указателями небезопасны в том смысле, что компилятор почти не способен указать разработчику на ошибки в коде, если такие будут. Поэтому применять быстрые низкоуровневые средства доступа к отдельным символам строки следует только тогда, когда в этом действительно есть необходимость.

3.3.6. Нулевой символ в середине строки

Хотя символ #0 и добавляется в конец каждой строки AnsiString, он уже не является признаком ее конца, т. к. длина строки хранится отдельно. Это позволяет размещать символы #0 и в середине строки. Но нужно учитывать, что полноценное преобразование такой строки в PChar невозможно — это иллюстрируется примером Zero на компакт-диске (листинг 3.32).

Листинг 3.32. Потеря остатка строки после символа #0

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S1, S2, S3: string;
  P: PChar;
```

```
begin
  S1 := 'Test'#0'Test';
  S2 := S1;
  UniqueString(S2);
  P := PChar(S1);
  S3 := P;
  Label1.Caption := IntToStr(Length(S2));
  Label2.Caption := IntToStr(Length(S3));
end;
```

В первую метку будет выведено число 9 (длина исходной строки), во вторую — 4. Мы видим, что при копировании одной строки `AnsiString` в другую символ `#0` в середине строки — не помеха (вызов `UniqueString` добавлен для того, чтобы обеспечить реальное копирование строки, а не только копирование указателя). А вот как только мы превращаем эту строку в `PChar`, информация о ее истинной длине теряется, и при обратном преобразовании компилятор ориентируется на символ `#0`, в результате чего строка "обрубается".

Потеря куска строки после символа `#0` происходит всегда, когда есть преобразование `ShortString` или `AnsiString` в `PChar`, даже неявное. Например, все API-функции работают с нуль-терминированными строками, а визуальные компоненты — просто обертки над этими функциями, поэтому вывести с их помощью на экран строку, содержащую `#0`, целиком невозможно.

Но главный "подводный камень", связанный с символом `#0` в середине строки, заключается в том, что целый ряд стандартных функций для работы со строками `AnsiString` на самом деле вызывают API-функции (или даже библиотечные функции Delphi, предназначенные для работы с `PChar`), что приводит к игнорированию "хвоста" после `#0`. Следующий код (листинг 3.33, пример ZeroFind на компакт-диске) иллюстрирует эту проблему.

Листинг 3.33. Некорректная работа функции `AnsiPos` с символом `#0`

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := IntToStr(AnsiPos('Z', 'A'#0'Z'));
end;
```

Хотя символ "Z" присутствует в строке, в которой производится поиск, на экран будет выведен "0", что означает отсутствие искомой подстроки. Это связано с тем, что функция `AnsiPos` использует функции `StrPos` и `CompareString`, предназначенные для работы со строками `PChar`, поэтому поиск за символом `#0` не производится. Если заменить в этом примере функцию `AnsiPos` на `Pos`, которая работает с типом `AnsiString` должным образом, на экран будет выведено правильное значение "3".

Описанные проблемы заставляют очень осторожно относиться к возможному появлению символа #0 в середине строк AnsiString — это может стать источником неожиданных проблем.

3.3.7. Функция, возвращающая *AnsiString*

Очень интересный "подводный камень", связанный с типом AnsiString, рассмотрен в статье [4]. Проиллюстрируем его следующим кодом (листинг 3.34, пример StringResult на компакт-диске).

Листинг 3.34. Неожиданное значение результата

```
function AddOne: string;
begin
  Result := Result + '1';
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  S: string;
begin
  S := 'Test';
  S := AddOne;
  Label1.Caption := S;
end;
```

Если человека, не знакомого с этой особенностью компилятора, попросить предсказать, что появится на экране в результате выполнения этого кода, его рассуждения будут звучать, скорее всего, примерно так: "Так как Result в функции AddOne — это локальная переменная типа string, то, как и все такие переменные, она будет инициализирована пустым значением. Добавление символа '1' к пустой строке даст в результате строку '1', которая и будет выведена на экран. Кстати, на строке S := 'Test' компилятор должен выдать предупреждение, что значение, присвоенное переменной S, нигде не используется".

Однако эти рассуждения неверны. На экране появится надпись **Test1**, т. е. первоначальное значение переменной S будет учтено в функции AddOne. Это происходит потому, что с точки зрения двоичного кода переменная Result это не локальная переменная, а параметр-переменная, как если бы функция AddOne была объявлена так:

```
procedure AddOne(var Result: string);
```

Именно так компилятор обрабатывает функции, тип результата которых AnsiString (и ShortString, кстати, тоже). Какая переменная будет передана в качестве параметра, — это зависит от того, как вызвана функция, точнее, куда идет ее результат. Иногда компилятору приходится неявно имитировать какую-то переменную, а иногда он может воспользоваться реально существующей переменной. В нашем случае он воспользовался переменной S, передав ее в качестве параметра. Строковые параметры-переменные, в отличие от локальных переменных, по понятным причинам не инициализируются пустой строкой, поэтому переменная Result сохраняет значение переменной S, что и приводит к наблюдаемому результату.

Из этого следует правило, которое должен помнить разработчик: функция, возвращающая строковое значение, не должна делать никаких предположений о первоначальном значении переменной Result, т. к. оно может оказаться любым.

Следует заметить, что аналогичным образом компилятор обходится и с другими сложными типами: если функция возвращает такой тип, то Result становится не локальной переменной, а неявным параметром-переменной. Просто с другими типами это не так заметно, потому что от них никто не ожидает автоматической инициализации в прологе функции, и обращаются с переменной Result так, будто она содержит случайный мусор.

3.3.8. Строки в записях

Поля в записях могут иметь любой строковый тип без дополнительных ограничений. Однако следует учитывать, что, в отличие от полей простых типов, значения полей типа PChar и AnsiString лежат вне пределов структуры, причем в случае AnsiString это не так бросается в глаза, т. к. вручную выделять и освобождать память не приходится. Это может привести к неприятному сюрпризу, если работать со структурой как с цельным блоком данных. Чаще всего проблема появляется при записи структуры в поток, файл и т. п. В этом случае записывается только значение указателя, которое не имеет никакого смысла для того, кто потом эти данные читает: такой указатель указывает либо в никуда, либо на данные, никакого отношения к строке не имеющие.

Для иллюстрации этой проблемы, а также методов ее решения нам понадобятся два проекта: RecordRead и RecordWrite (на компакт-диске они оба находятся в папке RecordReadWrite). Обойтись одним проектом здесь нельзя — указатель, переданный в пределах проекта, остается корректным, поэтому проблема маскируется. В проекте RecordWrite три кнопки, соответствующие трем методам сохранения записи в поток TFileStream (в файлы Method1.stm, Method2.stm и Method3.stm соответственно). В три целочисленных поля заносятся текущие час, минута, секунда и сотая доля секунды, строка — произ-

вольная, введенная пользователем в поле ввода Edit1. Файлы пишутся в текущую папку, из-за этого программы нельзя запускать непосредственно с компакт-диска. В проекте RecordRead три кнопки соответствуют трем методам чтения (каждый из своего файла). Сначала рассмотрим первый метод — как делать ни в коем случае нельзя.

В проекте RecordWrite имеем следующий код (листинг 3.35).

Листинг 3.35. Неправильный метод записи структуры со строкой в файл

```
type
  TMethod1Record = packed record
    Hour: Word;
    Minute: Word;
    Second: Word;
    MSec: Word;
    Msg: string;
  end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Rec: TMethod1Record;
  Stream: TFileStream;
begin
  DecodeTime(Now, Rec.Hour, Rec.Minute, Rec.Second, Rec.MSec);
  Rec.Msg := Edit1.Text;
  Stream := TFileStream.Create('Method1.stm', fmCreate);
  Stream.WriteBuffer(Rec, SizeOf(Rec));
  Stream.Free;
end;
```

В проекте RecordRead соответствующий код (листинг 3.36).

Листинг 3.36. Неправильный метод чтения структуры со строкой из файла

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Rec: TMethod1Record;
  Stream: TFileStream;
begin
  Stream := TFileStream.Create('Method1.stm', fmOpenRead);
  Stream.ReadBuffer(Rec, SizeOf(Rec));
  Stream.Free;
```

```
  Label1.Caption := TimeToStr(EncodeTime(Rec.Hour,
    Rec.Minute, Rec.Second, Rec.MSec));
  Label2.Caption := Rec.Msg; { * }
end;
```

Примечание

В проекте RecordRead объявлена такая же запись TMethod1Record, описание которой во втором случае для краткости опущено.

Запись в файл происходит нормально, но при чтении в строке, отмеченной звездочкой, скорее всего, возникает исключение Access violation (в некоторых случаях исключения может не быть, но вместо сообщения будет выведен мусор). Причину этого мы уже обсудили ранее — указатель Msg, действительный в контексте процесса RecordWrite, не имеет смысла в процессе RecordRead, а сама строка передана не была. Без ошибок этим методом можно передать только пустую строку, потому что ей соответствует указатель nil, имеющий одинаковый смысл во всех процессах. Однако метод передачи строк, умеющий передавать только пустые строки, имеет весьма сомнительную ценность с практической точки зрения.

Самый простой способ исправить ситуацию — изменить тип поля Msg на ShortString. Больше ничего в приведенном коде менять не придется. Однако использование ShortString имеет два недостатка. Во-первых, длина строки в этом случае ограничена 255 символами. Во-вторых, если длина строки меньше максимально возможной, часть памяти, выделенной для структуры, останется незаполненной. Если средняя длина строки существенно меньше максимальной, то таких неиспользуемых кусков в потоке будет много, т. е. файл окажется неоправданно раздутым. Это всегда плохо, а в некоторых случаях — вообще недопустимо, поэтому ShortString можно посоветовать только в тех случаях, когда строки имеют примерно одинаковую длину (напомним, что ShortString позволяет ограничить длину строки меньшим, чем 255, числом символов — в этом случае поле будет занимать меньше места).

С одним из этих недостатков можно бороться: если заменить в записи ShortString статическим массивом типа Char, то можно передавать строки большей, чем 255 символов, длины. Второй метод демонстрирует этот способ.

В проекте RecordWrite этому соответствует код (листинг 3.37).

Листинг 3.37. Запись в файл структуры с массивом символов

```
const
  MsgLen = 15;
```

```
type
  TMethod2Record = packed record
    Hour: Word;
    Minute: Word;
    Second: Word;
    MSec: Word;
    Msg: array[0..MsgLen – 1] of Char;
  end;

procedure TForm1.Button2Click(Sender: TObject);
var
  Rec: TMethod2Record;
  Stream: TFileStream;
begin
  DecodeTime(Now, Rec.Hour, Rec.Minute, Rec.Second, Rec.MSec);
  StrPLCopy(Rec.Msg, Edit1.Text, MsgLen – 1);
  Stream := TFileStream.Create('Method2.stm', fmCreate);
  Stream.WriteBuffer(Rec, SizeOf(Rec));
  Stream.Free;
end;
```

В проекте RecordRead это следующий код (листинг 3.38).

Листинг 3.38. Чтение из файла структуры с массивом символов

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Rec: TMethod2Record;
  Stream: TFileStream;
begin
  Stream := TFileStream.Create('Method2.stm', fmOpenRead);
  Stream.ReadBuffer(Rec, SizeOf(Rec));
  Stream.Free;
  Label1.Caption := TimeToStr(EncodeTime(Rec.Hour,
    Rec.Minute, Rec.Second, Rec.MSec));
  Label2.Caption := Rec.Msg;
end;
```

Константа `MsgLen` задает максимальную (вместе с завершающим нулем) длину строки. В приведенном примере она взята достаточно маленькой, чтобы наглядно продемонстрировать, что данный метод имеет ограничения на длину строки. Переделки по сравнению с кодом предыдущего метода минимальны: при записи для копирования значения `Edit1.Text` вместо присваивания

нужно вызывать функцию StrPLCopy. В коде RecordRead изменений (за исключением описания самой структуры) вообще нет — это достигается за счет того, что массив Char считается компилятором совместимым с PChar, а выражения типа PChar могут быть присвоены переменным типа AnsiString — конвертирование выполнится автоматически.

Однако проблему неэффективного использования файлового пространства мы таким образом не решили. Более того, мы до конца не решили и проблему максимальной длины: хотя ограничение на длину строки теперь может быть произвольным, все равно оно должно быть известно на этапе компиляции. Чтобы полностью избавиться от этих проблем, необходимо вынести строку за пределы записи и сохранять ее отдельно, вместе с длиной, чтобы при чтении сначала читалась длина строки, затем выделялась для нее память, и в эту память читалась строка. Именно так работает третий метод.

В проекте RecordWrite это будет следующий код (листинг 3.39).

Листинг 3.39. Запись в файл строки отдельно от структуры

```
type
  TMethod3Record = packed record
    Hour: Word;
    Minute: Word;
    Second: Word;
    MSec: Word;
  end;

procedure TForm1.Button3Click(Sender: TObject);
var
  Rec: TMethod3Record;
  Stream: TFileStream;
  Msg: string;
  MsgLen: Integer;
begin
  DecodeTime(Now, Rec.Hour, Rec.Minute, Rec.Second, Rec.MSec);
  Msg := Edit1.Text;
  MsgLen := Length(Msg);
  Stream := TFileStream.Create('Method3.stm', fmCreate);
  Stream.WriteBuffer(Rec, SizeOf(Rec));
  Stream.WriteBuffer(MsgLen, SizeOf(MsgLen));
  if MsgLen > 0 then
    Stream.WriteBuffer(Pointer(Msg)^, MsgLen);
  Stream.Free;
end;
```

В проекте RecordRead это следующий код (листинг 3.40).

Листинг 3.40. Чтение из файла строки отдельно от структуры

```
procedure TForm1.Button3Click(Sender: TObject);
var
  Rec: TMethod3Record;
  Stream: TFileStream;
  Msg: string;
  MsgLen: Integer;
begin
  Stream := TFileStream.Create('Method3.stm', fmOpenRead);
  Stream.ReadBuffer(Rec, SizeOf(Rec));
  Stream.ReadBuffer(MsgLen, SizeOf(Integer));
  SetLength(Msg, MsgLen);
  if MsgLen > 0 then
    Stream.ReadBuffer(Pointer(Msg)^, MsgLen);
  Stream.Free;
  Label1.Caption := TimeToStr(EncodeTime(Rec.Hour,
    Rec.Minute, Rec.Second, Rec.MSec));
  Label2.Caption := Msg;
end;
```

Наконец-то мы получили код, который безошибочно передает строку, не имея при этом ограничений длины (кроме ограничения на длину `AnsiString`) и не расходуя понапрасну память. Правда, сам код получился сложнее. Во-первых, из записи исключено поле типа `string`, и теперь ее можно без проблем читать и писать в поток. Во-вторых, в поток после нее записывается длина строки. В-третьих, записывается сама строка.

Параметры вызова методов `ReadBuffer` и `WriteBuffer` для чтения/записи строки требуют дополнительного комментария. Метод `WriteBuffer` пишет в поток ту область памяти, которую занимает указанный в качестве первого параметра объект. Если бы мы указали саму переменную `Msg`, то записалась бы та часть памяти, которую занимает эта переменная, т. е. сам указатель. А нам не нужен указатель, нам необходима та область памяти, на которую он указывает, поэтому указатель следует разыменовать с помощью оператора `^`. Но просто взять и применить этот оператор к переменной `Msg` нельзя — с точки зрения синтаксиса она не является указателем. Поэтому приходится сначала приводить ее к указателю (здесь подошел бы любой указатель, не обязательно нетипизированный). То же самое относится и к ReadBuffer: чтобы прочитанные данные укладывались не туда, где хранится указатель на строку, а туда, где хранится сама строка, приходится прибегнуть к такой же конструк-

ции. И обратите внимание, что прежде чем читать строку, нужно зарезервировать для нее память с помощью SetLength.

Вместо приведения строки к указателю с последующим его разыменованием можно было бы использовать другие конструкции:

```
Stream.ReadBuffer(Msg[1], MsgLen);
```

и

```
Stream.WriteBuffer(Msg[1], MsgLen);
```

Это дает требуемый результат и даже более наглядно: действительно, при чтении и записи мы работаем с той областью памяти, которая начинается с первого символа строки, т. е. с той, где хранится сама строка. Но такой способ менее производителен из-за неявного вызова UniqueString. В нашем случае мы и так защищены от побочных изменений других строк (при записи строка не меняется, при чтении она и так уникальна — это обеспечивает SetLength), поэтому вполне можем обойтись без этой в данном случае излишней опеки со стороны компилятора.

Примечание

Если сделать MsgLen не независимой переменной, а полем записи, можно сэкономить на одном вызове ReadBuffer и WriteBuffer.

Недостатком этого метода является то, что мы вынуждены переделывать под него запись. В нашем примере это не составило проблемы, но в реальных проектах запись обычно предназначена не только для чтения и сохранения в поток, и если взять и выкинуть из нее строки, то все прочие участки кода станут более громоздкими. Поэтому в реальности приходится писать отдельные процедуры, которые сохраняют запись не как единое целое, а по отдельным полям.

Ранее мы говорили о том, что копирование записей, содержащих поля типа AnsiString, в рамках одного процесса маскирует проблему, т. к. указатель остается допустимым и даже (какое-то время) правильным. Но сейчас с помощью приведенного в листинге 3.41 кода (пример RecordCopy на компакт-диске) мы увидим, что проблема не исчезает, а просто становится менее заметной.

Листинг 3.41. Побочное изменение переменной после низкоуровнего копирования

```
type
  TSomeRecord = record
    SomeField: Integer;
```

```
    Str: string;
  end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Rec: TSomeRecord;
  S: string;

procedure CopyRecord;
var
  LocalRec: TSomeRecord;
begin
  LocalRec.SomeField := 10;
  LocalRec.Str := 'Hello!!!';
  UniqueString(LocalRec.Str);
  Move(LocalRec, Rec, SizeOf(TSomeRecord));
end;

begin
  CopyRecord;
  S := 'Good bye';
  UniqueString(S);
  Label1.Caption := Rec.Str;
  Pointer(Rec.Str) := nil;
end;
```

На экране вместо ожидаемого **Hello!!!** появится **Good bye**. Это происходит вот почему: процедура `Move` осуществляет простое побайтное копирование одной области памяти в другую, механизм изменения счетчика ссылок при этом не срабатывает. В результате менеджер памяти не будет знать, что после завершения локальной процедуры `CopyRecord` остаются ссылки на строку "Hello!!!". Память, выделенная этой строке, освобождается. Но `Rec.Str` продолжает ссылаться на эту уже освобожденную память. Для строки `S` выделяется свободная память — та самая, где раньше была строка `LocalRec.Str`. А поскольку `Rec.Str` продолжает ссылаться на эту область памяти, поэтому обращение к ней дает строку "Good bye", которая теперь там размещена.

Обратите внимание на последнюю строку — приведение `Rec.Str` к типу `Pointer` и обнулению. Это сделано для того, чтобы менеджер памяти не пытался финализировать строку `Rec.Str` после завершения процедуры, иначе он попытается освободить память, которая уже освобождена, и возникнет ошибка.

Чтобы показать, насколько коварна эта ошибка, рассмотрим следующий код (листинг 3.42, из того же примера RecordCopy на компакт-диске).

Листинг 3.42. Сокрытие ошибки при низкоуровневом копировании записи со строкой

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Rec: TSomeRecord;
  S: string;

procedure CopyRecord;
var
  LocalRec: TSomeRecord;
begin
  LocalRec.SomeField := 10;
  LocalRec.Str := 'Привет!';
  Move(LocalRec, Rec, SizeOf(TSomeRecord));
end;

begin
  CopyRecord;
  S := 'Пока!';
  Label1.Caption := Rec.Str;
end;
```

От предыдущего случая этот пример отличается только тем, что в нем нет вызовов `UniqueString`, и строки указывают на литералы в сегменте кода, которые никогда не удаляются. На экране получаем вполне ожидаемое **Привет!**. Обнулять указатель здесь уже нет смысла, потому что освобождать литерал менеджер памяти все равно не будет. Так ошибка оказалась скрытой.

Продолжим наши эксперименты. Запустим пример `RecordCopy` и понажимаем попеременно кнопки `Button1` и `Button2`. Мы видим, что результат не зависит от порядка, в котором мы нажимаем кнопки.

Модифицируем код: в локальной процедуре обработчика `Button1Click` уберем из строки "Hello!!!" восклицательные знаки, сократив ее до "Hello". Теперь можно наблюдать интересный эффект: если после запуска нажать сначала `Button1`, то никаких изменений мы не заметим. А вот если кнопка `Button2` будет нажата раньше, чем `Button1`, то при последующих нажатиях `Button1` никаких видимых эффектов не будет. Это связано с тем, что теперь строка "Hello" не равна по длине строке "Good bye", поэтому разместится ли "Good bye" в том же месте памяти, где раньше была "Hello", или в каком-то другом, зависит от истории выделения и освобождения памяти. Если мы начинаем "с чистого листа", память после строки "Hello" остается свободной, поэтому туда можно поставить более длинную строку. А вот если раньше па-

мять уже выделялась и освобождалась (внутри методов `TLabel`), то тот кусочек свободной памяти, который достаточен для "Hello", слишком мал для "Good bye", и эта строка размещается в другом месте. А там, куда указывает `Rec.Str`, остается мусор, работать с которым нормально невозможно, поэтому при попытке присвоить его свойству `Label1.Caption` последнее не меняется (эффект наблюдается только до Delphi 7 включительно; в более новых версиях Delphi используется новый менеджер памяти FastMem, который немного по-другому размещает строки в памяти, поэтому с ним зависимости от порядка нажатия кнопок не будет).

Примечание

Если увеличить длину строки "Привет!" хотя бы на один символ, чтобы она была не короче, чем "Good bye" (или наоборот, сократить его так, чтобы оно стало короче "Hello"), мы снова увидим, что порядок нажатия кнопок не влияет на результат. Это происходит потому, что строка "Hello" размещается там, где раньше была строка "Привет!", а вот "Good bye" там уже не помещается. Если же обе строки там помещаются (или обе не помещаются), они снова оказываются в одной области памяти. Внимательный читатель может спросить: а при чем здесь длина строки "Привет!", если эта строка хранится в сегменте кода и никогда не освобождается? Дело в том, что когда мы присваиваем эту строку свойству `Label1.Caption`, внутри методов `TLabel` происходит ее перенос в динамическую память для внутренних нужд этого класса.

Даже на таком простом примере видно, насколько коварна эта ошибка и как незначительные изменения в коде могут кардинально изменить ее проявления. Между тем приведенный здесь код — плод долгого "приручения" этой ошибки, чтобы она всегда проявлялась предсказуемым образом. Но даже сейчас мы не можем дать полной гарантии, что у кого-то из читателей из-за какой-то неучтенной мелочи не возникнет ситуация, когда эта ошибка проявляется как-то по-другому (как мы уже видели, даже в разных версиях Delphi эта ошибка проявляет себя немного по-разному). В реальных проектах все гораздо сложнее, и поведение программы из-за этой ошибки может стать таким неожиданным, а проявление этой ошибки — настолько далеким от того места, где она сделана, что впору будет "прыгать вокруг компьютера с бубном", изгоняя бесов. Чтобы не оказаться в таком положении, нужно очень аккуратно работать со строками (а также с другими автоматически финализируемыми типами: динамическими массивами, интерфейсами, вариантами), чтобы тот код, который неявно генерирует компилятор, не оказался в тупике. Чаще всего проблемы возникают при побайтном копировании переменной типа `AnsiString` (не обязательно в составе записи) или при работе с ней как с указателем другого типа. Это не значит, что приводить `AnsiString` к другим указателям категорически нельзя — ранее мы уже делали это, и вполне успешно. Но, применяя любой низкоуровневый инструмент к таким строкам, разработчик должен четко представлять, как это отразится на внутренних механизмах работы с ними. Иначе — вот такая непонятная ошибка.

Еще одна ситуация, когда записи со строками могут преподнести сюрприз — это выделение динамический памяти для них. Динамическую память можно выделить двумя способами: с помощью процедуры New или GetMem (освобождать ее надо, соответственно, с помощью Dispose или FreeMem). Для записей, не содержащих строки, эти способы практически эквивалентны, за исключением того, что при использовании New объем выделяемой памяти определяет компилятор, поэтому New считается более безопасным вариантом. Если же запись содержит строку, то эта строка должна быть инициализирована, иначе попытка работы с ней приведет к ошибке. Процедура GetMem ничего не делает с содержимым выделяемой ею памяти, и строка остается неинициализированной, в то время как New выполняет инициализацию. Это не значит, что GetMem непригодна для выделения памяти для такой записи, просто после вызова GetMem нужно не забыть вызвать специальную процедуру Initialize, которая правильно инициализирует строки в записи. Соответственно, прежде чем удалить такую запись с помощью FreeMem, необходимо вызвать процедуру Finalize для финализации строк. Это создает дополнительные проблемы, не давая никаких преимуществ, поэтому целесообразнее все-таки использовать New и Dispose.

Преимущество GetMem перед New заключается в том, что за один вызов GetMem можно выделить память сразу для нескольких записей (с последующей их ручной инициализацией, конечно же), в то время как New выделяет память только для одного экземпляра записи. Но с появлением в языке динамических массивов это преимущество тоже перестало быть особо полезным. Проще объявить динамический массив из записей и создать требуемое число элементов в нем — компилятор сам позаботится об инициализации таких переменных. Поэтому мы рекомендуем отказаться от GetMem при выделении памяти под записи со строками, а если уж вы столкнулись с ситуацией, когда без этого совсем никак, не забывайте вызывать Initialize и Finalize.

Примечание

Память для записей можно выделять и в обход менеджера памяти Delphi, напрямую вызывая системные функции типа HeapAlloc, VirtualAlloc или CoTaskMemAlloc. Разумеется, компилятор в этом случае не сможет инициализировать и финализировать выделяемую память, поэтому, как и в случае с GetMem, для строк с записями необходимо пользоваться процедурами Initialize и Finalize.

3.3.9. Использование ShareMem

Пример, который мы сейчас рассмотрим, — это даже не "подводный камень", это то, что в форумах обычно называется "грабли". Все новые и новые программисты с завидным упорством наступают на эти грабли и получают по

лбу, хотя, казалось бы, вокруг стоят таблички, предупреждающие об опасности, только не ленись читать.

Итак, создаем новую динамически компонуемую библиотеку (DLL). Delphi предлагает нам следующую заготовку (листинг 3.43).

Листинг 3.43. Базовый

```
library Project1;

{ Important note about DLL memory management: ShareMem must be the
  first unit in your library's USES clause AND your project's (select
  Project-View Source) USES clause if your DLL exports any procedures or
  functions that pass strings as parameters or function results. This
  applies to all strings passed to and from your DLL--even those that
  are nested in records and classes. ShareMem is the interface unit to
  the BORLNDMM.DLL shared memory manager, which must be deployed along
  with your DLL. To avoid using BORLNDMM.DLL, pass string information
  using PChar or ShortString parameters. }

uses
  SysUtils,
  Classes;

{$R *.RES}

begin
end.
```

Самое важное здесь — комментарий. Его следует внимательно прочитать и осознать, а главное — выполнить эти советы, иначе при передаче строк `AnsiString` между DLL и программой вы будете получать ошибку Access violation в самых неожиданных местах. Почему-то многие им пренебрегают, а потом бегут с вопросами в разные форумы, хотя минимум внимательности и отсутствия снобизма по отношению "к этим, из Borland'а, которые навставляли тут никому не нужных комментариев" могли бы уберечь от ошибки.

Для начала выясним источник ошибки. Менеджер памяти Delphi работает следующим образом: он берет у системы большие блоки памяти, а потом по мере необходимости выделяет их по частям. Это позволяет избежать частых выделений памяти системными средствами и тем самым повышает производительность. Следствием этого становится то, что менеджер памяти должен иметь информацию о том, какими блоками он распределил полученную от системы память между различными потребителями.

Менеджер памяти реализуется модулем System. Так как DLL компонуется отдельно от использующего ее exe-файла, у нее будет своя копия кода System и, следовательно, свой менеджер памяти. И если объект, память для которого была выделена в коде основного модуля программы, попытаться освободить в коде DLL, то получится, что освобождать память будет совсем не тот менеджер, который ее выделил. А сделать он этого не сможет, т. к. не обладает информацией о выделенном блоке. Результат — ошибка (скорее всего, Access violation при выходе из процедуры). А при работе со строками AnsiString память постоянно выделяется и освобождается, поэтому, попытавшись работать с одной и той же строкой и в главном модуле, и в DLL, мы получим ошибку.

Теперь, когда мы поняли, почему возникает проблема, разберемся, как ShareMem ее решает. Delphi предоставляет возможность заменить стандартный менеджер памяти своим: для этого нужно написать низкоуровневые функции выделения, освобождения и перераспределения памяти и сообщить их адреса через процедуру SetMemoryManager. После этого через них будут работать все высокоуровневые функции для манипуляций с памятью (New, GetMem и т. п.). Именно это и делает ShareMem: в секции инициализации этого модуля содержится код, заменяющий функции работы с памятью своими, которые находятся во внешней библиотеке BORLNDMM.DLL. Получается, что и библиотека, и главный модуль работают с одним менеджером памяти, что решает описанные проблемы.

Если менеджер памяти попытаться поменять не в самом начале программы, то ему придется освобождать память, которую успел выделить предыдущий менеджер памяти, что приведет к той же самой проблеме. Поэтому заменить менеджер памяти нужно до того, как будет выполнена первая операция по ее выделению. Отсюда возникает требование вставлять ShareMem первым модулем в dpr-файлах главного модуля и DLL — чтобы его секция инициализации была первым выполняемым программой кодом.

В Интернете часто можно встретить утверждения, что в новых версиях Delphi (BDS 2006 и выше) ShareMem не нужен, потому что стандартный менеджер памяти там заменен на FastMM, который прекрасно обходится без ShareMem. Это неверно. Оригинальный FastMM действительно может функционировать без ShareMem при выполнении определенных условий. Модуль, использующий FastMM ("модуль" здесь значит модуль в понимании системы, т. е. module, а не unit), может предоставить свой менеджер памяти в общее пользование, а все остальные модули, подключившие FastMM, будут пользоваться этим менеджером вместо своего. Получится, что все модули в процессе будут работать с одним менеджером памяти, и проблем не будет. В общее пользование свой менеджер памяти предоставляет тот модуль, который инициализируется самым первым (т. к. основной модуль программы инициализируется только

после того, как будут проинициализированы все статически связанные с ним DLL, в общее пользование свой менеджер памяти предоставляет одна из DLL).

Тот вариант FastMM, который входит в состав новых версий Delphi, тоже может быть предоставлен в общее пользование, но по умолчанию этого не происходит, так что с передачей строк в DLL возникнут те же проблемы, что и в старых версиях Delphi. Но решить эти проблемы теперь можно двумя способами. Первый — это использовать ShareMem и распространять с программой библиотеку BORLNDMM.dll, точно так же, как и в более ранних версиях Delphi. Второй способ — подключить к dpr-файлам библиотек и главного модуля модуль SimpleShareMem. Этот модуль в своей секции инициализации проверяет, есть ли уже переданный в общее пользование менеджер памяти, и если есть, переключает свою программу или DLL на него, а если еще нет, делает текущий менеджер памяти общим. Использование модулей SimpleShareMem и ShareMem идентично: его так же нужно указывать первым в списке uses главного файла проекта. Но никаких дополнительных библиотек распространять с программой не придется. Таким образом, новые версии Delphi действительно позволяют обойтись без библиотеки BORLNDMM.DLL, но это все-таки получается не автоматически, а после некоторых усилий.

Кстати, к данному в комментарии совету заменить AnsiString на PChar, чтобы избавиться от необходимости использования ShareMem, следует относиться осторожно: если мы попытаемся, например, вызвать StrNew в основной программе, а StrDispose — в DLL, то получим ту же проблему. Вопрос не в типах данных, а в том, как манипулировать памятью. Поэтому обычный способ работы с PChar следующий: программа выделяет буфер своим менеджером памяти и передает указатель на этот буфер, а также его длину в качестве параметров функции из DLL. Эта функция заносит в буфер требуемую строку, не перераспределяя память. Затем программа освобождает эту строку своим же менеджером памяти. В листинге 3.44 приведен пример кода такой функции в DLL.

Листинг 3.44. Код функции в DLL

```
function GetString(Buf: PChar; BufLen: Integer): Integer;
var
  S: string;
begin
  // Формируем строку для возврата программе
  ...
  // Копируем строку в буфер
  if BufLen > 0 then
    StrLCopy(Buf, PChar(S), BufLen – 1);
```

```
  // Возвращаем требуемый размер буфера
  Result := Length(S) + 1;
end;
```

Здесь параметр Buf содержит указатель на буфер, выделенный вызывающей программой, BufLen — размер этого буфера в байтах. Для примера здесь взят случай, когда строка, которую нужно возвратить, формируется в переменной типа string, т. к. в большинстве случаев это наиболее удобный способ. После того как строка сформирована, ее содержимое копируется в буфер с учетом его длины. Результат, который возвращает функция, — это необходимый размер буфера. Программа по этому результату может сделать вывод, поместилась ли вся строка в выделенный ей буфер, и если не поместилась, принять меры, например, вызвать функцию еще раз, выделив под буфер больше памяти.

Если не существует ограничения на длину возвращаемой строки, программа "не знает", буфер какого размера потребуется. Наиболее простое решение этой проблемы следующее: программа сначала вызывает функцию GetString, передавая nil в качестве указателя на буфер и 0 в качестве размера буфера. Затем по результату функции определяется требуемый размер буфера, выделяется память и функция вызывается еще раз, уже с буфером нужного размера. Такой способ обеспечивает правильную передачу строки любой длины, но требует двукратного вызова функции, что снижает производительность, особенно в том случае, если на формирование строки тратится много времени.

Повысить среднюю производительность можно, применяя комбинированный метод получения буфера. Программа создает массив в стеке такого размера, чтобы в большинстве случаев возвращаемая строка умещалась в нем. Этот размер определяется в каждом конкретном случае, исходя из особенностей функции и условий ее вызова. А на тот случай, если она все-таки там не поместилась, предусмотрен запасной вариант с выделением буфера в динамической памяти. Этот подход иллюстрирует листинг 3.45.

Листинг 3.45. Быстрый (в среднем) способ получения строки через буфер

```
const
  StatBufSize = ...; // Размер, подходящий для данного случая
var
  StatBuf: array[0..StatBufSize – 1] of Char;
  Buf: PChar;
  RealLen: Integer;
begin
  // Пытаемся разместить строку в буфере StatBuf
  RealLen := GetString(StatBuf, StatBufSize);
```

```
  if RealLen > StatBufSize then
  begin
    // Если StatBuf оказался слишком мал, динамически выделяем буфер
    // нужного размера и вызываем функцию еще раз
    Buf := StrAlloc(RealLen);
    GetString(Buf, RealLen);
  end
  else
    // Размера статического буфера хватило. Пусть Buf указывает
    // на StatBuf, чтобы нижеследующий код мог в любом случае
    // обращаться к буферу через переменную Buf
    Buf := StatBuf;
  // Что-то делаем с содержимым буфера
  ...
  // Если выделяли память, ее следует очистить
  if Buf <> StatBuf then
    StrDispose(Buf);
end;
```

Следует также упомянуть о еще одной альтернативе передачи строк в DLL — типе `WideString`, который хранит строку в кодировке Unicode и является, по сути, оберткой над системным типом `BSTR`. Работать с `WideString` так же просто, как и с `AnsiString`, перекодирование из ANSI в Unicode и обратно выполняется автоматически при присваивании значения одного типа переменной другого. В целях совместимости с COM и OLE при работе с памятью для строк `WideString` используется специальный системный менеджер памяти (через API-функции `SysAllocString`, `SysFreeString` и т. п.), поэтому передавать эти строки из DLL в главный модуль и обратно можно совершенно безопасно даже без `ShareMem`. Правда, при этом не стоит забывать о расходовании процессорного времени на перекодировку, если основная работа идет не с Unicode, а с ANSI.

Отметим одну ошибку, которую делают новички, прочитавшие комментарий про `ShareMem`, но не умеющие работать с `PChar`. Они пишут, например, такой код для функции, находящейся в DLL и возвращающей строку (листинг 3.46).

Листинг 3.46. Неправильный способ возврата строки из DLL

```
function SomeFunction(...): PChar;
var
  S: string;
begin
  // Здесь присваивается значение S
  Result := PChar(S);
end;
```

Такой код компилируется и даже, за редким исключением, дает ожидаемый результат. Но тем не менее, в этом коде грубая ошибка. Указатель, возвращаемый функцией, указывает на область памяти, которая считается свободной, поскольку после выхода переменной `S` за пределы области видимости память, которую занимала эта строка, освободилась. Менеджер памяти может в любой момент вернуть эту память системе (тогда обращение к ней вызовет Access violation) или задействовать для других целей (тогда новая информация уничтожит содержащуюся там строку). Проблема маскируется тем, что обычно результат используется немедленно, до того как менеджер памяти что-то сделает с этим блоком. Тем не менее полагаться на это и писать такой код не следует.

3.4. Прочие "подводные камни"

В этом разделе собрана небольшая коллекция не связанных между собой "подводных камней", с которыми пришлось столкнуться автору книги.

3.4.1. Порядок вычисления операндов

Эта проблема связана с тем, что у человека есть определенные интуитивные представления о порядке выполнения действий программой, однако компилятор не всегда им соответствует. Рассмотрим следующий код (листинг 3.47, пример OpOrder на компакт-диске).

Листинг 3.47. "Неправильный" порядок вычисления операндов

```
var
  X: Integer;

function GetValueAndModifyX: Integer;
begin
  X := 1;
  Result := 2;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  A1, A2: Integer;
begin
  X := 2;
  A1 := X + GetValueAndModifyX;
  X := 2;
```

```
  A2 := GetValueAndModifyX + X;
  Label1.Caption := IntToStr(A1);
  Label2.Caption := IntToStr(A2);
end;
```

Суть этого примера заключается в том, что функция GetValueAndModifyX имеет побочный эффект — изменяет значение глобальной переменной X. И эту же переменную мы используем при вычислении выражения, в которое входит также вызов GetValueAndModifyX. При вычислении A1 в выражении сначала упоминается X, а потом GetValueAndModifyX, при вычислении A2 — наоборот. Логично было бы предположить, что A1 получит значение 4, A2 — 3, т. к. вычисление первого операнда должно выполняться раньше второго. В действительности же обе переменные получат значение 3, поскольку компилятор сам выбирает порядок вычисления операндов независимо от того, в каком порядке они упоминаются в выражении. То же самое касается любых коммутативных операций: умножения, арифметических and, or и xor.

Посмотрим, что будет для некоммутативных операций, например, для деления (листинг 3.48).

Листинг 3.48. "Неправильный" порядок вычисления операндов при делении

```
procedure TForm1.Button2Click(Sender: TObject);
var
  A1, A2: Extended;
begin
  X := 2;
  A1 := X / GetValueAndModifyX;
  X := 2;
  A2 := GetValueAndModifyX / X;
  Label1.Caption := FloatToStr(A1);
  Label2.Caption := FloatToStr(A2);
end;
```

В результате выполнения этого кода A1 получает значение 0.5, A2 — 2, т. е. и здесь сначала вычисляется функция, а потом берется значение переменной X.

Если бы функция GetValueAndModifyX не имела побочных эффектов (т. е. только возвращала бы результат и больше ничего не меняла), порядок вычисления аргументов был бы нам безразличен. Вообще, функции, имеющие побочные эффекты, считаются потенциальным источником ошибок, поэтому их написание нежелательно. Но в некоторых случаях (например, в функции Random) обойтись без побочных эффектов невозможно.

Примечание

Побочные эффекты в функциях настолько небезопасны, что в некоторых языках они полностью запрещены. Например, в Аде изменять значения глобальных переменных могут только процедуры, но не функции.

Ради интереса посмотрим, что будет, если вторым аргументом тоже будет функция, зависящая от X (листинг 3.49).

Листинг 3.49. Сложение двух операндов с побочными эффектами

```
function GetX: Integer;
begin
  Result := X;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  A1, A2: Integer;
begin
  X := 2;
  A1 := GetX + GetValueAndModifyX;
  X := 2;
  A2 := GetValueAndModifyX + GetX;
  Label1.Caption := IntToStr(A1);
  Label2.Caption := IntToStr(A2);
end;
```

Здесь A1 получит значение 4, A2 — 3, т. е. интуитивно ожидаемые. Тем не менее полагаться на интуицию все же не стоит: в более сложных случаях она может подвести. Дело в том, что стандарт языка Паскаль разрешает разработчикам конкретной реализации языка самому выбирать порядок вычисления операндов [5]. Поэтому, даже если вам удалось добиться желаемого порядка вычисления, в следующих версиях Delphi (или при переносе на другую платформу) программа может начать работать неправильно. Таким образом, разработчик не имеет права делать какие-то предположения о том, в каком порядке будут вычисляться операнды, а когда изменение этого порядка может повлиять на результат, код должен быть написан таким образом, чтобы исключить эту возможность. В частности, пример со сложением должен быть переписан так (листинг 3.50).

Листинг 3.50. Явное управление порядком вычисления операндов

```
procedure TForm1.Button4Click(Sender: TObject);
var
  A1, A2: Integer;
```

```
begin
  X := 2;
  A1 := X;
  Inc(A1, GetValueAndModifyX);
  X := 2;
  A2 := GetValueAndModifyX;
  Inc(A2, X);
  Label1.Caption := IntToStr(A1);
  Label2.Caption := IntToStr(A2);
end;
```

Такой код, несмотря на побочные эффекты функции `GetValueAndModifyX`, даст ожидаемые значения при любом порядке вычисления операндов, т. к. здесь вычисление операндов разнесено по разным операторам, а порядок выполнения операторов четко определен.

Примечание

Другие компиляторы могут использовать иной порядок вычисления операндов. Так, FreePascal вычисляет их в том порядке, в каком они встречаются в выражении, т. е. в первом примере A1 получит значение 4, A2 — 3.

3.4.2. Зацикливание обработчика *TUpDown.OnClick* при открытии диалогового окна в обработчике

Для демонстрации этого "подводного камня" нам потребуется проект, на форме которого находится компонент `TUpDown` со следующим обработчиком события `OnClick` (листинг 3.51, пример UpDownDlg на компакт-диске).

Листинг 3.51. Обработчик события `OnClick` компонента `UpDown1`

```
procedure TForm1.UpDown1Click(Sender: TObject; Button: TUDBtnType);
begin
  Application.MessageBox('Text', 'Caption', MB_OK);
end;
```

Теперь, если запустить программу и нажать на верхнюю кнопку `UpDown1`, откроется окно с сообщением (при нажатии на нижнюю кнопку окно не будет открываться потому, что по умолчанию у компонента `TUpDown` свойства `Position` и `Min` равны нулю, поэтому нажатие на нижнюю кнопку не приводит к изменению значения `Position`, и событие `OnClick` не возникает; если изменить значение свойства `Min` или `Position`, то тот же эффект будет наблюдаться и при нажатии на нижнюю кнопку). Если закрыть это окно, то щелчок мышью в любом месте формы снова приведет к срабатыванию события

OnClick и открытию окна, и так до бесконечности: любой щелчок по форме в любом ее месте будет снова и снова приводить к появлению сообщения. Эффект наблюдается и в том случае, когда вместо стандартного сообщения в обработчике показывается любая другая модальная форма. Кроме того, тот же эффект будет, и если использовать события OnChanging или OnChangingEx вместо OnClick, но мы далее для определенности будем говорить только об OnClick.

Если этот код пройти по шагам в отладчике, то никакого зацикливания не возникает: OnClick вызывается один раз, любое последующее нажатие кнопки мыши на форме не приводит ни к каким необычным результатам.

Причина этой проблемы в том, как VCL обрабатывает сообщения, которые система помещает в очередь. При нажатии на кнопку компонента TUpDown в очередь сообщений помещаются два сообщения: WM_LBUTTONDOWN и WM_NOTIFY. Компонент TUpDown по умолчанию имеет стиль csCaptureMouse — это означает, что при обработке WM_LBUTTONDOWN VCL захватывает мышь в монопольное пользование для данного компонента.

Примечание

Монопольное использование мыши означает, что любые сообщения, связанные с мышью, будут поступать захватившему мышь окну, даже если ее курсор в это время находится за пределами данного компонента. Примером захвата мыши может служить любая кнопка: щелкните мышью над любой кнопкой на экране и, не отпуская клавиши мыши, начните перемещать курсор. Когда курсор будет выходить за пределы кнопки, она будет отжиматься, находить на нее — снова нажиматься. Теперь отведите курсор за пределы кнопки, отпустите клавишу мыши и снова подведите его к кнопке. Кнопка не нажмется. Это происходит потому, что пока клавиша мыши удерживается нажатой, мышь захвачена кнопкой, и сообщение об отпускании клавиши мыши передается кнопке, независимо от того, над каким окном находится курсор. Это позволяет кнопке правильно реагировать на отпускание пользователем мыши, в том числе и за ее пределами.

Затем начинает обрабатываться событие WM_NOTIFY, которое уведомляет программу о том, что пользователь нажал на кнопку компонента TUpDown. Именно при обработке этого сообщения VCL вызывает событие TUpDown.OnClick, в котором открывается модальное окно. Все это происходит очень быстро, поэтому кнопку мыши пользователь отпускает тогда, когда модальное окно уже оказалось на экране. В результате сообщение WM_LBUTTONUP либо попадает в очередь открывшегося диалогового окна, если мышь находилась над ним, либо вообще никуда не попадает, если мышь была вне модального окна. На время существования модального окна система "забывает" о том, что мышь захвачена для монопольного использования, но "вспоминает" об этом, как только модальное окно закрывается. Монопольное использование мыши компонентом TUpDown должно отменяться при обработке сообщения WM_LBUTTONUP,

но оно, как было сказано ранее, в очередь не попадает, поэтому после закрытия окна мышь остается захваченной данным компонентом. Поэтому любое нажатие кнопки мыши воспринимается системой как относящееся к UpDown1, и снова приводит к помещению в очередь сообщений WM_LBUTTONDOWN и WM_NOTIFY, которые обрабатываются описанным образом. Так получается порочный круг, из которого при нормальной работе программы нет выхода. Этот круг может быть разорван, например, отладчиком, который отменяет монопольное использование мыши компонентами программы, чтобы иметь возможность работать.

В этой проблеме виновата VCL, которая зачем-то назначает компоненту TUpDown стиль csCaptureMouse. Данный компонент реализуется не средствами VCL, — это стандартное окно системного класса UPDOWN_CLASS, а компонент TUpDown — это только оболочка для него. Поэтому все необходимые перехваты мыши выполняются самой системой, VCL нет нужды в это вмешиваться.

Чтобы избавиться от проблемы, нужно убрать csCaptureMouse из списка стилей компонента. Делается это так:

```
UpDown1.ControlStyle := UpDown1.ControlStyle – [csCaptureMouse];
```

Этот код достаточно выполнить один раз (например, в обработчике события OnCreate формы), и проблемы с зацикливанием исчезнут (в примере UpDownDlg эта строка закомментирована).

Отметим, что в Windows предусмотрено специальное сообщение — WM_CANCELMODE, — посылаемое при открытии диалогового окна тому окну, которое захватило мышь, чтобы оно ее освободило. Один из способов решения проблемы — добавление в UpDown1 обработчика этого сообщения (для этого можно написать наследника TUpDown или же воспользоваться свойством WindowProc — *см. разд. 1.1.8*), который отменит захват мыши. Отсутствие этого обработчика — тоже явная ошибка VCL.

3.4.3. Access violation при закрытии формы с перекрытым методом *WndProc*

Чтобы увидеть этот "подводный камень", создадим проект, содержащий две формы: главную Form1 и вспомогательную Form2. В Form1 добавим код, который по нажатию кнопки открывает Form2.

Во второй форме напишем обработчик события OnClose таким образом, чтобы он устанавливал по закрытию действие caFree. Добавим поле строкового типа, перекроем конструктор и метод WndProc так, чтобы окончательный код выглядел следующим образом (листинг 3.52, пример CloseAV на компакт-диске).

Листинг 3.52. Код класса TForm2

```
type
  TForm2 = class(TForm)
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    S: string;
  protected
    procedure WndProc(var Message: TMessage); override;
  public
    constructor Create(AOwner: TComponent); override;
  end;

....

constructor TForm2.Create(AOwner: TComponent);
begin
  S := 'abc';
  inherited;
end;

procedure TForm2.WndProc(var Message: TMessage);
begin
  inherited;
  S[2] := 'x';  { * }
end;

procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;s
```

Обратите внимание, что в конструкторе сначала присваивается значение полю S, и лишь потом вызывается унаследованный конструктор. Это сделано потому, что по умолчанию S содержит пустую строку, т. е. nil, а уже при вызове унаследованного конструктора окно получит сообщения, для обработки которых будет вызван метод WndProc. Если в этот момент S будет попрежнему nil, попытка обратиться ко второму символу строки вызовет Access violation. Поэтому еще до начала работы унаследованного конструктора поле S должно получить подходящее значение.

Запустим программу и попытаемся закрыть второе окно. Возникнет исключение Access Violation: Write of address 00000001. Проблема будет в строке,

отмеченной {*}. При этом любые другие манипуляции с окном никаких исключений вызывать не будут.

При Action = caFree после завершения работы метода FormClose VCL вызывает метод TCustomForm.Release. Проблема именно в нем: если попытаться закрыть Form2 с помощью Release, возникнет то же самое исключение. В справке Release позиционируется как безопасный способ удаления формы из ее собственного метода. К сожалению, в действительности это не так: реализация этого удаления оставляет желать лучшего и может приводить к попыткам работать с объектом тогда, когда его уже не существует.

При вызове Release в очередь помещается сообщение CM_RELEASE, адресатом которого является сама удаляемая форма. В очередном цикле петли сообщений CM_RELEASE извлекается из очереди и передается на обработку. Так как сообщение адресовано форме, она же его и обрабатывает. Рассмотрим более подробно, как это происходит. (Детально механизм обработки сообщений в VCL описан в *разд. 1.1.8*; мы здесь рассмотрим только ту часть, которая относится к обработке CM_RELEASE.)

Система передает управление оконной процедуре. Для каждого экземпляра визуального компонента VCL создает свою оконную процедуру с помощью MakeObjectInstance. Эта процедура вызывает метод объекта MainWndProc, передающий управление тому методу, на который указывает свойство WindowProc. По умолчанию это WndProc. WndProc не обрабатывает CM_RELEASE самостоятельно, а передает его методу Dispatch. Dispatch пытается найти для этого сообщения специальный обработчик (метод с директивой message) и, т. к. в TCustomForm такой обработчик описан (он называется CMRelease), передает управление ему.

И здесь начинается самое интересное. CMRelease просто вызывает Free, удаляя тем самым объект, т. е. объект удаляется из метода самого объекта, что делать запрещено. Таким образом, после выполнения Free управление вновь получает CMRelease. Из него управление возвращается в Dispatch, оттуда — в WndProc, затем — в MainWndProc, далее — в оконную процедуру, и только после этого управление получает код, который никак не связан с конкретным экземпляром компонента. Мы видим, что после обработки CM_RELEASE и удаления объекта его методы продолжают работать. Методы уже не существующего объекта!

В принципе, методы несуществующего объекта могут вполне нормально завершить свою работу, если не будут обращаться к его полям или иным образом использовать указатель Self, который к этому моменту будет уже недействительным. Но стоило нам только вставить в один из этих методов код, задействующий поле объекта, как возникла ошибка.

В данном примере получается следующее: сначала CM_RELEASE передается стандартному обработчику, который вызывает деструктор. При работе деструктора финализируются все поля объекта, для которых это требуется. В нашем случае это означает, что в поле S заносится nil (освобождения памяти при этом не происходит, потому что S до этого ссылалась на литерал, хранящийся в кодовом сегменте, а не в динамической памяти). После этого начинает работать наш код, который пытается изменить второй символ в строке. Программа пытается обратиться к ячейке с адресом nil + 1, т. е. 00000001, что и приводит к ошибке Access violation.

Обращение в аналогичной ситуации к нефинализируемым полям (целым, вещественным, логическим и т. п.) обычно не приводит к исключению. Это связано с тем, что менеджер памяти Delphi обычно не сразу отдает системе ту память, которую освобождает объект, поэтому программа, с точки зрения системы, имеет полное право ею пользоваться. Поля объекта не очищаются, и его образ продолжает храниться в памяти, просто менеджер памяти помечает эту область как неиспользуемую и может в любой момент выделить ее для хранения другого объекта. Это создает иллюзию того, что объект продолжает существовать и позволяет работать с уже несуществующим объектом. Но это все равно некорректно, потому что любое перераспределение памяти в данной ситуации может привести к непонятной ошибке.

Посмотрим, что будет, если строку S[1] := 'x' заменить на S := IntToStr(Msg.Msg). Как мы уже выяснили, после уничтожения объекта в той области памяти, где хранилось значение S, будет nil. Указатель на вновь созданную строку будет помещен в эту область памяти. Но к ней уже не будет применяться финализация, т. к. менеджер памяти будет считать эту область памяти финализированной. Произойдет утечка памяти.

Отметим, что для вновь созданной строки память может быть выделена таким образом, что она наложится на те ячейки, в которых хранились значения полей уничтоженной формы, в том числе значение S. В этом случае попытка обратиться к такому полю приведет к непредсказуемым результатам.

Аналогичная проблема может появляться не только при перекрытии WndProc, а вообще при любом способе внедрения своего кода в цепочку обработки так, чтобы он выполнялся после CMRelease.

Совершенно непонятно, почему разработчики VCL реализовали такой заведомо некорректный механизм работы Release. Чтобы избежать всех описанных проблем, достаточно было бы просто посылать CM_RELEASE не самой форме, а окну, создаваемому объектом Application, а указатель на освобождаемую форму передавать через параметры этого сообщения. Тогда деструктор формы вызывался бы из метода объекта Application, и никаких проблем не было бы.

Эта проблема обнаружена во всех версиях Delphi с 3-й по 2007-ю (в других версиях не проверялась). Самый простой способ ее преодоления — отмена опасных действий, если получено сообщение CM_RELEASE. Например, в описанном случае безопасным будет следующий код (листинг 3.53).

Листинг 3.53. Безопасный вариант метода WndProc

```
procedure TForm2.WndProc(var Message: TMessage);
begin
  inherited;
  if Msg.Msg <> CM_RELEASE then
    S[2] := 'x';
end;
```

Другой способ заключается в том, чтобы перенести обработку CM_RELEASE в объект Application с помощью его события OnMessage. Проблема заключается лишь в том, что адрес удаляемой формы будет неизвестен, но его легко найти по дескриптору окна. Например, в данном случае можно положить на первую форму TApplicationEvents и в его обработчике OnMessage написать следующий код (листинг 3.54; в примере CloseAV этот код закомментирован).

Листинг 3.54. Обработка сообщения CM_RELEASE объектом Application

```
procedure TForm1.ApplicationEvents1Message(var Msg: tagMSG;
  var Handled: Boolean);
var
  I: Integer;
begin
  if Msg.Message = CM_RELEASE then
    for I := 0 to Screen.FormCount – 1 do
      if Screen.Forms[I].Handle = Msg.hwnd then
      begin
        Screen.Forms[I].Free;
        Handled := True;
        Exit;
      end;
end;
```

Событие OnMessage позволяет перехватить сообщения до того, как они будут диспетчеризованы окну-адресату, соответственно, форма будет уничтожена раньше, чем начнет обрабатывать CM_RELEASE.

3.4.4. Подмена имени оконного класса, возвращаемого функцией *GetClassInfo*

Создадим новый проект в Delphi, поместим на форму кнопку и метку и создадим следующий обработчик нажатия кнопки (листинг 3.55, пример ClassName на компакт-диске).

Листинг 3.55. Подмена имени оконного класса

```
procedure TForm1.Button1Click(Sender: TObject);
var
  CI: TWndClass;
  S: string;

procedure DoGetClassInfo;
begin
  GetClassInfo(hInstance, PChar('TForm' + IntToStr(1)), CI);
end;

begin
  DoGetClassInfo;
  S := 'abcde' + IntToStr(2);
  Label1.Caption := CI.lpszClassName;
end;
```

Что будет выведено на экран в результате выполнения этого кода? Так как класс называется "TForm1", логично предположить, что именно это и будет выведено. На самом деле мы увидим **abcde2** — ту строку, которая присвоена переменной `S`.

Разберемся, как значение переменной `S` оказывается в поле `CI.lpszClassName`.

Согласно MSDN поле `lpszClassName` имеет тип `LPCTSTR` (`PChar`), и в него функция `GetClassInfo` заносит указатель на строку, содержащую имя оконного класса. Но нигде не сказано, в какой области памяти должна располагаться эта строка.

Функция `GetClassInfo` поступает очень просто, но не совсем корректно: один из ее аргументов — указатель на строку с именем класса. Именно его функция и помещает в `lpszClassName`.

В приведенном примере в качестве аргумента `GetClassInfo` передается выражение типа `string`, приведенное к `PChar`, которое не может быть вычислено на этапе компиляции, поэтому компилятор генерирует код, вычисляющий данное выражение. Этот код размещает вычисленное выражение в динамической памяти, и в `GetClassInfo` передается указатель на эту строку.

Все строковые выражения, вычисленные подобным образом, должны удаляться из памяти, чтобы не было утечек. Компилятор помещает код, освобождающий эту память, в эпилог той функции, в которой встретилось выражение. В данном случае — в эпилог локальной процедуры `DoGetClassInfo`.

Освободившуюся память менеджер памяти не сразу возвращает системе, а придерживает, чтобы иметь возможность быстрее выделить память при следующем запросе. Таким образом, после завершения работы `DoGetClassInfo` память, в которой хранится вычисленное имя оконного класса (и на которую указывает `CI.lpszClassName`), по-прежнему принадлежит процессу, но менеджер памяти полагает ее свободной и считает себя вправе использовать ее по своему усмотрению.

Когда присваивается значение переменной `S`, для размещения новой строки менеджер памяти выделяет ту самую область, в которой ранее хранилось имя класса. Так как `CI.lpszClassName` по-прежнему содержит этот адрес, обращение к этому полю возвращает новую строку, которая присвоена переменной `S`.

Примечание

В Delphi до 7-й версии включительно описанный эффект наблюдается при любой длине строки, присваиваемой переменной `S`, в более новых версиях Delphi — только в том случае, если длина этой строки находится в пределах от 5 до 11 символов. Это связано с тем, что новый менеджер памяти, появившийся в этих версиях Delphi, с целью уменьшения фрагментации разбивает кучу на несколько областей, в каждой из которых выделяет блоки памяти, укладывающиеся в соответствующий данной области диапазон размеров блоков. Если строка, присваиваемая переменной `S`, слишком сильно отличается по размеру от `'TForm1'`, для этой строки выделяется память в другой области, и подмены не происходит.

Если в данном примере не выносить вызов функции `GetClassInfo` в отдельную процедуру `DoGetClassInfo`, а вызывать ее напрямую из `Button1Click`, описанного эффекта не будет, потому что в этом случае освобождение памяти, занятой для вычисленного имени класса, будет производиться в эпилоге `Button1Click`, и на момент присваивания значения переменной `S` эта память будет считаться занятой, поэтому для `S` менеджер памяти выделит другую область.

Принципиально и то, что в обоих случаях (в функции `GetClassInfo` и при присваивании значения переменной `S`) используются не строковые литералы, а выражения, вычисляемые только на этапе выполнения программы. Строковые литералы размещаются компилятором в сегменте кода, и указатели, переданные в `GetClassInfo` и присвоенные переменной `S`, будут указывать не на динамическую память, а на эти литералы, и подмены не произойдет.

Избежать проблемы можно двумя способами. Во-первых, не следует передавать значение поля `lpszClassName` за пределы той функции, в которой была вызвана `GetClassName`. Во-вторых, имя оконного класса должно быть известно программе до вызова `GetClassName`. Лучше использовать ту строку, в которой хранится это имя, чем поле `lpszClassName`.

3.4.5. Ошибка EReadError при использовании вещественных свойств

Если в секции `published` компонента имеются свойства вещественного типа (`Single`, `Double` или `Extended`), то попытка присвоить в режиме проектирования формы этим свойствам некоторые вполне корректные значения приводит к ошибке EReadError при чтении ресурсов формы (т. е. при создании формы). Для типов `Double` и `Extended` ошибка возникает, если значение свойства `X` лежит в одном из указанных диапазонов:

```
-1e15 < X <= MinInt – 1
```

или

```
MaxInt + 1 <= X < 1e15
```

Не совсем понятно, при чем здесь значения `MaxInt` и `MinInt`, если речь идет о вещественных числах, но проблема существует.

Типу `Single` не хватает точности, чтобы передавать значения `MaxInt` и `MinInt` без искажений. Тем не менее, с поправкой на уменьшение точности границ диапазонов, эта же ошибка возникает и для свойств типа `Single`.

Ошибка возникает только в случае текстовой формы dfm-файла (все версии Delphi, начиная с пятой, по умолчанию используют эту форму). При бинарной форме dfm-файла ошибки не происходит.

Ошибка обнаружена в Delphi 5 и 6, причем в Delphi 5 попытка ввести значение из указанного диапазона также может привести к ошибке и в режиме проектирования, при переключении между текстом модуля и формой. В Delphi 6 были замечены ошибки только при запуске программы, в режиме проектирования они не возникали. В Delphi 7 эта проблема уже решена, указанные значения свойств не приводят к ошибкам. В более ранних версиях Delphi проблема, естественно, также отсутствует, потому что в них dfm-файл всегда представляется в бинарной форме.

Для решения проблемы могут быть рекомендованы два способа:

1. Обновить Delphi до седьмой (или более поздней) версии.
2. Выбрать бинарную форму dfm-файла. Для этого нужно щелкнуть правой кнопкой мыши на форме и в открывшемся меню убрать галочку с пункта **Text DFM**.

Можно также отказаться от присвоения проблемных значений свойствам в режиме проектирования и присваивать их во время выполнения программы.

3.4.6. Ошибка List index out of bounds при корректном значении индекса

Windows позволяет с каждой строкой списка элементов управления `ListBox` и `ComboBox` связать либо число, либо указатель (точнее — некоторую четырехбайтную величину, которую программа может трактовать как число, как указатель или как что-либо еще). В VCL эта возможность обеспечивает привязку к строкам списка объектов (четырехбайтная величина по умолчанию трактуется как `TObject`). Доступ к этим объектам осуществляется через свойства `TComboBox.Items.Objects[Index]` и `TListBox.Items.Objects[Index]`.

Иногда все-таки требуется привязывать к строкам не объекты, а числа. Для этого можно воспользоваться явным приведением типов, например:

```
ComboBox1.Items.Objects[I] := TObject(17);
I := Integer(ComboBox1.Items.Objects[I]);
```

Если таким образом связать со строкой значение –1, то при попытке получить его во всех версиях Delphi до 7-й включительно возникнет исключение `EStringListError` с комментарием "List index out of bounds". Рассмотрим следующий код (листинг 3.56, пример ListIndex на компакт-диске).

Листинг 3.56. Пример кода, вызывающего исключение `EStringListError`

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  ComboBox1.Items.Clear;
  ComboBox1.Items.AddObject('Text', TObject(-1));
  I := Integer(ComboBox1.Items.Objects[0]);         { * }
  Label1.Caption := IntToStr(I);
end;
```

Исключение возникнет при попытке выполнить строку, отмеченную звездочкой, хотя очевидно, что индекс в данном случае корректен.

Чтобы понять причину ошибки, необходимо рассмотреть, как осуществляется чтение значения, привязанного к строке, на уровне Windows API. Рассмотрим это на примере `TComboBox`. Для получения значения необходимо послать окну `ComboBox` сообщение `CB_GETITEMDATA`. Результатом обработки этого сообщения

будет значение, связанное с указанной строкой, или CB_ERR, если при обработке сообщения возникнет ошибка. При этом документация не уточняет, какие именно ошибки могут в принципе возникнуть и как узнать, какая из них произошла.

Метод TComboBoxStrings.GetObject, через который читается значение свойства Objects, в Delphi 7 и более ранних версиях интерпретирует получение CB_ERR однозначно: генерирует исключение EStringListError с комментарием "List index out of bounds".

Проблема заключается в том, что константа CB_ERR имеет численное значение –1. Поэтому и в случае ошибки, и в случае, когда строке сопоставлено значение –1, системный обработчик сообщения CB_GETITEMDATA вернет одинаковый результат. И метод TComboBoxStrings.GetObject интерпретирует его как ошибку. (А что ему еще остается делать?)

Аналогичная проблема по тем же причинам возникает и для ListBox (аналогичная по смыслу константа LB_ERR также имеет значение –1). Это прямое следствие документированных особенностей работы Windows и модели работы VCL встречается во всех версиях Windows. Та же проблема возникает при попытке указать значение 4 294 967 295, т. к. на двоичном уровне это число записывается той же комбинацией битов, что и –1.

При использовании свойства Objects по прямому назначению, т. е. для хранения объектов, эта проблема не может возникнуть, потому что $FFFFFFFF — это адрес самого старшего байта в четырехгигабайтном виртуальном адресном пространстве программы. Эта область адресного пространства зарезервирована системой, и менеджер памяти Delphi не может выделить память для объекта в этой области.

Рекомендуемые способы решения проблемы:

1. Пересмотреть алгоритм и отказаться от связывания значения –1 со строками.
2. Напрямую посылать CB_GETITEMDATA окну ComboBox, а попадание индекса в диапазон контролировать самостоятельно другими методами.

Приведенный в листинге 3.57 код иллюстрирует последний совет.

Листинг 3.57. Получение связанного с элементом значения вручную

```
procedure TForm1.Button2Click(Sender: TObject);
var
  I: Integer;
begin
  ComboBox1.Items.Clear;
```

```
  ComboBox1.Items.AddObject('Text', TObject(-1));
  I := SendMessage(ComboBox1.Handle, CB_GETITEMDATA, 0, 0);
  Label1.Caption := IntToStr(I);
end;
```

Как уже было отмечено ранее, в BDS 2006 и более поздних версиях исключение не возникает. Это связано с новой реализацией метода `TCustomComboBoxStrings.GetObject`, который отвечает за получение значения свойства `Items.Object` (листинг 3.58).

Листинг 3.58. Получение значения свойства `Items.Object` в BDS 2006 и выше

```
function TCustomComboBoxStrings.GetObject(Index: Integer): TObject;
begin
  Result := TObject(SendMessage(ComboBox.Handle, CB_GETITEMDATA,
    Index, 0));
  // Do additional checking on Count and Index here is so in the event
  // the object being retrieved is the integer -1 the call will succeed
  if (Longint(Result) = CB_ERR) and ((Count = 0) or (Index < 0) or
    (Index > Count)) then
    Error(SListIndexError, Index);
end;
```

Решение спорное, т. к. проверка корректности системой дополняется собственной проверкой индекса, и не совсем понятно, что делать в том случае, если система фиксирует какую-либо ошибку, не связанную с индексом. Но здесь Windows ставит разработчика в такие условия, что любое решение будет спорным, так что упреком по отношению к разработчикам VCL такая оценка их решения не является.

В таких элементах управления, как `TListView` и `TTreeView`, тоже существует возможность связывания 4-байтного значения с элементом (см. свойства `TTreeNode.Data`, `TListItem.Data`), но сообщения `TVM_GETITEM` и `LVM_GETITEM`, через которые можно получить значения этих свойств, устроены иначе, поэтому связывание с элементом значения –1 (а также любого другого 4-байтного значения) не приводит к аналогичным проблемам.

3.4.7. Неправильное поведение свойства *Anchors*

Свойство `Anchors`, появившееся в Delphi 5, является очень удобным средством управления положением и размерами визуальных компонентов при изменении размера родителя. Однако в тех случаях, когда начальные размеры формы по каким-то причинам не совпадают с установленными при проекти-

ровании, задание значения свойства Anchors не приносит желаемого эффекта: первоначальное расположение визуальных компонентов на форме соответствует размерам, установленным при проектировании, а не тем, которые реально получила форма. Примеры такого некорректного поведения демонстрирует программа WrongAnchors на компакт-диске.

Программа WrongAnchors — это MDI-приложение, в котором открываются две дочерние формы разных классов: ChildForm1 (класс TChildForm1) и ChildForm2 (класс TChildForm2). Во время проектирования эти две формы выглядят совершенно одинаково, но при запуске программы только вторая форма сохраняет заданные при проектировании размеры, а первая становится больше. При этом панель, лежащая на ней, не адаптирует свои размеры к изменившемуся размеру формы, хотя свойство Anchors обязывает ее к этому (это легко видеть, изменяя размеры формы после ее создания). Самый простой способ борьбы с этой неприятностью — заставить дочернюю MDI-форму иметь такой же начальный размер, какой задан при проектировании.

Дочерняя MDI-форма приобретает отличный от заданного размер потому, что метод CreateParams для ширины и высоты окна устанавливает не те значения, которые хранятся в свойствах Width и Height, а значение CW_USEDEFAULT. Это значение говорит системе, что она должна выбрать размеры окна на свое усмотрение. Чтобы этого не происходило, нужно вновь вернуть установленные при проектировании значения ширины и высоты в перекрытом методе CreateParams. Именно этим класс TChildForm2 отличается от TChildForm1 (листинг 3.59).

Листинг 3.59. Установка значений ширины и высоты, заданных при проектировании

```
procedure TChildForm2.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  Params.Width := Width;
  Params.Height := Height;
end;
```

Значение CW_USEDEFAULT присваивается ширине и высоте окна не только в том случае, если это дочерняя MDI-форма, но и когда значение свойства Position формы равно poDefault или poDefaultSizeOnly. Но в этом случае перекрывать CreateParams нет нужды, достаточно просто изменить значение свойства Position на другое. Просто необходимо помнить, что если свойство Position формы имеет одно из этих значений, свойства Anchors лежащих на форме компонентов должны иметь значения по умолчанию.

Другой случай, когда окно может при создании иметь размеры, отличные от заданных при проектировании, — это когда свойство `WindowState` равно `wsMaximized`. При этом окно растягивается на весь экран. В примере WrongAnchors в главном меню есть пункты **Развернутое окно 1** и **Развернутое окно 2**, которые открывают диалоговые окна, развернутые на весь экран. Но в первом из этих окон панель опять не адаптируется к новым размерам окна, в то время как во втором — адаптируется, хотя значения свойства `Anchors` у обеих панелей одинаковые. Это происходит потому, что в первом случае значение `wsMaximized` присваивается свойству `WindowState` во время проектирования, и поэтому окно сразу создается развернутым. А во втором случае значение `wsMaximized` присваивается свойству `WindowState` только при обработке события `OnShow` формы, т. е. тогда, когда форма уже создана с заданными при проектировании размерами, но еще не видна на экране. При этом свойство `Anchors` работает так, как требуется. Это и есть решение проблемы — значение свойству `WindowState` нужно присваивать не во время проектирования, а в обработчике события `OnShow`.

Но самое интересное происходит, если свойство `WindowState` во время проектирования получило значение `wsMaximized`, а свойство `Position` — значение `poDefault` или `poDefaultSizeOnly`. Тогда размеры и положения визуальных компонентов на форме будут адаптированы к размеру, который не совпадает ни с размером развернутой формы, ни с размером, заданным во время проектирования. Если такой форме отменить развертывание на весь экран, то визуальные компоненты получат размеры и положения, установленные в режиме проектирования.

Нельзя сказать, что разработчики Delphi не знакомы с этой проблемой, они даже что-то делают, чтобы ее решить. Начиная с BDS 2006 можно устанавливать значение свойства `WindowState` в режиме проектирования, и визуальные компоненты на такой форме будут вести себя интуитивно ожидаемым образом, т. е. адаптироваться к размеру формы, растянутой на весь экран. Правда, с двумя существенными оговорками. Во-первых, свойство `Position` формы не должно быть равно `poDefault` или `poDefaultSizeOnly`. Во-вторых, это относится только к главной форме приложения, для всех остальных форм проблема сохраняется. Поэтому пример WrongAnchors будет работать одинаково и в новых версиях Delphi, и в старых — там на весь экран разворачиваются не главные формы.

3.4.8. Ошибка при сравнении указателей на метод

Процедурные типы в Delphi делятся на обычные (унаследованные от Turbo Pascal) и указатели на методы. Первые — это указатели на простые процедуры и функции, вторые — на методы объектов. Чтобы вызвать метод

объекта, недостаточно знать, где его код располагается в памяти, нужно еще иметь ссылку на конкретный экземпляр класса, к которому относится данный метод (т. е. необходимо значение указателя `Self`, который будет передан в данный метод). Поэтому указатели на методы называются указателями лишь условно: на самом деле это не один указатель, а два (на код и на объект). Размер переменных такого типа равен 8 байтам, в чем нетрудно убедиться с помощью функции `SizeOf`.

Очевидно, что два указателя на метод равны тогда и только тогда, когда указывают на один и тот же метод одного и того же объекта, т. е. входящие в них указатели попарно равны. Однако компилятор сравнивает указатели на методы неправильно, и пример MethodPtrCmp на компакт-диске демонстрирует это. На форме этого примера расположены две кнопки класса `TButton`. Обработчик нажатия на первую из них выглядит так, как в листинге 3.60.

Листинг 3.60. Пример неправильного сравнения указателей на метод

```
procedure TForm1.Button1Click(Sender: TObject);
var
  P1, P2: procedure of object;
begin
  P1 := Button1.Update;
  P2 := Button2.Update;
  // Здесь компилятор сравнивает указатели на методы неверно,
  // давая ошибочный результат "равно"
  if @P1 = @P2 then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

Здесь мы получаем указатели на один и тот же метод разных объектов (для примера взяты класс `TButton` и метод `Update`, но подошел бы любой класс и любой метод). Сравнение указателей в этом примере дает ошибочный результат **Равно**, хотя эти указатели не равны между собой. Просмотр кода, который генерирует компилятор, показывает, что здесь сравниваются только указатели на код метода, а указатели на объекты игнорируются. Так как у нас метод один и тот же, различаются только объекты, то и получается ошибочный результат.

Сравнить указатели на методы правильно можно с помощью типа `TMethod` из модуля `SysUtils`, объявленного следующим образом:

```
TMethod = record
  Code, Data: Pointer;
end;
```

Так можно получать доступ к отдельным указателям, входящим в указатель на метод. Сравнение указателей на метод с помощью этого типа иллюстрирует листинг 3.61.

Листинг 3.61. Правильный способ сравнения указателей на метод

```
procedure TForm1.Button2Click(Sender: TObject);
var
  P1, P2: procedure of object;
begin
  P1 := Button1.Update;
  P2 := Button2.Update;
  // Правильный способ сравнения указателей на методы
  if (TMethod(P1).Data = TMethod(P2).Data) and
     (TMethod(P1).Code = TMethod(P2).Code) then
    Label1.Caption := 'Равно'
  else
    Label1.Caption := 'Не равно';
end;
```

Здесь мы явным образом заставляем компилятор сравнивать оба указателя, поэтому получаем правильный результат **Не равно**.

3.4.9. Возможность получения адреса свойства

Пусть у нас есть класс, описанный следующим образом (листинг 3.62).

Листинг 3.62. Класс со свойствами, читаемыми из переменной и из функции

```
TSomeClass = class
private
  FProp1: Integer;
  function GetProp2: Integer;
public
  property Prop1: Integer read FProp1;
  property Prop2: Integer read GetProp2;
end;
```

В этом классе два свойства `Prop1` и `Prop2`, значение одного из которых определяется полем `FProp1`, а другого — функцией `GetProp2`. Оба свойства предназначены только для чтения, но для того эффекта, о котором здесь пойдет речь, это не принципиально: свойства, значения которых можно менять, ведут себя в этом отношении точно так же.

Пусть `X` — это переменная типа `TSomeClass`. Легко убедиться, что компилятор допускает получение адреса свойства `Prop1`, т. е. конструкция вида `@X.Prop1` считается допустимой. Результатом выполнения этого оператора станет указатель на поле `FProp1`. А вот конструкцию `@X.Prop2` компилятор не допускает, выдает ошибку Variable required.

Ошибкой компилятора здесь является то, что он допускает получение адреса в первом случае, т. е. для свойства, значение которой берется из переменной. Это грубейшее нарушение принципа инкапсуляции, лежащего в основе объектно-ориентированного программирования, причем сразу по двум причинам. Во-первых, пользователь класса не должен видеть его внутреннюю реализацию, а здесь пользователь может определить, как читается свойство, по возможности применения оператора `@` к нему. Во-вторых, пользователь класса должен взаимодействовать с ним строго через предоставленный интерфейс, а у нас получается, что, узнав адрес поля `FProp1`, пользователь сможет менять его значение в обход предусмотренных для этого в классе механизмов.

К счастью, ситуации, в которых эта недоработка компилятора могла бы принести пользу, крайне редки. Но если вы все-таки столкнулись с такой ситуацией, настоятельно рекомендуем не поддаваться соблазну и искать другие способы решения проблемы. Если класс, к полю которого вы хотите получить доступ таким образом, написан вами, то это веский повод пересмотреть внешний интерфейс класса, т. к. при его проектировании, скорее всего, были допущены серьезные ошибки. Если это "чужой" класс, подумайте о том, что в следующей версии этого класса автор может изменить реализацию свойства, и тогда ваш код откажется компилироваться.

3.4.10. Невозможность использования некоторых свойств оконного компонента в деструкторе

Проблема, о которой пойдет речь в этом разделе, гораздо шире, чем это явствует из заголовка. Однако наиболее ярко она проявляется именно в этом случае. Поэтому мы начнем именно с этой ситуации, а потом рассмотрим проблему более широко.

Проблему демонстрирует пример ParentWnd на компакт-диске. В нем создан компонент `TWrongCombo`, наследник `TComboBox`. Листинг 3.63 содержит код компонента.

Листинг 3.63. Компонент `TWrongCombo`

```
type
  TWrongCombo = class(TComboBox)
```

```
  public
    destructor Destroy; override;
    procedure AddItem(const Title: string);
  end;

destructor TWrongCombo.Destroy;
var
  I: Integer;
begin
  for I := 0 to Items.Count – 1 do
    if  Assigned(Items.Objects[I]) then
      Dispose(PDateTime(Items.Objects[I]));
  inherited;
end;

procedure TWrongCombo.AddItem(const Title: string);
var
  P: PDateTime;
begin
  New(P);
  P^ := Now;
  Items.AddObject(Title, TObject(P));
end;
```

Класс `TWrongCombo` с каждым элементом, добавленным с помощью метода `AddItem`, связывает значение типа `TDateTime`, хранящее время добавления элемента. В *разд. 3.4.6* мы уже познакомились с возможностью связывания данных с элементом списка с помощью свойства `Items.Objects`. Но так мы можем связать с элементом только 4-байтное значение, а тип `TDateTime` занимает 8 байтов. Поэтому значение `TDateTime` мы будем хранить в динамической памяти, а с элементом свяжем указатель на него.

Раз мы выделили динамическую память, ее нужно освободить при удалении компонента. Наиболее подходящим местом для этого кажется деструктор, и именно в нем помещен код освобождения выделенной памяти.

Теперь попробуем воспользоваться компонентом. На главной форме программы `ParentWnd` находится кнопка **Wrong Combo**, при нажатии на которую создается компонент типа `TWrongCombo` (листинг 3.64).

Листинг 3.64. Реакция на кнопку *Wrong Combo*

```
procedure TForm1.BtnWrongComboClick(Sender: TObject);
begin
  if FWrongCombo = nil then
```

```
  begin
    FWrongCombo := TWrongCombo.Create(Self);
    FWrongCombo.Left := 10;
    FWrongCombo.Top := 10;
    FWrongCombo.Parent := Self;
    FWrongCombo.AddItem('One');
    FWrongCombo.AddItem('Two');
    FWrongCombo.AddItem('Three');
  end;
end;
```

Теперь, если нажать эту кнопку и затем попытаться закрыть форму, в деструкторе `TWrongCombo` возникнет исключение `EInvalidOperation` с сообщением "Control has no parent window". Если откомпилировать программу с включенной опцией **Use Debug DCUs**, видно, что исключение возникает в методе `TWinControl.CreateWnd`. Одно только это способно обескуражить — действительно, зачем метод создания окна вызывается при его удалении?

Причина заключается в том, что к моменту вызова деструктора окно компонента уже удалено, свойство `Handle` имеет нулевое значение, и свойство `Parent` тоже имеет значение `nil`. Обращение к свойству `Items.Count` приводит к отправке окну сообщения `CB_GETCOUNT`. Отправка осуществляется с помощью функции `SendMessage`, одним из параметров которой является дескриптор окна, в качестве которого, естественно, передается свойство `Handle`. А это свойство, напомним, к этому моменту равно нулю. В *разд. 1.1.7* обсуждалось, что обращение к этому свойству в тот момент, когда оно равно нулю, приводит к попытке создания окна (см. листинг 1.8). Именно поэтому вызывается метод `CreateWnd`. И он возбуждает исключение, потому что окно, которое создает компонент `TWrongCombo`, имеет стиль `WS_CHILD`, т. е. не может не иметь родителя. А родитель компоненту не назначен, поэтому и возникает исключение с таким странным, на первый взгляд, сообщением.

Отсюда следует важный вывод, что никакое обращение в деструкторе компонента к тем свойствам и методам, которые требуют наличия окна, невозможно. Окно уже удалено, а попытка создания нового окна приведет к исключению. Поэтому, например, в нашем коде требуется искать другое место, чтобы корректно освободить занятую память.

Поиск этого места оказывается не такой простой задачей, как хотелось бы, потому что разработчики VCL весьма странным образом реализовали удаление дочерних оконных компонентов в деструкторе класса `TWinControl`: сначала вызывается системная функция `DestroyWindow`, которая удаляет и само окно, и, разумеется, все его дочерние окна, а потом только дочерние компоненты начинают уведомляться о том, что их удаляют, т. е. к этому моменту они

уже не имеют возможности как-то задействовать свои окна. Соответственно, в нашем случае деструктор формы уничтожает окна всех дочерних компонентов до того, как будут вызваны деструкторы этих компонентов.

Положение спасает то, что об уведомлении окон заботится система Windows: всем окнам, которые удаляются в результате вызова функции DestroyWindow, отправляется сообщение WM_DESTROY, причем в момент получения этого сообщения ни окно, ни его родитель еще не уничтожены. Это позволяет компоненту как-то реагировать на свое удаление до того, как окно будет уничтожено.

Казалось бы, выход найден: нужно освобождать память в обработчике сообщения WM_DESTROY. Но и тут не все так просто. Дело в том, что окно может уничтожаться не только при удалении компонента, но и при изменении некоторых свойств (например, Parent). При этом окно удаляется, а вместо него создается новое, и при удалении старого окна компонент тоже получает сообщение WM_DESTROY. Что же касается компонента TComboBox, он обеспечивает, что при удалении и последующем создании окна все элементы, в том числе связанные с ними значения, восстанавливаются. Таким образом, если мы в наследнике TComboBox в обработчике сообщения WM_DESTROY всегда будем освобождать выделенную память, после повторного создания окна получим "битые" ссылки в свойстве Items.Objects, чего, естественно, хотелось бы избежать. Требуется научиться отличать полное удаление компонента от удаления окна с целью повторного создания.

Вообще говоря, механизм для этого предусмотрен в VCL — это флаг csDestroying в свойстве ComponentState. Выполнение деструктора TWinControl.Destroy начинается с вызова метода Destroying, добавляющего этот флаг во всех компонентах, которыми владеет данный компонент. Однако по наличию этого флага у компонента мы не можем в обработчике WM_DESTROY узнать, удаляется весь компонент, или только окно для создания заново. Рассмотрим, например, ситуацию, когда на форму во время проектирования разработчик положил панель, а на эту панель — любой оконный компонент, например, кнопку. Владельцем кнопки в этом случае все равно является форма, а панель — только родителем. Если теперь удалить панель, не удаляя форму, метод Destroying панели не затронет кнопку, и на момент получения кнопкой сообщения WM_DESTROY флаг csDestroying у нее еще не будет установлен, несмотря на то, что кнопка удаляется.

Тем не менее флаг csDestroying все же может помочь нам. Компонент удаляется в одном из трех случаев:

1. Удаляется непосредственно данный компонент.
2. Удаляется владелец компонента.
3. Удаляется родитель компонента.

В первом случае удаление начинается не с удаления окна, а с вызова деструктора компонента, и окно компонент удаляет уже сам, когда флаг csDestroying уже установлен деструктором. Во втором случае деструктор владельца, прежде чем удалить окно, заботится о том, чтобы компонент получил флаг csDestroying, поэтому даже если владелец является одновременно и родителем, флаг у компонента в момент удаления окна уже будет. И, наконец, остается третья ситуация, в которой флага csDestroying у компонента может и не быть. Но в любом случае удаление цепочки компонентов начинается с вызова деструктора "главного" из них. По линии владельца флаг csDestroying передается, по линии родителя — нет, но самый верхний из цепочки родителей обязательно имеет такой флаг. Соответственно, чтобы определить, удаляется ли окно из-за уничтожения визуального компонента, нужно искать флаг csDestroying не только у самого компонента, но и у всей цепочки его родителей. Если флаг нигде не найден, значит, удаляется только окно, но не сам компонент.

На главном окне примера ParentWnd есть также кнопка **Right Combo**, которая создает на форме визуальный компонент типа TRightCombo. Это правильный вариант класса TWrongCombo, в котором деструктор не переопределяется, а обработчик сообщения WM_DESTROY реализован в соответствии с тем, что написано ранее (листинг 3.65).

Листинг 3.65. Обработчик сообщения WM_DESTROY класса TRightCombo

```
procedure TRightCombo.WMDestroy(var Msg: TMessage);
var
  I: Integer;
  FinalDestruction: Boolean;
  P: TControl;
begin
  FinalDestruction := False;
  P := Self;
  while Assigned(P) do
  begin
    if csDestroying in P.ComponentState then
    begin
      FinalDestruction := True;
      Break;
    end;
    P := P.Parent;
  end;
  if FinalDestruction then
    for I := 0 to Items.Count – 1 do
```

```
      Dispose(PDateTime(Items.Objects[I]));
  inherited;
end;
```

Такой компонент корректно освобождает память при его удалении, но не освобождает ее тогда, когда окно создается заново.

Примечание

Есть еще одна очень распространенная причина получения ошибки "Control has no parent window" при разработке собственных компонентов — попытка обращения к свойствам, требующим наличия окна, до назначения свойства `Parent`. Например, такая ошибка появилась бы, если бы мы в наших наследниках `TComboBox` попытались при создании добавить элементы, вызвав в конструкторе метод `AddItem`. Свойство `Items.Objects` в случае `TComboBox` работает через оконные сообщения `CB_GETITEMDATA` и `CB_SETITEMDATA`, при попытке отправить которые будет использовано свойство `Handle`. Это также приведет к попытке создания окна, которая завершится исключением из-за отсутствия родителя. Другими словами, ошибку мы получим не при удалении компонента, а при его создании. Бороться с этой проблемой можно, выполняя начальную инициализацию тогда, когда родитель уже назначен, например, в перекрытом методе `SetParent` после того, как отработает унаследованный `SetParent`. Необходимо только помнить, что `SetParent` может быть вызван не только при создании компонента и при необходимости позаботиться о том, чтобы инициализация выполнялась только при первом вызове `SetParent` с аргументом, отличным от `nil`.

Чтобы убедиться, насколько некорректно реализовано удаление компонентов в VCL, рассмотрим еще один пример (на компакт-диске он называется FrameDel). В этом примере на форму помещается фрейм с одним компонентом типа `TComboBox`. Код фрейма показан в листинге 3.66.

Листинг 3.66. Код фрейма

```
type
  TFrame1 = class(TFrame)
    ComboBox1: TComboBox;
  private
    { Private declarations }
  public
    destructor Destroy; override;
    procedure AddComboItem;
  end;

destructor TFrame1.Destroy;
var
  I: Integer;
```

```
begin
  for I := 0 to ComboBox1.Items.Count – 1 do
    if Assigned(ComboBox1.Items.Objects[I]) then
      Dispose(PDateTime(ComboBox1.Items.Objects[I]));
  inherited;
end;

procedure TFrame1.AddComboItem;
var
  P: PDateTime;
begin
  New(P);
  P^ := Now;
  ComboBox1.Items.AddObject('Item ' + TimeToStr(P^), TObject(P));
end;
```

На форму в обработчике события `OnShow` помещается такой фрейм и вызывается его метод `AddComboItem`, чтобы в компоненте `ComboBox1` появился один элемент в списке. Если закрыть такую форму, никаких исключений не возникает, все выглядит нормально. Но при трассировке можно заметить, что цикл внутри деструктора не выполняется ни разу, потому что `ComboBox1.Items.Count` возвращает 0. Это происходит потому, что на момент вызова этого деструктора и окно фрейма, и окно `ComboBox1` уже не существуют, в чем легко убедиться, проверив в деструкторе значение поля `ComboBox1.FHandle` (до обращения к свойству `ComboBox1.Items.Count` оно равно нулю). А при обращении к этому свойству происходит попытка создать окно. Так как свойство `TComboBox1.Parent` в этот момент еще не обнулено, предпринимается попытка создать заново и фрейм тоже, и эта попытка становится успешной. К этому моменту свойство `Parent` фрейма уже обнулено, но метод `TCustomFrame.CreateParams` реализован таким образом, что родителем всех фреймов, для которых родитель не задан явно, становится невидимое окно приложения, которое на этот момент еще не разрушено. Таким образом, окно фрейма и окно компонента `TComboBox1` успешно создаются заново, и им можно посылать сообщения.

Ранее мы говорили, что код компонента `TComboBox` обеспечивает перенос элементов при удалении и последующем создании окна. Но в данном случае этот код даже не догадывается, что после удаления окно может быть создано еще раз, и потому механизм переноса не задействуется. Вновь созданное окно компонента `ComboBox1` не получает в свой список ни одного элемента, что и приводит к тому, что свойство `Items.Count` равно нулю. Но динамическая память, выделенная в методе `AddComboItem`, остается не освобожденной. В результате имеем утечку памяти вместо исключения. Кроме того, имеем утечку

и других ресурсов, т. к. код, ответственный за удаление окна фрейма, на этот момент уже отработал и не будет запущен еще раз, чтобы удалить вновь созданное окно.

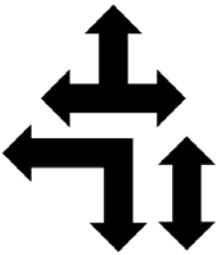
Решением проблемы может стать уже опробованный способ: нужно обрабатывать сообщение `WM_DESTROY`, посылаемое фрейму, выполнять в нем все те же проверки, что и в листинге 3.65, и при необходимости освобождать память, которую нельзя освободить в деструкторе.

Примечание

Если бы мы попытались использовать наследника от класса, например, `TPanel` вместо `TFrame`, ошибка при завершении работы программы возникла бы, т. к. компоненту `TPanel` не назначается родитель по умолчанию, и попытка создания его окна в деструкторе закончилась бы неудачей. Назначение родителя по умолчанию приводит еще к одному интересному эффекту: мы можем добавлять в `ComboBox1` элементы до того, как будет назначено свойство `Parent` фрейма. Ошибки не возникнет, потому что окно фрейма будет создано успешно, а при последующем назначении свойства `Parent` фрейма в компоненте `ComboBox1` сработает механизм переноса элементов при создании нового окна, и пользователь увидит добавленные элементы.

Главным выводом этого раздела должно стать то, что последовательность удаления визуальных компонентов в VCL очень плохо продумана (видимо, это одно из самых неудачных мест в VCL), поэтому нужно соблюдать особую осторожность в тех случаях, когда освобождение ресурсов удаляемого оконного компонента может быть связано с обращением к его окну. Деструктор для этих целей, как мы убедились, не подходит, удаление следует выполнять в обработчике `WM_DESTROY`, проверяя, действительно ли удаляется сам компонент, а не только его окно.

ГЛАВА 4



Разбор и вычисление выражений

Перед программистом нередко возникает задача вычисления арифметических или иных выражений, не известных на этапе компиляции программы. Готовых средств для этого в Delphi нет. В Интернете можно найти компоненты и законченные примеры вычисления выражений, но нередко требуется создать что-то свое. В этой главе мы рассмотрим способы программного разбора и вычисления арифметических выражений. Кроме самих примеров будут изложены основы теории синтаксического анализа, с помощью которой эти примеры написаны. Эти сведения не только помогут лучше понять приведенные примеры, но и позволят легко написать код для синтаксического разбора любых выражений, которые подчиняются некоторому формальному описанию синтаксиса.

Синтаксический анализатор мы будем создавать поэтапно, переходя от простых примеров к сложным. Сначала научимся распознавать вещественное число и напишем простейший калькулятор, который умеет выполнять четыре действия арифметики над числами без учета приоритета операций. Затем наша программа научится учитывать приоритет этих операций, а чуть позже — использовать скобки для изменения этого приоритета. Далее калькулятор обретет способность работать с переменными, вычислять функции и возводить в степень, т. е. станет вполне полноценным. И на последнем этапе мы добавим лексический анализатор — средство, которое формализует разбор выражений со сложной грамматикой и тем самым существенно облегчает написание синтаксических анализаторов.

4.1. Синтаксис и семантика

Прежде чем двигаться дальше, введем базовые определения. *Языком* мы будем называть множество строк (в большинстве случаев это будет бесконеч-

ное множество). Каждое *выражение* (в некоторых источниках вместо "выражение" употребляются термины "предложение" или "утверждение") может принадлежать или не принадлежать языку. Например, определим язык так: любая строка произвольной длины, состоящая из нулей и единиц. Тогда выражения "000101001" и "1111" принадлежат языку, а выражения "5x" и "R@8" — нет.

Синтаксисом называется набор правил, которые позволяют сделать заключение о том, принадлежит ли заданное выражение языку или нет.

С практической точки зрения наиболее интересны те языки, выражения которых не только подчиняются каким-либо синтаксическим правилам, но и несут смысловую нагрузку. Например, выражения языка Delphi — программы — приводят к выполнению компьютером тех или иных действий. В данном случае семантика языка Delphi — это правила, определяющие, к выполнению каких именно действий приведет то или иное выражение. В более общем смысле *семантика языка* — это описание смысла языковых выражений.

Другими словами, синтаксические правила позволяют понять, допустимо ли в выражении, принадлежащем заданному языку, появление в указанной позиции данного символа, а семантические — что означает появление этого символа в данной позиции.

Чтобы подчеркнуть разницу между синтаксисом и семантикой, рассмотрим такой оператор присваивания в Delphi: `X := Y + Z;`. С точки зрения синтаксиса это правильное выражение, т. к. требования синтаксиса заключаются в том, чтобы слева от знака присваивания стоял корректный идентификатор, справа — корректное выражение. Очевидно, что эти правила выполнены. Но с точки зрения семантики это выражение может быть ошибочным, если, например, один из встречающихся в нем идентификаторов не объявлен, или их типы несовместимы, или же идентификатор `X` объявлен как константа. Таким образом, синтаксически правильное выражение не всегда является семантически верным. Примером подобного арифметического выражения может служить "0 / 0" — два корректных числа, между которыми стоит допустимый знак операции, т. е. синтаксически все верно. Однако смысла такое выражение не имеет, т. к. данная операция неприменима к указанным операндам.

Таким образом, синтаксический анализ арифметических выражений — это всего лишь выяснение, корректно ли выражение. Мы же говорили о вычислении выражений, а это уже относится к семантике, т. е., строго говоря, мы здесь будем заниматься не только синтаксическим, но и семантическим анализом. С точки зрения теории синтаксический и семантический анализ разделены, т. е. анализировать семантику можно начинать "с нуля" после того, как анализ синтаксиса закончен. Но на практике легче объединить эти два про-

цесса в один, чтобы пользоваться результатами синтаксического разбора при семантическом анализе. Из-за этого, как мы увидим в дальнейшем, иногда приходится вводить сложные синтаксические правила, которые в итоге описывают тот же язык, что и более простые, чтобы упростить семантический анализ.

На примере выражения `X := Y + Z;` мы могли наблюдать интересную особенность: для заключения о синтаксической корректности или некорректности отдельной части выражения языка нам достаточно видеть только эту часть, в то время как для выяснения ее семантической корректности необходимо знать "предысторию", т. е. то, что было в выражении раньше. Это объясняется следующим образом: существуют формальные способы описания синтаксиса, позволяющие выделить отдельные синтаксические конструкции. В принципе, язык может использовать другие синтаксические правила, не позволяющие однозначно выделить отдельные конструкции и, соответственно, сделать вывод о допустимости вырванной из контекста строки (примером такого языка является FORTRAN, особенно его ранние версии), но на практике такой синтаксис неудобен, поэтому при разработке языков конструкции стараются все-таки выделять. Это облегчает как чтение программы, так и создание трансляторов языка.

Что касается семантики, то формальные правила ее описания отсутствуют. Поэтому семантика описывается словами, или же язык использует интуитивно понятную семантику. Например, арифметическое выражение "2+2" выглядит очень понятно в силу того, что мы к нему привыкли, хотя с точки зрения математики объяснить, что такое число и что такое операция сложения двух чисел, не так-то просто.

Кроме синтаксического и семантического анализа существует еще и лексический анализ — разделение выражения на отдельные лексемы. *Лексемами* называются последовательности символов языка, которые имеют смысл только как единое целое. Например, выражение "2+3" не относится к лексемам, т. к. его части — "2", "3" и "+" — имеют значение и вне выражения, а смысл всего выражения будет суперпозицией значений этих частей. А вот идентификатор `TForm` является лексемой, т. к. его невозможно разделить на имеющие смысл части. Таким образом, лексема — это синтаксическая единица самого нижнего уровня. Описание лексических правил может быть обособлено от синтаксических, и тогда сначала лексический анализатор выделяет из выражения все лексемы, а потом синтаксический анализатор проверяет правильность выражения, составленного из этих лексем. Попутно лексический анализатор может удалять из выражения комментарии, лишние разделители и т. п.

Для разбора простого синтаксиса нет нужды проводить отдельный лексический анализ, лексемы выделяются непосредственно при синтаксическом ана-

лизе. Поэтому большинство примеров, приведенных далее, будет обходиться без лексического анализатора.

4.2. Формальное описание синтаксиса

Существует несколько различных (но, тем не менее, эквивалентных) способов описания синтаксиса. Мы здесь познакомимся только с самой употребляемой из них — *расширенной формой Бэкуса — Наура*. Эта форма была предложена Джоном Бэкусом и немного модифицирована Питером Науром, который использовал ее для описания синтаксиса языка Алгол. (Примечательно, что практически идентичная форма была независимо изобретена Ноамом Хомски для описания синтаксиса естественных языков.) В русскоязычной литературе форму Бэкуса — Наура обычно обозначают аббревиатурой *БНФ* (Бэкуса — Наура Форма). Несколько неестественный для русского языка порядок слов принят, чтобы сохранилось сходство с английской аббревиатурой BNF (Backus-Naur Form). Со временем в БНФ были добавлены новые правила описания синтаксиса, и эта форма получила название *РБНФ* — расширенная БНФ (далее для краткости мы не будем делать различия между БНФ и РБНФ). Совокупность правил, записанных в виде БНФ (или другим формализованным способом), называется *грамматикой языка*.

Основные понятия БНФ — терминальные и нетерминальные символы. *Терминальные символы* — это отдельные символы или их последовательности, являющиеся с точки зрения синтаксиса неразрывным целым, не сводимым к другим символам. Другими словами, терминальные символы — это лексемы. Терминальные символы могут состоять из одного или нескольких символов в обычном понимании этого слова. Примером терминальных символов, состоящих из нескольких символов, могут служить зарезервированные слова языка Паскаль и символы операций >=, <= и <>. Чтобы отличать терминальные символы от служебных символов БНФ, мы будем заключать их в одинарные кавычки.

Нетерминальный символ — это некоторая абстракция, которая по определенным правилам сводится к комбинации терминальных и/или других нетерминальных символов. Правила должны быть такими, чтобы существовала возможность выведения из них выражения, полностью состоящего из терминальных символов, за конечное число шагов, хотя рекурсивные определения терминальных символов друг через друга или через самих себя допускаются. Нетерминальные символы имеют имена, которые принято обрамлять угловыми скобками: `<Operator>`.

Операция `::=` означает определение нетерминального символа. Слева от этого знака ставится нетерминальный символ, смысл которого надо определить,

справа — комбинация символов, которой соответствует данный нетерминальный символ. Примером может служить следующее определение:

```
<Separator> ::= '.'
```

В данном примере мы определили нетерминальный символ <Separator>, который можем использовать в дальнейшем, например, при описании синтаксиса записи вещественного числа. Если мы затем захотим поменять разделитель с точки на запятую, нам достаточно будет переопределить смысл символа <Separator>, а не менять определения всех остальных символов, где встречается этот разделитель.

В более сложных случаях нетерминальному символу ставится в соответствие не один символ, а их цепочка, в которую могут входить как терминальные, так и нетерминальные символы. Примером такого определения может служить описание синтаксиса оператора присваивания в Delphi:

```
<Assignment> ::= <Var> ':=' <Expression>
```

При записи синтаксиса в БНФ часто сначала дают определение абстракции самого верхнего уровня, описывающей все выражение в целом, и только потом — определения абстракций нижнего уровня, которые необходимы при ее определении, т. е. порядок определения абстракций может отличаться от принятого в языках программирования определения идентификаторов, согласно которому идентификатор должен быть сначала описан, и лишь затем использован. В частности, в данном примере символы <Var> (переменная) и <Expression> (выражение) могут быть определены после определения <Assignment>.

Операция | в БНФ означает "или" — показывает одну из двух альтернатив. Например, если под нетерминальным символом <Sign> может подразумевать знак "+" или "–", его определение будет выглядеть следующим образом:

```
<Sign> ::= '+' | '-'
```

Если альтернатив больше, чем две, они записываются в ряд, разделенные символом |, например:

```
<Digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Здесь мы определили нетерминальный символ <Digit> (цифра), под которым можем понимать один из символов диапазона '0'..'9'.

Операция | подразумевает, что все, что стоит слева от этого знака, является альтернативой того, что стоит справа (до конца определения или до следующего символа |). Если в качестве альтернативы выступает только часть определения, то чтобы обособить эту часть, ее заключают в круглые скобки, например:

```
<for> ::= 'for' <Var> ':=' <Expression>
  ('to' | 'downto') <Expression> 'do' <Operator>
```

Здесь с помощью БНФ описан синтаксис оператора `for` языка Delphi.

В квадратные скобки заключается необязательная часть определения, как присутствие, так и отсутствие которой допускается синтаксисом, например:

```
<if> ::= 'if' <Condition> 'then' <Operator> ['else' <Operator>]
```

Здесь дано определение условного оператора `if` языка Delphi. Квадратные скобки указывают на необязательность части `else`.

Примечание

Строго говоря, определения операторов `if` и `for` в Delphi сложнее, чем те, которые мы здесь привели. Это связано с тем, что `<if>` и `<for>` — это альтернативы символа `<Operator>`. Поэтому может возникнуть конструкция типа `if Condition1 then if Condition2 then Operator1 else Operator2`. Из нашего определения невозможно сделать вывод о том, к какому из двух `if` в данном случае относится `else`. В языках программирования принято, что `else` относится к последнему из `if`, который еще не имеет `else`. Чтобы описать это правило, требуется более сложный синтаксис, чем мы здесь привели. Однако этот вопрос выходит за рамки данной книги и более подробно рассмотрен в [5].

Фигурные скобки означают повторение того, что в них стоит, ноль или более раз. Например, целое число без знака записывается повторением несколько раз цифр, т. е. соответствующий нетерминальный символ можно определить так:

```
<Unsigned> ::= {<Digit>}
```

Это простое определение не совсем верно, т. к. фигурные скобки указывают на повторение ноль или большее число раз, т. е. пустая строка также будет соответствовать нашему определению `<Unsigned>`. Чтобы этого не происходило, исправим наше определение:

```
<Unsigned> ::= <Digit> {<Digit>}
```

Теперь синтаксическое правило, определяемое символом `<Unsigned>`, требует, чтобы выражение состояло из одной или более цифр.

В некоторых случаях после закрывающей фигурной скобки ставят символ "+" в верхнем индексе, чтобы показать, что содержимое скобок должно повторяться не менее одного раза. Например, следующее определение `<Unsigned>` эквивалентно предыдущему:

$$\texttt{<Unsigned> ::= \{<Digit>\}}^{+}$$

Однако это обозначение не является общепризнанным, поэтому мы не будем им пользоваться.

Этим исчерпывается набор правил БНФ. Далее мы будем использовать эти правила для описания различных синтаксических конструкций. При этом мы увидим, что, несмотря на простоту, БНФ позволяет описывать очень сложные конструкции, и это описание просто для понимания.

4.3. Синтаксис вещественного числа

Попытаемся описать синтаксис вещественного числа с помощью БНФ. Сначала опишем этот синтаксис словами: "Перед числом может стоять знак — плюс или минус. Затем идет одна или несколько цифр. Потом может следовать точка, после которой будет еще одна или несколько цифр. Затем может быть указан показатель степени "E" (большое или малое), после которого может стоять знак плюс или минус, а затем должна быть одна или несколько цифр". Указанные правила описывают синтаксис записи вещественных чисел, принятый в Delphi. Согласно им, правильными вещественными числами считаются, например, выражения "10", "0.1", "+4", "–3.2", "8.26e–5" и т. п. Такие выражения, как, например, ".6" и "–.5", этим правилам не удовлетворяют, т. к. перед десятичной точкой должна стоять хотя бы одна цифра. В некоторых языках программирования такая запись допускается, но Delphi требует обязательного наличия целой части.

Теперь переведем эти правила на язык БНФ (листинг 4.1).

Листинг 4.1. Синтаксис вещественного числа

```
<Number> ::= [<Sign>] <Digit> {<Digit>}
  [<Separator> <Digit> {<Digit>}]
  [<Exponent> [<Sign>] <Digit> {<Digit>}]
<Digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<Sign> ::= '+' | '-'
<Separator> ::= '.'
<Exponent> ::= 'E' | 'e'
```

На основе этих правил можно написать функцию `IsNumber`, которая в качестве параметра принимает строку и возвращает `True`, если эта строка удовлетворяет правилам записи числа, и `False`, если не удовлетворяет (листинг 4.2).

Листинг 4.2. Функция для определения соответствия строки синтаксису вещественного числа

```
// Проверка символа на соответствие <Digit>
function IsDigit(Ch: Char): Boolean;
begin
  Result := Ch in ['0'..'9'];
end;
```

```
// Проверка символа на соответствие <Sign>
function IsSign(Ch: Char): Boolean;
begin
  Result := (Ch = '+') or (Ch = '-');
end;

// Проверка символа на соответствие <Separator>
function IsSeparator(Ch: Char): Boolean;
begin
  Result := Ch= '.';
end;

// Проверка символа на соответствие <Exponent>
function IsExponent(Ch: Char): Boolean;
begin
  Result := (Ch = 'E') or (Ch = 'e');
end;

function IsNumber(const S: string): Boolean;
var
  P: Integer; // Номер символа выражения, который сейчас проверяется
begin
  Result := False;
  // Проверка, что выражение содержит хотя бы один символ — пустая строка
  // не является числом
  if Length(S) = 0 then
    Exit;
  // Начинаем проверку с первого символа
  P := 1;
  // Если первый символ — <Sign>, переходим к следующему
  if IsSign(S[P]) then
    Inc(P);
  // Проверяем, что в данной позиции стоит хотя бы одна цифра
  if (P > Length(S)) or not IsDigit(S[P]) then
    Exit;
  // Переходим к следующей позиции, пока не достигнем конца строки
  // или не встретим не цифру
  repeat
    Inc(P);
  until (P > Length(S)) or not IsDigit(S[P]);
  // Если достигли конца строки, выражение корректно — число,
  // не имеющее дробной части и экспоненты
  if P > Length(S) then
```

```
begin
  Result := True;
  Exit;
end;
// Если следующий символ — <Separator>, проверяем, что после него
// стоит хотя бы одна цифра
if IsSeparator(S[P]) then
begin
  Inc(P);
  if (P > Length(S)) or not IsDigit(S[P]) then
    Exit;
  repeat
    Inc(P);
  until (P > Length(S)) or not IsDigit(S[P]);
  // Если достигли конца строки, выражение корректно — число
  // без экспоненты
  if P > Length(S) then
  begin
    Result := True;
    Exit;
  end;
end;
// Если следующий символ — <Exponent>, проверяем, что после него
// стоит все то, что требуется правилами
if IsExponent(S[P]) then
begin
  Inc(P);
  if P > Length(S) then
    Exit;
  if IsSign(S[P]) then
    Inc(P);
  if (P > Length(S)) or not IsDigit(S[P]) then
    Exit;
  repeat
    Inc(P);
  until (P > Length(S)) or not IsDigit(S[P]);
  if P > Length(S) then
  begin
    Result := True;
    Exit;
  end;
end;
```

```
  // Если выполнение дошло до этого места, значит, в выражении остались
  // еще какие-то символы. Так как никакие дополнительные символы
  // синтаксисом не предусмотрены, такое выражение не считается
  // корректным числом.
end;
```

Для каждого нетерминального символа мы ввели отдельную функцию, разбор начинается с символа самого верхнего уровня — `<Number>` — и следует правилам, записанным для этого символа. Такой способ синтаксического анализа называется *левосторонним рекурсивным нисходящим анализом*. Левосторонним потому, что символы в выражении перебираются слева направо, нисходящим — потому, что сначала анализируются символы верхнего уровня, а потом — символы нижнего. Рекурсивность метода на данном примере не видна, т. к. наша грамматика не содержит рекурсивных определений, но мы с этим столкнемся в последующих примерах.

Пример использования функции `IsNumber` содержится на компакт-диске и называется IsNumberSample.

В заключение рассмотрим альтернативный способ записи грамматики вещественного числа — графический (такой способ называется синтаксическим графом, или рельсовой диаграммой). Это направленный граф, узлами которого являются терминальные (круги) и нетерминальные (прямоугольники) символы. Двигаться от одного узла к другому можно только по линиям в направлениях, указанных стрелками. В таком графе достаточно легко разобраться, а по возможностям описания синтаксиса он эквивалентен БНФ. На рис. 4.1 показана запись синтаксиса вещественного числа с помощью рельсовой диаграммы.

В качестве самостоятельного упражнения рекомендуем нарисовать с помощью рельсовой диаграммы грамматику символа "Цифра", используемого на рис. 4.1.

4.4. Простой калькулятор

Теперь у нас уже достаточно знаний, чтобы создать простейший калькулятор, т. е. функцию, которая будет на входе принимать выражение, а на выходе, если это выражение корректно, возвращать результат его вычисления. Для начала ограничимся простым калькулятором, который умеет работать только с числовыми константами и знает только четыре действия арифметики. Изменение порядка вычисления операторов с помощью скобок также оставим на потом.

Таким образом, наш калькулятор будет распознавать и вычислять цепочки чисел, между которыми стоят знаки операции, которые над этими числами

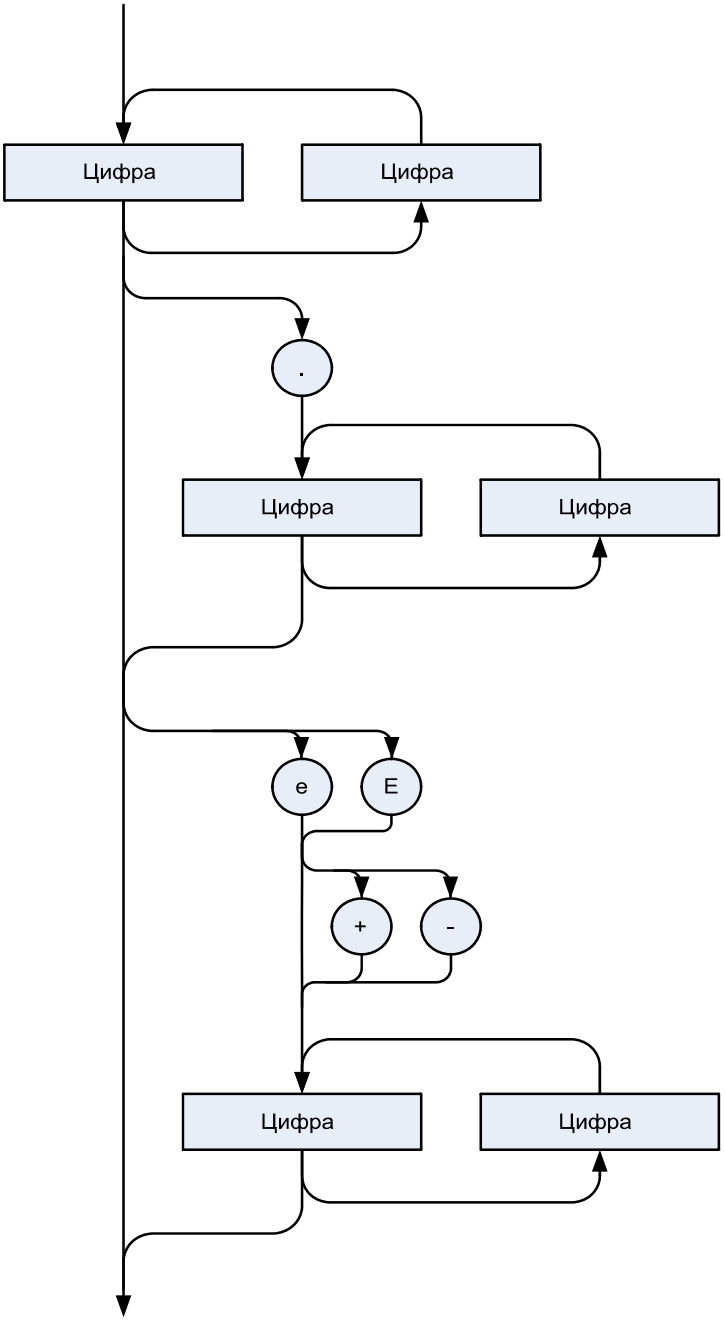


Рис. 4.1. Диаграмма синтаксиса вещественного числа

выполняются. В вырожденном случае выражение может состоять из одного числа и, соответственно, не содержать ни одного знака операции. Опишем эти правила с помощью БНФ и ранее определенного символа <Number>.

```
<Expr> ::= <Number> {<Operation> <Number>}
<Operation> ::= '+' | '-' | '*' | '/'
```

Примечание

В нашей грамматике не предусмотрено, что между оператором и его операндами может находиться пробел, т. е. выражение "2 + 2", в отличие от "2+2", не удовлетворяет данной грамматике. В отсутствие лексического анализатора игнорирование пробелов и прочих разделителей (переводов строки, комментариев) является трудоемкой рутинной операцией, поэтому во всех примерах без лексического анализатора мы будем требовать, чтобы выражения не содержали пробелов.

Для написания калькулятора нам понадобятся две новых функции — IsOperator, которая проверяет, является ли следующий символ оператором, и Expr, которая получает на входе строку, анализирует ее в соответствии с указанными правилами и вычисляет результат. Кроме того, функция IsNumber сама по себе нам тоже больше не нужна — мы создадим на ее основе функцию Number, которая получает на входе строку и номер позиции, начиная с которой в этой строке должно быть расположено число, проверяет, так ли это, и возвращает это число. Кроме того, функция Number должна перемещать указатель на следующий после числа символ строки, чтобы функция Expr, вызвавшая Number, могла узнать, с какого символа продолжать анализ. Если последовательность символов не является корректным числом, функция Number возбуждает исключение ESyntaxError, определенное специально для указания на ошибку в записи выражения.

Сама по себе задача преобразования строки в вещественное число достаточно сложна, и чтобы не отвлекаться на ее решение, мы воспользуемся функцией StrToFloat из модуля SysUtils. Когда функция Number выделит из строки последовательность символов, являющуюся числом, эта последовательность передается функции StrToFloat, и преобразованием занимается она. Здесь следует учесть два момента. Во-первых, в нашей грамматике разделителем целой и дробной части является точка, а StrToFloat использует системные настройки, т. е. разделителем может быть и запятая. Чтобы обойти эту проблему, слегка изменим синтаксис и будем сравнивать аргумент функции IsSeparator не с символом ".", а с DecimalSeparator (таким образом, наш калькулятор тоже станет чувствителен к системным настройкам). Во-вторых, не всякое выражение, соответствующее нашей грамматике, будет допустимым числом с точки зрения StrToFloat, т. к. эта функция учитывает диапазон типа Extended. Например, синтаксически верное выражение "2e5000" даст

исключение EConvertError, т. к. данное число выходит за пределы этого диапазона. Но пока мы остаемся в рамках типа Extended, мы вынуждены мириться с этим.

Новые функции приведены в листинге 4.3.

Листинг 4.3. Реализация простейшего калькулятора

```
// Выделение из строки подстроки, соответствующей
// определению <Number>, и вычисление этого числа
// S — строка, из которой выделяется подстрока
// P — номер позиции в строке, с которой должно
// начинаться число. После завершения работы функции
// этот параметр содержит номер первого после числа
// символа
function Number(const S: string; var P: Integer): Extended;
var
  InitPos: Integer;
begin
  // InitPos нам понадобится для выделения подстроки,
  // которая будет передана в StrToFloat
  InitPos := P;
  if (P <= Length(S)) and IsSign(S[P]) then
    Inc(P);
  if (P > Length(S)) or not IsDigit(S[P]) then
    raise ESyntaxError.Create(
      'Ожидается цифра в позиции ' + IntToStr(P));
  repeat
    Inc(P);
  until (P > Length(S)) or not IsDigit(S[P]);
  if (P <= Length(S)) and IsSeparator(S[P]) then
  begin
    Inc(P);
    if (P > Length(S)) or not IsDigit(S[P]) then
      raise ESyntaxError.Create(
        'Ожидается цифра в позиции ' + IntToStr(P));
    repeat
      Inc(P);
    until (P > Length(S)) or not IsDigit(S[P]);
  end;
  if (P <= Length(S)) and IsExponent(S[P]) then
  begin
    Inc(P);
```

```
    if P > Length(S) then
      raise ESyntaxError.Create('Неожиданный конец строки');
    if IsSign(S[P]) then
      Inc(P);
    if (P > Length(S)) or not IsDigit(S[P]) then
      raise ESyntaxError.Create(
        'Ожидается цифра в позиции ' + IntToStr(P));
    repeat
      Inc(P);
    until (P > Length(S)) or not IsDigit(S[P]);
  end;
  Result := StrToFloat(Copy(S, InitPos, P – InitPos));
end;

// Проверка символа на соответствие <Operator>
function IsOperator(Ch: Char): Boolean;
begin
  Result := Ch in ['+', '-', '*', '/'];
end;

// Проверка строки на соответствие <Expr>
// и вычисление выражения
function Expr(const S: string): Extended;
var
  P: Integer;
  OpSymb: Char;
begin
  P := 1;
  Result := Number(S, P);
  while (P <= Length(S)) and IsOperator(S[P]) do
  begin
    OpSymb := S[P];
    Inc(P);
    case OpSymb of
      '+': Result := Result + Number(S, P);
      '-': Result := Result – Number(S, P);
      '*': Result := Result * Number(S, P);
      '/': Result := Result / Number(S, P);
    end;
  end;
  if P <= Length(S) then
   raise ESyntaxError.Create(
     'Некорректный символ в позиции ' + IntToStr(P));
end;
```

Код приведен практически без комментариев, т. к. он очень простой, и все моменты, заслуживающие упоминания, мы уже разобрали в тексте. На прилагаемом компакт-диске находится программа SimpleCalcSample, которая демонстрирует работу нашего калькулятора. Калькулятор выполняет действия над числами слева направо, без учета приоритета операций, т. е. вычисление выражения "2+2*2" даст 8.

Грамматика выражения является простой для разбора, т. к. разбор выражения идет слева направо, и для соотнесения очередной части строки с тем или иным нетерминальным символом на любом этапе анализа достаточно знать только следующий символ. Такие грамматики называются *LR(1)-грамматиками* (в более общем случае требуется не один символ, а одна лексема). Класс этих грамматик был исследован Кнутом.

Грамматика Паскаля не относится к классу LR(1)-грамматик из-за уже упоминавшейся проблемы отнесения `else` к тому или иному `if`. Чтобы решить эту проблему, приходится вводить два нетерминальных символа — завершенной формы оператора `if` (с `else`) и незавершенной (без `else`). Таким образом, встретив в тексте программы лексему `if`, синтаксический анализатор не может сразу отнести ее к одному из этих символов, пока не продвинется вперед и не натолкнется на наличие или отсутствие `else`. А поскольку оператор `if` может быть оператором в циклах `for`, `while` или в операторе `with`, для них также приходится вводить завершенную и незавершенную форму. Именно из-за этой проблемы Вирт (разработчик Паскаля) отказался от идеи составного оператора и модифицировал синтаксис в своем новом языке Оберон таким образом, чтобы проблема `else` не возникала.

Другое достоинство нашей простой грамматики — ее однозначность. Любая синтаксически верная строка не допускает неоднозначной трактовки. Неоднозначность могла бы возникнуть, например, если бы какая-то операция обозначалась символом "." (точка). Тогда было бы непонятно, должно ли выражение "1.5" трактоваться как число "одна целая пять десятых" или как выполнение операции над числами 1 и 5. Этот пример выглядит несколько надуманным, но неоднозначные грамматики, тем не менее, иногда встречаются на практике. Например, если запятая служит для отделения дробной части числа от целой и для разделения значений в списке параметров функций, то выражение `f(1,5)` может, с одной стороны, трактоваться как вызов функции `f` с одним аргументом 1.5, а с другой — как вызов ее с двумя аргументами 1 и 5. Правила решения неоднозначных ситуаций не описываются в виде БНФ, их приходится объяснять "на словах", что затрудняет разбор соответствующих выражений. Другой пример неоднозначной грамматики — грамматика языков C/C++. В них оператор инкремента, записывающийся как

"++", имеет две формы записи — префиксную (перед увеличиваемой переменной) и постфиксную (после переменной). Кроме того, этот оператор возвращает значение, поэтому его можно использовать в выражениях. Синтаксически допустимо, например, выражение `a+++b`, но грамматика не дает ответа, следует ли это трактовать как `(a++)+b` или как `a+(++b)`. Кроме того, т. к. существует операция "унарный плюс", возможно и третье толкование — `a+(+(+b))`.

4.5. Учет приоритета операторов

Следующим нашим шагом станет модификация калькулятора таким образом, чтобы он учитывал приоритет операций, т. е. чтобы умножение и деление выполнялись раньше сложения и умножения.

Для примера рассмотрим выражение "2*4+3*8/6". Наш синтаксис должен как-то отразить то, что аргументами операции сложения в данном случае являются не числа 4 и 3, а "2*4" и "3*8/6". В общем случае это означает, что выражение — это последовательность из одного или нескольких слагаемых, между которыми стоят знаки "+" или "–". А слагаемые — это, в свою очередь, последовательности из одного или нескольких чисел, разделенных знаками "*" и "/". А теперь запишем то же самое на языке БНФ (листинг 4.4).

Листинг 4.4. Грамматика выражения с учетом приоритета операций

```
<Expr> ::= <Term> {<Operator1> <Term>}
<Term> ::= <Number> {<Operator2> <Number>}
<Operator1> ::= '+' | '-'
<Operator2> ::= '*' | '/'
```

Примечание

Определение символа `<Operator1>` совпадает с определением введенного ранее символа `<Sign>`. Но использовать `<Sign>` в определении `<Expr>` было бы неправильно, т. к., в принципе, в выражении могут существовать и другие операции, имеющие тот же приоритет (как, например, операции арифметического или и арифметического исключающего или в Delphi), и тогда определение `<Operator1>` будет расширено. Но это не должно затронуть определение символа `<Number>`, в которое входит `<Sign>`.

Чтобы приспособить калькулятор к новым правилам, нужно заменить функцию `Operator` на `Operator1` и `Operator2`, добавить функцию `Term` (слагаемое) и внести изменения в `Expr`. Функция `Number` остается без изменения. Обновленная часть калькулятора выглядит следующим образом (листинг 4.5).

Листинг 4.5. Реализация калькулятора с учетом приоритета операций

```
// Проверка символа на соответствие <Operator1>
function IsOperator1(Ch: Char): Boolean;
begin
  Result := Ch in ['+', '-'];
end;

// Проверка символа на соответствие <Operator2>
function IsOperator2(Ch: Char): Boolean;
begin
  Result := Ch in ['*', '/'];
end;

// Выделение подстроки, соответствующей <Term>,
// и ее вычисление
function Term(const S: string; var P: Integer): Extended;
var
  OpSymb: Char;
begin
  Result := Number(S,P);
  while (P <= Length(S)) and IsOperator2(S[P]) do
  begin
    OpSymb := S[P];
    Inc(P);
    case OpSymb of
      '*': Result := Result * Number(S, P);
      '/': Result := Result / Number(S, P);
    end;
  end;
end;

// Проверка строки на соответствие <Expr>
// и вычисление выражения
function Expr(const S: string): Extended;
var
  P: Integer;
  OpSymb: Char;
begin
  P := 1;
  Result := Term(S, P);
  while (P <= Length(S)) and IsOperator1(S[P]) do
  begin
    OpSymb := S[P];
    Inc(P);
```

```
    case OpSymb of
      '+': Result := Result + Term(S, P);
      '-': Result := Result – Term(S, P);
    end;
  end;
  if P <= Length(S) then
    raise ESyntaxError.Create(
      'Некорректный символ в позиции ' + IntToStr(P));
end;
```

Если вы разобрались с предыдущими примерами, приведенный здесь код будет вам понятен. Некоторых комментариев требует только функция `Term`. Она выделяет, начиная с заданного символа, ту часть строки, которая соответствует определению `<Term>`. Вызвавшая ее функция `Expr` должна продолжить разбор выражения со следующего за этой подстрокой символа, поэтому функция `Term`, как и `Number`, имеет параметр-переменную `P`, которая на входе содержит номер первого символа слагаемого, а на выходе — номер первого после этого слагаемого символа.

Пример калькулятора, учитывающего приоритет операций, находится на компакт-диске под именем PrecedenceCalcSample. Поэкспериментировав с ним, легко убедиться, что теперь вычисление "2+2*2" дает правильное значение 6.

В заключение заметим, что язык, определяемый такой грамматикой, полностью совпадает с языком, определяемым грамматикой из предыдущего примера, т. е. любое выражение, принадлежащее первому языку, принадлежит и второму, и наоборот. Усложнение синтаксиса, которое мы здесь ввели, требуется именно для отражения семантики выражений, а не для расширения самого языка.

4.6. Выражения со скобками

Порядок выполнения операций в выражении может меняться с помощью скобок. Внутри них должно находиться выражение, которое, будучи выделенным в отдельную строку, само по себе отвечает требованиям синтаксиса к выражению в целом.

Выражение, заключенное в скобки, допустимо везде, где допускается появление отдельного числа (из этого, в частности, следует, что допускаются вложенные скобки). Таким образом, мы должны расширить нашу грамматику так, чтобы аргументом операций сложения и умножения могли служить не только числа, но и выражения, заключенные в скобки. Это автоматически позволит использовать такие выражения и в качестве слагаемых, потому что

слагаемое — это последовательность из одного или нескольких множителей, разделенных знаками умножения и деления.

На языке БНФ все сказанное иллюстрирует листинг 4.6.

Листинг 4.6. Грамматика выражения со скобками (первое приближение)

```
<Expr> ::= <Term> {<Operation1> <Term>}
<Term> ::= <Factor> {<Operation2> <Factor>}
<Factor> ::= <Number> | '('<Expr>')'
```

В этих определениях появилась рекурсия, т. к. в определении `<Expr>` используется (через `<Term>`) символ `<Factor>`, а в определении `<Factor>` — `<Term>`. Соответственно, подобная грамматика будет реализовываться рекурсивными функциями.

Наша грамматика не учитывает, что перед скобками может стоять знак унарной операции "+" или "–", хотя общепринятые правила записи выражений вполне допускают выражения типа `3*-(2+4)`. Поэтому, прежде чем приступить к созданию нового калькулятора, введем правила, допускающие такой синтаксис. Можно было бы модифицировать определение `<Factor>` таким образом:

```
<Factor> ::= <Number> | [Sign] '(' <Expr> ')'
```

Однако такой подход страдает отсутствием общности. В дальнейшем мы усложним наш синтаксис, введя другие типы множителей (функции, переменные). Перед каждым из них, в принципе, может стоять знак унарной операции, поэтому логичнее определить синтаксис таким образом, чтобы унарная операция допускалась вообще перед любым множителем. В этом случае можно будет слегка упростить определение `<Number>`, т. к. знак "+" или "–" в начале числа можно будет трактовать не как часть числа, а как унарный оператор, стоящий перед множителем, представленным в виде числовой константы.

С учетом этого новая грамматика запишется следующим образом (листинг 4.7).

Листинг 4.7. Окончательный вариант грамматики выражения со скобками

```
<Expr> ::= <Term> {<Operation1> <Term>}
<Term> ::= <Factor> {<Operation2> <Factor>}
<Factor> ::= <UnaryOp> <Factor> | <Number> | '('<Expr>')'
<Number> ::= <Digit> {<Digit>} [<Separator> <Digit> {<Digit>}]
  [<Exponent> [<Sign>] <Digit> {<Digit>}]
<UnaryOp> ::= '+' | '-'
```

Здесь опущены определения некоторых вспомогательных символов, которые не изменились.

Мы видим, что грамматика стала "более рекурсивной", т. е. в определении символа `<Factor>` используется он сам. Соответственно, функция `Factor` будет вызывать саму себя.

Символ `<UnaryOp>`, определение которого совпадает с определениями `<Operator1>` и `<Sign>`, мы делаем независимым нетерминальным символом по тем же причинам, что и ранее: в принципе, синтаксис может допускать унарные операции (как, например, `not` в Delphi), которые не являются ни знаками, ни допустимыми бинарными операциями.

Побочным эффектом нашей грамматики стало то, что, например, `-5` воспринимается как множитель, а потому перед ним допустимо поставить унарный оператор, т. е. выражение `--5` также является корректным множителем и трактуется как `-(-5)`. А перед `--5`, в свою очередь, можно поставить еще один унарный оператор. И так — до бесконечности. Это может показаться не совсем правильным, но, тем не менее, такая грамматика широко распространена. Легко, например, убедиться, что компилятор Delphi считает допустимым выражение `2+-+-2`, трактуя его как `2+(-(+(-2)))`.

Листинг 4.8 иллюстрирует реализацию данной грамматики.

Листинг 4.8. Реализация калькулятора со скобками

```
// Так как грамматика рекурсивна, функция Expr
// должна быть объявлена заранее
function Expr(const S: string; var P: Integer): Extended;
  forward;

// Выделение подстроки, соответствующей <Factor>,
// и ее вычисление
function Factor(const S: string; var P: Integer): Extended;
begin
  if P > Length(S) then
    raise ESyntaxError.Create('Неожиданный конец строки');
  // По первому символу подстроки определяем,
  // какой это множитель
  case S[P] of
    '+': // унарный "+"
      begin
        Inc(P);
        Result := Factor(S, P);
      end;
```

```
    '-': // унарный "-"
      begin
        Inc(P);
        Result := -Factor(S, P);
      end;
    '(': // выражение в скобках
      begin
        Inc(P);
        Result := Expr(S, P);
        // Проверяем, что скобка закрыта
        if (P > Length(S)) or (S[P] <> ')') then
          raise ESyntaxError.Create(
            'Ожидается ")" в позиции ' + IntToStr(P));
        Inc(P);
      end;
    '0'..'9': // Числовая константа
      Result := Number(S, P);
    else
      raise ESyntaxError.Create(
        'Некорректный символ в позиции ' + IntToStr(P));
  end;
end;

// Выделение подстроки, соответствующей <Term>,
// и ее вычисление
function Term(const S: string; var P: Integer): Extended;
var
  OpSymb: Char;
begin
  Result := Factor(S, P);
  while (P <= Length(S)) and IsOperator2(S[P]) do
  begin
    OpSymb := S[P];
    Inc(P);
    case OpSymb of
      '*': Result := Result * Factor(S, P);
      '/': Result := Result / Factor(S, P);
    end;
  end;
end;

// Выделение подстроки, соответствующей <Expr>,
// и ее вычисление
function Expr(const S: string; var P: Integer): Extended;
```

```
var
  OpSymb: Char;
begin
  Result := Term(S, P);
  while (P <= Length(S)) and IsOperator1(S[P]) do
  begin
    OpSymb := S[P];
    Inc(P);
    case OpSymb of
      '+': Result := Result + Term(S, P);
      '-': Result := Result – Term(S, P);
    end;
  end;
end;

// Вычисление выражения
function Calculate(const S: string): Extended;
var
  P: Integer;
begin
  P := 1;
  Result := Expr(S, P);
  if P <= Length(S) then
    raise ESyntaxError.Create(
      'Некорректный символ в позиции ' + IntToStr(P));
end;
```

По сравнению с предыдущим примером функция `Term` осталась такой же с точностью до замены вызовов `Number` на новую функцию `Factor`. Функция `Factor` выделяет подстроку, отвечающую отдельному множителю. Множители, напомним, могут быть трех типов: число, выражение в скобках, множитель с унарным оператором. Различить их можно по первому символу подстроки. Функция `Factor` распознает тип множителя и вызывает соответствующую функцию для его вычисления.

Функция `Expr` теперь может применяться не только к выражению в целом, но и к отдельной подстроке. Поэтому она, как и все остальные функции, теперь имеет параметр-переменную `P`, через который передается начало и конец этой подстроки. Из функции убрана проверка того, что в результате ее использования строка проанализирована полностью, т. к. теперь допустим анализ части строки.

Функция `Expr` в своем новом виде стала не очень удобной для конечного пользователя, поэтому была описана еще одна функция — `Calculate`. Это

вспомогательная функция, которая избавляет пользователя от вникания в детали "внутренней кухни" калькулятора, т. е. использования переменной `P` и проверки того, что строка проанализирована до конца.

Пример калькулятора со скобками записан на компакт-диске под названием BracketsCalcSample. Анализируя его код, можно заметить, что по сравнению с предыдущим примером незначительно изменена функция `Number` — из нее в соответствии с новой грамматикой убрана проверка знака в начале выражения.

4.7. Полноценный калькулятор

Последняя версия нашего калькулятора может считать сложные выражения, но чтобы он имел практическую ценность, этого мало. В этом разделе мы научим наш калькулятор использовать функции и переменные. Также будет введена операция возведения в степень, обозначающаяся значком "^".

Имена переменных и функций — это идентификаторы. Идентификатор определяется по общепринятым правилам: он должен начинаться с буквы латинского алфавита или символа "_", следующие символы должны быть буквами, цифрами или "_". Таким образом, грамматика идентификатора выглядит так:

```
<Letter> ::= 'A' | ... | 'Z' | 'a' | ... | 'z' | '_'
<Identifier> ::= <Letter> {<Letter> | <Digit>}
```

Примечание

Следствием этой грамматики является то, что отдельно взятый символ "_" считается корректным идентификатором. И хотя это может на первый взгляд показаться абсурдным, тем не менее, именно таковы общепринятые правила. Легко убедиться, что, например, Delphi допускает объявление переменных с именами "_", "__" и т. п.

В нашей грамматике переменной будет называться отдельно стоящий идентификатор, функцией — идентификатор, после которого в скобках записан аргумент, в качестве которого может выступать любое допустимое выражение (для простоты мы будем рассматривать только функции с одним аргументом, т. к. обобщение грамматики на большее число аргументов очевидно). Другими словами, определение будет выглядеть так:

```
<Variable> ::= <Identifier>
<Function> ::= <Identifier> '(' <Expr> ')'
```

Из приведенных определений видно, что грамматика, основанная на них, не относится к классу LR(1)-грамматик, т. к. обнаружив в выражении идентификатор, анализатор не может сразу решить, является ли этот идентификатор

переменной или именем функции, это выяснится только при проверке следующего символа — скобка это или нет. Тем не менее реализация такой грамматики достаточно проста, и это не будет доставлять нам существенных неудобств.

Переменные и функции, так же, как и выражения, заключенные в скобки, выступают в роли множителей. Соответственно, их появление в грамматике учитывается расширением смысла символа <Factor>.

```
<Factor> ::= <UnaryOp> <Factor> |
  <Variable> |
  <Function> |
  <Number> |
  '(' <Expr> ')'
```

Теперь рассмотрим свойства оператора возведения в степень. Во-первых, его приоритет выше, чем у операций сложения и деления, т. е. выражение a*b^c трактуется как a*(b^c), а a^b*c — как (a^b)*c. Во-вторых, он правоассоциативен, т. е. a^b^c означает a^(b^c), а не (a^b)^c. В-третьих, его приоритет выше, чем приоритет унарных операций, т. е. -a^b означает -(a^b), а не (-a)^b. Тем не менее, a^-b означает a^(-b).

Таким образом, мы видим, что показателем степени может быть любой отдельно взятый множитель, а основанием — число, переменная, функция или выражение в скобках, т. е. любой множитель, за исключением начинающегося с унарного оператора. Запишем это в виде БНФ.

```
<Factor> ::= <UnaryOp> <Factor> | <Base> ['^' <Factor>]
<Base> ::= <Variable> | <Function> | <Number> | '(' <Expr> ')'
```

Правая ассоциативность также заложена в этих определениях. Рассмотрим, как будет разбираться выражение a^b^c. Сначала функция Factor (через вызов функции Base) выделит и вычислит множитель a, а потом вызовет саму себя для вычисления остатка b^c. Таким образом, a будет возведено в степень b^c, как это и требуют правила правой ассоциативности.

Вообще, вопросы правой и левой ассоциативности операторов, которые мы здесь опустили, оказывают влияние на то, как определяется грамматика языка. Более подробно об этом написано в [5].

Так как определения символов <Expr> и <Term> в нашей новой грамматике не изменились, не изменятся и соответствующие функции. Для реализации нового синтаксиса нам потребуется изменить функцию Factor и ввести новые функции Base, Identifier и Func (примем такое сокращение, т. к. function в Delphi является зарезервированным словом). Идентификаторы будем полагать нечувствительными к регистру символов.

Для простоты обойдемся тремя функциями: sin, cos и ln. Увеличение количества функций, допустимых в выражении, — простая техническая задача, не представляющая особого интереса.

Если у нас появились переменные, то мы должны как-то хранить их значения, чтобы при вычислении выражения использовать их. В нашем примере мы будем хранить их в объекте типа TStrings, получая доступ через свойство Values. С точки зрения производительности, этот способ — один из самых худших, поэтому при создании реального калькулятора лучше придумать что-нибудь другое. Мы здесь выбрали этот способ исключительно из соображений наглядности. Получившийся в итоге код показан в листинге 4.9.

Листинг 4.9. Реализация полноценного калькулятора

```
// Вычисление функции, имя которой передается через FuncName
function Func(const FuncName, S: string; var P: Integer): Extended;
var
  Arg: Extended;
begin
  // Вычисляем аргумент
  Arg := Expr(S, P);
  // Сравниваем имя функции с одним из допустимых
  if AnsiCompareText(FuncName, 'sin') = 0 then
    Result := Sin(Arg)
  else if AnsiCompareText(FuncName, 'cos') = 0 then
    Result := Cos(Arg)
  else if AnsiCompareText(FuncName, 'ln') = 0 then
    Result := Ln(Arg)
  else
    raise ESyntaxError.Create('Неизвестная функция ' + FuncName);
end;

// Выделение из строки идентификатора и определение,
// является ли он переменной или функцией
function Identifier(const S: string; var P: Integer): Extended;
var
  InitP: Integer;
  IDStr, VarValue: string;
begin
  // Запоминаем начало идентификатора
  InitP := P;
  // Первый символ был проверен еще в функции Base.
  // Сразу переходим к следующему
```

```
  Inc(P);
  while (P <= Length(S)) and
    (S[P] in ['A'..'Z', 'a'..'z', '_', '0'..'9']) do
    Inc(P);
  // Выделяем идентификатор из строки
  IDStr := Copy(S, InitP, P – InitP);
  // Если за ним стоит открывающая скобка — это функция
  if (P <= Length(S)) and (S[P] = '(') then
  begin
    Inc(P);
    Result := Func(IDStr, S, P);
    // Проверяем, что скобка закрыта
    if (P > Length(S)) or (S[P] <> ')') then
      raise ESyntaxError.Create(
        'Ожидается ")" в позиции ' + IntToStr(P));
    Inc(P);
  end
  // если скобки нет — переменная
  else
  begin
    VarValue := Form1.ListBoxVars.Items.Values[IDStr];
    if VarValue = '' then
      raise ESyntaxError.Create(
        'Необъявленная переменная ' + IDStr +
        ' в позиции ' + IntToStr(P))
    else
      Result := StrToFloat(VarValue);
  end;
end;

// Выделение подстроки, соответствующей <Base>,
// и ее вычисление
function Base(const S: string; var P: Integer): Extended;
begin
  if P > Length(S) then
    raise ESyntaxError.Create('Неожиданный конец строки');
  // По первому символу подстроки определяем,
  // какое это основание
  case S[P] of
    '(': // выражение в скобках
      begin
        Inc(P);
        Result := Expr(S, P);
```

```
        // Проверяем, что скобка закрыта
        if (P > Length(S)) or (S[P] <> ')') then
          raise ESyntaxError.Create(
            'Ожидается ")" в позиции ' + IntToStr(P));
        Inc(P);
      end;
    '0'..'9': // Числовая константа
      Result := Number(S, P);
    'A'..'Z', 'a'..'z', '_': // Идентификатор (переменная или функция)
      Result := Identifier(S, P);
    else
      raise ESyntaxError.Create(
        'Некорректный символ в позиции ' + IntToStr(P));
  end;
end;

// Выделение подстроки, соответствующей <Factor>,
// и ее вычисление
function Factor(const S: string; var P: Integer): Extended;
begin
  if P > Length(S) then
    raise ESyntaxError.Create('Неожиданный конец строки');
  // По первому символу подстроки определяем,
  // какой это множитель
  case S[P] of
    '+': // унарный "+"
      begin
        Inc(P);
        Result := Factor(S, P);
      end;
    '-': // унарный "-"
      begin
        Inc(P);
        Result := -Factor(S, P);
      end;
    else
      begin
        Result := Base(S, P);
        if (P <= Length(S)) and (S[P] = '^') then
        begin
          Inc(P);
          Result := Power(Result, Factor(S, P));
        end;
```

```
      end;
  end;
end;
```

Пример калькулятора называется FullCalcSample. Его интерфейс (рис. 4.2) содержит новые элементы, с помощью которых пользователь может задавать значения переменных. В левой нижней части окна находится список переменных с их значениями (при запуске программы этот список пустой). Правее расположены поля ввода **Имя переменной** и **Значение переменной**, а также кнопка **Установить**. В первое поле следует ввести имя переменной, во второе — ее значение. При нажатии на кнопку **Установить** переменная будет внесена в список, а если переменная с таким именем уже есть в списке, то ее значение будет обновлено. Все переменные, которые есть в списке, могут использоваться в выражении. Если требуемая переменная в списке не найдена, попытка вычислить выражение приводит к ошибке.

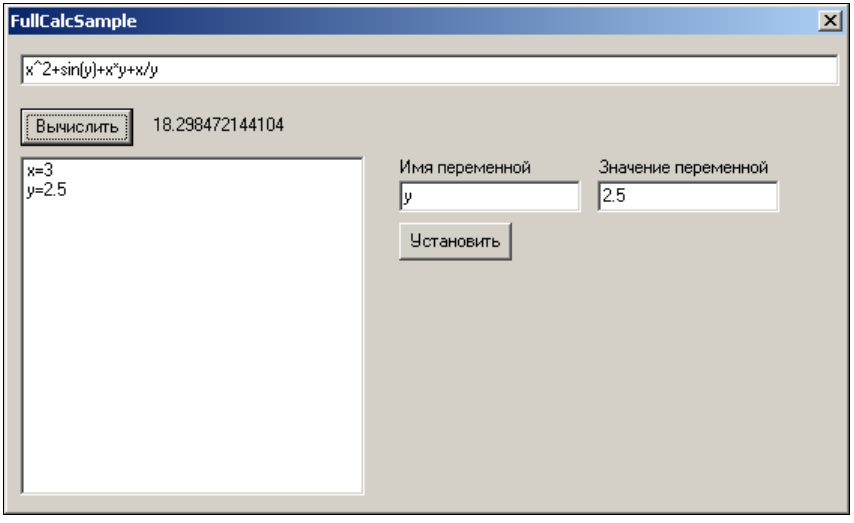


Рис. 4.2. Главное окно программы FullCalcSample

Заметим, что символ `<Factor>` можно было бы определить несколько иначе:

```
<Factor> ::= [<UnaryOp>] <Base> ['^' <Factor>]
```

В нашем случае, когда есть только два унарных оператора и применение сразу двух (разных или одинаковых) практически бессмысленно, такой синтаксис реализовать было бы проще (пример реализации такого синтаксиса дан в программе FullCalcSample в виде комментария). При этом исчезла бы возможность ставить несколько знаков унарных операций подряд. В общем случае такой подход неверен, т. к. при большем количестве унарных операций

это может пригодиться, да и выглядит естественно. Поэтому в качестве основного был выбран несколько более сложный, но и более функциональный вариант.

4.8. Калькулятор с лексическим анализатором

Прежде чем двигаться дальше, рассмотрим недостатки последней версии нашего калькулятора. Во-первых, бросается в глаза некоторое дублирование функций. Действительно, с одной стороны, выделением числа из подстроки занимается функция `Number`, но в функции `Base` также содержится проверка первого символа числа. Функция `Identifier` тоже частично дублируется функцией `Base`.

Второй недостаток — нельзя вставлять разделители, облегчающие чтение выражения. Например, строка "2 + 2" не является допустимым выражением — следует писать "2+2" (без пробелов). Если же попытаться учесть возможность вставки пробелов, придется в разные функции добавлять много однотипного рутинного кода, который существенно усложнит восприятие программы.

Третий недостаток — сложность введения новых операторов, которые обозначаются не одним символом, а несколькими, например: `>=`, `and`, `div`. Если посмотреть функции `Expr` и `Term`, которые придется в этом случае модифицировать, видно, что переделка будет достаточно сложной.

Решить все эти проблемы позволяет лексический анализатор, который выделяет из строки все лексемы, пропуская пробелы и иные разделители, и определяет тип каждой лексемы, не заботясь о том, насколько уместно ее появление в данной позиции выражения. А после лексического анализа начинает работать анализатор синтаксический, который будет иметь дело не с отдельными символами строки, а с целыми лексемами.

В качестве примера рассмотрим реализацию следующей грамматики (листинг 4.10).

Листинг 4.10. Грамматика калькулятора с лексическим анализатором

```
<Expr> ::= <MathExpr> [<Comparison> <MathExpr>]
<Comparison> ::= '=' | '>' | '<' | '>=' | '<=' | '<>'
<MathExpr> ::= <Term> {<Operator1> <Term>}
<Operator1> ::= '+' | '-' | 'or' | 'xor'
<Term> ::= <Factor> {<Operator2> <Factor>}
<Operator2> ::= '*' | '/' | 'div' | 'mod' | 'and'
```

```
<Factor> ::= <UnaryOp> <Factor> | <Base> ['^' <Factor>]
<UnaryOp> ::= '+' | '-' | 'not'
<Base> ::= <Variable> | <Function>| <Number> | '(' <MathExpr> ')'
<Function> ::= <FuncName> '(' <MathExpr> ')'
<FuncName> ::= 'sin' | 'cos' | 'ln'
<Variable> ::= <Letter> {<Letter> | <Digit>}
<Letter> ::= 'A' | ... | 'Z' | 'a' | ... | 'z' | '_'
<Digit> ::= '0' | ... | '9'
<Number> ::= <Digit> {<Digit>} [<DecimalSeparator> <Digit> {<Digit>}]
   [('E' | 'e') ['+' | '-'] <Digit> {<Digit>}]
```

Примечание

Здесь используется нетерминальный символ <DecimalSeparator>, который мы не определили. Он полагается равным точке или запятой в зависимости от системных настроек.

Эта грамматика на первый взгляд может показаться существенно более сложной, чем все, что мы реализовывали ранее, но это не так: просто здесь приведены определения всех (за исключением <DecimalSeparator>) нетерминальных символов. Определение символа <Number> несколько изменено, но это касается только формы его представления — синтаксис числа остался без изменения. То, что раньше обозначалось как <Expr>, теперь называется <MathExpr>, а выражение <Expr> состоит из одного <MathExpr>, с которым, возможно, сравнивается другое <MathExpr>. Семантика <Expr> такова: если в выражении присутствует только обязательная часть, результатом будет число, которое получилось при вычислении <MathExpr>. Если же имеется необязательное сравнение с другим <MathExpr>, то результатом будет "True" или "False" в зависимости от результатов сравнения.

В новой грамматике также расширен набор операторов. Операторы or, xor, and и not здесь арифметические, т. е. применяются к числовым, а не к логическим выражениям. Все операторы, которые применимы только к целым числам (т. е. вышеперечисленные, а также div и mod), игнорируют дробную часть своих аргументов.

Лексический анализатор должен выделять из строки следующие лексемы:

1. Все знаки операций, которые используются в определении символов <Comparison>, <Operator1>, <Operator2>, <UnaryOp>, а также символ "^".
2. Открывающую и закрывающую скобки.
3. Имена функций.
4. Идентификаторы (т. е. переменные).
5. Числовые константы.

Напомним, что лексический анализатор не должен определять допустимость появления лексемы в данном месте строки. Он просто сканирует строку, выделяет из нее последовательности символов, распознаваемые как отдельные лексемы, и сохраняет информацию о них в специальном списке, которым потом пользуется синтаксический анализатор. Так, например, встретив цифру, лексический анализатор выделяет числовую константу. Встретив букву, он выделяет последовательность буквенно-цифровых символов. Затем сравнивает эту последовательность с одним из зарезервированных слов (`and`, `div` и т. п.) и распознает лексему соответственно как идентификатор (переменную) или как зарезервированное слово. При этом выяснение, объявлена ли такая переменная, также не входит в обязанности лексического анализатора — это потом сделает синтаксический анализатор.

Из нашей грамматики следует, что имена функций являются зарезервированными словами, т. е. объявить переменные с именами `sin`, `cos` и `ln`, в отличие от предыдущего примера, нельзя. Это само по себе не упрощает и не усложняет задачу, а сделано только в качестве демонстрации возможной альтернативы (просто если именами служат зарезервированные слова, то их распознает лексический анализатор, а если идентификаторы, то синтаксический).

Отдельные лексемы выделяются по следующему алгоритму: сначала, начиная с текущей позиции, пропускаются все разделители — пробелы и символы перевода строки. Затем по первому символу определяется лексема — знак, слово (которое потом может оказаться зарезервированным словом или идентификатором) или число. Дальше лексический анализатор выбирает из строки все символы до тех пор, пока они удовлетворяют правилам записи соответствующей лексемы. Следующая лексема ищется с позиции, идущей непосредственно за предыдущей лексемой.

В зависимости от типа лексем разделители между ними могут быть обязательными или необязательными. Например, в выражении "2+3" разделители между лексемами "2", "+" и "5" не нужны, потому что они могут быть отделены друг от друга и без этого. А в выражении `6 div 3` разделитель между "div" и "3" необходим, потому что в противном случае эти лексемы будут восприняты как идентификатор `div3`. А вот разделитель между "6" и "div" не обязателен, т. к. `6div` не является допустимым идентификатором, и анализатор сможет отделить эти лексемы друг от друга и без разделителя. Вообще, если подстрока, получающаяся в результате слияния двух лексем, может быть целиком интерпретирована как какая-либо другая лексема, разделитель между ними необходим, в противном случае — необязателен. Разделитель внутри отдельной лексемы не допускается (т. е. подстрока "a 1" будет интерпретироваться как последовательность лексем "a" и "1", а не как лексема "a1").

Чтобы продемонстрировать возможности лексического анализатора, добавим поддержку комментариев. Комментарий — это последовательность символов, начинающаяся с "{" и заканчивающаяся "}", которая может содержать внутри себя любые символы, кроме "}". Комментарий считается разделителем, он допустим в любом месте, где возможно появление других разделителей, т. е. в начале и в конце строки и между лексемами.

Пример калькулятора с лексическим анализатором также находится на компакт-диске и называется LexicalSample.

Лексический анализатор на входе получает строку, на выходе он должен дать список структур, каждая из которых описывает одну лексему. В нашем примере эти структуры выглядят следующим образом (листинг 4.11).

Листинг 4.11. Тип `TLexeme` для хранения информации об одной лексеме

```
TLexemeType = (ltEqual, ltLess, ltGreater, ltLessOrEqual,
               ltGreaterOrEqual, ltNotEqual,
               ltPlus, ltMinus, ltOr, ltXor,
               ltAsterisk, ltSlash, ltDiv, ltMod, ltAnd,
               ltNot,
               ltCap,
               ltLeftBracket, ltRightBracket,
               ltSin, ltCos, ltLn,
               ltIdentifier,
               ltNumber,
               ltEnd);

TLexeme = record
  LexemeType: TLexemeType;
  Pos: Integer;
  Lexeme: string;
end;
```

`LexemeType` — поле, содержащее информацию о том, что это за лексема. Тип `TLexemeType` — это специально определенный перечислимый тип, каждое из значений которого соответствует одному из возможных типов лексемы. Поле `Pos` хранит номер позиции в строке, начиная с которой идет данная лексема. Это поле нужно только для того, чтобы синтаксический анализатор мог точно указать место ошибки, если встретит недопустимую лексему.

Поле `Lexeme` хранит саму подстроку, распознанную как лексема. Оно используется, только если тип лексемы равен `ltIdentifier` или `ltNumber`. Для остальных типов лексем достаточно информации из поля `LexemType`.

Лексический анализатор реализован в виде класса `TLexicalAnalyzer`. В конструкторе класса выполняется разбор строки и формирование списка лексем. Через этот же класс синтаксический анализатор получает доступ к лексемам: свойство `Lexeme` возвращает текущую лексему, метод `Next` позволяет перейти к следующей. Так как наша грамматика предусматривает разбор слева направо, таких примитивных возможностей навигации синтаксическому анализатору вполне хватает. Код анализатора показан в листинге 4.12.

Листинг 4.12. Код лексического анализатора

```
type
  TLexicalAnalyzer = class
  private
    FLexemeList: TList;
    // Номер текущей лексемы в списке
    FIndex: Integer;
    function GetLexeme: PLexeme;
    // Пропуск всего, что является разделителем лексем
    procedure SkipWhiteSpace(const S: string; var P: Integer);
    // Выделение лексемы, начинающейся с позиции P
    procedure ExtractLexeme(const S: string; var P: Integer);
    // Помещение лексемы в список
    procedure PutLexeme(LexemeType: TLexemeType; Pos: Integer;
      const Lexeme: string);
    // Выделение лексемы-числа
    procedure Number(const S: string; var P: Integer);
    // Выделение слова и отнесение его к идентификаторам
    // или зарезервированным словам
    procedure Word(const S: string; var P: Integer);
  public
    constructor Create(const Expr: string);
    destructor Destroy; override;
    // Переход к следующей лексеме
    procedure Next;
    // Указатель на текущую лексему
    property Lexeme: PLexeme read GetLexeme;
  end;

constructor TLexicalAnalyzer.Create(const Expr: string);
var
  P: Integer;
begin
  inherited Create;
```

```
  // Создаем список лексем
  FLexemeList := TList.Create;
  // И сразу же заполняем его
  P := 1;
  while P <= Length(Expr) do
  begin
    SkipWhiteSpace(Expr, P);
    ExtractLexeme(Expr, P);
  end;
  // Помещаем в конец списка специальную лексему
  PutLexeme(ltEnd, P, '');
  FIndex := 0;
end;

destructor TLexicalAnalyzer.Destroy;
var
  I: Integer;
begin
  for I := 0 to FLexemeList.Count – 1 do
    Dispose(PLexeme(FLexemeList[I]));
  FLexemeList.Free;
  inherited Destroy;
end;

// Получение указателя на текущую лексему
function TLexicalAnalyzer.GetLexeme: PLexeme;
begin
  Result := FLexemeList[FIndex];
end;

// Переход к следующей лексеме
procedure TLexicalAnalyzer.Next;
begin
  if FIndex < FLexemeList.Count – 1 then
    Inc(FIndex);
end;

// Помещение лексемы в список. Параметры метода задают
// одноименные поля типа TLexeme.
procedure TLexicalAnalyzer.PutLexeme(LexemeType: TLexemeType;
  Pos: Integer; const Lexeme: string);
var
  NewLexeme: PLexeme;
```

```
begin
  New(NewLexeme);
  NewLexeme^.LexemeType := LexemeType;
  NewLexeme^.Pos := Pos;
  NewLexeme^.Lexeme := Lexeme;
  FLexemeList.Add(NewLexeme);
end;

// Пропускает пробелы, символы табуляции, комментарии и переводы строки,
// которые могут находиться в начале и в конце строки и между лексемами
procedure TLexicalAnalyzer.SkipWhiteSpace(const S: string;
  var P: Integer);
begin
  while (P <= Length(S)) and (S[P] in [' ', #9, #13, #10, '{']) do
    if S[P] = '{' then
    begin
      Inc(P);
      while (P <= Length(S)) and (S[P] <> '}') do
        Inc(P);
      if P > Length(S) then
        raise ESyntaxError.Create('Незавершенный комментарий');
      Inc(P);
    end
    else
      Inc(P);
end;

// Функция выделяет одну лексему и помещает ее в список
procedure TLexicalAnalyzer.ExtractLexeme(const S: string;
  var P: Integer);
begin
  if P > Length(S) then
    Exit;
  case S[P] of
    '(':
      begin
        PutLexeme(ltLeftBracket, P, '');
        Inc(P);
      end;
    ')':
      begin
        PutLexeme(ltRightBracket, P, '');
        Inc(P);
      end;
```

```
'*':
  begin
    PutLexeme(ltAsterisk, P, '');
    Inc(P);
   end;
'+':
  begin
    PutLexeme(ltPlus, P, '');
    Inc(P);
  end;
'-':
  begin
    PutLexeme(ltMinus, P, '');
    Inc(P);
  end;
'/':
  begin
    PutLexeme(ltSlash, P, '');
    Inc(P);
  end;
'0'..'9':
  Number(S, P);
'<':
  if (P < Length(S)) and (S[P + 1] = '=') then
  begin
    PutLexeme(ltLessOrEqual, P, '');
    Inc(P, 2);
  end
  else if (P < Length(S)) and (S[P + 1] = '>') then
  begin
    PutLexeme(ltNotEqual, P, '');
    Inc(P, 2);
  end
  else
  begin
    PutLexeme(ltLess, P, '');
    Inc(P);
  end;
'=':
  begin
    PutLexeme(ltEqual, P, '');
    Inc(P);
  end;
```

```
    '>':
      if (P < Length(S)) and (S[P + 1] = '=') then
      begin
        PutLexeme(ltGreaterOrEqual, P, '');
        Inc(P, 2);
      end
      else
      begin
        PutLexeme(ltGreater, P, '');
        Inc(P);
      end;
    'A'..'Z', 'a'..'z', '_':
      Word(S, P);
    '^':
      begin
        PutLexeme(ltCap, P, '');
        Inc(P);
      end;
    else
      raise ESyntaxError.Create('Некорректный символ в позиции ' +
        IntToStr(P));
  end;
end;

// Выделение лексемы-числа
procedure TLexicalAnalyzer.Number(const S: string; var P: Integer);
var
  InitPos, RollbackPos: Integer;

  function IsDigit(Ch: Char): Boolean;
  begin
    Result := Ch in ['0'..'9'];
  end;

begin
  InitPos := P;
  // Выделяем целую часть числа
  repeat
    Inc(P);
  until (P > Length(S)) or not IsDigit(S[P]);
  // Проверяем наличие дробной части и выделяем ее
  if (P <= Length(S)) and (S[P] = DecimalSeparator) then
  begin
    Inc(P);
```

```
    if (P > Length(S)) or not IsDigit(S[P]) then
      Dec(P)
    else
      repeat
        Inc(P);
      until (P > Length(S)) or not IsDigit(S[P]);
  end;
  // Выделяем экспоненту
  if (P <= Length(S)) and (UpCase(S[P]) = 'E') then
  begin
    // Если мы дошли до этого места, значит, от начала строки
    // и до сих пор набор символов представляет собой
    // синтаксически правильное число без экспоненты.
    // Прежде чем начать выделение экспоненты, запоминаем
    // текущую позицию, чтобы иметь возможность вернуться к ней,
    // если экспоненту выделить не удастся.
    RollBackPos := P;
    Inc(P);
    if P > Length(S) then
      P := RollBackPos
    else
    begin
      if S[P] in ['+', '-'] then
        Inc(P);
      if (P > Length(S)) or not IsDigit(S[P]) then
        P := RollbackPos
      else
        repeat
          Inc(P);
        until (P > Length(S)) or not IsDigit(S[P]);
    end;
  end;
  PutLexeme(ltNumber, InitPos, Copy(S, InitPos, P – InitPos));
end;

// Выделение слова из строки и проверка его на совпадение
// с зарезервированными словами языка
procedure TLexicalAnalyzer.Word(const S: string; var P: Integer);
var
  InitPos: Integer;
  ID: string;
begin
  InitPos := P;
  Inc(P);
```

```
  while (P <= Length(S)) and
    (S[P] in ['0'..'9', 'A'..'Z', 'a'..'z', '_']) do
    Inc(P);
  ID := Copy(S, InitPos, P – InitPos);
  if AnsiCompareText(ID, 'or') = 0 then
    PutLexeme(ltOr, InitPos, '')
  else if AnsiCompareText(ID, 'xor') = 0 then
    PutLexeme(ltXor, InitPos, '')
  else if AnsiCompareText(ID, 'div') = 0 then
    PutLexeme(ltDiv, InitPos, '')
  else if AnsiCompareText(ID, 'mod') = 0 then
    PutLexeme(ltMod, InitPos, '')
  else if AnsiCompareText(ID, 'and') = 0 then
    PutLexeme(ltAnd, InitPos, '')
  else if AnsiCompareText(ID, 'not') = 0 then
    PutLexeme(ltNot, InitPos, '')
  else if AnsiCompareText(ID, 'sin') = 0 then
    PutLexeme(ltSin, InitPos, '')
  else if AnsiCompareText(ID, 'cos') = 0 then
    PutLexeme(ltCos, InitPos, '')
  else if AnsiCompareText(ID, 'ln') = 0 then
    PutLexeme(ltLn, InitPos, '')
  else
    PutLexeme(ltIdentifier, InitPos, ID);
end;
```

В конец списка лексем помещается специальная лексема типа `ltEnd`. В предыдущих примерах приходилось постоянно сравнивать указатель позиции `P` с длиной строки `S`, чтобы не допустить выход за пределы диапазона. Если бы не было лексемы `ltEnd`, точно так же пришлось бы проверять, не вышел ли указатель за пределы списка. Но лексема `ltEnd` не рассматривается как допустимая ни одной из функций синтаксического анализатора, поэтому, встретив ее, каждая из них возвращает управление вызвавшей ее функции, и заканчивается эта цепочка только на функции `Expr`. Таким образом, код получается более ясным и компактным.

Примечание

Аналогичный алгоритм возможен и в предыдущих версиях калькулятора: достаточно добавить в конец строки символ, который в ней заведомо не должен был появляться (например, #1), и проверять в функции `Expr` или `Calculate`, что разбор выражения остановился именно на этом символе.

Лексический анализ выражения заключается в чередовании вызовов функций `SkipWhiteSpace` и `ExtractLexeme`. Первая из них пропускает все, что может разделять две лексемы, вторая распознает и помещает в список одну лексему.

Обратите внимание, как в лексическом анализаторе реализован метод `Number`. Рассмотрим выражение "1e*5". В калькуляторе без лексического анализатора функция `Number`, дойдя до символа "*", выдавала исключение, т. к. ожидала увидеть здесь знак "+", "–" или число. Но лексический анализатор не должен брать на себя такую ответственность — поиск синтаксических ошибок. Поэтому в данном случае он должен, дойдя до непонятного символа в конструкции, которую он счел за экспоненту, откатиться назад, выделить из строки лексему "1" и продолжить выделение лексем с символа "e". В результате список лексем будет выглядеть так: "1", "e", "*", "5". И уже синтаксический анализатор должен потом разобраться, допустима ли такая последовательность лексем или нет.

Отметим, что для нашей грамматики непринципиально, зафиксирует ли в таком выражении ошибку лексический или синтаксический анализатор. Но в общем случае может существовать грамматика, в которой такое выражение допустимо, поэтому лексический анализатор должен действовать именно так, т. е. выполнять откат, если попытка выделить число зашла на каком-то этапе в тупик (самый простой пример — наличие в языке бинарного оператора, начинающегося с символа "e" — тогда пользователь сможет написать этот оператор после числа без пробела, и чтобы справиться с такой ситуацией, понадобится откат). Функция `Number` вызывается из `ExtractLexeme` только в том случае, когда в начале лексемы встречается цифра, а с цифры может начинаться только лексема `ltNumber`. Таким образом, сам факт вызова функции `Number` говорит о том, что в строке гарантированно обнаружена подстрока (состоящая, по крайней мере, из одного символа), которая является числом.

Функции синтаксического анализатора очень похожи на аналогичные функции из предыдущих примеров, за исключением того, что работают не со строкой, а со списком лексем. Поэтому мы приведем здесь только одну из них — функцию `Term` (листинг 4.13).

Листинг 4.13. Пример функции, использующей лексический анализатор

```
const
  Operator2 = [ltAsterisk, ltSlash, ltDiv, ltMod, ltAnd];

function Term(LexicalAnalyzer: TLexicalAnalyzer): Extended;
var
  Operator: TLexemeType;
begin
  Result := Factor(LexicalAnalyzer);
  while LexicalAnalyzer.Lexeme.LexemeType in Operator2 do
  begin
    Operator := LexicalAnalyzer.Lexeme.LexemeType;
    LexicalAnalyzer.Next;
```

```
    case Operator of
      ltAsterisk: Result := Result * Factor(LexicalAnalyzer);
      ltSlash: Result := Result / Factor(LexicalAnalyzer);
      ltDiv: Result := Trunc(Result) div Trunc(Factor(LexicalAnalyzer));
      ltMod: Result := Trunc(Result) mod Trunc(Factor(LexicalAnalyzer));
      ltAnd: Result := Trunc(Result) and Trunc(Factor(LexicalAnalyzer));
    end;
  end;
end;
```

Если сравнить этот вариант `Term` с аналогичной функцией из листинга 4.2, легко заметить их сходство.

Использование лексического анализатора может повысить скорость многократного вычисления одного выражения при разных значениях входящих в него переменных (например, при построении графика функции, введенной пользователем). Действительно, лексический анализ в этом случае достаточно выполнить один раз, а потом пользоваться готовым списком. Можно сделать такие операции еще более эффективными, переложив вычисление числовых констант на лексический анализатор. Для этого в структуру `TLexeme` нужно добавить поле `Number` типа `Extended` и модифицировать метод `Number` таким образом, чтобы он сразу преобразовывал выделенную подстроку в число. Тогда дорогостоящий вызов функции `StrToFloat` будет перенесен из многократно повторяющейся функции `Base` в однократно выполняемый метод `TLexicalAnalyzer.Number`. Но самое радикальное средство повышения производительности — переделка синтаксического анализатора таким образом, чтобы он не вычислял выражение самостоятельно, а формировал машинный код для его вычисления. Однако написание компилятора математических выражений выходит за рамки данной книги.

4.9. Однопроходный калькулятор и функции с несколькими переменными

В предыдущем примере выражение сначала от начала до конца просматривается лексическим анализатором и переводится в иную форму (список лексем). Затем этот список обрабатывается синтаксическим анализатором. Таким образом, калькулятор получается двухпроходным, хотя из синтаксиса и семантики выражения необходимость нескольких проходов не вытекает. Попробуем переделать его так, чтобы он стал однопроходным.

Примечание

В некоторых языках многопроходность — обязательное требование к реализации компилятора. Например, в языке C++ реализацию функций класса можно

вставлять в само описание класса. При этом внутри этих функций можно обращаться к тем полям и функциям класса, которые объявлены ниже. Таким образом, откомпилировать подобный код может только компилятор как минимум с двумя проходами, чтобы на первом проходе можно было найти все поля класса, а на втором — откомпилировать функции класса.

В предыдущей реализации калькулятора синтаксический анализатор работал с лексическим через процедуру `Next` и свойство `Lexeme`: процедура `Next` передвигала текущую позицию в списке лексем, а свойство `Lexeme` давало доступ к текущей лексеме. Легко видеть, что при таком алгоритме лексическому анализатору нет необходимости хранить полный список лексем, достаточно помнить текущую, а при вызове `Next` анализировать очередную часть строки, выделяя из нее следующую лексему и делая ее текущей. Таким образом, синтаксический и лексический анализаторы будут работать по очереди, обрабатывая каждый по одной лексеме.

В реализации лексического анализатора требуются следующие изменения. Во-первых, теперь конструктор не запускает полный цикл лексического анализа, а только сохраняет переданную строку и выделяет из нее первую лексему. Во-вторых, выражение и позиция в выражении теперь должны сохраняться между вызовами методов лексического анализатора и поэтому становятся полями этого класса. В-третьих, метод `Next` теперь выполняет выделение очередной лексемы, которую помещает в специально созданное для этого поле, а свойство `Lexeme` возвращает указатель на это поле, а не на элемент списка. Остальные функции лексического анализатора изменились только в том отношении, что теперь выражение и указатель на позицию в строке получают не через параметры, а напрямую обращаются к соответствующим полям.

Пример однопроходного калькулятора с лексическим анализатором находится на компакт-диске в папке SinglePassSample. В листинге 4.14 показан код той части нового варианта класса `TLexicalAnalyzer`, которую понадобилось изменить, чтобы обеспечить однопроходность.

Листинг 4.14. Однопроходный вариант класса `TLexicalAnalyzer`

```
type
  TLexicalAnalyzer = class
  private
    // Выражение для вычисления
    FExpr: string;
    // Текущая позиция
    FP: Integer;
    // Текущая лексема
    FCurrLexeme: TLexeme;
    function GetLexeme: PLexeme;
```

```
    procedure SkipWhiteSpace;
    procedure ExtractLexeme;
    procedure PutLexeme(LexemeType: TLexemeType; Pos: Integer;
      const Lexeme: string);
    procedure Number;
    procedure Word;
  public
    constructor Create(const Expr: string);
    procedure Next;
    property Lexeme: PLexeme read GetLexeme;
  end;

constructor TLexicalAnalyzer.Create(const Expr: string);
begin
  inherited Create;
  FP := 1;
  FExpr := Expr;
  Next;
end;

// Получение указателя на текущую лексему
function TLexicalAnalyzer.GetLexeme: PLexeme;
begin
  Result := @FCurrLexeme;
end;

// Получение следующей лексемы
procedure TLexicalAnalyzer.Next;
begin
  if FP <= Length(FExpr) then
  begin
    SkipWhiteSpace;
    ExtractLexeme;
  end
  else
    PutLexeme(ltEnd, FP, '');
end;

// Замещение текущей лексемы новой лексемой
procedure TLexicalAnalyzer.PutLexeme(LexemeType: TLexemeType;
  Pos: Integer; const Lexeme: string);
begin
  FCurrLexeme.LexemeType := LexemeType;
```

```
  FCurrLexeme.Pos := Pos;
  FCurrLexeme.Lexeme := Lexeme;
end;
```

Теперь класс `TLexicalAnalyzer` хранит не список лексем, а только одну текущую лексему, а функция `PutLexeme` не добавляет лексему в список, а изменяет значение текущей лексемы. Функция `Next` вместо простого изменения индекса выделяет очередную лексему, т. е. выполняет одну итерацию цикла лексического анализа. Функции `SkipWhiteSpace`, `ExtractLexeme` и т. п. избавились от параметров, через которые передавалось выражение и позиция, потому что теперь выражение и позиция хранятся в полях класса.

Синтаксический анализатор при этом остается без изменений, т. к. интерфейс лексического анализатора не изменился.

Чтобы не реализовывать дважды одну и ту же грамматику, введем в наш синтаксис еще одну возможность — поддержку функций с несколькими аргументами. Конкретно — функцию с двумя аргументами `Log(a, x)`, возвращающей логарифм `x` по основанию `a`, а также функцию `Mean`, которая принимает произвольное число аргументов и возвращает их среднее. Для этого правила, связанные с функциями, переопределим так:

```
<Function> ::= <FuncName> '(' <MathExpr> {<ListSeparator> <MathExpr>} ')'
<FuncName> ::= 'sin' | 'cos' | 'ln' | 'log' | 'mean'
```

Отдельного комментария требует символ `<ListSeparator>`, разделяющий аргументы в функции. В Delphi, как и во многих других языках программирования, таким разделителем служит запятая. Но наша грамматика определена так, что запятая, в принципе, может служить разделителем целой и дробной части числа. Как уже говорилось, в этом случае может возникнуть неоднозначность в выражениях типа `f(1,5)` — это вызов функции `f` то ли с одним аргументом 1.5, то ли с двумя аргументами 1 и 5. Чтобы избежать подобных неоднозначностей, в нашей грамматике разделителем аргументов будет символ, выбранный разделителем элементов списка (в русской локализации Windows это точка с запятой). Для корректной работы программы следите, чтобы на вашем компьютере разделители элементов списка, а также целой и дробной частей не оказались одинаковыми.

Особенность нашего нового синтаксиса в том, что он допускает любое число аргументов для любой функции, т. е., например, выражение `sin(0, 1, 2, 4)` синтаксически корректно (при условии, что разделителем элементов списка является запятая), хотя смысла это выражение не имеет. Можно было бы ввести отдельные синтаксические правила для функций с одним аргументом, с двумя аргументами и с произвольным числом аргументов, но такой подход встречается редко, т. к. обычно намного проще осуществить проверку на эта-

пе семантического анализа (т. е. в нашем случае — при вычислении функции).

Для реализации новых синтаксических и семантических правил в код вносятся следующие изменения. Во-первых, появляются новые лексемы `ltLog`, `ltMean` и `ltListSeparator`, а соответствующие методы лексического анализатора модифицируются так, чтобы распознавать их. Во-вторых, модифицируется функция `Func` — она сначала вычисляет все аргументы, переданные функции, а потом проверяет, является ли количество аргументов допустимым, и если да, вычисляет требуемую функцию.

Для лучшего понимания работы лексического и синтаксического анализатора рекомендуем самостоятельно выполнить следующие задания (или хотя бы просто подумать, как их выполнить).

1. Расширить определение `<Expr>` таким образом, чтобы в нем можно было объединять несколько операций сравнения с помощью `or`, `and`, `xor`. При этом потребуется поддержка скобок, т. к. иначе анализатор во многих случаях не сможет отличить логические операторы с низким приоритетом от одноименных арифметических.
2. Изменить грамматику таким образом, чтобы имя функции стало идентификатором, а не зарезервированным словом.
3. Сделать комментарии вложенными. Сейчас в последовательности символов "{a{b}c}" считается, что комментарий заканчивается перед символом "c", т. к. лексический анализатор игнорирует все открывающие фигурные скобки в комментариях. Сделать так, чтобы комментарий считался закрытым только тогда, когда число закрывающих скобок сравняется с числом открывающих.
4. Добавить поддержку шестнадцатеричных целых констант. Для их записи использовать, как и в Delphi, символ "$", после которого должна идти последовательность из одной или нескольких шестнадцатеричных цифр.
5. Добавить возможность изменения приоритета операций с помощью не только круглых, но и квадратных скобок. Рассмотреть два варианта: когда круглые и квадратные скобки полностью взаимозаменяемы (т. е., например, допустимо выражение `2*(2+2]`) и когда закрывающая скобка должна быть такой же формы, как и открывающая.

Еще одна возможность, которую дает лексический анализатор — это обработка ошибок без исключений (иногда это может быть полезно). Пусть в анализаторе есть флаг, который взводится при обнаружении ошибки. Пока этот флаг сброшен, лексический анализатор работает обычным образом. Но если он взведен, вызов функции `Next` не делает ничего, а свойство `Lexeme` всегда возвращает лексему `ltEnd`, независимо от того, дошел ли анализатор до конца

строки или нет. После выполнения анализа проверяется этот флаг, и по его состоянию делается вывод о том, произошла ли ошибка. Соответственно, лексический анализатор должен иметь метод для установки этого флага извне, чтобы синтаксический анализатор мог его установить при обнаружении ошибки.

Примечание

Флагом можно сделать строковое поле, хранящее сообщение об ошибке. Пока эта строка пуста, флаг считается сброшенным, когда строка не пуста, считается, что флаг взведен. Таким образом, синтаксический анализатор формирует при необходимости сообщение об ошибке и помещает его в это поле лексического анализатора, и тот переходит в "ошибочный" режим. Так мы обеспечиваем и реализацию флага, и передачу сообщения об ошибке. В этом случае в структуре `TLexeme` можно избавиться от поля `Pos` — позицию последней выделенной лексемы можно сделать внутренним полем лексического анализатора, и тот сам добавит номер позиции к сообщению, сформированному синтаксическим анализатором.

4.10. Еще немного теории

Теперь, познакомившись с синтаксическим анализом на практике, вернемся к теории и немного поговорим о типах грамматик и об альтернативных методах синтаксического анализа и вычисления выражений. Эти вопросы мы здесь рассмотрим только ознакомительно, а более детальное их описание можно найти в [6—8].

Грамматики языков по способу описания можно разделить на четыре типа, причем каждый следующий тип является подмножеством предыдущего.

1. *Общие грамматики*. Синтаксические правила в этих грамматиках имеют вид `a ::= b`, где `a` и `b` — произвольные цепочки из терминальных и нетерминальных символов (возможно, пустые). Единственное требование — хотя бы в одной из этих цепочек должен быть хотя бы один нетерминальный символ.
2. Контекстно-зависимые грамматики. Здесь правила имеют следующий вид `a<X>b ::= acb`, где `a`, `b` и `c` — произвольные цепочки терминальных и нетерминальных символов, `<X>` — некоторый нетерминальный символ. Таким образом, символ `<X>` может заменяться на последовательность символов `c` только в контексте цепочек `a` и `b`.
3. *Контекстно-свободные грамматики*. Это контекстно-зависимые грамматики, из которых убран контекст, т. е. правила записываются в виде `<X> ::= c`. В контекстно-свободных грамматиках нетерминальный символ `<X>` заменяться на цепочку `c` в любом контексте.

4. *Регулярные* (они же — *автоматные*) грамматики. Это контекстно-свободные грамматики, в которых запрещены любые формы рекурсивных определений.

Из этих определений легко сделать вывод, что в данной главе, пока мы не ввели в выражения скобки, наши грамматики относились к классу регулярных, а со скобками — к классу контекстно-свободных грамматик. Что же касается первых двух классов грамматик, то они неудобны ни для распознавания человеком, ни для написания анализаторов, поэтому данные грамматики применяются, в основном, только для описания естественных языков.

Регулярные грамматики описывают множество синтаксических правил, встречающихся в жизни, поэтому их часто применяют. Существует также альтернативный способ записи регулярной грамматики — *регулярные выражения* (мы их здесь рассматривать не будем). Различные библиотеки для распознавания регулярных выражений очень популярны, классы для распознавания регулярных выражений входят в .NET. Функция поиска в Delphi (меню **Search/Find...** и т. п.) включает в себя возможности поиска последовательностей символов, заданных регулярным выражением (опция **Regular expressions** в диалоговом окне), поэтому краткое описание синтаксиса регулярных выражений можно найти в справке Delphi.

Примечание

Рекурсии в регулярных выражениях очень не хватает, когда нужно описать, например, возможность бесконечной вложенности скобок. Поэтому в некоторых анализаторах к регулярным выражениям добавляется возможность описывать бесконечное вложение структур. Эти выражения, строго говоря, уже не являются регулярными, хотя их обычно продолжают так называть.

С регулярными грамматиками тесно связаны конечные автоматы. *Конечный автомат* — это устройство (виртуальное), с входом, на который подаются данные, набором состояний и набором правил перехода из одного состояния в другое. Правила перехода определяются символами, подаваемыми на вход, и формулируются следующим образом: "Если автомат находится в состоянии A, и на вход поступил символ X, автомат переходит в состояние B". Таким образом, выражение посимвольно передается на вход конечного автомата, и каждый символ вызывает переход автомата из одного состояния в другое (допустима ситуация, когда символ оставляет текущее состояние неизменным). Если при поступлении очередного символа автомат не находит правила, которое определяет очередной переход, считается, что на вход подан некорректный символ, т. е. выражение ошибочно. Допустимость выражения определяется также тем, в каком состоянии оказывается автомат после того, как все выражение подано на его вход. Часть состояний считается допустимыми в качестве конечного состояния, часть — недопустимыми. Если по окончании своей работы автомат оказывается в недопустимом состоянии, выражение также признается ошибочным.

Можно доказать, что для каждой регулярной грамматики можно построить конечный автомат, и, наоборот, для каждого конечного автомата можно построить регулярную грамматику. Именно поэтому регулярные грамматики называются также автоматными.

Конечный автомат очень наглядно представляется с помощью графа, узлами которого служат состояния автомата, ребрами — переходы между состояниями. Каждое ребро помечается символами, при поступлении на вход которых этот переход становится возможным. На рис. 4.3 показан пример такого изображения конечного автомата, соответствующего грамматике вещественного числа. Кружки с одинарной границей изображают состояния, недопустимые в качестве конечного, с двойной границей — допустимые. До начала работы автомат находится в состоянии 0, каждый следующий символ переводит его в соответствующее состояние. Конечное состояние 1 соответствует числу без дробной части и экспоненты, состояние 3 — числу с дробной частью без экспоненты, состояние 6 — числу с экспонентой.

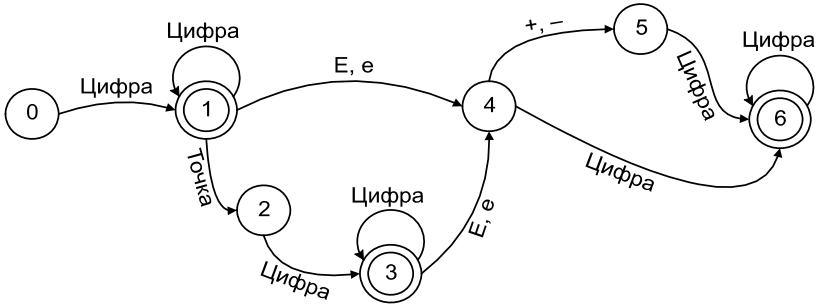


Рис. 4.3. Конечный автомат для грамматики вещественного числа

Контекстно-свободные автоматы не пригодны для распознавания контекстно-свободных грамматик с рекурсией. Для этого класса грамматик можно применить *МП-автоматы* (автоматы с магазинной памятью). Эти автоматы обладают стеком, и символ, поступающий на вход, не только определяет правило перехода, но и может быть сохранен в стеке, а правила перехода могут учитывать не только поступивший на вход символ, но и символ, лежащий на вершине стека. Если символ на вершине стека учитывается правилом, при применении этого правила символ извлекается из стека.

Главное достоинство МП-автоматов по сравнению с методом рекурсивного спуска (так называется метод построения синтаксического анализатора, который мы использовали) является то, что код автомата универсален и может быть применен к любому набору правил. Таким образом, появляется возмож-

ность создавать анализаторы, правила для которых хранятся, например, во внешнем файле или в базе данных, и грамматика может быть изменена без перекомпиляции анализатора. Недостатки МП-автоматов — малая наглядность кода и медленная работа из-за возможности захода в тупиковые ветки. Поэтому метод рекурсивного спуска применяется всегда, когда нет нужды менять грамматику во время работы программы.

В книге [6] описана интересная разновидность МП-автоматов — табличный анализатор, который в некоторых случаях может оказаться предпочтительнее метода рекурсивного спуска.

Арифметические выражения, которые мы разбирали в этой главе, записаны в привычной нам *инфиксной форме*, т. е. когда знак бинарной операции ставится между операндами. Кроме инфиксной, существует также *префиксная* и *постфиксная* формы записи выражения, в которых оператор записывается, соответственно, перед и после операндов. Например, выражение "2+2" в префиксной форме запишется как "+ 2 2", в постфиксной — "2 2 +". Префиксная форма называется польской записью, постфиксная — польской инверсной записью (в честь польского математика Яна Лукасевича, который разработал эти формы записи).

Достоинства префиксной и постфиксной форм записи — отсутствие скобок и одинаковый приоритет всех операций. Например, выражение "2+(2*2)" в постфиксной записи имеет вид "2 2 * 2 +", а выражение "(2+2)*2", соответственно, "2 2 + 2 *". Операции всегда выполняются в том порядке, в котором они следуют в выражении.

Примечание

Префиксная запись имеет много общего с принятым обозначением функций. Представим, что в некотором языке программирования нет встроенной инфиксной операции сложения, но есть функция +, которая принимает два аргумента и возвращает их сумму, и аналогичные функции для других бинарных операторов. В привычной форме записи функций, когда аргументы заключаются в скобки, приведенное выражение будет выглядеть так: "+(2, *(2, 2))". Теперь достаточно убрать из него скобки и запятые, чтобы получить префиксную запись выражения в классическом виде. Постфиксная запись получается из функциональной подобным образом, надо только ввести правило, что имя функции пишется не перед списком аргументов, а после него.

По своим выразительным возможностям постфиксная и префиксная записи равноценны, но при вычислении выражения, заданного префиксной записью, требуется рекурсивный алгоритм, а при вычислении выражения в постфиксной записи достаточно линейного алгоритма и стека, поэтому чаще встречается постфиксная форма. Алгоритм вычисления постфиксного выражения очень прост. Если очередная лексема — это число, кладем его в стек. Если очередная лексема — бинарный оператор, выталкиваем из стека два верхних

значения, применяем к ним операцию и результат помещаем обратно в стек. Алгоритм легко обобщается на операторы с любым количеством операндов: соответствующая операция выталкивает из стека не два, а нужное ей число параметров. Функция от N аргументов рассматривается как операция, применяющаяся к N операндам.

Простота постфиксной записи делает ее очень привлекательной для низкоуровнего программирования. Методом рекурсивного спуска достаточно легко создать код, переводящий выражение из инфиксной формы в постфиксную, а затем вычислить выражение уже в постфиксной форме. В простейшем случае такой промежуточный перевод только замедляет вычисления, и поэтому не используется, но иногда (например, при многократном вычислении одного выражения) перевод в постфиксную запись может сильно ускорить вычисления, тем более что выражение в постфиксной форме можно хранить не в виде строки, а в виде списка лексем, что еще больше ускорит его вычисление. В частности, код для стековой Java-машины, вычисляющий выражения, по сути эквивалентен постфиксной записи выражения.

Конечно, синтаксический анализ — вещь непростая, и здесь мы рассмотрели только самые его основы. За рамками книги остались атрибутивные грамматики, семантические деревья, генераторы языков и многое другое. Этим сложным вопросам посвящены специализированные книги. Долгое время ощущалась нехватка книг по данной тематике, но за последние два года вышли сразу три книги ([6—8]), посвященные созданию трансляторов. В этих книгах детально разбираются фундаментальные основы теории и даются примеры ее использования. Особенно стоит отметить книгу [6], в которой описан очень интересный язык программирования — Оберон-2, созданный при участии Никлауса Вирта; в нем развиваются идеи, заложенные Виртом в Паскаль. Ряд идей, предложенных при создании различных версий Оберона, уже позаимствованы другими языками (Java, C#, Ада), и еще многие ждут своего часа, поэтому программисту следует хотя бы ознакомительно изучить Оберон, чтобы понимать, в каком направлении может пойти развитие языков программирования.

В качестве источника полезных сведений можно также посоветовать книги, посвященные не столько теории разработки языков программирования, сколько истории их развития, например, [5, 9]. Теория синтаксического и семантического анализа в них изложена относительно неглубоко, но тесная связь изложения с практическими примерами позволяет существенно расширить кругозор в данной области. Особенно рекомендуем [5]. Книга [9] содержит больше сведений, но написана более тяжелым языком, а ее авторы крайне предвзято относятся к Паскалю, ставя ему в вину его достоинства и упрекая в несуществующих недостатках. Тем не менее эту книгу тоже следует прочитать.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ 1

Сайт "Королевство Delphi"

Эта книга появилась на свет благодаря сайту "Королевство Delphi" (**http://www.delphikingdom.com**), поэтому будет справедливо, если мы уделим ему здесь немного внимания. Тем более что этот сайт сам по себе интересен для программиста, использующего Delphi. Главная страница сайта показана на рис. П1.1.

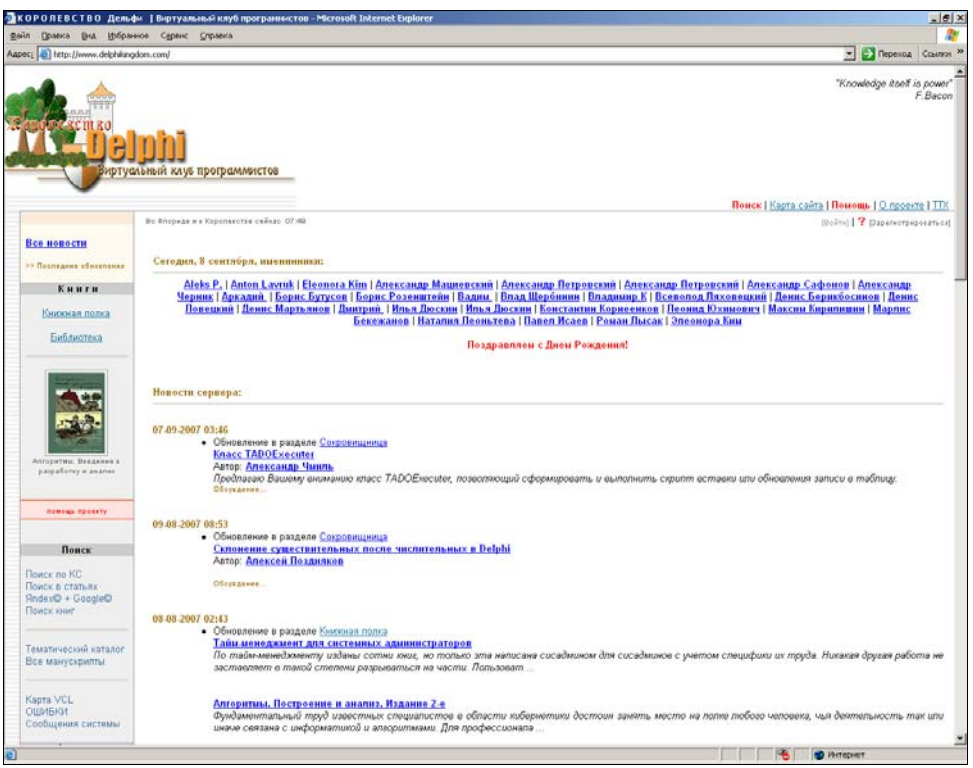


Рис. П1.1. Главная страница сайта "Королевство Delphi"

История сайта "Королевство Delphi" началась 20 ноября 1998 года (об истории создания см. страницу **http://www.delphikingdom.com/team/about.asp**). Он задумывался как виртуальный клуб программистов для взаимопомощи независимо от географии и расстояний (для тех, кто в Интернете недавно, заметим, что в 1998 году тематических форумов практически не было, и до такой идеи еще надо было додуматься). На данный момент "Королевство Delphi" является одним из самых популярных сайтов, посвященных Delphi. В Королевстве имеется форум (который называется "Круглый стол"), где можно задавать вопросы и ряд разделов для публикации различных материалов: от небольших советов до серьезных циклов статей. Королевство принципиально не копирует чужие статьи и публикует только оригинальные материалы, написанные специально для сайта и присланные лично авторами. Некоторые количественные характеристики сайта (по состоянию на 7 сентября 2007 года) приведены в табл. П1.1 (информация взята со страницы **http://www.delphikingdom.com/asp/tth.asp**).

Таблица П1.1. *Характеристики сайта "Королевство Delphi"*

Наименование показателя	Значение
Зарегистрировано жителей	15 351
Опубликовано материалов	905
Задано вопросов	48 348
Из них с ответами	47 335
Всего дано ответов	179 704
В среднем в день задается вопросов	26
В среднем в день дается ответов	115

Сайт "Королевство Delphi" был создан Еленой Филипповой (**http://www.delphikingdom.com/asp/users.asp?ID=10**) и некоторое время она работала в одиночку. Сейчас сайт поддерживается командой из шести человек во главе с Еленой (свои впечатления о ведении этого проекта Елена с недавних пор начала описывать в блоге, который находится по адресу **http://delphikingdom.blogspot.com**). Команда Королевства поддерживает контакты с российским отделением CodeGear, благодаря чему в новостной ленте появляется информация о проводимых этой компанией мероприятиях и об интересных новостях, связанной с ней. Кроме того, на встречу с генеральным директором CodeGear Джимом Дугласом, посетившим Россию в июне 2007 года, были приглашены два представителя Королевства (с отчетом об этой встрече можно познакомиться по адресу **http://www.delphikingdom.com/asp/viewitem.asp?catalogid=1320**).

На сайте "Королевство Delphi" присутствует легкий антураж настоящего средневекового королевства, из-за чего разделы имеют непривычные названия. Чтобы разобраться во всех этих непонятных ссылках, требуются определенные усилия, которые, впрочем, вознаграждаются. Для тех, кто заинтересовался сайтом, приведем описание его основных разделов.

Круглый стол. Так в Королевстве называется форум, где каждый может задать вопрос. Вопросы сначала просматривает модератор и только потом они появляются (или не появляются) на Круглом столе. Если модератор принял решение отклонить вопрос, автору вопроса по почте отправляется уведомление с описанием причин, по которым вопрос отклонен. Наиболее частая причина отклонения — такой вопрос уже задавался ранее (и, может быть, много раз), ответы на него уже даны. В этом случае письмо будет содержать ссылки на ответы или рекомендации, как эти ответы быстро найти. Тем не менее не стоит злоупотреблять этой особенностью сайта и сразу задавать вопрос, надеясь, что ответ будет получен если не на форуме, то через модератора. Королевство имеет целый ряд сервисов для самостоятельного поиска ответа, и такой поиск в итоге дает больше знаний, чем готовый ответ. Задать вопрос и написать ответ можно без предварительной регистрации.

Для удобства навигации вопросы на Круглом столе можно сортировать по дате поступления или по дате последнего ответа. Можно также получить список вопросов, заданных за определенный период, и вопросов, на которые даны ответы за определенный период. Для зарегистрированных пользователей доступна также выборка всех своих вопросов, всех вопросов, на которые пользователь давал ответы, и сервис "Избранные вопросы", с помощью которого пользователь может отметить заинтересовавшие его вопросы и отслеживать появление ответов на них. Уведомления о новых вопросах и ответах при желании можно получать с помощью RSS.

Вопросы на Круглом столе остаются навсегда, они не отправляются в архив, и ссылки на них остаются действительными. Обсуждение вопроса не закрывается (за исключением случаев, когда модератор закрывает обсуждение из-за нарушения автором правил), поэтому все, даже самые старые вопросы, можно не только прочитать, но и что-то ответить или попросить уточнить, если это потребуется.

Существует список offtopic-вопросов. Туда попадают проблемы, которые обсуждаются часто, и их решения уже есть на Круглом столе. Каждый, кто спрашивает о чем-то, согласно правилам Королевства сначала должен ознакомиться с этим списком, чтобы не задать такой вопрос, который там уже есть.

Круглый стол посвящен только решению конкретных технологических проблем. Вопросы, связанные, например, с обсуждением стоимости работы, про-

читанных книг, новостей программирования и т. п., сюда не попадают. Для них есть отдельные форумы.

Базарная площадь. В этом разделе можно обсуждать "неформатные" для Круглого стола темы. Но они все же должны иметь какое-то отношение к компьютерным наукам. Темы ориентированы на длительные обсуждения, есть такие, в которых обсуждение ведется в течение многих лет, то затухая, то вновь возобновляясь. В подобной ситуации неизбежны постоянные лирические отступления от темы ветки слегка в сторону, но модераторы к этому не придираются, если это не заходит совсем уж далеко. Это создает на Базарной площади атмосферу неформального общения на интересные темы.

Городская площадь. Этот раздел предназначен для поиска и предложений работы и сотрудничества. Любой работодатель может разместить здесь объявление о вакансии программиста, а программист — о поиске работы. Также допускаются сообщения о поиске и предложении разовой работы. Отдельный раздел на Городской площади посвящен поиску подельника для совместной работы "за так" над каким-нибудь интересным проектом.

Также на Городской площади можно размещать объявления о поиске готовых компонентов. Их нужно задавать именно здесь, на Круглом столе они не приветствуются, потому что основная цель Круглого стола — помочь человеку с чем-то разобраться, что-то понять, а не дать ссылку на готовое решение проблемы.

Помимо форумов, в Королевстве также публикуются статьи. Для них отведено несколько разных разделов, чтобы можно было не мешать в одну кучу статьи, разные по стилистике, глубине охвата темы, требованиям к уровню читающего.

Сокровищница. Сюда попадают небольшие статьи, посвященные частным задачам. Здесь можно найти интересное решение какой-то проблемы или поделиться своей находкой.

Подземелье магов. Этот раздел предназначен для статей, посвященных технологиям, которые считаются уделом "крутых спецов". В основном, материалы ориентированы на изложение основ соответствующих технологий для тех, кто с ними не знаком. Отметим, что именно здесь опубликованы статьи Михаила Краснова, посвященные использованию OpenGL в Delphi, которые затем легли в основу его книги "OpenGL в проектах Delphi".

Свитки. Этот раздел посвящен обзорным статьям, описывающим какие-то общие проблемы. Обзоры могут быть как технического характера, так и общефилософского. Хотя многое из того, что там написано, можно с успехом применять на практике.

Hello, World! Назначение этого раздела понятно из его названия. Сюда помещаются статьи для начинающих. Вопросы могут быть достаточно слож-

ными (поэтому многие статьи из Hello, World! могли бы быть отнесены и к Подземелью магов), но основное требование раздела — подробное объяснение, рассчитанное на начинающего. На того, кто может не знать не только данную тему, но и многие другие вещи, так что небольшие экскурсы в сторону в этих статьях вполне допустимы.

Лицей. По своей целевой аудитории этот раздел очень близок к Hello, World!, но отличается от него по характеру изложения. Здесь публикуются не отдельные статьи, а циклы связанных уроков, посвященные какой-то обширной теме или общим вопросам Delphi. Среди авторов Лицея такие известные в сообществе Delphi люди, как Юрий Зотов и Анатолий Подгорецкий.

Подводные камни. В этом разделе (его ведет Александр Малыгин) собирается информация о ситуациях, когда что-то работает не так, как ожидалось. Это может происходить по самым разным причинам: из-за аппаратной ошибки, ошибок компилятора, библиотек и самого программиста. Главный критерий — чтобы ошибка оказалась там, где ее не ждут. Основные требования к статьям раздела: это должно быть четкое описание ошибки (по возможности с примером), должны быть описаны условия ее возникновения (версия Delphi, операционной системы и т. п.) и пути решения. Раздел уже превратился в достаточно большую коллекцию подводных камней, подстерегающих программиста.

Полигон. Здесь публикуются законченные решения достаточно объемных вопросов. Обязательное условие публикации — наличие готового модуля или библиотеки, которые можно скачать и использовать. При этом Королевство Delphi не является хранилищем компонентов и не конкурирует с такими сайтами, как, например, **http://www.torry.net**. Публикация компонента в разделе Полигон — это приглашение к участию в совместном тестировании и оценке получившегося кода. В результате автор получает подсказки, что можно и нужно исправить, как еще можно расширить компонент, а остальные посетители сайта — код, который можно применить в своих программах.

Королевские Хроники. В этом разделе собираются статьи, посвященные событиям, происходящим вокруг Delphi, и интересным людям, которые имеют какое-то отношение к Delphi. В этот раздел попал, например, уже упоминавшийся отчет о встрече представителей российского сообщества Delphi с генеральным директором CodeGear Джимом Дугласом, а также впечатления от лекции Никлауса Вирта в Политехническом музее в Москве. Здесь же опубликованы интервью, которые специально для Королевства Delphi давали интересные для сообщества Delphi люди, в частности, такие известные авторы книг по Delphi, как Валерий Фаронов, Анатолий Тенцер и Валентин Озеров.

Все статьи во всех разделах доступны для оценки и обсуждения. Каждый посетитель сайта может оценить стиль изложения и актуальность сведений,

приведенных в статье, добавить свой комментарий или задать вопрос автору. Посетители сайта имеют возможность подписаться на получение по RSS уведомлений о новых комментариях к заинтересовавшей их статье.

Одна из самых острых проблем для всех форумов, посвященных программированию — это то, что информация быстро обновляется, и полезные сведения быстро "тонут" под все новыми и новыми слоями других ответов (может быть, не менее полезных). И поэтому вопросы, на которые уже были даны исчерпывающие ответы, задаются снова и снова. Опубликованные статьи постепенно тоже опускаются вниз, прячась под новыми статьями. В Королевстве Delphi предусмотрены специальные средства для поиска ответов на свой вопрос среди уже накопленных материалов. Помимо уже упоминавшегося списка offtopic-вопросов и обычного поиска по Круглому столу и по статьям, существуют еще четыре раздела, назначение которых также заключается в том, чтобы помочь найти имеющуюся на сайте информацию.

Тематический каталог. В Королевстве создан каталог тем (общим числом более шестисот), относящихся к Delphi. Каждый вопрос на Круглом столе и каждая статья из любого раздела может быть связана с одной или несколькими темами. Тематический каталог позволяет выбрать тему и посмотреть все вопросы и все статьи, связанные с ней. Для удобства поиска темы в каталоге упорядочены в иерархическую структуру.

Карта VCL. Помимо тем, каждый вопрос и статья могут быть связаны с одним или несколькими классами из стандартной библиотеки VCL. С помощью раздела Карта VCL можно получить список всех вопросов и статей, связанных с нужным классом.

ОШИБКИ. В этом разделе собрана информация о различных ошибках, которые могут возникнуть при компиляции и выполнении программы, при взаимодействии с конкретным пакетом и т. п. Общее число сообщений об ошибке в списке Королевства — около восьмисот. Вопросы и статьи могут привязываться к ним, а в разделе ОШИБКИ, соответственно, можно получить список вопросов и статей, связанных с требуемой ошибкой. Для удобства нахождения нужной ошибки в списке предусмотрен поиск по ключевому слову в сообщениях об ошибке.

Системные сообщения. Еще один список, к элементам которого могут привязываться вопросы и статьи, — это оконные сообщения Windows. В этом разделе можно получить список вопросов и статей, в которых идет речь о заданном оконном сообщении.

Королевство содержит также еще ряд разделов, которые трудно отнести к какой-либо группе.

Фолианты. В этом разделе публикуется информация о книгах, относящихся к программированию: краткая аннотация, изображение обложки и ссылки на

эту книгу в различных интернет-магазинах. По каждой книге посетитель Королевства может оставить свой отзыв и прочитать отзывы других посетителей. Здесь же помещаются рецензии на книги.

Арсенальная башня. Здесь публикуется справочная информация о различных сторонних утилитах и библиотеках, которые могут быть полезны при разработке программ на Delphi. По каждой утилите даются краткие сведения о ее предназначении и ссылка на сайт данной утилиты. Для удобства поиска утилиты разбиты по категориям.

Рыцарский зал. Здесь перечислены все зарегистрировавшиеся в Королевстве люди. Отдельно отмечены люди, имеющие какие-то особые заслуги перед Королевством, например, опубликовавшие статью или написавшие какой-нибудь сценарий для сайта. Регистрация не является обязательной, но некоторые неосновные сервисы незарегистрированным пользователям недоступны по техническим причинам. При регистрации указание адреса электронной почты обязательно, но сам адрес не публикуется. Со страницы посетителя в Рыцарском зале ему можно отправить сообщение по почте.

Служба личных сообщений (СЛС). С помощью этого сервиса зарегистрированные пользователи могут отправлять друг другу сообщения, которые можно прочитать на сайте Королевства.

Глас народа! В этом разделе проводятся опросы по разным актуальным для программистов темам.

Книга Песка. Здесь собираются ссылки на сайты по Delphi и интернет-ресурсы смежной тематики. Ссылки упорядочены по темам, благодаря чему легко получить список сайтов нужной тематики.

Дальние земли. Здесь представлены ссылки на сайты, с которыми у Королевства установились особо дружественные отношения вплоть до проведения совместных проектов. Когда Королевство только-только появилось, существовало очень много сайтов, посвященных Delphi (буквально каждый второй программист считал необходимым создать свой сайт), и казалось, что в этом множестве сайтов действительно следует выделять особо близкие. Но в дальнейшем количество сайтов резко уменьшилось (далеко не у всех хватило сил наполнять сайт оригинальным содержанием), и смысл в особом выделении кого-то пропал. Поэтому раздел практически пустой.

Школа ОБЕРОНА. Этот раздел посвящен языку Оберон и его потомкам. Оберон — на данный момент последний из разработанных Никлаусом Виртом языков, в котором заложен ряд интересных решений. В разделе Школа ОБЕРОНА собраны ссылки на ресурсы, посвященные этому языку, а также ссылки на несколько обсуждений на Базарной площади, которые имеют к нему отношение.

В Королевстве регулярно проводятся конкурсы, позволяющие выявить тех, кто лучше других отвечает на вопросы, появляющиеся на Круглом столе. В период проведения конкурса каждый зарегистрированный посетитель Королевства может проголосовать за понравившийся ему ответ, указав, на какую из возможных номинаций он его выдвигает. Голоса проверяются жюри (состоящем из членов команды Королевства) на соответствие требованиям номинации, несоответствующие отбраковываются, и составляется рейтинг отвечающих. По рейтингу определяется победитель в каждой из номинаций. Победители получают значок щита к своему нику и венок в личную страницу в Рыцарском зале. Обычно в конкурсе три номинации: "Самый фундаментальный ответ", "Самый терпеливый рыцарь" и "Хороший совет". Первая подразумевает ответы, которые не только устраняют проблему, но и детально объясняют, почему она возникает и почему решать ее следует именно так. Вторая номинация предназначена для людей, которые терпеливо объясняют что-либо, раз за разом возвращаясь к одному и тому же вопросу и дают дополнительные пояснения до тех пор, пока автор вопроса не поймет все. Под третью номинацию попадают ответы, в которых автору вопроса действительно дан полезный совет, но с обязательной оговоркой, что вопрос не должен быть слишком простым, а ответ на него — тривиальным.

Королевство Delphi — принципиально некоммерческий проект, на котором отсутствует реклама. Поддержка и развитие сайта осуществляется исключительно за счет энтузиазма команды Королевства и его жителей, которые время от времени помогают улучшать сайт. В настоящее время сайт настолько разросся, что найти для него бесплатный хостинг стало невозможно, поэтому для оплаты хостинга собираются добровольные пожертвования. В соответствии с принципиальной позицией команды Королевства все эти пожертвования анонимны, и пожертвовавший не получает никаких преимуществ по сравнению с теми, кто не заплатил.

Автор данной книги является постоянным посетителем Королевства Delphi. В эту книгу вошел ряд статей, опубликованных автором на страницах Королевства и переработанных с учетом замечаний и пожеланий, высказанных посетителями сайта. Связаться с автором можно через его личную страницу в Рыцарском зале **http://www.delphikingdom.com/asp/users.asp?ID=73**.

Заметка, посвященная данной книге, находится по адресу **http://www.delphikingdom.com/asp/viewitem.asp?catalogid=1326**, и если вы хотите, чтобы ваш отзыв или вопрос увидел не только автор, пишите комментарий к этой заметке.

ПРИЛОЖЕНИЕ 2

Содержимое компакт-диска

Прилагаемый к книге компакт-диск содержит примеры программ, разобранные в тексте книги. Примеры разбиты на четыре папки, каждая из которых соответствует одной главе. Все примеры могут быть откомпилированы в любой версии Delphi, начиная с 5-й.

Примеры к главе 1

Примеры к первой главе находятся в папке **1 Windows API и Delphi**. Содержимое папки приведено в табл. П2.1.

Таблица П2.1. *Примеры к главе 1*

Папка	Подпапка	Описание	Разделы главы
Основы работы с Win API в VCL-приложениях	EnumWnd	Пример работы с функцией EnumWnd	1.1.5. Функции обратного вызова, 1.1.13. Строки в Windows API, 1.2.1. Пример EnumWnd
	Line	Пример перехвата невизуальным компонентом сообщений формы-владельца	1.1.8. Обработка сообщений с помощью VCL, 1.2.2. Пример Line
	CoordLabel	Пример перехвата визуальным компонентом сообщений родительского окна	1.1.8. Обработка сообщений с помощью VCL, 1.2.3. Пример CoordLabel
	PanelMsg	Пример перехвата формой сообщений расположенного на нем компонента и обработки перехваченного сообщения `WM_PAINT`	1.1.8. Обработка сообщений с помощью VCL, 1.1.10. Особые сообщения, 1.1.11. Графика в Windows API, 1.2.4. Пример PanelMsg

Таблица П2.1 *(окончание)*

Папка	Подпапка	Описание	Разделы главы
	NumBroadcast	Пример регистрации глобального сообщения, его широковещательной отправки и получения	1.1.8. Обработка сообщений с помощью VCL, 1.1.9. Сообщения, определяемые пользователем, 1.2.5. Пример NumBroadcast
	ButtonDel	Пример удаления кнопки при ее нажатии	1.1.8. Обработка сообщений с помощью VCL, 1.2.6. Пример ButtonDel
	GDIDraw	Пример использования графических средств GDI, не поддерживающихся классом TCanvas	1.1.11. Графика в Windows API, 1.2.7. Пример GDIDraw
	BitmapSpeed	Программа для сравнения скорости различных операций на DDB- и DIB-растрах	1.1.11. Графика в Windows API, 1.2.8. Пример BitmapSpeed
Обобщающий пример 1	ProcInfo	Программа, показывающая информацию о запущенных в системе процессах и открытых ими окнах	1.3.1. Обобщающий пример 1 — Информация о процессах
Обобщающий пример 2	DKSView	Программа, регистрирующая в реестре расширение своих файлов и не допускающая запуска двух копий одновременно	1.3.2. Обобщающий пример 2 — Ассоциированные файлы и предотвращение запуска второй копии приложения
Обобщающий пример 3	WndHole	Программа, демонстрирующая, как сделать окно с прямоугольным отверстием, размеры которого могут изменяться пользователем	1.3.3. Обобщающий пример 3 — "Дырявое" окно
Обобщающий пример 4	Lines	Пример рисования прямых нестандартными стилями и создания "резиновой" линии	1.3.4. Обобщающий пример 4 — Линии нестандартного стиля
	Bezier	Пример рисования кривых Безье нестандартными стилями и создания "резиновой" кривой	1.3.4. Обобщающий пример 4 — Линии нестандартного стиля

Примеры к главе 2

Примеры ко второй главе находятся в папке **2 Использование сокетов в Delphi**, содержимое которой приведено в табл. П2.2.

Таблица П2.2. *Примеры к главе 2*

Папка	Описание	Разделы главы
UDPChat	Простейший чат с использованием UDP. Прием и отправка сообщений в разных нитях через различные сокеты	2.1.5. Протокол UDP, 2.1.8. Создание сокета, 2.1.9. Передача данных при использовании UDP, 2.1.10. Пример программы — простейший чат на UDP
SimplestServer	Простейший TCP-сервер, реализованный в виде консольного приложения и работающий на блокирующих сокетах в одной нити. Способен взаимодействовать только с одним клиентом одновременно	2.1.6. Протокол TCP, 2.1.8. Создание сокета, 2.1.11. Передача данных при использовании TCP, 2.1.12. Примеры передачи данных с помощью TCP
SimpleClient	Простой TCP-клиент, использующий блокирующие сокеты и работающий с одним сервером	2.1.6. Протокол TCP, 2.1.8. Создание сокета, 2.1.11. Передача данных при использовании TCP, 2.1.12. Примеры передачи данных с помощью TCP
MultithreadedServer	Многонитевой TCP-сервер на блокирующих сокетах, работающий с неограниченным числом клиентов. Для каждого клиента создается отдельная нить	2.1.6. Протокол TCP, 2.1.8. Создание сокета, 2.1.11. Передача данных при использовании TCP, 2.1.12. Примеры передачи данных с помощью TCP
SelectChat	UDP-чат с одним сокетом и одной нитью и для приема, и для отправки сообщений. Для определения момента получения данных используется функция `select`	2.1.13. Определение готовности сокета, 2.1.14. Примеры использования функции *select*
SelectServer	Однонитевой TCP-сервер, работающий на блокирующих сокетах и способный к взаимодействию одновременно с неограниченным числом клиентов. Для определения момента получения данных используется функция `select`	2.1.13. Определение готовности сокета, 2.1.14. Примеры использования функции *select*
NonBlockingServer	Однонитевой TCP-сервер, работающий на неблокирующих сокетах и способный к взаимодействию одновременно с неограниченным числом клиентов	2.1.15. Неблокирующий режим, 2.1.16. Сервер на неблокирующих сокетах
AsyncSelectServer	TCP-сервер, использующий оконные сообщения для взаимодействия с неограниченным числом клиентов	2.2.5. Асинхронный режим, основанный на сообщениях, 2.2.6. Пример сервера, основанного на сообщениях

Таблица П2.2 *(окончание)*

Папка	Описание	Разделы главы
EventSelectServer	Многонитевой TCP-сервер. Нить, устанавливающая подключения, и нити, взаимодействующие с клиентами, управляются событиями, связанными с сокетами	2.2.7. Асинхронный режим, основанный на событиях, 2.2.8. Пример использования сокетов с событиями
EventSelectClient	TCP-клиент, поддерживающий расширенную версию протокола обмена, реализованную в примере EventSelectServer. Использует сокеты, основанные на оконных сообщениях	2.2.5. Асинхронный режим, основанный на сообщениях, 2.2.7. Асинхронный режим, основанный на событиях, 2.2.8. Пример использования сокетов с событиями
OverlappedServer	TCP-сервер на основе перекрытого ввода-вывода с использованием процедур завершения	2.2.9. Перекрытый ввод-вывод, 2.2.10. Сервер, использующий перекрытый ввод-вывод

Примеры к главе 3

Примеры к третьей главе находятся в папке **3 Подводные камни**, содержимое которой приведено в табл. П2.3.

Таблица П2.3. *Примеры к главе 3*

Папка	Подпапка	Описание	Разделы главы
Неочевидные особенности целых чисел	Assignment1	Пример потери значения при присваивании беззнаковой переменной отрицательного значения	3.1.2. Выход за пределы диапазона при присваивании
	Assignment2	Пример потери значения при присваивании переменной значения, большего, чем допускается типом	3.1.2. Выход за пределы диапазона при присваивании
	Overflow1	Пример перехода беззнакового значения через ноль при вычитании	3.1.3. Переполнение при арифметических операциях
	Overflow2	Пример невозможности контроля переполнения с помощью опции {$R+} при использовании функции Dec	3.1.3. Переполнение при арифметических операциях
	Compare1	Пример корректного сравнения знакового и беззнакового числа	3.1.4. Сравнение знакового и беззнакового числа

Таблица П2.3 *(продолжение)*

Папка	Подпапка	Описание	Разделы главы
	Compare2	Пример некорректного сравнения знакового и беззнакового числа при использовании приведения типов	3.1.4. Сравнение знакового и беззнакового числа
	ForRange	Пример неправильного вычисления границы диапазона цикла `for` при использовании беззнаковой управляющей переменной	3.1.5. Неявное преобразование в цикле *for*
Неочевидные особенности вещественных чисел	WrongValue	Пример присваивания вещественной переменной значения, отличного от заданного в программе	3.2.6."Неправильное" значение
	Compare1	Пример ошибки при сравнении вещественной переменной и вещественного литерала	3.2.7. Сравнение
	Compare2	Пример ошибки при сравнении вещественных переменных разных типов	3.2.8. Сравнение разных типов
	Subtraction	Пример накопления ошибки при многократном вычитании	3.2.9. Вычитание в цикле
	Epsilon	Пример поиска машинного эпсилон (минимального числа, которое при добавлении к единице дает значение, отличное от единицы)	3.2.12. Машинное эпсилон
Тонкости работы со строками	Constants	Пример, позволяющий исследовать, где в различных ситуациях хранятся строковые литералы и как они присваиваются переменным	3.3.2. Хранение строковых литералов
	PCharLit	Пример, демонстрирующий, что явное приведение строковых литералов к типу `PChar` в большинстве случаев бесполезно, а иногда приводит к ошибке	3.3.3. Приведение литералов к типу *PChar*
	Comparisons	Пример, демонстрирующий то, как в различных ситуациях компилятор сравнивает строки	3.3.4. Сравнение строк

Таблица П2.3 *(продолжение)*

Папка	Подпапка	Описание	Разделы главы
	SideChange	Пример нежелательного изменения значения строковой переменной при низкоуровневом изменении значения другой строковой переменной	3.3.5. Побочное изменение
	Zero	Пример, демонстрирующий невозможность правильного преобразования строки, содержащей символ `#0`, из `AnsiString` в `PChar`	3.3.6. Нулевой символ в середине строки
	ZeroFind	Пример неправильной работы функции `AnsiPos` со строками, содержащими символ `#0`	3.3.6. Нулевой символ в середине строки
	StringResult	Пример, демонстрирующий особый способ хранения результата функции, возвращающей строковое значение	3.3.7. Функция, возвращающая *AnsiString*
	RecordReadWrite	Примеры, демонстрирующие правильные и неправильные способы записи типов, содержащих строки, в поток	3.3.8. Строки в записях
	RecordCopy	Пример, демонстрирующий возникновение нежелательных эффектов при низкоуровневом копировании записей, содержащих строки	3.3.8. Строки в записях
Прочие подводные камни	OpOrder	Пример того, что компилятор может вычислять операнды бинарной операции в порядке, отличном от интуитивно ожидаемого	3.4.1. Порядок вычисления операндов
	UpDownDlg	Пример зацикливания обработчика нажатия кнопки мыши компонента `TUpDown` из-за неоправданного захвата мыши в монопольное использование	3.4.2. Зацикливание обработчика *TUpDown. OnClick* при открытии диалогового окна в обработчике
	CloseAV	Пример возникновения ошибки в перекрытом методе `WndProc` из-за неправильной реализации метода `TCustomForm.Release`	3.4.3. Access violation при закрытии формы с перекрытым методом *WndProc*

Таблица П2.3 *(окончание)*

Папка	Подпапка	Описание	Разделы главы
	ClassName	Пример, демонстрирующий, где хранится имя оконного класса, возвращаемое функцией `GetClassInfo`, и как эта память может быть использована для других нужд раньше, чем указатель на нее покинет область видимости	3.4.4. Подмена имени оконного класса, возвращаемого функцией *GetClassInfo*
	ListIndex	Пример, демонстрирующий ошибку обращения к свойству `TComboBox.Items.Objects` при значении свойства, равном –1	3.4.6. Ошибка List index out of bounds при корректном значении индекса
	WrongAnchors	Пример того, что компоненты на форме располагаются не так, как предписывает свойство `Anchors`, если начальный размер формы во время выполнения программы не совпадает с размером, заданным при проектировании, и методы борьбы с этой проблемой	3.4.7. Неправильное поведение свойства *Anchors*
	MethodPtrCmp	Пример генерирования компилятором неправильного кода при сравнении указателей на методы и способ решения этой проблемы	3.4.8. Ошибка при сравнении указателей на метод
	ParentWnd	Пример возникновения ошибки при использовании в деструкторе оконного компонента свойств, требующих существования окна	3.4.10. Невозможность использования некоторых свойств оконного компонента в деструкторе
	FrameDel	Пример скрытой ошибки при использовании свойств, требующих существования окна, в деструкторе фрейма: исключение не возникает, но происходит утечка ресурсов	3.4.10. Невозможность использования некоторых свойств оконного компонента в деструкторе

Примеры к главе 4

Примеры к четвертой главе находятся в папке **4 Разбор и вычисление выражений**, содержимое которой приведено в табл. П2.4.

Таблица П2.4. *Примеры к главе 4*

Папка	Описание	Разделы главы
IsNumberSample	Пример анализа выражения на предмет соответствия синтаксису вещественного числа. Анализирует введенную пользователем строку и возвращает результат "Число" или "Не число"	4.3. Синтаксис вещественного числа
SimpleCalcSample	Пример простейшего калькулятора с четырьмя действиями арифметики над числами без учета приоритета операций	4.4. Простой калькулятор
PrecedenceCalc-Sample	Пример калькулятора с четырьмя действиями арифметики над числами с учетом приоритета операций	4.5. Учет приоритета операторов
BracketsCalcSample	Пример простейшего калькулятора с четырьмя действиями арифметики над числами с возможностью изменять приоритет операций с помощью круглых скобок	4.6. Выражения со скобками
FullCalcSample	Пример калькулятора, в котором поддерживаются переменные, функции и возведение в степень	4.7. Полноценный калькулятор
LexicalSample	Пример двухпроходного калькулятора с лексическим анализатором. Лексический анализатор дает возможность вставлять в выражение пробелы, переводы строки и комментарии	4.8. Калькулятор с лексическим анализатором
SinglePassSample	Пример однопроходного калькулятора с лексическим анализатором. Добавлена также поддержка функций с несколькими аргументами	4.9. Однопроходной калькулятор и функции с несколькими переменными

Список литературы

1. Фень Юань. Программирование графики для Windows. — СПб.: Питер, 2002.
2. Рихтер Дж. Windows. Для профессионалов. — СПб.: Питер, 2000.
3. Джонс Э., Оланд Д. Программирование в сетях Microsoft Windows. — СПб.: Питер; М.: Издательско-торговый дом "Русская редакция", 2002.
4. Вишневский П. Длинные строки и динамические массивы в Delphi // RSDN Magazine, № 3, 2004.
5. Себеста Роберт У. Основные концепции языков программирования, 5-е изд.: Пер. с англ. — М.: Издательский дом "Вильямс", 2001.
6. Свердлов С. З. Языки программирования и методы трансляции. — СПб.: Питер, 2007.
7. Карпов Ю. Г. Теория и технология программирования. Основы построения трансляторов. — СПб.: БХВ-Петербург, 2005.
8. Опалева Э. А., Самойленко В. П. Языки программирования и методы трансляции. — СПб.: БХВ-Петербург, 2005.
9. Пратт Т., Зелковиц М. Языки программирования: разработка и реализация, 4-е изд.: Пер. с англ. — СПб.: Питер, 2002.

Предметный указатель

A

API (Application Programming Interface) 14
◊ константа ALTERNATE 125
◊ константа CB_ERR 475
◊ константа CLR_DEFAULT 77
◊ константа CLR_NONE 77
◊ константа CW_USEDEFAULT 477
◊ константа ERROR_IO_PENDING 391
◊ константа FILE_FLAG_SEQUENTIAL_SCAN 393
◊ константа GA_PARENT 145
◊ константа GM_ADVANCED 131
◊ константа GW_OWNER 145
◊ константа GWL_WNDPROC 45
◊ константа HTCAPTION 157, 158
◊ константа HTCLIENT 157, 158
◊ константа HTLEFT 157
◊ константа HWND_BROADCAST 42, 117
◊ константа INFINITE 328, 366
◊ константа LB_ERR 475
◊ константа R2_COPYPEN 171
◊ константа R2_NOT 172
◊ константа WAIT_IO_COMPLETION 328, 366
◊ константа WINDING 125
◊ константа WM_APP 71
◊ константа WM_USER 71, 153
◊ константа WS_CHILD 30—32, 145, 483
◊ константа WS_EX_APPWINDOW 52
◊ макрос PaletteIndex 77
◊ макрос PaletteRGB 77
◊ макрос RGB 77
◊ макрос UNICODE 83
◊ получение справки 15
◊ тип TColorRef 76
◊ тип TOverlapped 390, 394
◊ функции CreateDialogXXXX 43, 44
◊ функции DialogBoxXXXX 44
◊ функция BeginPaint 73, 75, 111, 113
◊ функция BeginPath 123, 175
◊ функция BitBlt 172
◊ функция BroadcastSystemMessage(Ex) 42, 117
◊ функция CallWindowProc 45
◊ функция CharToOem 82
◊ функция CoCreateInstance 26, 422
◊ функция CombineRgn 158
◊ функция CoTaskMemAlloc 455
◊ функция CreateEllipticRgn 158
◊ функция CreateEvent 153
◊ функция CreateFile 393
◊ функция CreateFont 130, 131

API (Application Programming Interface) *(прод.)*:
◊ функция CreatePolygonRgn 158
◊ функция CreateProcess 148
◊ функция CreateRectRgn 158
◊ функция CreateWindow(Ex) 30, 48, 51, 143, 422
◊ функция DefDlgProc 38, 44
◊ функция DefFrameProc 38, 65, 144
◊ функция DefWindowProc 38, 65, 144
◊ функция DeleteObject 27
◊ функция DestroyWindow 483
◊ функция DispatchMessage 38, 41, 42, 54, 56, 62
◊ функция DrawText(Ex) 83, 123
◊ функция EndPaint 73, 113
◊ функция EndPath 123, 175
◊ функция EnumChildWindows 89, 93, 94, 140
◊ функция EnumWindows 33, 89, 93, 140
◊ функция ExtTextOut 83
◊ функция FlattenPath 175
◊ функция FormatMessage 220
◊ функция FreeProcInstance 35
◊ функция GetAncestor 145
◊ функция GetChildMDIProc 38
◊ функция GetClassInfo 144, 471
◊ функция GetCurrentDirectory 88
◊ функция GetDC(Ex) 75
◊ функция GetDeviceCaps 80
◊ функция GetKeyState 74
◊ функция GetMessage 38, 39, 44, 73, 74
◊ функция GetMessagePos 43
◊ функция GetMessageTime 43
◊ функция GetModuleHandle 29
◊ функция GetOverlappedResult 391
◊ функция GetParent 145
◊ функция GetPath 175
◊ функция GetStockObject 79
◊ функция GetSysColor 78
◊ функция GetWindowDC 75
◊ функция GetWindowInfo 138
◊ функция GetWindowLong 45
◊ функция GetWindowLongPtr 46
◊ функция GetWindowText 86, 88, 93, 143
◊ функция GetWindowTextLength 88
◊ функция GetWindowThreadProcessID 140
◊ функция GrayString 128
◊ функция HeapAlloc 455
◊ функция InvalidateRect 73
◊ функция InvalidateRgn 73
◊ функция InvertRect 127
◊ функция IsDialogMessage 43, 56
◊ функция IsWindowUnicode 53
◊ функция LineDDA 168, 170, 175, 179
◊ функция LoadLibrary 14, 29, 420
◊ функция MakeProcInstance 35
◊ функция OemToChar 82
◊ функция OpenProcess 139
◊ функция PathToRegion 123, 175
◊ функция PeekMessage 38, 41, 44, 53, 73, 74
◊ функция PolyBezier 174
◊ функция PolyBezierTo 174
◊ функция PolyDraw 174, 175
◊ функция PolyPolygon 125, 126
◊ функция PostMessage 38, 72, 73, 117
◊ функция PostThreadMessage 41
◊ функция PtInRegion 167
◊ функция QueryPerforamceCounter 134
◊ функция QueryPerforamceFrequency 134
◊ функция RealGetWindowClass 138, 144
◊ функция Rectangle 175
◊ функция RedrawWindow 73
◊ функция RegisterWindowMessage 71, 83, 84, 113, 117
◊ функция SelectClipRgn 123
◊ функция SelectObject 78, 130

◊ функция SendMessage 44, 65, 70, 72, 73, 143, 483
◊ функция SendMessageTimeout 143
◊ функция SetEvent 325
◊ функция SetForegroundWindow 155
◊ функция SetGraphicsMode 131
◊ функция SetMapMode 76
◊ функция SetPixel 172
◊ функция SetROP2 171
◊ функция SetTimer 73
◊ функция SetWindowLong 45, 51, 68
◊ функция SetWindowLongPtr 46
◊ функция SetWindowRgn 156, 158
◊ функция SetWindowText 86
◊ функция SetWorldTransform 76
◊ функция ShellExecute 26, 145, 147, 148
◊ функция SHGetSpecialFolderPath 88
◊ функция ShowWindow 27
◊ функция SleepEx 366
◊ функция TextOut 83, 128
◊ функция TranslateAccelerator 43, 56
◊ функция TranslateMDISysAccel 43, 56
◊ функция TranslateMessage 40, 56
◊ функция UpdateWindow 73
◊ функция VirtualAlloc 455
◊ функция WaitForMultipleObjects 325
◊ функция WaitForSingleObject 139, 153, 325, 326
◊ функция WaitMessage 58, 59
◊ функция WidenPath 175

D

DoS-атака 252, 262, 267, 299, 334, 344, 351

E

Ethernet 194, 195

F

FPU 410
◊ команда FLDCW 413
◊ команда FNSTCW 413
◊ управляющее слово 413, 419, 420

G

GDI (Graphic Device Interface) 74, 171, 179
◊ алгоритмы рисования линий 168
◊ вывод текста под углом 129
◊ кривые Безье 173
◊ нестандартный стиль линии 168
◊ растровые операции 171, 172
◊ регион отсечения 122, 123
◊ регионы 76, 122, 158, 165
◊ режим заливки 124
◊ траектории 76, 123, 174
◊ цвета 76

I

Indy 288, 395
IP-адрес 195, 212

M

MAC-адрес 194
MSDN (Microsoft Developer's Network) 16
MSLU (Microsoft Layer for Unicode) 83

O

OSI (Open Software Interconnection) 192

P

PSAPI 137

T

TApplication:
◊ метод HandleMessage 57, 60

TApplication *(прод.)*:
◊ метод Idle 57, 58, 60, 61
◊ метод IsDlgMsg 56
◊ метод IsHintMessage 54
◊ метод IsKeyMsg 56
◊ метод IsMDIMsg 56
◊ метод IsPreProcessMessage 54
◊ метод Minimize 155
◊ метод ProcessMessage 52, 57, 58, 70
◊ метод ProcessMessages 58, 120
◊ метод Restore 155
◊ метод Run 58
◊ метод WakeMainThread 58
◊ свойство ActionUpdateDelay 57
◊ свойство Handle 52
◊ событие OnIdle 57, 60
◊ событие OnMessage 54, 70, 117, 470
◊ событие OnShortCut 56
TBitmap:
◊ свойство Handle 27
◊ свойство HandleType 81
◊ свойство Palette 27
◊ свойство PixelFormat 80
◊ свойство ScanLine 81, 131, 135
TCanvas:
◊ метод Polygon 74
◊ свойство Handle 74, 75
TClientSocket 395
TComboBox, свойство Items.Objects 474, 482
TComponent:
◊ свойство ComponentState 484
◊ свойство Owner 30
TControl:
◊ метод Perform 65, 70
◊ метод SetParent 107
◊ свойство Anchors 476
◊ свойство Constraints 159, 161
◊ свойство Parent 30, 32, 52, 107, 483
◊ событие OnMouseDown 171
◊ событие OnMouseEnter 61
◊ событие OnMouseExit 61
◊ событие OnMouseMove 171
◊ событие OnMouseUp 171
TControlParent, метод SetParent 486
TCustomForm:
◊ метод Release 118, 468
◊ метод ShowModal 60
◊ свойство Position 477
◊ свойство WindowState 478
◊ событие OnClose 466
◊ событие OnHide 27
◊ событие OnShortCut 56
TCustomGrid, неправильное поведение дочерних компонентов 68
TFont:
◊ свойство Handle 130, 131
◊ свойство Orientation 129, 131
THintWindow, метод IsHintMsg 54
TListBox, свойство Items.Objects 474
TObject:
◊ метод DefaultHandler 64, 65, 69, 98
◊ метод Dispatch 62, 64, 65, 98, 118
Tool Help 137
◊ константа TH32CS_SNAPMODULE 140
◊ константа TH32CS_SNAPPROCESS 138
◊ снимок 138
◊ функция CreateToolhelp32Snapshot 138, 140
◊ функция Module32First 140
◊ функция Module32Next 140
◊ функция Process32First 138
◊ функция Process32Next 138
TServerSocket 395
TThread:
◊ метод Queue 59
◊ метод RemoveQueuedEvents 59
◊ метод Synchronize 57, 59
TTimer, событие OnTimer 74
TTL (Time To Live) 195, 199, 388
TUpDown:
◊ свойство Min 464
◊ свойство Position 464
◊ событие OnChanging 465
◊ событие OnChangingEx 465
◊ событие OnClick 464

TWinControl:
◊ метод CreateHandle 47
◊ метод CreateParams 49, 65, 477, 487
◊ метод CreateSubClass 144
◊ метод CreateWnd 49, 483
◊ метод DefaultHandler 72
◊ метод Dispatch 468
◊ метод GetHandle 47
◊ метод HandleNeeded 47
◊ метод MainWndProc 61, 65, 118, 468
◊ метод PaintControls 112
◊ метод PreProcessMessage 54, 70
◊ метод WndProc 61, 65, 69, 72, 118, 466
◊ свойство DefWndProc 65, 69
◊ свойство Handle 47, 68, 483, 486
◊ свойство WindowProc 61, 65, 69, 98, 108, 161, 466, 468
◊ событие OnKeyDown 40
◊ событие OnPress 40

W

WinSock API:
◊ версии библиотек 204, 286, 385, 388
◊ инициализация 204
◊ константа AF_INET 205, 207, 301, 385
◊ константа CF_ACCEPT 298
◊ константа CF_DEFER 298
◊ константа CF_REJECT 298
◊ константа FD_ACCEPT 308
◊ константа FD_CLOSE 308
◊ константа FD_CONNECT 308, 330, 387
◊ константа FD_MAX_EVENTS 331
◊ константа FD_READ 306, 307, 329, 330
◊ константа FD_SETSIZE 253
◊ константа FD_WRITE 330
◊ константа FIONBIO 269
◊ константа FIONREAD 269
◊ константа FROM_PROTOCOL_INFO 294
◊ константа INADDR_ANY 210, 216, 296, 298, 385
◊ константа INADDR_NONE 210
◊ константа INVALID_SOCKET 206, 232
◊ константа IP_ADD_MEMBERSHIP 384, 385
◊ константа IP_DROP_MEMBERSHIP 385
◊ константа IP_MULTICAST_IF 388
◊ константа IP_MULTICAST_LOOP 387, 388
◊ константа IP_MULTICAST_TTL 388
◊ константа IPPROTO_IP 206, 384
◊ константа IPPROTO_RDP 206
◊ константа IPPROTO_TCP 206, 293
◊ константа IPPROTO_UDP 206, 293
◊ константа JL_BOTH 386
◊ константа JL_RECEIVER_ONLY 386
◊ константа JL_SENDER_ONLY 386
◊ константа MSG_PARTIAL 389
◊ константа MSG_PEEK 218
◊ константа PF_INET 207, 209
◊ константа SD_BOTH 213
◊ константа SD_RECEIVE 213
◊ константа SD_SEND 213, 308
◊ константа SIOCATMARK 268
◊ константа SO_BROADCAST 284
◊ константа SO_CONDITIONAL_ACCEPT 299
◊ константа SO_CONNECT_TIME 391
◊ константа SO_LINGER 285
◊ константа SO_RCVBUF 283
◊ константа SO_SNDBUF 283
◊ константа SO_UPDATE_ACCEPT_CONTEXT 392
◊ константа SOCKET_ERROR 205
◊ константа SOL_SOCKET 283, 284, 391
◊ константа SOMAXCONN 232

WinSock API *(прод.)*:
◊ константа TF_USE_DEFAULT_WORKER 395
◊ константа TF_USE_KERNEL_APC 395
◊ константа TF_USE_SYSTEM_THREAD 395
◊ константа WSA_FLAG_MULTIPOINT_C_LEAF 384
◊ константа WSA_FLAG_MULTIPOINT_D_LEAF 384
◊ константа WSA_FLAG_OVERLAPPED 294, 359
◊ константа WSA_INFINITE 328
◊ константа WSA_INVALID_EVENT 326
◊ константа WSA_MAXIMIUM_WAIT_EVENTS 328
◊ константа WSA_WAIT_EVENT_0 328
◊ константа WSA_WAIT_FAILED 328
◊ константа WSA_WAIT_TIMEOUT 328
◊ константы FD_XXX_BIT 331
◊ макрос FD_CLR 253
◊ макрос FD_ISSET 253
◊ макрос FD_SET 253, 257
◊ макрос FD_ZERO 253
◊ макрос WSAGETSELECTERROR 306
◊ макрос WSAGETSELECTEVENT 306
◊ ошибка WSA_IO_INCOMPLETE 362
◊ ошибка WSA_IO_PENDING 361, 363
◊ ошибка WSAECANCELLED 289
◊ ошибка WSAECONNREFUSED 299
◊ ошибка WSAEINPROGRESS 289
◊ ошибка WSAEMSGSSIZE 217
◊ ошибка WSAENOBUFS 293
◊ ошибка WSAEWOULDBLOCK 270, 272, 307, 308
◊ ошибка WSATRY_AGAIN 299
◊ семейство адресов 205
◊ семейство протоколов 207
◊ соглашения об именах 189
◊ тип PROTOCOL_INFO 293
◊ тип sockaddr 207
◊ тип sockaddr_in 208
◊ тип TConditionProc 297
◊ тип TFDSet 253, 311
◊ тип TGroup 294
◊ тип TIPMreq 384
◊ тип TSockaddr 208, 300
◊ тип TSockaddrIn 208
◊ тип TSocket 206
◊ тип TTimeVal 254
◊ тип TTransmitFileBuffers 394
◊ тип TWSABuf 360
◊ тип TWSAEvent 326
◊ тип TWSANetworkEvents 331
◊ тип TWSAOverlapped 361, 365, 391
◊ тип TWSAProtocolInfo 290, 294
◊ функции htons, htonl, ntohs, ntohl 191, 302
◊ функции WSAHtons, WSAHtonl, WSANtohs, WSANtohl 302
◊ функция accept 232—234, 255, 268, 296, 299, 308, 329, 389
◊ функция AcceptEx 389
◊ функция bind 206, 216, 218, 230, 233, 308, 384
◊ функция closesocket 214, 232, 285, 303, 309, 387, 392
◊ функция connect 218, 230, 232, 233, 255, 268, 270, 295, 299, 308, 387
◊ функция EnumProtocols 293, 388
◊ функция GetAcceptExSockAddrs 392
◊ функция getsockopt 205, 282, 391
◊ функция inet_addr 210, 288, 300
◊ функция ioctlsocket 268
◊ функция listen 231, 233, 255
◊ функция recv 219, 230—232, 235, 255, 268, 329, 359, 388, 392

◊ функция recvfrom 217, 221, 229, 230, 255, 268, 389
◊ функция select 252, 270, 311
◊ функция send 219, 230—232, 235, 255, 268, 272, 283, 308, 359, 387, 392
◊ функция sendto 215, 218, 224, 229, 230, 255, 268, 283, 308, 387
◊ функция setsockopt 282, 392
◊ функция shutdown 213, 214, 308
◊ функция socket 205, 215, 233, 294, 359, 384
◊ функция setsockopt 384, 385
◊ функция TransmitFile 392, 393
◊ функция WSAAccept 296, 299, 308, 329, 390
◊ функция WSAAddressToString 301
◊ функция WSAAsyncSelect 329
◊ функция WSACancelBlockingCall 289
◊ функция WSACleanup 204, 205
◊ функция WSACloseEvent 327
◊ функция WSAConnect 295, 387
◊ функция WSACreateEvent 326, 328
◊ функция WSADuplicateSocket 303
◊ функция WSAEnumNetworkEvents 330
◊ функция WSAEnumProtocols 292
◊ функция WSAEventSelect 328
◊ функция WSAGetLastError 204, 205, 272
◊ функция WSAGetOverlappedResult 359, 362, 367, 391
◊ функция WSAIsBlocking 289
◊ функция WSAJoinLeaf 386
◊ функция WSARecv 359, 365, 388, 392
◊ функция WSARecvDisconnect 299
◊ функция WSARecvEx 388
◊ функция WSARecvFrom 369
◊ функция WSAResetEvent 326, 330
◊ функция WSASend 358, 359, 370, 387, 392
◊ функция WSASendDisconnect 299
◊ функция WSASendTo 370, 387
◊ функция WSASetBlockingHook 289
◊ функция WSASetEvent 326, 331
◊ функция WSASocket 294, 384
◊ функция WSAStartup 204, 286
◊ функция WSAStringToAddress 300
◊ функция WSAUnhookBlockingHook 289
◊ функция WSAWaitForMultipleEvents 327, 329, 331, 361, 363, 366

WSAAsyncSelect:
◊ функция accept 306
◊ функция recv 306
◊ функция recvfrom 306
◊ функция WSAAsyncSelect 304
◊ функция WSADuplicateSocket 305
◊ функция WSARecv 306
◊ функция WSARecvFrom 306

А

Автомат:
◊ конечный 535
◊ с магазинной памятью 536
Алгоритм Нагля 201

В

Вещественные числа:
◊ INF 412
◊ NAN 412
◊ двоичное представление 408
◊ денормализованное представление 411, 412
◊ мантисса 408
◊ машинное эпсилон 423
◊ нормализованное представление 411
◊ экспонента 408
Виртуальный код клавиши 40
Всплывающая подсказка (Hint) 54
Вызов унаследованных protected-методов 113
Выравнивание в записях 22
Выражение 490

Г

Главная форма 52
Глобальные переменные с локальной областью видимости 127
Грамматика 492
◊ LR(1) 503, 511
◊ автоматная 535
◊ графический способ записи 498
◊ контекстно-зависимая 534
◊ контекстно-свободная 534
◊ общая 534
◊ однозначность и неоднозначность 503
◊ регулярная 535

Д

Дейтаграмма 198, 216, 217
Дескриптор 26
◊ контекста устройства 75
◊ модуля 29
◊ невидимого окна приложения 52
◊ объекта процесса 139
◊ оконного компонента 47
◊ родительского окна 30
◊ снимка (Tool Help) 138
◊ события 153
◊ сокета 190, 232
◊ сокетного события 326
Диалоговая процедура 44
Динамически компонуемая библиотека (DLL) 14, 456
◊ динамическая загрузка 14
◊ статическая загрузка 14
Директива:
◊ export 36
◊ message 62, 69, 72, 116, 118, 468
◊ register 15
◊ stdcall 15, 33, 288
Директивы компилятора:
◊ $H, $LONGSTRINGS 425
◊ $Q, $OVERFLOWCHECKS 404
◊ $R, $RANGECHECKS 402, 404
◊ $REALCOMPATIBILITY 410

З

Захват (capture) мыши 173, 465

И

Идентификатор процесса (PID) 138
Инфиксная форма 537
Исключение:
◊ EAccessViolation 88, 105, 118, 406, 430, 432, 441, 456, 461, 467
◊ EConvertError 501
◊ EIntOverflow 404
◊ EInvalidOperation 483
◊ ERangeError 402, 403
◊ EReadError 473
◊ EStringListError 474

К

Клиент 200, 212, 230
Кодировки ANSI и Unicode 20, 54, 82, 293
Коллизия в сети 194
Командная строка 148
Компоненты:
◊ компонент-владелец 30, 98
◊ оконные и неоконные 28
◊ передача сообщений неоконным компонентам 61, 67
◊ родительский компонент 28, 106
Контекст устройства 74

Л

Лексема 491, 517
◊ разделители между лексемами 519
Лексический анализатор 491, 517
Локальный узел 195, 230

М

Макроопределение (макрос) 189
Маршрутизатор 192, 194, 197, 387
Менеджер памяти 452, 456, 461, 472
◊ FastMM 457
◊ предоставление в общее пользование 457
TWinControl 47
Метод рекурсивного спуска 536
Методы-обработчики сообщений 62, 69, 118, 468
Модель:
◊ OSI 192, 193
◊ вызова функции 15, 33, 288
Модуль:
◊ ShareMem 457
◊ SimpleShareMem 458

О

Общие типы 399
Окно 27
◊ MDI 30, 38, 43, 56, 65
◊ верхнего уровня 52
◊ диалоговое 38, 43, 44, 60
◊ клиентская и неклиентская область 156
◊ модальная форма 60
◊ невидимое окно приложения 52, 117, 469, 487
◊ нестандартная форма 156
◊ окно-владелец 32, 52, 145
◊ перевод на передний план 155
◊ родительское и дочернее 30, 145
◊ стили 30, 49, 145
Оконная процедура 29, 36, 49, 60
◊ стандартные 38, 144
Оконные сообщения 36
◊ BN_CLICKED 68
◊ CB_GETCOUNT 483
◊ CB_GETITEMDATA 474, 486
◊ CB_SETITEMDATA 486
◊ CM_APPKEYDOWN 57
◊ CM_EXECPROC 59
◊ CM_MOUSEENTER 61
◊ CM_MOUSELEAVE 61
◊ CM_RECREATEWND 68
◊ CM_RELEASE 468
◊ CN_COMMAND 68
◊ CN_KEYDOWN 57
◊ WM_CANCELMODE 466
◊ WM_CHAR 40
◊ WM_COMMAND 67, 118
◊ WM_COPYDATA 72
◊ WM_DESTROY 484, 488
◊ WM_DISPLAYCHANGE 186
◊ WM_ENTERIDLE 45, 60
◊ WM_GETTEXT 64, 93, 143
◊ WM_GETTEXTLENGTH 37, 64, 93, 94
◊ WM_KEYDOWN 40
◊ WM_LBUTTONDOWN 70, 156, 157, 465
◊ WM_LBUTTONUP 465
◊ WM_MOUSEHOVER 61
◊ WM_MOUSELEAVE 61
◊ WM_NCCALCSIZE 157

Окно *(прод.)*:
◊ WM_NCHITTEST 157, 158, 161, 167
◊ WM_NCLBUTTONDOWN 156, 157
◊ WM_NCMOUSEMOVE 63
◊ WM_NCPAINT 156
◊ WM_NOTIFY 465
◊ WM_NULL 58
◊ WM_PAINT 72, 108, 111, 112, 156
◊ WM_QUIT 39, 53, 54, 58, 60
◊ WM_RBUTTONDBLCLK 108, 111
◊ WM_SETICON 37
◊ WM_SETTEXT 64
◊ WM_SIZE 62, 65, 158, 159, 161, 163, 164
◊ WM_SIZING 158, 159, 161
◊ WM_TIMER 73, 74
◊ глобальные сообщения 70
◊ отправка 38
◊ очередь сообщений 38
◊ посылка 38
◊ сообщения оконного класса 70
◊ сообщения приложения 70
◊ широковещательные 42, 113
Оконный класс 28, 144, 471
◊ подкласс 45
◊ системные классы 29, 71

П

Перевод объявлений с C++ на Delphi 19
◊ struct 22
◊ union 22
◊ void 23
Перекрытый ввод-вывод 153, 295, 358, 389, 394
◊ процедура завершения 359, 365, 371
Переменная:
◊ Application 52, 117, 469
◊ CreationControl 50
◊ DDBsOnly 81
◊ DecimalSeparator 500
◊ Default8087CW 413
◊ HInstance 29
◊ HPrevInstance 150
◊ MainInstance 29
◊ RandSeed 136
◊ WakeMainThread 58
Петля сообщений 38
◊ локальная 41, 58
◊ реализация в VCL 58, 60
Польская запись 537
Почтовый ящик (mailslot) 150
Правила именования типов 22
◊ дублирование типа TBitmap 22
Предложение 490
Предотвращение запуска второй копии 150
Префиксная и постфиксная форма 537
Процедуры и функции Delphi:
◊ _LStrToPChar 440, 442
◊ _LStrToPhar 87
◊ AnsiPos 443
◊ CheckSyncronize 57, 58
◊ ColorToRGB 78
◊ CompareString 443
◊ Dec 404, 427
◊ Dispose 455
◊ FillChar 211
◊ Finalize 455
◊ FloatToStr 416, 424
◊ FloatToStrF 424
◊ FreeMem 455
◊ Get8087CW 413, 419
◊ GetMem 455
◊ GetRValue, GetGValue, GetBValue 77
◊ Inc 427
◊ Initialize 455
◊ InitWndProc 49, 61
◊ MakeObjectInstance 60, 118, 468
◊ Move 452
◊ New 455
◊ ParamCount 149
◊ ParamStr 149
◊ Pos 443
◊ Randomize 136

◊ SafeLoadLibrary 420
◊ Set8087CW 413, 419
◊ SetMemoryManager 457
◊ SetPrecisionMode 414
◊ StrComp 433
◊ StrDispose 89, 427
◊ StrNew 87, 427
◊ StrPos 443
◊ StrToFloat 500
◊ UniqueString 439, 440, 442

Р

Разделение пакетов 201
Растры, форматы DIB и DDB 79, 131
Регулярные выражения 535
Рельсовая диаграмма 498

С

Семантика 490
Семантический анализ 490
Сервер 200, 211, 231
Сетевой протокол:
◊ ARP 196
◊ HTTP, FTP, SMTP 197
◊ ICMP 196
◊ IGMP 196, 382
◊ IP 194
◊ RDP 206
◊ TCP 197, 199, 206
◊ UDP 197
◊ UPD 206
◊ групповой адрес 382
◊ информация о протоколе 290
◊ канальный уровень 192, 193, 194
◊ надежный 196
◊ номер порта 198
◊ определение 192
◊ потоковый 201
◊ разделение пакетов 235, 237
◊ сетевой уровень 192, 194, 195
◊ склеивание пакетов 201, 237
◊ соединение 198, 200, 296
◊ транспортный уровень 192, 197
◊ уровень представлений 193, 197
◊ уровень приложений 197
◊ уровень сессии 193, 197
◊ физический уровень 192, 193
◊ хорошо известные порты 199
Сетевой формат числа 191, 209
Сетевой экран 203
Символ:
◊ нетерминальный 492
◊ терминальный 492
Синтаксис 490
◊ вещественного числа 495
Синтаксический анализ 490
◊ левосторонний 498
◊ нисходящий 498
◊ рекурсивный 498
Синтаксический граф 498
Событие (Event) 153, 325
Сокеты:
◊ SPI (Service Provider Interface) 288
◊ адрес сокета 190, 206, 216, 384
◊ асинхронный режим 304, 325, 387
◊ блокирование 216, 217, 234, 252, 255, 358
◊ буферы сокета 191, 198, 202, 216, 231, 283, 306, 307
◊ готовность сокета 254
◊ качество обслуживания (QoS) 288
◊ многоадресная рассылка 382
◊ неблокирующий режим 255, 268, 305, 329, 358
◊ определение понятия "сокет" 190
◊ параметры сокета 282
◊ протокол сокета 190, 206
◊ соединение 218, 230
◊ создание 205
◊ сокеты Windows 188
◊ сокеты Беркли 188
◊ стандартные сокеты 188
◊ широковещательные сообщения 199, 284
Состояние ожидания нити 366

Стек протоколов 192, 193
◊ TCP/IP 193, 206
Строки:
◊ счетчик ссылок 85, 428
◊ копирование при необходимости 85, 428
◊ макросы TEXT и _T 21
◊ строки, завершающиеся нулем 86, 129, 427
◊ тип AnsiString 85, 427
◊ тип PChar 86, 427
◊ тип ShortString 425
◊ тип WideString 85, 460
◊ типы TCHAR, LPTSTR, LPCTSTR 21, 83, 86, 89

Т

Табличный анализатор 537
Типы VCL:
◊ TColor 78
◊ TCreateParams 48, 65
◊ TMessage 62

У

Утверждение 490

Ф

Файл, ассоциация 145
Форма Бэкуса — Наура (БНФ) 492, 503
◊ расширенная (РБНФ) 492
Фрейм (Frame) 486
Фундаментальные типы 399
Функция обратного вызова 32, 89

Ц

Целые числа:
◊ беззнаковые 399
◊ знаковые 399
◊ знаковый бит 399
◊ переполнение 403

Я

Язык 489