Михаил Фленов

ПРОГРАММИРОВАНИЕ В Delphi ГЛАЗАМИ ХАКЕРА

2-е издание

Санкт-Петербург
«БХВ-Петербург»
2007

УДК 681.3.068
ББК 32.973.26-018.1
Ф69

Фленов М. Е.

Ф69 Программирование в Delphi глазами хакера. — 2-е изд., перераб. и доп.— СПб.: БХВ-Петербург, 2007. — 480 с.: ил. + CD-ROM

ISBN 978-5-9775-0081-4

Рассмотрены нестандартные приемы программирования, а также примеры использования недокументированных функций и возможностей языка Delphi в ОС Windows при разработке шуточных программ и серьезных сетевых приложений для диагностики сетей, управления различными сетевыми устройствами и просто при повседневном использовании интернет-приложений.

Компакт-диск содержит исходные коды примеров и откомпилированные программы, а также популярные приложения компании CyD Software Labs.

Для программистов

УДК 681.3.068
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Инны Тачиной*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.09.07.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 38,7.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

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

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

ISBN 978-5-9775-0081-4

Оглавление

Введение **1**

Замечания ко второму изданию 1
Благодарности 1
О книге 2
Кто такой хакер и как им стать? 6
Обратная связь 14

Глава 1. Минимизация, скорость и безопасность **15**

1.1. Методы минимизации 15
1.2. Сжатие исполняемых файлов 17
1.3. Без окон, без дверей... 21
1.4. Шаблон минимального приложения 27
1.5. Создание элементов управления с помощью Windows API 34
1.6. Прячем программы 36
1.7. Использование библиотек KOL и MCK 37
1.8. Динамические библиотеки 40
1.9. Оптимизация программ 43
Закон № 1 44
Закон № 2 45
Закон № 3 47
Закон № 4 49
Закон № 5 50
Закон № 6 51
Закон № 7 52
Закон № 8 53
Итог 53
1.10. Приемы оптимизации 54
1.10.1. Быстрый старт 54
1.10.2. Динамические библиотеки 56
1.10.3. Цепочки 59
1.11. Ассемблер и Delphi 62
1.11.1. Использование встроенного ассемблера 63
1.11.2. Внешний ассемблер 66
1.11.3. Встроенный оптимизатор 70
1.12. Безопасный код 71

Глава 2. Простые шутки **75**

2.1. Летающая кнопка *Пуск* 76
2.2. Полный контроль над кнопкой *Пуск* 81
2.3. Панель задач 87

2.4. Настольные розыгрыши 90
2.5. Контролируем системную палитру 92
2.6. Изменение разрешения экрана 95
2.7. Маленькие шутки 101
2.7.1. Как программно "потушить" монитор? 101
2.7.2. Запуск системных cpl-файлов 102
2.7.3. Программное управление устройством для чтения компакт-дисков 102
2.7.4. Отключение сочетания клавиш <Ctrl>+<Alt>+<Del> 103
2.7.5. Отключение сочетания клавиш <Alt>+<Tab> 104
2.7.6. Удаление часов с панели задач 104
2.7.7. Исчезновение чужого окна 105
2.7.8. "Клеим" обои 105
2.7.9. Запрет кнопки *Закрыть* в заголовке окна 107
2.7.10. Окно, которое нельзя закрыть 107
2.7.11. Спрятать окно 108
2.7.12. Закрыть чужое окно 108
2.8. Шутки с мышью 109
2.8.1. Безумная мышь 109
2.8.2. Мышеловка 110
2.8.3. Изменчивый указатель 111
2.8.4. Как щелкнуть в нужном месте? 112
2.9. Работа с чужими окнами 114
2.10. Дрожь в ногах 119
2.11. Оконные страсти 121
2.12. Буфер обмена 123
2.13. Служба сообщений 128

Глава 3. Система 133

3.1. Подсматриваем пароли, спрятанные под звездочками 133
3.2. Мониторинг исполняемых файлов 140
3.3. Переключающиеся экраны 150
3.4. Безбашенные окна 155
3.5. Права доступа к объектам 165
3.5.1. Дескриптор безопасности 165
3.5.2. Дескриптор безопасности 172
3.5.3. Редактирование прав доступа 182
3.6. Сервисы 189
3.7. Управление менеджером сервисов 193
3.8. Оснастка сервисов 201
3.9. Управление пользователями 208
3.9.1. Получение списка пользователей 209
3.9.2. Управление пользователями и группами 218
3.10. Изменение параметров окна 220
3.11. Создание ярлыков 222
3.12. Управление ярлыками 229
3.13. Прозрачность окон 232

Глава 4. Работа с сетью 237

4.1. Немного теории 237
4.1.1. Сетевые протоколы: протокол IP 240
4.1.2. Сопоставление адресов ARP и RARP 241
4.1.3. Транспортные протоколы: быстрый UDP 242
4.1.4. Транспортные протоколы: медленный, но надежный TCP 243
4.1.5. Прикладные протоколы: загадочный NetBIOS 244
4.1.6. NetBEUI 246
4.1.7. Сокеты Windows 246
4.1.8. Протокол IPX/SPX 246
4.1.9. Порты IP 247
4.2. Их разыскивают бойцы 139-го порта 248
4.3. Чат для локальной сети 252
4.3.1. UDP в Delphi 7 255
4.3.2. UDP в Delphi 2006 257
4.4. Сканирование открытых ресурсов 258
4.5. Telnet-клиент 263

Глава 5. Сеть на низком уровне 271

5.1. Инициализация WinSock 271
5.1.1. Пример инициализации 273
5.1.2. Подключение заголовочных файлов 273
5.1.3. Получение информации о сокетах 276
5.2. Обработка сетевых ошибок 277
5.3. Функции соединения с сервером 279
5.3.1. Синхронность/асинхронность работы порта 280
5.3.2. Соединение с сервером 281
5.3.3. Порты 282
5.3.4. Закрытие соединения 283
5.4. Сканер портов 283
5.5. Самый быстрый сканер портов 287
5.5.1. Работа с событиями 289
5.5.2. Время и количество 290
5.5.3. Кодинг 291
5.5.4. Структура типа *TFDSet* 298
5.6. Продолжаем знакомиться с WinSock 304
5.7. Определение локального/удаленного IP-адреса 307
5.8. Пишем TCP/IP-сервер и клиента 309
5.8.1. TCP-сервер 310
5.8.2. TCP-клиент 313
5.9. Передача данных 315
5.9.1. Блокирующий режим 315
5.9.2. Блокирующий TCP-сервер 316
5.9.3. Неблокирующий сокет 319
5.9.4. Обмен через сообщения 324
5.9.5. Пример работы через сообщения 328

5.10. Как написать троян 330
5.11. Работа с UDP 331
5.11.1. UDP-сервер 332
5.11.2. UDP-клиент 334
5.11.3. Замечания 335
5.12. HTTP-клиент 336

Глава 6. Железная мастерская 343

6.1. Общая информация о компьютере и ОС 343
6.1.1. Платформа компьютера 347
6.1.2. Информация о процессоре 347
6.1.3. Информация о платформе Windows 347
6.1.4. Дополнительная информация о Windows 348
6.1.5. Переменные окружения Windows 348
6.2. Информация о памяти 350
6.3. Информация о дисках 353
6.4. Частота и загрузка процессора 357
6.4.1. Частота процессора 358
6.4.2. Загрузка процессора 362
6.5. Работа с COM-портом 364
6.6. Работа с LPT-портом 369
6.7. Получение информации об устройстве вывода 374
6.8. Работа с типами файлов 378
6.8.1. Получение информации о типе файлов 378
6.8.2. Связывание своей программы с неопределенным типом файлов 383
6.9. Работа со сканером 386
6.10. IP-config собственными руками 392
6.11. Получение информации о сетевом устройстве 397

Глава 7. Полезное 405

7.1. Работа с NetBIOS 405
7.2. Работа с ARP 411
7.3. Изменение записей ARP-таблицы 419
7.3.1. Добавление ARP-записей 419
7.3.2. Удаление ARP-записей 425
7.4. Работа с сетевыми ресурсами 430
7.5. Быстрая проверка состояний портов: вариант 1 441
7.6. Быстрая проверка состояний портов: вариант 2 451
7.7. Работа с ICMP на примере *ping* 461
7.8. Trace Route 468

Приложение. Описание компакт-диска 471

Литература 472

Предметный указатель 473

Введение

Замечания ко второму изданию

Это второе издание книги, но, в отличие от других книг, у меня второе издание отличается от первого как собака от кошки. Я переписал больше 80% книги, и не только исправил ошибки, но и добавил просто непристойное количество нового материала. Устаревшую информацию, которая может быть использована только в Windows 95, я убрал из книги вовсе. Те темы, которые могут быть еще интересными, я перенес в виде статей в PDF-формате на компакт-диск.

Компакт-диск заслуживает отдельного внимания. В данном издании я постарался сделать его максимально интересным. В каталоге Документация вы найдете множество интересных документов и статей. Поэтому если вы читаете эту страницу еще в магазине, то обязательно требуйте компакт-диск. Его содержимое вас приятно удивит. Наверно, никто еще не давал такого количества дополнительной документации своим читателям.

Благодарности

Первым делом хочется поблагодарить своих родителей за то, что они произвели меня на свет. Если бы не они, не было бы этой книги, по крайней мере, в моем исполнении. Свою семью, жену, детей я просто обязан отблагодарить за их терпение к моей работе и за частые исчезновения в виртуальной реальности.

Давыдова Дениса — главного редактора "Игромании". Когда-то он был редактором игровой части в журнале "Хакер", и мне повезло, что я начинал свою писательскую деятельность в журнале именно с ним. Журнал тогда только начинал свое развитие, и Денис уделял мне достаточно много времени. Я программист и выжать из себя пару литературных строк в то время вообще не мог (да и сейчас я в этом деле не гений, у меня в школе по русскому и литературе всегда было 3 с большим минусом), а Денис мне дал пинок, от которого я, можно сказать, начал свою карьеру.

Покровского Сергея aka SINtez — главного редактора журнала "Хакер". После того как Давыдов Денис ушел из журнала, я какое-то время стал меньше публиковаться, но потом меня, можно так сказать, взял под крыло Сергей.

Сначала я стал вести рубрику "Hack-Faq", а потом и "Кодинг", которая очень быстро развилась и получила популярность.

Хочется поблагодарить всех друзей, которые меня поддерживают и помогают мне в моей работе и в жизни. Я могу еще очень долго перечислять людей, которым я благодарен в своей жизни, поэтому перечислю коротко: воспитателям в детсаде, учителям в школе, преподавателям в институте, всей команде журнала "Хакер" и всем моим друзьям, за то, что они меня терпят, вероятно, любят, и по возможности помогают. Ну и самое главное, я хочу поблагодарить всех моих читателей за то, что они читают мои опусы.

Отдельная благодарность редакторам, которые делают мои книги лучше, находят и помогают исправить мои ошибки, опечатки и недочеты, редакции издательства "БХВ-Петербург" за длительное и плодотворное сотрудничество.

О книге

Как вы, наверное, уже поняли, я помешанный на программировании человек и в течение всей книги не буду блистать литературным стилем. Зато я постараюсь поделиться своими знаниями и надеюсь рассказать вам что-то новое.

Когда вышел в свет первый вариант моей работы о Delphi глазами хакера, то книга вызвала бурю эмоций, как положительных, так и отрицательных. Я прекрасно понимаю, что такая книга мало кого оставит равнодушным, но надеюсь, что положительных эмоций у вас останется намного больше.

В книге я расскажу вам о программировании для хакера. Я буду достаточно часто использовать один термин — "кодинг". Что это такое? Под этим словом мы будем, как и все, подразумевать слово "программирование". А вот под словом "хакер" лично я подразумеваю немного иной смысл, чем другие. Я считаю, что хакер — это профессионал в компьютерной сфере, но не обязательно доставляющий много неприятностей другим людям своими знаниями. Так вот в этой книге я постарался показать много интересных вещей с точки зрения сетевого программиста-профессионала, а не взломщика. Более подробно о понятии "хакер" вы найдете в следующем разделе.

Хакер — понятие обширное, и затронуть все стороны просто нереально. Я специализируюсь на сетях, их безопасности, Web-программировании, Web-безопасности и шутках над системой. Драйверами и нулевым кольцом я занимаюсь не очень серьезно. Да, я знаю основы популярных ошибок, я знаю основные проблемы, но не собираюсь посвящать этим неприятностям свою жизнь.

В данной работе я попробую привести как можно больше нестандартных приемов программирования, недокументированные функции и возможности, а главное — продемонстрирую вам приемы работы с сетью в операционной системе Windows.

В книге приведено максимальное количество примеров на языке программирования Delphi. Для этого я написал множество шуточных программ и сетевых приложений. Чистой теории будет мало, зато практических занятий — хоть отбавляй. Практические примеры позволят вам лучше понять тему, потому что лучше один раз увидеть результат собственными глазами, чем сто раз услышать или прочитать. Есть еще один плюс у большого количества кода — вы можете использовать мои заготовки в ваших собственных приложениях. Да, код шуточных программ особо нигде не пригодится, а вот примерам работы с сетью и написанным мною функциям вы сможете найти применение.

Для понимания книги вам понадобятся хотя бы начальные знания о среде Delphi и сносное умение общаться с компьютером и мышью. Что касается сетевого программирования, то его я опишу полностью, начиная от основ и заканчивая сложными примерами. Так что тут начальные знания желательны, но не обязательны. Если вы начинающий программист, то могу посоветовать для получения основ прочитать мою книгу "Библия Delphi"[1] и посетить мой сайт **www.vr-online.ru**, где выложено достаточно много полезной информации.

Если вы ожидали увидеть в данной книге примеры и описания вирусов, то вы сильно ошиблись. Ничего разрушительного я делать и рассказывать не буду. Я занимаюсь созиданием, а не разрушением. Чего и вам советую. Меня больше интересует безопасность, а не взлом. Да, я знаю, как взламываются программы и сайты, потому что все, кто хочет защищаться, должен знать, как вас могут атаковать.

Для эффективной работы с книгой вам понадобятся хотя бы начальные знания Delphi. Вы должны уметь создавать простое приложение, знать, что такое циклы и как с ними работать. Не помешают знание адресации, указателей, и для чего они нужны.

Эта книга построена не так как многие другие. В ней нет длинных и нудных теоретических рассуждений, а только максимум примеров и исходного кода. Ее можно воспринимать как практическое руководство к действию. Чтобы книга не была занудной, я постарался описывать все простым и неформальным языком и в некоторых случаях даже в веселой форме.

Я постарался облегчить вам задачу, описав все как можно проще. Большинство кода расписано очень подробно, и в тексте программ вы найдете мак-

[1] Фленов М. Библия Delphi. — СПб.: БХВ-Петербург, 2004.

симум комментариев, которые помогут получать наслаждение от чтения кода вместо обычной головной боли. Некоторые меня критикуют за то, что я уж слишком сильно все разжевываю, и остается только проглотить, но мне кажется, что во время чтения книги, вы не должны слишком сильно отвлекаться на изучение кода. Таким образом, изучать код можно будет даже в метро, без наличия под рукой компьютера. Но для закрепления материала, я бы рекомендовал все же писать вам код самостоятельно и пробовать воссоздать мои примеры без книги или хотя бы без обращения к исходным кодам на компакт-диске. Это позволит еще лучше усвоить материал и глубже понять проблему. Пробуйте, экспериментируйте и читайте.

Программисты в чем-то похожи на врачей: если врач теоретически знает симптомы болезни, но на практике не может точно различить отравление от аппендицита, то такого врача лучше не подпускать к больному. Точно так же и программист, если он знает, как работает протокол, но не может с ним работать, то его сетевые программы никогда не будут работать правильно.

Это сравнение приведено здесь не просто так. В 2002 году я попал в больницу с температурой и болями в области живота. Меня положили в хирургическое отделение и хотели вырезать аппендицит. Я пролежал три дня, и ни один врач не решался меня отправить на операцию, но в то же время никто не знал, откуда у меня боли, и почему температура под вечер поднимается до 39 градусов.

На третий день вечером я сбежал из больницы, потому что у моей мамы был день рождения. На нем присутствовал знакомый врач (по специализации акушер), который, осмотрев меня, сказал пить по 1 таблетке через каждые 12 часов (не будем уточнять, что это был за препарат, но это от желудка) и выписываться из больницы. Может кто-то не поверит, но после первой таблетки температура упала, а после второй я вообще плясал, как Борис Моисеев. Результат: врачи перепутали отравление с аппендицитом и чуть не лишили меня моего аппендикса. А ведь могли же вырезать по ошибке или просто ради интереса.

Уже после выхода первой версии Delphi глазами хакера, в начале 2006 года меня жена снова отправляет в больницу с температурой и болями в животе. В этот день по скорой дежурила другая больница, и в ней снова решили, что у меня аппендицит. Здесь врачи не стали рисковать, а сразу же отправили меня в общий наркоз видеть сны, и среди ночи сделали мне в пузе две дырки, чтобы обследовать внутренности. Конечно же, аппендикс был в порядке, и ничего не удалив, меня зашили. Кстати, дело было в субботу и оперировали меня дежурные врачи, и, конечно же, за деньги, несмотря на нашу бесплатную медицину. В понедельник, когда на работу вышли все врачи, на обходе главный хирург выписал меня, чтобы я с температурой никого не заразил.

Вот так вот во второй раз я заплатил деньги за то, что мне сделали две дырки, ничего не нашли, и выкинули через 36 часов после операции, когда я только начал понемногу ходить. Могли бы хоть перевести в другое отделение, и найти, что же со мной было.

Этот случай еще больше закрепил мое отношение к практике. Ничто не может привести к такому пониманию предмета, как хорошее практическое занятие, потому что когда вы можете ощутить все своими руками, то никакая теория не нужна.

Еще один пример из компьютерной жизни. В 2000 году я проходил обучение в МГТУ им. Баумана на нескольких курсах Microsoft SQL Server. Курсы были очень хорошие, и преподаватель старался все очень подробно и легко преподнести. Но сам курс был поставлен корпорацией как теоретический с небольшим добавлением лабораторных работ. В результате, нам очень хорошо объяснили, ЧТО может делать SQL Server. Но когда после курса я столкнулся с реальной проблемой, то понял, что я не знаю КАК сделать что-либо. Приходилось снова открывать книгу, которую выдали в центре обучения (она была предоставлена Microsoft и, конечно же, на английском языке), и, читая обширную теорию и маленькие практические примеры, разбираться с реальной задачей. Уж лучше бы я узнал на курсах, как практически выполнять примеры из жизни, а не что можно теоретически выполнить, потому что такое обучение, по-моему, — только пустая трата времени и денег.

Несмотря на это, я не противник теории и не пытаюсь сказать, что теория не нужна. Просто нужно описывать, КАК решить задачу, и рассказывать, ЗАЧЕМ мы делаем какие-то определенные действия. После этого, когда вы будете сталкиваться с подобными проблемами, вы уже будете знать, как сделать что-то, чтобы добиться определенного результата.

Именно практических руководств и просто хороших книг с разносторонними примерами не хватает на полках наших магазинов. Я надеюсь, что моя работа восполнит хотя бы небольшой пробел в этой сфере и поможет вам в последующей работе и решении различных задач программирования.

Все примеры, которые описаны в книге, можно найти на компакт-диске в каталоге Примеры. Несмотря на это, я советую все описанное создавать самостоятельно и обращаться к готовым файлам, только если возникли какие-то проблемы, чтобы сравнить и найти ошибку. Любая информация после практической работы откладывается в памяти намного лучше, чем прочтенные сто страниц теории.

Даже если вы решили воспользоваться готовым примером, обязательно досконально разберитесь с ним. Попробуйте изменить какие-то параметры и посмотреть на результат. Можете попытаться улучшить программы, добавив

какие-то возможности. Только так вы сможете понять принцип работы используемых функций или алгоритмов.

Помимо этого, на компакт-диске можно найти следующие каталоги:

- Headers — здесь находятся все необходимые заголовочные файлы, которые нужно будет подключать к Delphi для компиляции некоторых примеров;
- Source — здесь я выложил исходные коды своих простых программ, чтобы вы могли ознакомиться с реальными приложениями. Их немного, но посмотреть стоит;
- Документация — дополнительная документация, которая может понадобиться для понимания некоторых глав. Документации достаточно много, поэтому будет что почитать на досуге. Если вы скачали книгу из Интернета, и содержимого диска нет, то могу только посочувствовать;
- Программы — программы, которые пригодятся при программировании. Среди них Header Convert — программа, которая конвертирует заголовочные файлы с языка C на Delphi, и ASPack — программа сжатия исполняемых файлов.

Кто такой хакер и как им стать?

Слово "хакер" в названии книги — это не дань моде и не рекламный ход. Это стиль жизни, но чтобы понять, что я вкладываю в это понятие, следует узнать, кто такой хакер. Многие вкладывают в слово "хакер" совершенно другой смысл, поэтому могут воспринять данную работу неправильно.

Прежде чем приступить к практике, я хочу "загрузить" вашу голову небольшой теорией. Нет, теорией не по программированию, а просто познавательной информацией. А именно, прежде чем читать книгу, вы обязаны знать, кто такой хакер в моем понимании. Если вы будете подразумевать одно, а я другое, то мы не сможем понять друг друга.

В мире полно вопросов, на которые большинство людей не знают правильного ответа. Мало того, люди используют множество слов, даже не догадываясь об их настоящем значении. Например, многие сильно заблуждаются в понимании термина "хакер". Почему я уверен, что заблуждаются? Да потому что мнений много, но только одно из них может быть верным. Однако каждый вправе считать свое мнение наиболее правильным. И я не буду утверждать, что именно мое мнение единственно верное, но оно отталкивается от действительно корректного и правильного понятия, с которого все начиналось.

Но для начала, посмотрим на перевод этого слова, который дает нам ABBYY Lingvo, и оказывается, что один из вариантов — дилетант, непрофессионал (рис. В1). Оригинально, не правда ли? И как это слово связано с компьютерным взломщиком?

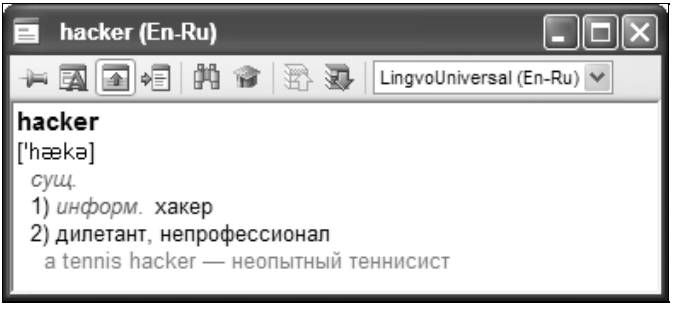


Рис. В1. Перевод слова "hacker" в ABBYY Lingvo

А теперь посмотрим на перевод слова "cracker" (рис. В2). Это слово также очень часто связывают со взломщиками, и оно уже лучше отражает суть (дробилка, босяк), а иногда и слова хвастун или лжец тоже верно отражают суть.

Прежде чем начать обсуждать всем известный термин, я должен углубиться в историю и вспомнить, как все начиналось. А начиналось все еще тогда, когда не было даже международной сети Интернет.

Понятие "хакер" зародилось, когда только начинала распространяться первая сеть ARPAnet. Тогда это понятие обозначало человека, хорошо разбирающегося в компьютерах. Некоторые даже подразумевали под хакером человека, помешанного на компьютерах. Понятие ассоциировали со свободным компьютерщиком, человеком, стремящимся к свободе во всем, что касалось его любимой "игрушки". Именно благодаря этому стремлению к свободному обмену информацией и началось такое бурное развитие Всемирной сети. Именно хакеры помогли развитию Интернета и создали FIDO. Благодаря им, появилась UNIX — система с открытым исходным кодом, на котором сейчас работает большое количество серверов.

В те времена еще не было вирусов, и не внедрилась практика взломов сетей или отдельных компьютеров. Образ хакера-взломщика появился немного позже. Но это только образ. Настоящие хакеры никогда не имели никакого отношения к взломам, а если хакер направлял свои действия на разрушение, то это резко не приветствовалось виртуальным сообществом.

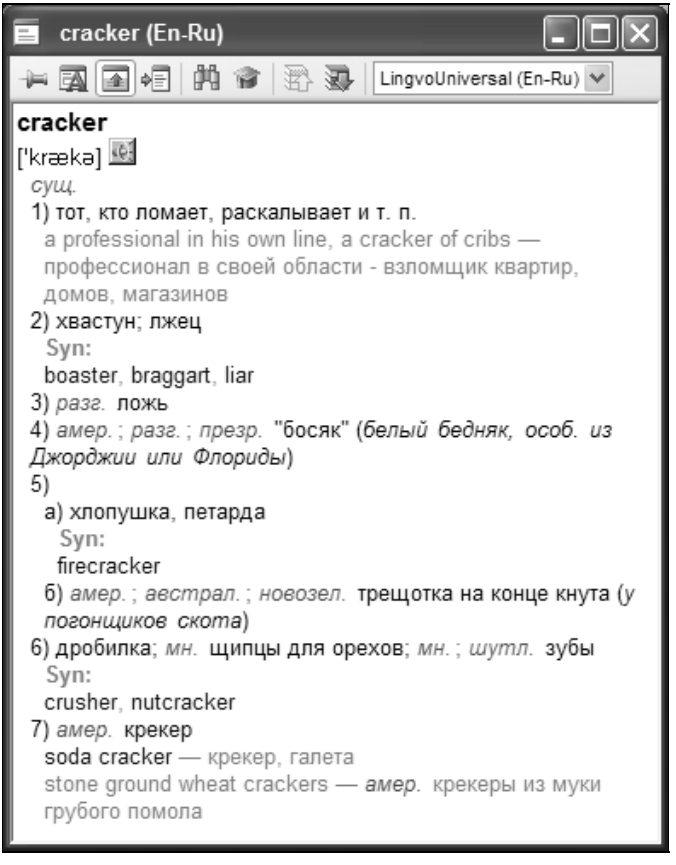


Рис. В2. Перевод слова "cracker" в ABBYY Lingvo

Настоящий хакер — это творец, а не разрушитель. Так как творцов оказалось больше, чем разрушителей, то истинные хакеры выделили тех, кто занимается взломом, как отдельную группу и назвали их крекерами (взломщиками) или просто вандалами. И хакеры, и взломщики являются гениями виртуального мира. И те и другие борются за свободу доступа к информации. Но только крекеры взламывают сайты, закрытые базы данных и другие источники информации. Если крекер совершает взлом исключительно ради свободы информации, то это еще можно простить, но если это делается ради собственной наживы, ради денег или минутной славы, то такого человека можно назвать только преступником (кем он по закону и является!).

Существует еще одно деление, когда хакером называют интернет-взломщика, а крекером — взломщиков программ. При этом White Hat (белая

шапка) хакеры являются "добрыми", т. е. взламывают сети только для изучения и улучшения нашей жизни, а Black Hat (черная шапка) — это все те же вандалы. Только журналистам в основном все равно, кто перед ним — белая или черная шапка. Они готовы любого назвать взломщиком и хакером, лишь бы раздуть сенсацию и получить за это деньги. Именно поэтому я склоняюсь к классическому делению на хакеров, без каких-либо шапок и крекеров. Это уже разные слова, и тут что-то спутать сложнее, а значит, меньше путаницы с понятиями.

Как видите, хакер — это просто человек, который в чем-то хорошо разбирается, который живет знаниями и безопасностью. Все, кто приписывает этим людям вандализм, сильно ошибаются. Истинные хакеры никогда не используют свои знания во вред другим.

Теперь давайте разберемся, как стать настоящим хакером. Это обсуждение поможет вам больше узнать об этих людях.

- Вы должны знать свой компьютер и научиться эффективно им управлять. Если вы будете еще и знать в нем каждую железку, то это только добавит к вашей оценке по "хакерству" большой жирный плюс.

  Что я подразумеваю под умением эффективно управлять своим компьютером? Это значит знать все возможные способы каждого действия и в каждой ситуации уметь использовать наиболее оптимальный. В частности, вы должны научиться пользоваться "горячими" клавишами, и не дергать мышь по любому пустяку. Нажатие клавиш выполняется быстрее, чем любое, даже маленькое перемещение мыши. Просто приучите себя к этому, и вы увидите все прелести работы с клавиатурой. Лично я использую мышь очень редко и стараюсь всегда применять клавиатуру.

  Маленький пример на эту тему. Мой начальник всегда копирует и вставляет из буфера с помощью кнопок на панели инструментов или команд контекстного меню, которое появляется при щелчке правой кнопкой мыши. Но если вы делаете также, то наверно знаете, что не везде есть кнопки **Копировать**, **Вставить** или такие же пункты в контекстном меню. В таких случаях мой начальник набирает текст вручную. А ведь можно было бы воспользоваться копированием/вставкой с помощью "горячих" клавиш <Ctrl>+<C>/<Ctrl>+<V> или <Ctrl>+<Ins>/<Shift>+<Ins>, которые достаточно универсальны и присутствуют практически во всех современных приложениях.

- Вы должны досконально изучать все, что вам интересно о компьютерах. Если вас интересует графика, то вы должны изучить лучшие графические пакеты, научиться рисовать в них любые сцены и создавать самые сложные миры. Если вас интересуют сети, то старайтесь узнать о них все. Если вы считаете, что уже знаете все, то купите книгу по данной теме по-

толще и вы поймете, что сильно ошибались. Компьютеры — это такая вещь, в которой невозможно знать все!!!

Хакеры — это, прежде всего, профессионалы в каком-нибудь деле. И тут даже не обязательно должен быть компьютер. Хакером можно стать в любой области, но я буду рассматривать только компьютерных хакеров.

- Желательно уметь программировать. Любой хакер должен знать, как минимум, один язык программирования. А лучше знать даже несколько языков. Лично я рекомендую всем изучить для начала Delphi. Он достаточно прост, быстр, эффективен, а главное — это очень мощный язык. Но сие не значит, что не надо знать другие языки. Вы можете научиться программировать на чем угодно, даже на языке Basic (хотя использовать его не советую, но знать не помешало бы).

  Хотя я не очень люблю Visual Basic за его ограниченность, неудобность и сплошные недостатки, я видел несколько великолепных программ, который были написаны именно на этом языке. Глядя на них, сразу хочется назвать их автора Хакером, потому что это действительно виртуозная и безупречная работа. Создание из ничего чего-то великолепного, как раз и есть искусство хакерства.

  Хакер — это созидатель, человек, который что-то создает. В большинстве случаев это относится к коду, но можно создавать и графику, и музыку. Все это тоже относится к искусству хакера. Но даже если вы занимаетесь компьютерной музыкой, знания программирования повысят ваш уровень. Сейчас создавать свои программы стало уже не так сложно, как раньше. С помощью Delphi можно создавать простенькие утилиты за очень короткое время. Так что не поленитесь и изучите программирование. А я на протяжении всей книги буду показывать то, что необходимо знать программисту-хакеру, и покажу множество интересных приемов и примеров.

- Не тормози прогресс. Хакеры всегда боролись за свободу информации. Если вы хотите быть хакером, то тоже должны помогать другим. Хакеры обязаны способствовать прогрессу. Некоторые делают это через написание программ с открытым кодом, а кто-то просто делится своими знаниями.

  Открытая информация — не значит, что вы не можете зарабатывать деньги. Это никогда не возбранялось, потому что хакеры тоже люди и тоже хотят кушать и должны содержать свою семью. Но деньги не должны быть главным в вашей жизни. Самое главное — это созидание, процесс создания. Вот тут проявляется еще одно отличие хакеров от крекеров — хакеры "создают", а крекеры "уничтожают" информацию. Если вы написали какую-нибудь уникальную шуточную программу, то это вас делает хакером. Но если вы написали вирус, который уничтожает диск, то это вас делает крекером, а я бы даже сказал "преступником".

В борьбе за свободу информации может применяться даже взлом, но только не в разрушительных целях. Вы можете взломать какую-нибудь программу, чтобы посмотреть, как она работает, но не убирать с нее систем защиты. Нужно уважать труд других программистов, потому что защита программ — это их хлеб. Если в программах не будет защиты, то программисту не на что будет есть.

Представьте себе ситуацию, если бы вы украли телевизор. Это было бы воровство и преследовалось бы по закону. Многие люди это понимают и не идут на преступления из-за боязни наказания. Почему же тогда крекеры спокойно ломают программы, не боясь закона? Ведь это тоже воровство. Лично я приравниваю взлом программы к воровству телевизора с полки магазина и считаю это одним и тем же.

- Не изобретай велосипед. Тут опять действует созидательная функция хакеров. Они не должны стоять на месте и обязаны делиться своими знаниями. Например, вы написали какой-то уникальный код, поделитесь им с другими, чтобы людям не пришлось создавать то же самое. Вы можете не выдавать все секреты, но должны помогать другим.

  Ну а если вам попал код другого человека, то не стесняйтесь его использовать (с его согласия!). Не выдумывайте то, что уже сделано другими и обкатано пользователями. Если каждый будет создавать колесо, то никто и никогда не создаст повозку.

- Хакеры — не просто отдельные личности, а целая культура. Но это не значит, что все хакеры одеваются одинаково и выглядят как китайцы — все на одно лицо. Каждый из них — это отдельный индивидуум и не похож на других. Не надо копировать другого человека. Тем, что вы удачно скопируете кого-то, вы не станете продвинутым хакером. Только ваша индивидуальность может сделать вам имя.

  Если вас знают в каких-то кругах, то это считается очень почетным. Хакеры — это люди, добывающие себе славу своими знаниями и добрыми делами. Поэтому любого хакера должны знать.

- Ну и самое главное — нужно учиться, читать, пробовать и тестировать. Лично я себя хакером не считаю, хотя у меня достаточно большой опыт в программировании и в безопасности. Мне приходилось взламывать программы и сайты, но глядя на всю ту информацию, которую еще необходимо впитать, изучить и проверить, так становится понятно, что мои знания по сравнению с тем, что можно знать — это капля в море. Поэтому я продолжаю все время что-то изучать.

  Но даже если на секунду представить, что вы смогли изучить абсолютно все, что вас интересовало, то ощущение полноты будет недолгим. ИТ-сфера развивается очень быстро, и через год знания могут очень сильно

устареть. Знания постоянно нужно обновлять, следить за изменениями и изучать все новое.

Как вам узнать, являетесь ли вы хакером или нет? Очень просто, если о вас говорят, как о хакере, то вы он и есть. Да, некоторые непосвященные в компьютерах иногда называют хакерами своих друзей, которые умеют банально установить Windows, или что-то поменять в системном блоке, например заменить плату.

Жаль, что реального признания добиться очень сложно, потому что большинство считает хакерами взломщиков. Поэтому, чтобы о вас заговорили, как о хакере, нужно что-то взломать. Но это неправильно, и не надо поддаваться на этот соблазн. Старайтесь держать себя в рамках и добиваться славы только добрыми делами. Это намного сложнее, но что поделаешь... Никто не говорил, что будет просто.

В нашей стране получить признание еще сложнее. Может быть это наш менталитет, а может недостаток воспитания. Когда кто-то становится популярным, то его стараются опозорить или даже унизить. Всеми уважаемый хакер — Кевин Митник для своего времени был настоящим гением. Да, его знания немного устарели за время пребывания в тюрьме, но старые заслуги от этого не становятся пустышкой. В нашей же стране большинство считает его просто ламером. А жаль, ведь это простое неуважение к другим и показ собственного плохого воспитания.

Некоторые считают, что правильно надо произносить "хэкер", а не "хакер". Это так, но только для английского языка. У нас в стране оно обрусело и стало "хакером". Мы русские люди, и давайте будем любить свой язык и признавать его желания.

Напоследок советую почитать статью "Как стать хакером" на сайте **www.sekachev.ru**. Эта статья написана знаменитым человеком в ИТ-области по имени Eric S. Raymond. С некоторыми его взглядами я не согласен, но в большинстве своем она также отражает дух хакерства.

Тут же возникает вопрос: "Почему же автор относит к хакерскому искусству написание шуточных и сетевых программ?" Попробую ответить на этот вопрос. Во-первых, хакеры всегда пытались доказать свою силу и знания методом написания каких-либо интересных, веселых программ. В эту категорию я не отношу вирусы, потому что они несут в себе разрушение, хотя они тоже бывают с изюминкой и юмором. Зато простые и безобидные шутки всегда ценились в узких кругах. Этим хакер показывает не только свои знания особенностей операционной системы, но и старается заставить ближнего своего улыбнуться. Не секрет, что многие хакеры обладают хорошим чувством юмора, и он поневоле ищет своего воплощения. Я советую шутить с помощью безобидных программ.

Ну а сетевое программирование неразделимо с понятием "хакер" с самого его рождения. Хакеры получили распространение благодаря сети, понимания ее функционирования и большому вкладу в развитие Интернета.

Ну и последнее. Я уже сказал, что любой хакер должен уметь программировать на каком-нибудь языке программирования. Некоторые заведомо считают, что если человек — хакер, то он должен знать и уметь программировать на языке ассемблера. Это не так. Знания ассемблера желательно, но не обязательно. Я люблю Delphi, и он позволяет мне сделать все, что я захочу. А главное, что я могу сделать это быстро и качественно.

Я по образованию экономист-менеджер и 6 лет проучился в институте по этой специальности. Но даже до этого я знал, что заказчик всегда прав. Почему-то в компьютерной области стараются избавиться от этого понятия. Например, Microsoft делает упор на программистов, пытаясь воспитывать их писать определенные программы, не объясняя, зачем это нужно пользователям. Многие тупо следуют этим рекомендациям и не задумываются о необходимости того, что они делают.

Тут же приведу простейший пример. Сейчас все программисты вставляют в свои продукты поддержку XML, и при этом никто из них не задумывается о необходимости этого. А ведь не всем пользователям этот формат нужен и не во всех программах он востребован. Следование рекомендациям Microsoft не означает правильность действий, потому что заказчик — не Билл Гейтс, а ваш потребитель. Поэтому надо всегда делать то, что требует конечный пользователь.

Я вообще рекомендую не обращать внимания на корпорацию Microsoft, потому что считаю ее только тормозом прогресса. И это тоже можно доказать на примере. Сколько технологий доступа к данным придумала Microsoft? Просто диву даешься: DAO, RDO, ODBC, ADO, ADO.NET, и это еще не полный список. Корпорация Microsoft регулярно выкидывает на рынок что-то новое, но при этом сама этим не пользуется. При появлении новой технологии все программисты кидаются переделывать свои программы под новый стандарт и в результате тратят громадные ресурсы на постоянные переделки. Таким образом, конкуренты сильно тормозят, а Microsoft движется вперед, потому что не следует собственным рекомендациям и ничего не переделывает.

Если программа при создании использовала для доступа к данным DAO, то можно спокойно оставить ее работать через DAO и не переделывать на ADO, потому что пользователю все равно, каким образом программа получает данные из базы, главное, чтобы данные были. Несмотря на то, что уже давно появились ADO и ADO.NET, некоторые утилиты Microsoft продолжают использовать старые интерфейсы, и никто их не собирается переписывать, чтобы не тратить на это время.

Программисты и хакеры навязывают другим свое мнение о любимом языке программирования как о единственно приемлемом и делают это обычно успешно, потому что заказчик очень часто ничего не понимает в программировании. На самом же деле, заказчику все равно, на каком языке вы напишете программу, его интересуют только сроки и качество. Лично я могу обеспечить минимальные сроки написания приложения вкупе с хорошим качеством, только работая на Delphi. Такое же качество на VC++ я (да и любой другой программист) смогу обеспечить только в значительно бóльшие сроки. Ну что поделаешь, если объектное программирование проигрывает компонентному в скорости разработки. Недаром у Microsoft появилась компонентная платформа .NET и компонентный язык C#.

Вот когда заказчик требует минимальный размер или наивысшую скорость работы программы, тогда я берусь за ASM и C. Но это бывает очень редко, потому что сейчас носители информации уже практически не испытывают недостатка в размерах и компьютеры работают в миллионы раз быстрее первых своих предков. Таким образом, размер и скорость программы уже не являются критичными и на первый план ставится скорость и качество выполнения заказа.

Итак, на такой деловой ноте мы закончим вводную лекцию и перейдем к практическим упражнениям по воинскому искусству, где часто главное — скрытность и победа минимальными силами.

Обратная связь

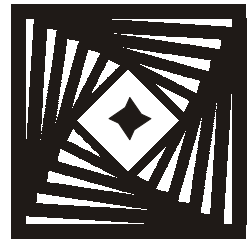
Первое издание этой книги содержало некоторые ошибки, неточности и недостатки. Эта версия тоже может содержать ошибки, потому что я полностью переработал материал. Если вы что-то нашли, то обязательно сообщите мне через мой сайт **http://www.vr-online.ru** или на e-mail **horrific@vr-online.ru**. Лучше сообщать через сайт, потому что его я посещаю каждый день.

Каждая работа может содержать ошибки, и я прекрасно это понимаю. Недавно я прочитал книгу "Совершенный код"[2], там автор признался, что после выхода первого издания он нашел около 200 ошибок и неточностей. Но, несмотря на это, книга была бестселлером, и ошибками не пытались оскорбить автора, как это делают в нашей стране. Если вы нашли ошибку, то сообщите мне о ней.

Как я уже говорил, все ошибаются, и все могут совершить ошибку. Я тоже человек, поэтому нормально отношусь к критике и всегда жду отзывов читателей.

---

[2] Макконелл С. Совершенный код. — СПб.: Питер, 2005.



Глава 1

Минимизация, скорость и безопасность

Что самое главное при написании шуточных программ? Ну, конечно же, невидимость. Программы, созданные в этой и следующих главах, будут незаметно "сидеть" в системе и выполнять нужные действия при наступлении определенного события. Это значит, что программа не должна появляться на панели задач или в списке запущенных программ в окне, выдаваемом при нажатии комбинации клавиш <Ctrl>+<Alt>+<Del>. Да, в Windows 2000/XP и других ОС есть возможность увидеть запущенные процессы, и от них спрятаться труднее, но это уже не такая большая проблема. Так что, прежде чем начать что-то писать, нужно узнать, как спрятать свое творение от чужого глаза.

Помимо этого, программы-приколы должны иметь маленький размер. Приложения, создаваемые Delphi, достаточно "весомые". Даже простейшая программа, выводящая одно окно, занимает более 200 Кбайт на диске. Если вы захотите отослать такую шутку по электронной почте, то отправка и получение письма с вашей программой отнимет лишнее время у вас и получателя. Это не очень приятно, поэтому в данной главе я познакомлю вас с тем, как можно уменьшить размер программ, создаваемых в Delphi.

1.1. Методы минимизации

Чтобы понять, какие методы есть для уменьшения размера программы, необходимо ответить на вопрос: "Из-за чего программа, созданная Delphi, получается большой?" Ответ очень прост, Delphi является объектным языком, а если точнее — компонентным. *Компоненты* — это следующий шаг в программировании. В Delphi каждый элемент выглядит как объект или компо-

нент, который обладает своими свойствами, методами и событиями. Любой объект вполне автономен, он многое уже умеет делать без ваших указаний и без необходимости в написании дополнительного кода. Это значит, что вам нужно только подключить его к своей форме, изменить нужным образом свойства, и приложение готово! И оно будет работать без какого-либо прописывания его деятельности.

Простой пример автономности объектов можно увидеть на примере какого-либо визуального компонента. Например, кнопка обладает множеством свойств: положение, текст, тип кнопки, рисунок и т. д. Вам не нужно заботиться о том, как будет отображаться текст или картинка на кнопке, все это уже реализовано в самом компоненте. Вам не надо заботиться о том, чтобы кнопка правильно визуально реагировала на действия пользователя при наведении фокуса или нажатии, об этом уже позаботились разработчики, которые создавали кнопку.

Но в объектном программировании есть и свои недостатки. В объектах реализовано большое количество действий, которые вы и пользователь сможете производить с ним. Но реально в любой программе мы используем два-три из всех этих свойств. Все остальное — для программы лишний груз, который никому не нужен, но от которого невозможно избавиться. Каждый объект — неделимое целое, из которого просто нельзя вырвать определенные свойства и методы.

А если вспомнить про такую возможность объектного программирования, как наследование, то получается, что у кнопки может быть несколько предков и весь их код, все их возможности также должны быть включены в исполняемый файл. В сложных библиотеках (а библиотека VCL, используемая в Delphi очень сложная) очень много различных объектов, и в большинстве случаев иерархия наследования в библиотеках древовидная. В этом дереве все объекты и компоненты (ветви дерева) происходят от одного объекта (ствол), что немного экономит количество исходного кода и упрощает жизнь, но сэкономить результирующий код все равно не позволяет.

Но как же тогда создать компактный код, чтобы программа занимала минимум места на винчестере и в оперативной памяти? Тут есть несколько вариантов.

- Сжимать готовые программы с помощью компрессоров. Объектный код сжимается в несколько раз, и программа, созданная с использованием VCL, может превратиться из монстра в 300 Кбайт в скромного по размерам "зверя", "весящего" всего 30—50 Кбайт. Главное преимущество состоит в том, что вы не лишаетесь возможностей объектного программирования и можете спокойно забыть про неудобства WinAPI.
- Не использовать визуальную или объектную библиотеку, которая упрощает программирование. В Delphi такой библиотекой является VCL, а в

Visual C++ — это MFC. В этом случае весь код придется набирать вручную и работать только с WinAPI, что достаточно не просто. Программа в таком случае получается очень маленькой и быстрой. Но таким образом вы лишаетесь простоты визуального программирования и можете ощутить все неудобства программирования с помощью чистого WinAPI. Небольшую утилиту написать с использованием только Windows API еще можно, но большую программу — нереально.

- Использовать визуальные библиотеки, которые не сильно влияют на размер результирующего исполняемого файла. Для Delphi есть такая библиотека — это связка MCK-KOL.
- Использовать динамическую компиляцию библиотеки. Коротко рассмотрим, как это помогает нам. В этом случае, в исполняемый файл попадают только логика работы программы и ваши собственные объекты или компоненты. Стандартная библиотека языка не попадает в исполняемый файл, а подключается динамически из dll-файлов, и должна поставляться совместно с программой или быть заранее установленной на компьютере пользователя.

В данной главе мы рассмотрим все эти методы уменьшения размера. Причем вы можете использовать даже сочетания из двух методов, например, написать программу без использования визуальной библиотеки, а потом еще и сжать ее с помощью специализированного компрессора.

1.2. Сжатие исполняемых файлов

Самый простой способ уменьшить размер приложения — использование программы для сжатия файлов. Лично я очень люблю ASPack, которую вы можете найти на компакт-диске в каталоге Программы (файл установки называется ASPack.exe). Она прекрасно сжимает исполняемые exe-файлы и динамические dll-библиотеки.

Я не буду подробно описывать процесс установки ASPack, потому что там абсолютно нет ничего сложного. Только одно нажатие на кнопке **Next**, и все готово! Теперь запустите установленную программу, и вы увидите окно, изображенное на рис. 1.1. Главное окно состоит из нескольких вкладок:

- **Open File**;
- **Compress**;
- **Options**;
- **About**;
- **Help**.

На вкладке **Open File** есть только одна кнопка — **Open**. Нажмите ее и выберите файл, который вы хотите сжать. Как только вы выберете файл, программа перейдет на вкладку **Compress** и начнет сжатие (рис. 1.2).

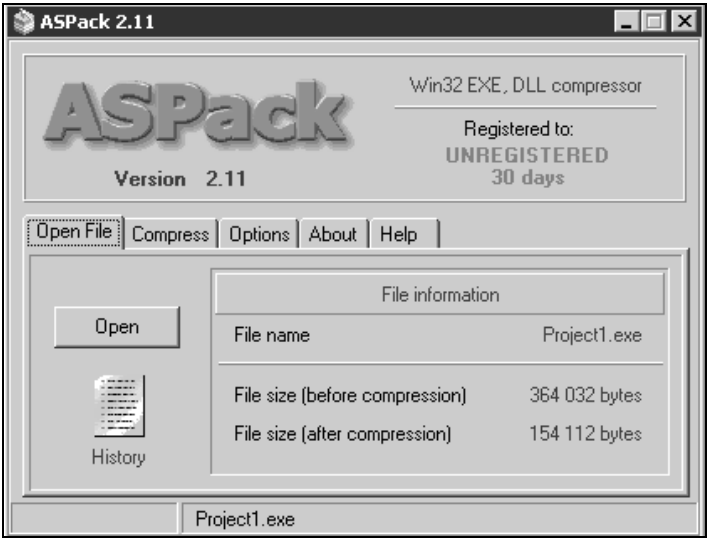


Рис. 1.1. Главное окно ASPack

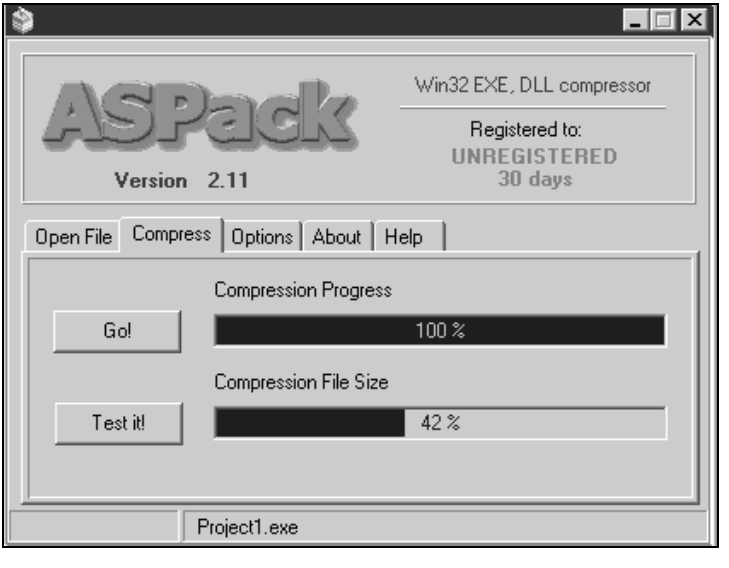


Рис. 1.2. Сжатие файла

Сжатый файл сразу перезаписывает существующий, а старая несжатая версия сохраняется на всякий случай под тем же именем, но с расширением bak. Настроек у ASPack не так уж много (рис. 1.3), и с ними вы сможете разобраться без моей подсказки. Лучше я расскажу вам, как это работает.

Рис. 1.3. Настройки ASPack

Давайте разберемся, как работает сжатие. Сначала весь код программы сжимается архиватором. Если вы думаете, что он какой-то "навороченный", то сильно ошибаетесь. Для сжатия используется обычный архиватор, только оптимизированный для сжатия двоичного кода. После этого, в конец сжатого кода добавляется код разархиватора, который будет программу распаковывать обратно. И в самом конце ASPack изменяет заголовок исполняемого файла так, чтобы при старте сначала запускался распаковщик.

Теперь, когда вы запустите сжатую программу, сначала заработает распаковщик, который "развернет" бинарный код программы и аккуратно разместит его в памяти компьютера. Как только этот процесс закончится, разархиватор передаст управление вашей программе.

Некоторые считают, что из-за расходов на распаковку программа будет работать медленней!!! Я бы сказал, что вы не заметите разницу. Даже если и будут какие-то потери, то они окажутся неощутимыми (по крайней мере, на современных компьютерах). Это происходит потому, что архивация хорошо оптимизирована под двоичный код. И по сути дела, распаковка происходит

только один раз и в дальнейшем никакого влияния на работу программы не оказывает. В результате потери в скорости из-за сжатия будут неощутимы и только на этапе запуска исполняемого файла.

При нормальном программировании с использованием всех современных возможностей типа визуальности и объектного программирования код получается большим, но его можно сжать на 60—70% специальным архиватором. А писать такой код намного легче и быстрее.

Еще один аргумент в пользу применения сжатия — заархивированный код труднее взломать, потому что не каждый дизассемблер сможет прочитать упакованные команды. Так что, помимо уменьшения размера вы получаете защиту, способную отпугнуть большинство взломщиков. Конечно же, профессионала не отпугнешь даже этим, но взломщик средней руки не будет мучиться со сжатым двоичным кодом.

Еще одна утилита для сжатия файлов, получившая большую популярность — UPX. Ее преимуществом является то, что она абсолютно бесплатна и даже распространяется в исходных кодах. Скачать ее можно здесь: **http://upx.sourceforge.net**. Но тут есть одна проблема — утилита выполняется из командной строки, а в наши времена, когда миром правят курсор мыши и простые визуальные окна, люди отвыкли от черного экрана с курсором.

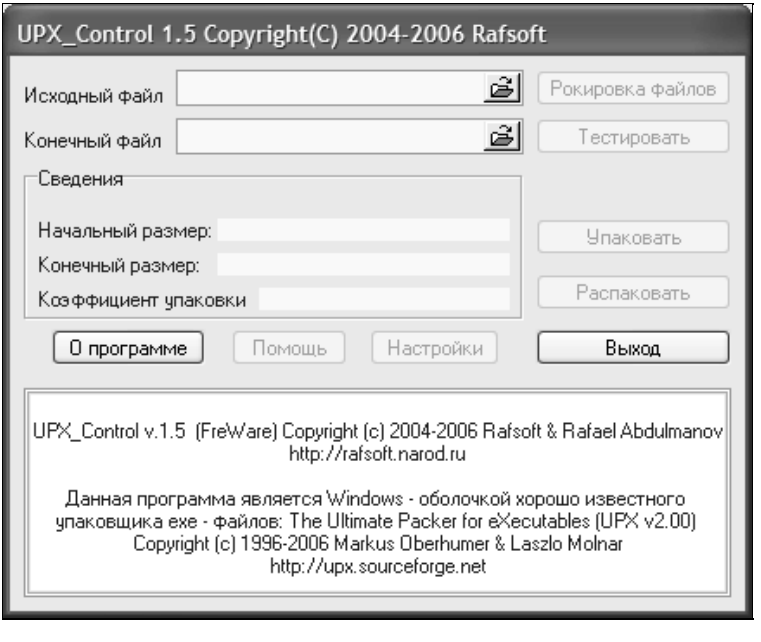


Рис. 1.4. Главное окно программы UPX_Control

Для тех, кто любит визуальность и понятность, могу посоветовать небольшую утилиту UPX_Control, которую можно скачать здесь: **http://rafsoft.narod.ru**. Главное окно еще проще, чем у UPX, а возможности не хуже (рис. 1.4).

В этой программе нужно выбрать исходный файл — исполняемый файл, который необходимо сжать, и нажать кнопку **Упаковать**.

Существует множество программ для сжатия исполняемых файлов. Они отличаются алгоритмом сжатия и возможностями. Некоторые программы позволяют еще шифровать код и предоставлять механизмы, предотвращающие взлом утилит, но это уже необходимо платным программам, а эту тему мы не рассматриваем. К тому же, для большинства универсальных утилит защиты уже давно есть универсальные программы взлома, так что платить за такую защиту деньги — смысла нет, ведь она не эффективна.

1.3. Без окон, без дверей...

Если вы хотите создать программу действительно маленького размера, то необходимо забыть обо всех удобствах. Вы не сможете подключать визуальные формы или другие удобные модули, написанные фирмой Borland для упрощения жизни программиста. Только API-функции самой Windows — и ничего больше.

Для того чтобы создать маленькую программу в Delphi, нужно создать новый проект (по умолчанию система Delphi при открытии сама создаст новый файл проекта, но вы всегда можете самостоятельно создать новое приложение, выбрав в меню **File | New | Application**) и зайти в менеджер проектов (меню **View | Project Manager**). Здесь нужно удалить все модули и формы (пункт **Unit*N***, он выделен на рис. 1.5), чтобы остался только файл самого проекта (по умолчанию его имя Project1.exe). Никаких модулей в проекте не должно быть.

Теперь нужно щелкнуть правой кнопкой мыши на имени проекта и выбрать из появившегося контекстного меню пункт **View Source** или в главном меню **Project** выбрать пункт **View Source**. В редакторе кода откроется файл проекта Project1.dpr. Если вы уже удалили все модули, то его содержимое должно быть таким:

```
program Project1;

uses
  Forms;

{$R *.res}
```

```
begin
  Application.Initialize;
  Application.Run;
end.
```

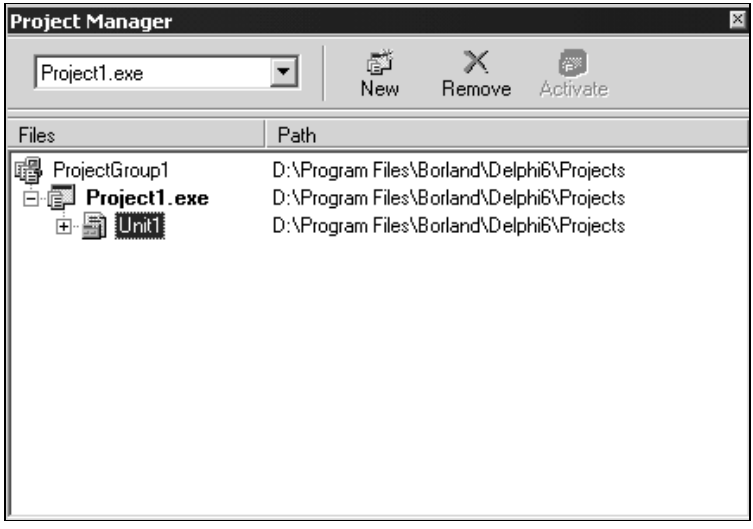


Рис. 1.5. Project Manager

Теперь можно скомпилировать абсолютно пустой проект. Для этого надо выбрать в меню **Project** пункт **Compile Project** или нажать комбинацию клавиш <Ctrl>+<F9>. После компиляции следует выбрать в меню **Project** команду **Information for Project1**. Появится окно с информацией о проекте. Окно, которое появилось передо мной, вы можете увидеть на рис. 1.6.

В правой части окна должны быть описаны используемые пакеты. Так как вы все удалили, значит, там должна красоваться надпись **(None)**. А вот с левой стороны должна быть описана информация о скомпилированном коде. Самая последняя строка показывает размер файла, и у меня он равен 370 688 байтов. Ничего себе "пустая программа"!!! Мы же ничего еще не написали. Откуда же тогда такой большой код?

Давайте разберемся, что осталось в нашем проекте, чтобы обрезать все то, что еще не обрезано. Сразу обратите внимание, что в разделе `uses` подключен модуль `Forms`. Это объектный модуль, написанный "дядей Борландом", а значит, его использовать нельзя, потому что именно он увеличивает размер нашей программы. Между командами `begin` и `end` используется объект

`Application`. Этот объект тоже указывать не надо, потому что он использует объектные возможности и не заботится о "фигуре" программы.

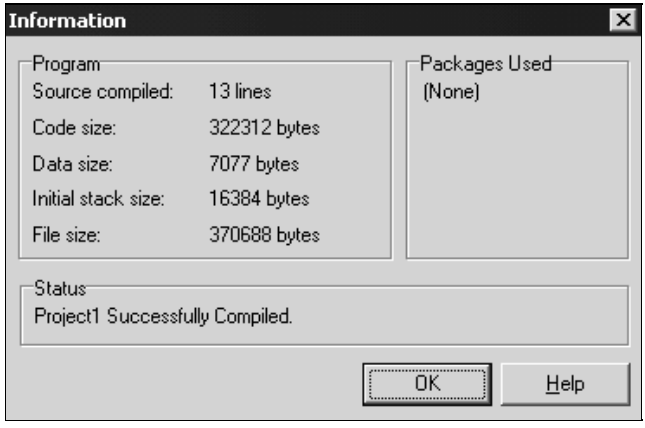


Рис. 1.6. Окно информации о проекте

Большой объем, который появляется даже у пустой программы, как раз и связан с объектом `Application`, который объявлен в модуле `Forms`. Хотя мы использовали только два метода — `Initialize` и `Run`, — при компиляции в exe-файл попадает весь объект `TApplication`, а он состоит из сотен, а может и тысяч строчек кода. Помимо этого, вместе с `TApplication` в исполняемый файл потянется целая куча и еще маленький вагончик дополнительного мусора, который реально в данном проекте не используется.

Чтобы избавиться от накладных расходов, нужно заменить модуль `Forms` на `Windows`, который описывает только функции Windows API и не связан с объектами Delphi. Его подключение является обязательным, иначе вы не сможете вызвать ни одной функции из набора WinAPI. А между `begin` и `end` вообще все можно удалить. В итоге, самый минимальный (с учетом использования модуля `Windows`) код программы будет выглядеть так:

```
program Project1;

uses Windows;

begin

end.
```

Снова откомпилируйте проект. Откройте окно информации и посмотрите на размер получившегося файла. У меня получилось 8192 байта (рис. 1.7). Вот это уже по-человечески. Стоило только отказаться от библиотеки VCL, как размер файла сократился в несколько раз. Есть еще приемы, которые позволят сократить размер исполняемого файла до 6 Кбайт или даже менее, но это уже не важно. Я не вижу смысла мучаться из-за 2—4 Кбайт, которые не сыграют большой роли.

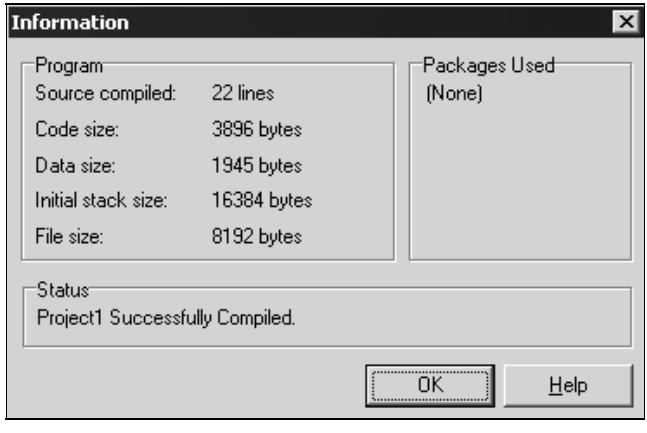


Рис. 1.7. Информации о проекте после удаления ненужного кода

Заготовка минимальной программы с использованием WinAPI готова. Теперь вы можете смело добавлять свой код. Мне нужно только объяснить вам, какие модули можно подключать к своему проекту в раздел `uses`, а какие — не стоит, дабы "жирность" исполняемого файла резко не увеличилась. Тут все очень просто и не займет много времени.

Если при установке Delphi вы не отключали копирование исходных кодов библиотек, то перейдите в каталог, куда вы установили Delphi. Здесь перейдите в папку Source, затем в Rtl и, наконец, в Win. Если вы отключили копирование исходников, то вставьте компакт-диск с Delphi и поищите эти каталоги там. В них расположены исходные коды модулей, в которых описаны все API-функции Windows. Именно эти модули вы можете подключать к своим проектам, если хотите получить маленький код. Если вы подключите что-то другое, то я уже не гарантирую минимум размера вашей программы (хотя, есть и исключения). Самое опасное — это подключение модулей из каталога Delphi7\Source\Vcl\.

Сразу же рассмотрим пример. Если вы хотите, чтобы в вашей программе были возможности работы с сетью, то вам нужно подключить к нему биб-

лиотеку Window Socket. Среди модулей WinAPI есть файл с именем winsock.pas. Значит, вы должны в разделе `uses` написать `Winsock` (расширение писать не надо), и ваша программа сможет работать с сетью.

Пока что я описал минимальный проект, в который можно добавлять свой код. Но код, который вы вставите, выполнится один раз, и программа выгрузится из памяти. А что если вам надо, чтобы программа постоянно "висела" в памяти и что-то делала? Для этого нужно использовать следующий шаблон для своих программ:

```
program Project1;

uses Windows;

var
  Msg: TMsg;
Begin

// Сюда можно добавлять свой код

// Дальше идет код, который заставит программу "висеть"
// в памяти вечно и не будет сильно загружать систему
     while GetMessage(Msg, HInstance, 0, 0) do
      begin
        TranslateMessage(Msg);
        DispatchMessage(Msg);
      end;
end.
```

Сейчас я не буду описывать этот шаблон, потому что дальше мы подробно обсудим написание полноценного шаблона минимального приложения. Там и будут описаны функции, которые используются в этом примере.

Самое интересное, что такое минимальное приложение будет не видно в системе. Мы не создавали никаких окон, значит, на экране ничего отображаться не будет. Программа не будет иметь фокуса ввода, значит, в панели задач тоже незачем что-то отображать.

У нас получилась не программа, а процесс. Именно поэтому его можно заметить в Windows 2000/XP только с помощью программ, которые умеют отображать все процессы, запущенные в системе. В Windows 2000/XP при

нажатии комбинации клавиш <Ctrl>+<Alt>+<Del> появляется окно **Диспетчер задач Windows** (рис. 1.8). В этом окне есть вкладка **Процессы**, где и можно найти такую программу. Вообще-то, на этой вкладке можно найти любую работающую программу, и от инспекторского взора Диспетчера задач спрятаться очень тяжело. Так что, достаточно только назвать исполняемый файл не сильно вызывающе, и простой пользователь ничего не заподозрит, потому что список процессов достаточно большой, и я не думаю, что кто-то знает хотя бы половину из них. Большинство пользователей в этот список просто не заглядывают.

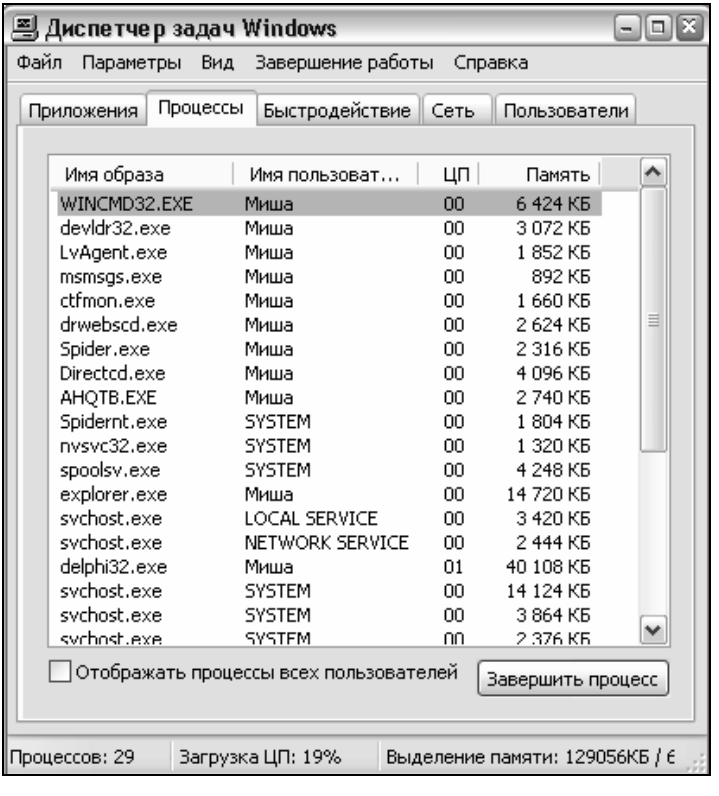


Рис. 1.8. Вкладка **Процессы** в Диспетчере задач Windows XP

В Windows 9*x* при нажатии комбинации клавиш <Ctrl>+<Alt>+<Del> такая программа пока еще будет видна. Чтобы окончательно скрыть ее из виду, нужно добавить следующий код:

```
procedure RegisterServiceProcess; external 'kernel32.dll' name 'Register-
ServiceProcess';
```

Здесь объявлена недокументированная процедура `RegisterServiceProcess`, которая есть в ядре kernel32.dll. Эта процедура делает нашу программу процессом в системе Windows 9*x*. Данную строку с описанием процедуры нужно добавить сразу после раздела `uses`.

Чтобы вызвать эту процедуру, нужно написать следующий код в любом месте программы (желательно в начале):

```
asm
 push 1
 push 0
 call RegisterServiceProcess;
end;
```

Этот код использует код на языке ассемблера, чтобы вызвать WinAPI-функцию `RegisterServiceProcess`. Да, в Delphi есть встроенный ассемблер. Вы можете прямо среди кода на Delphi писать код на ассемблере. Инструкции языка ассемблера нужно заключать между словами `asm` и `and;`. В примере используются три инструкции. В первой в стек помещается число `1` с помощью операции `push`. Во второй строке в стек отправляется значение `0`. Эти два числа, находящиеся в стеке, являются параметрами для функции `RegisterServiceProcess`. То же самое можно было бы сделать и с помощью вызова функции на Delphi: `RegisterServiceProcess(0, 1)`, но мне захотелось показать вам, как работает встроенный ассемблер. Да и для этого нужно изменить описание процедуры `RegisterServiceProcess`, которая регистрирует программу как процесс и в Windows 9*x* делает программу невидимой в списке запущенных программ, который вы вызываете нажатием комбинации клавиш <Ctrl>+<Alt>+<Del>. В последней инструкции ассемблера я вызываю процедуру `RegisterServiceProcess` с помощью оператора `call`.

Теперь наша программа не будет видна и в Windows 9*x*. Только вы должны учитывать, что этот код прекрасно работает в Windows 9*x*, но выдаст ошибку в Windows 2000/XP, потому что в его ядре нет функции `RegisterServiceProcess`. Поэтому вы должны знать, в какой системе будет запускаться программа, и можно ли использовать процедуру `RegisterServiceProcess`.

1.4. Шаблон минимального приложения

Теперь нам пора написать хороший шаблон минимального приложения, который мы будем использовать в этой книге для написания наших программ-приколов. Для этого нам надо познакомиться с "внутренностями" Windows и его WinAPI.

Запустите уже полюбившийся Delphi. Как всегда, сразу откроется новый проект. Так как мы очень часто будем делать минимальные программы, то нам абсолютно не нужны никакие формы, их надо удалить. В меню выберите **View | Project Manager** (рис. 1.9). Перед вами появится окно менеджера проектов. Выделите **Unit1** и нажмите кнопку **Remove** для удаления лишней формы (рис. 1.10).

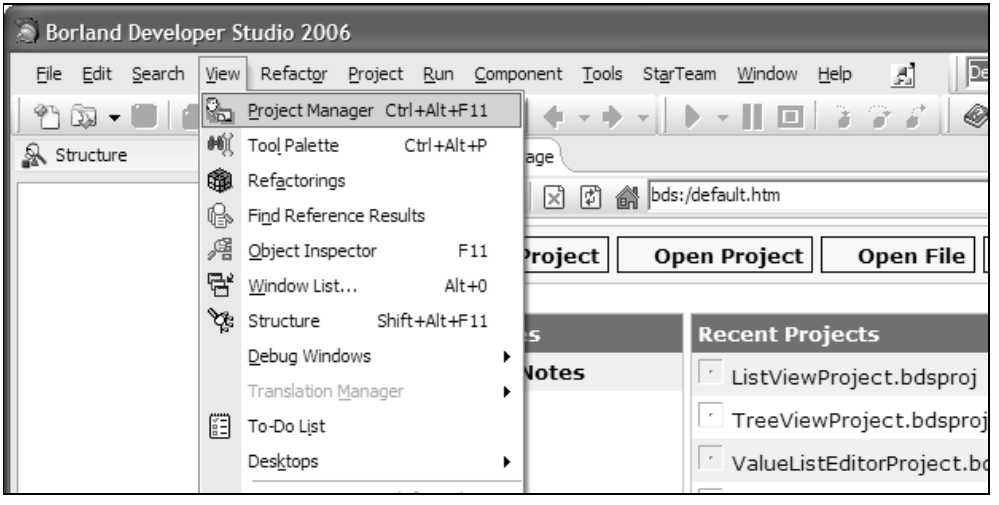


Рис. 1.9. Вызов менеджера проектов

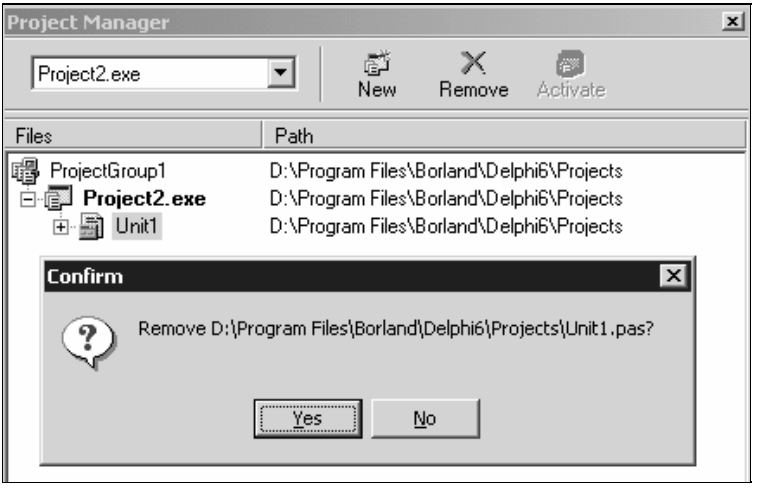


Рис. 1.10. Удаление ненужной формы

В Delphi 2006 по умолчанию все панели расположены в одном окне, и менеджер проектов будет выглядеть в виде панели справа вверху.

Теперь выбираем в меню **Project** пункт **View Source**. Если все в норме, то в редакторе кода вы увидите код вашего проекта. Оставляем только первую строчку `program Project1`, а все остальное удаляем и вместо этого пишем код, представленный в листинге 1.1 (комментарии — строки текста, которые идут после символов `//`, — естественно, можно опускать).

Листинг 1.1. Шаблон минимальной программы

```
uses
  windows, messages, sysutils;

{$R *.RES}

var
 Instance: HWnd;
 WindowClass: TWndClass;
 Handle: HWnd;
 msg: TMsg;

// Процедура выхода из программы
procedure DoExit;
begin
 Halt;
end;

// Функция обработки событий Windows
function WindowProc(Hwnd, msg, wparam, lparam: longint): longint;
 stdcall;
begin
 if msg=wm_destroy then
  DoExit;

 if msg=wm_KeyDown then
  if wparam=VK_ESCAPE then
   DoExit;
```

```
 result:=defwindowproc(Hwnd,msg,wparam,lparam);
end;

// Отсюда начинается выполнение программы
begin
// Получаем описатель модуля
 instance :=GetModuleHandle(nil);

// Заполняем структуру WindowClass
 WindowClass.style:=CS_HRedraw or CS_VRedraw;
 WindowClass.Lpfnwndproc:=@windowproc;
 WindowClass.Hinstance:=Instance;
 WindowClass.HbrBackground:= color_btnface;
 WindowClass.LpszClassName:='DX';
 WindowClass.Hcursor:=LoadCursor(0,IDC_ARROW);

// Регистрируем новый класс
 RegisterClass(WindowClass);

// Создаем окно
 Handle:=CreateWindowEx(0,'DX','',WS_POPUP, 1,1, 200, 200,0,0,
                        Instance, nil);

// Отображаем окно
 ShowWindow(Handle, SW_SHOW);
 UpdateWindow(Handle);

// Здесь можно производить инициализацию

// Цикл обработки сообщений
 while (GetMessage(msg, 0, 0, 0)) do
  begin
   translatemessage(msg);
   dispatchmessage (msg);
  end;
end.
```

Теперь давайте подробно разберем листинг 1.1. После старта программа начинает выполнение с первого оператора `begin` (это место обозначено соответствующим комментарием). Первой строкой кода идет вызов WinAPI-функции `GetModuleHandle`. Она возвращает манипулятор модуля, который сохраняется в переменной `Instance`. Этот манипулятор нам пригодится немного позже.

Далее заполняется структура `WindowClass`. Эта структура используется при создании нового класса окна. Для минимального приложения нам понадобится заполнить следующие поля:

- `style` — стиль окна;
- `Lpfnwndproc` — сюда нужно записать указатель на процедуру, которая будет вызываться в ответ на все пользовательские или системные события;
- `Hinstance` — манипулятор, который мы получили в первой строчке кода;
- `HbrBackground` — цвет фона (в принципе, он необязателен, но я решил покрасить фон системным цветом кнопок);
- `LpszClassName` — имя создаваемого класса;
- `Hcursor` — курсор. Сюда загружается стандартный курсор-стрелка.

Все, структура готова, и мы можем зарегистрировать новый класс будущего окна. Для этого вызывается WinAPI-функцию `RegisterClass(WindowClass);`. После этого в системе есть описание вашего будущего окна. Почему будущего? Да потому что само окно мы еще не создали. Для этого нужно еще вызвать функцию `CreateWindowEx`. У нее много параметров, и давайте рассмотрим их подробнее:

```
CreateWindowEx(0,'DX','',WS_POPUP,1,1,200,200,0,0,Instance,nil);
```

Рассмотрим параметры этой функции.

- Расширенный стиль окна. Нам он не нужен, поэтому у меня первый параметр равен нулю. Рассмотрим основные флаги, которые могут быть использованы в этом параметре:
  - `WS_EX_ACCEPTFILES` — окно, на которое можно перетягивать файлы;
  - `WS_EX_APPWINDOW` — главное окно, кнопка которого будет отображаться в панели задач;
  - `WS_EX_CLIENTEDGE` — окно имеет клиентскую область с видимой границей;
  - `WS_EX_CONTEXTHELP` — в заголовке окна есть кнопка помощи для вызова контекстного меню;
  - `WS_EX_CONTROLPARENT` — разрешается перемещение по элементам окна с помощью клавиши <Tab>;

- WS_EX_DLGMODALFRAME — окно с двойной границей. Такие окна используются в качестве модальных диалоговых окон;
- WS_EX_MDICHILD — дочернее окно для многодокументного приложения;
- WS_EX_RIGHTSCROLLBAR — с правой стороны будет полоса прокрутки;
- WS_EX_TOOLWINDOW — окно инструментов. Такие окна имеют узкое системное меню и используются для хранения кнопок или других вспомогательных инструментов для главного окна;
- WS_EX_TOPMOST — окно будет поверх всех остальных.

❒ Имя класса. Мы зарегистрировали класс DX, значит, и здесь мы должны указать именно этот класс.

❒ Имя окна. При программировании графики имя окна не нужно. Потому что окно будет полноэкранным.

❒ Стиль окна. Нас интересует простейшее WS_POPUP-окно. Помимо этого, можно использовать сочетания из следующих значений:

- WS_BORDER — окно имеет границу в виде тонкой линии;
- WS_CAPTION — окно имеет заголовок;
- WS_CHILD — дочернее окно;
- WS_DISABLED — окно будет отключено и не сможет получать данные от пользователя;
- WS_DLGFRAME — окно будет иметь границу, как у диалоговых окон;
- WS_MAXIMIZE — после создания окно будет максимизировано;
- WS_MAXIMIZEBOX — окно будет иметь кнопку максимизации;
- WS_MINIMIZE — окно создается минимизированным;
- WS_MINIMIZEBOX — окно будет иметь кнопку минимизации;
- WS_SYSMENU — окно будет иметь системное меню;
- WS_TABSTOP — окно сможет получать фокус ввода;
- WS_VISIBLE — после создания окно будет видимо.

❒ Следующие четыре параметра — это левая и правая позиции, ширина и высота окна. Если указать все равными нулю, то значения будут выбраны по умолчанию.

❒ Главное окно по отношению к создаваемому. Наше окно само по себе главное, поэтому я указываю 0.

❒ Меню.

- Это манипулятор модуля (`Instance`), полученный после вызова `GetModuleHandle`.
- Параметры окна. Этот параметр используется при создании многодокументных окон, поэтому я указываю "пусто" (`nil`).

После создания окна его надо отобразить. Делается это с помощью вызова процедуры `ShowWindow`. У этой процедуры два параметра:

- созданное окно;
- параметры отображения окна. Здесь указано `SW_SHOW`, чтобы просто активизировать и отобразить окно. Остальные значения параметра можно посмотреть в файле справки по WinAPI-функциям (**Help | Windows SDK**).

И последняя подготовительная функция — `UpdateWindow`. Это просто прорисовка созданного окна.

Теперь разберемся с циклом обработки сообщений. Функция `GetMessage` ожидает пользовательского или системного сообщения, и как только оно наступает, возвращает `true` (истина). Полученное сообщение преобразуется в необходимый вид с помощью `translatemessage` и отправляется обработчику сообщений помощью вызова функции `dispatchmessage`.

В каждой программе должна быть процедура обработки сообщений. Какая именно? Мы указали ее при создании класса окна в свойстве `WindowClass.Lpfnwndproc`. Я ее назвал `WindowProc` — стандартное имя, используемое по умолчанию. Сама же процедура должна выглядеть приблизительно так, как в листинге 1.1.

В процедуре-обработчике событий желательно всегда делать вызов функции `defwindowproc`. Эта функция ищет в системе обработчик полученного сообщения, установленный по умолчанию. Это очень важно, чтобы вам не пришлось писать то, что может сделать сама ОС.

После этого можно начинать обработку полученного сообщения. Это происходит с помощью сравнивания параметра `msg` со стандартными событиями. Например, если `msg` равно `wm_destroy`, то это значит, что программа прощается с нами и хочет уничтожиться. По этому событию можно освободить выделенную память под пользовательские данные.

В обработчике есть еще два сравнения. Если `msg` равно `wm_KeyDown`, т. е. пользователь нажал какую-то клавишу, то проверяется, не является ли она клавишей <Esc> — выражение `wparam=VK_ESCAPE`, и в случае его истинности приложение также завершает свою работу.

Вот и все, с шаблоном мы разобрались. Если вы сейчас запустите созданную программу, то перед вами появится пустое окно черного цвета. Чтобы его закрыть, просто нажмите комбинацию клавиш <Ctrl>+<F4>, потому что никаких кнопок на нем нет.

Если вы захотите сделать это окно невидимым, то просто уберите из кода функцию `ShowWindow`, которая отображает окно на экране. После этого, ваша программа сразу же станет невидимой в системе. Можно так же изменить второй параметр этой процедуры. Сейчас он равен `SW_SHOW`, но можно указать `SW_HIDE`, чтобы спрятать окно. В принципе, указание параметра `SW_HIDE` внешне равносильно отсутствию вызова процедуры.

Чуть позже мы еще встретимся с процедурой `ShowWindow`, и не один раз.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 1\Minimum\ вы можете увидеть исходный код этого примера.

1.5. Создание элементов управления с помощью Windows API

Пустое окно не принесет нам счастья, поэтому давайте посмотрим, как можно с помощью Windows API поместить на него элементы управления, и самое интересное — обработать события от этих элементов.

Для создания элементов управления используется функция `CreateWindow`:

```
function CreateWindow(
 lpClassName: PChar;
 lpWindowName: PChar;
 dwStyle: DWORD;
 X, Y,
 nWidth, nHeight: Integer;
 hWndParent: HWND;
 hMenu: HMENU;
 hInstance: HINST;
 lpParam: Pointer
): HWND;
```

Ее отличие от функции `CreateWindowEx`, которую мы рассмотрели в *разд. 1.4*, состоит в том, что нет первого параметра (расширенного стиля окна). В принципе, простые окна тоже можно создавать с помощью функции `CreateWindow`, но она устарела, поэтому рекомендуется использовать более современный вариант — `CreateWindowEx`.

При создании элементов управления в качестве класса окна (параметр `lpClassName`) нужно указывать заранее предопределенные в системе имена классов элементов управления:

- `BUTTON` — кнопка;
- `COMBOBOX` — выпадающий список;
- `EDIT` — поле ввода;
- `LISTBOX` — список выбора;
- `STATIC` — статический текст;
- `SCROLLBAR` — полоса прокрутки.

Как видите, перечисленные элементы управления — это те же окна, только имеют заранее определенный вид и действуют по определенным правилам. А в остальном к этим элементам можно обращаться как к окнам и с помощью тех же функций.

Результатом выполнения функции `CreateWindow` является указатель на созданный элемент управления. Его необходимо сохранить для того, чтобы в дальнейшем можно было работать с этим элементом.

Итак, откроем пример, который мы написали в *разд. 1.4*, и после создания основного окна добавим код создания пары кнопок:

```
pButton1 := CreateWindow('button', 'This is button 1',
  WS_CHILD or WS_VISIBLE or BS_DEFPUSHBUTTON,
  10, 10, 200, 25, Handle, 0, hInstance, nil);

pButton2 := CreateWindow('button', 'This is button 2',
  WS_CHILD or WS_VISIBLE or BS_DEFPUSHBUTTON,
  10, 50, 200, 25, Handle, 0, hInstance, nil);
```

Кнопки — наиболее часто используемые компоненты, и они достаточно просты. При этом мы заодно узнаем, как отловить события, когда пользователь нажимает кнопки.

Чтобы отловить события, необходимо добавить соответствующий обработчик в функцию обработки событий — `WindowProc`, все кнопки генерируют событие `WM_COMMAND`, а вот чтобы определить, какая именно из них сработала, необходимо проверить последний параметр (`lparam`), передаваемый в `WindowProc`. Через него передается экземпляр окна, сгенерировавшего событие.

Итак, добавьте следующий код в функцию обработки события:

```
function WindowProc(Hwnd, msg, wparam, lparam: longint): longint;
 stdcall;
```

```
begin
 // Событие WM_COMMAND
 if msg=WM_COMMAND then
  begin
   // Нажата первая кнопка
   if pButton1=lparam then
    MessageBox(0, 'Событие от кнопки 1', 'Внимание', 0);

   // Нажата вторая кнопка
   if pButton2=lparam then
    MessageBox(0, 'Событие от кнопки 2', 'Внимание', 0);
  end;

 // Обработка других событий
 ...
 ...
end;
```

По мере необходимости мы еще познакомимся с другими элементами управления и WinAPI-функциями, с помощью которых можно управлять ими.

1.6. Прячем программы

Написание программ маленького размера — достаточно сложное занятие. Чаще всего нам нужно просто написать какой-то пример, в котором размер файла не имеет особого значения, а невидимость обеспечить необходимо. В таком случае стратегия написания невидимой программы будет немного другой, и мы ее сейчас рассмотрим на практике.

Создайте новый проект в Delphi. Перенесите на форму одну только кнопку и измените ее свойство `Caption` на `Спрятать`. При нажатии этой кнопки наше приложение будет скрываться. Теперь создайте обработчик события `OnClick` для этой кнопки и там напишите:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 ShowWindow(Handle,SW_HIDE);
 ShowWindow(Application.Handle,SW_HIDE);
end;
```

Здесь использована уже знакомая функция `ShowWindow`. Как я уже говорил, эта процедура выводит окно. В качестве второго параметра процедуре в обоих случаях передается значение `SW_HIDE`, которое заставляет делать окно невидимым. При первом вызове процедуры в качестве первого параметра указано `Handle` — свойство, в котором хранится манипулятор текущего (в данном случае главного) окна, чтобы сделать его невидимым.

При втором вызове указано `Application.Handle` — манипулятор всего приложения. Теперь приложение становится абсолютно невидимым. Точнее сказать — видимым, но только на вкладке **Процессы** в окне **Диспетчер задач Windows**.

Как я уже говорил, большинство пользователей никогда не заглядывает на эту вкладку, потому что разобраться там в чем-то очень тяжело. Достаточно дать вашей программе какое-нибудь наименее вызывающее имя, и она станет абсолютно незаметной.

Тут же вспоминается случай, когда несколько лет назад я написал своего первого трояна, с помощью которого я подшучивал над своим начальником. Я приблизительно таким же образом спрятал программу и дал ей название internat32.exe. В системе Windows 95/98 есть такой сервис, как internat.exe, и он присутствует всегда. Именно поэтому появление нового сервиса internat32 ни у кого не вызвало подозрений. Мой начальник долго искал причину, по которой на его компьютере исчезает звук, и компьютер сам перегружается, но ничего не нашел. Его компьютер даже рассматривали компьютерщики нашей фирмы, и даже они ничего не нашли. Просто никто не обращал внимания на такой неприглядно названный файл.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 1\Hide App\ вы можете увидеть исходник этого примера.

1.7. Использование библиотек KOL и MCK

Следующий способ уменьшить размер программы — это использовать библиотеки KOL и MCK. Этот метод имеет очень серьезное преимущество — вы не отказываетесь от объектного программирования и можете использовать основные его преимущества. Почему основные? Просто визуально доступны далеко не все компоненты, но их количества достаточно для большого разнообразия программ.

Библиотека KOL (Key Objects Library, библиотека ключевых объектов) содержит объекты, которые упрощают программирование на Windows API и

при этом не увеличивают код. Хотя — нет, увеличение существует, просто оно не настолько значительное, как при использовании VCL.

MCK (Mirror Classes Kit, комплект зеркальных классов) — библиотека, позволяющая использовать KOL визуально. Это отличная надстройка, которая максимально эффективно использует возможности KOL и при этом минимально влияет на размер исполняемого файла.

Так как визуальная среда Delphi плохо подходит для создания компактного кода и больше ориентирована на использование родных библиотек VCL и CLX, то при создании визуальности с использованием KOL разработчикам пришлось неплохо попотеть. С одной стороны, у разработчиков получилась немного неудобная реализация и с небольшими недостатками, но с другой стороны — гениальная. После этого задумываешься, чем занимались в Borland, если библиотека VCL нагружает исполняемый файл почти в 100 раз сильнее.

Примечание

Web-сайты проекта **http://bonanzas.rinet.ru** и **http://xcl.cjb.net**, а автор — наш соотечественник Владимир Кладов.

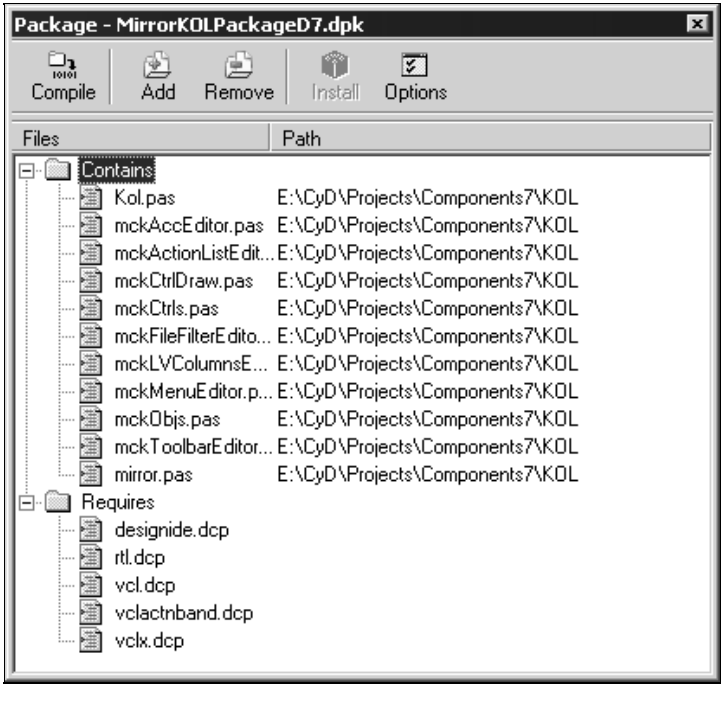


Рис. 1.11. Окно установки пакета

Установка KOL и VCL проста, как три копейки. Разархивируем содержимое архивов kol.zip и mck.zip в один и тот же каталог. Теперь открываем файл MirrorKOLPackageD*X*.dpk, где *X* — это номер версии вашей системы Delphi. У меня 7-я версия Delphi, но даже при установке файла для 6-й версии библиотека устанавливается без проблем. Итак, после открытия пакета перед нами откроется окно установки (рис. 1.11). Нажимаем в нем кнопку **Install**, и ждем секунду до окончания установки пакета.

После установки пакета нам становится доступной новая вкладка **KOL** (рис. 1.12). Здесь расположены зеркальные компоненты для основных компонентов Delphi. Чтобы код был компактным, нужно использовать именно их. Но перед этим нужно правильным образом создать проект.

Рис. 1.12. Компоненты MCK

Чтобы создать проект с использованием KOL и MCK, нужно выполнить несколько нехитрых операций. На первый взгляд они глупы и непонятны, но потом привыкаешь. Для начала создаем стандартный проект Delphi для простого приложения и сохраняем все файлы проекта в одном каталоге. Помните, что все файлы должны быть в одном и том же каталоге, и разбрасывать их по разным папкам нежелательно. При сохранении проекта его название не имеет значения, потому что запускаемый файл будет иметь другое имя.

Теперь бросаем на форму компонент `KOLProject` и в свойстве `projectDest` пишем осмысленное название проекта, потому что это имя будет использоваться для запускаемого файла и для имени проекта. Да-да, мы еще только создаем проект.

Теперь бросаем на форму компонент `KOLForm`. Этот компонент говорит о том, что у нас будет визуальная форма. Сохраняем все.

Загляните сейчас в каталог, где сохранялся проект. Обратите внимание, что здесь появилась куча файлов с расширением inc и новый файл проекта с именем, которое мы указали в свойстве `projectDest` компонента `KOLProject` и расширением dpr. В данном случае это будет TestProject.dpr (можете сразу же открыть его). Это и есть проект MCK, который нужно открывать и визуально работать как с любым другим проектом.

Приложение, которое мы создали первым, можно закрыть и больше уже не открывать. Оно нужно только для создания размещения компонентов

KOLProject и KOLForm. Все остальные действия и компиляция должны происходить в автоматически созданном проекте TestProject.dpr.

Итак, откройте созданный проект и откомпилируйте его. Посмотрим на размер полученного запускного файла. Если пустой VCL-проект занимает намного больше 200 Кбайт, то проект с KOL и MCK у меня занял чуть более 23 Кбайт. Ну, разве не прелесть? Уменьшение кода почти в 10 раз без потери визуальности. А ведь полученный исполняемый файл можно еще и сжать с помощью UPX или ASPack.

Теперь можно визуально расставлять на форме компоненты с вкладки **KOL** и использовать привычным образом, и при этом размер файла будет увеличиваться очень медленно. По крайней мере, пока вы не подключите какой-нибудь заголовочный файл из состава VCL, и он не потянет паровозом в исполняемый файл всякие ненужные добавки, которые разработчики Borland написали в модулях своей библиотеки.

1.8. Динамические библиотеки

Уменьшить размер исполняемого файла можно с помощью использования динамических библиотек. Как хорошо было бы, если бы тяжеловесная визуальная библиотека была выполнена в виде библиотеки и присутствовала в ОС. Тогда нам достаточно было бы только писать логику программы и использовать функции из динамически загружаемых библиотек.

На самом деле, две популярные библиотеки — VCL и MFC — уже давно поддерживают динамическое подключение. В Visual C++ на этапе создания проекта или уже после создания (с помощью окна свойств проекта) можно указать, как должна подключаться библиотека MFC. Есть два варианта выбора (рис. 1.13):

- **Use MFC in a shared DLL**;
- **Use MFC in a static library**.

В первом случае исполняемый файл получается намного меньше, зато для запуска программы необходимо скопировать на компьютер пользователя dll-файлы библиотеки MFC. Вы когда-нибудь встречали сообщения типа: "Dll-файл mfcXX.dll не найден". Файл mfcXX.dll (XX — это номер версии библиотеки, которая использовалась во время компиляции проекта) как раз и содержит все классы MFC, и для четвертой версии он имеет размер чуть более 900 Кбайт.

Библиотека VCL из Delphi тоже может применяться динамически, только здесь для хранения стандартных объектов используются не классические dll-файлы, а bpl-файлы. Чтобы библиотека подключалась динамически, выби-

раем меню **Project | Options** и в разделе **Packages** устанавливаем флажок в параметре **Build with runtime packages** (рис. 1.14).

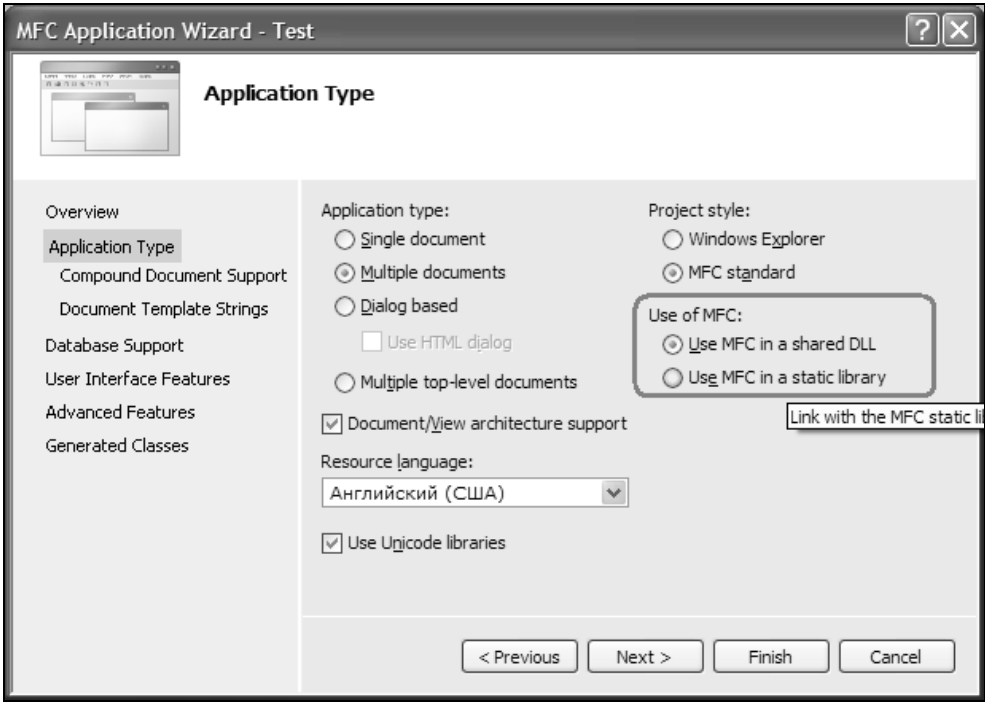


Рис. 1.13. Во время создания проекта можно указать, как использовать библиотеку

Я провел небольшой тест в Delphi 2006. Сначала был создан пустой проект и откомпилирован со статической стыковкой с VCL, а потом — с динамической. В первом случае файл занял на диске 384 Кбайт. В 7-й версии размер был намного меньше, и с каждой новой версией Delphi размер все растет и растет. Во втором случае (с динамическим подключением VCL) в Delphi 2006 файл занял 18 Кбайт. Но теперь проект не будет запускаться без bpl-файлов, которые содержат все необходимые классы.

Но какие bpl-файлы необходимы? Компилируем проект и выбираем пункт меню **Project | Information for Project1**. В данном случае, Project1 — это имя моего открытого проекта. Если вы назвали его по-другому, то и пункт меню будет иным. Если не было предварительной компиляции, то пункт меню может быть недоступным.

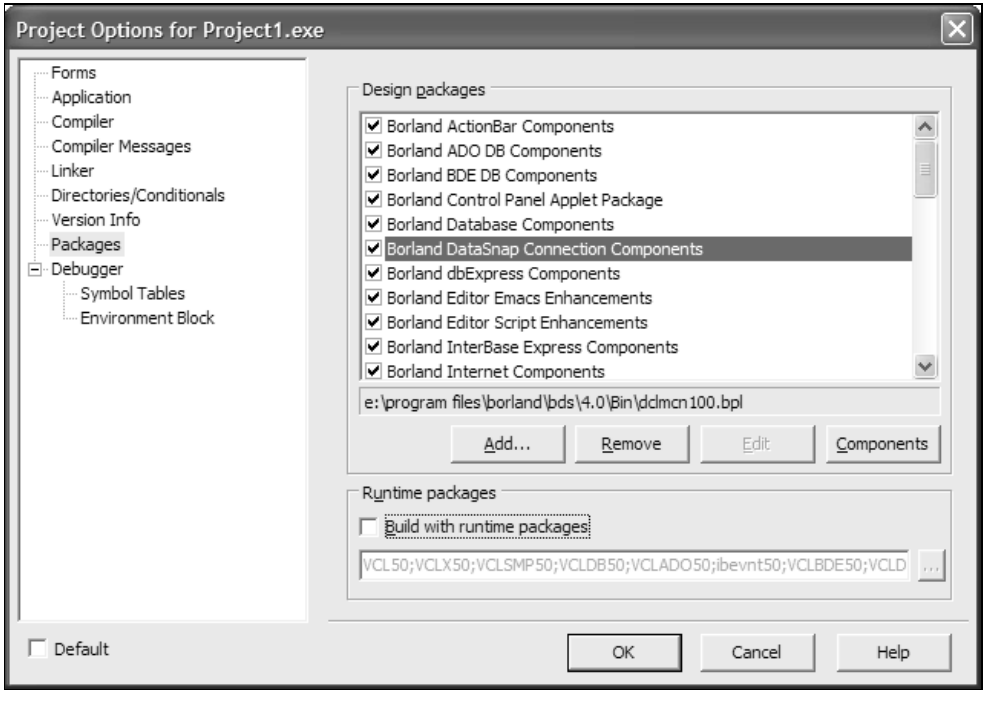


Рис. 1.14. Окно настроек пакетов

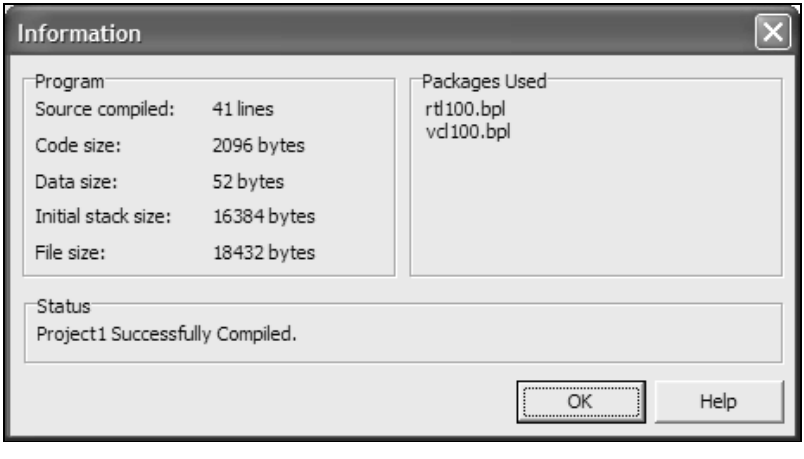


Рис. 1.15. Информация о проекте,
с динамическим использованием VCL

Перед вами откроется окно информации о проекте (рис. 1.15). Справа, в разделе **Packages Used**, можно увидеть имена файлов библиотек, которые использованы динамически. Если их не будет на компьютере пользователя, где должна работать программа, то она не запустится. Чтобы все работало корректно, эти файлы необходимо скопировать на компьютер пользователя и лучше поместить в каталог Windows\System32, чтобы библиотека была доступна всем программам.

А теперь посмотрим, сколько места на диске занимают библиотеки vcl100.bpl и rtl100.bpl. Первая в моем случае занимает мегабайт, а вторая более 800 Кбайт. Получается, что, помимо 18 Кбайт самой программы, мы должны скопировать пользователю еще 1,8 Мбайт, иначе могут быть проблемы. Выгода получается не очень хорошей.

Когда у пользователя будет выполняться несколько программ, написанных на одном и том же языке, то с помощью динамического подключения функций можно сэкономить достаточно большое дисковое пространство. Но при этом скорость программы не уменьшается, а даже немного увеличивается. У вас получается лишняя прослойка между программой и ОС, и это — библиотека классов/объектов.

1.9. Оптимизация программ

Вся наша жизнь — это борьба с тормозами и нехваткой времени. Каждый день мы тратим по несколько часов на оптимизацию. Каждый из нас старается оптимизировать все, что попадает под руку, особенно когда приходится выполнять рутинную работу. А вы уверены, что делаете это правильно? Может быть, есть возможность что-то сделать еще лучше?

Я понимаю, что многие сейчас разленились и выполняют свои обязанности спустя рукава. Лично я до такой степени привык, что за меня все делает компьютер, что даже забыл, как выглядит шариковая ручка. Недавно мне пришлось писать заявление на отпуск на простой бумаге, так я забыл, как пишется буква "ю". Пришлось подглядывать, как она выглядит на клавиатуре. Это не шутка. Это прогресс, благодаря которому я все делаю на компьютере.

Даже для того, чтобы написать текст из двух строк, мы включаем свой компьютер и загружаем MS Word, тратя на это драгоценное время. А может легче было бы написать этот текст вручную? Я вас понимаю — не солидно!!!

Программисты — так это вообще "полное бесстыдство", как говорил один из моих преподавателей: "Тра-та-та". Если они считают, что их творение (в виде исходного кода) никто не увидит, то можно писать что угодно? Так они

ошибаются. С этой точки зрения программы с открытым исходным кодом в большом преимуществе, потому что они намного чище и быстрей. Создавая код, мы ленимся его оптимизировать не только с точки зрения размера, но и с точки зрения скорости работы. Глядя на такие творения, хочется ругаться, только программа от этого лучше не станет.

Хакеры далеко не ушли. Если раньше, глядя на программиста или хакера, создавался образ прокуренного, заросшего и немытого молодого человека, то сейчас это цифровое существо, залитое пивом "Балтика" по самые уши, за которого все выполняют машины. Вам медсестра в поликлинике не говорила, что у вас вместо крови одно только пиво льется? Нет, я ничего против пива не имею, я и сам его люблю, но надо же и меру знать.

Все это — деградация по методу Microsoft! Мы берем в руки мышку и начинаем тыкать ею где попало, забывая про клавиатуру и "горячие" клавиши. Я считаю, что надо бороться с этим. В последнее время меня самого посещает такая лень, что я убираю клавиатуру, запускаю экранную клавиатуру и начинаю работать только мышью. Осталось только покрыть мое тело шерстью и посадить в клетку к таким же ленивым шимпанзе.

Не надо тратить большие деньги на модернизацию компьютера!!! Начните улучшения с себя. Давайте оптимизируем свою работу и то, что мы делаем, и тогда компьютер заработает намного быстрее.

Изначально этот фрагмент книги задумывался как рассказ об оптимизации кода программ, но впоследствии я перенес в него свой "труд", который можно найти и в Интернете на моем сайте, потому что оптимизировать надо все. Мы будем говорить о теории оптимизации, а ее законы действуют практически везде. По тем же законам вы можете оптимизировать свой распорядок дня, чтобы успевать все сделать, и свою ОС, чтобы она работала быстрей. Но основа все же будет относиться к коду программ.

Начну я с законов, которые работают не только в программировании, но и в реальной жизни. Ну а напоследок я оставлю только то, что может пригодиться при оптимизации кода.

Закон № 1

Оптимизировать можно все и везде. Даже там, где вам кажется, что все и так работает быстро, можно сделать еще быстрее.

Это действительно так. И этот закон очень сильно проявляется в программировании. Идеального кода не существует. Даже простую операцию сложения 2 + 2 тоже можно оптимизировать. Чтобы достичь максимального результата, нужно действовать последовательно и желательно в описанном далее порядке.

Помните, что любую задачу можно решить хотя бы двумя способами (или больше), и ваша задача — выбрать наилучший метод, который обеспечит желаемую производительность и универсальность.

Несмотря на то, что код всегда можно написать еще лучше, и он будет еще быстрее, должен быть какой-то предел, когда следует остановиться и сказать себе: "Теперь программа работает идеально". Если этого не сделать, то оптимизацией можно заниматься вечно, и мир никогда не увидит вашу программу. Чтобы не завязнуть в вечной оптимизации, лучше сначала написать программу и только потом заняться улучшением слабых мест, но об этом мы еще поговорим в других законах.

Закон № 2

Первое, с чего нужно начинать, — это поиск самых слабых и медленных мест программы. Зачем начинать оптимизацию с того, что и так работает достаточно быстро или, по крайней мере, приемлемо! Если вы будете оптимизировать сильные места, то можете нарваться на неожиданные конфликты. Да и эффект от затраченных усилий будет минимален.

Тут же я вспоминаю пример из собственной жизни. Как-то в 1996-м году меня посетила одна невероятная идея — написать собственную игру в стиле Doom. Я не собирался ее делать коммерческой, а хотел только потренировать свои мозги на сообразительность. Четыре месяца невероятного труда, и нечто похожее на движок уже было готово. Я создал один голый уровень, по которому можно было перемещаться, и с чувством гордости побежал по коридорам.

Никаких монстров, дверей и атрибутики на нем не было, а тормоза ощущались достаточно значительные. Да, у меня был тогда всего лишь Pentium 100 с видеокартой от S3 без каких-либо ускорений, но и графика движка не отличалась большим качеством, разрешение было небольшим. Тут я представил себе, что будет, если добавить монстров и атрибуты, да еще и наделить все это AI (artificial intelligence, искусственный интеллект)... Вот тут чувство собственного достоинства поникло. Кому нужен движок, который при разрешении 320×200 (тогда это было круто!) в "голом" виде тормозит со страшной силой? Вот именно...

Понятное дело, что мой виртуальный мир нужно было оптимизировать. Целый месяц я бился над кодом и "вылизывал" каждый оператор моего движка. Результат — мир стал прорисовываться на 10% быстрей, но тормозить не перестал. И тут я увидел самое слабое место — вывод на экран. Мой движок просчитывал сцены достаточно быстро, а пробоиной был именно вывод изображения. Тогда еще не было шины AGP, и я использовал простую PCI-

видеокарту от S3 с 1 Мбайт видеопамяти. Пару часов колдовства, и я выжал из PCI все возможное. Откомпилировав движок, я снова погрузился в свой виртуальный мир. Одно нажатие клавиши "вперед", и я очутился у противоположной стены. Никаких тормозов, сумасшедшая скорость просчета и моментальный вывод на экран.

Как видите, моя ошибка была в том, что вначале я неправильно определил слабое место своего движка. Я месяц потратил на оптимизацию математики, и что в результате? Мизерные 10% прироста в производительности. Но когда я реально нашел слабое звено, то смог повысить производительность в несколько раз.

Именно поэтому я говорю, что надо начинать оптимизировать только со слабых мест. Если вы ускорите работу самого слабого звена вашей программы, то, может быть, и не понадобится ускорять другие места. Вы можете потратить дни на оптимизацию сильных сторон и увеличить производительность только на 10% (что может оказаться недостаточным), или несколько часов на улучшение слабой части и получить улучшение в 10 раз!.

Слабые места компьютера

Меня поражают люди, которые гонятся за мегагерцами процессора и сидят на доисторической видеокарте от S3, жестком диске на 5400 оборотов и с 32 Мбайт памяти. Посмотрите в корпус своего компьютера и оцените его содержимое. Если вы увидели, что памяти у вас не более 256 Мбайт, то встаньте и громко произнесите: "Уважаемый DIMM, команда выбрала вас. Вы — сегодня самое слабое звено и должны покинуть мой компьютер". После этого, покупаете себе 512 Мбайт, а лучше целый гигабайт оперативной памяти, и наслаждаетесь ускорением работы Delphi, Photoshop и других "тяжелых" программ.

В данном случае, наращивание сотни мегагерц у процессора даст гораздо меньший прирост в скорости. Если вы используете "тяжелые" приложения при нехватке памяти, то процессор начинает тратить слишком много времени на загрузку и выгрузку данных. Ну а если в вашем компьютере достаточно оперативной памяти, то процессор уже занимается только расчетами и не расходуется по лишним загрузкам-выгрузкам. Получается, что память позволяет избавить процессор от лишних расчетов и повысить его коэффициент полезного действия, и он за одну секунду сможет выполнить больше полезных операций.

То же самое с видеоадаптером. Если видеокарта у вас слабенькая, то процессор будет просчитывать сцены быстрей, чем они будут выводиться на экран. А это грозит простоями и минимальным приростом производительности.

Закон № 3

Следующим шагом вы должны разобрать все операции по косточкам и выяснить, где происходят регулярно повторяющиеся операции. Начинать оптимизацию нужно именно с них.

Опять начнем рассмотрение этого закона с программирования. Допустим, что у вас есть следующий код (приведена просто логика, а не реальная программа):

```
1. A:=A*2;
2. B:=1;
3. X:=X+B;
4. B:=B+1;
5. Если B<100, то перейти на шаг 3;
```

Любой программист скажет, что здесь слабым местом является первая строка, потому что там используется умножение. Это действительно так. Умножение всегда выполняется дольше, и если заменить его на сложение (`A:=A+A`) или еще лучше на сдвиг, то вы выиграете пару тактов процессорного времени. Но это только пару тактов и для процессора это будет незаметно.

Теперь посмотрим еще раз на наш код. Больше ничего не видите? А я вижу. В этом коде используется цикл: пока `B<100`, будет выполняться операция `X:=X+B`. Это значит, что процессору придется выполнить 100 переходов с шага 5 на шаг 3. А это уже не мало. Как можно здесь что-то оптимизировать? Очень легко. Здесь у нас внутри цикла выполняются две строки: 3 и 4. А что если мы внутри цикла размножим их 2 раза:

```
2. B:=1;

3. X:=X+B;
4. B:=B+1;

5. B:=X+B;
6. B:=B+1;

7. Если B<100, то перейти на шаг 3;
```

Здесь я исходный цикл превратил в более маленький. Вторую и третью операцию я повторил два раза. Это значит, что за один проход цикла я выполню два раза строки 3 и 4 и только после этого перейду на строку 3, чтобы

повторить операцию. Такой цикл уже нужно повторить только 50 раз (потому что за один раз выполняются два действия). Это значит, что я сэкономил 50 операций переходов. Неплохо? А это уже несколько сотен тактов процессорного времени.

А что если внутри цикла написать строки 2 и 3 десять раз. Это значит, что за один проход цикла строки 2 и 3 будут вычисляться 10 раз, и мне понадобится повторить такой цикл только 10 раз, чтобы получить в результате 100. А это уже экономия 90 операций переходов.

Недостаток этого подхода — увеличился код программы, зато повысилась скорость, и очень значительно. Этот подход очень хорош, но им не стоит злоупотреблять. С одной стороны увеличивается скорость, а с другой — увеличивается размер. А большой размер — это враг любой программы. Используйте разворачивание циклов только там, где это действительно нужно, т. е. в тех местах программы, где цикл выполняется долго.

Еще один недостаток разворачивания циклов — уменьшается читабельность кода. В будущем вам будет сложнее понять, что же тут происходит, а сторонний программист, который будет читать ваш код, может вообще упустить из виду то, что цикл развернут. Если вы работаете в команде, то обязательно комментируйте развернутые циклы и предупреждайте в них своих коллег о выполненной оптимизации.

В любом деле главное — разумная достаточность. Чем больше вы увеличиваете код ради оптимизации скорости, тем меньше результирующий эффект от оптимизации.

В жизни таких примеров намного больше. Любую циклическую операцию можно оптимизировать. Хотите пример? Пожалуйста. Допустим, у вашего провайдера Интернета есть несколько телефонов доступа. Вы ежедневно перезваниваете на каждый из них в ожидании найти свободный. Начинающий тут же скажет, что провайдер обязан оптимизировать свои пулы модемов в один, чтобы не надо было трезвонить по всем номерам сразу. Но опытный пользователь должен знать, что не у каждого пользователя хорошая связь с любой телефонной станцией города. Поэтому провайдеры держат пулы на разных станциях, чтобы вы могли выбрать тот, с которым у вас связь лучше. Поставьте программу-дозвонщик (таких сейчас полно в Интернете), которая сама будет перебирать номера телефонов. С переходом на высокоскоростные соединения, такие как ADSL, этот пример теряет свою наглядность, но тот, кто работал с dial-up, меня поймет.

А теперь другой пример — вам досталась карточка на 1 час какого-то нового провайдера. Заносить ее в программу дозвона не имеет смысла, потому что вы можете больше никогда не позвонить ему. Из-за этой одноразовой операции вам придется перенастраивать свой дозвонщик на нового провайдера

и потом обратно, а выигрыш практически нулевой, потому что пока вы меняете настройки, уже можно было дозвониться стандартными средствами Windows. Отсюда сразу же напрашивается вывод — нужно правильно выбирать средства для выполнения необходимых задач и не оптимизировать операции, которые не будут выполняться в будущем или будут выполняться очень редко.

Закон № 4

(Этот закон — расширение предыдущего.)

Оптимизировать одноразовые операции — это только потеря времени. Сто раз подумайте, прежде чем начать мучиться с редкими операциями.

Несколько лет назад в Интернете я прочитал рассказ "Записки жены программиста" (**http://www.exler.ru/novels/wife.htm**). Очень даже "некислый" и жизненный рассказ. Когда я его читал, у меня было ощущение, что его написала моя жена. Слава "Красной Шапочке", что она на такую подлость не способна. Так вот там была очень интересная ситуация, которую мы сейчас рассмотрим.

Очаровательная девушка выходит замуж за программиста, и им надо разослать приглашения на свадьбу. Вместо того чтобы набрать их на печатной машинке, программист кричит, что он крутой, и пишет специальную программу. Ее написание заняло один день и столько же ее отладка.

Главная ошибка — неправильная оптимизация своего труда. Легче набрать шаблон в любом текстовом редакторе и только потом менять фамилии приглашенных на этот "траурный" день (это я сужу по себе и надеюсь, что моя жена этого не прочитает). Но даже если нет текстового редактора, писать программу действительно нет смысла. Затраты большие, а пользоваться ею будешь только один раз. Здесь действительно легче будет даже набрать на печатной машинке.

Получается, что одноразовые операции оптимизировать просто бессмысленно. Затраты в этом случае себя не окупают, поэтому не стоит тратить свои нервы на этот бессмысленный труд.

В самом начале этого раздела я раскритиковал компьютерщиков, как людей, которые ленятся хоть что-нибудь делать. Так вот именно здесь вы можете проявлять свою врожденную леность в полном объеме. В данном случае крутым считается не тот, кто целый день промучился и ничего не добился, а тот, кто выполнил свою работу наиболее быстро и эффективно. И эти две вещи путать нельзя.

Закон № 5

Нужно знать "внутренности" компьютера и принципы его работы. Чем лучше вы знаете, каким образом компьютер будет выполнять ваш код, тем лучше вы сможете его оптимизировать.

Этот закон относится только к программированию. Тут трудно привести полный набор готовых решений, но некоторые приемы я постараюсь описать.

- Старайтесь поменьше использовать вычисления с плавающей запятой. Любые операции с целыми числами выполняются в несколько раз быстрее.
- Операции умножения и тем более деления так же выполняются достаточно долго. Если вам нужно умножить как-то число на 3, то для процессора будет легче три раза сложить одно и то же число, чем выполнить умножение.

  А как же тогда экономить на делении? Вот тут нужно знать математику. У процессора есть такая операция, как сдвиг. Вы должны знать, что процессор "думает" в двоичной системе, и числа в компьютере хранятся именно в ней. Например, число 198 для процессора будет выглядеть как 11000110. Теперь посмотрим, как работают операции сдвига.

  Сдвиг вправо — если сдвинуть число 11000110 вправо на одну позицию, то последняя цифра исчезнет и останется только 1100011. Теперь введите это число в калькулятор и переведите его в десятичную систему. Ваш результат должен быть 99. Как видите — это ровно половина числа 198. Вывод: когда вы сдвигаете число вправо на одну позицию, то вы делите его на 2.

  Сдвиг влево — возьмем то же самое число 11000110. Если сдвинуть его влево на одну позицию, то с правой стороны освободится место, которое заполняется нулем — 110001100. Теперь переведите это число в десятичную систему. Должно получится 396. Что оно вам напоминает? Это 198 умноженное на 2.

  Вывод: когда вы сдвигаете число вправо, то вы делите его на 2; когда сдвигаете влево, то умножаете его на 2. Так что используйте эти сдвиги везде, где возможно, потому что сдвиги работают в десятки раз быстрее умножения и деления. Но тут необходимо учитывать одну особенность — результат деления всегда будет целочисленным. А что будет, если сдвинуть вправо число 99 (в двоичной системе это 1100011)? Последняя единица исчезает и результат будет 110001, что соответствует числу 49 в десятичной системе. Правильный результат должен быть 49,5, но во время сдвига дробная часть исчезает.

  Итак, сдвигать лучше всего целые числа и только тогда, когда результат тоже должен быть целым. В принципе, с помощью сдвигов можно реализовать и работу с дробной частью, но эту тему мы опустим.

- При создании процедур не обременяйте их большим количеством входных параметров. Перед каждым вызовом процедуры ее параметры помещаются в специальную область памяти (стек), а после входа изымаются оттуда. Чем больше параметров, тем больше расходы на общение со стеком.

  Тут же нужно сказать, что вы должны действовать аккуратно и с самими параметрами. Не вздумайте пересылать процедурам переменные, которые могут содержать данные большого объема в чистом виде. Лучше передать адрес ячейки памяти, где хранятся данные, а внутри процедуры работать с этим адресом. Вот представьте себе ситуацию, когда вам нужно передать текст размером с томик романа "Война и мир"... Перед входом в процедуру программа попытается вогнать все это в стек. Если вы не увидите его переполнение, то задержка точно будет значительная. Намного лучше передать указатель на память, где вы уже расположили томик книги, и пусть процедура работает с этими данными.

- В самые критичные моменты (как, например, вывод на экран) можно воспользоваться языком ассемблера. Даже встроенный в Delphi или C++ ассемблер намного быстрее штатных функций языка. Хотя, все зависит от вашего знания ассемблера. Если вы его не знаете, то ваш результат будет хуже. Ну а если скорость в каком-то месте уж слишком критична, то код ассемблера можно вынести в отдельный модуль. Там его нужно откомпилировать с помощью компиляторов TASM или MASM и подключить к своей программе.

  Ассемблер — достаточно быстрая и компактная вещь, но писать большой проект только на нем — это очень сложно. Поэтому я не советую им увлекаться, и использовать его только в самых критичных для скорости местах.

Закон № 6

Для сложных расчетов можно заготовить таблицы с заранее рассчитанными результатами и потом использовать эти таблицы в режиме реального времени.

Когда появился первый Doom, игровой мир поразился качеству графики и скорости работы. Это действительно был шедевр программистской мысли, потому что компьютеры того времени не могли рассчитывать трехмерную графику в реальном времени. В те годы еще даже и не думали о 3D-ускорителях, и видеокарты занимались только отображением информации и не выполняли никаких дополнительных расчетов.

Как же тогда программистам игры Doom удалось создать трехмерный мир? Секрет прост, как и все в этом мире. Игра не просчитывала сцены, все сложные математические расчеты были рассчитаны заранее и занесены в

отдельную базу, которая загружалась при старте программы. Конечно же, занести все возможные результаты нельзя, поэтому база хранила основные результаты. Когда нужно было получить расчет значения, которого не было в заранее рассчитанной таблице, то бралось наиболее приближенное число. Таким образом, Doom получил отличную производительность и достаточное качество 3D-картинки.

С выходом программы Quake игровой мир опять поразился качеству освещения и теней в сценах виртуального мира Quake. Расчет света — очень сложная задача, не говоря уже о тенях. Как же тогда программисты игры смогли добиться такого качества сцен и в то же время высокой скорости работы игры? Ответ опять будет таким же — за счет таблиц с заранее рассчитанными значениями.

Через некоторое время игровая индустрия поразилась еще больше. Когда вышел Quake 3, в котором освещение рассчитывалось в реальном времени, то его мир оказался немного неестественным, и даже Half-Life, который вышел позже и на движке старого Quake, выглядел намного естественнее и привычнее. Это получилось в результате того, что мощности компьютера не хватило для полных расчетов в реальном времени, а округления и погрешности пошли не на пользу игровому окружению.

Закон № 7

Лишних проверок не бывает.

Чаще всего оптимизация может привести к нестабильности исполняемого кода, потому что для увеличения производительности некоторые убирают ненужные на первый взгляд проверки. Запомните, что ненужных проверок не бывает! Если вы думаете, что какая-то нестандартная ситуация может и не возникнуть, то она не возникнет только у вас. У пользователя, который будет запускать вашу программу, может произойти все, что угодно. Он обязательно нажмет на то, на что не нужно или введет неправильные данные.

Обязательно делайте проверки всего того, что вводит пользователь. Делайте это сразу же, и не ждите, когда введенные данные понадобятся. Хотя нет, подождать можно, но только в том случае, если данные действительно могут не понадобиться, и если они будут использоваться единожды. Если введенный пользователем параметр действительно окажется ненужным в расчетах, то вы сэкономите лишнюю проверку, но если он будет использоваться дважды, то проверка перед использованием также будет срабатывать дважды, что абсолютно неэффективно.

Не делайте проверки в цикле, а выносите за его пределы. Любые лишние операторы `if` внутри цикла очень сильно влияют на производительность,

поэтому, по возможности, проверки нужно делать до или после цикла. Это как бы продолжение закона 3, который требует оптимизацию повторяющихся операций.

Циклы — это слабое место любой программы, поэтому оптимизацию надо начинать именно с них и стараться не использовать в них лишние проверки. Внутри циклических операций не должно выполняться ничего лишнего — ведь это будет выполнено много раз! Экономия одной операции внутри цикла, который выполняется 100 раз, равносильна экономии ста операций.

Закон № 8

Лучший оптимизатор — быстрый алгоритм.

Как бы вы не пытались оптимизировать код, если алгоритм изначально медленный, то вы никогда не добьетесь оптимальной скорости выполнения. Одних только алгоритмов сортировок существует множество, и каждый из них может быть оптимален для определенного случая, а некоторые могут работать только при конкретных условиях, но выполняют свою задачу с максимальной скоростью.

Поэтому, если ваш код после всех оптимизаций работает очень медленно, нужно подумать о том, чтобы оптимизировать алгоритм вычислений. Вообще-то оптимальный алгоритм нужно выбирать еще на этапе проектирования, но, как показывает практика, это далеко не всегда возможно, особенно в нашей стране.

Мы не привыкли писать задания и производить подготовительное проектирование программ. Все делается нахрапом, да и пользователи далеко не всегда могут правильно сформулировать то, что они хотят. Поэтому приходится сначала набросать алгоритм программы в стиле "лишь бы работало", а потом оптимизировать код. И вот тут, когда уже пользователь увидел, что он хочет, а вы поняли свою цель, необходимо обдумать оптимальный алгоритм и переписать свой код заново. Да, это скучно и нудно, а что делать?

Не хотите переписывать? Заранее проектируйте программу, утверждайте и анализируйте. Набросайте алгоритм на бумаге или с помощью диаграмм UML, после этого вероятность выбора неверного алгоритма значительно уменьшается.

Итог

Если вы прочитали этот раздел внимательно, то можете считать, что с основами оптимизации вы уже знакомы. Но это только основы, и тут есть куда

развиваться. Я бы мог рассказать больше, но не вижу особого смысла, потому что оптимизация — это процесс творческий, и в каждой отдельной ситуации к нему можно подойти с разных сторон. И все же, те законы, которые я сегодня описал, действуют в 99,9% случаев.

Если вы хотите познать теорию оптимизации в программировании более глубоко, то вам нужно больше изучать принципы работы процессора и операционных систем. Главное, что законы вы уже знаете, а остальное придет со временем.

Во всех своих книгах я стараюсь уделять достаточно большое внимание оптимизации. Например, в книгах по DirectX [2], [3], [6] я описывал оптимизацию графики, в Transact-SQL [5] — оптимизацию баз данных, а в книге по PHP [4] большое внимание отведено оптимизации Web-приложений.

1.10. Приемы оптимизации

Мы уже рассмотрели основные законы оптимизации и даже затронули некоторые методы, с помощью которых вы сможете повысить производительность своих приложений. В этом разделе мы двинемся дальше и разберем еще несколько методов повышения производительности. Мы рассмотрим несколько реальных примеров, и я надеюсь, что они пригодятся вам во время разработки собственных программ.

1.10.1. Быстрый старт

Прежде чем начать выполнение, исполняемый код программы и необходимые ресурсы должны загрузиться в память. Я видел программы, где исполняемый файл занимал десятки мегабайт, и этот файл перед выполнением загружается в память! Задержка перед появлением главного окна программы может быть очень серьезной, и это неприятно.

Запуская программу, пользователь хочет начать работать максимально быстрее и может нервничать от излишних ожиданий. Лично меня раздражает ожидание загрузки Delphi 2006. Когда тебя осенила какая-то идея или хочется что-то быстро подправить/улучшить, то абсолютно не хочется ждать пять минут до окончания загрузки среды разработки.

Как можно повысить скорость загрузки программы? Ну, конечно же, загружать и инициализировать только то, что необходимо для начала работы. Проекты на Delphi создаются так, что все формы инициализируются автоматически во время загрузки, но далеко не все из них будут реально использоваться. Например, окно с информацией о программе большинство поль-

зователей никогда не вызывает, а в больших проектах с большим количеством функций пользователи работают максимум с 50% диалоговых окон.

На самом деле, в большинстве программ для начала полноценной работы достаточно выделить память только для основного окна. Все остальное можно инициализировать при первом обращении. На рис. 1.16 вы можете увидеть окно настройки форм, которые должны создаваться автоматически во время запуска приложения.

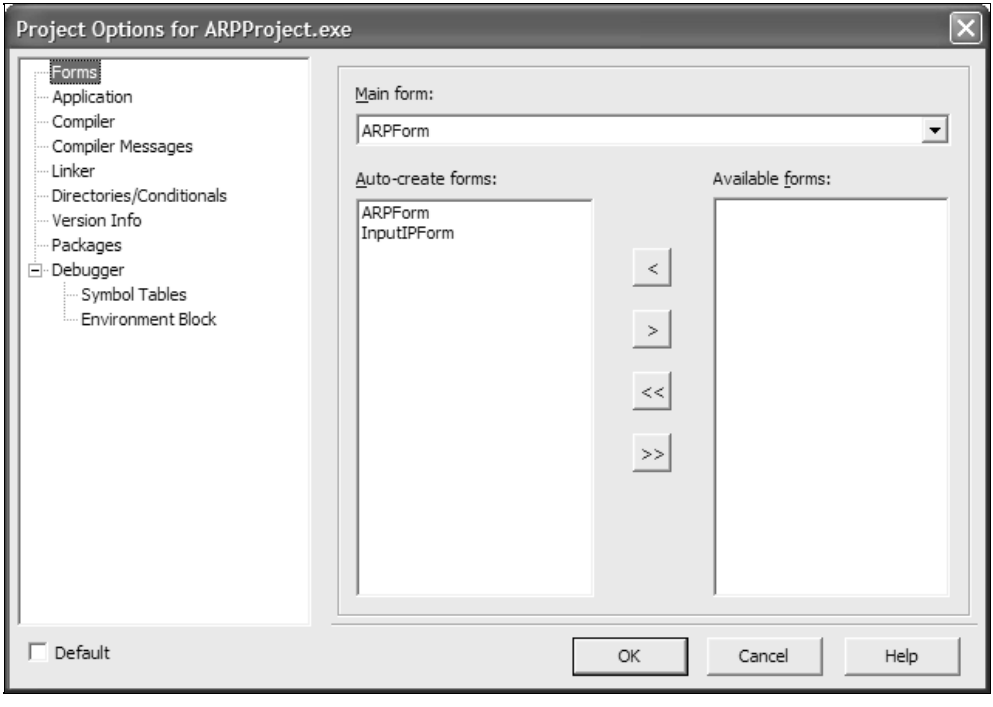


Рис. 1.16. Настройка автоматически создаваемых форм

Тут следует уйти немного в сторону и определиться — нужно ли оставлять в памяти созданную форму после ее закрытия? На первый взгляд, лучше не тратить время на освобождение ресурсов, а освободить все сразу после закрытия окна. С одной стороны, это логично, ведь окно может пригодиться еще раз, и тратить время на лишнее освобождение и инициализацию накладно. Но с другой стороны, лишние расходы памяти тоже никому не нужны.

Я бы рекомендовал хранить постоянно в памяти только те окна, которые действительно используются очень часто. Если в среднем форма будет ини-

циализироваться редко во время работы, то хранить в памяти постоянно крупный объект не имеет смысла.

Мы говорим "очень часто" или "редко", но сколько это в реальном выражении — один, два, три или более раз, — сказать сложно, и тут все решать вам.

1.10.2. Динамические библиотеки

Чтобы сократить размер основного исполняемого файла, можно поступить классическим методом — использовать динамические библиотеки. В больших проектах в основном файле необходимо хранить только главные функции, которые необходимы очень часто и только для основного окна. Весь остальной функционал можно перенести в динамические библиотеки и подгружать по мере необходимости.

Но перенос кода в dll-файлы может отрицательно сказаться на производительности, а точнее, на загрузке программы. Следующий пример показывает, как объявить функцию из динамической библиотеки и при этом навести статическую связь:

```
function FunctionName(Param1: Type) : Boolean; stdcall; external
 'library.dll' index 1;
```

Здесь объявляется функция с именем `FunctionName`, которая расположена в файле library.dll. После этого функцию можно использовать, как будто она описана в основном проекте и не вынесена в отдельный файл. При таком объявлении Delphi скомпилирует код так, что библиотека будет загружаться во время запуска программы, а это сведет преимущество dll-файлов на нет, или даже уменьшит скорость. ОС нужно сначала загрузить основной исполняемый файл, тут же загрузить динамические библиотеки, создать таблицы экспорта, связать функции и т. д.

Единственное преимущество от такой стыковки — функция реализована в отдельном файле, и ее можно использовать из различных программ. Получается принцип: один раз реализуем и много раз используем.

Чтобы библиотека не загружалась во время старта программы, необходимо объявлять функции из динамической библиотеки немного по-другому. Для начала в разделе `var` объявляем переменную типа функции или процедуры, которую мы будем экспортировать.

Например, следующий код объявляет переменную `CalcFunc`, которая равна функции, получающей в качестве параметра число и возвращающей число.

```
var
 CalcFunc=function(Summ: Integer):Integer; stdcall;
```

Теперь посмотрим, как загрузить библиотеку динамически. В листинге 1.2 показан код, в котором загружается динамическая библиотека, в ней ищется необходимая функция, функция используется для расчета, и после этого библиотека освобождается. Что значит освобождается? Приложение может сказать ОС только то, что библиотека уже не нужна. При этом, код dll-файла останется в памяти, если другое приложение загрузило эту же библиотеку. Память может остаться занятой и по причине того, что ОС не освобождает ее сразу.

Листинг 1.2. Функция загрузки библиотеки динамически

```
procedure TForm1.Button1Click(Sender: TObject);
var
 dllHandle:THandle;
 iResult: Integer;
begin
 // Загружаем библиотеку
 dllHandle:=LoadLibrary('test.dll');

 // Если результат равен нулю, то ничего не загружено.
 // Возможно, файл не найден
 if dllHandle=0 then
  exit;

 // Ищем адрес функции в загруженной библиотеке
 @CalcFunc:=GetProcAddress(dllHandle, 'FunctionName');

 // Если CalcFunc равна nil, то функция не найдена
 if @CalcFunc = nil then
  exit;

 // Вызов функции
 iResult:= CalcFunc(100);

 // Освобождение библиотеки
 FreeLibrary(dllHandle);
end;
```

Теперь давайте разберем подробно, что здесь происходит. Тем более, что подобные примеры мы будем использовать в данной книге для работы с WinAPI-функциями.

Для начала явно загружается dll-файл с помощью функции `LoadLibrary`. В качестве параметра нужно указать библиотеку, или полный/относительный путь к файлу (если она расположена в папке, которая недоступна программе). Если загрузка прошла успешно, то функция вернет ненулевое значение, иначе результат будет равен 0. Результат работы функции `LoadLibrary` — это указатель на загруженную библиотеку, поэтому его надо сохранить для дальнейшей работы.

Следующим этапом ищем внутри загруженной библиотеки нужную нам функцию или процедуру. Для этого используется функция `GetProcAddress`. В качестве первого параметра передается указатель на загруженную библиотеку, а второй — имя искомой функции. Если поиск увенчался успехом, то результатом будет указатель на соответствующий код. Этот указатель сохраняем в переменной, имеющей тип библиотечной функции. Если же результат равен нулю, то ничего не найдено. Это может быть из-за того, что библиотека была изменена, или мы по ошибке загрузили другой dll-файл с таким же именем.

Если все прошло успешно, то вызываем функцию через переменную.

По окончании работы с библиотечными функциями обязательно выгружаем динамическую библиотеку из памяти. Это делается функцией `FreeLibrary`, а в качестве указателя передается указатель на загруженную библиотеку.

Конечно же, если в библиотеке описано сразу несколько функций, использующихся в программе, или эти функции вызываются часто, то выгружать библиотеку сразу нет необходимости. Это можно сделать после завершения работы программы. Однажды загрузив по мере необходимости, мы можем оставить код в памяти и использовать в дальнейшем без лишних расходов на загрузку.

Чтобы не загружать библиотеку каждый раз, можно использовать логику, показанную в листинге 1.3.

Листинг 1.3. Загрузка библиотеки по мере необходимости

```
procedure TForm1.Button1Click(Sender: TObject);
var
 iResult: Integer;
begin
 if @CalcFunc = nil then
```

```
  begin
   if dllHandle=nil then
    begin
     dllHandle:=LoadLibrary('test.dll');
     if dllHandle=0 then
      exit;
    end;

   @CalcFunc:=GetProcAddress(dllHandle, 'FunctionName');
  end;

 iResult:= CalcFunc(100);
end;
```

Здесь переменная `dllHandle`, в которой хранится указатель на библиотеку, должна быть перенесена в глобальную область или быть объявленной в качестве члена класса главного окна. Вначале проверяем, если переменная функции равна нулю, то мы ее еще не использовали, и нужно найти ее в библиотеке. После этого проверяем, равна ли нулю переменная `dllHandle`. Возможно, библиотека уже была загружена в другом месте программы, и тогда нет смысла делать это повторно. Достаточно сразу вызвать `GetProcAddress`.

Если вы будете использовать примеры, описанные выше, то dll-файл не будет загружаться автоматически при старте программы, а только тогда, когда это нужно, и так можно повысить скорость запуска основного файла.

Если функции какой-то библиотеки используются в программе часто, то имеет смысл загружать ее на этапе старта программы, чтобы в последующей работе не было лишних затрат на загрузку и выгрузку dll-файла. Если же функции нужны редко, то стоит написать немного лишнего кода для явной загрузки.

Помните, что dll-файлы нужны не только для увеличения скорости загрузки запускных файлов, но и для разделения одних и тех же функций между несколькими приложениями.

1.10.3. Цепочки

Допустим, что у нас в проекте есть две формы: главная (`MainForm`) и дочерняя (`DitailForm`). На `DitailForm` расположен компонент `ComboBox`, содер-

жащий список чисел. Нам нужно в форме MainForm вывести на экран список из ComboBox, в котором к каждому элементу будет прибавлено число, равное количеству элементов. Эта проблема решается следующим образом:

```
procedure MainForm.SomeProc();
var
 i:Integer;
begin
for i:=0 to DitailForm.ComboBox1.Items.Count-1 do
  Canvas.TextOut(10, i*16,
    StrToInt(DitailForm.ComboBox1.Items[i])+
    DitailForm.ComboBox1.Items.Count);
end;
```

Здесь запускается цикл перебора всех элементов из выпадающего списка, в котором выводим на экран i-й элемент с добавлением количества элементов.

На первый взгляд тут оптимизировать нечего. Разложить цикл невозможно, потому что мы не знаем количество элементов, а ведь там может быть вообще одно значение. Но мы же говорили, что оптимизировать можно все, главное — подойти к вопросу с правильной стороны.

Посмотрим, как определяется количество элементов в выпадающем списке. Для этого нужно взять форму DitailForm, найти на ней компонент списка ComboBox1, затем надо обратиться к элементам списка Items и только потом определить количество Count. Посчитайте, сколько нужно выполнить операций? И эта операция выполняется несколько раз. Второй раз мы обращаемся к количеству элементов в цикле. Сейчас даже не будем обращать внимание на цикл, но такое длинное обращение выполняется не один раз, и это не может не волновать.

Если бы мы писали этот код на ассемблере, то количество элементов можно было бы сохранить в регистре, и это значительно повысило бы производительность. Операции с регистрами выполняются быстрее, чем с памятью. Но компилятор Delphi может не сделать этого и не увидеть такой явной оптимизации. Компилятор может сформировать код, в котором перед каждым обращением к свойству Count списка число копируется в регистр EAX, а это лишние затраты. Мы не можем этим управлять, но мы должны хотя бы как-то ускорить доступ, и это можно сделать следующим образом:

```
var
 count, i :Integer;
```

```
begin
 iCount := DitailForm.ComboBox1.Items.Count;
 for i:=0 to iCount-1 do
  begin
   Canvas.TextOut(10, i*16, StrToInt(DitailForm.ComboBox1.Items[i])+
         iCount);
  end;
end;
```

Здесь мы завели только одну лишнюю переменную, но зато сократили обращение к длинной цепочке до одного раза. В самом начале мы сохраняем значение свойства `Count` в локальной переменной `count` и работаем уже с ней. Таким образом, производительность повышается достаточно сильно, особенно если цикл будет выполняться сотню раз.

Теперь немного изменим задачу. Допустим, что нам надо заполнить элементы списка следующим образом:

```
var
 i:Integer;
begin
 for i:=0 to DitailForm.ComboBox1.Items.Count-1 do
  DitailForm.ComboBox1.Items[i]:=IntToStr(i);
end;
```

Здесь две длинных цепочки, и они разные. Но, несмотря на это, мы можем сократить цепочку, таким образом упростить жизнь компилятору и создать более быстрый код:

```
var
 i:Integer;
 sl:TStrings;
begin
 sl:= DitailForm.ComboBox1.Items;
 for i:=0 to sl.Count-1 do
  sl[i]:=IntToStr(i);
end;
```

Здесь мы работаем со свойством `Items` компонента `ComboBox`. Это свойство имеет тип `TStrings`. Мы просто сохраняем указатель на `Items` в локальной переменной и работаем уже с ней. Таким образом, благодаря локальной пе-

ременной длинная цепочка `DitailForm.ComboBox1.Items` превратилась в короткую переменную с именем `sl`.

Вот парадокс: мы добавили в процедуре одну строку, а в результате получили экономию в исполняемом файле. При использовании длинных цепочек результирующий исполняемый файл всегда будет больше, и будет выполнять больше команд. Когда одна и та же длинная цепочка используется 10 раз в течение большой процедуры, сокращение может сэкономить очень много процессорных тактов.

Когда цепочка используется только два раза, то экономия может оказаться несущественной, ведь во втором случае нам нужна была локальная переменная, а ее еще нужно создать в стеке.

Да, компиляторы совершенствуются и пытаются самостоятельно оптимизировать подобные места, но лучше все же им помогать, явно сокращая цепочку с помощью локальной переменной.

Цепочки можно сохранять не только в локальных, но и глобальных переменных. Глобальный разрыв лучше всего показывать на примере графической программы и компонента `TImage`.

Допустим, что вы пишете графический редактор, и в качестве хранилища для растрового изображения используется компонент `TImage`. В этом случае, при обращении к битовой матрице и ее функциям/свойствам во всех процедурах придется писать `Image1.Picture.Bitmap`. При входе в каждую процедуру разрыв такой цепочки не принесет желаемого результата, поэтому намного лучше будет разорвать цепь только один раз.

В случае с графическим редактором вы должны создать в объявлении формы переменную, например `bImage` типа `TBitmap`, и при создании картинки сохранить в ней значение `Image1.Picture.Bitmap`. Теперь можно во всех процедурах, где нужно обратиться к функциям растрового изображения, использовать нашу переменную вместо длинной цепочки компонента `TImage`.

Данный метод очень хорош, но пользоваться им надо очень аккуратно, потому что в определенный момент такая переменная может стать недоступной или вы случайно можете изменить ее значение. Чтобы ее легче было контролировать, я рекомендую описывать переменную в разделе `private`, чтобы ее нельзя было изменить в других формах вашего проекта, и легче было контролировать ее содержимое.

1.11. Ассемблер и Delphi

Если оптимизация зашла в тупик, и вы уже не видите, что еще можно сделать для повышения производительности, можно прибегнуть к крайним ме-

рам. К таким мерам я отношу использование ассемблера. Только переписывать нужно не всю программу, а лишь те участки, которые выполняются очень медленно.

Если после переписывания кода скорость не очень выросла, я бы порекомендовал вернуть код на Delphi и использовать его. Дело в том, что код на языке ассемблера поддерживать намного сложнее, а раз производительность не удалось увеличить, то зачем создавать себе сложности.

Компилятор Delphi очень часто вставляет ненужные операции или использует неоптимальные решения для достижения универсальности. Это связано с тем, что во время компиляции не всегда можно узнать, чего именно мы хотим добиться. Когда код пишется на ассемблере, вы можете построить логику более оптимально и убрать все лишнее.

Ассемблер позволяет не только оптимизировать логику, но и максимально эффективно использовать современные инструкции процессоров, такие как MMX, SSE, 3DNow от AMD и т. д. Дело в том, что компилятор вашего языка программирования может не знать о современных инструкциях из-за своей старости, или разработчики просто не решились внедрять новые инструкции. В этом случае, единственный способ воспользоваться всеми современными "вкусностями" — это воспользоваться ассемблером или другим компилятором, который знает современные инструкции и умеет их использовать.

Да, одна программа может быть написана на разных языках. Дело в том, что большинство компиляторов создают только промежуточные файлы, но не исполняемые файлы. В Delphi компилятор создает файлы dcu, а в C++ и ассемблере — файлы obj. После этого запускается компоновщик, который из промежуточных файлов создает непосредственно исполняемый файл.

В Delphi весь процесс сборки спрятан от нас, но с помощью нехитрых действий можно заставить компоновщик использовать необходимый нам obj-файл.

Как можно написать код на языке ассемблера? Это не так уж сложно, если вы знаете хотя бы основные операторы этого языка. Но для того чтобы реально увеличить скорость благодаря ассемблеру, необходимо быть экспертом и знать все его преимущества и недостатки.

Язык ассемблера можно использовать в двух вариантах — встроенный в оболочку и внешний. Рассмотрим использования обоих видов.

1.11.1. Использование встроенного ассемблера

В Delphi есть встроенный ассемблер, с помощью которого вы можете писать код прямо в модулях pas абсолютно в любой момент. При этом для компиляции данного кода в машинный будет использоваться встроенный ассемблер.

Код на языке ассемблера в pas-модулях заключается между операторами `asm` и `and`. Например, код из листинга 1.4 показывает, как можно сложить значения двух переменных, используя встроенный ассемблер.

Листинг 1.4. Сложение встроенным ассемблером

```
procedure TSomeForm.SummWithAsm();
var
 iResult, i, j:DWORD;
begin
 i := 10;
 j := 20;

 asm
  MOV EAX, i
  ADD EAX, j
  MOV iRes, EAX
 end;

 edResult.Text := IntToStr(iResult);
end;
```

Ключевое слово `asm` является началом блока, который заканчивается словом `end`. Между этими ключевыми словами можно писать код на языке ассемблера. Принцип идентичен началу и концу блока на языке Delphi, где началом выступает `begin`, а концом — `end`.

В данном примере используются два оператора ассемблера:

- `MOV X,Y` помещает значение `Y` в `X`. Это эквивалент присваивания в Delphi. В большинстве случаев в качестве `X` может выступать только регистр `EAX` или его более короткие варианты (`AX` или `AL`);
- `ADD X,Y` складывает два значения, а результат помещается в регистр `X`.

Итак, сначала мы помещаем содержимое переменной `i` в регистр `EAX`. Регистр — это как переменная внутри процессора. `EAX` — это регистр общего значения, который можно использовать на свое усмотрение. Вы должны знать, что результат большинства операций записывается именно в него.

После этого складываем значение регистра `EAX` и переменной `j`. А почему сразу же нельзя сложить два значения — `ADD i,j`? Насколько я знаю, в ассемблере это запрещено. Во время сложения один из операндов должен

быть регистром. Я конечно же не эксперт по ассемблеру, поэтому могу и ошибаться.

Теперь результат перемещаем из регистра `EAX` в переменную `iRes`. Так как ключевое слово `asm` заменяет `begin`, и благодаря этому можно писать целые процедуры и функции на ассемблере.

Следующий пример показывает функцию, код которой целиком состоит из инструкций на языке ассемблера:

```
function iMulWithAsm(X, Y: Integer): Longint;
asm
 MOV  EAX,X
 IMUL Y
end;
```

В данном примере нет `begin`, потому что его заменяет ключевое слово `asm`. Функция должна перемножать два числа и возвращать результат. В первой строке в регистр `EAX` помещается содержимое переменной `X`. После этого выполняется оператор `IMUL`, который умножает указанное число (в данном случае переменную `Y`) на содержимое регистра `EAX`. Результат вычисления помещается в регистр `EAX`.

Эта функция, а при выходе из нее результатом выполнения должно быть содержимое зарезервированной для данных целей переменной `Result`. В данном примере мы не задаем для этой переменной значения, так что же будет результатом? Ответ прост — содержимое регистра `EAX`, где у нас как раз и находится результат умножения. Функции в Delphi всегда возвращают результат работы именно через этот регистр.

Напоследок небольшое замечание. Вы должны учитывать, что комментарии во встроенном ассемблере надо писать в стиле Delphi. В самом языке ассемблера принято писать комментарии после точки запятой, а во встроенном варианте это недопустимо и вызовет ошибки:

```
asm
 MOV EAX,i      ;  Ошибочный комментарий
 ADD EAX, j     // Это комментарий
 MOV iRes, EAX
end;
```

На этом краткую экскурсию завершаю. Данная книга посвящена Delphi, но если вы хотите продолжить изучение ассемблера, то рекомендую купить соответствующую книгу. Я этот ассемблер полноценно использовал лет 10 назад и писал тогда еще по MS-DOS. С тех пор многое изменилось, и мои знания сильно устарели.

1.11.2. Внешний ассемблер

Встроенный ассемблер хорош, но обладает все же меньшей производительностью и ограничен возможностями среды разработки. Если вы используете старую версию Delphi, то встроенный ассемблер в ней не будет ничего знать о современных инструкциях процессоров 3DNow или SSE2. Но это поправимо, если использовать внешний ассемблер. В этом случае вы пишете модуль полностью на ассемблере, компилируете его и только подключаете получившийся в результате объектный файл к своему проекту.

Недостаток этого способа в том, что процедуру или функцию приходится полностью писать на языке ассемблера. Если при встроенном варианте можно было делать только вставки, то в данном случае операторы Delphi недопустимы, потому что компилятор ассемблера не сможет собрать такой код.

Давайте рассмотрим пример использования внешних процедур, написанных на ассемблере. Создайте файл GetNumber.asm со следующим содержимым:

```
.386
.model flat,STDCALL

.code

PUBLIC   GetNumber
GetNumber PROC
         push   ds
         mov    EAX, 10
         pop    ds
         ret
ENDP

ENDS
END
```

Здесь объявлена открытая процедура `GenNumber`, в которой выполняются следующие действия:

1. `push ds` — сохраняем содержимое регистра `ds` (указатель в сегменте данных). Для нашего примера это не обязательно, потому что мы не используем данные, но я вставил эту операцию, чтобы при написании собственных процедур на ассемблере вы не забывали сохранять этот регистр.

2. `mov EAX, 10` — помещаем в регистр `EAX` значение `10`. Мы уже говорили, что через этот регистр функции возвращают свои значения. Внешние функции не являются исключением.
3. `pop ds` — восстанавливаем регистр `ds` перед выходом.
4. `ret` — выход из процедуры.

При компиляции этого примера я использовал TASM32 фирмы Borland. Для этого в командной строке нужно выполнить команду:

```
Tasm32.exe GetNumber.asm
```

После компиляции мы получаем файл GetNumber.obj, который и будем подключать к проекту Delphi. Поместите этот файл в папку, где будет располагаться соответствующий проект.

Теперь посмотрим, как в Delphi использовать функцию `GetNumber` из файла GetNumber.obj, созданного на ассемблере. Создайте новый проект в Delphi. Чтобы компилятор знал, где искать функцию `GetNumber` и как ее нужно описать. Посмотрим на исходный код формы с необходимым описанием (листинг 1.5).

Листинг 1.5. Описание внешней функции, написанной на ассемблере

```
unit MainUnit;

interface

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

function GetNumber:integer;

type

  TAsmForm = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
```

```
  public
    { Public declarations }
  end;

var
  AsmForm: TAsmForm;

implementation

function GetNumber; external;
{$L GetNumber.obj}
{$R *.dfm}

end.
```

Перед разделом `type` стоит описание функции `GetNumber`:

```
function GetNumber:integer;
```

Теперь ее можно использовать в проекте, но компилятору надо еще сообщить, где искать саму функцию. Для этого в разделе `implementation` указаны две строки:

```
function GetNumber; external;
{$L GetNumber.obj}
```

В первой строке мы сообщаем компилятору, что функция внешняя (`external`), а вторая строка подключает файл GetNumber.obj. Подключение obj-файла схоже с подключением ресурсного dfm-файла, только здесь используется ключ `$L` или `$LINK` (подключение).

На первый взгляд описание сложное, но это в самом страшном варианте, когда необходимая функция должна использоваться до раздела `implementation`. Просто подключить файл и описать внешнюю функцию можно только в этом разделе, а если она должна использоваться раньше, то компилятор выдаст ошибку. Чтобы этих ошибок не было, приходится делать короткое описание до раздела `type`.

Если функция не используется в разделе `type`, то достаточно в разделе `implementation` написать следующие две строки объявления функции:

```
function GetNumber:Integer; external;
{$L GetNumber.obj}
```

Здесь в первой строке кода мы описываем функцию и сразу же сообщаем, что она внешняя.

Теперь внешняя функция может использоваться как родная. Посмотрим на пример использования нашей работы:

```
var
 i:Integer;
begin
 i:=GetNumber();
 ShowMessage(IntToStr(i));
end;
```

С помощью ключа `{$L Имя_файла}` можно подключать объектные файлы, скомпилированные не только на ассемблере, но и на C/C++. При подключении файлов, скомпилированных на C, проблем не будет. При использовании объектной модели C++ могут возникнуть некоторые трудности, особенно при подключении MFC.

Пока все у нас получилось хорошо и гладко, но на самом деле Delphi накладывает некоторые ограничения на подключаемые obj-файлы, и далеко не каждый из них можно подключить. Дело в том, что существуют два формата obj-файлов — OMF (поддерживается компиляторами Borland) и COFF (которые использует Microsoft). Компилятор Delphi умеет работать только с первым типом файлов.

Да, ограничение серьезное, но не совсем критичное. Дело в том, что есть утилиты, которые преобразовывают файлы из COFF-формата в OMF. К тому же, если использовать компилятор ассемблера от Microsoft (MASM), то можно указать ключ `/omf`, и тогда ничего преобразовывать не нужно будет. Выходные файлы MASM будут сразу формироваться в OMF-виде.

Помимо формата obj-файлы имеют еще следующие ограничения:

- не должно быть более 10 сегментов;
- не должно быть более 255 внешних символов;
- не более 50 локальных имен в записях `LNAMES`;
- только 32-битные файлы. Старый 16-битный формат невозможен.

Это далеко не полный список ограничений и не все из них критичны. В зависимости от версии Delphi этот список может изменяться, поэтому следует заглянуть в справку вашей версии компилятора.

Примечание

На компакт-диске в каталоге \Документация\Coding\ вы можете найти интересный документ useCobj.pdf, в котором можно прочитать о моих исследованиях при подключении объектных файлов различных компиляторов и проблемах, с которыми вы можете столкнуться.

1.11.3. Встроенный оптимизатор

Современные компиляторы достаточно умны и в большинстве случаев могут произвести оптимизацию сами. Компилятор Visual C++ от Microsoft обладает большим количеством настроек, которые влияют на оптимизацию результирующего кода. Компиляторы от Borland тоже имеют такие возможности, например, Borland C++ 5.2 имел гибкие настройки.

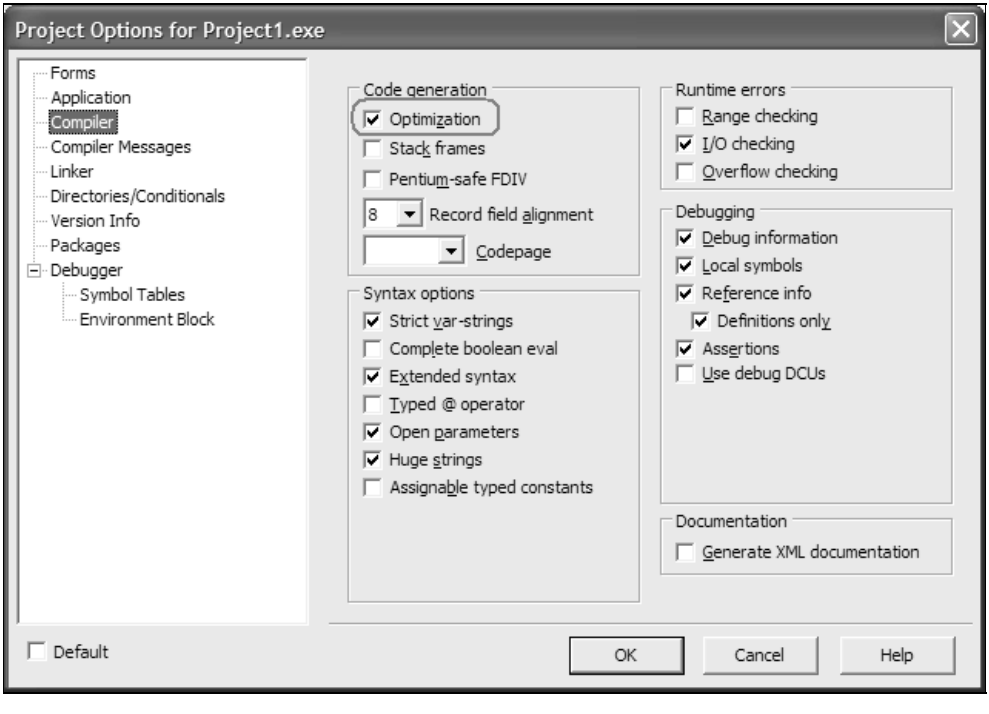


Рис. 1.17. Окно настроек проекта и флажок включения оптимизации

Если вы программировали на C++, то, скорее всего, видели эти настройки. Вероятно, вы заметили, что можно выбрать между скоростью и объемом ко-

да. Да, оптимизация скорости очень часто приводит к увеличению результирующего файла, и поэтому приходится выбирать между двумя крайностями — размер или производительность.

В Borland Delphi с настройками серьезная проблема. Здесь можно только включить встроенный оптимизатор или отключить его, но, как показывают мои опыты, в большинстве случаев этот параметр ни на что не влияет. Результирующий файл абсолютно не изменяется при включенной и отключенной оптимизации. Может, у меня версия такая странная, но не пойму, зачем тогда вообще включали эту опцию.

Итак, чтобы включить оптимизацию, выберите меню **Project | Options** и в разделе **Compiler** вы увидите среди настроек **Code generation** флажок **Optimization** (рис. 1.17).

Если включить оптимизацию, то компилятор, по идее, должен искать проблемные инструкции и генерировать максимально оптимальный код. Но каким бы не был умным компилятор, никогда не стоит на него надеяться, а лучше писать сразу же код максимально быстрым. Да, в некоторых случаях можно увидеть, что деление можно заменить сдвигом, и система способна сделать это за нас, но далеко не всегда. Ну, а если ваш алгоритм ужасен и изначально работает медленно, то никакой оптимизатор тут уже не спасет. Нужно переписывать алгоритм заново.

1.12. Безопасный код

Ошибки есть всегда и везде. Когда я учился в институте, то в коридоре на одном из стендов висело одно очень интересное изречение:

"Если вы написали программу более 5000 строк кода, в которой нет ошибок, то вы плохой программист".

Я не знаю, кому принадлежат эти строки, но тут есть доля шутки, а также немного скрытого смысла. Какой тут мог спрятаться смысл? Каждый видит его по-своему, и каждый пытается найти свое объяснение этому великому изречению. Что гениального тут вижу я, вы узнаете в конце этой главы, а сейчас мы поговорим о теории безопасного кода.

Я в основном путешествую в американском Интернете и общаюсь с американскими специалистами. Один из блогов, который я регулярно читаю, принадлежит специалисту и преподавателю по безопасности — Jungsonn (**http://www.jungsonnstudios.com**). Однажды он выложил на своем блоге размышление о том, что нужно запретить язык BASIC, потому что он портит

будущих программистов. Вот один абзац, который очень хорошо описывает базовый принцип безопасного кода:

"I grew up with BASIC on a very old computer called the Philips P2000. It looks something like a cash register and only had 16kb memory. BASIC was my first program language and it really poisoned my programming skills from day one. Many people don't understand why many website have so much vulnerabilities. The main factor and root of this is sloppy programming, and mostly the programmers don't know anything from real program languages like the C range. In C you have to be careful what you do, no remorse here like in PHP or ASP. Understanding how to program a piece of software requires skills and understanding on how computers work. If anyone lacks this kind of knowledge he is likely to program a piece of software which is vulnerable."

Я не профессиональный переводчик, но попробую перевести этот абзац для тех, кто не знаком с английским языком:

"Я вырос с языком BASIC на очень старом компьютере, который назывался Philips P2000. Он выглядит как что-то похожее на кассовый аппарат и имеет только 16 Кбайт памяти. Язык BASIC был моим первым языком программирования, и он действительно отравил мои навыки программиста с самого начала. Люди не понимают, почему масса сайтов содержит так много уязвимостей? Главный фактор и корень проблемы — это неряшливое программирование. Программисты главным образом не знают ничего о реальных языках программирования, таких как C. В C вы должны быть аккуратным в своих действиях. Чтобы понять, как программировать отдельный участок программы, требуются навыки и понимание того, как работают компьютеры. Если у кого-то отсутствуют данные знания, то он обречен писать программы, которые будут уязвимы."

Мне очень понравился этот абзац, потому что тут скрыто много интересного и весьма полезного с точки зрения безопасности. Единственное, с чем я не могу согласиться с автором, — запрещать использование таких языков, как BASIC, нельзя. Такие языки хороши даже при обучении программистов, потому что на них очень хорошо показывать логику и алгоритмы. А все остальное идеально.

Итак, используя языки высокого уровня (и даже упомянутый C/C++), многие не задумываются, как все происходит внутри, и как компьютер будет обрабатывать данный код. Чтобы код был безопасным, вы должны знать принцип работы компьютера, программ и ОС. Вы должны изучать систему и понимать, как функции выполняют поставленные задачи. В языках высокого уровня очень часто добавляют простые функции для решения сложных задач, но, не зная их принцип работы, вы не можете чувствовать себя в безопасности. Например, в языке PHP есть такие функции, как `htmlspecialchars`, `addslashes`. А вы знаете, что они не решают проблемы безопасности, и `addslashes` легко обманывается?

Если не понимать, какие именно проблемы решает функция, то безопасность программы будет очень низкой. В низкоуровневых языках таких надстроек намного меньше (если не считать сторонних библиотек, которых очень много). А если вы будете писать программу проще, минимально использовать библиотеки, то вероятность ошибки будет намного ниже.

Вот тут мы приходим еще к одной великолепной идее — код должен быть максимально простым. В простых решениях и в простой логике сложнее допустить ошибку, но если вы придумываете что-то сложное и ищете нестандартные решения, то обязательно упустите что-то из виду, а это, в свою очередь, обязательно приведет к ошибке.

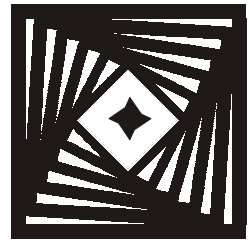
Изучайте свой компьютер и функции, которые предоставляет язык программирования. Вы должны четко понимать, как работает и то, и другое, и тогда вы сможете построить логику программы максимально безопасно.

А теперь вернемся к высказыванию, которое я привел в самом начале главы. Так почему же, если в коде нет ошибок, то вы плохой программист? Хороший и опытный кодер будет использовать современные возможности языка, выдумывать красивые решения и усложнять задачу. Плохой программист банально решит задачу простыми путями. На одном американском сайте я видел изречение, якобы принадлежащее Леонардо да Винчи: "Simplicity is the ultimate sophistication". Я понимаю, что Леонардо да Винчи сказал это на другом языке, но это изречение приводил кто-то из специалистов по безопасности. Я не думаю, что смысл выражения не потерялся от того, что его перевели на английский, а потом я перевел его на русский, потому что оно гениально: *"Простота является наибольшей изысканностью"*.

Не ищите сложных решений, сделайте все максимально просто, и вы получите изысканный код с минимальным количеством проблемных мест.

На тему безопасного кода написано немало работ, и тут можно писать отдельную книгу. Кстати, недавно я видел что-то подобное в Интернете, но пока что еще не читал, но собираюсь сделать это в ближайшее время. Я читал уже достаточно много статей разных специалистов по безопасности, но рецепт идеального кода я не могу вам сказать. Каждый имеет свою точку зрения на программирование, ведь сколько людей — столько и мнений.

Глава 2



Простые шутки

Теперь можно приступать к написанию простых программ-приколов в Windows. Так как эта ОС самая распространенная, то и приколы в ней самые ценные, да и рассматриваем мы язык программирования Delphi, который создает исполняемые файлы именно для этой ОС. Я думаю, что любой компьютерщик с удовольствием подкинет своему другу какую-нибудь веселую программку, которая введет жертву в легкий шок. В каждом из нас заложено еще при рождении стремление к превосходству. Все мы хотим быть лучшими, и программисты часто показывают свое превосходство с помощью написания чего-то уникального, интересного и вызывающего. Чаще всего в виде самовыражения выступают программы-приколы.

Хотя мои программы не будут вредоносными, но все же они должны быть кому-нибудь подброшены. Поэтому человека, которому должна быть подкинута программа, я буду называть жертвой.

Искусство хакера состоит в том, чтобы сделать из абсолютно безобидной вещи что-нибудь веселое, интересное и даже шокирующее. Возьмем буфер обмена. Ну что можно сделать с этой абсолютно безобидной возможностью Windows запоминать и вставлять какие-то данные? Но если подойти к задаче с нужным энтузиазмом, то даже такая простая вещь может превратиться в орудие красивой шутки, и в этой главе мы увидим, как подойти к этому вопросу с энтузиазмом и подшутить.

Большинство приколов этой главы основано на простых функциях WinAPI (Windows Application Program Interface, интерфейс прикладных программ Windows). Хотя я и сказал, что начальные знания Delphi желательны, но все же весь используемый в книге код я постараюсь разъяснять очень подробно. Особый упор сделан на используемые в примерах WinAPI-функции. Если некоторые возможности Delphi вы применяете каждый день, то функции WinAPI можете использовать достаточно редко, поэтому я даже не надеюсь, что вы знаете их все.

2.1. Летающая кнопка *Пуск*

Вспоминаю, как я первый раз увидел Windows 95. Мне так понравилась кнопка **Пуск**, что я ее полюбил до глубины **Выключить компьютер**. Вскоре в нашем институте обновили парк машин, и на них тоже поставили Windows 95. Мне так захотелось приколоться над бедными однокурсниками, что я написал программку, которая подбрасывала кнопку **Пуск**. Сказано — сделано, написал и запустил на всех машинах. С каждым взлетом кнопки **Пуск** ламеры испуганно взлетали вместе с ней. А через некоторое время я увидел и в Интернете подобный прикол.

Сейчас я повторю свой старый подвиг и покажу вам, как самому написать такую программу. Так что усаживайтесь поудобнее, наша кнопка **Пуск** взлетает на высоту 100 пикселов, а может и больше!

Но прежде чем начать, нужно подготовить картинку с изображением кнопки **Пуск**. Для этого вы можете нарисовать ее своими руками в любом графическом редакторе. Ну а если вы "IBM PC"-совместимый человек, то можете нажать клавишу <Print Screen>, чтобы запомнить содержимое экрана в буфере обмена, а потом выполнить вставку содержимого буфера в любом графическом редакторе. Далее, простыми манипуляциями вырезать изображение кнопки и сохранить его в отдельном файле.

Создайте новый проект в Delphi. Сразу сохраните его. Теперь изменим параметры окна. Для этого перейдем в Инспектор объектов (**Object Inspector**). Здесь параметр `BorderStyle` устанавливаем равным `bsNone`, чтобы у окна не было никаких обрамлений. Параметру `FormStyle` задаем значение `fsStayOnTop`, чтобы окно всегда располагалось поверх других. Все, форма готова.

Теперь нужно бросить на форму компонент `Image` с палитры компонентов **Additional**. На форме появится соответствующий компонент с именем `Image1`. У этого компонента установим свойства `Left` и `Top` равными 0, чтобы картинка располагалась точно в левом верхнем углу формы (рис. 2.1).

Теперь дважды щелкните справа от параметра `Picture` в окне Инспектора объектов (или выделите параметр и нажмите кнопку с тремя точками), и перед вами появится окно, позволяющее загрузить в компонент `Image1` картинку (рис. 2.2). Нажмите кнопку **Load** и выберите файл, в котором было сохранено изображение кнопки **Пуск**. После этого установите свойство `AutoSize` у `Image1` равным `true`, чтобы компонент стал размером с рисунок.

Хорошо. Форма почти готова. Осталось только поправить ширину и высоту самого окна, чтобы оно было размером с картинку. Однако и высота, и ширина не могут быть меньше, чем размер кнопок на обрамлении окна (даже

если обрамление удалить). Но и это мы победим! Просто размеры окна нужно задать программно во время создания окна или его отображения.

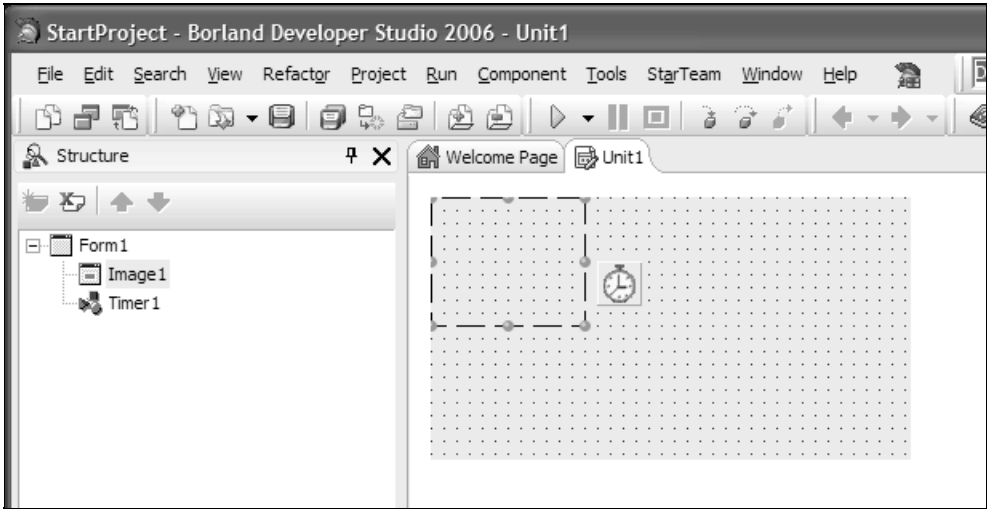


Рис. 2.1. Форма будущей программы

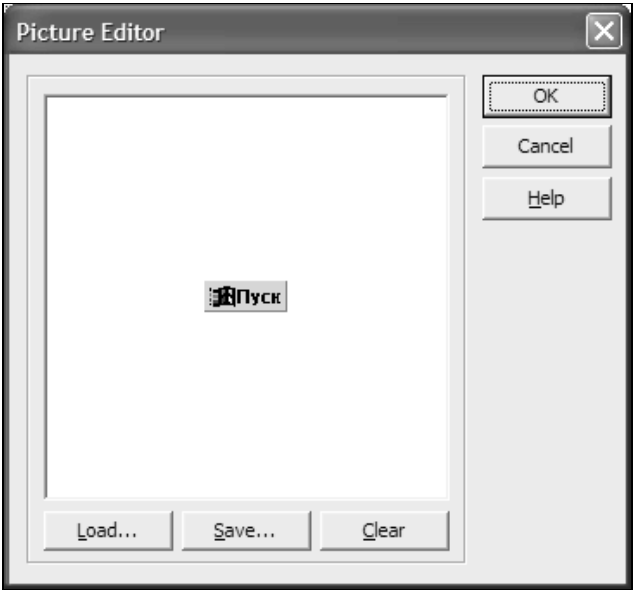


Рис. 2.2. Загрузка картинки

Для примера мы все сделаем в обработчике события `OnShow`. Создайте его и напишите там следующий код:

```
procedure TForm1.FormShow(Sender: TObject);
begin
 Width:=51;  // Установить ширину окна
 Height:=21; // Установить высоту
 Left:=-100; // Убрать окно за левую границу экрана
end;
```

Здесь устанавливаются значения ширины и высоты окна. В визуальном редакторе есть проблемы с установкой этих значений, а программно можем задать то, что нам нужно. Ваши значения могут быть другими, все зависит от того, какого размера получилось изображение вашей кнопки. Моя вышла габаритами 21×51, даже несмотря на то, что я больше люблю габариты из трех чисел 90×60×90. Хотя нет, это совсем другие габариты, и к изображению кнопки они никакого отношения не имеют.

Теперь перенесите на форму компонент `Timer` с палитры компонентов **System**. В его свойствах нужно найти и изменить значение `Interval`. По умолчанию оно равно 1000 миллисекунд (1 секунда). Для нас больше подойдет 10 000 миллисекунд (10 секунд). Не интересно, если кнопка **Пуск** будет летать каждую секунду. С такими темпами она может даже не успеть приземлиться.

Теперь перейдите на вкладку **Events** и дважды щелкните справа в строке события `OnTimer`. Будет создан обработчик события, который будет вызываться каждые 10 секунд (значение свойства `Interval`). Здесь мы развернем центр управления полетом кнопки **Пуск**. В этом обработчике напишите код из листинга 2.1.

Листинг 2.1. Обработчик события `OnTimer`

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
 i:Integer;
begin
 Visible:=true; // Сделать окно видимым

 // Установить верхнюю позицию окна в левый нижний угол экрана
 Top:=Screen.Height-Height;
 Left:=1;
```

```
 // Сейчас будем поднимать кнопку.
 // От 1 до 80 выполнять действия от begin до end
 for i:=1 to 80 do
  begin
   // Увеличить значение верхней позиции окна с кнопкой
   Top:=Screen.Height-Height-i*5;
   Repaint;  // Перерисовать окно
   Sleep(15);// Задержка в 15 миллисекунд
  end;

 // Дальше идет опускание кнопки. Алгоритм тот же,
 // просто выполнение идет в обратном порядке
 for i:=80 downto 1 do
  begin
   Top:=Screen.Height-Height-i*5;
   Repaint;
   Sleep(15);
  end;

  Visible:=false; // Прячется окно
end;
```

Я постарался снабдить этот листинг подробными комментариями, так что, надеюсь, вы разберетесь с приведенным кодом.

В принципе, программа готова, и ее можно запускать. Но если вы сделаете это сейчас, то она появится в панели задач, что абсолютно недопустимо для программы-прикола. Давайте напишем код, который спрячет наше приложение от грозного взгляда панели задач. Для этого выберите из меню **Project** пункт **View Source**. Перед вами появится исходный текст самого проекта. Сравните его с листингом 2.2 и недостающее допишите.

Листинг 2.2. Скрытие программы от панели задач

```
program Project1;

uses
  Forms,
```

```
  Windows,  // Это добавлен модуль Windows
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

// Далее добавлена новая переменная
var
 EStyle : integer;
begin
  Application.Initialize;

// Далее идет установка невидимости приложения
  EStyle:=GetWindowLong(application.Handle, GWL_EXSTYLE);
  SetWindowLong(Application.Handle, GWL_EXSTYLE,
                EStyle or WS_EX_TOOLWINDOW);

// Остальное не менять
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Чтобы вам было проще разобраться, изменения выделены комментариями. Это всего лишь один из способов спрятать программу от пользователя. По мере необходимости мы будем знакомиться и с другими.

Все. Играйте на здоровье (рис. 2.3). Только будьте осторожны. Слабонервные ламеры, видя летающую кнопку, могут уйти в бессознательное состояние на пару десятков лет. Нашатырь тут уже не поможет, так что за побочные действия отвечать будете сами.

Единственный недостаток такого простого способа — он будет эффективен только под определенными версиями Windows, т. к. в Windows XP кнопка имеет другой вид. Хотя кто вам мешает нарисовать кнопку в стиле XP и в зависимости от версии, установленной у пользователя, подбрасывать то или иное изображение? Действуйте.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 2\Кнопка Пуск\ вы можете увидеть пример работающей программы и цветные версии рисунков этого раздела.

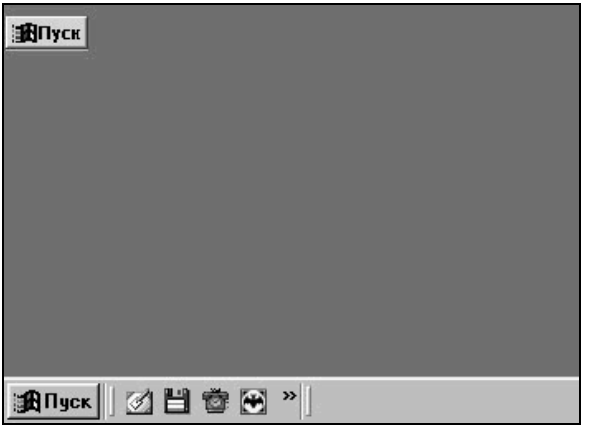


Рис. 2.3. Ловите, ловите, она летит

2.2. Полный контроль над кнопкой *Пуск*

В предыдущем примере я смухлевал и подбрасывал на экране бутафорию, а реальная кнопка оставалась на месте без изменений. Сейчас я покажу вам, как можно получить доступ к самой кнопке **Пуск**, управлять ею и изменять ее вид (последнее будет доступно только в Windows 95/98/ME).

Стартовая кнопка — это простое окно с картинкой, которое выглядит таким образом! Чтобы получить к ней доступ, нужно знать идентификатор этого окна. Как его можно получить? С помощью API-функции `FindWindow`. У этой функции два параметра: первый — это имя класса окна, а второй — это имя окна. Кнопка **Пуск** имеет имя класса `Shell_TrayWnd`. Точнее сказать, это класс всей панели задач. Имя нам знать необязательно, потому что если его не задать, то `FindWindow` укажет нам на первое найденное окно этого класса. Спешу вас обрадовать, что для такого класса в Windows есть только одно окно, и это именно панель задач.

Чтобы получить доступ к картинке кнопки на панели задач, можно воспользоваться более продвинутой функцией — `FindWindowEx`. Эта функция позволяет нам получить доступ к любому элементу в окне. У нее уже есть четыре параметра:

- окно, на котором нужно искать элемент управления;
- элемент управления на этом окне, с которого нужно начинать поиск; если здесь указать 0, то поиск будет начинаться с самого первого элемента управления;

- класс элемента управления; в нашем случае это кнопка, значит, нужно указать `Button`;
- имя; если указать `nil`, то будет происходить поиск всех элементов подобного класса.

Итак, чтобы получить контроль над кнопкой **Пуск**, нужно написать следующий код:

```
StartBtnWnd:=FindWindow('Shell_TrayWnd', nil);
StartBtnBmp:=FindWindowEx(StartBtnWnd, 0, 'Button', nil);
```

Здесь в первой строчке ищется окно панели задач. Результат поиска сохраняется в переменной `StartBtnWnd`. Во второй строчке находим саму кнопку **Пуск** внутри найденной панели задач. Результат этого поиска будет сохраняться в переменной `StartBtnBmp`.

Теперь запускаем Delphi и готовимся шкодить. Создайте новый проект и обработчик события для формы `OnCreate`. В этом обработчике напишите те две строчки, которые указаны выше. Теперь подымитесь немного выше и найдите в коде раздел `private` в описании. Добавьте туда две переменные, чтобы хранить идентификатор панели и кнопки с картинкой в качестве членов класса:

```
private
  { Private declarations }
  StartBtnWnd, StartBtnBmp: hWnd;
```

Вот теперь у нас есть идентификатор окна нужной кнопки и идентификатор картинки, и мы готовы приступить к управлению кнопкой **Пуск**. Давайте сделаем это!

Заготовьте рисунок кнопки размером примерно 50×20. Мою заготовку вы можете увидеть на рис. 2.4.

Рис. 2.4. Новая картинка для кнопки **Пуск**

Теперь перенесите на форму один компонент `Image` с палитры компонентов **Additional**. Перейдите в окно Инспектора объектов и дважды щелкните справа свойства `Picture`. Перед вами должно появиться окно для загрузки картинки. Нажмите кнопку **Load**, найдите заготовленную для кнопки картинку и откройте ее.

И последнее: перенесите на форму пять кнопок (рис. 2.5) и назовите их так:

- **Изменить картинку**;
- **Отключить**;
- **Включить**;
- **Спрятать картинку**;
- **Спрятать панель**.

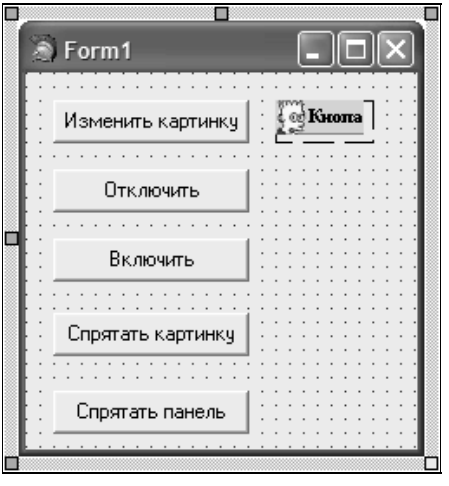


Рис. 2.5. Форма будущей программы

Все!!! Приготовления закончены. На рис. 2.5 вы можете видеть форму программы, которая получилась у меня. Осталось только заставить эти кнопки выполнять то, что на них написано.

Сейчас мы применим несколько приемов самбо к кнопке **Пуск**. Начнем с изменения рисунка кнопки. Для этого создайте обработчик события `OnClick` для кнопки **Изменить картинку** и напишите в нем следующее:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 SendMessage(StartBtnBmp, BM_SetImage, 0, Image1.Picture.Bitmap.Handle);
end;
```

Здесь использована WinAPI-функция `SendMessage`, которая посылает системное сообщение. У этой функции есть четыре параметра:

- окно, которому надо послать сообщение — указан идентификатор окна-кнопки;

- тип сообщения — указано `BM_SetImage`, что заставит окно изменить картинку;
- первый параметр сообщения — он равен нулю;
- второй параметр сообщения — указатель на картинку, которую надо подставить.

Просто и со вкусом. Программа отправила это сообщение системе — окну кнопки **Пуск**. Это окно, увидев сообщение о том, что надо изменить картинку, беспрекословно выполняет указание. На рис. 2.6 вы можете увидеть результат работы данной маленькой программы. Жаль, что этот прием работает только в Windows 9*x*. В Windows XP кнопка не хочет менять свой внешний вид, а в Windows 2000 я не пробовал. Так почему же я оставил этот пример в наше время, когда миром правят XP, а Vista постепенно начинает покорять сердца пользователей? Дело в том, что этот пример очень поучительный.

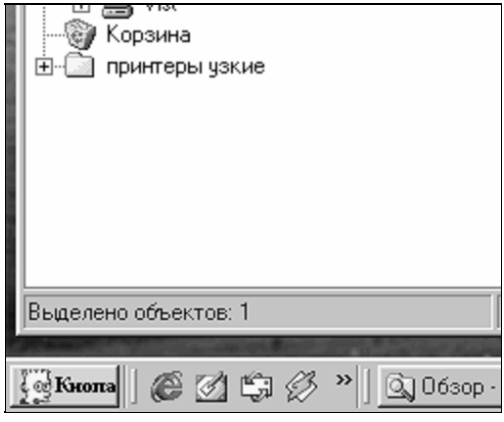


Рис. 2.6. Результат изменения картинки

Описанные дальше приемы работают в любой системе Windows — мы научимся включать и отключать кнопку **Пуск**. Создайте обработчики событий `OnClick` для кнопок **Выключить** и **Включить**. В первом напишите следующее:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
 EnableWindow(StartBtnWnd, false);
end;
```

Во втором обработчике события (для кнопки **Включить**) введите следующее:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
 EnableWindow(StartBtnWnd, true);
end;
```

В обоих случаях использована одна и та же функция `EnableWindow`. Эта функция делает окно доступным и отключает его в зависимости от переданных ей параметров. Первый параметр — идентификатор окна. Во втором вы собственно и указываете — включить окно (`true`) или отключить (`false`). Если его надо отключить, то кнопка перестанет реагировать на нажатия, и сколько бы вы не щелкали по ней, реакции никакой не будет. Можете хоть монитор мышью проткнуть, никакого меню вы не увидите.

Теперь научимся прятать кнопку и всю панель задач. Создайте обработчик события `OnClick` для кнопки **Спрятать картинку** и напишите следующее:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
 ShowWindow(StartBtnBmp, SW_HIDE);
end;
```

Здесь опять использована функция `ShowWindow`. В качестве второго параметра указано `SW_HIDE`, чтобы спрятать окно. Чтобы показать его снова, нужно указать параметр `SW_SHOW`. В качестве первого параметра задан указатель на картинку. Как видите, с помощью данной функции можно прятать не только окна, но и компоненты. Только эти компоненты должны быть оконными и иметь указатель на себя типа `hWnd`.

Теперь спрячем всю панель. Для этого создайте обработчик события `OnClick` для кнопки **Спрятать панель** и напишите в нем следующее:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
 ShowWindow(StartBtnWnd, SW_HIDE);
end;
```

Здесь используется тот же прием, что и для предыдущей кнопки, только в качестве первого параметра функции `ShowWindow` указан идентификатор всей панели.

Несмотря на то, что в Windows XP картинку я изменить не смог, зато можно легко изменить надпись на кнопке. Дело в том, что это просто текст окна, который можно менять с помощью функции `SetWindowText`. Следующий пример устанавливает в качестве заголовка надпись **Hack**:

```
SetWindowText(StartBtnBmp, 'Hack');
```

Теперь реально проанимируем кнопку **Пуск**. В *разд. 2.1* мы обманом пугали пользователя и подбрасывали рисунок, при этом настоящая кнопочка оставалась на месте. Теперь давайте подбросим настоящую кнопку. Это не так уж и сложно.

Среди Windows API есть несколько функций, позволяющих позиционировать окна. А мы же уже знаем, что кнопка **Пуск** — на самом деле окно, а значит, любая из функций подойдет нам. Но не все так просто. Дело в том, что кнопка находится в другом окне и может двигаться только в его пределах. Чтобы обойти это ограничение, необходимо сначала оторвать кнопку от панели задач и потом двигать ее куда угодно.

Итак, следующий код показывает, как можно оторвать кнопку и проанимировать:

```
var
 i:Integer;
begin
 // Устанавливаем в качестве владельца 0
 Windows.SetParent(StartBtnBmp, 0);

 // Цикл анимации
 for I := 0 to 500 do
  begin
   // Устанавливаем новую позицию
   SetWindowPos(StartBtnBmp, 0,  10, i, 200, 80, SWP_SHOWWINDOW);

   // Задержка в 15 миллисекунд
   Sleep(15);
  end;

 // Возвращаем родителя
 Windows.SetParent(StartBtnBmp, StartBtnWnd);
end;
```

Чтобы изменить владельца кнопки, мы используем функцию `SetParent`. Такая функция есть в составе WinAPI и в VCL. Нам нужен вариант из WinAPI, поэтому перед функцией стоит имя модуля `Windows`, в котором находится объявление нужного нам варианта.

Функция `SetParent` получает в качестве параметров идентификатор окна, владельца которого мы хотим изменить, и идентификатор нового владельца.

Если в качестве владельца указать 0, то окно становится бесхозным и может гулять, куда хочет и когда хочет. Так что теперь мы можем изменять позицию окна кнопки, и эта позиция будет меняться в пределах всего рабочего стола.

После завершения анимации возвращаем в качестве владельца панель задач, благо у нас она есть и хранится в переменной `StartBtnWnd`. Несмотря на то, что кнопку мы возвращаем полноправному владельцу, кнопка после анимации не будет видна. Видимо, она становится неправильно позиционированной, или возникает какая-то другая проблема. Все возвращается на родину, если ручками подвинуть панель задач. Программно я пытался заставить панель прорисоваться или двинуться, но кнопка появляется только после ручных манипуляций.

> ***ПРИМЕЧАНИЕ***
>
> На компакт-диске в каталоге \Примеры\Глава 2\ Кнопка Пуск2\ вы можете увидеть цветные версии рисунков этого раздела.

2.3. Панель задач

Не менее интересным объектом для шуток может быть и панель задач. Хакеры любят над ней шутить, потому что пользователи хорошо реагируют на изменения в этой области.

Панель задач — это тоже окно, поэтому в него можно помещать элементы управления, как и на рабочий стол, мы можем изменять его форму и делать другие операции. Мы рассмотрим наиболее интересные приемы работы с панелью.

Следующий пример показывает, как можно добавить в панель задач кнопку:

```
procedure TForm1.Button2Click(Sender: TObject);
var
 h:HWND;
begin
 h:= FindWindow('Shell_TrayWnd', nil);
 windows.SetParent(Button2.Handle, h);
end;
```

Чтобы воспользоваться данным кодом, создайте новый проект и поместите на его форму кнопку `TButton` с именем `Button2`. Приведенный выше код напишите в обработчике события `OnClick` этой кнопки.

Из *разд. 2.2* мы знаем, что панель задач имеет класс `Shell_TrayWnd`. В данном примере, в первой строке кода, мы уже знакомым нам способом (с помощью функции `FindWindow`) находим панель задач.

Во второй строке вызывается метод `SetParent` из состава Windows API. Этот метод нам также уже знаком по *разд. 2.2*. В данном случае, кнопке `Button2`, которую мы поместили на форму, устанавливаем в качестве родителя панель задач.

Запустите проект и нажмите кнопку. В результате она исчезнет с формы вашей программы и появится на панели задач в той же позиции и с тем же размером, как и на исходной форме. Если закрыть окно программы, то кнопка тоже исчезнет с панели задач.

Вот тут нужно сделать одно замечание. Дело в том, что поместить свой элемент управления в чужое окно не так уж и сложно. Проблема в том, что этот элемент не будет работать. В данном случае кнопка меняет родителя, и именно этот родитель будет обрабатывать все события. Так как панель задач ничего не знает о нашей кнопке, то и работать она не будет.

Давайте теперь попробуем внедрить в панель задач меню. Для этого поместите на форму компонент `TMainMenu` и смастерите что-нибудь, чтобы были пункты меню и подпункты. Теперь бросим на форму кнопку, по нажатию которой все и будет происходить, и в обработчике события `OnClick` этой кнопки напишем следующий код:

```
procedure TForm1.Button3Click(Sender: TObject);
var
 h:HWND;
begin
 h:= FindWindow('Shell_TrayWnd', nil);
 SetMenu(h, MainMenu1.Handle);
end;
```

Самое странное, что если у вас включен стиль Windows XP, то данный код не даст никакого эффекта. Меню появится только при классическом стиле отображения или в более старой версии Windows (рис. 2.7).

Рис. 2.7. Вот так вот можно внедрить меню в панель задач

Теперь попробуем поиграть с формой панели задач. Мы можем без проблем придать панели произвольную форму, например овал или любой другой вид. В данном случае мы не будем далеко углубляться в игры с формами окна, а ограничимся заданием только овала. Более подробно создания окон произвольной формы мы рассмотрим чуть позже.

Итак, бросьте на форму проекта, в котором вы тестируете описываемый мною код, очередную кнопку. Для события, возникающего по ее нажатию, напишите следующий код:

```
procedure TForm1.Button4Click(Sender: TObject);
var
 EllipseRgn:HRGN;
 h:HWND;
begin
 // Ищем окно панели задач
 h:= FindWindow('Shell_TrayWnd', nil);

 // Создаем овальный регион
 EllipseRgn:=CreateEllipticRgn(1,1, Screen.Width, 200);

 // Устанавливаем панели овальный регион
 SetWindowRgn(h,EllipseRgn,True);
end;
```

В первой строке, уже знакомая нам строка поиска панели задач. Во второй строке с помощью функции `CreateEllipticRgn` мы создаем овальную область. Ну и, наконец, устанавливаем панели задач овальный регион.

А теперь то же самое, но немного поподробнее. Для создания овального региона окна используется функция `CreateEllipticRgn`, у которой четыре параметра, описывающие прямоугольник (`X`-координата левого верхнего угла, `Y`-координата левого верхнего угла, `X`-координата правого нижнего угла, `Y`-координата правого нижнего угла), в который будет вписан овал. Левый верхний угол я установил в позиции (1, 1), т. е. сделал отступ в один пиксел. А правый нижний угол по оси *x* смещен на ширину окна, а по оси *y* — на 200.

Чтобы назначить регион, используется функция `SetWindowRgn`, у которой три параметра:

- окно, которому нужно назначить регион;
- созданный заранее регион;

- нужно ли перерисовать окно — если здесь указать `true`, то ОС перерисовывает окно сразу после изменения региона.

Пользы от этого примера не так уж и много, потому что с овальной панелью задач неудобно работать, но в шуточной программе эту возможность очень удобно использовать. Например, можно попробовать анимировать панель задач, плавно изменяя ее форму.

Во время моих исследований панели задач я заметил еще один интересный эффект. Допустим, что панель расположена, как принято по умолчанию, внизу. Теперь если ее расширять вверх, то она будет заполнять всю нижнюю область. Но если к панели применить прямоугольный регион с маленьким размером высоты, то эффект растягивания исчезнет.

Следующий код устанавливает панели задач прямоугольную форму высотой всего 70 пикселов, поэтому окно больше не будет растягиваться:

```
procedure TForm1.Button4Click(Sender: TObject);
var
 EllipseRgn:HRGN;
 h:HWND;
begin
 h:= FindWindow('Shell_TrayWnd', nil);
EllipseRgn:=CreateRectRgn(1,1, Screen.Width, 70);
 SetWindowRgn(h,EllipseRgn,True);
end;
```

Создание прямоугольного региона идентично созданию овала, только в данном случае используется функция `CreateRectRgn`. В качестве высоты региона взято значение в 70 пикселов, и теперь панель никак не сможет быть более этого значения, даже если вы попытаетесь ее растянуть.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 2\TaskControl\ вы можете увидеть пример этой программы.

2.4. Настольные розыгрыши

Теперь переходим к рабочему столу Windows. Рабочий стол достаточно большой, и поле для розыгрышей тут тоже нешуточное. Да, в нашем деле главное — воображение, и я не могу привести абсолютно все возможные варианты, поэтому рассмотрю то, с чем приходилось сталкиваться или о чем спрашивали читатели.

Рабочий стол — это тоже окно, а значит, его можно найти и использовать полученный идентификатор, но есть способ лучше — воспользоваться WinAPI-функцией `GetDesktopWindow`. Ей не нужны параметры, а идентификатор окна мы получим в виде возвращаемого функцией значения. Итак, простейший способ найти рабочий стол:

```
var
 h:HWND;
begin
 h:=GetDesktopWindow();
end;
```

Давайте попробуем поместить на рабочий стол кнопку, тем более, что мы ее уже помещали на панель задач и знаем, как это делается:

```
var
 h:HWND;
begin
 h:=GetDesktopWindow();
 windows.SetParent(Button1.Handle, h);
end;
```

Код изменился не сильно, и ничего нового здесь пока рассказать не могу. Кнопка, попав на рабочий стол, не будет больше реагировать на нажатия, точнее сказать, она потеряет связь со своим обработчиком событий.

Еще один недостаток — в панели задач появляется новая кнопка, как для нашего приложения, но по нажатию которой активируется кнопка на рабочем столе. Получается, что кнопка стала как бы отдельным, практически независимым окном, даже несмотря на то, что связь осталась.

Как теперь можно подшутить над пользователем? Например, можно засыпать рабочий стол кучей кнопок. Для этого необходимо в цикле создавать новый экземпляр кнопки и тут же бросать его на рабочий стол.

```
var
 h:HWND;
 b:TButton;
 i:Integer;
begin
 h:=GetDesktopWindow();
 for i:=1 to 50 do
  begin
   b:=TButton.Create(Owner);          // Создаем кнопку
```

```
    b.Left:=random(Screen.Width);      // Левая позиция
    b.Top:=random(Screen.Height);      // Верхняя позиция
    b.Caption:='Button'+IntToStr(i);   // Заголовок для красоты

    b.Parent:=Self;                    // Временный родитель
    windows.SetParent(b.Handle, h);    // Родителем делаем рабочий стол
  end;
end;
```

Здесь запускается цикл из 50 шагов для создания такого же количества кнопок. Внутри цикла динамически создается экземпляр кнопки `TButton`. После этого случайным образом устанавливаем левую (`Left`) и верхнюю (`Top`) позиции кнопочки, а также заголовок, чтобы на поверхности хоть что-то было.

Напоследок мы должны установить кнопку на рабочий стол, но тут есть одна проблема — для выполнения функции `SetParent` у компонента уже должен быть родитель. Если бы кнопку мы поставили в визуальном дизайнере во время проектирования программы, то родителем автоматически стала бы форма. При динамическом создании мы и родителя должны установить. Именно поэтому в свойство `Parent` созданной кнопки помещаем текущую форму (т. е. `Self`).

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 2\Desktop\ вы можете увидеть пример этой программы.

2.5. Контролируем системную палитру

Системная палитра — достаточно интересный объект для программиста, который хочет написать какую-нибудь шуточную программу. Изменив всего один системный цвет, изменяется внешний вид всех программ. Попробуем написать программу, которая будет хаотично менять основные цвета системной палитры.

Для нашей программы нам понадобятся одна кнопка и таймер. На кнопке (`Button1`) напишите **Анимация палитры**. Вид формы не имеет значения — в нашем примере мы делаем упор на изменение палитры, а не на интерфейс.

У таймера установите свойство `Enabled` равным `false`, чтобы по умолчанию он был выключен. При нажатии кнопки мы его включим и начнем измене-

ния палитры. Свойство `Interval` у таймера нужно задать большим 5000 (меньше не рекомендуется). Внешне палитра меняется достаточно долго (1—3 секунды, в зависимости от количества открытых программ и версии Windows). Если пример будет запускаться в Windows XP, то интервал лучше увеличить до 10 секунд, т. е. установить значение 10 000.

Итак, для изменения системного цвета существует функция `SetSysColors`. У нее всего три параметра:

- количество цветов, которые мы хотим изменить;
- номера цветов в системной палитре для изменения;
- новые значения цветов.

Допустим, что вы хотите поменять пятый цвет на красный. В этом случае нужно дать такую команду:

```
SetSysColors(1, 5, clRed);
```

Но менять так все цвета неудобно, и мы пойдем по пути прогресса — автоматизируем процесс. По нажатию первой кнопки **Анимация палитры** запустим таймер:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 Timer1.Enabled:=true;
end;
```

Теперь создадим обработчик события срабатывания таймера (для этого просто дважды щелкните по нему). Здесь мы напишем код смены палитры:

```
procedure TForm1.Timer1Timer(Sender: TObject);
const
  SySColorArray: array [0..13] of Integer = (COLOR_ACTIVEBORDER,
    COLOR_ACTIVECAPTION, COLOR_APPWORKSPACE, COLOR_BACKGROUND,
    COLOR_BTNFACE, COLOR_BTNTEXT, COLOR_CAPTIONTEXT,
    COLOR_INACTIVEBORDER, COLOR_INFOTEXT, COLOR_MENU,
    COLOR_MENUTEXT, COLOR_WINDOW, COLOR_WINDOWFRAME,
    COLOR_WINDOWTEXT);

  ColorArray:array [0..10] of TColor = (clAqua, clBlack, clBlue,
           clYellow, clFuchsia, clGreen, clNavy, clRed, clSilver,
           clWhite, clSkyBlue);
```

```
begin
 SetSysColors(1, SySColorArray[random(13)], ColorArray[random(10)]);
end;
```

У процедуры есть собственные константы, и обе они объявлены как массивы.

Первый массив `SySColorArray` — это массив из 14 целых чисел. Так как этот массив — константа, то при его определении необходимо поставить в конце знак равно и в скобках перечислить значения всех элементов массива. В качестве элементов перечислены системные цвета, которые будут меняться (`COLOR_ACTIVEBORDER`, `COLOR_ACTIVECAPTION` и т. д.). Мы будем случайным образом выбирать из этого массива имя системного цвета и присваивать ему новое значение.

Что такое `COLOR_ACTIVEBORDER`, `COLOR_ACTIVECAPTION` и т. д.? Это системные константы. Под каждой из них спрятано целое число. Например, `COLOR_ACTIVEBORDER` — это обычное число 10. Каждое такое число означает номер системного цвета. Я мог бы использовать вместо констант числа, но читать такой код будет неудобно.

Вторая константа `ColorArray` — массив из 10 цветов. Из него мы будем выбирать случайным образом элемент и присваивать его в качестве значения системного цвета.

Код процедуры очень прост и состоит только из вызова функции `SetSysColors`. В качестве параметра указываются следующие значения:

- `1` — за один раз будет меняться только один системный цвет;
- `SySColorArray[random(13)]` — из массива `SySColorArray` выбирается случайный элемент — системный цвет, который будет изменен. Функция `random(13)` выдает случайное число от 0 до 13;
- `ColorArray[random(10)]` — выбирается из массива `ColorArray` случайное значение цвета от 0 до 10 с помощью вызова функции `random(10)`. Это значение будет присвоено системному цвету — кандидату на изменение.

Теперь можете построить проект и насладиться переливающимся всеми цветами радуги монитором.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 2\Palette\ вы можете увидеть пример этой программы.

2.6. Изменение разрешения экрана

Сейчас мы напишем пример, который сможет узнавать у системы возможные видеорежимы и заставит компьютер перейти на любой из них. Для примера нам понадобятся на форме один компонент `ListBox` и две кнопки. При нажатии первой кнопки мы будем спрашивать у системы все возможные варианты видеорежимов и записывать их в `ListBox`. При нажатии второй кнопки мы будем делать активным режим выделенный в списке `ListBox`. Можете расположить все компоненты как угодно, а можете сделать, как я (рис. 2.8).

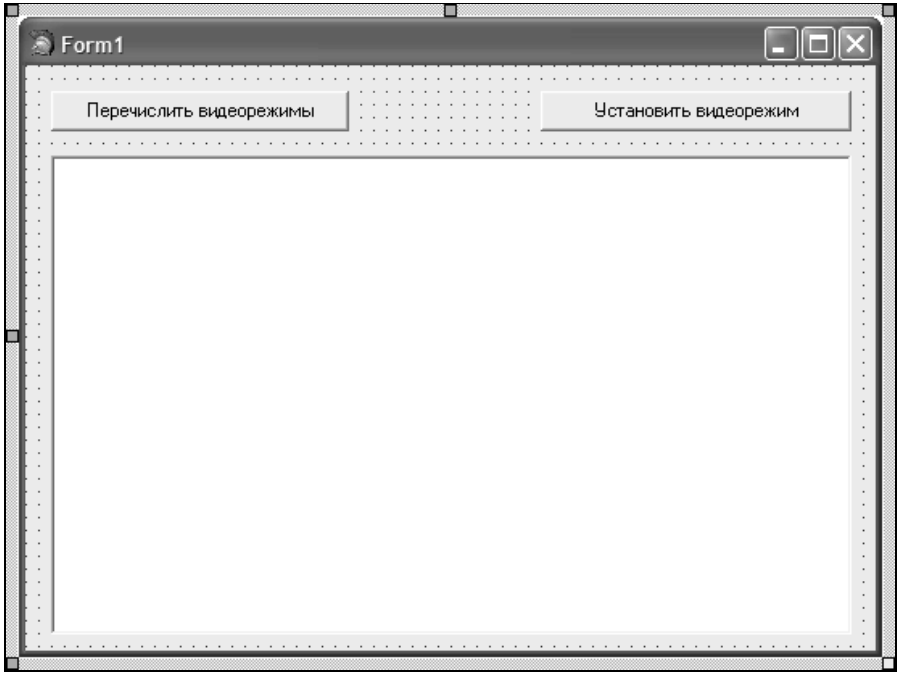


Рис. 2.8. Форма будущей программы

Итак, на первой кнопке напишите что-нибудь типа **Перечислить видеорежимы**, а на второй — **Установить видеорежим**. Теперь форма выглядит более солидно, и можно приступать к программированию.

Прежде чем написать код, надо объявить массив из переменных типа `TDevMode`, в котором мы будем сохранять параметры найденных видеорежимов.

Для этого найдите раздел `private` и запишете в нем:

```
private
  { Private declarations }
    modes:array[0..255] of TDevMode;
public
  { Public declarations }
```

Таким образом, мы объявили массив из 255 элементов типа `TDevMode`.

> ***Примечание***
>
> После выхода первого издания книги я получил письмо, в котором читатель сообщил мне, что его монитор возвращает аж 304 варианта режимов. Из-за того что мы завели массив только из 255 символов, у читателя происходило переполнение. Чтобы избежать такой проблемы, необходимо увеличить количество элементов в списке.

Для начала создадим обработчик события `OnClick` для кнопки считывания возможных видеорежимов и напишем в нем следующий текст:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i: Integer;
begin
 ListBox1.Items.Clear;
 i := 0;
while EnumDisplaySettings(nil, i, Modes[i]) do
  begin
   ListBox1.Items.Add(IntToStr(Modes[i].dmBitsPerPel)+' '+
        IntToStr(Modes[i].dmPelsWidth)+' '+
        IntToStr(Modes[i].dmPelsHeight)+'   '+
        IntToStr(Modes[i].dmDisplayFrequency));
   Inc(i);
  end;
end;
```

В первой строке кода очищается содержимое списка `ListBox` с помощью вызова метода `ListBox1.Items.Clear`. В следующей строке обнуляется переменная `i`, которая будет использоваться в качестве простого счетчика и указывать, какой по счету видеорежим нам нужен.

Дальше вызывается функция `EnumDisplaySettings`, которая перечисляет доступные системе видеорежимы. У этой функции три параметра:

- устройство, для которого надо перечислять режимы — здесь нужно что-то указывать, только если в системе несколько мониторов. Если вы простой смертный и пользуетесь одним монитором, то можно смело указывать `nil`;
- здесь следует задавать индекс режима, который нам нужен; в приведенном коде указана переменная `i`, которая перед каждым вызовом будет увеличиваться на 1, а значит, будет каждый раз получать следующий по счету режим;
- переменная типа `TDevMode`, в которую нужно записать параметры режима — сюда подставляется очередной элемент массива.

Вызов функции происходит между операторами `while` и `do`:

```
while EnumDisplaySettings(nil, i, Modes[i]) do
```

Это значит, что пока результат вызова функции `EnumDisplaySettings` не станет ошибкой, будут выполняться операторы цикла, размещенные между `begin` и `end`. В этом цикле добавляется строка в список `ListBox`, в которой записаны параметры найденного режима, и увеличивается счетчик `i` на единицу.

Когда добавляется новый элемент в список `ListBox`, то в качестве строки передается следующая информация:

- `Modes[i].dmBitsPerPel` — глубина (количество битов на пиксел) цвета найденного режима;
- `Modes[i].dmPelsWidth` — ширина экрана;
- `Modes[i].dmPelsHeight` — высота экрана;
- `Modes[i].dmDisplayFrequency` — частота развертки.

Можно теперь запустить этот промежуточный вариант нашей программы и нажать кнопку **Перечислить видеорежимы**. Список `ListBox` должен заполниться строками, содержащими информацию о найденных видеорежимах (рис. 2.9).

Все строки разбиты как бы на четыре колонки:

- глубина (количество битов на пиксел) цвета найденного режима;
- ширина экрана;
- высота экрана;
- частота развертки.

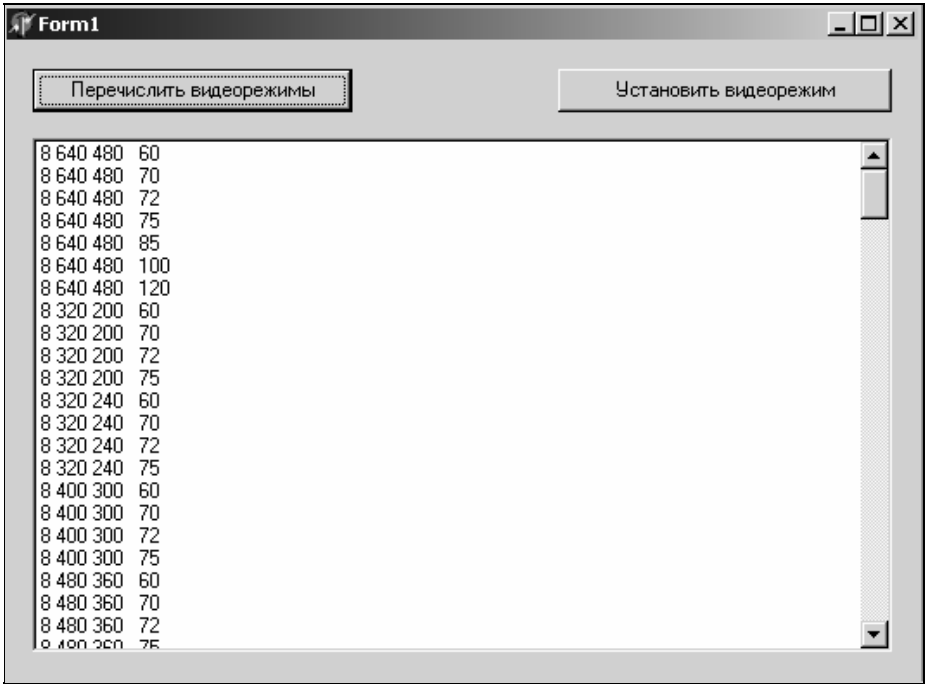


Рис. 2.9. Результат перечисления доступных видеорежимов

При нажатии кнопки **Установить режим** мы должны установить выделенный в списке режим. Для этого в обработчике нажатия второй кнопки пишем следующий код:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
 Modes[ListBox1.ItemIndex].dmFields := DM_BITSPERPEL or
     DM_PELSWIDTH or DM_PELSHEIGHT or
     DM_DISPLAYFLAGS or DM_DISPLAYFREQUENCY;
 ChangeDisplaySettings(Modes[ListBox1.ItemIndex], CDS_UPDATEREGISTRY);
end;
```

Свойство `ListBox1.ItemIndex` указывает на выделенный элемент в списке. Это значит, что для того чтобы найти нужную структуру для выделенного элемента, в массиве `Modes` надо записать `Modes[ListBox1.ItemIndex]`. Все достаточно просто, потому что элементы в списке `ListBox1` находятся в том же порядке, что и соответствующие структуры `TDevMode` в массиве `Modes`. Правда, это будет истиной, только если у вашего списка выбора свойство

`Sorted` равно `false`. Иначе компонент `ListBox` отсортирует видеорежимы по-своему, и их последовательность уже может не совпадать с массивом `Modes`. Поэтому для данного примера не устанавливайте `true`.

Прежде чем менять видеорежим, надо в структуре `Modes` заполнить свойство `dmFields`, в котором указывается, что именно нужно поменять. Здесь вы можете указать сочетание следующих флагов:

- `DM_BITSPERPEL` — будет меняться количество битов на пиксел;
- `DM_PELSWIDTH` — будет меняться ширина экрана;
- `DM_PELSHEIGHT` — будет меняться высота экрана;
- `DM_DISPLAYFREQUENCY` — будет меняться частота развертки;
- `DM_DISPLAYFLAGS` — изменить флаги дисплея.

Если вы хотите поменять только глубину цвета, то в свойство `dmFields` достаточно указать только `DM_BITSPERPEL`. Я буду менять все, поэтому перечислил все флаги через оператор `or`, который объединяет все в одно целое.

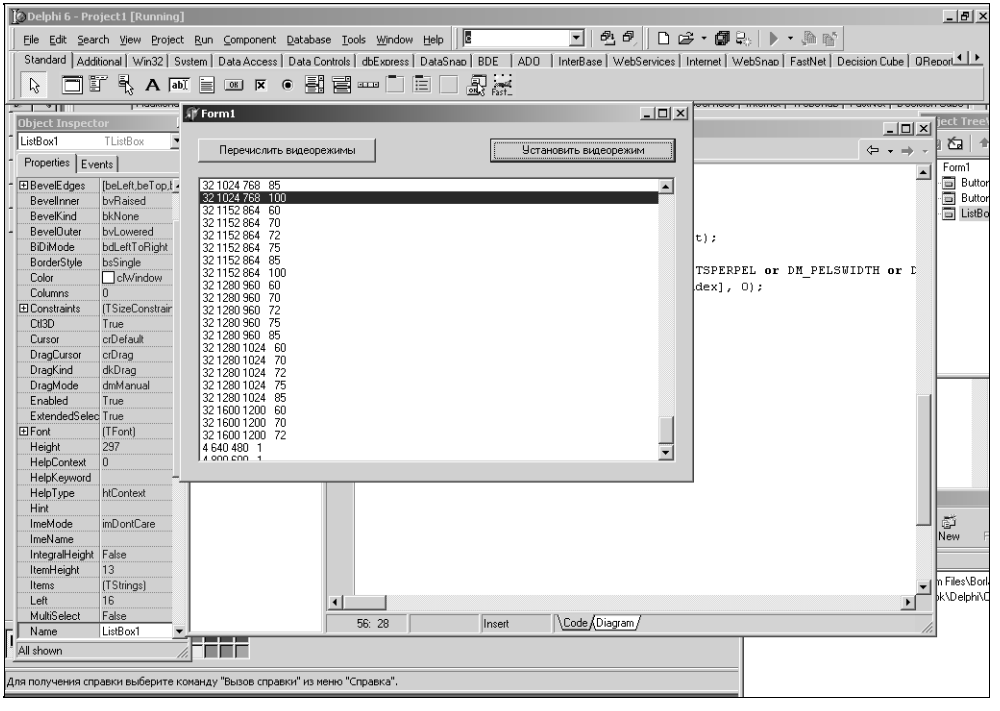


Рис. 2.10. Побочный эффект смены разрешения экрана на меньшее

После заполнения этого свойства можно вызывать процедуру `ChangeDisplaySettings`. У нее два параметра:

- структура типа `TDevMode`;
- способ перехода.

В качестве способа перехода можно указать одно из следующих значений:

- `0` — если просто поставить ноль, то разрешение экрана изменится динамически;
- `CDS_UPDATEREGISTRY` — в этом случае разрешение также изменится динамически, но с обновлением параметров в реестре;
- `CDS_TEST` — если указать это значение, то произойдет только проверка возможности переключения видеорежима. Реального переключения не будет.

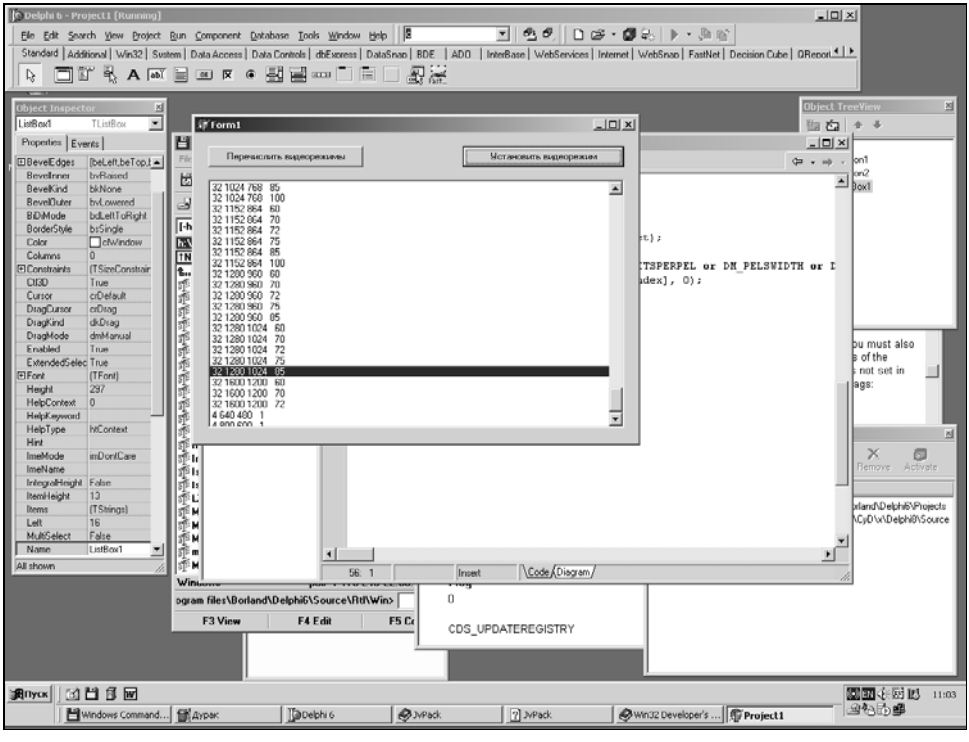


Рис. 2.11. Побочный эффект смены разрешения экрана на большее

Очень интересный эффект происходит при смене видеорежима, если указать в качестве способа 0. В этом случае режим меняется, но окна не реагируют на

эту смену и остаются в том же положении. Посмотрите на рис. 2.10, на котором я сменил режим на меньший, и при этом исчезла панель задач Windows с кнопкой **Пуск**. Она просто не отреагировала на смену режима. На рис. 2.11 показан экран, на котором, наоборот, режим изменен на больший, и панель также не отреагировала и осталась намного выше нижней части экрана.

Некоторые считают, что если часто менять разрешение экрана, то можно испортить или спалить монитор. Да, это верно, но только для старых моделей. Современные мониторы защищены от такой примитивной атаки, так что сделанный на этом эффекте вирус не принесет большого вреда. Я, конечно же, не железячник и не знаю строения современных мониторов, но никогда не слышал о том, что какой-то из экранов вышел из строя по вине смены видеорежима. Мне кажется, если что-то сломается, то только по вине заводского брака, но лучше все же не играть с огнем.

Но если включить свою соображалку на большие обороты, то можно придумать какой-нибудь более бьющий по нервам прикол. Например, многие мониторы выводят черный экран с надписью об ошибке, если попытаться установить неправильные параметры видеорежима. Это значит, что можно написать программу, которая будет менять режим на недопустимый, и поместить ее в автозагрузку. И тогда при каждой загрузке компьютера программа будет менять видеорежим, и монитор автоматом будет уходить в "даун", и пользователь не сможет некоторое время нормально работать.

ПРИМЕЧАНИЕ

На компакт-диске в каталоге \Примеры\Глава 2\Video Mode\ вы можете увидеть цветные версии рисунков этого раздела.

2.7. Маленькие шутки

В этом разделе собраны маленькие шутки и просто ответы на часто задаваемые мне вопросы. Некоторая информация, описанная в этой главе, пригодится в будущих примерах, а кое-что в книге не понадобится, хотя может оказаться вам полезным в ваших реальных проектах.

2.7.1. Как программно "потушить" монитор?

Не знаю, как программно, а огнетушителем тушится за пять секунд. Я даже помню, как в детстве получил значок юного огнетушителя, тфю, пожарника. А если серьезно, то команда системы выглядит так:

```
SendMessage(Application.Handle, WM_SYSCOMMAND, SC_MONITORPOWER, 0)
```

Чтобы "зажечь" монитор, достаточно облить монитор бензином, преподнести спичку, а чтобы включить, измените последний параметр при вызове `SendMessage` на `-1`.

2.7.2. Запуск системных cpl-файлов

Для этого сначала добавьте в раздел `uses` модуль `ShellApi`, чтобы вы могли использовать функцию `ShellExecute`. Теперь напишите следующий код:

```
ShellExecute(Application.Handle, Pchar('Open'),
  Pchar('Rundll32.exe'),
  Pchar('shell32,Control_RunDLL filename.cpl'),
  '',SW_SHOWNORMAL);
```

Функция `ShellExecute` запускает указанную в третьем параметре программу. Для запуска cpl-файлов используется утилита Rundll32, которой в качестве параметра нужно передать текст вот такого вида: `shell32,Control_RunDLL`, и потом через пробел указать имя запускаемого cpl-файла.

А вот такой код отобразит окно настроек сети Интернет:

```
ShellExecute(Application.Handle, Pchar('Open'),
    Pchar('Rundll32.exe'),
    Pchar('shell32,Control_RunDLL inetcpl.cpl'),
    '',SW_SHOWNORMAL);
```

Следующая строка отобразит окно настроек экрана:

```
ShellExecute(Application.Handle, Pchar('Open'),
    Pchar('Rundll32.exe'),
    Pchar('shell32,Control_RunDLL desk.cpl'),
    '',SW_SHOWNORMAL);
```

2.7.3. Программное управление устройством для чтения компакт-дисков

Следующий код запускает бесконечный цикл, в котором каждые 5 секунд открывается или закрывается CD-ROM:

```
var
 OpenParm: TMCI_Open_Parms;
 GenParm: TMCI_Generic_Parms;
 SetParm: TMCI_Set_Parms;
```

```
 DI : Cardinal;
 OK: boolean;
begin
 OK:=false;
 OpenParm.lpstrDeviceType := 'CDAudio';
 repeat
  mciSendCommand(0, mci_Open, mci_Open_Type, Longint(@OpenParm));
  DI := OpenParm.wDeviceID;
  mciSendCommand(DI, mci_Set, mci_Set_Door_Open, Longint(@SetParm));
  mciSendCommand(DI, mci_Set, mci_Set_Door_Closed, Longint(@SetParm));
  mciSendCommand(DI, mci_Close, mci_Notify, Longint(@GenParm));
  sleep(5000);
 until OK;
end;
```

В первой строчке кода переменной `OK` присваивается значение `false`. Потом запускается цикл `repeat..until`, который будет выполняться, пока эта переменная не станет равной `true`. Но так как нигде и никто не изменяет значение этой переменной, то она всегда будет равна `false`, а значит, цикл получится бесконечный.

Перед началом цикла необходимо еще заполнить параметр `lpstrDeviceType` структуры `OpenParm`, чтобы не делать этого в цикле. Сюда нужно занести значение `CDAudio`, что и будет указывать на необходимость работы с CD-ROM.

Ну а дальше запускается цикл, в котором поочередно то открывается дверца, то закрывается. В конце цикла ставится задержка в пять секунд `Sleep(5000)`, чтобы CD-ROM успел "увидеть" диск после того, как закрылась беспокойная дверца.

Для работы этого примера в раздел `uses` необходимо добавить модуль `MMSystem`. Именно в этом модуле прописаны все необходимые функции.

2.7.4. Отключение сочетания клавиш <Ctrl>+<Alt>+<Del>

В Windows 9*x* это можно сделать достаточно просто с помощью следующего кода:

```
var
 i:integer;
```

```
begin
 i := 0;
 SystemParametersInfo(SPI_SCREENSAVERRUNNING, 1, @i, 0);
end;
```

В Windows 2000/XP и более новых версиях подобного нет, и, скорее всего, не будет. Но в этих системах запрет вполне возможен через реестр, правда, это потребует перезагрузки системы.

2.7.5. Отключение сочетания клавиш <Alt>+<Tab>

Опять же можно привести небольшой код для Windows 9*x*:

```
var
 i:integer;
begin
 i := 0;
 SystemParametersInfo(SPI_SETFASTTASKSWITCH, 1, @i, 0);
end;
```

2.7.6. Удаление часов с панели задач

Это можно сделать почти так же, как мы работали с кнопкой **Пуск**. Нужно сначала найти окно панели задач. Потом на нем найти окно `TrayBar`, и на нем уже найти часы. После этого часики легко убираются функцией `ShowWindow`, которой нужно передать в качестве первого параметра указатель на окно часов, а во втором параметре нужно указать `SW_HIDE`.

```
var
 Wnd:THandle;
begin
 Wnd := FindWindow('Shell_TrayWnd', nil);
 Wnd := FindWindowEx(Wnd, HWND(0),'TrayNotifyWnd', nil);
 Wnd := FindWindowEx(Wnd, HWND(0),'TrayClockWClass', nil);
 ShowWindow(Wnd, SW_HIDE);
end;
```

Часы — это тоже окно, которое можно найти и перепрятать или хотя бы просто спрятать с помощью функции `ShowWindow`. Часы расположены в па-

нели задач, поэтому сначала ищем уже знакомое нам окно `Shell_TrayWnd`. На панели задач есть окно `TrayNotifyWnd`, внутри которого располагаются часы и все значки, которые вы видите левее, в том числе значок уровня звука (если этот значок у вас включен) и другие значки. Так что если спрятать это окно, то исчезнут часы и все значки. Но это нам не нужно, поэтому ищем внутри `TrayNotifyWnd` часы, которые имеют класс `TrayClockWClass`. Именно это окно и нужно спрятать.

2.7.7. Исчезновение чужого окна

Как работать с чужими окнами, мы увидим еще не раз. Но все же я покажу вам один интересный прикол, связанный с исчезновением чужих программ уже сейчас:

```
var
 h:HWND;
begin
 while true do
  begin
   h:=GetForegroundWindow();
   ShowWindow(h,SW_HIDE);
   Sleep(2000);
  end;
end;
```

В этом примере запускается бесконечный цикл `while`, внутри которого выполняются следующие шаги:

1. Ищем активное окно с помощью функции `GetForegroundWindow`. Результат выполнения этой функции — идентификатор окна, с которым сейчас работает пользователь. Результат сохраняем в переменной `h`.
2. Окно прячется с помощью функции `ShowWindow`.
3. Делается задержка в 2 секунды, чтобы пользователь успел очнуться и выбрать другое окно или заново запустить программу, с которой он работал.

2.7.8. "Клеим" обои

Как установить собственные обои на рабочий стол? Проще некуда:

```
SystemParametersInfo(SPI_SetDeskWallPaper, 0,
                     PChar(TempStr), SPIF_UpdateIniFile);
```

Функция `SystemParametersInfo` имеет следующие параметры.

- Действие, которое надо выполнить. Этих действий очень много, и описывать все нереально. Но самые интересные я приведу:
  - `SPI_SETDESKWALLPAPER` — установить собственные обои. Путь к файлу с обоями должен быть передан в третьем параметре;
  - `SPI_SETDOUBLECLICKTIME` — время двойного щелчка. Количество миллисекунд между первым и вторым щелчком мышью нужно указать во втором параметре. Попробуйте указать здесь число меньше 10, и я думаю, что вы никогда не успеете щелкнуть дважды, чтобы между щелчками прошло менее 10 миллисекунд. Таким образом, практически отключается возможность двойного щелчка;
  - `SPI_SETKEYBOARDDELAY` — во втором параметре устанавливается задержка между нажатиями клавиш при удерживании кнопки;
  - `SPI_SETMOUSEBUTTONSWAP` — если во втором параметре 0, то используется стандартное расположение кнопок мыши, иначе кнопки меняются местами, как для левши.
- Второй параметр зависит от состояния первого.
- Третий параметр зависит от состояния первого.
- На месте четвертого параметра устанавливаются флаги, в которых указывается, что надо делать после выполнения действия. Возможны следующие варианты:
  - `SPIF_UPDATEINIFILE` — обновить пользовательский профиль;
  - `SPIF_SENDCHANGE` — сгенерировать `WM_SETTINGCHANGE`-сообщение;
  - `SPIF_SENDWININICHANGE` — то же самое, что и предыдущий параметр.

Если функция выполнилась удачно, то она вернет число, не равное нулю, иначе функция вернет ноль.

Как видите, функция умеет не только менять обои. Вот пример кода, который меняет кнопки мыши местами:

```
// Установить мышь для левши
SystemParametersInfo(SPI_SETMOUSEBUTTONSWAP, 1, 0,
 SPIF_SENDWININICHANGE);

// Вернуть на родину
SystemParametersInfo(SPI_SETMOUSEBUTTONSWAP, 0, 0,
 SPIF_SENDWININICHANGE);
```

2.7.9. Запрет кнопки *Закрыть* в заголовке окна

Почти в каждом окне Windows есть кнопка **Закрыть** — это такой крестик в правом верхнем углу. С помощью нескольких строк кода ее легко можно сделать недоступной.

```
var
 SysMenu: HMenu;
begin
 SysMenu := GetSystemMenu(Handle, False);
 Windows.EnableMenuItem(SysMenu, SC_CLOSE, MF_DISABLED or MF_GRAYED);
end;
```

В первой строке мы получаем указатель на системное меню. Во второй пользуемся функцией `EnableMenuItem` из модуля Windows, чтобы отключить кнопку ×.

2.7.10. Окно, которое нельзя закрыть

Как сделать окно, которое нельзя закрыть? Да очень просто. Создаем обработчик события `OnCloseQuery` и пишем в нем всего одну строчку:

```
procedure TForm1.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
 CanClose:=false;
end;
```

Событие `OnCloseQuery` генерируется тогда, когда окно собирается закрыться. В качестве второго параметра передается переменная `CanClose`. Если этой переменной присвоить значение `true`, то окно разрешено закрыть, а если `false`, то закрывать нельзя.

В данном примере переменной `CanClose` безусловно присваивается значение `false`, а значит, окно закрыть нельзя.

Как можно использовать окно, которое нельзя закрыть? С точки зрения шуток можно написать невидимую форму, которая никогда не закрывается, и запустить программу. После этого невозможно будет корректно завершить работу Windows.

Почему не выключилась Windows. Когда вы выбираете **Пуск | Завершение работы**, то ОС посылает всем запущенным приложениям сообщение о том, что они должны завершить работу, и ожидает корректного завершения. Ес-

ли какое-то окно отказывается закрываться, то выключение невозможно. Это сделано для того, чтобы вы могли сохранить измененный документ, о котором могли забыть, когда начинали завершение работы.

Если какое-то окно не отвечает, то Windows предлагает нам завершить его работу, но наша программа работает вполне корректно, и отвечает на события. В этом случае Windows не будет предлагать завершения работы и не может предложить этого сделать. Ведь может быть такой вариант, когда открыт какой-то текстовый редактор и пользователь не сохранил данные, а значит, завершение работы приведет к потере данных. Если окно невидимо, то пользователю придется искать его среди процессов, завершать работу, и только после этого Windows сможет корректно отключиться.

Вот так, наше невидимое окно отказывается закрываться и не дает нормальными средствами выключить компьютер. Если пользователь не увидит задачу, то единственный выход — выключать компьютер кнопкой на системном блоке, что является не совсем корректным выключением. Какие-то данные или настройки могут не сохраниться и потеряться, не говоря о проблемах с жестким диском, если он отформатирован в FAT32. Проблемы могут быть и на других файловых системах, и даже на NTFS, правда, она более надежна и за счет журналирования быстрее и лучше откатывает проблемные места.

2.7.11. Спрятать окно

Очень часто ко мне приходят вопросы о том, как спрятать окно из панели задач? Ничего сложного тут нет. По событию `OnActivate`, а лучше `OnPaint` пишем:

```
ShowWindow(Handle, SW_HIDE);
ShowWindow(Application.Handle, SW_HIDE);
```

В обоих случаях вызывается функция `ShowWindow`, которую мы уже не раз использовали. В качестве параметров функции нужно передать идентификатор окна и флаг, определяющий новое значение отображения окна. В обоих случаях второй параметр равен `SW_HIDE`, т. е. требует спрятать окно. А вот первый параметр отличается. В первом случае передаем идентификатор текущего окна, который хранится в свойстве `Handle` текущей формы, а во втором случае указываем идентификатор приложения.

2.7.12. Закрыть чужое окно

Теперь посмотрим, как корректно закрыть чужое окно. Почему корректно? Мы можем прервать выполнение чужого процесса, что будет не совсем кор-

ректно и может привести к потере данных, а можно послать сообщение `WM_QUIT`, чтобы программа сама корректно завершилась или закрыла определенное окно.

Итак, чтобы послать сообщение `WM_QUIT`, можно использовать функцию `PostMessage`, которой передается четыре параметра:

- идентификатор окна, которому нужно отправить сообщение — если вы не знаете идентификатора, его всегда можно найти с помощью функции `FindWindow`;
- сообщение — в нашем случае здесь нужно указать `WM_QUIT`;
- параметр `WParam`, передаваемый в обработчик событий окна — здесь можно указать единицу (возвращаемое значение при закрытии окна);
- параметр `LParam`, передаваемый в обработчик событий окна — указываем здесь 0, потому что этот параметр для данного сообщения не нужен.

Значения, которые передаются в `WParam` и `LParam`, зависят от типа сообщения, поэтому нельзя сказать, что все время передаются именно те цифры, которые мы сейчас увидели.

Итак, следующий код показывает, как найти и уничтожить окно с определенным заголовком:

```
var
 wnd:HWND;
begin
 wnd:=FindWindow(nil, 'Заголовок окна');
 if wnd<>0 then
  PostMessage(wnd, WM_QUIT, 1,0);
end;
```

В *разд. 2.11* мы рассмотрим еще один метод закрытия окна и, чтобы не повторяться, там мы будем отправлять сообщение `WM_DESTROY`, и делать мы это будем с помощью функции `SendMessage`.

2.8. Шутки с мышью

2.8.1. Безумная мышь

Как показывает практика, пользователи боятся прыгающих указателей мыши. Давайте попробуем заставить указатель беспорядочно бегать по экрану:

```
var
 OK: boolean;
```

```
begin
 OK:=false;
 repeat
  Randomize;
  SetCursorPos(random(Screen.Width-1),random(Screen.Height-1));
  Sleep (5000);
 until OK;
end;
```

Здесь запускается уже знакомый вам бесконечный цикл, в котором положение указателя мыши переносится в случайную позицию с помощью WinAPI-функции `SetCursorPos`.

Этой функции нужно передать два параметра в виде целых чисел — координат `X` и `Y` новой позиции указателя. В приведенном примере передаются случайные числа от 0 до значений разрешения экрана.

2.8.2. Мышеловка

Хотите ограничить свободу перемещения курсора. Попробуйте выполнить следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 cr:TRect;
begin
 cr:=Rect(0,0,100,100);
 ClipCursor(@cr);
end;
```

В первой строке кода инициализируется переменная `cr` типа `TRect`. Туда заносится прямоугольная область размером 100×100 пикселов. Во второй строчке эта область становится диапазоном движения мыши. После этого указатель мыши окажется запертым в левом верхнем углу в квадрате размером 100×100 пикселов. Дальше мышь даже при всем желании пользователя (да и своем тоже!) выбраться не сможет.

А вот еще один интересный трюк:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 cr:TRect;
```

```
begin
 cr:=Rect(0, 0, 1, 1);
 ClipCursor(@cr);
end;
```

Здесь размер области движения мыши равен 1 пикселу по горизонтали и вертикали. В результате мышь просто замирает в этом пикселе, и курсор невозможно будет сдвинуть. Такая вот виртуальная мышеловка. Если пользователь привык работать только мышью и не знает о существовании клавиатуры, то он подумает, что компьютер завис.

2.8.3. Изменчивый указатель

Есть такая интересная WinAPI-функция — `SetSystemCursor`. У нее два параметра.

- Курсор, который надо изменить. Чтобы указать на системный курсор можно использовать функцию `GetCursor`.
- Вид системного курсора, который нужно установить. Здесь можно указать одно из следующих значений:
  - `OCR_NORMAL` — нормальный курсор в виде стрелки;
  - `OCR_IBEAM` — курсор, используемый для выделения текста;
  - `OCR_WAIT` — большие песочные часы;
  - `OCR_CROSS` — крестик;
  - `OCR_UP` — стрелка вверх;
  - `OCR_SIZE` — курсор изменения размера;
  - `OCR_ICON` — значок;
  - `OCR_SIZENWSE` или `OCR_SIZENESW` — курсор, используемый для растягивания объекта;
  - `OCR_SIZEWE` — курсор горизонтального изменения размера;
  - `OCR_SIZENS` — курсор вертикального изменения размера;
  - `OCR_SIZEALL` — курсор горизонтального и вертикального изменения размера;
  - `OCR_SIZENO` — курсор, отображаемый, когда нельзя изменять размер;
  - `OCR_APPSTARTING` — маленькие песочные часы со стрелкой.

И сразу же небольшой пример изменения текущего вида указателя мыши:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 SetSystemCursor(GetCursor, OCR_CROSS);
end;
```

Этот код изменяет текущий курсор на крестик, который используется при графическом выделении.

2.8.4. Как щелкнуть в нужном месте?

Теперь рассмотрим пример, в котором мы попробуем щелкнуть кнопкой мыши в определенной точке экрана. Этот пример может быть полезен, если вы хотите программно щелкнуть мышью в чужой программе.

```
var
 p : TPoint;
 hPointWnd : HWnd;
begin
 // Позиционируем мышь в точке 100x100
 p.X:=100;
 p.Y:=100;
 SetCursorPos(p.X, p.Y);

 // Определяем окно, которое находится в позиции 100x100
 hPointWnd := WindowFromPoint(p);

 // Посылаем события нажатия кнопки
 SendMessage(hPointWnd, WM_LBUTTONDOWN, MK_LBUTTON, MAKELONG(1, 1));
 SendMessage(hPointWnd, WM_LBUTTONUP, 0, MAKELONG(1, 1));
end;
```

Первые три строчки должны быть уже знакомы, потому что здесь мы переносим курсор мыши в позицию 100×100. После этого с помощью функции `WindowFromPoint` определяем окно, которое находится под курсором, т. е. в позиции 100×100.

Зная окно, мы можем сделать программный щелчок. Для этого найденному окну посылаем два события: `WM_LBUTTONDOWN` и `WM_LBUTTONUP`. Если вы не

знакомы с этими константами, то поясню: первая соответствует нажатию левой кнопкой мыши, а вторая — отпусканию кнопки.

Теперь посмотрим реальный пример отправки события от мыши. Какое окно можно щелкнуть, которое точно будет у вас? Ну, конечно же, кнопку **Пуск**:

```
var
 p : TPoint;
 hPointWnd : HWnd;
begin
 p.X:=20;
 p.Y:=Screen.Height-20;

 while true do
  begin
   // Устанавливаем курсор
   SetCursorPos(p.X, p.Y);

   // Отправляем окно
   hPointWnd := WindowFromPoint(p);

   // Отправляем сообщения о нажатии и отпускании кнопки
   SendMessage(hPointWnd,WM_LBUTTONDOWN,MK_LBUTTON,MAKELONG(1, 1));
   SendMessage(hPointWnd,WM_LBUTTONUP,0,MAKELONG(0, 0));

   Sleep(1000);
  end;
end;
```

Данный пример через каждую секунду щелкает по кнопке **Пуск**. Чтобы щелкнуть, мы позиционируем мышь в точку 20 по горизонтали, а по вертикали берем высоту экрана — 20. Где-то в этом месте должна находиться кнопка, если у вас панель задач вытянута вдоль нижнего края экрана.

А теперь подумаем о том, как можно сделать эффект программного перетаскивания. Для этого нужно:

1. Поместить указатель мыши в начальную позицию.
2. Узнать, какое окно находится в данной позиции.
3. Послать событие `WM_LBUTTONDOWN`.

4. Переместить курсор мыши в конечную позицию.
5. Послать событие `WM_LBUTTONUP`.

Как видите, все очень элегантно и просто.

2.9. Работа с чужими окнами

Я регулярно получаю письма с вопросами: "Как уничтожить чужое окно или изменить что-то в нем". В этом и следующих разделах я попытаюсь ответить на данный волнующий вопрос. Для начала мы напишем программу, которая будет менять заголовки всех окон на надпись "][с тобой".

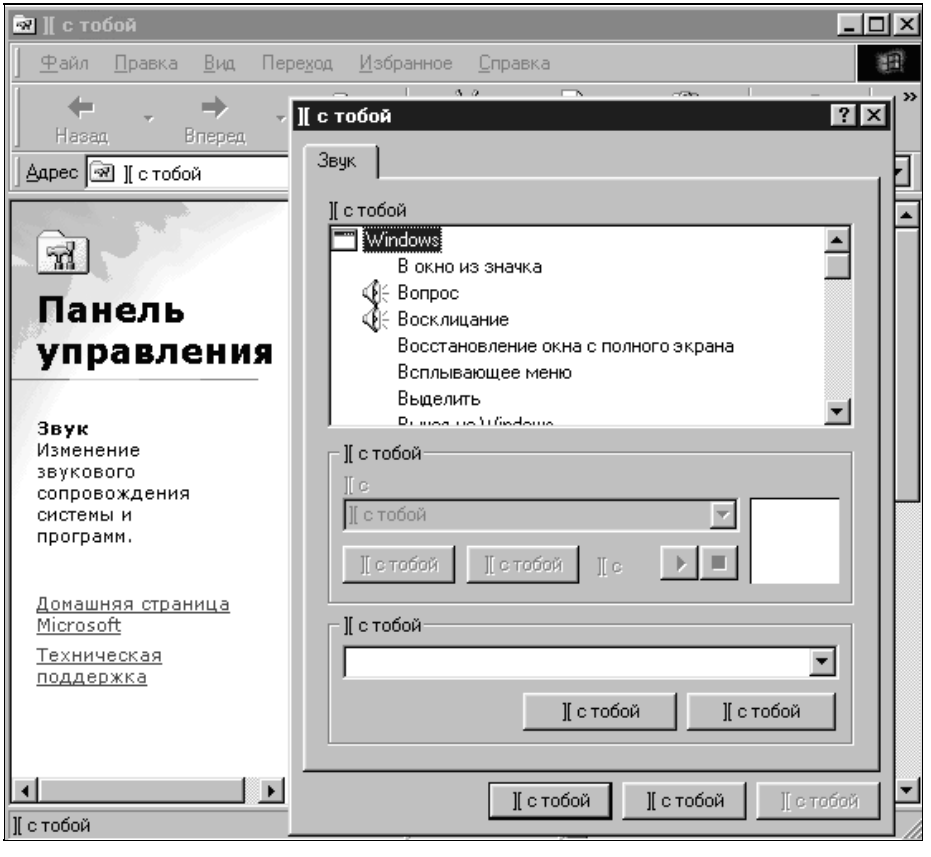


Рис. 2.12. Вид программ после запуска примера

На рис. 2.12 показан вид нескольких окон после запуска примера, который нам предстоит создать. Как видите, большинство надписей изменилось на нашу **][с тобой**. Это достигается простым путем.

1. Запускаем цикл поиска всех открытых окон.
2. В найденном окне изменяем текст заголовка и запускаем поиск его дочерних окон.
3. В найденном дочернем окне тоже изменяем текст заголовка.

Таким образом, перебираются все окна в системе и все их дочерние элементы. После этой "доброй" шутки бедный пользователь уже не сможет нормально работать за компьютером.

Эта программа будет использовать функции WinAPI. Таким образом, мы закрепим полученные знания о минимальных и невидимых приложениях.

Запустите Delphi. Если среда программирования уже запущена, то создайте новый проект. Как всегда, перед вами возникнет пустая форма. Она нам сегодня не понадобится, поэтому удалим ее. Для этого в меню **Project** выберите пункт **Remove from Project**. Перед вами появится окно, в котором нужно выделить имя формы и нажать кнопку **OK**. Вас попросят подтвердить удаление, на что отвечаем твердым и громким согласием.

Теперь нужно открыть исходный файл самого проекта. Для этого выберите в меню **Project** пункт **View Source** (можно еще в окне **Project Manager** щелкнуть правой кнопкой мыши на названии программы и в появившемся контекстном меню выбрать пункт **View Source**). Здесь можно удалять все, кроме первой строки:

```
program Project1;
```

Ее можно оставить, а потом вставить код из листинга 2.3.

Листинг 2.3. Код программы замены заголовков всех окон

```
program Project1;

uses
  Windows,
  Messages;

// Эта функция вызывается, когда найдено дочернее окно
function EnumChildWnd(h: hwnd): BOOL; stdcall;
```

```
begin
 SendMessage(h,WM_SETTEXT,0,lparam(LPCTSTR(']['  с тобой')));
 Result:=true;
end;

// Эта функция вызывается, когда найдено главное окно
function EnumWindowsWnd(h: hwnd): BOOL; stdcall;
begin
 SendMessage(h,WM_SETTEXT,0,lparam(LPCTSTR(']['  с тобой')));
 EnumChildWindows(h,@EnumChildWnd,0);
end;

begin
 // Запускаем бесконечный цикл
 while true do
  begin
   // Запускаем перечисление всех окон
   EnumWindows(@EnumWindowsWnd,0);

   // Делаем задержку в 1000 мс
   Sleep(1000);
  end;
end.
```

Все, наша программа готова. Теперь можно создать исполняемый файл и даже запустить его, чтобы посмотреть результат. Если вы пользуетесь Windows 9*x*, то сможете увидеть результат в полном объеме. В Windows 2000/XP пример работает, но заголовки изменяются далеко не на всех окнах. Учтите, что для Windows окнами являются и почти все элементы управления (подробнее см. далее), и вместо набранного текста в поле ввода вы можете увидеть надпись "][с тобой".

Теперь подробно разберем код примера. Вы уже знаете, что после ключевого слова `uses` пишут подключаемые модули. У нас их будет всего два: `Windows` и `Messages`. В этих двух модулях идет описание основных WinAPI-функций (модуль `Windows`) и сообщений операционной системы (модуль `Messages`). Из этих модулей Delphi узнает о существовании функций WinAPI и способах работы с ними.

Дальше идет описание функций `EnumChildWnd` и `EnumWindowsWnd`. О них мы поговорим немного позже, а сейчас перейдем на начало программы.

После старта программа сразу же запускает бесконечный цикл `while`. В данном случае в качестве условия указано `true`, а значит, цикл будет выполняться бесконечно, потому что `true` никогда не станет равным `false`.

Внутри цикла вызывается функция — `EnumWindows`. Это WinAPI-функция, которая используется для перечисления всех запущенных окон. В качестве единственного параметра ей нужно передать адрес другой функции, которая будет вызываться каждый раз, когда найдено какое-нибудь окно. Для этого служит функция `EnumWindowsWnd`. Таким образом, каждый раз, когда `EnumWindows` найдет окно, будет выполняться код, написанный в `EnumWindowsWnd`. Этот код выглядит вот так:

```
// Эта функция вызывается, когда найдено главное окно
function EnumWindowsWnd(h: hwnd): BOOL; stdcall;
begin
 SendMessage(h,WM_SETTEXT,0,lparam(LPCTSTR(']['  с тобой')));
 EnumChildWindows(h,@EnumChildWnd,0);
end;
```

У функции `EnumWindowsWnd` есть один параметр — идентификатор найденного окна. Этого достаточно, чтобы мы могли изменить его заголовок. Есть такая WinAPI-функция `SendMessage`, которая посылает сообщения. У нее 4 параметра:

- первый — идентификатор окна, которому надо отослать сообщение. Этот идентификатор мы получаем в качестве параметра `EnumWindowsWnd`;
- второй параметр — тип сообщения. Указано `WM_SETTEXT`. Сообщения данного типа заставляют окно сменить заголовок или свое содержание;
- третий — для данного сообщения должен быть `0`;
- четвертый параметр — новое имя или текст окна.

Итак, с помощью `SendMessage` мы посылаем найденному окну сообщение о том, что надо поменять текст. Новый текст указан в четвертом параметре функции `SendMessage`.

В *разд. 2.2* мы видели еще один метод изменения текста и заголовков элементов управления — `SetWindowText`. Этот метод будет работать везде, где есть текст окна, и в любой ОС (по крайней мере, известной мне на данный момент, а про будущие — гарантировать не могу). Но в данном случае я решил использовать для изменения заголовка отправку сообщений. Просто так захотелось.

После того как изменен текст главного окна, нужно запустить поиск дочерних окон. Обычно каждое окно имеет еще очень много разных элементов управления (например, кнопок), текст на которых тоже можно изменить. Именно поэтому нужно вызывать функцию `EnumChildWindows`. Эта функция ищет все элементы управления внутри указанного окна. У нее всего 3 параметра:

- идентификатор окна, дочерние элементы которого нужно искать;
- адрес функции обратного вызова, которая будет вызываться каждый раз, когда найдено дочернее окно;
- просто число, которое может быть передано в функцию обратного вызова.

Как вы можете заметить, работа функции `EnumChildWindows` похожа на `EnumWindows`, только если вторая ищет окна во всей системе, то первая внутри указанного окна (т. е. ищет элементы управления внутри окна).

Функция обратного вызова `EnumChildWnd` выглядит следующим образом:

```
// Эта функция вызывается, когда найдено дочернее окно
function EnumChildWnd(h: hwnd): BOOL; stdcall;
begin
 SendMessage(h,WM_SETTEXT,0,lparam(LPCTSTR(']['  с тобой')));
 Result:=true;
end;
```

Здесь мы так же изменяем текст найденного окна с помощью функции `SendMessage`. После этого результату присваивается значение `true`, чтобы поиск продолжился.

В принципе программу можно считать законченной, но у нее есть один недостаток, о котором нельзя умолчать. Допустим, что программа нашла окно, и начала перечисление в нем дочерних окон, и в этот момент окно закрыли. Программа пытается послать сообщение окну об изменении текста, а окно уже не существует, и возникает ошибка выполнения. Чтобы этого не происходило, в функциях обратного вызова в самом начале нужно поставить проверку на правильность полученного идентификатора окна:

```
if h=0 then exit;
```

Вот теперь приложение можно считать законченным и абсолютно рабочим.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 2\][\ с тобой\ вы можете найти пример этой программы и цветные рисунки этого раздела.

2.10. Дрожь в ногах

Теперь немного изменим написанный в прошлом разделе пример и сделаем программу, которая будет перебирать все окна в системе и изменять их размеры. Можете открыть файл предыдущего примера и подкорректировать его или написать новый с кодом из листинга 2.4.

Листинг 2.4. Код программы изменения размеров и координат всех окон

```
program Project1;

uses
  Windows,
  Messages;

// Функция-ловушка
function EnumWindowsWnd(h: hwnd): BOOL; stdcall;
var
 rect:TRect;
 index:Integer;
begin
 if not IsWindowVisible(h) then
  begin
   Result:=true;
   exit;
  end;

 // Получаем размеры найденного окна
 GetWindowRect(h,rect);

 // Генерируем случайное число
 index:=random(2);

 if index=0 then
  begin
   // Если оно 0, то увеличиваем...
   rect.Top:=rect.Top+3;
```

```
   rect.Left:=rect.Left+3;
  end
 else
  begin
   // Иначе уменьшаем...
   rect.Top:=rect.Top-3;
   rect.Left:=rect.Left-3;
  end;

 MoveWindow(h,rect.Left,rect.Top,rect.Right-rect.Left,
            rect.Bottom-rect.Top,true);
 Result:=true;
end;

// Основной код
var
 h:THandle;
begin
 // Запускаем цикл
 while true do
  begin
   // Запускаем перечисление всех окон
   EnumWindows(@EnumWindowsWnd,0);

   // Делаем задержку в 1000 мс
   Sleep(1000);
  end;
end.
```

Если вы посмотрите на этот код, то заметите, что основное содержание программы не изменилось. Здесь так же запускается бесконечный цикл, внутри которого вызывается функция перебора всех окон и делается задержка в 1000 миллисекунд. В этой части абсолютно никаких изменений нет.

Теперь посмотрим на функцию-ловушку `EnumWindowsWnd`, которая будет вызываться каждый раз, когда найдено окно. Тут первой вызывается функция `IsWindowVisible`. Эта функция проверяет, является ли найденное окно ви-

димым. Если нет, то переменной `Result` присваивается значение `true`, и происходит выход из ловушки. Если `Result` равно `true`, то поиск следующего окна будет продолжен, иначе он остановится, и очередное окно не будет найдено.

После этого вызывается функция `GetWindowRect`. Этой функции в первом параметре передается идентификатор найденного окна, а она возвращает во втором параметре размеры этого окна в виде переменной типа `TRect`.

Получив габариты окна, генерируется случайное число с помощью функции `random`. Этой функции надо передать только значение максимально допустимого числа, а она вернет случайное число от `0` до указанного значения. После этого я проверяю, если число после округления (`index` объявлена как целая переменная) равно `0`, то я увеличиваю свойства `Top` и `Left` структуры `rect` на `3`. Иначе эти значения уменьшаются.

Изменив значения структуры, в которой хранились габариты найденного окна, это окно перемещается с помощью функции `MoveWindow`. Эта функция имеет 5 параметров:

- идентификатор окна, позицию которого надо изменить (`h`);
- новая позиция левого края (`rect.Left`);
- новая позиция верхнего края (`rect.Top`);
- новая ширина (`rect.Right-rect.Left`);
- новая высота (`rect.Bottom-rect.Top`).

Ну, и напоследок, переменной `Result` присваивается значение `true`, чтобы поиск продолжился.

Получается, что если запустить программу, то мы увидим дрожание всех запущенных окон. Программа будет перебирать все окна и случайным образом изменять их положение. Попробуйте запустить программу и посмотреть этот эффект в действии (дрожат только неразвернутые окна).

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 2\Дрожь\ вы можете увидеть пример этой программы.

2.11. Оконные страсти

Теперь мы напишем еще один пример программы, которая будет работать с чужими окнами. Она будет искать в системе определенное окно и уничтожать его. На первый взгляд мы должны воспользоваться методом, описан-

ным в предыдущем разделе, но это только на первый взгляд, потому что есть способ намного проще. Давайте создадим приложение, которое будет искать в системе окно с заголовком **Microsoft Word** и уничтожать его.

Создайте новое приложение и перенесите на форму только одну кнопку с заголовком **Найти и уничтожить**. В обработчике нажатия этой кнопки пишем следующий код:

```
procedure TForm1.FindAndDestroyButtonClick(Sender: TObject);
var
 h:hWnd;
begin
 h:=FindWindow(nil, 'Microsoft Word');
 if h=0 then exit;
 SendMessage(h, WM_DESTROY, 0,0);
end;
```

В данном случае для поиска мы используем функцию `FindWindow`, которую не раз уже применяли в этой главе.

Итак, единственное, что нам нужно знать для реализации примера, — заголовок окна, но знать его надо точно. В этом и состоит главный недостаток данного подхода программы, потому что Microsoft Word с открытым документом имеет заголовок ***Имя документа*** **- Microsoft Word**. Если нет открытых документов, то заголовок простой — **Microsoft Word**. В этом случае функция `FindWindow` однозначно определит окно, которое надо найти. Иначе программа возвратит `0`. Как видите, поиск окна по заголовку нельзя назвать надежным, но класс окна мы не всегда можем определить.

После того как мы выполнили функцию `FindWindow`, проверяется возвращенное значение. Если оно равно `0`, то ничего не найдено, и надо выходить из программы. Если нет, то найденному окну посылается сообщение `WM_DESTROY` (уничтожиться) с помощью функции `SendMessage`.

Вот и все. Пример оказался очень простым, и мне больше уже нечего добавить.

> ***ПРИМЕЧАНИЕ***
>
> На компакт-диске в каталоге \Примеры\Глава 2\Уничтожение окна\ вы можете найти пример этой программы.

Теперь рассмотрим еще один интересный пример, в котором все программы мы переместим внутрь своего окна. Получится эффект, в котором рабочий стол вместе со всеми программами переместится внутрь главной формы вашей программы.

Чтобы реализовать этот пример, необходимо запустить перебор всех окон в системе с помощью `EnumWindows`, а в функции обратного вызова написать следующий код:

```
function EnumWindowsProc(h: hwnd; lparam:Integer): BOOL; stdcall;
begin
 if IsWindowVisible(h) then
  SetParent(h, Form1.Handle);

 Result:=true;
end;
```

В самом начале происходит проверка, если найденное окно видимо, то перемещаем его внутрь главной формы нашей программы. Для этого вызывается функция `SetParent`, которая в качестве главной формы найденному окну устанавливает главную форму нашей программы.

Как видите, все очень просто. Извините, что не могу сделать снимок результата, но, переместившись внутрь формы, Windows становится практически не рабочей системой, а после закрытия программы уж точно ничего нельзя сделать.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 2\WindowsInForm\ вы можете найти пример этой программы.

2.12. Буфер обмена

В большинстве приложений, если есть кнопка вставки из буфера обмена, то она должна быть доступна, только если в буфере есть данные подходящего формата. Если в буфере находятся несовместимые данные, то оставлять кнопку активной будет некорректно. Какая бы программа не поместила в буфер данные, все остальные должны тут же проверить формат помещенных в буфер данных и отреагировать соответствующим образом.

Некоторые начинающие программисты не делают такой проверки, потому что она может вызвать сложности, поэтому проверяют допустимость вставки только после нажатия соответствующей кнопки. Это неправильно. И мы сначала научимся следить за изменениями буфера обмена, а потом я покажу, как легко превратить этот пример в шуточный.

Давайте разберемся, как работают наблюдатели за буфером обмена. В системе существует цепочка из функций. Чтобы следить за изменениями, каждое приложение должно зарегистрироваться в этой цепочке. ОС Windows знает адрес только первой функции из цепочки зарегистрированных программ. Когда происходит изменение в буфере, то система вызывает эту функцию, а она уже должна вызвать следующую функцию из цепочки.

Итак, для начала регистрируем наше окно в системе в качестве наблюдателя с помощью функции `SetClipboardViewer`, которая возвращает указатель на следующий наблюдатель в системе. Таким образом, программы сами строят цепочку. Когда в буфере произошли изменения, система посылает сообщение первому в списке наблюдателей, а он уже перенаправляет его следующему. Таким образом, приложения сами без вмешательства системы сообщают друг другу об изменениях. Это и хорошо, и плохо, ведь какой-нибудь программист может написать программу, не перенаправляющую событие следующему наблюдателю, и цепочка разорвется.

Давайте рассмотрим все сказанное на практике. Создайте новое приложение и поместите на главную форму кнопку `TButton` и компонент типа `TRichEdit`, в который будут вставляться данные из буфера. На кнопке напишем текст **Вставить**, по нажатию этой кнопки будет происходить вставка из буфера обмена.

Сразу же добавим в раздел `uses` модуль `Clipbrd`, в котором объявлены функции.

В Delphi нет готового обработчика события, который мог бы отвечать на изменения буфера обмена, поэтому придется описать самостоятельно. Для этого в разделе `private` главной формы пишем следующий код:

```
private
  FNextViewer:HWnd;
  procedure WMChangeCBChain(var Msg: TWMChangeCBChain);
                                    message WM_CHANGECBCHAIN;
  procedure WMDrawClipboard(var Msg: TWMDrawClipboard);
                                    message WM_DRAWCLIPBOARD;
```

Здесь у нас объявлена переменная `FNextViewer` типа `HWnd`, в которой будем хранить указатель на следующее в системе окно, зарегистрированное в качестве наблюдателя.

Процедура `WMChangeCBChain` является обработчиком системного события `WM_CHANGECBCHAIN`. Об этом говорит добавленный в конце объявления процедуры оператор `message` с названием системного события. Это событие генерируется каждый раз, когда изменяется очередь, а именно происходит

удаление или добавление какого-то наблюдателя. Чуть позже мы рассмотрим наши действия, которые должны быть в этом обработчике.

Процедура `WMDrawClipboard` будет обработчиком системного события `WM_DRAWCLIPBOARD`. Оно генерируется каждый раз, когда в буфере обмена изменились данные, и его нужно перерисовать.

Прежде чем рассматривать процедуры `WMChangeCBChain` и `WMDrawClipboard`, давайте зарегистрируем свое приложение в качестве наблюдателя за буфером. Для этого создадим для формы обработчик события `OnCreate` и напишем в нем вызов функции `SetClipboardViewer`:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 FNextViewer := SetClipboardViewer(Handle);
end;
```

Функция `SetClipboardViewer` получает в качестве параметра идентификатор окна, которое регистрируется как наблюдатель за буфером обмена. Именно этому окну будут направляться сообщения `WM_CHANGECBCHAIN` и `WM_DRAWCLIPBOARD`, и вы должны реализовать корректные обработчики событий. В качестве результата функция возвращает идентификатор окна, следующего за вашим в цепочке наблюдателей. После обработки события вы должны сообщить этому окну о том, что в буфере произошли изменения, т. е. передать событие по цепочке.

Теперь посмотрим на реализацию функции `WMChangeCBChain`, которая вызывается при изменении очереди наблюдателей за буфером обмена:

```
procedure TForm1.WMChangeCBChain(var Msg: TWMChangeCBChain);
begin
 if Msg.Remove = FNextViewer then
  FNextViewer := Msg.Next
 else
  SendMessage(FNextViewer, WM_CHANGECBCHAIN, Msg.Remove, Msg.Next);
end;
```

В качестве параметра функции мы получаем переменную `Msg` типа структуры `TWMChangeCBChain`. В этой структуре нас будут интересовать два свойства: `Remove` и `Next`. Первое свойство указывает на окно, которое должно быть удалено из очереди, а второе указывает на окно, следующее за ним. Наша задача проверить, если удаляемое окно является наблюдателем, которому мы посылаем сообщения, то должна быть произведена замена следующим наблюдателем:

```
FClipboardOwner := Msg.Next
```

Если удаляется тот наблюдатель, которому перенаправляем сообщения мы, то отправляем нашему наблюдателю событие WM_CHANGECBCHAIN, указав в качестве параметров удаляемое и следующее окно:

```
SendMessage(FClipboardOwner, WM_CHANGECBCHAIN,
   Msg.Remove, Msg.Next);
```

Следующее окно сделает ту же проверку, и таким образом очередь будет перестроена без вмешательства системы. Мы, наблюдатели за буфером, должны сами следить за цепочкой и реагировать на ее изменения.

Теперь посмотрим на реализацию функции WMDrawClipboard, которая вызывается при изменении содержимого буфера обмена:

```
procedure TForm1.WMDrawClipboard(var Msg: TWMDrawClipboard);
var
 i:Integer;
begin
 Button1.Enabled:=false;
 SendMessage(FNextViewer, WM_DRAWCLIPBOARD, 0, 0);
 Msg.Result := 0;
 for i := 0 to Clipboard.FormatCount - 1 do
 if Clipboard.HasFormat(CF_TEXT) then
  begin
   Button1.Enabled:=true;
  end;
end;
```

Первым делом мы сделали кнопку вставки (Button1) недоступной. Мы поменяем состояние на активное, только если в буфере будут храниться данные нужного нам формата.

После этого с помощью WinAPI-функции SendMessage отправляем сообщение WM_DRAWCLIPBOARD следующему окну в очереди наблюдателей, чтобы и оно смогло отреагировать на изменения.

Теперь можно обрабатывать сообщение самостоятельно. Для этого запускаем цикл, в котором будут перебираться все форматы, к которым можно получить доступ, и только если формат соответствует CF_TEXT, делаем кнопку вставки активной.

Помимо текстового формата можно искать:

- CF_BITMAP — растровую картинку;
- CF_METAFILEPICT — векторную графическую картинку в формате Windows Metafile;

- `CF_PICTURE` — объект типа `TPicture`;
- `CF_COMPONENT` — компонент Delphi.

Это далеко не полный список. С полным списком вы можете познакомиться в файле помощи Delphi или Windows API.

Чтобы пример был более наглядным для тестирования, по событию `OnClick` для кнопки **Вставить** напишем код, который будет вставлять данные из буфера обмена в `Memo`-компонент. Для этого достаточно вызвать метод `PasteFromClipboard` компонента `TRichEdit`:

```
RichEdit1.PasteFromClipboard;
```

Пока что наш пример очень познавательный, но абсолютно безобидный. Теперь посмотрим, как легко можно превратить этот безобидный пример в шутку:

```
procedure TForm1.WMDrawClipboard(var Msg: TWMDrawClipboard);
var
 i:Integer;
begin
 Button1.Enabled:=false;

 for i := 0 to Clipboard.FormatCount - 1 do
  if Clipboard.HasFormat(CF_TEXT) then
   begin
    Button1.Enabled:=true;
    Clipboard.AsText:='You are Hacked!!!';
   end;

 SendMessage(FNextViewer, WM_DRAWCLIPBOARD, 0, 0);
 Msg.Result := 0;
end;
```

В этом коде если найден нужный формат, то помимо активации кнопки мы помещаем в буфер текст "You are Hacked!!!". Таким образом, какой бы текст не скопировала любая программа, он будет заменен на наше злое сообщение.

Для большей эффектности можно сделать окно невидимым и подбросить кому-нибудь из друзей.

Можно проказничать и тем, что не передавать событие об изменении содержимого буфера другим приложениям. Таким образом может быть нарушена цепочка, и какая-то программа не отработает событие, а значит, со-

стояние кнопки вставки из буфера в других программах (которые находятся в цепочке после нашего) не изменится. Но это может привести к плачевным последствиям. Например, программа может принимать текст, а кнопка в начальной стадии была активной. При изменении содержимого на картинку состояние не изменилось. Теперь пользователь пытается вставить картинку как текст, и программа, не сумев обработать буфер, может завершиться аварийно.

Такой нежелательный эффект может произойти в тех программах, которые надеются на корректность цепочки и не проверяют формат буфера непосредственно перед работой с данными в буфере. Некоторые считают, что лишний раз проверять не стоит, ведь кнопка доступна только при корректном формате, но это утверждение верно, только если цепочка корректна. Чтобы этого не произошло, всегда в своих программах проверяйте формат перед вставкой:

```
var
 i:Integer;
begin
 for i := 0 to Clipboard.FormatCount - 1 do
 if Clipboard.HasFormat(CF_TEXT) then
   RichEdit1.PasteFromClipboard;
end;
```

В этом примере вставка из буфера произойдет только после проверки корректности формата данных, и такой код более надежен к нарушению цепочки.

В нашем примере абсолютно безобидный буфер обмена стал предметом шутки. Главное было подойти к вопросу с энтузиазмом. Таким образом, практически любая вещь может показаться в наших глазах совершенно в новом свете, и искусство хакера состоит в том, чтобы видеть такие вещи, использовать и удивлять своих коллег или просто знакомых.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 2\Clipboard\ вы можете найти пример этой программы.

2.13. Служба сообщений

В системах на базе Windows NT есть очень удобная служба — служба обмена сообщениями. Вы можете отправлять любому компьютеру в сети текстовые сообщения, и если соответствующая служба работает, то компьютер отобра-

зит окно с полученным текстом. Чтобы передать текст на другой компьютер, достаточно написать в командной строке следующую команду:

```
NET SEND [Адрес или имя компьютера] [Текст сообщения]
```

В качестве адреса можно указывать как имя компьютера, так и IP-адрес. Вот примеры отправки сообщения "Ты под атакой хакеров" на IP-адрес 192.168.0.1 и на компьютер с именем NetComputer:

```
NET SEND 192.168.0.1 Ты под атакой хакеров
NET SEND NetComputer Ты под атакой хакеров
```

После выполнения этой команды, на компьютере пользователя, которому вы посылали сообщение, появляется окно с отправленным текстом (рис. 2.13). Таким образом удобно обмениваться короткими сообщениями в локальной сети на любые расстояния без установки дополнительного софта.

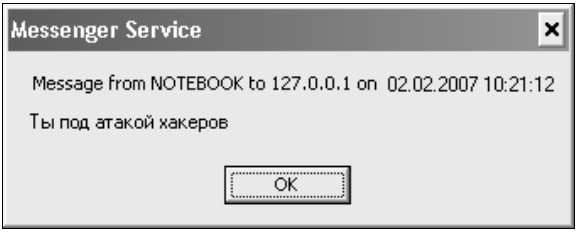


Рис. 2.13. Окно с сообщением, отправленным командой `NET SEND` с моего компьютера самому себе

Самое интересное в этой технологии, что она абсолютно не защищена от флуда. Вы легко можете отправить хоть сотню сообщений, и все они достигнут получателя. Мой зам. начальника очень часто донимал меня вопросами через `NET SEND`, потому что находился в другом здании (на расстоянии где-то 500 метров) и у него постоянно возникали вопросы по программированию. Мне стали надоедать сообщения, которые появляются с постоянной периодичностью, и я написал небольшую программу, которая бомбила зама сообщениями. Через пять минут завязалась самая настоящая война сообщений.

Давайте напишем небольшую программу, которая будет бомбить любой адрес в сети. Создайте новый проект. На главную форму нужно поместить три поля ввода и кнопку с заголовком **Бомбить**. Поля ввода будут иметь следующие имена:

- `AddressEdit` — поле для ввода IP-адреса или имени машины, которую надо бомбить;

- `MessageEdit` — текст, который будет находиться в сообщениях;
- `MessageNumberEdit` — количество отправляемых сообщений.

На рис. 2.14 вы можете увидеть форму моей будущей программы.

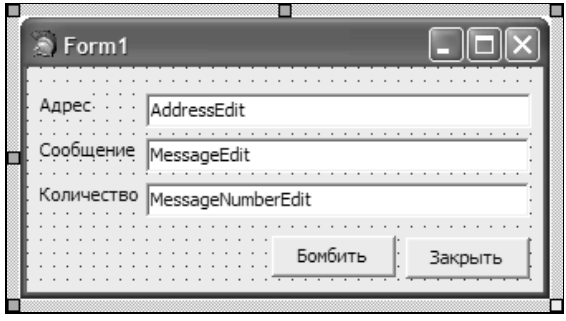


Рис. 2.14. Форма будущей программы

А теперь посмотрим на обработчик события `OnClick` для кнопки бомбардировки:

```
procedure TForm1.Button2Click(Sender: TObject);
var
 i:Integer;
begin
 for i:=1 to StrToIntDef(MessageNumberEdit.Text, 1) do
  begin
   WinExec(PChar('NET SEND '+
           AddressEdit.Text+' '+
           MessageEdit.Text),
           SW_SHOW);
   Sleep(1000);
  end;
end;
```

Здесь запускается цикл, в котором выполняется команда `NET SEND` с помощью функции `WinExec`. Между отправками происходит задержка в 1 секунду (функция `Sleep`), чтобы хоть немного было времени на закрытие появляющихся окон.

Обратите внимание, что при преобразовании строки в число мы используем функцию `StrToIntDef`, чтобы не возникло ошибки, если пользователь введет в количество отправляемых сообщений недопустимое значение. В этом случае будет отправлено только одно сообщение.

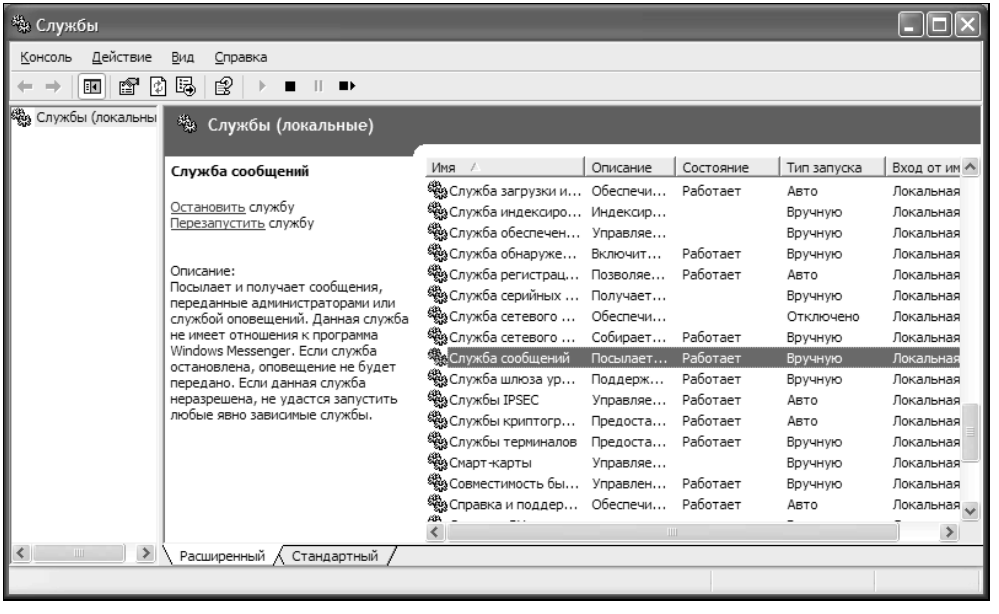


Рис. 2.15. Оснастка Службы.
Выделенная строка соответствует службе сообщений

Если вас начали бомбить, то первым делом советую выдернуть из компьютера сетевой шнур, чтобы немедленно прекратить атаку. После этого выберете меню **Пуск | Панель управления | Администрирование | Службы**, и перед вами откроется окно настройки служб (рис. 2.15). Найдите здесь службу сообщений (Messenger) и остановите ее. Для этого есть несколько вариантов:

- щелкнув правой кнопкой мыши, выберите в появившемся меню пункт **Стоп**;
- дважды щелкните по службе сообщений и в появившемся окне на вкладке **Общие** нажмите кнопку **Стоп**;
- выделите службу сообщений и выберите меню **Действие | Стоп**.

После этого можете возвращать кабель на место. Таким образом, вы остаетесь без возможности обмена сообщениями, зато никто не сможет забомбить ваш компьютер.

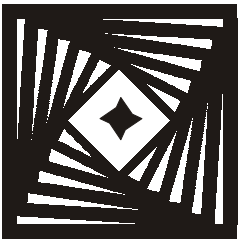
Я вообще рекомендую держать службу отключенной. Для этого уберите с нее признак автоматического запуска. Чтобы сделать это, снова запустите оснастку Службы, дважды щелкните по службе сообщений и в появившемся окне в выпадающем списке **Тип запуска** выберите пункт **Вручную**. В этом случае, служба не будет запускаться при старте системы, но если необходимо, вы вручную можете ее запустить.

В Windows XP данная служба уже не запускается по умолчанию. Если я не ошибаюсь, то автоматический старт по умолчанию был только в Windows 2000, но после того, как выяснилось, что защититься от бомбардировки очень сложно, служба была отключена.

Примечание

На компакт-диске в каталоге \Примеры\Глава 2\NetSend\ вы можете увидеть пример этой программы.

Глава 3



Система

В этой главе мы рассмотрим разные системные утилиты. Сюда вошли примеры программ, способных следить за происходящим в системе. Это уже не просто создание программ-приколов, это работа с системой, хотя и в предыдущих главах мы уже начали работать с ОС и ее функциями. Как я уже говорил, любой хакер — это профессионал, и он должен знать и уметь оперировать "внутренностями" той операционной системы, в которой он работает.

При написании главы я подразумевал, что вы программируете и работаете в Windows. В данной главе я попробую научить вас лучше понимать эту систему. Но не бойтесь, вас не будут загружать теорией, я постараюсь делать упор на практических примерах. Если вы уже читали мои труды, то знаете мой стиль. Я всегда говорю, что теория — загружает мозги, и только практика дает знания и бесполезна, когда вы не умеете пользоваться теорией. Грош цена тем знаниям, которые не знаешь, как применить на практике. Такие знания быстро забываются. Именно поэтому все главы данной книги наполнены практическими примерами, и эта — не исключение.

Я покажу несколько интересных примеров, и мы подробно разберем их. Таким образом, мы осветим некоторые особенности работы с ОС Windows, и вы поймете, как применять эти особенности на практике. Надеюсь, что это вам поможет в работе.

3.1. Подсматриваем пароли, спрятанные под звездочками

Как увидеть пароль, спрятанный под звездочками? Иногда так раздражает набирать одно и то же по несколько раз: то раскладка не та, то клавиша <Caps Lock> нажата. Для того чтобы открыть взгляду набираемое, есть

очень много разных специальных программ. Но вы же не думаете, что я буду вас отправлять к таким программам в своей книге? Конечно же, сейчас мы разберем, как самому написать подобную программу.

Тут необходимо сделать одно замечание — в системе Windows, начиная с 2000, и в хорошо защищенных программах вы не сможете увидеть пароли из-под звездочек. Например, в окне смены пароля для учетных записей пароли не появятся. Проблема в том, что в таких программах звездочки — это бутафория. Реально под ними ничего нет, потому что система сохраняет у себя только результат одностороннего шифрования. Обратного дешифрования не существует, и пароль в чистом виде нигде не хранится, поэтому под звездочками нечего хранить. Для авторизации введенный пользователем пароль шифруется по тому же алгоритму и сравнивается с зашифрованной версией, сохраненной в базе. Если результаты одинаковы, то пользователю вход разрешен, иначе введенный пароль считается некорректным.

Возвращаемся к нашему примеру и посмотрим, как же можно убрать звездочки. Пример очень интересный и познавательный, поэтому может пригодиться вам в будущем.

Программа будет состоять из двух частей. Первый файл — запускаемый — будет загружать другой файл — динамическую библиотеку в память. Эта библиотека будет регистрироваться в системе в качестве обработчика системных сообщений. Этот обработчик будет ожидать, когда пользователь не щелкнет в каком-нибудь окне кнопкой мыши, удерживая клавишу <Ctrl>. Как только такое событие произойдет, мы сразу же должны будем получить текст этого окна и конвертировать его из звездочек в нормальный текст. На первый взгляд все выглядит сложным, но реально вы сможете реализовать все за десять минут.

Для этого примера я написал dll-файл, создание которого будет сейчас расписано на ваших глазах. Хотя нет, ваши глаза я трогать не буду, и тем более писать на них я не собирался. Все будет расписано на страницах книги. Ничего особо визуального мы сегодня делать не будем — только сухое программирование.

Для начала создадим новый проект. Но не обычное приложение, какие мы использовали до этого, а проект динамической библиотеки (dll). Для этого нужно выбрать команду **File | New | Other** (для Delphi 5 — просто **File | New**). Перед вами откроется окно, как на рис. 3.1. Найдите элемент **DLL Wizard** и дважды щелкните на нем. Delphi создаст пустой проект динамической библиотеки. Сразу же нажмите кнопку **Save**, чтобы сохранить проект. В качестве имени введите hackpass, так же будет названа и библиотека.

Теперь самое вкусное (просто пальчики оближешь) — удалите весь код, который написала система Delphi, а вместо этого введите содержимое листинга 3.1.

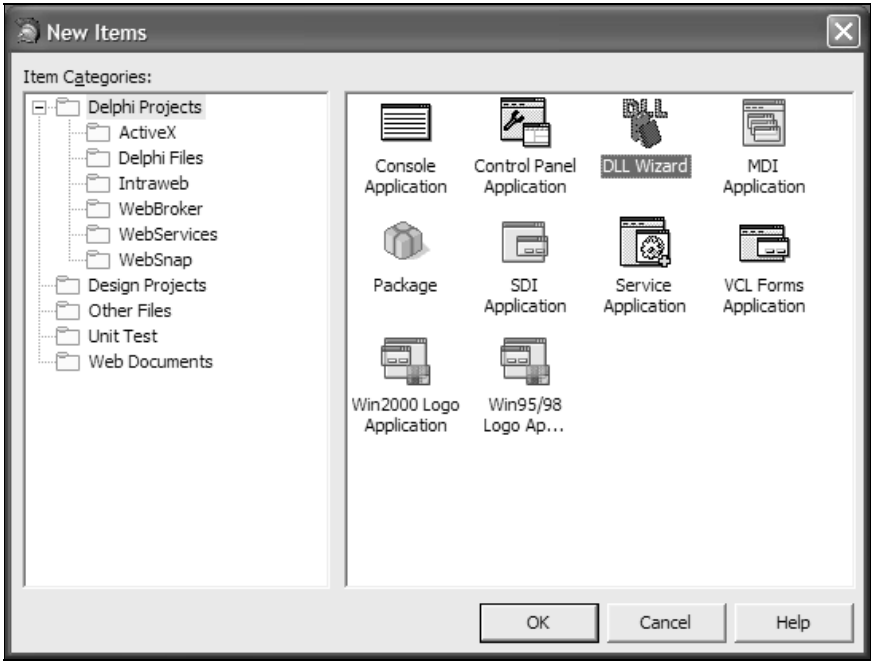


Рис. 3.1. Создание новой динамической библиотеки

Листинг 3.1. Код динамической библиотеки

```
library hackpass;

uses Windows, Messages;
var
 SysHook : HHook = 0;
 Wnd : Hwnd = 0;

function SysMsgProc(code : integer; wParam : word;
 lParam : longint) : longint; stdcall;
begin
 // Передаем сообщение другим ловушкам в системе
 CallNextHookEx(SysHook, Code, wParam, lParam);
 // Проверяем сообщение
 if code = HC_ACTION then
```

```
  begin
 // Получаем идентификатор окна, сгенерировавшего сообщение
   Wnd := TMsg(Pointer(lParam)^).hwnd;

  // Проверяем тип сообщения.
  // Если была нажата левая кнопка мыши
  // и удержана клавиша <Ctrl>, то...
   if TMsg(Pointer(lParam)^).message = WM_LBUTTONDOWN then
   if ((TMsg(Pointer(lParam)^).wParam and MK_CONTROL) = MK_CONTROL) then
    begin
     // Убрать в окне, отправившем сообщение, звездочки
     SendMessage(Wnd, em_setpasswordchar, 0, 0);
     // Перерисовать окно
     InvalidateRect(Wnd, nil, true);
    end;
  end;
end;

// Процедура запуска
procedure RunStopHook(State : Boolean) export; stdcall;
begin
 // Если State = true, то...
 if State=true then
  begin
  // Запускаем ловушку
   SysHook := SetWindowsHookEx(WH_GETMESSAGE,
      @SysMsgProc, HInstance, 0);
  end
 else// Иначе
  begin
   // Отключить ловушку
   UnhookWindowsHookEx(SysHook);
   SysHook := 0;
  end;
end;
```

```
exports RunStopHook index 1;

begin
end.
```

Самое основное в нашей библиотеке — это процедура `RunStopHook`. Ей передается один только параметр. Если он равен `true`, то регистрируется ловушка, которая будет ловить все сообщения, предназначенные Windows. Для этого используется функция `SetWindowsHookEx`. У этой функции должно быть четыре параметра:

- тип ловушки; указан `WH_GETMESSAGE`, такая ловушка ловит все сообщения;
- указатель на функцию, которой будут пересылаться сообщения Windows;
- указатель на приложение;
- идентификатор потока; если параметр равен нулю, то используется текущий.

В качестве второго параметра указано имя функции `SysMsgProc`. Она так же описана в этой dll-библиотеке. Но ее мы рассмотрим чуть позже. Значение, которое возвращает функция `SetWindowsHookEx`, сохраняется в переменной `SysHook`. Оно потом нам понадобится, когда мы будем отключать ловушку.

Если наша процедура `RunStopHook` получила в качестве параметра значение `false`, то нужно отключить ловушку. Для этого вызывается функция `UnhookWindowsHookEx`, которой передается значение переменной `SysHook`. Это то значение, которое мы получили при создании ловушки.

Процедура `RunStopHook` объявлена как экспортная:

```
exports RunStopHook index 1;
```

Это означает, что она будет доступна из внешних программ. После ее имени стоит ключевое слово `index` и значение `1`. Именно по этому индексу мы и будем обращаться к данной процедуре. Обращение по индексу лучше и быстрее, чем по имени, поэтому я рекомендую использовать именно этот метод.

Теперь давайте посмотрим на функцию `SysMsgProc`, которая будет вызываться при наступлении системных событий.

В первой строке пойманное сообщение передается остальным ловушкам, установленным в системе с помощью `CallNextHookEx`. Если этого не сделать, то другие обработчики не смогут узнать о наступившем событии, и система будет работать некорректно. Это тот же эффект, что и с буфером обмена, который мы рассматривали в *разд. 2.12*.

Далее проверяется тип полученного сообщения. Нам нужно обрабатывать событие нажатия кнопки мыши, значит, параметр `code` должен быть равен `HC_ACTION`, сообщения другого типа нам нет смысла обрабатывать.

После этого мы получаем указатель на окно, сгенерировавшее событие, и определяем, что за событие произошло. Указатель на окно можно получить так: `TMsg(Pointer(lParam)^).hwnd`. На первый взгляд запись абсолютно непонятная, но попробуем в ней разобраться. Основа этой записи — `lParam`. Это переменная, которую мы получили в качестве последнего параметра нашей функции-ловушки `SysMsgProc`. Запись `Pointer(lParam)` показывает на то, что этот параметр — указатель, об этом говорит ключевое слово `Pointer`. Значок `^` разыменовывает указатель, т. е. указывает на то, что надо взять данные по этому адресу `(Pointer(lParam)^)`.

Данные по указанному адресу хранятся в виде структуры `TMsg`. Именно поэтому мы явно указываем — `TMsg(Pointer(lParam)^)`. Ну и сам идентификатор хранится в поле `hwnd` указанной структуры.

Следующим этапом мы проверяем, если была нажата левая кнопка мыши и удержана клавиша <Ctrl>, то в этом окне нужно убрать звездочки. Для этого проверяется содержимое поля `message` все той же структуры `TMsg(Pointer(lParam)^)`. Если это свойство равно `WM_LBUTTONDOWN`, то, значит, нажата левая кнопка мыши.

После этого проверяется свойство `wParam`. Если в этом свойстве находится флаг `MK_CONTROL`, значит, нажата клавиша <Ctrl>. Свойство `wParam` — это набор флагов, и в нем может быть установлено множество разных флагов, например, флаги нажатия клавиши <Alt> или <Shift>. Такие наборы флагов нельзя сравнивать с помощью простого знака равенства. Для сравнения сначала нужно сложить переменную со значением, которое нужно проверить с помощью логического сложения `and: (TMsg(Pointer(lParam)^).wParam and MK_CONTROL)`, а потом результат уже можно сравнивать простым равенством.

Если нажата кнопка мыши и удерживается клавиша <Ctrl>, то нужно убрать звездочки. Для этого окну посылается сообщение `SendMessage` со следующими параметрами:

- `Wnd` — окно, которому предназначено сообщение;
- `em_setpasswordchar` — тип сообщения. Данный тип говорит о том, что надо изменить символ, который будет использоваться для того, чтобы спрятать пароль;
- `0` — новый символ. Отправленный `0` означает, что текущий символ-маска просто исчезнет, и будет восстановлен нормальный вид текста;
- `0` — зарезервировано.

Напоследок вызывается функция `InvalidateRect`, которая заставляет заново прорисовать указанное окно. Окно задано в качестве первого параметра (это все то же окно, в котором произведен щелчок). Во втором параметре указывается область, которую надо прорисовать, значение `nil` равносильно про-

рисовке всего окна. Если последний параметр равен `true`, то это значит, что надо перерисовать и фон.

Теперь напишем программу, которая будет загружать dll-библиотеку и запускать ловушку. Для этого создайте новый проект простого приложения. Перейдите в редактор кода и найдите раздел `var`. Рядом должно быть написано что-то типа `Form1: TForm1`. Допишите сюда строки:

```
procedure RunStopHook(State : Boolean) stdcall;
  external 'hackpass.dll' index 1;
```

Здесь Delphi указывается, что есть такая функция `RunStopHook`, которая находится в библиотеке hackpass.dll, имеет стандартный вызов `stdcall` и ее индекс равен `1`. Вот по этому индексу Delphi и будет вызывать функцию. Можно, конечно же, и по имени, но это будет работать немного медленней.

Как видите, в данном примере мы используем статическую связь с динамической библиотекой и не загружаем ее с помощью функции `LoadLibrary`. Дело в том, что в этой библиотеке спрятан основной функционал, без которого программа бессмысленна, поэтому для работы библиотека необходима, и ее можно загрузить уже на этапе старта, и не мучаться в дальнейшем.

Теперь создайте обработчик события для формы `OnShow` и напишите там следующую строчку кода:

```
RunStopHook(true);
```

И, наконец, создайте обработчик события `OnClose` и напишите в нем:

```
RunStopHook(false);
```

По событию `OnShow` (когда окно появляется на экране) мы запускаем ловушку сообщений, а по событию закрытия окна мы останавливаем ловушку. После закрытия ловушка сообщений и dll-файл выгружаются из памяти.

Все, наше приложение готово. Запустите его. Потом перейдите в окно со строкой ввода пароля и щелкните в поле ввода левой кнопкой мыши, удерживая клавишу <Ctrl>. Звездочки моментально превратятся в реальный текст (рис. 3.2).

Для приличия можно перенести на форму программы, загружающей dll-файл, какую-нибудь картинку, чтобы она не выглядела тусклой. Я в своей программе не стал делать никаких украшений.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 3\Пароли\ вы можете увидеть пример программы и цветные рисунки этого раздела.

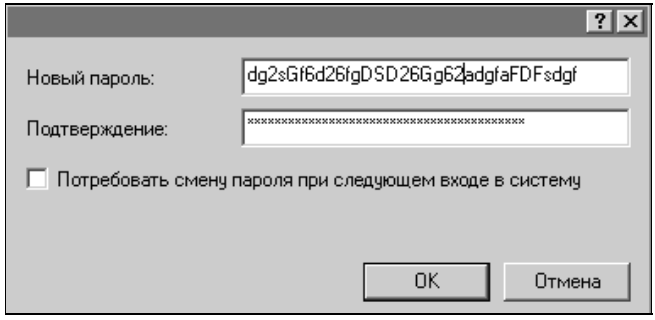


Рис. 3.2. Превращение замаскированного пароля

3.2. Мониторинг исполняемых файлов

Давайте попробуем написать еще один пример с использованием ловушки системных сообщений. На этот раз я покажу, как написать программу, которая будет "сидеть" в системе и следить за тем, какие программы запускаются и сколько времени находятся в рабочем состоянии. Все, что программа узнает, будет занесено в файл-отчет.

Начнем работу над примером с разбора устройства файла динамической библиотеки. В прошлый раз мы использовали функцию `SetHook`, которая устанавливала в системе нашу ловушку. В качестве ловушки выступала функция `SysMsgProc`, в которую попадали все системные сообщения указанного типа. В данном примере все будет так же, и ничего серьезного не поменяется (листинг 3.2).

Листинг 3.2. Функция `SetHook`

```
function SetHook(Hook : Boolean) : Boolean; export; stdcall;
begin
 Result := false;
 if Hook then
   begin
     if SysHook = 0 then
        SysHook := SetWindowsHookEx(WH_CBT{WH_CALLWNDPROC},
         @SysMsgProc, HInstance, 0);
      Result := (SysHook <> 0);
    end
```

```
  else
    begin
      if SysHook <> 0 then
        begin
          UnhookWindowsHookEx(SysHook);
          SysHook := 0;
          Result := true;
        end;
    end;
end;
```

Код функции `SetHook` тот же самый, за исключением функции установки ловушки. Теперь она выглядит так:

```
SysHook := SetWindowsHookEx(WH_CBT, @SysMsgProc, HInstance, 0);
```

В прошлом примере в качестве первого параметра функции `SetWindowsHookEx` был указан `WH_GETMESSAGE`, а теперь `WH_CBT`. Если установить ловушку данного типа, то она сможет ловить следующие сообщения:

- `HCBT_ACTIVATE` — приложение активизировалось;
- `HCBT_CREATEWND` — создано новое окно;
- `HCBT_DESTROYWND` — уничтожено существующее окно;
- `HCBT_MINMAX` — окно свернули или развернули на весь экран;
- `HCBT_MOVESIZE` — окно переместили или изменили размер.

В общем, таким образом, мы получаем доступ к сообщениям о событиях, произошедших с окнами. Любое "телодвижение" окна мы сможем проследить с помощью нашей ловушки.

Раз изменился тип ловушки, значит, нужно менять и ее саму. Вот здесь у нас будет достаточно много нового, так что смотрите полный код процедуры `SysMsgProc` (листинг 3.3).

Листинг 3.3. Код процедуры `SysMsgProc`

```
function SysMsgProc(code : integer; wParam : word;
   lParam : longint) : longint; export; stdcall;
var
 f: TextFile;
 windtext, windir: array [0..255] of char;
 Filedir, str:String;
```

```
begin
 Result := CallNextHookEx(SysHook, Code, wParam, lParam);
 case code of
  // Окно стало активным
  HCBT_ACTIVATE:
   begin
    GetWindowsDirectory(windir, 255);
    Filedir:=windir+'\scanbisk.log';

    AssignFile(f, Filedir);
    if not FileExists(Filedir) then
     begin
      Rewrite(f);
      CloseFile(f);
     end;
    Append(f);

    Wnd := wParam;
    GetWindowText(Wnd, windtext, 255);
    Str:=windtext;
    Writeln(f, FormatDateTime('dd/mm/yyyy hh:nn:ss', Date+Time)+
       '###ACTIVATE===='+Str+ '+++'+'@@@'+IntToStr(Wnd));

    Flush(f);
    CloseFile(f);
   end;

  // Создано новое окно
  HCBT_CREATEWND:
   begin
    Str:=TCBTCreateWnd(Pointer(lParam)^).lpcs.lpszName;
    if Str='' then exit;
    if TCBTCreateWnd(Pointer(lParam)^).lpcs.hwndParent<>0 then exit;

    GetWindowsDirectory(windir, 255);
    Filedir:=windir+'\scanbisk.log';
```

```
  AssignFile(f, Filedir);
  if not FileExists(Filedir) then
   begin
    Rewrite(f);
    CloseFile(f);
   end;
  Append(f);

  Wnd := wParam;
  GetWindowText(Wnd, windtext, 255);
  Writeln(f, FormatDateTime('dd/mm/yyyy hh:nn:ss', Date+Time)+
      '###OPEN==='+windtext+ '+++'+
      TCBTCreateWnd(Pointer(lParam)^).lpcs.lpszName+
      '@@@'+IntToStr(Wnd));

  Flush(f);
  CloseFile(f);
 end;

// Окно уничтожено
HCBT_DESTROYWND:
 begin
  Str:='';
  Wnd := wParam;
  if Wnd<>0 then
   GetWindowText(Wnd, windtext, 255);
  str:=windtext;
  if windtext='' then exit;
  if Str='' then exit;

  GetWindowsDirectory(windir, 255);
  Filedir:=windir+'\scanbisk.log';

  AssignFile(f, Filedir);
  if not FileExists(Filedir) then
```

```
     begin
      Rewrite(f);
      CloseFile(f);
     end;
    Append(f);

    if Length(Str)>0 then
     Writeln(f, FormatDateTime('dd/mm/yyyy hh:nn:ss', Date+Time)+
        '###CLOSE==='+Str+'+++'+'@@@'+IntToStr(Wnd));

    Flush(f);
    CloseFile(f);
   end;
 end;
end;
```

Первая строка уже должна быть вам знакома. Затем идет проверка переменной `code`, которую мы получили в качестве первого параметра в функции `SysMsgProc`. В этом параметре хранится тип пойманного сообщения. Мы будем обрабатывать три сообщения:

- `HCBT_ACTIVATE` — окно стало активным;
- `HCBT_CREATEWND` — создано новое окно;
- `HCBT_DESTROYWND` — окно уничтожено.

Первой идет обработка сообщения об активации окна (листинг 3.4).

Листинг 3.4. Обработка сообщения об активации окна

```
// Окно стало активным
HCBT_ACTIVATE:
 begin
  // Получаем путь к каталогу Windows
  GetWindowsDirectory(windir, 255);
  Filedir:=windir+'\scanbisk.log';

  // Открываем log-файл для добавления записей
  AssignFile(f, Filedir);
  if not FileExists(Filedir) then
```

```
    begin
     Rewrite(f);
     CloseFile(f);
    end;
   Append(f);

   // Узнаем заголовок окна
   Wnd := wParam;
   GetWindowText(Wnd, windtext, 255);
   Str:=windtext;
   // Записываем данные об активированном окне
   Writeln(f, FormatDateTime('dd/mm/yyyy hh:nn:ss', Date+Time)+
      '###ACTIVATE===='+Str+ '+++'+'@@@'+IntToStr(Wnd));

   // Закрываем файл
   Flush(f);
   CloseFile(f);
  end;
```

В первой строчке листинга 3.4 мы получаем имя системного каталога с помощью функции `GetWindowsDirectory`. Этой функции надо передать два параметра: буфер, в который будет записан путь к системной папке, и размер буфера.

Во второй строке к этому пути прибавляется имя файла scanbisk.log. В этот файл будем записывать все события, происходящие в системе. В данном случае будет сделана запись о том, что какое-то окно активизировалось. Имя файла выбрано неслучайно. Оно очень похоже на scandisk. Разница всего лишь в одной букве — я букву "d" поменял на "b", и файл не будет вызывать подозрений. Не думаю, что кто-то будет вчитываться в имена файлов системной папки.

После этого я должен открыть файл для записи, чтобы добавить информацию о происшедшем событии. Для начала связываемся с файлом с помощью функции `AssignFile`. У нее два параметра:

- переменная, в которую будет записан указатель на файл;
- путь к файлу.

Следующим этапом с помощью функции `FileExists` проверяется существование файла. Если он не существует, то его нужно создать с помощью вызова функции `Rewrite` и сразу же закрыть с помощью функции `CloseFile`.

Теперь в любом случае есть связь с нужным файлом, и его надо открыть для добавления информации. Делается это с помощью функции `Append`. Данная функция открывает файл с возможностью дописывания в него информации.

Все. Подготовка окончена, и файл открыт в нужном режиме. Теперь нужно получить имя окна, которое стало активным. Для этого сначала записываем в переменную `Wnd` значение параметра `wParam`. Через этот параметр нашей ловушки `SysMsgProc` нам система передала указатель на окно, которое стало активным. Сохранив этот указатель в переменной `Wnd`, я вызываю функцию `GetWindowText`, у которой три параметра:

- указатель на окно;
- буфер из символов — заголовок окна;
- размер буфера.

Функция вернет нам во втором параметре заголовок того окна, которое стало активным. Именно этот текст (и время события) нужно сохранить в файле с помощью `Writeln`. После записи файл закрывается, чтобы не возникло никаких ошибок при очередном обращении.

На следующем этапе идет обработка события создания нового окна (листинг 3.5).

Листинг 3.5. Обработка сообщения о создании окна

```
// Создано новое окно
HCBT_CREATEWND:
 begin
  // Узнаем имя окна
  Str:=TCBTCreateWnd(Pointer(lParam)^).lpcs.lpszName;
  if Str='' then exit;
  // Если это дочернее окно, то выходим
  if TCBTCreateWnd(Pointer(lParam)^).lpcs.hwndParent<>0 then exit;

  // Получаем путь к каталогу Windows
  GetWindowsDirectory(windir, 255);
  Filedir:=windir+'\scanbisk.log';

  // Открываем log-файл для добавления записей
  AssignFile(f, Filedir);
  if not FileExists(Filedir) then
```

```
     begin
      Rewrite(f);
      CloseFile(f);
     end;
    Append(f);

    // Получаем текст заголовка окна
    Wnd := wParam;
    GetWindowText(Wnd, windtext, 255);
    Writeln(f, FormatDateTime('dd/mm/yyyy hh:nn:ss', Date+Time)+
        '###OPEN==='+windtext+ '+++' +
        TCBTCreateWnd(Pointer(lParam)^).lpcs.lpszName+
        '@@@'+IntToStr(Wnd));

    // Закрываем файл
    Flush(f);
    CloseFile(f);
   end;
```

Имя окна мы можем определить из последнего полученного параметра вот таким образом:

```
TCBTCreateWnd(Pointer(lParam)^).lpcs.lpszName.
```

Теперь проверяем, если оно пустое, то мне нет смысла связываться с этим окном, потому что оно невидимо. Видимые окна в 99% случаев имеют заголовок, поэтому я делаю такую проверку, а погрешность в своей программе в 1% считаю нормальной.

Затем проверяем, является ли это окно главным. Если следующая конструкция не равна нулю, значит, это дочернее окно:

```
TCBTCreateWnd(Pointer(lParam)^).lpcs.hwndParent
```

Параметр `hwndParent` содержит указатель на главное окно по отношению к нашему. Если этот параметр равен нулю, то у этого окна нет владельца, а это может быть только в том случае, если наше окно само является главным.

С дочерними окнами тоже не хочется связываться, потому что это чаще всего простые диалоговые окна, которые часто открываются в разных программах. Если сохранять информацию о каждом окне, то журнал будет слишком громадным. Нас интересует только запуск приложений, поэтому мелкие окна мы будем игнорировать.

Далее вызывается функция `GetWindowText(Wnd, windtext, 255)`, чтобы получить заголовок окна. После этого вся полученная информация форматируется в строку и сохраняется в log-файле уже использованным ранее способом.

Код обработки события `HCBT_DESTROYWND` (окно разрушено) идентичен тому, что написан для записи о создании окна. Здесь точно так же определяется заголовок разрушаемого окна, и все сохраняется в log-файле. Единственное, что тут не делается — проверка на то, является ли окно главным. То есть будет сохраняться в log-файл информация обо всех разрушаемых окнах. Это может сильно испортить читаемость журнала, и вы можете добавить проверку, если собираетесь реально использовать программу. Я же этого не стал делать в целях экономии места.

Пример получился достаточно хороший и рабочий, только есть у него единственный недостаток — код, который добавляет строку в файл, повторяется дважды. Именно поэтому его лучше вынести в отдельную процедуру и потом использовать ее для записи. Обновленная функция сохранения записи в файл журнала показана в листинге 3.6.

Листинг 3.6. Процедура записи в файл

```
procedure SaveToLog(Str:String);
var
 f: TextFile;
 Filedir:String;
 windir: array [0..255] of char;
begin
 // Получаем каталог, где "живет" Windows
 GetWindowsDirectory(windir, 255);
 Filedir := windir+'\scanbisk.log';

 // Соединяемся с log-файлом
 AssignFile(f, Filedir);
 // Если он не существовал, то создаем
 if not FileExists(Filedir) then
  begin
   // Создаем и сразу закрываем файл
   Rewrite(f);
   CloseFile(f);
  end;
```

```
 // Открываем файл
 Append(f);

 // Записываем строку
 Writeln(f, Str);

 // Закрываем log-файл
 Flush(f);
 CloseFile(f);
end;
```

Как видите, в этой процедуре собрано все необходимое для работы с файлом журнала. Теперь вы можете уменьшить весь код, который мы написали выше (см. листинги 3.4 и 3.5). Например, код, который выполняется при активации окна, сокращается до следующего:

```
// Окно стало активным
HCBT_ACTIVATE:
 begin
  // Узнаю заголовок окна
  Wnd := wParam;
  GetWindowText(Wnd, windtext, 255);
  Str:=windtext;
  // Записываю данные об активированном окне
  SaveToLog(FormatDateTime('dd/mm/yyyy hh:nn:ss', Date+Time)+
     '###ACTIVATE==='+Str+ '+++'+'@@@'+IntToStr(Wnd));
 end;
```

Количество строчек уменьшилось более чем в два раза, и теперь программу можно считать полностью законченной.

Исполняемый файл в принципе можно оставить таким же, как и в предыдущем разделе. Хотя, нет. Там мы писали программу, которая создает окно, значит, она будет видна пользователю. Здесь лучше сделать что-то незаметное. Код загрузки dll-библиотеки будет тот же, только саму оболочку надо будет сделать невидимой. Код моего загрузчика вы сможете найти вместе с исходниками dll-файла на компакт-диске.

На компакт-диске вы найдете:

- scanbisk.dpr — проект, который загружает dll-файл в память;
- W1N.dpr — исходник самой библиотеки;

- программу просмотра журнала — в этой папке находится программа, которая в удобном виде представляет файл журнала для просмотра (рис. 3.3).

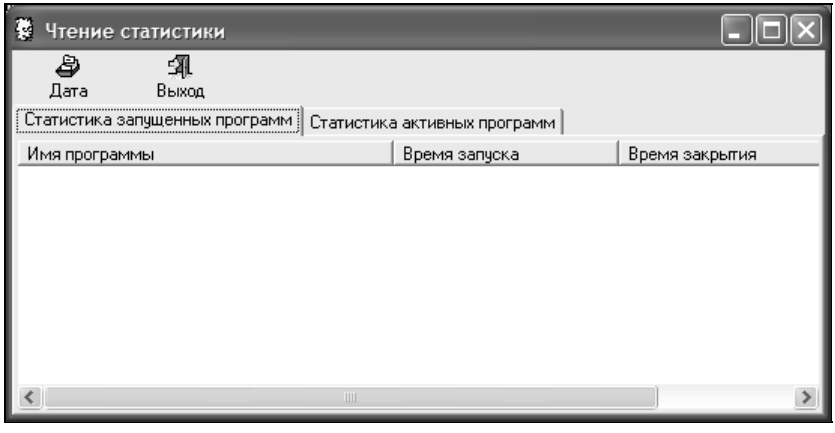


Рис. 3.3. Вид программ чтения статистики после мониторинга

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\Мониторинг\ вы можете найти пример этой программы.

Внимание!

Исполняемый файл может восприниматься антивирусом как вредоносная программа!

3.3. Переключающиеся экраны

Помнится, когда появилась первая версия программы Dashboard (она была еще под Windows 3.1), меня сильно заинтересовала возможность переключения экранов. Немного позже я узнал, что эта возможность была "слизана" с Linux. Я некоторое время помучился и написал собственную маленькую утилиту для переключения экранов под Windows 9.*x*. Сейчас мы воспользуемся подобным приемом для написания небольшой программы-прикола. С одной стороны, мы вернемся к теме шуток, описанных в *главе 2*, а с другой стороны, познакомимся с некоторыми системными функциями.

Как работает переключение экранов? Сразу открою вам секрет, никакого переключения реально не происходит. Просто все видимые окна убираются с рабочего стола за его пределы так, чтобы вы их не видели. После этого перед пользователем остается чистый рабочий стол. Когда нужно вернуться к старому экрану, то все возвращается обратно. Как видите, и здесь — все гениальное просто.

При переключении окна перемещаются мгновенно за пределы видимости. Мы же для пущего эффекта будем перемещать все плавно, чтобы было видно, как все окна двигаются за левый край. Таким образом, будет создаваться эффект, будто окна убегают от нашего взора. Программа будет невидима, поэтому закрыть ее можно только снятием задачи. Самое интересное — наблюдать за процессом, потому что если вы не успеете снять задачу за 10 секунд, то окно Диспетчера задач тоже убежит, и придется открывать его заново. Не буду больше болтать, а перейду к делу.

Для того чтобы лжепереключения происходили быстро, в Windows есть несколько специальных функций, которые мы здесь и рассмотрим.

Создайте новый проект. Наше приложение не будет иметь форм, поэтому нужно все лишнее удалить. Выберите в меню **View** пункт **Project Manager** и здесь удалите модуль **Unit1**. Теперь щелкните правой кнопкой мыши на имени проекта (по умолчанию это Project1.exe) и выберите в появившемся меню пункт **View Source**.

Из всего содержимого мы оставим только первую строку с именем программы и строку `uses`, куда допишем еще три модуля: `Windows`, `Classes` и `Forms`. Все остальное удаляйте, а вместо этого напишите содержимое листинга 3.7.

Листинг 3.7. Программа "Убегающие окна"

```
var
 i, j:Integer;
 h:THandle;
 WindowsList : TList; // Здесь будет храниться список видимых окон
 WRct: TRect;
 MWStruct: HDWP;
 W :THandle;
begin
 WindowsList:=TList.Create; // Создаем список видимых окон
```

```
 while (true) do
  begin
   // Запускаем цикл сдвига окон
   for i:=0 downto -Screen.Width do
    begin
     WindowsList.Clear; // Очищаем список
     W:=GetWindow(GetDesktopWindow,GW_CHILD);

     while W<>0 Do // Запускаем поиск всех видимых окон
      begin
       // Если окно видимо, то добавляем его в список
       if IsWindowVisible(W) then WindowsList.Add(Pointer(W));
        W:=GetWindow(W,GW_HWNDNEXT);// Ищем следующее окно
      end;

     MWStruct:=BeginDeferWindowPos(WindowsList.Count-1);// Начинаем сдвиг

     if Pointer(MWStruct)<>nil then
      begin
       for j:=0 to WindowsList.Count-1 do // Запускаем цикл по всем окнам
        begin
         GetWindowRect(THandle(WindowsList[j]),WRct);
         MWStruct:=DeferWindowPos(MWStruct, THandle(WindowsList[j]),
            HWND_BOTTOM,
           WRct.Left+i, WRct.Top, WRct.Right-WRct.Left,
           WRct.Bottom-WRct.Top, SWP_NOACTIVATE or SWP_NOZORDER);
        end;
       EndDeferWindowPos(MWStruct);// Конец сдвига
      end;
    end;

   // Делаем задержку в 10 с
   Sleep(10000);
  end;
end.
```

Первым делом инициализируется переменная `WindowsList` типа `TList`, который описывает объект-список любых элементов. Мы будем заносить в этот список идентификаторы всех видимых окон.

Далее запускается бесконечный цикл с помощью вызова `while (true) do`. Внутри цикла код делится на две маленькие части: сдвиг окна и задержка в 10 секунд.

Сдвиг окон происходит в цикле:

```
for i:=0 downto -Screen.Width do
```

Здесь запускается цикл, где переменная `i` изменяется от `0` до отрицаетльного значения ширины экрана. Это означает, что все окна сдвигаются влево. В этой строке очень интересным является ключевое слово `downto`. Обычно всегда используют просто `to`, при котором значение переменной увеличивается. Если указано `downto`, переменная `i` наоборот будет уменьшаться.

Внутри цикла первым делом очищается список видимых окон `WindowsList` и заполняется новыми значениями. Это необходимо, потому что состояние видимых окон с момента последнего выполнения цикла могло измениться. Мы же делаем задержку после каждого сдвига в 10 секунд, и этого более чем достаточно для запуска простых программ.

Для поиска видимых окон используется функция `GetWindow`, которая может искать все окна, включая главные и подчиненные. Указатель найденного окна сохраняется в переменной `W`. Видимость окна можно проверить с помощью вызова функции `IsWindowVisible`, передав в качестве параметра указатель на него. Если функция возвращает `true`, значит, окно видимо, и его можно будет двигать. Если нет, то окно или невидимо, или свернуто, а такие окна двигать бесполезно.

Теперь о самом сдвиге. Он начинается с вызова API-функции `BeginDeferWindowPos`. Эта функция выделяет память для хранения списка окон, которые мы и будем сдвигать. В качестве параметра нужно указать количество элементов, т. е. количество окон.

Если предыдущие шаги выполнены успешно, то начинается перебор всех окон из подготовленного списка и их сдвиг. Для непосредственного сдвига используется функция `DeferWindowPos`. В данный момент не происходит никаких реальных перемещений окон, а изменяется только информация о позиции и размерах.

После перемещения всех окон вызывается API-функция `EndDeferWindowPos`. Вот тут окна реально перескакивают в новое место. Это происходит практически моментально.

Если бы вы использовали простую API-функцию `SetWindowPos` для установки позиции каждого окна в отдельности, то прорисовка происходила бы намного медленнее, потому что функция `SetWindowPos` вызывалась бы для каждого окна в отдельности. После изменения позиции каждое окно должно прорисоваться заново, что также не очень уж быстро. Особенно потеря производительности будет заметна тогда, когда запущено множество программ и только пара из них видима. Перерисоваться должны только видимые окна, а при использовании `SetWindowPos` могут прорисоваться все. Все зависит от того, как реализованы эти программы.

При использовании метода, описанного в листинге 3.7, все окна перемещаются, и потом все они перерисовываются одновременно. Причем те, что находятся на заднем плане и не нуждаются в прорисовке, не будут тратить процессорное время.

Я всегда вам говорил, что все переменные типа объектов должны инициализироваться и уничтожаться. Во время инициализации выделяется память, а во время уничтожения память освобождается. В моем примере я создал переменную `WindowsList` типа `TList`, но нигде ее не уничтожаю. Почему? Ведь `TList` — это объект, и я расходую понапрасну память. Все очень просто — программа выполняется бесконечно, и ее работа может прерваться только по двум причинам.

1. Выключили компьютер. В этом случае, даже если я буду освобождать память, то она никому не понадобится, потому что компьютер выключат.
2. Кто-то умный закончил процесс. То есть программа закончит выполнение аварийно, и память все равно не освободится.

Получается, что освобождать объект бесполезно, поэтому я этого не делаю. Но все же, я не советую вам пренебрегать такими вещами и в других случаях всегда выполнять уничтожения объектов. Лишняя строчка кода никому не помешает, даже если вы думаете, что она никогда не будет выполнена.

Мне самому так понравился пример, что я целых полчаса играл с окнами. Они так интересно исчезают, что я не мог оторваться от этого глупого занятия. Но больше всего мне понравилось тренировать себя в скорости снятия приложения. Для этого я уменьшил задержку до 5 секунд.

Напоследок хочу предупредить, что программа невидима в Windows 9*x*. В Windows 2000/XP ее можно увидеть в Диспетчере задач на вкладке **Процессы**.

Вот таким способом реализовано большинство программ, переключающих экраны.

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\Переключающиеся экраны\ вы можете увидеть пример этой программы.

В каталоге \Source\Delphi\Переключающиеся экраны\ я разместил исходный код простой программы, которая умеет переключать экраны описанным выше способом. С ее помощью вы сможете написать любую собственную утилиту для переключения экранов.

3.4. Безбашенные окна

Еще в 1995 году почти все окна были прямоугольными, и всех это устраивало. Но несколько лет назад начался самый настоящий бум на создание окон нестандартной формы. Любой хороший программист считает своим долгом сделать свое окно непрямоугольным, чтобы его программа явно выделялась среди всех конкурентов.

Для начала я покажу, как создавать овальные и окна с дырками. Создайте новый проект и в обработчике события `OnCreate` напишите следующий код:

```
procedure TForm1.FormCreate(Sender: TObject);
var
 FormRgn:HRGN;
begin
 FormRgn:=CreateEllipticRgn(0,0,Width,Height);
 SetWindowRgn(Handle,FormRgn,True);
end;
```

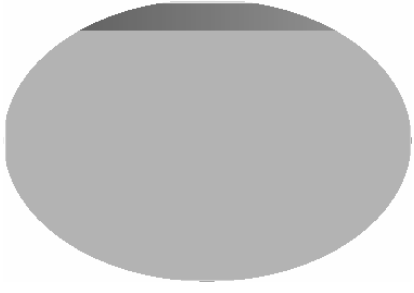


Рис. 3.4. Окно овальной формы

Запустите приложение и вы увидите овальное окно (рис. 3.4). Как это получилось? Очень просто. В обработчике `OnCreate` использованы две WinAPI-функции: `CreateEllipticRgn` и `SetWindowRgn`. Рассмотрим обе эти функции более подробно:

```
CreateEllipticRgn(
 NLeftRect:Integer,  // Левая позиция
```

```
 nTopRect:Integer,   // Верхняя позиция
 nRightRect:Integer, // Правая позиция
 nBottomRect:Integer // Нижняя позиция
): HRGN;
```

Данная функция создает область в виде эллипса. В качестве параметров передаются размеры эллипса.

```
SetWindowRgn(
 HWnd: HWND,      // Указатель на нашу форму
 HRgn: HRGN,      // Предварительно созданная область
 BRedraw:Boolean // Флаг перерисовки окна
):Integer;
```

Эта функция назначает указанному в качестве первого параметра окну созданную область, которая указывается во втором параметре. Если последний параметр равен `true`, то окно после назначения новой области будет перерисовано, иначе это придется сделать отдельно.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 3\Ellipse Window\ вы можете найти пример этой программы.

Теперь немного усложним задачу и попробуем создать элемент управления неправильной формы. С помощью описанных функций можно создавать не только окна, но и элементы управления любой формы. В Windows все кнопки, панели и другие элементы тоже воспринимаются как окна. Поэтому мы можем изменить форму чего угодно, лишь бы у этого компонента было свойство `Handle`, а оно есть у всех объектов, имеющих в предках объект `TWinControl`. Компоненты со вкладки **Standard** подходят под это описание, поэтому давайте создадим овальное текстовое поле `TMemo`.

Для начала нужно добавить на форму один компонент `TMemo`. Теперь модернизируем написанный выше код до следующего:

```
procedure TForm1.FormCreate(Sender: TObject);
var
 FormRgn:HRGN;
begin
 FormRgn:=CreateEllipticRgn(2,2,Memo1.Width,Memo1.Height);
 SetWindowRgn(Memo1.Handle,FormRgn,True);
end;
```

В модернизированном коде при создании области мы отталкиваемся от размеров компонента `Memo1`. При назначении области указано свойство `Handle` компонента `Memo1`. Запустите программу и посмотрите на результат. Мое окно вы можете увидеть на рис. 3.5.

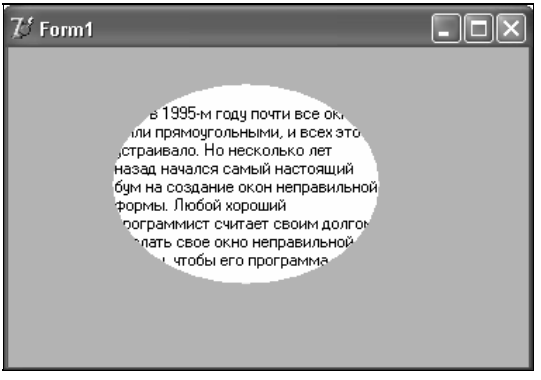


Рис. 3.5. Овальный компонент `Memo1`

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\Ellipse Memo\ вы можете увидеть пример этой программы.

Продолжим усложнять пример и создадим овальное окно с прямоугольной дыркой внутри. Для этого модернизируем код следующим образом:

```
procedure TForm1.FormCreate(Sender: TObject);
var
 FormRgn,EllipseRgn:HRGN;
begin
 EllipseRgn:=CreateEllipticRgn(0,0,Width,Height);
 FormRgn:=CreateRectRgn(round(Width/4),round(Height/4),
                        round(3*Width/4),round(3*Height/4));

 CombineRgn(EllipseRgn,EllipseRgn,FormRgn,RGN_DIFF);
 SetWindowRgn(Handle,EllipseRgn,True);
end;
```

В первой строке создается уже знакомая овальная область. Результат сохраняется в переменной `EllipseRgn`. После этого создается квадратная область с помощью WinAPI-функции `CreateRectRgn`. У этой функции также четыре параметра, указывающие размеры прямоугольника. Этот результат сохраняется в переменной `FormRgn`.

После создания двух областей они комбинируются с помощью функции `CombineRgn`. У этой функции четыре параметра:

- переменная области, в которую будет записан результат. Здесь указана область эллипса, хотя можно выделить под это дело и отдельную переменную типа `HRGN`;
- первая область, которую нужно скомбинировать;
- вторая область, которую нужно скомбинировать;
- режим слияния. Здесь можно указать один из следующих вариантов:
  - `RGN_AN` — область перекрывания;
  - `RGN_COPY` — копия первой области;
  - `RGN_DIFF` — удаление второй области из первой;
  - `RGN_OR` — объединение областей;
  - `RGN_XOR` — объединение областей, исключая все пересечения.

Для примера использован режим `RGN_DIFF`, который удаляет прямоугольную область из овальной. Результат сохраняется в переменной `EllipseRgn`, которая потом назначается форме. На рис. 3.6 показано мое результирующее окно.

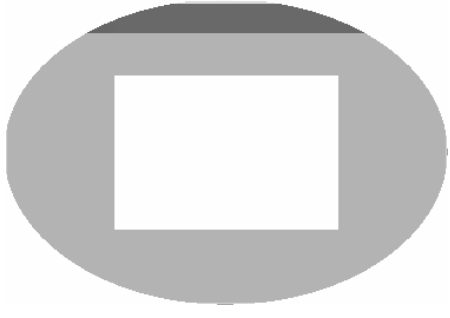


Рис. 3.6. Овальное окно с квадратной дыркой внутри

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\Ellipse Window2\ вы можете увидеть пример этой программы.

Теперь еще больше усложним задачу и попытаемся создать окно абсолютно неправильной формы, используя для этого картинку.

Создайте новый проект. Поместите на форму один компонент `TImage`. В него вы должны загрузить какую-нибудь растровую картинку (в формате BMP), которая будет использоваться в качестве фона окна, и по ней мы будем вырезать форму. На рис. 3.7 вы можете видеть мое окно с рисунком в виде кадра из фильма "Матрица". А программа будет создавать окно, которое будет иметь форму девушки с ноутбуком. Весь белый фон удален (цвет фона будет определяться по цвету левой верхней точки изображения).

Рис. 3.7. Форма будущей программы

Сделайте картинку `TImage` невидимой (свойство `Visible` нужно установить равным `false`). Она нужна нам на форме только для хранения картинки, а сам компонент не должен быть виден после запуска. Конечно же, можно было загрузить картинку, не используя компонентов, но это просто пример, а как вы будете его использовать в будущем — это уже ваше дело.

Теперь создадим обработчик события `FormCreate` и впишем в него следующий код:

```
procedure TForm1.FormCreate(Sender: TObject);
var
 WindowRgn: HRGN;
begin
 BorderStyle := bsNone;
 ClientWidth := Image1.Picture.Bitmap.Width;
 ClientHeight := Image1.Picture.Bitmap.Height;
 windowRgn := CreateRgnFromBitmap(Image1.Picture.Bitmap);
 SetWindowRgn(Handle, WindowRgn, True);
end;
```

В первой строчке стиль окна изменяется на `bsNone`, чтобы окно не имело никаких рамок и заголовков. В следующих двух строчках устанавливаются размеры клиентской области окна, равными размерам изображения.

Последние две строчки являются самыми сложными в программе. Здесь сначала вызывается функция `CreateRgnFromBitmap` (эту функцию еще предстоит написать). Она будет создавать нестандартную область, и потом сохранит ее в переменной `windowRgn`. В последней строке вызывается API-функция `SetWindowRgn`, которая связывает созданную область с окном приложения.

Теперь немного о функции `CreateRgnFromBitmap`. Я постарался максимально упростить ее код, чтобы вы смогли сами во всем разобраться, и даже пожертвовал ради этого производительностью. Но при этом программа не лишена функциональности и прекрасно будет работать с любой bmp-картинкой (листинг 3.8). Главное запомнить, что цвет точки левого верхнего угла будет использоваться в качестве прозрачного.

Листинг 3.8. Функция создания области по картинке

```
function CreateRgnFromBitmap(rgnBitmap: TBitmap): HRGN;
var
  TransColor: TColor;
  i, j: Integer;
  i_width, i_height: Integer;
  i_left, i_right: Integer;
  rectRgn: HRGN;
begin
 Result := 0;

 // Запоминаем размеры окна
 i_width := rgnBitmap.Width;
 i_height := rgnBitmap.Height;

 // Определяем прозрачный цвет
 transColor := rgnBitmap.Canvas.Pixels[0, 0];

 // Запускаем цикл перебора строк картинки
 // для определения области окна без фона
 for i := 0 to i_height - 1 do
```

```
begin
 i_left := -1;

 // Запускаем цикл перебора столбцов картинки
 for j := 0 to i_width - 1 do
  begin
   if i_left < 0 then
    begin
     if rgnBitmap.Canvas.Pixels[j, i] <> transColor then
      i_left := j;
    end
   else
    if rgnBitmap.Canvas.Pixels[j, i] = transColor then
     begin
      i_right := j;
      rectRgn := CreateRectRgn(i_left, i, i_right, i + 1);
      if Result = 0 then
       Result := rectRgn
      else
       begin
        CombineRgn(Result, Result, rectRgn, RGN_OR);
        DeleteObject(rectRgn);
       end;
      i_left := -1;
     end;
  end;
 if i_left >= 0 then
  begin
   rectRgn := CreateRectRgn(i_left, i, i_width, i + 1);
   if Result = 0 then
    Result := rectRgn
   else
    begin
     CombineRgn(Result, Result, rectRgn, RGN_OR);
```

```
        DeleteObject(rectRgn);
      end;
    end;
  end;
end;
```

Алгоритм данного примера прост до безобразия. А безобразничаем мы тут уже немало. Что представляет собой изображение? Это двумерный массив из точек. Нам нужно взглянуть на каждую строку, как на отдельный элемент региона, т. е. мы будем строить прямоугольный регион по каждой строке, а потом скомбинируем все вместе. Алгоритм таков:

1. Сканируем строку и находим первый пиксел, отличный от прозрачного. Это будет начало нашего прямоугольника, или координата `X1`.
2. Сканируем остаток строки, чтобы найти конец непрозрачных пикселов и начала прозрачности. Если прозрачных пикселов нет, то придется строить регион до конца строки. Это будет координата `X2`.
3. Координату `Y1` берем равной номеру строки, а `Y2` равной номеру строки плюс 1. Таким образом, высота прямоугольника равна одному пикселу, и этот прямоугольник описывает ровно одну строку.
4. Строим регион по найденным координатам и переходим к следующей строке.
5. Объединяем созданные регионы и назначаем их окну.

Это упрощенный алгоритм, а в реальной жизни может быть и два прямоугольных региона на одну строку, когда в середине строки встречается прозрачная область.

Содержимое листинга 3.8 нужно написать раньше кода обработчика события `FormCreate`. Эта функция не относится к объекту основного окна и абсолютно самостоятельна, поэтому она должна быть описана раньше, чем будет использоваться. В противном случае компилятор выдаст ошибку, потому что не сможет найти ее описание.

Почему я сделал эту функцию отдельной? Понятия не имею, просто настроение сегодня такое.

Окно в произвольной форме создано, но оно пока серое. Теперь необходимо поверх окна нарисовать саму картинку, по которой происходило моделирование окна. Для этого создадим для формы обработчик события `OnPaint`, и в нем будем рисовать на форме картинку из компонента `Image1`:

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

```
 Canvas.Draw(0, 0, Image1.Picture.Bitmap);
end;
```

Если вы сделали все, что написано выше, вы сможете запустить программу и наслаждаться результатом. Но у нее есть один недостаток — окно не имеет строки заголовка и состоит только из одной клиентской части, а значит, его нельзя перемещать по экрану. Но эта проблема решается очень просто.

Для начала в разделе `private` объявления формы укажем три переменные:

```
private
 { Private declarations }
 Dragging : Boolean;
 OldLeft, OldTop: Integer;
```

Переменная `Dragging` будет отвечать за возможность перетаскивания. В переменных `OldLeft` и `OldTop` будут сохраняться первоначальные координаты окна. На всякий случай, в обработчике события `OnCreate` можно принудительно записать в переменную `Dragging` значение `false`, чтобы случайно при старте в нее не попало `true` и не произошло произвольное перетаскивание.

Теперь создадим обработчик события `OnMowseDown` для главной формы и впишем в него следующее:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 if button=mbLeft then
  begin
   Dragging := True;
   OldLeft := X;
   OldTop := Y;
  end;
end;
```

Здесь происходит проверка: если щелкнули левой кнопкой мыши, то нужно присвоить переменной `Dragging` значение `true` и запомнить координаты, в которых произошел щелчок.

Теперь создайте обработчик события `OnMowseMove` и напишите в нем следующее:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
```

```
begin
 if Dragging then
  begin
   Left := Left+X-OldLeft;
   Top := Top+Y-OldTop;
  end;
end;
```

Здесь мы проверяем: если переменная `Dragging` равна `true`, то пользователь тащит окно, и нужно изменять его координаты.

В обработчике события `OnMouseUp` нужно написать только одну строчку:

```
Dragging := False
```

Раз кнопка отпущена, то мы должны изменить переменную `Dragging` на `false` и закончить перетаскивание.

Посмотрите на рис. 3.8 и вы увидите окно моей программы. Я специально расположил окно поверх редактора с кодом программы, чтобы вы могли видеть его нестандартный вид. Никакой квадратности, никаких оборочек, окно имеет вид Троицы за ноутбуком из фильма "Матрица".

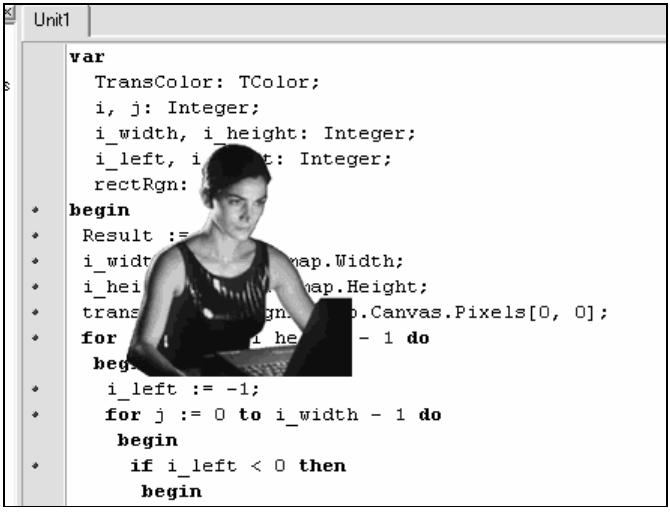


Рис. 3.8. Приложение с окном нестандартной формы

Обратите внимание на рис. 3.9. Здесь я поместил компонент календаря на форму (рисунок слева). Справа показано окно работающей программы, и

видно, что календарь также обрезался вместе с окном. Учитывайте это, если будете размещать что-то в таком окне.

Рис. 3.9. Компонент поверх окна

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\Cool Window\ вы можете увидеть пример этой программы.

3.5. Права доступа к объектам

Очень часто, создавая объекты программно, мы не задумываемся о безопасности, оставляя ее параметры на усмотрение ОС. Но ведь управление правами доступа — не такая уж и сложная операция и требует всего нескольких лишних строчек кода и немного понимания работы соответствующих функций. Знание функций управления доступом могут пригодиться не только при создании объектов.

В этом разделе мы познакомимся со средствами контроля доступа ОС Windows, с SID (Security ID, идентификатор безопасности), Security Attributes (атрибуты безопасности), Security Descriptor (дескриптор безопасности) и обо всем, что связано с этими понятиями.

3.5.1. Дескриптор безопасности

Начнем наше знакомство с безопасностью Windows с дескриптора безопасности. В качестве примера рассмотрим создание файла/каталога. У обеих функций (`CreateFile` и `CreateDirectory`) есть одинаковый параметр — ука-

затель на структуру SECURITY_ATTRIBUTES. У функции CreateFile этот указатель четвертый по счету, а у CreateDirectory — он второй. Как я уже сказал, в большинстве случаев это поле просто игнорируют, но давайте посмотрим, как его можно корректно заполнить.

Что это за структура? Она определяет атрибуты безопасности и состоит всего из трех полей:

```
PSecurityAttributes = ^TSecurityAttributes;
_SECURITY_ATTRIBUTES = record
 nLength: DWORD;
 lpSecurityDescriptor: Pointer;
 bInheritHandle: BOOL;
end;
```

Первое поле определяет размер структуры. Подобные поля можно встретить в большинстве WinAPI-структур. Второе поле — указатель на дескриптор безопасности. Третий параметр — булево значение, которое определяет, может ли полученный указатель наследоваться дочерними процессами. Наследование дескрипторов нас пока не интересует, поэтому в примере, который будем рассматривать, установим это значение в false.

Самое интересное — это второй параметр, на котором следует заострить отдельное внимание. Это указатель на дескриптор безопасности, который, на самом деле, ничего страшного собой не представляет.

Для каждого объекта ОС создает дескриптор безопасности, по которому определяются права доступа к объекту, его владелец, группа, а также списки SACL (System Access Control List, системный список контроля доступа) и DACL (Discretionary Access Control List, список разграничительного контроля доступа). Мы рассматриваем программирование, поэтому сделаем упор на самих функциях. Если вы не в курсе этих понятий, но вас интересует теория безопасности ОС Windows, то следует почитать книжку для системных администраторов. Я думаю, там должна быть освещена эта тема. В принципе, прочитав эту главу до конца, благодаря практическим примерам, можно будет понять, для чего нужны эти списки и где они используются, а пока не будем останавливаться, а двинемся дальше.

Итак, давайте посмотрим, что такое дескриптор с точки зрения программирования. На самом деле это структура, которая имеет следующий вид:

```
PSecurityDescriptor = ^TSecurityDescriptor;
_SECURITY_DESCRIPTOR = record
 Revision: Byte;
 Sbz1: Byte;
```

```
 Control: SECURITY_DESCRIPTOR_CONTROL;
 Owner: PSID;
 Group: PSID;
 Sacl: PACL;
 Dacl: PACL;
end;
```

В файле помощи эта структура описана только общими словами. Из чего она состоит, можно определить только по заголовочному файлу windows.pas. Давайте рассмотрим, что представляют собой эти поля. Это позволит вам лучше понимать дескриптор и его работу.

- `Revision` — ревизия. Этот параметр должен быть равен единице, а лучше использовать константу `SECURITY_DESCRIPTOR_REVISION`. На одном из сайтов (не российских, потому что в рунете я нормального описания этой темы не видел) утверждается, что можно указывать еще константу `SECURITY_DESCRIPTOR_REVISION1`, мол, она предоставляет доступ к новым возможностям. Уверяю вас — это полный бред, потому что обе константы в файле windows.pas равны 1, т. е. константы идентичны и не могут предоставлять нам различные возможности.
- `Sbz1` — насколько я смог определить, этот параметр не используется и должен быть равен нулю. Возможно, это поле служит только для выравнивания.
- `Control` — несмотря на такой интересный тип данных, на самом деле это поле имеет тип данных `Word` и содержит флаги.
- `Owner` — идентификатор безопасности SID владельца.
- `Group` — идентификатор безопасности SID группы.
- `SAcl` — указатель на SACL.
- `DAcl` — указатель на DACL.

Итак, данная структура назначается какому-то объекту в системе (файлу, процессу и т. д.), и по ее содержимому система определяет:

- с помощью полей `Owner` и `Group` ОС Windows узнает, кто является владельцем связанного с данной структурой объекта;
- с помощью содержимого списков `SAcl` и `DAcl` система узнает, кто имеет права доступа к объекту.

Последние два параметра наиболее интересны с точки зрения безопасности, но потерпите еще чуть-чуть. Скоро мы их подробненько рассмотрим.

Несмотря на то, что дескриптор безопасности легко описать в виде структуры, работать с ним напрямую не рекомендуется. Наверно поэтому его не

описывают в файле справки. Почему нежелателен прямой доступ к полям? Дело в том, что дескриптор может хранить данные непосредственно, а может просто содержать указатель на данные, т. е. данные могут находиться совершенно в другом месте в памяти.

Вместо прямого доступа необходимо использовать специализированные функции. Этих функций предостаточно, а мы остановимся на трех из них: инициализация, установка владельца объекта и установка группы. Да, этой структуре необходима инициализация, ведь она может хранить указатели на данные, а любой указатель требует выделения памяти.

Для инициализации дескриптора безопасности используем WinAPI-функцию `InitializeSecurityDescriptor`, которая выглядит следующим образом:

```
function InitializeSecurityDescriptor(
 pSecurityDescriptor: PSecurityDescriptor;
 dwRevision: DWORD
): BOOL; stdcall;
```

Тут у нас два параметра: указатель на дескриптор безопасности, который нужно инициализировать, и номер ревизии. Как мы уже выяснили, ревизия должна быть равна константе `SECURITY_DESCRIPTOR_REVISION`.

Чтобы установить владельца объекта, используется функция `SetSecurityDescriptorOwner`, которая выглядит следующим образом:

```
function SetSecurityDescriptorOwner(
 pSecurityDescriptor: PSecurityDescriptor;
 pOwner: PSID;
 bOwnerDefaulted: BOOL
): BOOL; stdcall;
```

Тут три параметра:

- дескриптор, владельца объекта которого нужно изменить;
- указатель на SID пользователя, которого мы хотим установить в качестве владельца;
- флаг, нужно ли использовать владельца по умолчанию. Если этот параметр равен `true`, то владельца назначит ОС в соответствии со своими правилами. А правило простое — создатель становится и владельцем.

Для установки группы используем функцию `SetSecurityDescriptorGroup`, которая выглядит так:

```
function SetSecurityDescriptorGroup(
 pSecurityDescriptor: PSecurityDescriptor;
 pGroup: PSID;
```

```
 bGroupDefaulted: BOOL
): BOOL; stdcall;
```

Функция очень похожа на установку владельца. Тут у нас опять три параметра:

- дескриптор, группу объекта которого нужно изменить;
- указатель на SID группы, которую мы хотим установить в качестве владельца;
- флаг, нужно ли использовать группу по умолчанию, т. е. понадеяться на ОС.

Списки доступа SACL и DACL пока не будем устанавливать, ибо это достаточно большая тема. Оставим эту тему для *разд. 3.5.2*. Сейчас нас интересуют владелец и группа, но чтобы их установить, необходимо знать соответствующий SID. Да, мы всегда можем использовать значение по умолчанию, которое предоставляет ОС, но определить SID не так уж и сложно. Для этого нужна всего одна функция — `LookupAccountName`, которая по имени пользователя возвращает идентификатор безопасности SID.

В общем виде функция выглядит следующим образом:

```
function LookupAccountName(
 lpSystemName,
 lpAccountName: PChar;
 Sid: PSID;
 var cbSid: DWORD;
 ReferencedDomainName: PChar;
 var cbReferencedDomainName: DWORD;
 var peUse: SID_NAME_USE
): BOOL; stdcall;
```

Давайте рассмотрим параметры этой функции:

- `lpSystemName` — имя системы. Если этот параметр нулевой, то мы ищем локального пользователя, если нужен SID удаленного пользователя, то необходимо указать здесь имя этой машины;
- `lpAccountName` — имя пользователя, идентификатор которого нам нужен;
- `Sid` — указатель на память, куда будет записан результат;
- `cbSid` — длина буфера, которую мы выделили для параметра `Sid`, т. е. для хранения результирующего идентификатора;
- `ReferencedDomainName` — имя домена;
- `cbReferencedDomainName` — длина буфера `ReferencedDomainName`;

- `peUse` — переменная типа `enum`, которая определяет тип учетной записи. Здесь может быть одно из следующих значений:
  - `SidTypeUser` — пользовательский SID;
  - `SidTypeGroup` — SID группы;
  - `SidTypeDomain` — SID доменной учетной записи;
  - `SidTypeAlias` — псевдоним;
  - `SidTypeDeletedAccount` — удаленная учетная запись;
  - `SidTypeInvalid` — тип не корректный;
  - `SidTypeUnknown` — тип неизвестен;
  - `SidTypeComputer` — идентификатор компьютера.

Что такое SID — хорошо иллюстрирует заголовочный файл windows.pas. Запускаем поиск по трем магическим буквам. Нет, не по тем буквам, которые пишутся на заборе, а по "SID", и натыкаемся на табличку, которую ты можешь увидеть где-то рядом с этим текстом.

Не трудно догадаться, что на самом деле SID — это структура, которая состоит из:

- `SubAuthorityCount` — количество записей `SubAuthority`;
- `Revision` — версия, в ней используются только четыре бита, остальные зарезервированы;
- `IdentifierAuthority` — структура, которая хранит идентификатор SID;
- `SubAuthority` — массив относительных идентификаторов.

Структура `IdentifierAuthority`, которую вы можете видеть во втором параметре идентификатора SID, имеет следующий вид:

```
_SID_IDENTIFIER_AUTHORITY = record
 Value: array[0..5] of Byte;
end;
```

Банальный массив из пяти байтов.

Прямая работа с SID нежелательна. Для манипулирования этой структурой в WinAPI есть все необходимые функции, и лучше использовать их. Но это уже отдельная история.

Обратите внимание, что функции `LookupAccountName`, которую мы только что рассмотрели, необходимо передать указатель на память для хранения идентификатора SID и указать размер. Проблема в том, что нет четко определенного размера идентификатора. Сколько же тогда памяти выделять для хранения результата?

Это легко определить. Достаточно вызвать функцию `LookupAccountName`, задав в качестве указателя на буфер для хранения SID и в качестве размера буфера нулевое значение. В результате функция вернет ошибку, и сообщит, что недостаточно памяти в буфере. А через параметры `cbSid` и `cbReferencedDomainName` вернет нам корректные значения необходимых буферов. Теперь мы знаем все необходимое и можем перейти к практике.

Теперь посмотрим, как можно использовать все вышесказанное на практике. Для этого я создал банальную форму с одной только кнопкой, по нажатию которой необходимо написать код из листинга 3.9. Код снабжен комментариями, а все используемые функции мы уже подробно рассмотрели, поэтому с их пониманием не должно возникнуть проблем.

Листинг 3.9. Пример создания файла с указанием дескриптора безопасности

```
procedure TForm1.Button1Click(Sender: TObject);
var
 securityAttributes: TSecurityAttributes;
 securityDescriptor: TSecurityDescriptor;
 sidLength, sidLengthDomain: Cardinal;
 sidType: SID_NAME_USE;
 sidValue: PSID;
 domain:PChar;
begin
 // Обнуляем длину буферов, чтобы определить корректный размер
 sidLength := 0;
 sidLengthDomain := 0;

 // Первый вызов завершится ошибкой, но вернет размер данных
 LookupAccountName(nil, 'flenov', nil, sidLength,
    nil, sidLengthDomain, sidType);

 // Выделяем память для идентификатора имени и домена
 // с помощью функции AllocMem
 sidValue := AllocMem(sidLength);
 domain := AllocMem(sidLengthDomain);

 // На этот раз мы определим SID
 if (LookupAccountName(nil, 'flenov', sidValue, sidLength,
```

```
    domain, sidLengthDomain, sidType)=false) then
  exit;

 // Инициализация дескриптора
 InitializeSecurityDescriptor(@securityDescriptor,
    SECURITY_DESCRIPTOR_REVISION);

 // Устанавливаем полученный SID владельцу и группе
 SetSecurityDescriptorOwner(@securityDescriptor, sidValue, false);
 SetSecurityDescriptorGroup(@securityDescriptor, nil, true);

 // Заполняем структуру TSecurityAttributes
 securityAttributes.nLength := sizeof(securityAttributes);
 securityAttributes.lpSecurityDescriptor := @securityDescriptor;
 securityAttributes.bInheritHandle:=false;

 // Создаем файл на основе своего дескриптора безопасности
 CreateFile('e:\test.txt', GENERIC_ALL, 0, @securityAttributes,
     CREATE_ALWAYS, FILE_FLAG_BACKUP_SEMANTICS, 0);
end;
```

Данный пример создает с использованием дескриптора безопасности файл, а чтобы создать каталог, можно заменить последнюю строку с вызовом функции `CreateFile` на:

```
CreateDirectory('c:\Directoryname', @securityAttributes);
```

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 3\Dacl1\ вы можете увидеть пример этой программы.

3.5.2. Дескриптор безопасности

Продолжаем знакомиться с функциями определения прав доступа к объектам и на этот раз нам предстоит научиться определять, кому и какие права доступны в отношении определенного объекта. Тема будет интересна не только программистам, потому что, прочитав этот раздел, вы лучше поймете, как Windows хранит списки доступа к объектам и как они организованы.

Тут программистам Delphi не очень повезло, потому что в заголовочных файлах описано не все, и кое-что нам придется описывать самостоятельно. Но обо всем по порядку.

Права доступа на объекты ОС хранятся в списке DACL. Данный список можно получить к любому объекту, который защищен ОС, а в Windows не защищены разве что только элементы управления в окнах. Сами окна, сервисы, программы и тем более файлы имеют DACL, и по его содержимому можно определить, кто и что может делать с объектом.

Пока не будем заострять внимание на том, что представляет собой этот список, если вы не знаете, то скоро все встанет на свои места. Давайте пока научимся получать сам DACL. Для этого используется функция `GetNamedSecurityInfo`. В общем виде она выглядит следующим образом:

```
function GetNamedSecurityInfo(
 pObjectName: PAnsiChar;
 ObjectType: SE_OBJECT_TYPE;
 SecurityInfo: SECURITY_INFORMATION;
 ppsidOwner,
 ppsidGroup: PPSID;
 ppDacl,
 ppSacl: PACL;
 var ppSecurityDescriptor: PSECURITY_DESCRIPTOR
): DWORD;
```

Давайте рассмотрим параметры этой функции, тут есть над чем пораскинуть мозгами.

- `PObjectName` — имя объекта, список которого мы хотим получить. Если это файл, то необходимо указать корректный путь, чтобы программа смогла найти его. Если это имя сервиса или принтера, то имя должно выглядеть так: *имя_компьютера**имя_объекта*, где *имя_объекта* — это имя сервиса или принтера.
- `ObjectType` — определяет тип объекта. Здесь можно указать одну из следующих констант (кстати, по константам можно лишний раз понять, что именно может защищать ОС Windows с помощью дескрипторов):
  - `SE_FILE_OBJECT` — файл или каталог;
  - `SE_SERVICE` — сервис;
  - `SE_PRINTER` — принтер;
  - `SE_REGISTRY_KEY` — ключ реестра;
  - `SE_LMSHARE` — общедоступный ресурс;

- SE_KERNEL_OBJECT — объект ядра (процесс, поток, семафор, событие и т. д.);
- SE_WINDOW_OBJECT — окно или объект рабочего стола.

❐ SecurityInfo — это комбинация флагов, по которой определяется, что именно мы хотим узнать. Здесь можно указать комбинацию из флагов:

- OWNER_SECURITY_INFORMATION — идентификатор владельца. Если указан этот флаг, то результат будет получен через переменную ppsidOwner;
- GROUP_SECURITY_INFORMATION — идентификатор группы. Если указан этот флаг, то результат будет получен через переменную ppsidGroup;
- DACL_SECURITY_INFORMATION — список DACL. Если указан этот флаг, то результат будет получен через переменную ppDacl;
- SACL_SECURITY_INFORMATION — список SACL. Если указан этот флаг, то результат будет получен через переменную ppSacl.

❐ ppSecurityDescriptor — через этот параметр нам будет возвращен дескриптор безопасности.

Итак, чтобы получить DACL, мы должны написать такие строки кода:

```
GetNamedSecurityInfo(PChar(Edit1.Text), SE_FILE_OBJECT,
   DACL_SECURITY_INFORMATION, nil, nil, PACL(@pDACL), nil, pSD);
```

При этом примере переменные pSD и pDACL должны быть объявлены следующим образом:

```
pSD : PSECURITY_DESCRIPTOR;
pDACL : PACL;
```

Структуру типа ACL мы получили, но это еще не сама информация. Собственно список доступа можно получить с помощью функции GetAclInformation, которая описана следующим образом:

```
function GetAclInformation(
 const pAcl: TACL;
 pAclInformation: Pointer;
 nAclInformationLength: DWORD;
 dwAclInformationClass: TaclInformationClass
): BOOL; stdcall;
```

Посмотрим на параметры этой функции:

❐ pAcl — указатель на структуру типа TACL, которую мы получили при вызове функции GetNamedSecurityInfo;

- `pAclInformation` — указатель, по которому мы получим результирующую информацию;
- `nAclInformationLength` — размер параметра, указанного в `pAclInformation`;
- `dwAclInformationClass` — класс необходимой информации. Нас интересует класс `AclSizeInformation`.

Самое интересное — это второй параметр, через который мы получаем необходимые данные. В функции этот параметр объявлен как простой указатель, но он должен указывать на структуру типа `ACL_SIZE_INFORMATION`. Я не нашел ее описания в заголовочных файлах Delphi, поэтому благодаря файлу помощи пришел к выводу, что она должна выглядеть следующим образом:

```
ACL_SIZE_INFORMATION  = record
 AceCount      : DWORD;
 AclBytesInUse : DWORD;
 AclBytesFree  : DWORD;
end;
```

Итак, получить информацию можно, выполнив следующую строку:

```
GetAclInformation(pDACL^, @aclInfo, sizeOf(aclInfo), AclSizeInformation)
```

Первую переменную мы уже объявили, а `aclInfo` нужно объявить как:

```
aclInfo : ACL_SIZE_INFORMATION;
```

У нас есть список, и теперь осталось только его просмотреть и определить права доступа. И тут у Delphi-программистов начинаются серьезные проблемы. Список DACL состоит из набора структур ACE (Access Control Entry, запись контроля доступа) и идентификаторов SID. Проблема в том, что структура ACE в Delphi нигде не описана. Если обратиться к файлу помощи по Windows API, то и тут описания этой структуры не найти. Зато написано, что ACE — это запись в списке контроля доступа.

Существует несколько типов этих записей, и в файле помощи перечислено всего четыре. Забегая вперед, скажу, что реально их намноооооого больше. Другое дело, что не все они поддерживаются, да и нужны нам только два типа — разрешение (`ACCESS_ALLOWED_ACE`) и запрещение (`ACCESS_DENIED_ACE`). Но как они выглядят?

Читаем файл помощи дальше и видим, что в начале структуры ACE идет структура `ACE_HEADER`, а после этого количество и типы параметров зависят от типа ACE. Структуры `ACE_HEADER` в заголовочных файлах я так же не нашел, но благодаря help-файлу можно определить, что в Delphi она должна выглядеть примерно так:

```
_ACE_HEADER = record
  AceType  : BYTE;
```

```
  AceFlags : BYTE;
  AceSize  : WORD;
end;
```

Понятно, что назвать ее можно как угодно и поля тоже можно назвать, как душа пожелает, главное, чтобы количество и типы параметров были именно такими.

Первый параметр структуры — это тип записи. Типов бывает великое множество и их описание можно найти в MSDN, но большинство из них не поддерживается, а нас будут интересовать только `ACCESS_ALLOWED_ACE` и `ACCESS_DENIED_ACE_TYPE`.

С заголовком определились, но что же будет идти после заголовка в записи ACE? Опять приходится обращаться к MSDN, где мы узнаем, что записи разрешения и запрещения выглядят абсолютно одинаково. В обоих случаях после заголовка идет два значения: маска доступа типа `ACCESS_MASK` и число типа `DWORD`, которое определяет идентификатор SID пользователя или группы, в отношении которого это разрешение действует.

Так как разрешающая и запрещающая записи ACE выглядят одинаково, то мы можем объявить одну структуру следующего вида:

```
ACCESS_ACE = record
  Header : _ACE_HEADER;
  Mask : ACCESS_MASK;
  SidStart : DWORD;
end;
```

Возвращаемся к практике. В переменной `aclInfo` у нас уже есть сведения о записях ACL. Теперь нужно просто просмотреть его и вывести информацию на экран. Для этого цикл должен выглядеть примерно следующим образом:

```
for i:=0 to aclInfo.AceCount-1 do
begin
 if not (GetAce(pDACL^, i, Pointer(ace))) then
  continue;
 // Разбор записи ACE
end;
```

Список ACL содержит только указатели на ACE-записи. Напоминаю, что ACL — это список контроля доступа, а ACE — это отдельная запись этого списка. Чтобы получить запись из списка контроля доступа, нужно использовать функцию `GetAce`. Ей передаются три параметра:

- указатель на ACL;

- индекс интересующей нас записи;
- указатель на переменную, в которую будет записан результат.

Если результат не нулевой, значит, процесс получения записи прошел удачно. В примере выше, в качестве третьего параметра передается указатель переменной `ace`. Ее нужно объявить следующим образом:

```
ace : ^ACCESS_ACE;
```

То есть это указатель на структуру `ACCESS_ACE`, формат которой мы так тщательно выясняли, благодаря файлу помощи и MSDN.

Просматривая список, мы получили ACE-запись и по ней теперь должны определить, что разрешено, а что запрещено. Но одна запись может быть либо разрешающей, либо запрещающей. Нет такой записи, которая описывала бы и то, и другое сразу. Если нужно что-то запретить, а что-то разрешить, то в списке ACL создаются две записи ACE. Одна запись что-то разрешает, а другая что-то запрещает.

Чтобы определить, какая именно перед нами запись, необходимо проверить поле `AceType` заголовка ACE-записи:

```
case (ace^.Header.AceType) of
 ACCESS_ALLOWED_ACE_TYPE: Это разрешение;
 ACCESS_DENIED_ACE_TYPE: Это запрещение;
 else Что-то другое;
end;
```

Константы `ACCESS_ALLOWED_ACE_TYPE` и `ACCESS_DENIED_ACE_TYPE` опять же в заголовочных файлах Delphi я не нашел, поэтому их придется описать в проекте самостоятельно следующим образом:

```
ACCESS_ALLOWED_ACE_TYPE = $00;
ACCESS_DENIED_ACE_TYPE  = $01;
```

Значения этих констант я взял из заголовочного файла Visual Studio.

Теперь, если это разрешение или запрещение, нужно узнать, что именно разрешается. Об этом можно узнать из параметра `mask`. Тут основные биты — первые три:

- чтение;
- запись;
- выполнение.

А на какого пользователя или группу влияют данные разрешения? Об этом можно узнать по полю `SidStart` структуры ACE. Но это всего лишь указа-

тель на идентификатор пользователя/группы. Реальное значение можно определить следующей строкой кода:

```
sid := PSID(@((ace)^.SidStart));
```

Теперь, по идентификатору необходимо узнать имя пользователя и домен, в котором он зарегистрирован. Для этого используем функцию `LookupAccountSid`, которая выглядит так:

```
function LookupAccountSid(
 lpSystemName: PChar;
 Sid: PSID;
 Name: PChar;
 var cbName: DWORD;
 ReferencedDomainName: PChar;
 var cbReferencedDomainName: DWORD;
 var peUse: SID_NAME_USE
): BOOL; stdcall;
```

Быстренько пробежимся по параметрам этой функции:

- `lpSystemName` — указатель на строку для системного имени. Мы будем определять пользователя по SID, поэтому этот параметр должен быть нулевым;
- `Sid` — идентификатор интересующего нас пользователя;
- `Name` — указатель на строку, куда будет записано имя пользователя;
- `cbName` — размер строки для имени пользователя;
- `ReferencedDomainName` — строка для имени домена;
- `cbReferencedDomainName` — размер строки с именем домена;
- `peUse` — структура, определяющая тип записи.

Пример использования функции:

```
var
 user, domain : array [0..200] of char;
 len: DWORD;
begin
 ...
 LookupAccountSid(nil, sid, user, len, domain, len, sid_nu);
 ...
end;
```

Теперь соберем все вышесказанное воедино и напишем полноценный пример. Для этого создадим новый проект, а на форме нам понадобятся: поле для ввода имени файла, кнопочка и список `ListView` из четырех колонок:

- имя пользователя;
- домен;
- доступ (тип ACE-записи);
- действия (разрешаемые или запрещаемые).

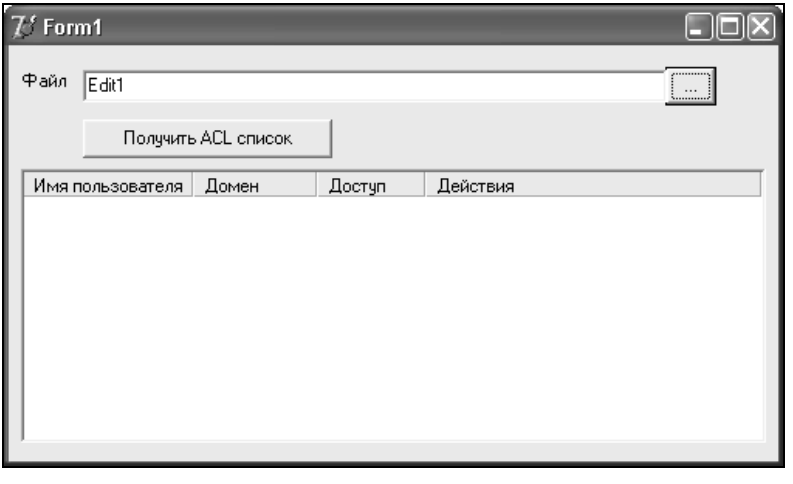


Рис. 3.10. Форма будущей программы

На рис. 3.10 вы можете увидеть форму будущей программы. Для события нажатия кнопки нужно написать код из листинга 3.10. Для понимания этого кода даже комментарии не нужны, потому что все функции и действия мы уже подробно рассмотрели. Не забудьте добавить в проект описание структур и переменных, которые мы рассматривали в этом разделе.

Листинг 3.10. Код чтения списков доступа для файла

```
// Очищаем компонент ListView
aclListView.Items.Clear;
 // Получаем ACL
if (GetNamedSecurityInfo(PChar(Edit1.Text), SE_FILE_OBJECT,
    DACL_SECURITY_INFORMATION, nil, nil, PACL(@pDACL), nil,
    pSD)<>ERROR_SUCCESS) then
```

```
 begin
  ShowMessage('Ошибка');
  exit;
 end;

// Проверяем список ACL на ноль
if (pDACL=nil) then
 begin
  ShowMessage('Список доступа пуст');
  exit;
 end;

// Читаем непосредственно массив ACL-записей
if (not GetAclInformation(pDACL^, @aclInfo, sizeOf(aclInfo),
    AclSizeInformation)) then
 begin
  ShowMessage('Не получилось определить информацию об ACL');
  exit;
 end;

// Цикл перебора массива
for i:=0 to aclInfo.AceCount-1 do
 begin
  // Прочитать ACE-запись
  if not (GetAce(pDACL^, i, Pointer(ace))) then
   continue;

  // Добавляем элемент в ListView
  newAcl:=aclListView.Items.Add;

  // Ищем имя пользователя по SID
  sid := PSID(@((ace)^.SidStart));
  len := 200;
  if (LookupAccountSid(nil, sid, user, len,
      domain, len, sid_nu)) then
```

```
   begin
    newAcl.Caption:=user;
    newAcl.SubItems.Add(domain)
   end
  else
   begin
    newAcl.Caption:='проблема';
    newAcl.SubItems.Add('проблема')
   end;

  // Смотрим, какой тип записи — разрешение или запрет
  case (ace^.Header.AceType) of
   ACCESS_ALLOWED_ACE_TYPE: newAcl.SubItems.Add('Разрешено');
   ACCESS_DENIED_ACE_TYPE: newAcl.SubItems.Add('Запрещено');
   else newAcl.SubItems.Add('Другое');
  end;

  actions:='';
  if (ace^.Header.AceType=ACCESS_ALLOWED_ACE_TYPE) or
     (ace^.Header.AceType=ACCESS_DENIED_ACE_TYPE) then
   begin
    // Что разрешено — чтение, запись или выполнение
    if (ace^.Mask and $1)=1 then
     actions:=actions+' Чтение';
    if (ace^.Mask and $2)=2 then
     actions:=actions+' Запись';
    if (ace^.Mask and $4)=4 then
     actions:=actions+' Выполнение';
   end;
  newAcl.SubItems.Add(actions);
 end;
end;
```

Для компиляции примера желательно подключить заголовочные файлы `AclApi`, `AccCtrl` и `Windows`, где описаны все необходимые для работы с ACL-функции и структуры. Хочу сделать одно замечание — в данном разде-

ле я просто рассматривал функции и примеры их вызова без проверок на ошибки, дабы сэкономить драгоценное место на страницах книги. В примере, который вы найдете на компакт-диске, все проверяется на ошибки, потому что они могут быть. Если пользователь выберет файл, находящийся на FAT32-разделе, то произойдет ошибка, ведь эта файловая система не поддерживает списки доступа.

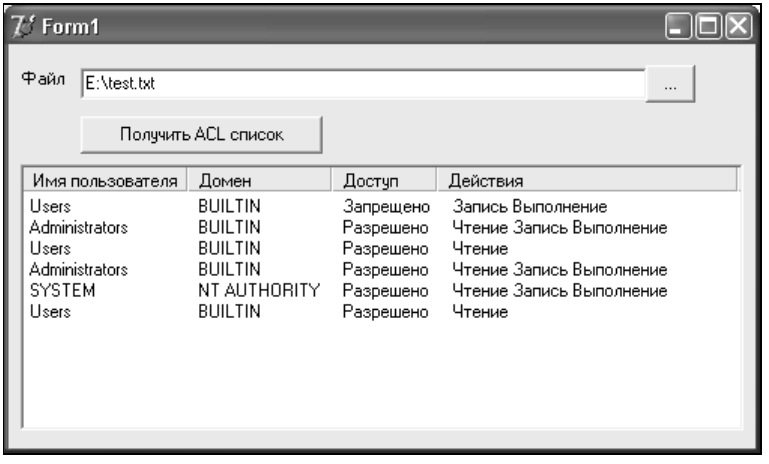


Рис. 3.11. Результат работы

Результат работы программы представлен на рис. 3.11.

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\DacI2\ вы можете увидеть пример этой программы.

3.5.3. Редактирование прав доступа

Мы уже узнали, как можно определить права доступа на определенный файл в файловой системе NTFS. В этот раз нам предстоит пойти дальше, и мы научимся модифицировать таблицу, добавляя или удаляя определенные права доступа. Все это будет происходить из Delphi программно — и никакого колдовства. Только ловкость рук и немного смекалки. Не забываем, что права доступа могут быть не только у файлов, но и у других объектов ОС.

Когда я впервые разбирался с дескрипторами, то большинство приходилось определять методом научного тыка. Всезнающий MSDN помогает, но очень

слабо, а в Интернете эта тема освещена просто ужасно, только обрывки информации и нет нормального кода. В MSDN описаны только функции и их параметры, а как пользоваться ими, сказано только поверхностно. Как говорил Винни-Пух: "Мед вроде есть, но его сразу нет". Здесь та же песня, вроде бы функции описаны, но чего-то не хватает.

Давайте для начала на пальцах разберемся, как нужно устанавливать права на объект ОС, а потом познакомимся с функциями и практическим примером. Под объектом в данном случае понимаются процессы и файлы, находящиеся на NTFS. Напоминаю, что в FAT32 слишком скудные возможности по защите файлов, и здесь списки ACL не работают, их просто невозможно сохранить на диске, ведь информация о доступе хранится в файловой системе, а в FAT эту информацию негде хранить.

Если вы все еще используете FAT, то рекомендую сегодня же конвертнуть ее в NTFS, и ничего не бойтесь. Файловая система NTFS намного надежнее и лучше. На рис. 3.12 можно увидеть окно свойств файла, а точнее, вкладку **Security** окна, где можно настраивать доступ.

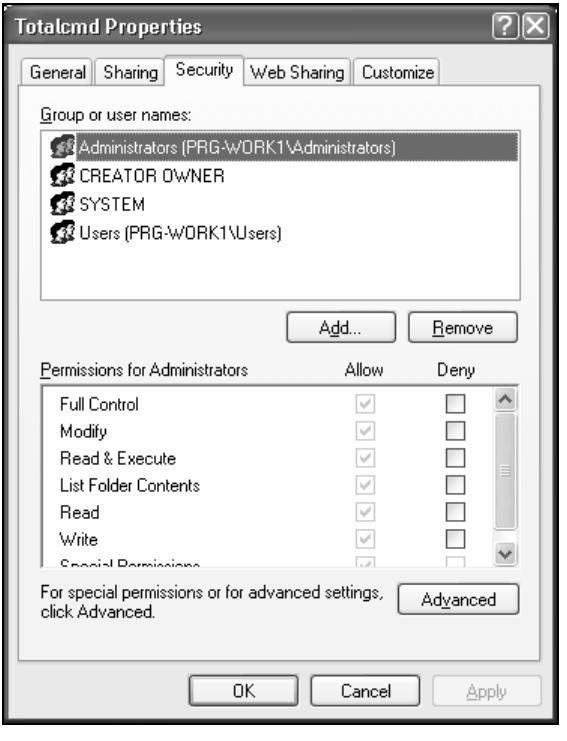


Рис. 3.12. Права доступа в NTFS

У каждого объекта есть свой дескриптор. В нем содержится указатель на список ACL, а уже в этом списке можно найти записи, которые определяют права доступа. Самый простой способ удалить определенную запись из списка — скопировать все необходимые записи в новый список и установить их дескриптору, а затем дескриптор назначить файлу. Так как списки в основном небольшие и занимают мало места, такой финт пролетит мгновенно.

Чтобы добавить разрешение или запрет, необходимо расширить память для ACL-списка и втиснуть туда новую запись. Можно поступить проще — опять же выделить память для нового списка, достаточную для хранения всех старых записей плюс одна новая. Теперь копируем в новый ACL все старые записи и добавляем новую. Остается только назначить созданный список дескриптору, а затем и файлу.

Для написания примера нам понадобятся все знания, которые мы получили в двух предыдущих разделах этой главы, плюс маленькая корзина новых функций. Давайте напишем пример, который будет добавлять в ACL новую разрешающую запись для всех пользователей на определенный файл. В ОС Windows для этого нужно найти SID учетной записи **Everyone** и добавить ее с разрешением `GENERIC_ALL` в список ACL.

Начнем с поиска SID необходимой записи. Следующий код показывает, как можно найти идентификатор для **Everyone**:

```
// Определяем размер SID
LookupAccountName(nil, 'everyone', nil, sidLength,
  nil, sidLengthDomain, sidType);

// Выделяем память
sidValue := AllocMem(sidLength);
domain1 := AllocMem(sidLengthDomain);

// Получаем SID
if (LookupAccountName(nil, 'everyone', sidValue,
 sidLength, domain1, sidLengthDomain, sidType)=false) then
  exit;
```

Все эти функции мы уже знаем, поэтому ничего нового я тут рассказать не могу.

Теперь определяем текущую информацию безопасности файла с помощью `GetNamedSecurityInfo` и ACL-информацию с помощью `GetAclInformation`. Эти функции мы использовали в *разд. 3.5.2*. Можете взять код один к одному и даже форму главного окна для выбора файла можно оставить старую. Нужно только добавить кнопку: по ее нажатию будет происходить разреше-

ние доступа, которое мы будем рассматривать в этом разделе. Обновленную форму можно увидеть на рис. 3.13.

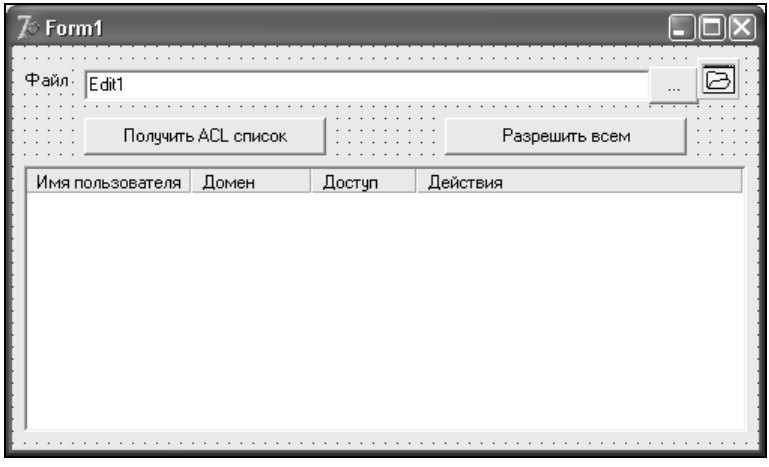


Рис. 3.13. Обновленная форма для нашего примера

Получив информацию о текущем списке, необходимо выделить память под новый список. Как рассчитать новый размер? Все достаточно просто. У нас есть размер текущего списка в поле `AclBytesInUse` структуры `AclInfo`. Эту структуру мы получили после вызова функции `GetAclInformation`. К этому размеру мы добавляем размер разрешающей структуры `ACCESS_ACE` и размер SID идентификатора пользователя, которого мы хотим внести в список:

```
newSize:=AclInfo.AclBytesInUse + sizeof(ACCESS_ACE) +
   GetLengthSid(sidValue);
```

В файле справки из этого результата еще вычитается размер `DWORD`, но я этого не делаю, и у меня пример прекрасно работает. Если будут проблемы, попробуй изменить строку так:

```
newSize:=AclInfo.AclBytesInUse + sizeof(ACCESS_ACE) +
  GetLengthSid(sidValue) - sizeof(DWORD);
```

Не знаю, зачем надо вычитать размер числа `DWORD`, возможно, что это действительно нужно.

Теперь выделяем память для нового ACL-списка и инициализируем его:

```
pNewDACL:=PACL(LocalAlloc(LPTR, newSize));
if not InitializeAcl(pNewDACL^, newSize, 2) then
 exit;
```

Для инициализации используется функция InitializeAcl, которая выглядит следующим образом:

```
function InitializeAcl(
 var pAcl: TACL;
 nAclLength,
 dwAclRevision: DWORD
): BOOL; stdcall;
```

Функция получает в качестве параметров три значения:

- указатель на выделенную для списка память;
- размер списка;
- ревизию, которая должна быть равна ACL_REVISION для данной версии Windows.

Мы будем использовать последнюю ревизию ACL_REVISION, которая равна 2, но если указать 1, то ошибки не должно быть. В старых версиях Windows лучше не экспериментировать, а сразу указывать первую версию, т. е. единицу. Я не проверял, но возможна неверная работа примера при других значениях.

Теперь запускаем цикл перебора всех записей и копируем их в новый список, дабы ничего не потерять:

```
for i:=0 to aclInfo.AceCount-1 do
begin
 // Получаем текущую запись
 if not (GetAce(pDACL^, i, Pointer(ace))) then
  continue;

 // Добавляем ее в новый список
 if not AddAce(pNewDACL^, 2, MAXWORD, ace, ace.Header.AceSize) then
  continue;
end;
```

Цикл прост, дальше некуда. Сначала пытаемся получить текущую запись. Если неудачно, то машем на нее рукой и переходим на следующий шаг к очередной записи. Если ACE-запись получена, то добавляем ее в новый список с помощью функции AddAce, которая в общем виде выглядит так:

```
function AddAce(
 var pAcl: TACL;
```

```
 dwAceRevision,
 dwStartingAceIndex: DWORD;
 pAceList: Pointer;
 nAceListLength: DWORD
): BOOL; stdcall;
```

Тут у нас 5 параметров:

- указатель на список, в который нужно добавить запись;
- уже знакомая нам ревизия, которая должна быть равна `ACL_REVISION`;
- позиция, в которую нужно вставить запись. Если указать ноль, то вставка произойдет в начало, а если указать `MAXWORD`, то вставка произойдет в конец списка;
- указатель на одну или несколько ACE-записей;
- общий размер добавляемых записей.

Все существующие записи мы уже скопировали в список. Если какую-то из них нужно было удалить, то ее можно было не копировать, но тогда следовало бы правильно пересчитать размер списка.

Теперь нам предстоит добавить новую разрешающую запись, которая не существовала ранее. Для этого используется функция `AddAccessAllowedAce`, которой необходимо передать: указатель на список ACL, ревизию, маску прав доступа и идентификатор SID пользователя, которому дается разрешение. В заголовочном файле windows.pas эта функция объявлена следующим образом:

```
function AddAccessAllowedAce(
 var pAcl: TACL;
 dwAceRevision: DWORD;
 AccessMask: DWORD;
 pSid: PSID
): BOOL; stdcall;
```

Маску прав доступа мы рассматривали в прошлый раз, когда учились читать записи. В нашем случае необходимо разрешить все, а значит, в третьем параметре указываем флаг `GENERIC_ALL`. SID пользователя **Everyone** мы уже нашли, а значит, добавление новой записи в ACL-список будет выглядеть следующим образом:

```
AddAccessAllowedAce(pNewDACL^, 2, GENERIC_ALL, sidValue);
```

Для добавления запрещающей записи необходимо использовать функцию `AddAccessDeniedAce`. У нее параметры такие же, как и у `AddAccessAllowedAce`,

разница только в том, что указанные флаги доступа действуют как запрещение. Так как функции очень похожи внешне и одинаковы при вызове, то не будем тратить на рассмотрение запрета лишнее место и ваше время.

Все, новый список готов, но он пока еще ни с чем не связан. Необходимо создать новый дескриптор для данного списка. Старый использовать не получится, потому что он занят. Для создания нового дескриптора заведем переменную `pNewSD` типа `PSECURITY_DESCRIPTOR`. Эта переменная является указателем, а значит, требует выделения памяти. Сколько памяти выделить? Вполне достаточно минимума, который равен значению константы `SECURITY_DESCRIPTOR_MIN_LENGTH`. В моем заголовочном файле windows.pas эта константа равна 20. После выделения памяти инициализируем дескриптор с помощью функции `InitializeSecurityDescriptor`:

```
// Выделяем память под дескриптор
pNewSD := PSECURITY_DESCRIPTOR(
  LocalAlloc(LPTR, SECURITY_DESCRIPTOR_MIN_LENGTH));
// Инициализируем дескриптор
InitializeSecurityDescriptor(pNewSD, SECURITY_DESCRIPTOR_REVISION);
```

Инициализация обязательна, иначе дескриптор будет недоступен, и его нельзя будет связать с ACL-списком.

У функции `InitializeSecurityDescriptor` два параметра — указатель на переменную-дескриптор и ревизия, которая должна быть равна константе `SECURITY_DESCRIPTOR_REVISION`. В заголовочном файле эта константа равна 1.

Теперь связываем дескриптор с нашим списком с помощью функции `SetSecurityDescriptorDacl`. У этой функции четыре параметра:

- указатель на дескриптор, который нужно связать с ACL-списком;
- флаг, есть ли ACL-список. Если параметр равен `true`, то он есть в третьем параметре;
- указатель на ACL-список, с которым происходит связь;
- флаг, который при равенстве `true` означает, что будет использоваться ACL по умолчанию.

Список доступа есть, он уже связан с дескриптором, осталось только назначить этот дескриптор файлу, и мы в шоколаде. Для этого используем функцию `SetFileSecurity`, у которой три параметра: путь к файлу, тип изменяемой информации (в нашем случае это `DACL_SECURITY_INFORMATION`) и новый дескриптор. Вот так объявлена эта функция:

```
function SetFileSecurity(
 lpFileName: PChar;
```

```
 SecurityInformation: SECURITY_INFORMATION;
 pSecurityDescriptor: PsecurityDescriptor
): BOOL; stdcall;
```

Вот и все, новый список назначен файлу, и теперь с этим файлом любой пользователь может делать все, что угодно, потому что мы дали разрешение с маской доступа `GENERIC_ALL`.

Работа с безопасностью окон — достаточно нудное и немного запутанное занятие. Но трудно не согласиться, что упрощать нет смысла, ведь все продумано до мелочей. Другое дело, как эти мелочи реализованы. На данный момент система безопасности Windows уже отлажена и при правильном подходе эффективна.

Когда мы программно формировали новый список, то все ACE-записи банально добавлялись в самый конец без сортировки. Если после этого отобразить окно свойств файла и перейти на вкладку **Security**, то ОС Windows предложит отсортировать список. Пример сообщения, запрашивающего сортировку, показан на рис. 3.14. Конечно же желательно согласиться, иначе вы можете увидеть некорректный список.

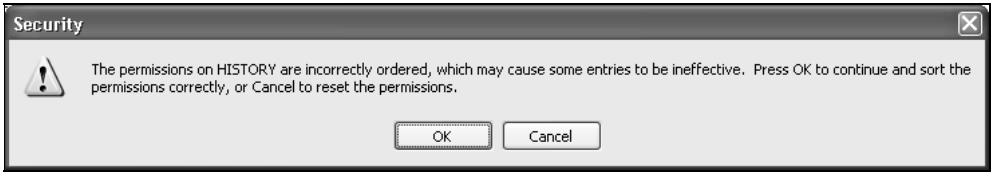


Рис. 3.14. Запрос на сортировку записей безопасности

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\Dacl3\ вы можете увидеть пример этой программы.

3.6. Сервисы

Сервисы (или службы), на мой взгляд, стали одним из самых слабых мест в Windows 2000, потому что здесь может зарегистрироваться практически любая программа, а пользователи боятся заглядывать в оснастку сервисов и смотреть, что здесь происходит. Именно поэтому среди зловредного кода поднимается волна использования технологии сервисов в целях сокрытия своих действий.

В Delphi создание служб было упрощено, дальше просто некуда. Давайте посмотрим, как в Delphi можно создать собственный сервис. Это можно сделать с помощью сложной и неприятной регистрации через WinAPI или с использованием средств, предоставляемых средой разработки Delphi. Зачем создавать себе лишние проблемы, когда есть готовый класс `TService`, которым мы сегодня и воспользуемся.

Запускаем Delphi и выбираем меню **File | New | Other** (для Delphi меньше 7-й версии просто **File | New**) и в появившемся окне ищем **Service Application** (рис. 3.15). Выбираем этот пункт, нажимаем кнопку **OK**, и можно считать, что сервис готов.

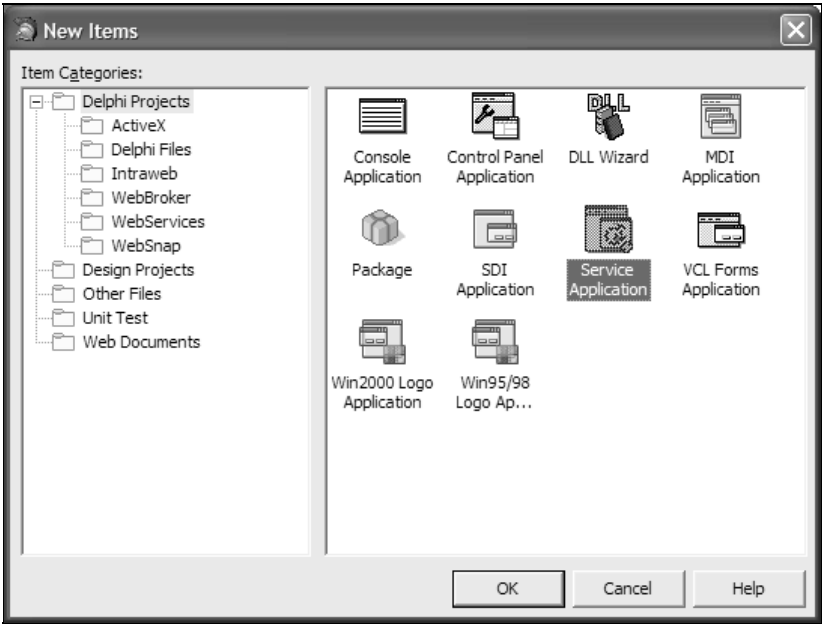


Рис. 3.15. Окно выбора типа создаваемого проекта с выделенным **Service Application**

Код главного модуля похож на модули других типов проектов. Но изюминка спрятана немного глубже, а именно в объекте, от которого все происходит. Если стандартные окна приложений происходят от класса `TForm`, то тут мы пляшем вальсом от `TService`.

Визуальная форма больше похожа на модуль данных Data Module, и на ней можно размещать только невизуальные компоненты. Это вполне логично, ведь сервис работает невидимо для пользователя, и ничего визуального на

главной форме ненужно и не должно быть. Визуальными могут быть какие-то дополнительные окна, которые выскакивают в ответ на определенные события, но сам сервис будет невидим.

Давайте посмотрим на самые "вкусные" свойства `TService`, которые вы можете увидеть в Инспекторе объектов.

- `AllowPause` — разрешить пользователю приостанавливать работу сервиса. Я злой, поэтому в своем сервисе отключаю эту возможность. Незачем тормозить мои великие творения.
- `AllowStop` — позволять пользователю останавливать сервис. И снова моя злость заставляет убрать эту возможность.

  Если запретить возможность остановки и приостановки сервиса, то в оснастке служб соответствующие кнопки будут недоступны, и на первый взгляд оставить такое зло нереально. Но не радуйтесь раньше времени, потому что для опытного пользователя удалить даже такой сервис не составит труда. Достаточно выполнить запускной файл сервиса и в качестве параметра указать ключ `/UNINSTALL`, так все остановится и удалится из системы. При следующем старте Windows данный сервис уже не запустится. От этого никуда не денешься, поэтому косметическая операция по запрету остановки подействует только на неопытных. Благо на нашей земле их более чем достаточно.
- `DisplayName` — отображаемое имя. Именно этот текст можно будет увидеть в оснастке сервисов в качестве имени. В реальном шуточном проекте я бы посоветовал подойти к выбору имени максимально тщательно. Как корабль назовешь, так он и поплывет.

  Например, можно написать здесь: "Служба безопасности NTFS". Тогда ни у кого не поднимется рука остановить такое. А лучше напишите на английском, чтобы совсем испугать бедного и несчастного пользователя. Но и запускной файл в этом случае должен будет иметь соответствующее имя, а не просто Project1.exe, а то с такими шутками только блох смешить.
- `ErrorSeverity` — что делать, если во время запуска произошла ошибка. Здесь можно указать одно из следующих значений:
  - `esIgnore` — продолжить выполнение;
  - `esNormal` — вывести сообщение, но продолжить работу;
  - `esSevere` — продолжить работу, если стартует конфигурация, которая уже стартовала удачно, или запустить конфигурацию, которая стартовала удачно;
  - `esCritical` — запустить конфигурацию, которая стартовала удачно, но если сейчас стартует именно она, то запуск выдает ошибку.

В любом случае в системном журнале прописывается запись о происшедшей ошибке, но вот на экране нам отображать ничего не надо. Незачем пользователя смущать лишними сообщениями, иначе из-за какой-то меленькой ошибки начнутся раскопки, какой за сервис выдал ошибку и зачем. Поэтому измените параметр `ErrorSeverity` на `esIgnore`, чтобы в случае внештатной ситуации не вызывать лишних подозрений. Системные журналы проверяют редко, а вот сообщение на экране упустить из виду просто нереально.

- `ServiceStartName` и `Password` — это имя учетной записи и пароль, под которыми будет работать сервис. От этого зависят права на доступ к различным объектам. Если вы заведомо знаете пароль администратора машины жертвы, то можете указать его здесь, иначе оставьте эти параметры пустыми, чтобы сервис работал под системной учетной записью.
- `Dependencies` — зависимости. Если дважды щелкнуть по этому свойству, то появится окно, где можно указать сервисы, от которых будет зависеть ваш. Это значит, что все они должны будут запуститься раньше.
- `ServiceType` — тип сервиса. Существует три типа:
  - `stWin32` — стандартный оконный сервис, то, что нам и нужно;
  - `stDevice` — для драйверов устройств;
  - `stFileSystem` — драйвер файловой системы.
- `StartType` — тип запуска сервиса по умолчанию. Здесь можно указать одно из следующих значений:
  - `stBoot` — используется оконный загрузчик, когда тип сервиса не `stWin32`;
  - `stSystem` — стартовать после инициализации системы;
  - `stAuto` — запускаться автоматически во время загрузки системы. Для злого сервиса это идеальный вариант;
  - `stManual` — сервис запускается вручную;
  - `stDisabled` — отключено.

Не будем скромничать, а сделаем так, чтобы сервис по умолчанию запускался автоматически `stAuto`.

Самое важное для сервиса прячется в событиях объекта `TService`. Давайте взглянем, что тут у нас есть:

- `AfterInstall` — генерируется после инсталляции сервиса;
- `AfterUninstall` — генерируется после удаления сервиса;
- `BeforeInstall` — генерируется после инсталляции сервиса;

- `BeforeUninstall` — генерируется после удаления сервиса;
- `OnContinue` — запуск после паузы;
- `OnPouse` — сервис приостановлен;
- `OnShutdown` — сервис остановлен на выключение;
- `OnStart` — сервис стартовал;
- `OnStop` — сервис остановился.

Какие события нам выбрать? На первый взгляд, злой код должен находиться в событии `OnStart`. Это верно, но не на все 100%, потому что злится надо на два события: `OnStart` и `OnInstall`. Когда пользователь инсталлирует сервис, мы уже можем сделать что-то интересное, не дожидаясь нормального запуска сервиса.

Уже сейчас можно скомпилировать проект и установить в систему получившийся сервис. Правда, он пока еще ничего не делает, но потренироваться с установкой можно. Чтобы проинсталлировать сервис, нужно скомпилировать проект (нажать комбинацию клавиш <Ctrl>+<F9>) и запустить программу с ключом `/INSTALL`. Чтобы удалить из системы, нужно выполнить программу с ключом `/UNINSTALL`.

Если вы пишите сервис с запрещенной возможностью остановки, то советую на время тестирования разрешить старт/стоп. Иначе после запуска сервиса невозможно перекомпилировать файл и приходится удалять его из системы и перегружаться. Когда все будет готово, вот тогда и выставите в свойстве `AllowStop` значение `false`.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 3\thebat\ вы можете увидеть пример сервиса, который собирает адреса e-mail в базе данных программы The Bat!. А в файле \Документация\Coding\thbat.pdf вы найдете подробное описание примера в виде статьи, которую я написал несколько лет назад для журнала "Хакер".

3.7. Управление менеджером сервисов

Буквально вчера я увидел в Интернете оригинальную новость о том, что появилась новая зло-программа, которая после заражения компьютера скачивает пиратскую версию Касперского и с его помощью уничтожает всех конкурентов. Оригинальное и поразительное решение. Как только автор додумался до такого. Мы много раз слышали о том, что вирусы пытаются обмануть антивирусы, но такая странная дружба — на моей памяти проис-

ходит впервые. Этот случай лишний раз подтолкнул меня к тому, чтобы поговорить о том, как можно обмануть антивирус.

Многие хакеры стремятся написать такой код вируса, трояна или другого зла, чтобы анализатор антивируса не смог найти злостный код. Да, это красиво и такая война оправдывает затраты, потому что элегантность в цене. Но иногда необходима скорость решения, а тут на первый план выходит простота. Именно в ней кроется вся гениальность.

К чему я все это? А к тому, что самое простое и эффективное решение кроется в отключении сервиса антивируса. Не знаю, как Касперский (давно его не юзал), а Norton и Dr.Web работают как сервисы, а значит, чтобы наш зло-код спокойно работал в системе, не боясь антисредств: достаточно только отключить соответствующий сервис, и никто тебя уже не остановит. Самое поразительное, что можно создавать сервисы, которые нельзя будет остановить, но известные мне антивирусы легко отключаются. Это вполне логично, ведь анализаторы иногда ошибаются (сам с таким встречался) и тогда пользователь должен иметь возможность отключить слишком умный антивирус и позволить продолжить работу с программами, которые анализатор воспринял как зло.

Еще пример — недавно мы говорили о том, что сервис `Messenger` не безопасен и подвержен атаке флудом, поэтому его начали отключать. Но кто мешает нам написать небольшую программу, которая будет запускать сервис на компьютере жертвы, чтобы потом его можно было атаковать флудом через `NET SEND`.

Итак, сегодня нам предстоит узнать, как можно программно запустить, остановить любой сервис в системе или вогнать в его паузу. Нетрудно догадаться, что все это мы будем делать программно и с помощью Delphi, ведь именно этому языку программирования посвящена данная книга.

Приступим. За управление сервисами в окнах отвечает менеджер сервисов (Service Control Manager, что на нашем великом и могучем означает — менеджер управления сервисами). Очень часто, в том числе и в документации MSDN, можно встретить сокращение SCManager.

На рис. 3.16 вы можете увидеть стандартное окно Windows для управления сервисами. Эта оснастка использует те же функции, что мы будем рассматривать сегодня, поэтому и возможности у нашей программы смогут быть идентичными.

Первое, что нам необходимо сделать, — подключиться к менеджеру. Для этого используется функция `OpenSCManager`, которая в общем виде выглядит так:

```
function OpenSCManager (
 lpMachineName,
```

```
 lpDatabaseName: PChar;
 dwDesiredAccess: DWORD
): SC_HANDLE; stdcall;
```

Тут у нас всего три параметра:

- имя компьютера, к менеджеру которого необходимо подключиться. Да, если есть права, то можно управлять даже удаленным компьютером. Для подключения к локальному компьютеру этот параметр можно оставить нулевым;
- база данных, которая нас интересует. Этот параметр должен быть равен `SERVICES_ACTIVE_DATABASE` или нулю. В обоих случаях результат один и тот же — выбирается активная база данных;
- флаг, определяющий желаемый доступ.

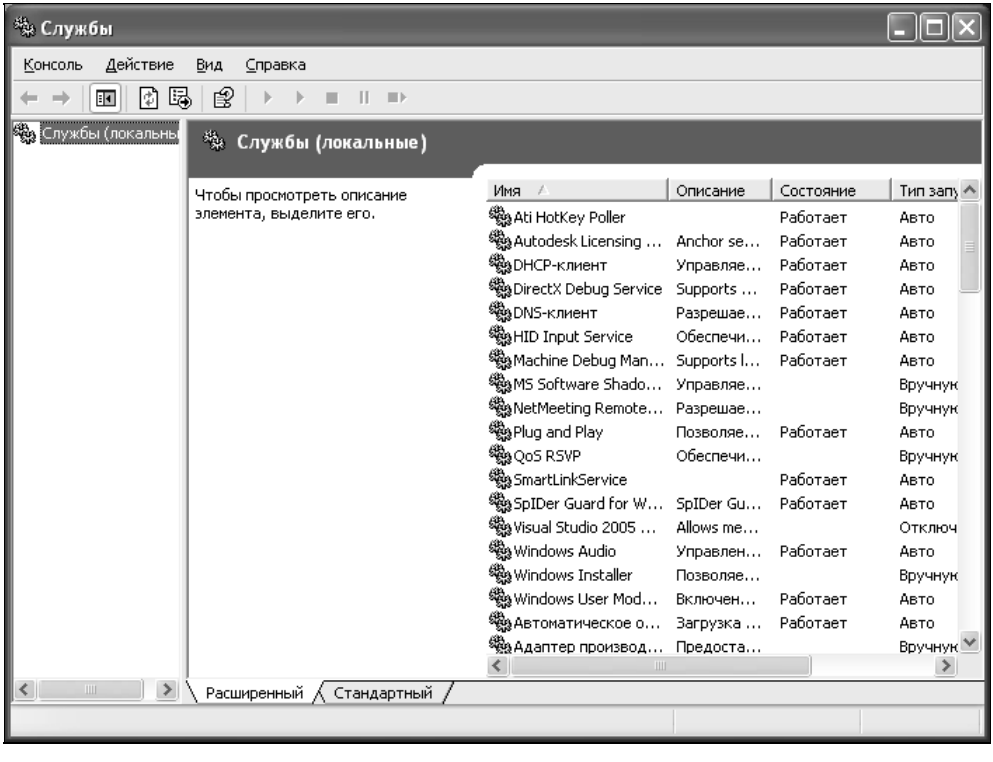


Рис. 3.16. Оснастка Службы управляет сервисами теми же функциями, которые мы будем рассматривать

В зависимости от текущих прав, под которыми работает пользователь, можно указать различные флаги прав доступа к менеджеру сервисов. Нас интересует полный доступ, а значит, в третьем параметре необходимо указать SC_MANAGER_ALL_ACCESS. Полный доступ будет доступен, только если компьютер работает под правами администратора. Слава богам за то, что большинство ламеров работает именно под такими правами, особенно в Windows Home Edition, где каждый юзер является локальным админом.

Если программа работает с правами **LocalSystem**, то можно указать следующие права доступа:

- SC_MANAGER_CONNECT — разрешено подключение к менеджеру;
- SC_MANAGER_ENUMERATE_SERVICE — разрешено перечисление сервисов;
- SC_MANAGER_QUERY_LOCK_STATUS — разрешен запрос состояния;
- STANDARD_RIGHTS_READ — стандартные права чтения;
- SC_MANAGER_MODIFY_BOOT_CONFIG — разрешено изменение загрузки.

Если функция отработала успешно и смогла подключиться, то результатом будет Hanndle выбранной базы данных SC-менеджера. Если подключение невозможно, то результатом будет ноль.

Итак, чтобы подключиться к менеджеру управления сервисами на локальном компьютере, достаточно написать следующий код:

```
var
 servicemanager:Cardinal;
begin
 servicemanager:=OpenSCManager(nil, nil, SC_MANAGER_ALL_ACCESS);
end;
```

К менеджеру мы подключились, теперь нужно подключиться к необходимому сервису. Для этого используется метод OpenService, который выглядит так:

```
function OpenService (
 hSCManager: SC_HANDLE;
 lpServiceName: PChar;
 dwDesiredAccess: DWORD
): SC_HANDLE; stdcall;
```

Тут у нас снова три параметра:

- дескриптор менеджера сервисов, который мы получили после выполнения функции OpenSCManager;
- имя сервиса, которым нужно управлять;

- права доступа. В качестве прав доступа при подключении к сервису можно указывать:
  - `SERVICE_ALL_ACCESS` — полный доступ;
  - `SERVICE_START` — разрешить запуск сервисов;
  - `SERVICE_STOP` — разрешить останавливать сервисы;
  - `SERVICE_PAUSE_CONTINUE` — разрешено отправлять сервис в паузу и возобновлять выполнение.

  Это основные права, которые могут вам понадобиться.

В качестве результата функция возвращает дескриптор сервиса, если выполнение прошло успешно. Если произошла ошибка, то результатом будет ноль. Обязательно проверяйте на ошибку, ведь сервис может быть не установленным, а значит, его невозможно будет открыть и впоследствии управлять. Получается, что по наличию сервиса можно определить, какой антивирусник установлен на компьютере жертвы. Если соответствующий сервис открылся, то можно останавливать его.

Теперь можно начинать управление, и делается это с помощью функции `ControlService`, которая в общем виде выглядит следующим образом:

```
function ControlService(
 hService: SC_HANDLE;
 dwControl: DWORD;
 var lpServiceStatus: TserviceStatus
): BOOL; stdcall;
```

И снова три магических параметра:

- дескриптор сервиса, которым нужно управлять;
- действие, которое нужно выполнить. Основные команды, которые можно здесь указать, это:
  - `SERVICE_CONTROL_STOP` — остановить сервис;
  - `SERVICE_CONTROL_PAUSE` — сделать паузу, скушать "твикс";
  - `SERVICE_CONTROL_CONTINUE` — продолжить выполнения сервиса, который кушает "твикс";
- переменная типа `SERVICE_STATUS`, через которую мы получим последнее состояние сервиса.

Во время остановки можно встретиться с одной серьезной проблемой. Если есть зависимые работающие сервисы, то их остановить будет невозможно. Следует определить зависимости и остановить все. Но об этом чуть позже.

Теперь у вас есть необходимая информация, чтобы написать пример, который незаметно будет останавливать все, что мешает программе. В листинге 3.11 вы можете увидеть пример кода, который останавливает сервис `Messenger`.

Листинг 3.11. Пример остановки сервиса

```
var
 servicemanager, service:Cardinal;
 ss:SERVICE_STATUS;
begin
 // Открываем менеджер сервисов
 servicemanager:=OpenSCManager(nil, nil, SC_MANAGER_ALL_ACCESS);

 // Открываем сервис Messenger
 service:=OpenService(servicemanager, 'Messenger', SERVICE_ALL_ACCESS);

 // Проверяем на ошибочки
 if service=0 then
  begin
   ShowMessage('Ошибка');
   exit;
  end;

 // Останавливаем Messenger
 ControlService(service, SERVICE_CONTROL_STOP, ss);

 // Закрываем сервис
 CloseServiceHandle(service);
end;
```

Единственная функция, которая есть в этом примере, и я ее еще не описал, — это `CloseServiceHandle`. Она закрывает дескриптор, который мы открыли для управления. В качестве единственного параметра необходимо передать дескриптор открытого сервиса. С помощью этой функции необходимо закрывать не только дескриптор сервиса, но и менеджера управления сервисами.

Ах, чуть не забыл. Все описанные функции работы с сервисами и менеджером описаны в модуле WinSvc.pas, поэтому не забудьте прописать его в разделе `uses`, иначе пример не скомпилируется.

Обратите внимание, что при рассмотрении функции контроля `ControlService` мы не видели флага, который отвечал бы за запуск сервиса. Нет, я не опустил его из-за ненадобности, просто за запуск отвечает совершенно другая функция: `StartService`. Эта функция в общем виде выглядит следующим образом:

```
function StartService(
 hService: SC_HANDLE;
 dwNumServiceArgs: DWORD;
 var lpServiceArgVectors: Pchar
): BOOL; stdcall;
```

Давайте посмотрим на параметры этой функции:

- дескриптор сервиса, который нужно запустить;
- количество передаваемых аргументов. Они передаются в виде строк в третьем параметре функции;
- массив из строк аргументов. Каждый элемент массива — это строка, которая заканчивается нулем.

Не знаю, для чего в Microsoft запуск сервиса выделили в отдельную функцию и не смогли все сделать в универсальной `ControlService`. Мне кажется, что это связано с безопасностью, но когда кажется, нужно креститься. Оставим лишнюю функцию на совести разработчиков, хотя, их совесть выдерживала и не такое.

С запуском могут быть проблемы. Не каждый может стартонуть, особенно если для корректной работы требуется запуск других сервисов. Если результат ненулевой (`true`), то запуск произошел успешно, а если нулевой (`false`), то произошла ошибка.

С помощью функций менеджера управления сервисов вы легко можете написать собственную оснастку управления сервисами.

Давайте кратко пробежимся по функциям, которые есть в Windows и также относятся к управлению сервисами:

- `QueryServiceStatus` — запрашивает информацию о состоянии сервиса. У функции два параметра: дескриптор сервиса и переменная типа `SERVICE_STATUS`, куда будет записан результат;
- `CreateService` — создает сервис и добавляет его в базу данных определенного менеджера управления сервисами. Параметров у этой функции,

ну, очень много, поэтому поэкономим место книги. Да и инсталлировать сервисы просто и без этой функции, достаточно только запустить исполняемый файл сервиса;

- `DeleteService` — удалить сервис. Параметр только один, и это дескриптор удаляемого сервиса;
- `EnumDependentServices` — перечислить все сервисы, которые зависят от указанного. Таким образом, можно вычислить зависимости;
- `GetServiceDisplayName` — определяет дружественное имя сервиса, которое можно увидеть в оснастке служб;
- `EnumServicesStatus` — перечислить все установленные сервисы из базы данных и вернуть для каждого из них текущее состояние.

При работе с SC-менеджером нужно быть очень внимательными, потому что результат наступает не мгновенно. Сервис не может остановиться или запуститься мгновенно. Если запустить пример, который вы найдете на диске, и программно остановить сервис `Messenger`, то можно увидеть, что в оснастке Службы сервис сообщений будет некоторое время недоступной, потому что система будет все еще останавливать сервис, обновлять базу данных и т. д.

Если останавливать и запускать сервисы из оснастки работы с сервисами, то запуск и остановка будут также происходить не мгновенно и в зависимости от многих факторов может отнять от нескольких секунд до нескольких минут. Так что не надейтесь, что выключенный программно антивирус тут же перестанет сканировать и можно заражать что угодно и когда угодно.

Для того чтобы зло-код не смог программно отрубить антивирус и внедриться в вашу систему, достаточно просто работать на компьютере с правами простого пользователя, а не администратора. В этом случае код не сможет получить полного доступа к системе. Хотя нет ничего невозможного. Ведь есть куча способов поднять права, но это уже намного сложнее.

Программное отключение сервисов лишний раз подтверждает то, что защиты в Windows просто не может быть, особенно если пользователь работает под правами администратора. Любая зло-программа может незаметно вырубить эту защиту и уничтожить компьютер бедного пользователя. Да, виноват останется все тот же бедный юзер, но что он может сделать в Windows Home Edition?

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\Service\ вы можете найти пример этой программы.

3.8. Оснастка сервисов

В предыдущем разделе мы выяснили, как можно управлять определенным сервисом. В этом разделе мы пойдем дальше и узнаем, как можно узнать, какие именно сервисы установлены в системе, т. е. практически напишем собственную оснастку управления сервисами. Форму будущей программы можно увидеть на рис. 3.17.

Как видите, на форме нам понадобятся:

- кнопка, по нажатию которой можно обновить список сервисов;
- два компонента типа `RadioButton`, с помощью которых можно выбирать тип отображаемых сервисов. Существуют два типа — стандартные сервисы и драйверы;
- компонент `ListView`, в котором будет происходить отображение. У этого компонента в свойстве `ViewStyle` установите параметр `vsReport`, чтобы он выглядел в виде таблицы. Теперь щелкните по нему дважды, чтобы открыть окно настроек колонок, и добавьте три колонки с именами:
  - имя;
  - дружественное имя;
  - состояние.

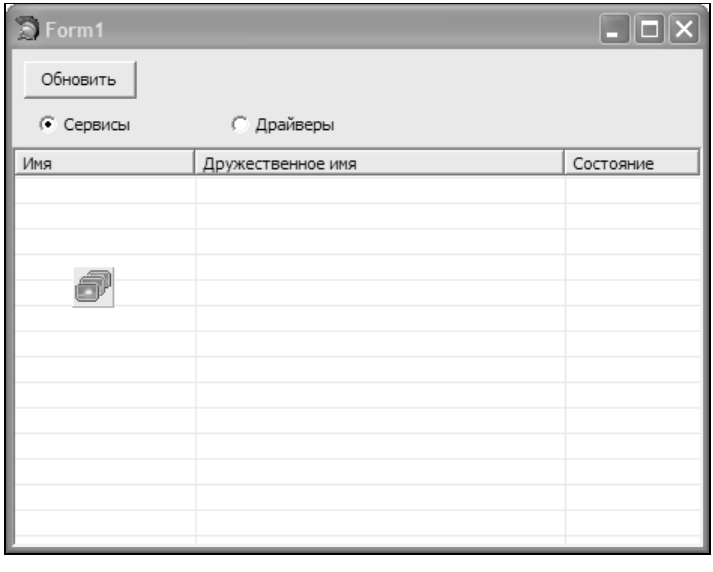


Рис. 3.17. Форма будущей программы

Теперь, для события нажатия кнопки обновления списка напишите код, показанный в листинге 3.12.

Листинг 3.12. Код получения списка установленных сервисов

```
procedure TForm1.Button1Click(Sender: TObject);
var
 ss: array of TEnumServiceStatus;
 dwNeaded, dwReturn, dwType: DWORD;
 lpResumeHandle: THandle;
 i:Integer;
begin
 // Начало обновления списка
 lwServices.Items.BeginUpdate;
 lwServices.Items.Clear;

 // Если соединение уже было установлено ранее, то закрыть его
 if servicemanager<>0 then
  CloseServiceHandle(servicemanager);
 // Открыть менеджер сервисов
 servicemanager:=OpenSCManager('', nil, SC_MANAGER_ALL_ACCESS);

 // Если соединение установлено, то...
 if servicemanager<>0 then
  begin
   // Определяем, что нужно отображать — сервисы или драйверы
   if rbServices.Checked then
    dwType:=SERVICE_WIN32
   else
    dwType:=SERVICE_DRIVER;

   // Вызываем EnumServicesStatus с нулевыми значениями,
   // чтобы определить необходимый размер памяти
   lpResumeHandle := 0;
   EnumServicesStatus(servicemanager, dwType, SERVICE_STATE_ALL, ss[0],
      sizeof(ss), dwNeaded, dwReturn, lpResumeHandle);
```

```
// Выделяем необходимую память и вызываем EnumServicesStatus повторно
SetLength(ss, dwNeaded div SizeOf(TEnumServiceStatus));
EnumServicesStatus(servicemanager, dwType, SERVICE_STATE_ALL, ss[0],
   Length(ss) * SizeOf(TEnumServiceStatus), dwNeaded, dwReturn,
   lpResumeHandle);

// Цикл перебора всех полученных сервисов
for i := 0 to dwReturn - 1 do
 with lwServices.Items.Add do
  begin
   Caption:=ss[i].lpServiceName;
   SubItems.Add(ss[i].lpDisplayName);
   case ss[i].ServiceStatus.dwCurrentState of
    SERVICE_STOPPED:
     begin
      SubItems.Add('Остановлен');
      ImageIndex:=2;
     end;
    SERVICE_START_PENDING:
     begin
      SubItems.Add('Запускается');
      ImageIndex:=1;
     end;
    SERVICE_STOP_PENDING:
     begin
      SubItems.Add('Останавливается');
      ImageIndex:=2;
     end;
    SERVICE_RUNNING:
     begin
      SubItems.Add('Запущен');
      ImageIndex:=1;
     end;
    SERVICE_CONTINUE_PENDING :
     begin
      SubItems.Add('Продолжает выполнение');
```

```
          ImageIndex:=1;
         end;
        SERVICE_PAUSE_PENDING:
         begin
          SubItems.Add('Пауза');
          ImageIndex:=3;
         end;
        SERVICE_PAUSED:
         begin
          SubItems.Add('Пауза');
          ImageIndex:=4;
         end;
       end;
      end;
  end
 else
  begin
   Application.MessageBox('Не могу подключиться к менеджеру',
      'Error', MB_OK+MB_ICONERROR);
  end;

 lwServices.Items.EndUpdate;
end;
```

Код получился очень большой, но он очень простой. Комментарии помогут нам разобраться с тем, что здесь происходит.

Самое интересное начинается с проверки переменной `servicemanager` на ненулевое значение. Эта переменная должна быть объявлена в разделе `private` вашей формы и иметь тип `Cardinal`. Почему именно так? Во время обновления данных мы будем сохранять открытый менеджер сервисов в этой переменной и не станем его закрывать. Если теперь вы захотите добавить кнопки запуска, остановки сервиса, то по их нажатию не нужно будет создавать соединение с менеджером сервисов. Достаточно использовать переменную `servicemanager`, где уже будет корректный идентификатор. Таким образом, управление будет происходить намного быстрее.

Итак, если переменная `servicemanager` не равна нулю, то менеджер сервисов уже был открыт ранее. Закроем его, чтобы открыть заново. В прин-

ципе, этого можно было не делать, а открытие вынести в обработчик события `OnShow` и выполнять всего один раз при запуске, но я решил сделать это здесь.

Теперь можно открывать менеджер сервисов с помощью функции `OpenSCManager`. Ее мы уже видели в *разд. 3.7*.

Если соединение прошло успешно, то делаем следующую проверку: если выделен `rbServices` (компонент `RadioButton`, требующий отображения сервисов), то в переменную `dwType` записываем константу `SERVICE_WIN32`. Иначе нужно отображать драйверы, и в эту же переменную будет записана константа `SERVICE_DRIVER`.

Теперь можно начинать перечисление доступных сервисов/драйверов с помощью функции `EnumServicesStatus`. Эту функцию мы еще не рассматривали, поэтому сделаем это сейчас. В общем виде она выглядит следующим образом:

```
function EnumServicesStatus(
  hSCManager: SC_HANDLE;
  dwServiceType: DWORD;
  dwServiceState: DWORD;
  var lpServices: TEnumServiceStatus;
  cbBufSize: DWORD;
  var pcbBytesNeeded: DWORD;
  lpServicesReturned: DWORD;
  lpResumeHandle: DWORD
): BOOL; stdcall;
```

Рассмотрим параметры этой функции:

- `hSCManager` — это дескриптор открытого менеджера;
- `dwServiceType` — тип записей, которые необходимо получить. Здесь можно указать одну из двух констант:
  - `SERVICE_DRIVER` — перечислить все драйверы;
  - `SERVICE_WIN32` — перечислить сервисы Windows 32;
- `dwServiceState` — состояние сервисов, которые нужно включить в перечисление. Здесь можно указать одну из констант:
  - `SERVICE_ACTIVE` — отображать активные сервисы, т. е. все, которые имеют состояние не равное `SERVICE_STOPPED`, иными словами, не остановленные;

- SERVICE_INACTIVE — отображать неактивные сервисы, т. е. все, которые имеют состояние равное SERVICE_STOPPED, иными словами, остановленные;
- SERVICE_STATE_ALL — отображать все сервисы;

- lpServices — указатель на буфер, который содержит записи типа ENUM_SERVICE_STATUS. Через этот параметр будет возвращен результирующий список;
- cbBufSize — размер буфера в параметре lpServices в байтах;
- pcbBytesNeeded — через эту переменную мы получим количество байтов, необходимых для оставшихся записей;
- lpServicesReturned — количество возвращенных записей;
- lpResumeHandle — эта переменная определяет идентификатор записи, с которой нужно начать перечисление. Если нужно начать с самой первой, то в этом параметре следует указать 0.

Чтобы получить список сервисов, нам нужно знать, сколько этих сервисов и выделить соответствующее количество памяти. Для этого первый раз вызываем функцию, но не читаем результат. В качестве параметра cbBufSize нужно передать 0 или размер структуры, чтобы функция ничего не вернула. Через шестой параметр (pcbBytesNeeded) нам вернут количество байтов, необходимых для чтения оставшихся записей. Так как мы ничего не запросили и ничего не получили, то этот параметр вернет размер необходимого буфера для чтения всех записей. Обратите внимание, что это количество байтов, чтобы получить количество записей, нужно разделить на размер одной записи (SizeOf(TEnumServiceStatus)).

Теперь можно выделять необходимую память. Для хранения результата в разделе var мы объявили переменную ss, которая является массивом типа TEnumServiceStatus. Чтобы этот массив получил необходимый размер, пишем следующую строку:

```
SetLength(ss, dwNeaded div SizeOf(TEnumServiceStatus));
```

После второго вызова функции перечисления в массиве ss будет список всех сервисов в виде структуры _ENUM_SERVICE_STATUSA. Эта структура имеет следующий вид:

```
_ENUM_SERVICE_STATUSA = record
   lpServiceName: PAnsiChar;
   lpDisplayName: PAnsiChar;
   ServiceStatus: TServiceStatus;
end;
```

Здесь всего три параметра:

- `lpServiceName` — имя сервиса;
- `lpDisplayName` — дружественное имя, с описанием;
- `ServiceStatus` — состояние.

Первые два параметра — это строки, которые можно читать напрямую, а вот состояние — это структура, которая имеет следующий вид:

```
_SERVICE_STATUS = record
  dwServiceType: DWORD;
  dwCurrentState: DWORD;
  dwControlsAccepted: DWORD;
  dwWin32ExitCode: DWORD;
  dwServiceSpecificExitCode: DWORD;
  dwCheckPoint: DWORD;
  dwWaitHint: DWORD;
end;
```

Эта структура дает нам полную информацию о состоянии сервиса. Давайте рассмотрим ее поля.

- `dwServiceType` — тип сервиса. Этот параметр может принимать одно из следующих значений:
  - `SERVICE_FILE_SYSTEM_DRIVER` — сервис, являющийся драйвером файловой системы;
  - `SERVICE_KERNEL_DRIVER` — сервис-драйвер ядра;
  - `SERVICE_WIN32_OWN_PROCESS` — сервис, выполняемый в собственном процессе;
  - `SERVICE_WIN32_SHARE_PROCESS` — сервис, разделяющий процесс с другими процессами;
  - `SERVICE_INTERACTIVE_PROCESS` — сервис может взаимодействовать с рабочим столом.
- `dwCurrentState` — текущее состояние. В нашем примере этот параметр проверяется, чтобы узнать состояние сервиса. Параметр может принимать одно из следующих значений:
  - `SERVICE_CONTINUE_PENDING` — в ожидании продолжения выполнения;
  - `SERVICE_PAUSE_PENDING` — сервис в ожидании паузы;
  - `SERVICE_PAUSED` — приостановлен;
  - `SERVICE_RUNNING` — выполняется;

- SERVICE_START_PENDING — сервис в ожидании запуска;
- SERVICE_STOP_PENDING — в ожидании остановки;
- SERVICE_STOPPED — остановлен.

- dwControlsAccepted — управляющие коды, которые способен принимать сервис. По этому параметру можно определить, можно ли остановить или выгрузить сервис. В файле справки можно найти пять управляющих кодов, но в Windows NT поддерживаются только три из них:
  - SERVICE_ACCEPT_PAUSE_CONTINUE — разрешается приостанавливать и продолжать выполнение;
  - SERVICE_ACCEPT_SHUTDOWN — ОС посылает сервису сообщение, когда происходит завершение работы;
  - SERVICE_ACCEPT_STOP — разрешается останавливать сервис.
- dwWin32ExitCode — код ошибки, который сообщает сервис, если во время запуска или остановки произошла ошибка.
- dwServiceSpecificExitCode — код ошибки, определяемый сервисом.
- dwCheckPoint — значение контрольной точки, которое периодически увеличивается, для определения процесса запуска, остановки, паузы или восстановления выполнения.
- dwWaitHint — приблизительное время для запуска, остановки, паузы или восстановления выполнения.

Знание этих структур позволит не только понять, что происходит в листинге 3.12, но и улучшить пример до полноценной оснастки управления сервисами.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 3\ServiceControl\ вы можете найти пример этой программы.

3.9. Управление пользователями

Управление пользователями в ОС Windows — не очень сложная тема, но все же, она весьма интересна, особенно для программистов Delphi. Дело в том, что в заголовочных файлах, поставляемых с Delphi, эти функции не прописаны, поэтому придется создавать заголовочный файл самостоятельно. Можно найти что-то готовое в Интернете, но, как показывает мой опыт, далеко не каждая портация Windows API работает корректно. Поэтому я

стараюсь описывать необходимые мне функции самостоятельно, тем более что это не сложно.

Весь заголовочный файл с необходимыми для управления пользователями функциями я не буду приводить на страницах книги, дабы не тратить лишнее место. Файл есть на компакт-диске, и если вы легально купили книгу в магазине, то компакт-диск обязан быть. Я буду приводить общий вид только тех функций, которые будут использоваться в проекте.

Итак, на компакт-диске есть файл netapi.pas, который необходимо прописать в разделе `uses` вашего проекта, а сам файл нужно поместить так, чтобы компилятор мог его найти, например, в каталог с файлами проекта. Да, файл содержит далеко не все функции, но основные есть. Дело в том, что я в него записал то, с чем самому приходилось работать.

Все функции, которые мы будем рассматривать в этой главе, относятся к Network Management SDK. Именно к этому файлу справки следует обращаться за дополнительной информацией. Если у вас нет проблем с пониманием C++, то описание всех функций можно найти в lm.h и lmaccess.h.

3.9.1. Получение списка пользователей

Начнем рассмотрение функций NetAPI с определения списка пользователей и групп в вашей системе. Вот тут необходимо заметить, что если вы используете версию Professional или Server, то у вас через **Панель управления | Администрирование** будет доступна оснастка **Управление компьютером**, в которой есть удобные средства управления пользователями. В Windows Home Edition программа управления намного проще и расположена она в Панели управления, а называется **Учетные записи пользователей** (рис. 3.18).

Да, управлять в Windows XP Home Edition стандартными средствами практически нечем, но сегодня мы напишем собственную утилиту, которая будет одинаково работать как в версии Professional, так и в Home Edition.

Итак, нам понадобится форма с кнопкой и компонентом `ListView`. Установим у `ListView` в свойстве `ViewStyle` значение `vsReport`. Ну что поделаешь, если я люблю этот компонент для отображения сеток. Теперь дважды щелкаем по `ListView` и в появившемся окне редактора создаем шесть колонок: тип, имя пользователя, полное имя, комментарий, привилегии и примечание (рис. 3.19).

По нажатию кнопки, которую мы разместили на форме, будем обновлять список пользователя. По событию `OnClick` напишите код, показанный в листинге 3.13.

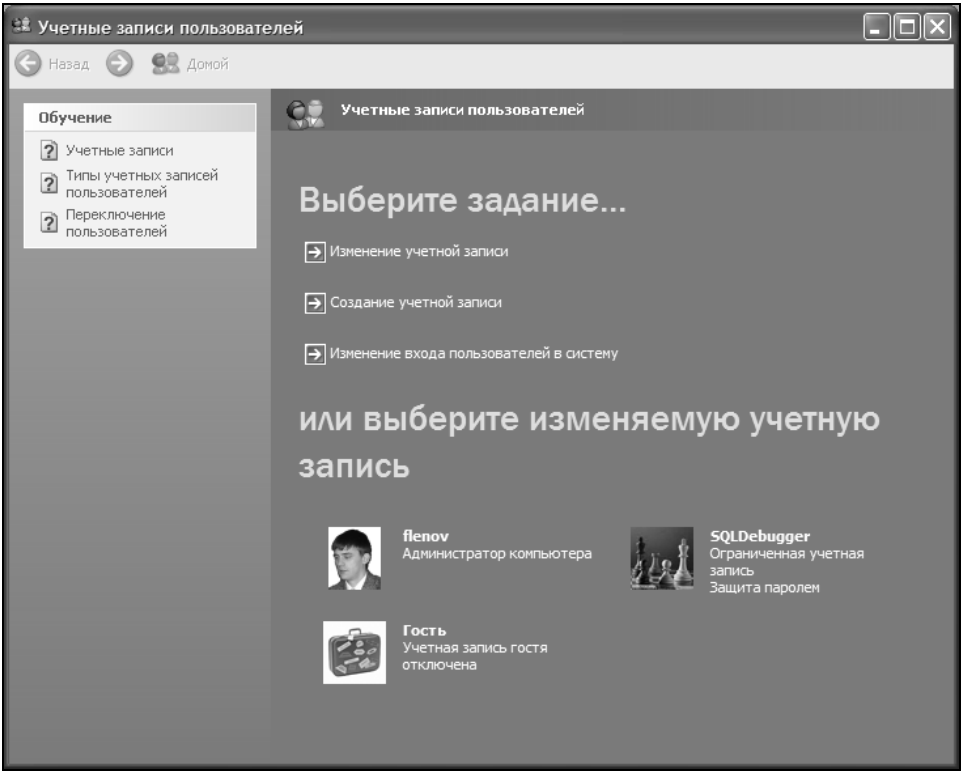


Рис. 3.18. Окно управления пользователями в Windows XP Home Edition

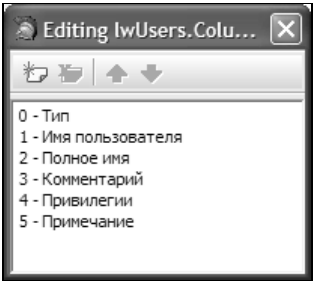


Рис. 3.19. Названия колонок

Листинг 3.13. Получение списка пользователей

```
var
 NetApiStatus: NET_API_STATUS;
```

```
 lpBuffer: Pointer;
 i, dwEntiesRead, dwTotalEntries, dwResumeHandle: DWORD;
 UserInfo: PUserInfo3;
 GroupInfo: PGroupInfo1;
begin
 lwUsers.Items.Clear;

 // Получить пользователей
 dwResumeHandle := 0;
 repeat
  NetApiStatus := NetUserEnum('', 3, 0, lpBuffer,
      MAX_PREFERRED_LENGTH, dwEntiesRead,
      dwTotalEntries, dwResumeHandle);

  UserInfo := lpBuffer;

  // Цикл перебора всех пользователей
  for i := 1 to dwEntiesRead do
   begin
    with lwUsers.Items.Add do
     begin
      Caption:='Юзер';
      SubItems.Add(WideCharToString(UserInfo^.usri3_name));
      SubItems.Add(WideCharToString(UserInfo^.usri3_full_name));
      SubItems.Add(WideCharToString(UserInfo^.usri3_comment));
      case UserInfo^.usri3_priv of
        USER_PRIV_GUEST: SubItems.Add('Гость');
        USER_PRIV_USER: SubItems.Add('Юзер');
        USER_PRIV_ADMIN: SubItems.Add('Админ');
      end;
     end;
    Inc(UserInfo);
   end;
  // Если буфер не нулевой, то освобождаем его
  if lpBuffer<>nil then
```

```
   NetApiBufferFree(lpBuffer);
 until (NetApiStatus <> ERROR_MORE_DATA);

 // Получаем группы
 dwResumeHandle := 0;
 repeat
  NetApiStatus := NetLocalGroupEnum('', 1, lpBuffer,
       MAX_PREFERRED_LENGTH, dwEntiesRead,
       dwTotalEntries, @dwResumeHandle);

  GroupInfo := lpBuffer;

  // Цикл перебора всех групп
  for i := 1 to dwEntiesRead do
   begin
    with lwUsers.Items.Add do
     begin
      Caption:='Группа';
      SubItems.Add(WideCharToString(GroupInfo^.grpi1_name));
      SubItems.Add('');
      SubItems.Add(WideCharToString(GroupInfo^.grpi1_comment));
     end;
    Inc(GroupInfo);
   end;
  // Если буфер не нулевой, то освобождаем его
  if lpBuffer<>nil then
   NetApiBufferFree(lpBuffer);
 until (NetApiStatus <> ERROR_MORE_DATA);
end;
```

Сразу же покажу результат, который вы можете увидеть на рис. 3.20. Этот пример я выполнял на своем компьютере с Windows XP Home Edition, и в стандартной программе управления пользователями было видно всего трех пользователей (вспомните рис. 3.18), а тут их намного больше, а также есть еще и группы. Оказывается, даже в Home Edition есть намного больше возможностей, просто одни из них отключены, а другие прячутся от нас.

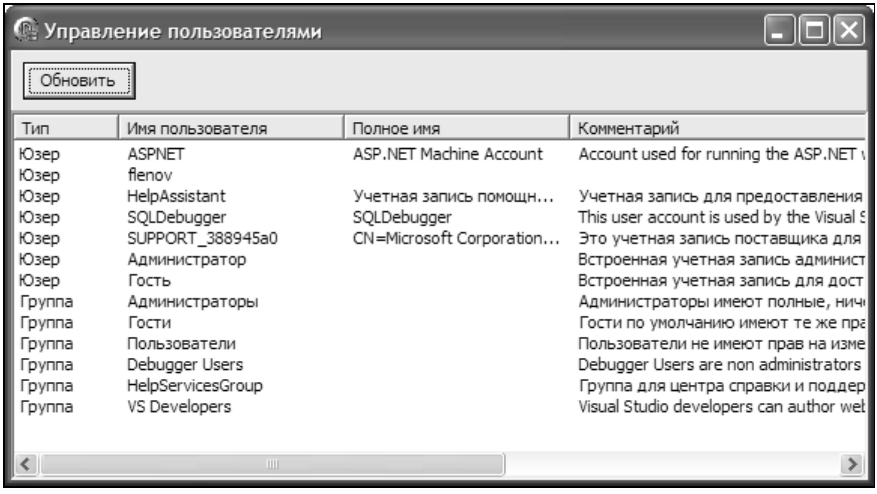


Рис. 3.20. Результат работы в Windows XP Home Edition

Теперь вернемся к листингу 3.13 и посмотрим, что в нем происходит. Для получения списка пользователей применяется функция `NetUserEnum`. В общем виде она выглядит следующим образом:

```
function NetUserEnum (
  ServerName: PWideChar;
  Level,
  Filter: DWORD;
  var Buffer: Pointer;
  PrefMaxLen: DWORD;
  var EntriesRead,
  TotalEntries,
  ResumeHandle: DWORD
): Longword; stdcall;
```

Тут у нас 8 параметров. Давайте рассмотрим, для чего они нужны.

- `ServerName` — строка, в которой можно указать NetBIOS- или DNS-имя компьютера, пользователей которого нужно определить. Да, если у вас есть администраторские права на чужом компьютере, то вы можете управлять и его пользователями. Если указать здесь нулевое значение, то будет использоваться локальный компьютер.
- `Level` — число, которое определяет уровень получаемой информации. В зависимости от уровня нам будут возвращаться различные структуры с

разной детализацией информации. Если указать 0, то в результате мы узнаем только имена пользователей. Все структуры мы рассматривать не станем, но я буду выбирать уровни, которые будут возвращать максимально полную информацию. В данном случае, мы будем использовать третий уровень. Какая будет в результате информация, вы узнаете чуть позже.

- `Filter` — через этот параметр можно отфильтровать записи и получить имена пользователей только определенного типа. В нашем примере фильтр равен нулю, а значит, будут отображаться все записи. Но вы можете указать одно из следующих значений:
  - `FILTER_TEMP_DUPLICATE_ACCOUNT` — показывать все учетные записи локальных пользователей на контроллере домена;
  - `FILTER_NORMAL_ACCOUNT` — перечислить всех глобальных пользователей на компьютере;
  - `FILTER_INTERDOMAIN_TRUST_ACCOUNT` — отображать доверительные записи домена на контроллере.
- `Buffer` — это указатель на буфер, через который нам вернут результат работы.
- `PrefMaxLen` — определяет максимальную длину возвращаемых данных. Если указать константу `MAX_PREFERRED_LENGTH`, то будет выделена необходимая память для хранения всех данных. Эта константа равна −1.
- `EntriesRead` — через этот параметр мы получим количество реально прочитанных записей.
- `TotalEntries` — количество записей, которые могут быть прочитаны, начиная с текущего значения `ResumeHandle`. Если в параметре `ResumeHandle` указан 0, то здесь будет общее количество записей.
- `ResumeHandle` — этот параметр используется для чтения данных по частям. Укажите дескриптор, начиная с которого нужно перечислять записи, или 0, чтобы читать с самого начала.

Тут нужно сделать еще небольшое замечание — память для буфера из четвертого параметра (через который мы получаем результат) выделяется самой функцией, но освобождать мы должны ее сами. После работы с буфером удалите его с помощью функции `NetApiBufferFree`. Эта функция получает только один параметр — указатель на буфер.

Как я уже сказал, для различных уровней детализации существуют разные структуры данных для результата. Для третьего уровня через параметр `bufptr` мы получим массив из структур типа `USER_INFO_3`. В своем заголовочном файле я дал ей (структуре) псевдоним `TUserInfo3`, чтобы именование было в стиле Delphi. Как я уже сказал, я выбрал максимально большую

структуру, поэтому ради экономии места не буду ее приводить здесь, но параметры лучше рассмотреть, потому что тут много интересного. К тому же эта структура может использоваться и в других функциях, не только при перечислении.

Итак, структура состоит из следующих полей:

- `usri3_name` — это Unicode-строка, которая содержит имя пользователя. Этот параметр игнорируется, когда выполняется изменение параметров пользователя, ведь при изменении нельзя менять имя;
- `usri3_password` — этот параметр содержит Unicode-строку с паролем. Только не стоит думать, что вы можете увидеть пароль. Это нереально. Параметр применяется только в функциях создания или изменения параметров пользователя. При чтении и перечислении здесь будет нулевое значение;
- `usri3_password_age` — если говорить прямо, то это возраст пароля, а если точнее, то количество секунд, прошедших с момента последней смены пароля. Этот параметр только для чтения, поэтому он игнорируется при добавлении или редактировании пользователя;
- `usri3_priv` — привилегии. Этот параметр может содержать одно из следующих значений:
  - `USER_PRIV_GUEST` — гостевая запись;
  - `USER_PRIV_USER` — непривилегированный пользователь;
  - `USER_PRIV_ADMIN` — администратор;
- `usri3_home_dir` — строка, определяющая путь к домашнему каталогу;
- `usri3_comment` — комментарий в виде Unicode-строки, который связан с учетной записью пользователя;
- `usri3_flags` — флаги. Флагов очень много, и рассматривать все я не буду, но основные опишу:
  - `UF_ACCOUNTDISABLE` — учетная запись отключена;
  - `UF_PASSWD_NOTREQD` — пароль не требуется;
  - `UF_PASSWD_CANT_CHANGE` — пользователь может менять пароль;
  - `UF_DONT_EXPIRE_PASSWD` — время пароля никогда не истекает;
  - `UF_PASSWORD_EXPIRED` — время действия пароля истекло;
- `usri3_script_path` — путь к файлу со сценарием, который должен выполняться во время входа в систему;
- `usri3_full_name` — полное имя пользователя;
- `usri3_usr_comment` — строка комментария;

- usri3_parms — параметры пользователя. Судя по файлу помощи, программы могут использовать эту информацию по своему усмотрению, но тут же дается предупреждение, что изменять этот параметр нежелательно, потому что ОС хранит здесь информацию о конфигурации;
- usri3_workstations — строка, содержащая имена рабочих станций, с которых пользователь может входить в систему;
- usri3_last_logon — время последнего входа в систему;
- usri3_last_logoff — по идее, это должно быть время последнего выхода, но MSDN говорит, что этот параметр не используется;
- usri3_acct_expires — определяет дату и время, когда устареет пароль;
- usri3_max_storage — максимальный размер дискового пространства, разрешенный для использования. Если указать USER_MAXSTORAGE_UNLIMITED, то можно использовать любое количество дискового пространства;
- usri3_logon_hours — строка из 168 битов (21 байт), с помощью которой можно определить время, когда пользователь может входить в систему;
- usri3_bad_pw_count — сколько раз можно указать неправильный пароль. Если указать единицу, то количество неопределенно;
- usri3_num_logons — сколько раз пользователь удачно входил в систему;
- usri3_logon_server — имя сервера, которому нужно отправлять запрос на вход;
- usri3_country_code и usri3_code_page — код страны и кодовая страница;
- usri3_user_id — идентификатор пользователя;
- usri3_primary_group_id — идентификатор первичной группы;
- usri3_profile — путь к пользовательскому профилю;
- usri3_home_dir_drive — строка, содержащая букву диска, с пользовательским каталогом.

Итак, следующая строка возвращает нам список всех пользователей в системе:

```
NetApiStatus := NetUserEnum('', 3, 0, lpBuffer, MAX_PREFERRED_LENGTH,
   dwEntiesRead, dwTotalEntries, dwResumeHandle);
```

После выполнения функции NetUserEnum в переменной lpBuffer будет находиться указатель на память, где расположен массив из структур PUserInfo3. Количество записей в этом массиве можно определить по содержимому переменной dwEntiesRead. Вся необходимая информация получена, а значит, можно запускать цикл и выводить информацию о пользователях в ListView.

Теперь поговорим о получении списка групп. Принцип работы аналогичен, только необходимо использовать функцию `NetLocalGroupEnum`.

```
function NetLocalGroupEnum(
 servername: LPCWSTR;
 level: DWORD;
 var bufptr: Pointer;
 prefmaxlen: DWORD;
 var entriesread: DWORD;
 var totalentries: DWORD;
 resumehandle: PDWORD
): NET_API_STATUS; stdcall;
```

Даже без увеличительного стекла видно, что эта функция очень похожа на `NetUserEnum`. По крайней мере, параметры имеют тот же смысл. Разница только во втором и третьем параметрах, и то, эта разница чисто идеологическая. Дело в том, что во втором параметре мы так же передаем уровень детализации информации, просто эти уровни немного другие и количество вариантов (если моя память не изменяет мне так же, как и моя жена, а жена мне вроде бы не изменяет) отличается.

Итак, для групп существуют всего два уровня детализации. На нулевом нам вернут массив структур, в которых будет только название группы, а на первом уровне нам вернут и название группы, и комментарий. Мы будем использовать уровень 1, поэтому в качестве третьего параметра получим массив из структур `TGroupInfo1`, а эта структура выглядит так:

```
PGroupInfo1 = ^TGroupInfo1;
_GROUP_INFO_1 = record
 grpi1_name: LPWSTR;
 grpi1_comment: LPWSTR;
end;
TGroupInfo1 = _GROUP_INFO_1;
```

Как видите, у структуры всего два члена:

- `grpi1_name` — название группы;
- `grpi1_comment` — комментарий.

Чтобы превратить эту структуру в формат, соответствующий уровню 0, достаточно удалить параметр `grpi1_comment`.

На этом описание новых функций для данного примера закончено. Больше ничего нового добавить не могу.

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\Users1\ вы можете найти пример этой программы.

3.9.2. Управление пользователями и группами

Одним из замечаний к первому изданию этой книги было то, что я расписывал все темы полностью и оставлял готовые примеры. Очень много было просьб оставлять какие-то задания. С одной стороны, эта книга не задачник, но с другой стороны, домашние задания могут положительно сказаться с точки зрения закрепления материала. В этом разделе мы рассмотрим функции работы с учетными записями пользователей.

Чтобы получить информацию о пользователе, можно использовать функцию перечисления, как было показано в *разд. 3.9.1*, но лучше воспользоваться функцией `NetUserGetInfo`. Эта функция возвращает информацию только об определенном пользователе, что намного экономнее и эффективнее.

В общем виде функция `NetUserGetInfo` выглядит следующим образом:

```
function NetUserGetInfo(
 servername: LPCWSTR;
 username: LPCWSTR;
 level: DWORD;
 var bufptr: Pointer
): NET_API_STATUS; stdcall;
```

Здесь у нас всего четыре параметра:

- `servername` — имя компьютера, учетную запись пользователя которого нужно прочитать. Если указать здесь 0, то система будет искать пользователя на локальном компьютере;
- `username` — имя пользователя, данные которого нужно вернуть;
- `level` — уровень детализации информации. Здесь можно указывать те же номера уровней, что и для функции `NetUserEnum`;
- `bufptr` — указатель, по адресу которого будет возвращена структура с данными о пользователе. Вид структуры зависит от запрошенного уровня, и если задать в параметре `level` число 3, то указатель данного параметра будет указывать на структуру типа `PUserInfo3`.

Получив структуру с данными от пользователя, можно отобразить информацию в каком-либо окне для редактирования. Но после редактирования необходимо же сохранить эту информацию обратно в системе, но как это сделать? Для этого можно использовать функцию `NetUserSetInfo`:

```
function NetUserSetInfo (
 servername: LPCWSTR;
 username: LPCWSTR;
 level: DWORD;
 buf: Pointer;
 parm_err: PDWORD
): NET_API_STATUS; stdcall;
```

Функция схожа с чтением информации, только добавлен один параметр `parm_err`, в котором в случае ошибки будет находиться индекс параметра из структуры пользовательских данных (`PUserInfo`), который привел к ошибке.

Помните, что далеко не все параметры можно редактировать. Некоторые данные чисто информационные, и их лучше не изменять.

Теперь перейдем к добавлению нового пользователя. Для этого нехитрого действия используется функция `NetUserAdd`, которая выглядит следующим образом:

```
function NetUserAdd(
 servername: LPCWSTR;
 level: DWORD;
 buf: Pointer;
 parm_err: PDWORD
): NET_API_STATUS; stdcall;
```

По именам параметров можно уже догадаться, для чего они нужны, потому что подобные параметры мы уже видели. Но на всякий случай коротко по ним пробежимся:

- `servername` — имя компьютера;
- `level` — уровень, в зависимости от которого система определяет, какую структуру мы ей передали и сколько информации о пользователе предоставили;
- `buf` — указатель на структуру `PUserInfo`. Для третьего уровня здесь нужно указать структуру `PUserInfo3`, которую мы рассматривали в *разд. 3.9.1*;
- `parm_err` — в случае ошибки, через этот параметр нам вернут индекс поля в структуре данных о пользователе.

Для удаления пользователя служит функция NetUserDel, которая выглядит следующим образом:

```
function NetUserDel (
 servername: LPCWSTR;
 username: LPCWSTR
): NET_API_STATUS; stdcall;
```

Тут всего два параметра — имя сервера и имя пользователя, которого нужно удалить на этом сервере.

Для управления группами существуют подобные функции с похожими именами, параметрами, но только влияют они на группы:

- NetLocalGroupGetInfo — возвращает информацию о локальной группе;
- NetLocalGroupSetInfo — назначает новые параметры указанной группе;
- NetLocalGroupAdd — добавляет новую группу;
- NetLocalGroupDel — удаляет группу.

На этом мы пока завершим рассмотрение функций Network Management SDK. Чтобы рассмотреть все возможности этого пакета разработчика, понадобится отдельная книга. К тому же большинство из этих функций выходит за рамки темы данной главы. Мы рассматриваем здесь систему, а не сеть. Управление пользователями и группами можно отнести к системе, поэтому мы рассмотрели их здесь. Но в дальнейшем мы еще не раз будем обращаться к этому SDK, но уже к другим функциям, которые работают с сетями.

3.10. Изменение параметров окна

Когда запускается программа, то на панели задач отображается только главное окно программы. Все дочерние окна там отображаться не будут. А ведь бывают такие случаи, когда просто необходимо иметь простой доступ к дочернему окну, чтобы пользователь мог с помощью панели задач выбирать между главным окном или дочерним. Несмотря на то, что подобных свойств у формы в Delphi нет, мы можем повлиять на ход событий, и в этом разделе я покажу, как это сделать.

Создайте новый проект и поместите на главную форму только одну кнопку. Теперь добавьте к проекту еще одну форму. При нажатии кнопки на главной форме мы будем отображать дочернее окно:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 Form2.Show;
end;
```

Окно отображается методом `Show`, чтобы оно было немодальным, и можно было переключаться между окнами. Таким образом, на экране будут находиться оба окна, и с обоими можно будет работать.

Теперь, чтобы второе окно появилось в панели задач, нужно создать для него обработчик события `OnCreate`. В нем мы напишем следующее:

```
procedure TForm2.FormCreate(Sender: TObject);
begin
 SetWindowLong(Handle, GWL_EXSTYLE,
      GetWindowLong(Handle, GWL_EXSTYLE) or WS_EX_APPWINDOW);
end;
```

Фокус достаточно прост. Здесь вызывается WinAPI-функция `SetWindowLong`, которая может изменять параметры окна. В инспекторе объектов нам доступны не все, а с помощью этой функции мы можем изменить любые свойства, какие есть. У функции имеются три параметра;

- указатель на окно, параметры которого надо изменить — указано текущее окно, т. е. наше дочернее окно;
- что мы хотим изменить — здесь можно указать несколько значений (их вы можете увидеть в файле справки по WinAPI, если запустите поиск слова "SetWindowLong"). Но реально вам могут пригодиться только два из доступных параметров:
  - `GWL_STYLE` — изменить стандартный стиль окна. Стандартные стили вы можете увидеть в файле справки по WinAPI, если запустите поиск слова `CreateWindow`. Такие стили начинаются с префикса `WS_`. Эти изменения можно сделать и в Инспекторе объектов с помощью свойств формы;
  - `GWL_EXSTYLE` — изменить расширенный стиль окна. Здесь находятся расширенные стили, которых нет в Инспекторе объектов. Расширенные стили вы можете увидеть в файле справки по WinAPI, если запустите поиск слова "CreateWindowEx". Такие стили начинаются с префикса `WS_EX_`;
- новое значение изменяемого параметра.

В нашем примере мы указали в качестве второго параметра значение `GWL_EXSTYLE`. Это значит, что мы хотим изменить расширенный стиль окна. В качестве третьего параметра вызывается функция `GetWindowLong`. Она возвращает существующие параметры окна. Данная функция имеет два параметра:

- указатель на окно;
- параметр, который мы хотим получить. Указан `GWL_EXSTYLE`, чтобы получить установленные расширенные стили окна.

В результате, в качестве третьего параметра мы получаем текущие значения расширенного стиля и добавляем к нему стиль `WS_EX_APPWINDOW`, что заставит окно появиться на панели задач.

Вот и все. Вот такой небольшой трюк дает нам достаточно сильные возможности в управлении окнами нашей программы. На рис. 3.21 вы можете увидеть пример работы моей программы.

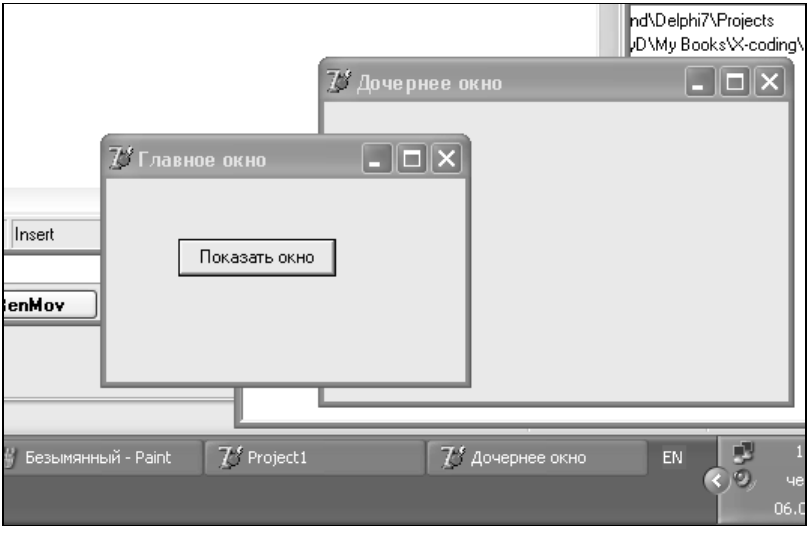


Рис. 3.21. Главное и дочернее окна на панели задач

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\ChildWindow\ вы можете увидеть пример этой программы и цветную версию рисунка.

3.11. Создание ярлыков

Сейчас мы напишем пример программы, которая будет размещать на рабочем столе и в меню **Пуск | Все программы** свой ярлык. Вообще-то этот пример можно было бы описать в *главе 2*, где рассматривались простые шутки, но пример может пригодиться не только в качестве шуток, чаще всего вы будете использовать при написании простых утилит или полноценных программ. Именно поэтому мне приходится описывать пример только сейчас.

Наш пример будет размещать только один ярлык в меню **Пуск** и на рабочем столе. Вы же можете немного модифицировать пример, и после этого ваша программа сможет засыпать хоть весь рабочий стол разными яркими ярлыками.

Итак, создаем новый проект в Delphi. Сразу же перейдем в раздел `uses` и добавим туда следующие модули: `ShlObj`, `ActiveX` и `ComObj`. Теперь разместим на форме три кнопки с надписями:

- **Создать ярлык в меню "Программы"**;
- **Создать ярлык на рабочем столе**;
- **Засыпать экран**.

Мой вариант формы программы вы можете увидеть на рис. 3.22. Я думаю, что смысл кнопок понятен, и мне больше нечего сказать, поэтому сразу и без лишних разговоров перейдем к программированию.

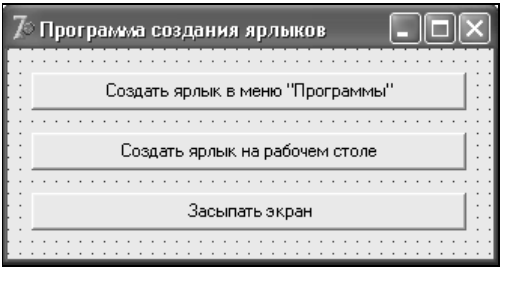


Рис. 3.22. Форма программы

В обработчике `OnClick` первой кнопки **Создать ярлык в меню "Программы"** пишем код, показанный в листинге 3.14.

Листинг 3.14. Размещение ярлыка в одном из меню кнопки *Пуск*

```
procedure TForm1.Button1Click(Sender: TObject);
var
 WorkTable:String;
 P:PItemIDList;
 C:array [0..1000] of char;
begin
 if SHGetSpecialFolderLocation(Handle,CSIDL_PROGRAMS,p)=NOERROR then
  begin
   SHGetPathFromIDList(P,C);
```

```
   WorkTable:=StrPas(C)+'\My Group';
  end;

 if not DirectoryExists(WorkTable) then
  MkDir(WorkTable);

 if FileExists(WorkTable+'\'+ExtractFileName(Application.ExeName)) then
  DeleteFile(WorkTable+'\'+ExtractFileName(Application.ExeName));

 CreateShotCut(Application.ExeName, WorkTable + '\'+
     ExtractFileName(Application.ExeName), '');
end;
```

В самом начале определяется место расположения папки Программы, которая отвечает за содержание одноименного меню кнопки **Пуск**. Да, это самая настоящая папка на диске, которая в Windows 9*x* по умолчанию располагалась по следующему пути: C:\Windows\Главное меню\Программы. В Windows 2000/XP расположение более сложное, и тут уже ее надо искать в папке Documents and Settings. Все, что находится в этой папке (файлы ярлыков, программы и любые другие файлы), отображается в главном меню.

Итак, чтобы разместить в меню **Все программы** свой объект, мы можем просто скопировать нужный файл в найденную папку. Копировать сами исполняемые файлы в папку Программы неэтично. Там принято создавать только файлы ярлыков, которые занимают очень мало места, потому что только ссылаются на реальную программу.

Кстати, когда мы определили путь к папке, содержащей объекты меню **Все программы**, мы добавили к этому пути \My Group. Таким образом, мы переместились в подменю **My Group** меню **Все программы**. Но прежде чем туда переместиться, необходимо проверить, существует ли такой подкаталог? Для этого проверяем, если нужный каталог не существует, то его следует создать с помощью функции `MkDir`:

```
if not DirectoryExists(WorkTable) then
  MkDir(WorkTable);
```

Следующим этапом проверяем, существует ли ярлык для программы. Если да, то удаляем файл ярлыка, потому что в нем могут находиться устаревшие данные, и лучше создать все заново.

Само создание происходит в последней строке с помощью вызова процедуры `CreateShotCut`.

У этой процедуры имеются три параметра:

- файл, запускаемый ярлыком;
- имя, которое будет отображаться на ярлыке;
- параметры, которые должны быть переданы программе при запуске.

Если вы сейчас попытаетесь запустить программу, то Delphi не сможет откомпилировать код, потому что не знает о существовании процедуры `CreateShotCut`. Ее нужно еще написать. Для этого в разделе `private` опишите процедуру `CreateShotCut` следующим образом:

```
private
 { Private declarations }
 procedure CreateShotCut(SourceFile, ShortCutName,
    SourceParams: String);
```

Теперь нажимаем заветную комбинацию клавиш <Ctrl>+<Shift>+<C> и получаем заготовку описанной процедуры. В нее нужно вставить код из листинга 3.15.

Листинг 3.15. Процедура создания ярлыка

```
procedure TForm1.CreateShotCut(SourceFile, ShortCutName,
  SourceParams: String);
var
 IUnk: IUnknown;
 ShellLink: IShellLink;
 ShellFile: IPersistFile;
 tmpShortCutName: string;
 WideStr: WideString;
 i: Integer;
begin
 IUnk := CreateComObject(CLSID_ShellLink);
 ShellLink := IUnk as IShellLink;
 ShellFile  := IUnk as IPersistFile;

 ShellLink.SetPath(PChar(SourceFile));
 ShellLink.SetArguments(PChar(SourceParams));
 ShellLink.SetWorkingDirectory(PChar(ExtractFilePath(SourceFile)));
```

```
 ShortCutName := ChangeFileExt(ShortCutName,'.lnk');

 if fileexists(ShortCutName) then
  begin
   ShortCutName := copy(ShortCutName,1,length(ShortCutName)-4);
   i := 1;
   repeat
    tmpShortCutName := ShortCutName +'(' + inttostr(i)+ ').lnk';
    inc(i);
   until not fileexists(tmpShortCutName);
   WideStr := tmpShortCutName;
 end
  else
   WideStr := ShortCutName;

 ShellFile.Save(PWChar(WideStr),False);
end;
```

В самом начале мы инициализируем переменную `IUnk`, как COM-объект с помощью API-функции `CreateComObject`. Затем инициализируются еще две переменные: `ShellLink` (ссылка) и `ShellFile` (файл). После этого вызываются следующие методы объекта ссылки `ShellLink`:

- `SetPath` — устанавливает полный путь к программе;
- `SetArguments` — устанавливает параметры, которые надо передать программе;
- `SetWorkingDirectory` — здесь указывается рабочий каталог.

Помимо этого у объекта-ссылки есть еще методы:

- `GetDescription` — указывает в ярлыке описание для программы;
- `SetShowCmd` — определяет режим отображения окна. Здесь можно использовать режимы, которые мы указывали в API-функции `ShowWindow`, например `SW_HIDE` (запускать и скрывать), `SW_MAXIMIZE` (запускать в окне, развернутом на весь экран), `SW_MINIMIZE` (минимизировать окно после старта) и т. д.

После указания необходимых параметров в переменной `ShortCutName` сохраняется имя ярлыка с добавлением расширения lnk. Это имя будет использоваться при создании самого файла ссылки. Далее проверяется, если такой ярлык уже существует, то запускается цикл, в котором к имени ссыл-

ки добавляется цифра. Таким образом находится новое имя ярлыка с цифрой, которого еще не существует в указанном месте.

В самой последней строке созданная ссылка сохраняется в файле ярлыка.

Как видите, процедура универсальна и готова для заполнения экрана или меню **Все программы** своими копиями. Вам нужно только изменить уже имеющийся обработчик события нажатия кнопки, что мы и сделаем чуть позже.

Сейчас мы создадим обработчик события нажатия второй кнопки, и я покажу, как создать ярлык на рабочем столе (листинг 3.16).

Листинг 3.16. Размещение ярлыка на рабочем столе

```
procedure TForm1.Button2Click(Sender: TObject);
var
 WorkTable:String;
 P:PItemIDList;
 C:array [0..1000] of char;
begin
 if SHGetSpecialFolderLocation(Handle,CSIDL_DESKTOP,p)=NOERROR then
  begin
   SHGetPathFromIDList(P,C);
   WorkTable:=StrPas(C);
  end;

 if FileExists(WorkTable+'\'+ExtractFileName(Application.ExeName)) then
  DeleteFile(WorkTable+'\'+ExtractFileName(Application.ExeName));

 CreateShotCut(Application.ExeName, WorkTable + '\' +
     ExtractFileName(Application.ExeName), '');
end;
```

Этот код похож на тот, что мы писали при создании ярлыка в меню **Все программы**. Единственная разница в том, что здесь с помощью функции `SHGetSpecialFolderLocation` мы узнаем расположение папки для ярлыков и программ рабочего стола (второй параметр равен `CSIDL_DESKTOP`).

Вот теперь рассмотрим код заполнения экрана ярлыками. Он выглядит так, как представлено в листинге 3.17.

Листинг 3.17. Заполнение рабочего стола ярлыками

```
var
 WorkTable:String;
 P:PItemIDList;
 C:array [0..1000] of char;
 i:Integer;
begin
 if SHGetSpecialFolderLocation(Handle,CSIDL_DESKTOP,p)=NOERROR then
  begin
   SHGetPathFromIDList(P,C);
   WorkTable:=StrPas(C);
  end;

 for i:=0 to 20 do
  CreateShotCut(Application.ExeName, WorkTable + '\' +
        ExtractFileName(Application.ExeName), '');
end;
```

Рис. 3.23. В результате работы программы рабочий стол превращается в свалку

Здесь нет проверки на существование ярлыка. Если он уже существует (а он с некоторого момента будет на столе не один), то наша процедура `CreateShotCut` изменит имя и создаст еще один ярлык. Вызов процедуры создания ярлыка помещен в цикл так, чтобы процедура создавала 20 ярлыков на рабочем столе. На рис. 3.23 вы можете увидеть мой рабочий стол после выполнения этого кода.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 3\Ярлык\ вы можете увидеть пример этой программы и цветные версии рисунков.

3.12. Управление ярлыками

Сейчас мы разберем небольшой пример кода, который может упорядочивать значки на рабочем столе. Для этого я создал отдельное приложение и положил на форму две кнопки:

- **Упорядочить влево**;
- **Упорядочить по верху**.

В обработчике нажатия первой кнопки напишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 DesktopHandle:Integer;
begin
 DesktopHandle := FindWindow('ProgMan', nil);
 DesktopHandle := GetWindow(DesktopHandle, GW_CHILD);
 DesktopHandle := GetWindow(DesktopHandle, GW_CHILD);
 SendMessage(DesktopHandle, LVM_ARRANGE, LVA_ALIGNLEFT, 0 );
end;
```

Здесь ищется окно с заголовком `ProgMan`. Хотя такое окно не видно, но оно существует при работе Windows, начиная с третьей версии (а может и раньше), и называется **Program Manager**. В этом окне мы ищем дочернее окно с помощью функции `GetWindow`. И затем получаем следующее дочернее окно. Вот теперь мы получили указатель на системный объект класса `SysListView32`. Этот компонент как раз и содержит все значки рабочего стола.

Теперь мы можем посылать сообщения, совместимые с компонентом `ListView`, системному хранилищу значков с помощью функции `SendMessage`.

В качестве примера отсылается сообщение с двумя параметрами:

- LVM_ARRANGE — указывает на необходимость отсортировать значки;
- LVA_ALIGNLEFT — упорядочить значки по левому краю.

Если второй из этих параметров заменить на LVA_ALIGNTOP, то значки будут выровнены по верхнему краю окна.

Давайте создадим на нашей форме кнопку **Удалить все элементы**. В обработчике события OnClick этой кнопки напишем следующий код:

```
procedure TForm1.Button3Click(Sender: TObject);
var
 DesktopHandle:Integer;
begin
 DesktopHandle := FindWindow('ProgMan', nil);
 DesktopHandle := GetWindow(DesktopHandle, GW_CHILD);
 DesktopHandle := GetWindow(DesktopHandle, GW_CHILD);
 SendMessage(DesktopHandle, LVM_DELETEALLITEMS, 0, 0 );
end;
```

Код похож на тот, что мы уже использовали. Только здесь мы посылаем команду LVM_DELETEALLITEMS, которая заставляет удалить все элементы. Попробуйте выполнить эту команду, и весь рабочий стол станет девственно чистым, как будто на нем ничего никогда не было.

Ну а теперь самое интересное — перемещение ярлыков по экрану. Для реализации этого эффекта поместим на форму еще одну кнопку с заголовком **Переместить** (рис. 3.24). В обработчике события OnClick этой кнопки вставим следующий код:

```
procedure TForm1.Button4Click(Sender: TObject);
var
 DesktopHandle:Integer;
begin
 DesktopHandle := FindWindow('ProgMan', nil);
 DesktopHandle := GetWindow(DesktopHandle, GW_CHILD);
 DesktopHandle := GetWindow(DesktopHandle, GW_CHILD);
 SendMessage(DesktopHandle, LVM_SETITEMPOSITION, 0, MAKELPARAM(10, 100));
end;
```

Здесь посылается сообщение SendMessage со следующими параметрами:

- указатель на окно, содержащее элементы. Мы этот указатель нашли в начале процедуры;

- `LVM_SETITEMPOSITION` — константа, которая указывает на необходимость изменить позицию ярлыка;
- `0` — надо переместить нулевой ярлык, но вы можете переместить и первый, и второй, в общем, любой ярлык;
- новые координаты ярлыка. Этот параметр нужно создать с помощью функции `MAKELPARAM`, которой в данном случае передаются два параметра: `X` и `Y` — координаты ярлыка.

Рис. 3.24. Результирующая форма нашей программы

И чтобы завершить разработку, поместим на форму еще одну кнопку — **Анимация**. Нажимая эту кнопку, мы постепенно будем двигать нулевой значок по экрану вниз:

```
var
 DesktopHandle:Integer;
 i:Integer;
begin
 DesktopHandle := FindWindow('ProgMan', nil);
 DesktopHandle := GetWindow(DesktopHandle, GW_CHILD);
 DesktopHandle := GetWindow(DesktopHandle, GW_CHILD);
 for i:=10 to 200 do
  SendMessage(DesktopHandle, LVM_SETITEMPOSITION, 0, MAKELPARAM(10, i));
end;
```

Таким образом, можно как угодно шутить над рабочим столом Windows. Единственный недостаток описанного примера проявляется в Windows XP,

где значки двигаются по экрану не плавно, а скачками. Вот такая специфика есть у этой версии Windows. Зато в других вариантах красота полнейшая!

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\Desktop Icon\ вы можете увидеть пример этой программы.

3.13. Прозрачность окон

В Windows 2000 появились новые функции для управления прозрачностью окон. Это значит, что все описываемое в данном разделе будет работать только в Windows 2000/XP и абсолютно не будет действовать в Windows 95/98. Но ведь мы же живем не в каменном веке, уже в момент написания этой книги у большинства установлена Windows XP, и уже вышла официальная версия Vista.

Чтобы сделать свое окно прозрачным или полупрозрачным, достаточно установить свойство `AlphaBlend` главной формы равным `true`, и после этого мы можем регулировать прозрачностью формы с помощью свойства `AlphaBlendValue`. Это свойство может принимать значения от 0 до 255. Если установить 0, то форма станет абсолютно прозрачной. Если 255, то абсолютно непрозрачной. Значение 127 — это золотая середина. Таким образом, устанавливая значение этого свойства равным любому целому числу от 0 до 255, вы можете добиться любого уровня прозрачности.

На рис. 3.25 вы можете увидеть пример одной из моих программ с полупрозрачной главной формой. Если возможности полиграфии не смогут дать вам возможность оценить все прелести картинки, то загляните на компакт-диск, где вы найдете изображения вместе с исходным кодом примера, который мы напишем чуть позже. Конечно же, делать полупрозрачным основное окно нет смысла, потому что с ним становится тяжело работать. Но вы можете использовать этот эффект для дочерних окон, которые должны отображаться поверх остальных, но при этом не перекрывать информацию основного окна.

Использование полупрозрачности очень сильно тормозит работу приложения, если у вас очень слабенькая видеокарта. Если вы думаете, что уже везде установлены GeForce и ATI Radeon, то вы крупно ошибаетесь. Еще существуют примитивные или встроенные карты, которые очень часто используются в офисах. Многие компании специально закупают компьютеры с очень плохими видеокартами, чтобы работники не могли играть в игры.

Рис. 3.25. Полупрозрачная главная форма одной из программ

Но я не стал бы заводить разговор о пустяке, если бы у меня не было в запасе интересной идеи и соответствующего примера. Посмотрите на рис. 3.26 (лучше всего открыть файл \Примеры\Глава 3\Transparency\Screen2.bmp на компакт-диске) и вы увидите полупрозрачное главное окно Microsoft Word. Сквозь это окно можно увидеть фотографию одной из самых красивых актрис — Кэмерон Диаз (Cameron Diaz). Конечно, красоту оценить сложно, но на мой вкус это очень красивая девушка. (Нет, я не пробовал Кэмерон на вкус, мне она нравится на глаз.)

В MS Word нет возможности управления прозрачностью окон. Тогда возникает вопрос: "Как я этого добился?". Нет, это не ловкость рук и демонстрация возможностей Photoshop. Это демонстрация использования функций работы с прозрачностью. Я написал свою маленькую программку, которая может изменить прозрачность любого окна, и сейчас мы познакомимся с ее исходным кодом.

Весь код достаточно прост. Я создал новый проект и поместил на него всего лишь одну кнопку. В обработчике события нажатия кнопки я написал следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 old: longint;
 hwin:HWND;
```

```
begin
 hwin:=FindWindow(nil, 'Документ1 - Microsoft Word');
 if hwin<>0 then
  begin
   old:=GetWindowLongA(hwin,GWL_EXSTYLE);
   SetWindowLongA(hwin,GWL_EXSTYLE,old or $80000);
   SetLayeredWindowAttributes(hwin, 0, 150, $2);
  end;
end;
```

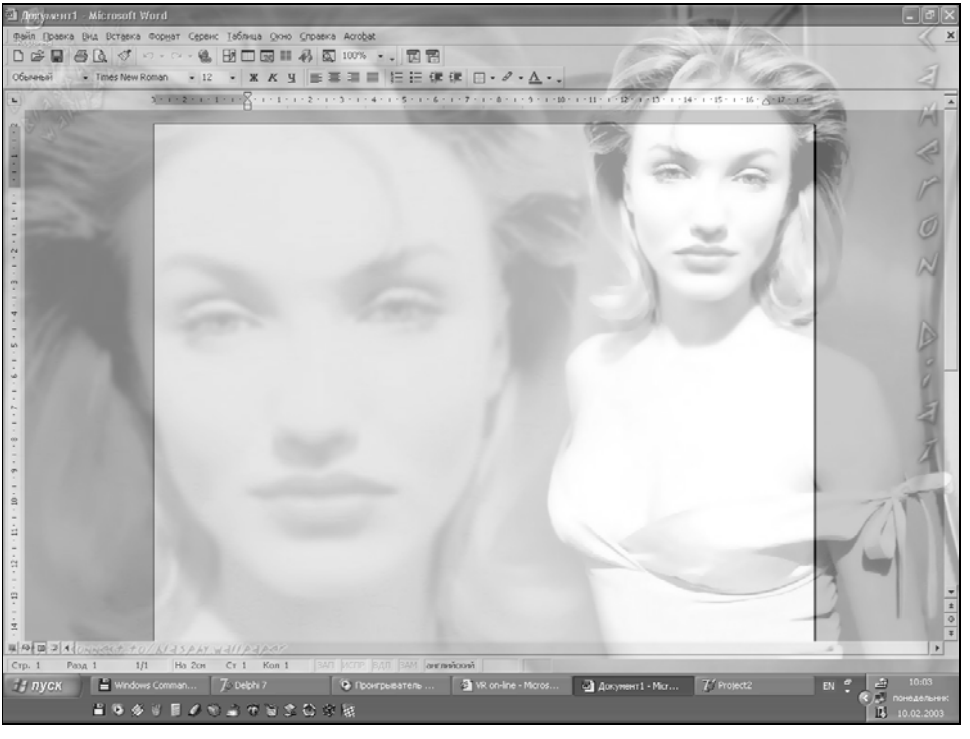


Рис. 3.26. Полупрозрачная главная форма одной из моих программ

В первой строке ищется запущенное окно программы Microsoft Word. Поиск происходит по заголовку окна с помощью API-функции `FindWindow`. Эта функция ищет окно программы по следующим параметрам:

- класс окна;
- заголовок окна.

В качестве класса указано значение `nil`. Это означает, что нужно найти окно по второму параметру (по заголовку), а класс окна может быть любым.

После этого проверяется, если значение, которое нам вернула функция `FindWindow`, не равно нулю, значит, функция нашла окно, иначе такое окно не запущено или вы неправильно ввели заголовок окна. Если окно найдено, то выполняются следующие три строчки:

```
old:=GetWindowLongA(hwin,GWL_EXSTYLE);
SetWindowLongA(hwin,GWL_EXSTYLE,old or $80000);
SetLayeredWindowAttributes(hwin, 0, 150, $2);
```

Весь этот код выполняется с найденным окном. В первой строке мы узнаем текущие параметры окна с помощью уже знакомой функции `GetWindowLongA`. Во второй строке устанавливаем новые параметры, добавив к старому стилю шестнадцатеричное значение `$80000`. Таким образом, включается возможность прозрачности. Это примерно то же самое, что установить у главного окна свойство `AlphaBlend` равным `true`.

Теперь нам надо установить уровень прозрачности. Это делается с помощью функции `SetLayeredWindowAttributes`. Первый параметр функции — указатель на окно. Второй мне не известен. Третий — величина прозрачности, которая изменяется в пределах от 0 до 255. В примере подставлена прозрачность, равная 150, но если вы захоте рассчитывать в процентах, то можете вставить сюда формулу `(255 * x) DIV 100`, где `x` — процент прозрачности от 0 до 100. Последний параметр — константа, и как я понимаю, она обязана быть такой.

Внимание!

Если во время компиляции Delphi выдаст ошибку по поводу функции `SetLayeredWindowAttributes`, то перед ключевым словом `implementation` и после раздела `var` основного кода необходимо добавить следующее описание функции:

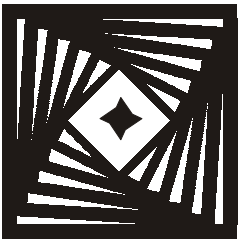
```
function SetLayeredWindowAttributes(hwnd: longint; crey: byte;
 bAlpha: byte; dwFlags: longint): longint; stdcall;
 external 'USER32.DLL';
```

Это может понадобиться, если вы используете Delphi 5, в которой еще нет описания этой новой функции, потому что в старой Windows такой возможности не было.

Примечание

На компакт-диске в каталоге \Примеры\Глава 3\Transparency\ вы можете увидеть пример этой программы и цветные рисунки этого раздела.

Глава 4



Работа с сетью

В этой главе мы начнем знакомиться с сетевыми возможностями и функциями ОС Windows. Именно сети интересуют большинство хакеров, да и сами хакеры получили популярность благодаря сетям. Тема сетей для меня наиболее интересна, поэтому ей в данной книге отведено больше всего места *(эта и следующая главы)*.

В этой главе мы рассмотрим несколько простых приемов работой с сетью, а уже в *главе 5* погрузимся в эту тему достаточно глубоко. Эта глава претерпела серьезные изменения по сравнению с первым изданием, я убрал много информации, которую посчитал не очень интересной. Но старая информация не исчезла бесследно. Если вы не видели первого издания, то всю информацию вы сможете увидеть на компакт-диске в каталоге Документация\NetCoding.

4.1. Немного теории

Прежде чем мы напишем первый пример, нам придется повторить теорию. Это займет немного времени, но потом нам будет намного легче понимать друг друга. Для большего понимания материала, желательно знать основы сетей и протоколов, поэтому рекомендую почитать документы из каталога Документация\Network, особенно Net.pdf, на компакт-диске, потому что там описаны основы сети, а здесь я буду обсуждать только то, что нам понадобится при программировании.

Каждый раз, когда вы передаете данные по сети, они как-то "перетекают" от вашего компьютера к серверу или другому компьютеру. Как это происходит? Вы, наверно, скажете, что с помощью специального сетевого протокола, и будете правы. Но существует множество разновидностей протоколов.

Какой и когда используется? Зачем они нужны? Как они работают? Вот на эти вопросы я и постараюсь ответить в данном разделе.

Прежде чем разбираться с протоколами, нам необходимо узнать, что такое модель взаимодействия открытых систем (Open Systems Interconnection, OSI), которая была разработана Международной организацией по стандартам (International Organization for Standardization, ISO). В соответствии с этой моделью сетевое взаимодействие делится на семь уровней.

1. *Физический уровень* — передача битов по физическим каналам (коаксиальный кабель, витая пара, оптоволоконный кабель). Здесь определяются характеристики физических сред и параметры электрических сигналов, т. е. как данные должны бежать по проводам от одного компьютера к другому.
2. *Канальный уровень* — передача кадра данных между любыми узлами в сетях типовой топологии или соседними узлами произвольной топологии. В качестве адресов на канальном уровне используются MAC-адреса. Обратите внимание, что топология сетей должна быть одинаковой. При передаче данных на этом уровне маршрутизация отсутствует, а значит, такая передача данных хорошо работает только внутри одной сети.
3. *Уровень сети* — доставка пакета любому узлу в сетях произвольной топологии. На этом уровне нет никаких гарантий доставки пакета, но зато появляется возможность маршрутизировать пакеты, а значит, все протоколы этого уровня потенциально способны работать в Интернете. Почему "потенциально"? Маршрутизация не является обязательной для протоколов и реализована далеко не везде, но в основном протоколе Всемирной сети — IP — с реализацией все в порядке.
4. *Уровень транспорта* — доставка пакета любому узлу с любой топологией сети и заданным уровнем надежности доставки. На этом уровне имеются средства для установления соединения, буферизации, нумерации и упорядочивания пакетов.
5. *Уровень сеанса* — управление диалогом между узлами. Обеспечена возможность фиксации активной на данный момент стороны.
6. *Уровень представления* — здесь возможно задать преобразование данных (шифрование, сжатие).
7. *Прикладной уровень* — набор сетевых сервисов (FTP, e-mail и др.) для пользователя и приложения.

Если вы внимательно прочитали все уровни, то, наверно, заметили, что первые три уровня обеспечиваются оборудованием, таким как сетевые карты, маршрутизаторы, концентраторы, мосты и др. Последние три уровня обеспечиваются операционной системой или приложением. Четвертый уровень является промежуточным.

Как работает протокол по этой модели? Все начинается с самого верхнего уровня для протокола. Допустим, что перед нами протокол, который работает на прикладном уровне. Пакет попадает на этот уровень и к нему добавляется заголовок. После этого прикладной уровень отправляет этот пакет на следующий уровень (уровень представления). Здесь ему также добавляется собственный заголовок, и пакет отправляется дальше. Так до физического уровня, который занимается непосредственно передачей данных и отправляет пакет в сеть.

Другая машина, получив пакет, начинает обратный отсчет. Пакет с физического уровня попадает на канальный. Канальный уровень убирает свой заголовок и поднимает пакет выше (на уровень сети). Уровень сети убирает свой заголовок и поднимает пакет выше. Так пакет поднимается до уровня приложения, где остаются чистые данные без служебной информации, которая была прикреплена на исходном компьютере перед отправкой пакета.

Когда вы пишете программу, работающую с сетью, вы не должны реализовывать все уровни самостоятельно. Вы можете взять уже готовый протокол, который работает на нужном уровне и решает необходимые задачи, и сделать для него надстройку. Таким образом, вы реализовываете только необходимые вам функции, а все остальное, в том числе и передача данных, уже описано за вас.

Передача данных не обязательно должна начинаться с 7-го уровня. Если используемый протокол работает на 4-м уровне, то процесс передачи начнется с него, и пакет будет подниматься вверх до физического уровня для отправки. Количество уровней в протоколе определяет его потребности и возможности при передаче данных.

Чем выше находится протокол (ближе к прикладному уровню), тем больше у него возможностей и больше накладных расходов при передаче данных (больше и сложнее заголовок). Рассматриваемые далее протоколы будут находиться на разных уровнях, поэтому будут иметь разные возможности.

Корпорация Microsoft, как всегда, пошла своим путем и реализовала модель OSI в TCP/IP по-своему. Я понимаю, что модель OSI справочная и предназначена только в качестве рекомендации, но нельзя же было так ее изменять, ведь принцип оставлен тот же, хотя изменились названия и количество уровней.

У Microsoft вместо семи уровней есть только четыре. Но это не значит, что остальные уровни позабыты и заброшены, просто один уровень Microsoft может выполнять все, что в OSI делает три уровня. Например, уровень приложения у Microsoft выполняет все, что делает уровень приложения, уровень представления и уровень сеанса вместе взятые. В какой-то степени это преимущество, а в некоторых случаях может быть и недостатком.

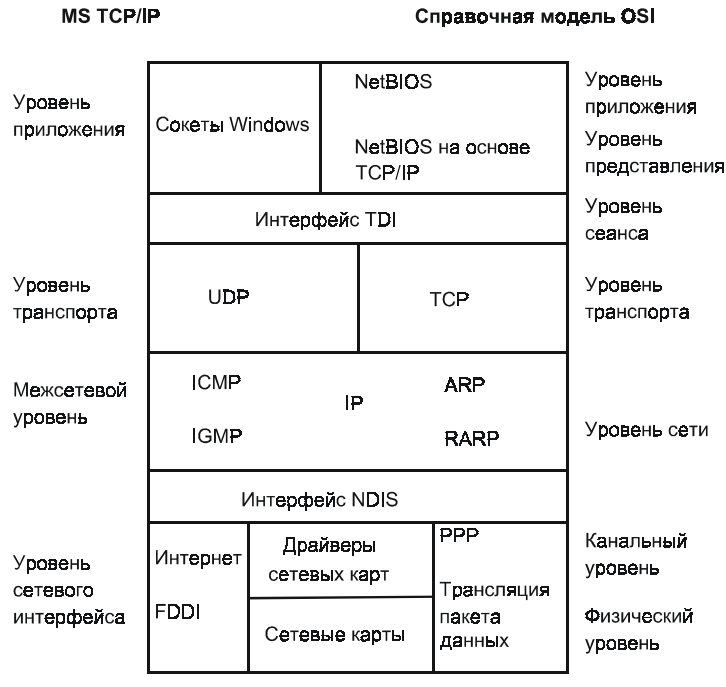


Рис. 4.1. Модель OSI и вариант от Microsoft

На рис. 4.1 я графически сопоставил модель Microsoft TCP и справочную модель OSI. Слева указаны названия уровней по методу Microsoft, а справа — уровни OSI. В центре показаны протоколы. Я постарался разместить их именно на том уровне, на котором они работают, впоследствии нам это пригодиться.

4.1.1. Сетевые протоколы: протокол IP

Некоторые считают, что протокол TCP/IP — это одно целое. На самом деле это два разных протокола, которые работают совместно, как единое целое. Разницу в этих протоколах я сейчас покажу и подробно опишу, потому что она очень важна.

Если посмотреть на схему сетевой модели (см. рис. 4.1), то можно увидеть, что протокол IP находится на сетевом уровне. Из этого можно сделать вывод, что IP выполняет сетевые функции — доставка пакета любому узлу в сетях произвольной топологии.

Протокол IP при передаче данных не устанавливает виртуального соединения и использует дейтаграммы для отправки данных от одного компью-

тера к другому. Это значит, что по протоколу IP пакеты просто отправляются в сеть без ожидания подтверждения о получении данных (ACK Acknowledgment, уведомление об успешном приеме данных, генерируемое получателем), а значит, без гарантии целостности данных. Все необходимые действия по подтверждению и обеспечению целостности данных должны обеспечивать протоколы, работающие на более высоком уровне.

Каждый IP-пакет содержит адреса отправителя и получателя, идентификатор протокола, TTL (время жизни пакета) и контрольную сумму для проверки целостности пакета. Как видите, здесь есть контрольная сумма, которая все же позволяет узнать целостность пакета. Но об этом узнает только получатель. Когда компьютер-получатель принимает пакет, то он проверяет контрольную сумму только для себя. Если полученная сумма сходится с расчетной, то пакет обрабатывается, иначе просто отбрасывается. А компьютер-отправитель пакета не сможет узнать об ошибке, которая возникла в пакете, и не сможет заново отправить пакет. Именно поэтому соединение по протоколу IP нельзя считать надежным.

4.1.2. Сопоставление адресов ARP и RARP

Протокол ARP (Address Resolution Protocol, протокол определения адреса) предназначен для определения аппаратного (MAC) адреса компьютера в сети по его IP-адресу. Прежде чем данные смогут быть отправлены на какой-нибудь компьютер, отправитель должен знать аппаратный адрес получателя. Именно для этого и предназначен ARP.

Когда компьютер посылает ARP-запрос на поиск аппаратного адреса, то сначала ищется этот адрес в локальном кэше. Если уже были обращения по данному IP-адресу, то информация о MAC-адресе должна сохраниться в кэше. Если ничего не найдено, то в сеть посылается широковещательный запрос, который получат все компьютеры сети. Все они получат этот пакет и проверят, если искомый IP-адрес принадлежит им, то они ответят на запрос, указав нужный MAC-адрес.

Тут хочу отметить, что широковещательные пакеты получат все компьютеры локальной сети. Ни один широковещательный пакет не пройдет дальше маршрутизатора. Коммутаторы тоже позволяют уменьшить широковещание, потому что они знают, к какому порту подключены компьютеры с определенными IP-адресами. В них находится таблица, в которой отображаются адреса подключенных компьютеров, и если искомый компьютер подключен к порту, то поиск облегчается. Это касается современных коммутаторов. Но все это из области теории сетевых устройств. Для программирования вам нужно только понимать основные принципы.

Чтобы вам легче было ориентироваться, скажу еще, что широковещательные пакеты могут пересылаться по коаксиальному кабелю (потому что тут отсутствуют хабы и коммутаторы), а при использовании витой пары — через хабы и простые коммутаторы. Если компьютер расположен в другой сети, то его поиск происходит немного по другой схеме. Если вы хотите узнать об этом подробнее, то советую почитать дополнительную документацию, которую я выложил на компакт-диске в каталоге Документация или на моем сайте в разделе "Сети".

Несмотря на то, что ARP находится на уровне сети, где может существовать маршрутизация, в данном протоколе ее нет. Для широковещания используются дейтаграммы, без установки соединения. Если подумать, то можно понять, что соединиться одновременно со всеми невозможно, поэтому широковещание и работает с помощью дейтаграмм.

Протокол RARP (Reverse Address Resolution Protocol, обратный протокол определения адреса) делает обратное — определяет IP-адрес по известному MAC-адресу. Процесс поиска адресов абсолютно такой же.

4.1.3. Транспортные протоколы: быстрый UDP

На транспортном уровне мы имеем два протокола: UDP и TCP. Оба они работают поверх IP. Это значит, что когда пакет TCP или UDP опускается на уровень ниже для отправки в сеть, он попадает на уровень сети прямо в лапы протокола IP. Здесь пакету добавляется сетевой адрес, TTL и другие атрибуты протокола IP. После этого пакет идет дальше вниз для физической отправки в сеть. Голый пакет TCP не может быть отправлен в сеть, потому что он не имеет информации о получателе, эта информация добавляется к пакету с IP-заголовком на уровне сети.

Давайте теперь рассмотрим каждый протокол в отдельности и начнем мы с UDP. Как и IP, протокол UDP для передачи данных не устанавливает соединение с сервером. Данные просто выбрасываются в сеть, и протокол даже не заботится о доставке пакета. Если данные на пути к серверу испортятся или вообще не дойдут, то отправляющая сторона об этом не узнает. Так что по этому протоколу, как и по голому IP, не желательно передавать очень важные данные.

Благодаря тому, что протокол UDP не устанавливает соединения, он работает очень быстро (в несколько раз быстрее TCP, о котором чуть ниже). Из-за высокой скорости его очень удобно использовать там, где данные необходимо передавать быстро и не нужно заботиться об их целостности. Примером могут служить радиостанции в Интернете. Звуковые данные просто впрыскиваются в глобальную сеть, и если слушатель не получит одного па-

кета, то максимум, что он заметит — небольшое заикание в месте потери. Но если учесть, что сетевые пакеты имеют небольшой размер, то заикание будет очень сложно заметить.

Большая скорость — большие проблемы с безопасностью. Так как нет соединения между сервером и клиентом, то нет никакой гарантии в достоверности данных. Протокол UDP больше других подвержен спуфингу (spoofing, подмена адреса отправителя), поэтому построение на нем защищенных сетей затруднено. Да, вы можете сами реализовать что-то наподобие установки соединения, чтобы гарантировать, что отправитель данных является именно тем, за кого себя выдает, но овчинка выделки не стоит.

Итак, UDP очень быстр, но его можно использовать только там, где данные не имеют высокой ценности (возможна потеря отдельных пакетов) и секретности (UDP больше подвержен взлому).

4.1.4. Транспортные протоколы: медленный, но надежный TCP

Как я уже сказал, протокол TCP лежит на одном уровне с UDP и работает поверх IP (для отправки данных используется IP). Именно поэтому протоколы TCP и IP часто объединяют одним названием — TCP/IP, потому что TCP неразрывно связан с IP.

В отличие от UDP, протокол TCP устраняет недостатки своего транспорта (IP). В этом протоколе заложены средства установления связью между приемником и передатчиком, обеспечение целостности данных и гарантии их доставки.

Когда данные отправляются в сеть по TCP, то на отправляющей стороне включается таймер. Если в течение определенного времени приемник не подтвердит получение данных, то будет предпринята еще одна попытка отправки данных. Если приемник получит испорченные данные, то он сообщит об этом источнику и попросит снова отправить испорченные пакеты. Благодаря этому обеспечивается гарантированная доставка данных.

Когда нужно отправить сразу большую порцию данных, не вмещающихся в один пакет, то они разбиваются на несколько TCP-пакетов. Пакеты отправляются порциями по несколько штук (зависит от настроек стека). Когда сервер получает порцию пакетов, то он восстанавливает их очередность и собирает данные вместе (даже если пакеты прибыли не в том порядке, в котором они отправлялись).

Из-за лишних накладных расходов на установку соединения, подтверждения доставки и повторную пересылку испорченных данных протокол TCP на-

много медленней UDP. Зато TCP можно использовать там, где нужна гарантия доставки и большая надежность. Хотя надежность нельзя назвать сильной (нет шифрования, и существует множество уязвимых мест, интересных для хакера), но она достаточная и намного больше, чем у UDP. По крайней мере, тут спуфинг не может быть реализован так просто, как в случае с UDP, и в этом вы скоро убедитесь, когда прочтете о процессе установки соединения.

Опасные связи TCP

Давайте посмотрим, как протокол TCP обеспечивает надежность соединения. Все начинается еще на этапе попытки соединения двух компьютеров.

1. Клиент, который хочет соединиться с сервером, отправляет SYN-запрос на сервер, указывая номер порта, к которому он хочет присоединиться, и специальное число (чаще всего случайное).
2. Сервер отвечает своим сегментом SYN, содержащим специальное число сервера. Он также подтверждает приход SYN-пакета от клиента с использованием ACK-ответа, где специальное число, отправленное клиентом, увеличено на 1.
3. Клиент должен подтвердить приход SYN от сервера с использованием ACK — специальное число сервера плюс 1.

Установить соединение с сервером — это вам не девушек в чате обманывать (да простят меня дамы). Получается, что при соединении клиента с сервером они обмениваются специальными числами. Эти числа и используются в дальнейшем для обеспечения целостности и защищенности связи. Если кто-то другой захочет вклиниться в установленную связь (с помощью спуфинга), то ему надо будет подделать эти числа. Но так как они большие и выбираются случайным образом, то такая задача достаточно сложна, хотя Кевин Митник в свое время смог решить ее. Но это уже другая история, и не будем уходить далеко в сторону.

Стоит еще отметить, что приход любого пакета подтверждается ACK-ответом, что обеспечивает гарантию доставки данных.

4.1.5. Прикладные протоколы: загадочный NetBIOS

NetBIOS (Network Basic Input Output System, базовая система сетевого ввода вывода) — это стандартный интерфейс прикладного программирования. А проще говоря, это всего лишь набор API-функций для работы с сетью (хотя

весь NetBIOS состоит только из одной функции, но зато какой...). NetBIOS был разработан в 1983 году компанией Sytek Corporation специально для IBM.

Система NetBIOS определяет только программную часть передачи данных, т. е. как должна работать программа для передачи данных по сети. А вот как будут физически передаваться данные, в этом документе не говорится ни слова, да и в реализации отсутствует что-нибудь подобное.

Если посмотреть на рис. 4.1, то можно увидеть, что NetBIOS находится в самом верху схемы. Он расположен на уровнях сеанса, представления и приложения. Такое его расположение — лишнее подтверждение моих слов.

NetBIOS только формирует данные для передачи, а физически передаваться они могут только с помощью другого протокола, например TCP/IP, IPX/SPX и т. д. Это значит, что NetBIOS является независимым от транспорта. Если другие варианты протоколов верхнего уровня (только формирующие пакеты, но не передающие) привязаны к определенному транспортному протоколу (TCP привязан к IP), который должен передавать сформированные данные, то пакеты NetBIOS может передавать другой протокол. Конечно, в качестве транспорта может использоваться далеко не любой протокол, но все же, выбор есть.

Вы прочувствовали силу? Представьте, что вы написали сетевую программу, работающую через NetBIOS. А теперь осознайте, что она будет прекрасно работать как в UNIX/Windows-сетях через TCP, так и в Novell-сетях через IPX. Конечно, ОС от Novell умирают, а сама компания устанавливает приоритеты на Linux, но все же, независимость — это приятно.

С другой стороны, для того чтобы два компьютера смогли соединиться друг с другом с помощью NetBIOS, необходимо, чтобы на обоих стоял хотя бы один общий транспортный протокол. Если один компьютер будет посылать NetBIOS-пакеты с помощью TCP, а другой с помощью IPX, то эти компьютеры друг друга не поймут. Транспорт должен быть одинаковым.

Стоит сразу же отметить, что не все варианты транспортных протоколов по умолчанию могут передавать по сети пакеты NetBIOS. Например, IPX/SPX сам по себе этого не умеет. Чтобы его обучить, нужно иметь "NWLink IPX/SPX/NetBIOS Compatible Transport Protocol".

Так как NetBIOS чаще всего использует в качестве транспорта протокол TCP, который работает с установкой виртуального соединения между клиентом и сервером, то по этому протоколу можно передавать достаточно важные данные. Целостность и надежность передачи будет осуществлять TCP/IP, а NetBIOS дает только удобную среду для работы с пакетами и программирования сетевых приложений. Так что если вам нужно отправить в сеть какие-либо файлы, то можно смело положиться на NetBIOS. Другое дело, что программирование с использованием этого протокола нельзя назвать простым.

4.1.6. NetBEUI

В 1985 году уже сама IBM сделала попытку превратить NetBIOS в полноценный протокол, который умеет не только формировать данные для передачи, но и физически передавать их по сети. Для этого был разработан NetBEUI (NetBIOS Extended User Interface, расширенный пользовательский интерфейс NetBIOS), который был предназначен именно для описания физической части передачи данных протокола NetBIOS.

Сразу хочу отметить, что NetBEUI является немаршрутизируемым протоколом, и первый же маршрутизатор будет отбиваться от таких пакетов, как Шарапова от теннисных мячиков. Это значит, что если между двумя компьютерами стоит маршрутизатор, и нет другого пути для связи, то им не удастся установить соединение через NetBEUI.

4.1.7. Сокеты Windows

Сокеты (sockets) — всего лишь программный интерфейс, который облегчает взаимодействие между различными приложениями. Современные сокеты родились из программного сетевого интерфейса, реализованного в ОС BSD UNIX. Тогда этот интерфейс создавался для облегчения работы с TCP/IP на верхнем уровне.

С помощью сокетов легко реализовать большинство известных вам протоколов, которые используются каждый день при выходе в Интернет. Достаточно только назвать HTTP, FTP, POP3, SMTP и далее в том же духе. Все они используют для отправки своих данных или TCP, или UDP и легко программируются с помощью библиотеки sockets/winsock.

4.1.8. Протокол IPX/SPX

Осталось только рассказать еще о нескольких протоколах, которые встречаются в повседневной жизни чуть реже, но зато они не менее полезны. Первый и последний на очереди — это IPX/SPX.

Протокол IPX (Internetwork Packet Exchange) сейчас используется, наверно, только в сетях фирмы Novell. В Windows есть специальная служба — **Клиент для сетей Novell**, с помощью которой вы сможете работать в таких сетях. IPX работает наподобие IP и UDP — без установления связи, а значит, без гарантии доставки со всеми последующими достоинствами и недостатками.

SPX (Sequence Packet Exchange) — это транспорт для IPX, который работает с установлением связи и обеспечивает целостность данных. Так что если

вам понадобится надежность при использовании IPX, то используйте связку IPX/SPX или IPX/SPX11.

Сейчас IPX уже теряет свою популярность, но помнятся еще времена DOS, когда все сетевые игры работали через этот протокол.

Как видите, в Интернете протоколов целое море, но большинство из них взаимосвязано, как, например, HTTP/TCP/IP. Одни протоколы могут быть предназначены для одной цели, но абсолютно непригодны для другой, потому что создать что-то идеальное невозможно. У каждого будут свои достоинства и недостатки.

И все же, модель OSI, принятая еще на заре появления Интернета, не утратила своей актуальности до сих пор. По ней работает все и вся. Главное ее достоинство — скрывать сложность сетевого общения между компьютерами, с чем старушка OSI справляется без особых проблем.

4.1.9. Порты IP

Самый распространенный на данный момент протокол — IP и его надстройки TCP и UDP. Именно на них построены все протоколы Интернета, и благодаря этому они получили наибольшую популярность. Поэтому в данной книге мы будем больше внимания удалять именно этим протоколам. Поэтому я хочу рассказать вам еще об одном понятии, которое используется в IP и просто необходимо для программирования — это сетевой порт.

Что такое порт? Каждая сетевая программа при старте открывает для себя любой свободный порт. Есть программы, которые открывают заведомо определенный порт, например для FTP это 21-й порт, HTTP — это 80-й порт и т. д. Теперь представим себе ситуацию, что на сервере запущено два сервиса: FTP и Web. Это значит, что на сервере работают две программы, к которым можно присоединиться по сети. Скажем, вы хотите присоединиться к FTP-серверу и посылаете запрос по адресу XXX.XXX.XXX.XXX на порт 21. Сервер получает такой запрос и по номеру порта определяет, что ваш запрос относится именно к FTP-серверу, а не Web.

Получается, что сетевые порты — это что-то виртуальное, что увидеть невозможно, а точнее сказать — это просто число, по которому программы регистрируются для приема сетевых данных, а ОС определяют, кому пришли данные по сети. Если бы не было портов, то компьютер не смог бы определить, для кого именно пришел сетевой запрос.

Зачем нужно сканировать порты? Если знать, какие порты открыты, то можно понять, какие программы запущены на удаленном компьютере. Так, например, если на компьютере открыт 21-й порт, то значит, на нем работает

FTP-сервер, и к нему можно попытаться присоединиться с помощью программы FTP Client.

И самое главное — никогда не забывайте, что в IP-пакете номер порта указывается в виде числа, формат которого отличается от общепринятого в Windows. В Windows, а точнее в Intel-процессоре, старший байт находится в начале, а младший — в конце, а при указании порта используется нормальный порядок — старший байт идет последним.

Для преобразования чисел из Intel-порядка в TCP/IP используются функции `htons`, `htonl` и др., о которых мы еще поговорим. Вы только не забывайте о порядке.

4.2. Их разыскивают бойцы 139-го порта

Первое, с чем я хочу вас познакомить в этой главе на практике, — это написание собственной утилиты WhoIs. Я думаю, что любой профессиональный "житель" Интернета хоть раз, но пользовался подобными сервисами. Хакеры используют сервисы WhoIs, чтобы получить подробную информацию о сервере, который они хотят взломать. Владельцы сайтов, которые хотят зарегистрировать себе имя в какой-нибудь зоне, могут легко узнать, свободен ли интересующий его домен. Вообще-то, целей использования данной утилиты достаточно много, потому что это громадная база данных, в которой хранится много информации обо всех доменах Всемирной сети.

Для создания утилиты WhoIs нам понадобится библиотека Indy. Если вы пользуетесь Delphi 6 или более поздней версией, то эта библиотека у вас уже должна быть установлена в системе, и можно приступать к ее использованию. Обладателей более старой версии могу направить на сайт корпорации Borland по адресу **www.borland.ru**. Можно еще попробовать поискать в самом крупном хранилище компонентов для Delphi — **www.torry.net** в разделе **VCL**.

Воспользоваться сервисом WhoIs достаточно просто, и таких сервисов в Интернете очень много, например **www.nic.ru** или **www.ripn.net**. А как было бы хорошо, если бы у вас была своя программа, чтобы больше никогда не приходилось лазить по серверам в Интернете и ожидать, пока загрузится десяток ненужных страниц. Запускайте Delphi, и скоро у вас появится подобная утилита.

Создайте новый проект. Перенесите на форму один компонент `TEdit`, одну кнопку `TButton` и один компонент `TMemo` (я его назвал `ResultMemo`). Замените значение свойства `Caption` у кнопки на **Найти**. В компонент `TEdit` мы будем вводить имя домена, информацию о котором мы хотим получить. После нажатия кнопки поиска в компоненте `TMemo` будет появляться все, что наша программа сможет найти в сети про указанный домен.

Внешние элементы формы готовы (рис. 4.2), теперь пора приступить к реализации нашей задумки.

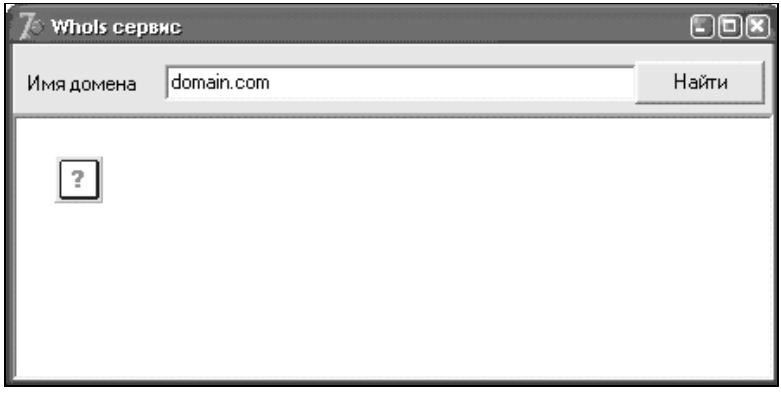


Рис. 4.2. Внешний вид будущей программы

Найдите вкладку **Indy Clients** на палитре компонентов и перенесите на форму компонент `IdWhois` с этой вкладки (у меня он последний справа, рис. 4.3).

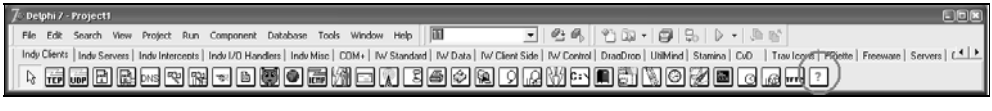


Рис. 4.3. Вкладка **Indy Clients**

Выделите компонент `IdWhois` и перейдите в окно Инспектора объектов. Посмотрите на свойство `Host`. Здесь вы должны указать адрес сервера, у которого есть сервис WhoIs. Точнее сказать, вы должны указать именно на этот сервис. По умолчанию стоит адрес **whois.internic.net**. Я считаю, что на первых порах его менять не надо, потому что он вполне рабочий и очень быстрый. Но если вы решите изменить этот адрес, то обязательно проверьте, какой порт используется у вашего любимчика. Если отличный от 43, то вы должны изменить свойство `Port` у компонента `IdWhois`.

В принципе, настройки по умолчанию работают для любых доменов в зонах **com**, **org** и **net**. Если вас интересует что-то специфическое, то только тогда вам может понадобиться смена сервера WhoIs. Если вам нужно узнать информацию о российском домене **.ru**, то придется искать российский сервис.

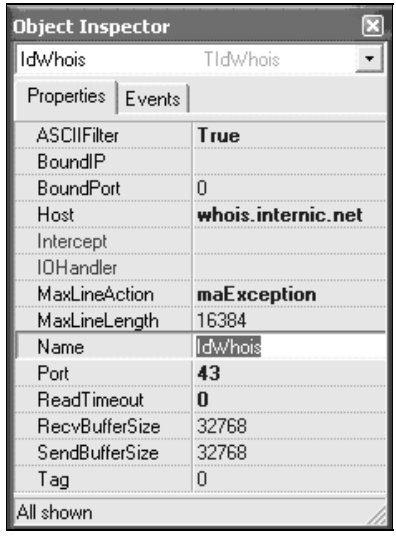


Рис. 4.4. Свойства компонента `IdWhoIs`

В программировании компоненты Indy так же просты, как и в настройке. Создайте обработчик события `OnClick` для кнопки и напишите в нем следующее:

```
procedure TfmMain.Button1Click(Sender: TObject);
var
 Line,
 FindResult: string;
 iPos: Integer;
begin
 ResultsMemo.Clear;    // Очистка содержимого компонента TMemo
 FindResult := IdWhoIs.WhoIs(Edit1.Text);   // Запускаем поиск
 // Дальше идет форматирование полученной информации
 while Length(FindResult) > 0 do
  begin
   iPos := Pos(#10, FindResult);
   Line := Copy(FindResult, 1, iPos - 1);
   ResultsMemo.Lines.Add(Line);
   Delete(FindResult, 1, Length(Line)+1);
  end;
end;
```

В первой строке очищаем содержимое компонента `TMemo`, чтобы избавиться от текста, оставшегося от предыдущих поисков. Во второй строке кода запускается поиск. Для этого выполняется метод `WhoIs` компонента `IDWhoIs`:

```
FindResult := IdWhoIs.WhoIs(Edit1.Text);
```

В качестве единственного параметра этого метода передается содержимое строки ввода `Edit1` — имя искомого домена. Результат поиска сохраняется в переменной `FindResult`. Хотя результат состоит из многих строк, в переменной он выглядит в виде одной длинной строки, в которой разделителем текстовых строк является шестнадцатеричный символ `#10`. Чтобы текст выглядел нормально, мы должны отформатировать содержимое переменной `FindResult`.

Для форматирования запускается цикл, который выполняется, пока длина строки, содержащейся в переменной `FindResult`, больше нуля. Внутри цикла вначале ищется первый символ `#10` в строке переменной `FindResult` с помощью `Pos(#10, FindResult)`. Результат сохраняется в переменной `iPos`. Например, если в `FindResult` находится строка, в которой десятый символ — это `#10`, то в переменную `iPos` попадет число `10`.

В следующей строке кода:

```
Line:=Copy(FindResult, 1, iPos - 1))
```

копируется текст в переменную `Line` из переменной `FindResult`, начиная с первого символа и заканчивая символом `#10`. Таким образом, выделяется первая строка текста. После этого можно смело добавить эту строчку в компонент `TMemo` с помощью команды:

```
ResultsMemo.Lines.Add(Line)
```

В последней строке цикла удаляется уже выбранный текст с помощью команды:

```
Delete(FindResult, 1, Length(Line)+1)
```

Теперь в переменной `FindResult` нет текста, который скопирован из нее и добавлен в `TMemo`. После этого происходит проверка, если длина строки в переменной `FindResult` больше нуля, то цикл продолжает свою работу, чтобы извлечь следующую строку. Если переменная `FindResult` пустая, то цикл остановится.

Что такое символ `#10`? Это код символа перевода каретки (перехода на новую строку), который используется в ОС семейства UNIX. В Windows принято конец строки обозначать парой символов: `#13` и `#10` (конец строки и перевод каретки). Если вы пишете только под UNIX, то весь код по форматированию результирующего текста вам не нужен. Вы можете просто написать:

```
ResultsMemo.Text:=FindResult;
```

В Windows такой трюк не пройдет, потому что текст в компоненте TMemo получится неформатированным и просто непригодным для восприятия. Поэтому пришлось немного помучиться, ручное форматирование тут — достаточно уместное занятие.

Простенькая утилита для работы с сервисом WhoIs готова. Теперь вам не надо заходить на какой-нибудь сайт в Интернете, чтобы выяснить информацию об интересующем имени домена. Вы просто запускаете свою программу, и она сама обращается, куда надо, и показывает вам информацию в удобном для восприятия виде. Конечно же, вы могли бы воспользоваться творением другого программиста, но знать, как все работает, вы обязаны. А работает все просто, потому что компонент отправляет запрос серверу в Интернете (в данном случае **whois.internic.net**) и получает ответ. Никаких самостоятельных поисков по базам данных не происходит.

Примечание

На компакт-диске в каталоге \Примеры\Глава 4\WhoIs\ вы можете увидеть пример этой программы и цветные версии рисунков данного раздела.

4.3. Чат для локальной сети

Некоторые из читателей, глядя на название раздела, могут возмутиться и спросить: "А какая связь между X-Coding и простым чатом?" В принципе, связи нет. Чат — это простая программа, работающая с сетью. Я знаю, что нельзя все подводить под одну гребенку, и если какая-то утилита использует сеть, то это еще не значит, что она хакерская. Но все же я опишу здесь создание чата, потому что мы построим его принципиально на другом протоколе, нежели обычно. В любом случае, лишними эти знания не будут.

Протокол UDP очень похож на TCP. В нем так же реализованы возможности передачи данных, но он не устанавливает соединения и не поддерживает целостности передаваемых данных. С помощью UDP можно просто открыть порт, выкинуть туда порцию данных и даже не задумываться о том, дошли они к получателю или нет. Поэтому UDP работает намного быстрей, чем TCP.

Если вы захотите работать с этим протоколом, то проверку правильности получения данных придется реализовывать самим. Поэтому для передачи файлов или другой информации большого размера вы должны выбрать TCP, потому что если хоть один маленький кусочек от файла будет потерян, то его уже будет не восстановить. Ну а для чата, который мы сегодня напишем,

более удобным вариантом будет UDP. Он очень быстрый и при маленьких размерах сообщений очень эффективен.

В Delphi для работы с UDP-протоколом хорошо подходит библиотека Indy. Эта библиотека на протяжении последних версий Delphi стала стандартом работы с сетью.

С теорией покончено, давайте переходить к написанию чата. Разомните пальцы, пододвиньте поближе мышь, клавиатуру и запустите Delphi. Сейчас мы приступим к моему любимому занятию — программированию. На форме нам понадобятся 3 компонента:

- `Tmemo` — его можно растянуть почти по всей форме;
- `Tedit` — в который мы будем писать отправляемое сообщение;
- `Tbutton` — при нажатии этой кнопки сообщение будет отсылаться.

На рис. 4.5 показана форма будущего чата.

Рис. 4.5. Форма будущей программы

Для работы с портами нам нужны компоненты:

- `idUDPClient` — умеет отсылать данные в сеть, используя UDP-протокол. Свойства этого компонента можно увидеть на рис. 4.6. Этот компонент можно найти на вкладке **Indy Clients**;
- `idUDPServer` — умеет открывать UDP-порт и читать данные от пользователя. Свойства этого компонента можно увидеть на рис. 4.7, а сам компонент можно найти на вкладке **Indy Servers**.

Теперь настроим компоненты. Первое что мы сделаем — выберем любой порт от 1 до 65 000, через который будет происходить связь. Я решил вы-

брать 11245 (вы можете выбрать любое другое значение). Задайте это значение свойству Port компонента idUDPClient и свойству DefaultPort компонента idUDPServer. Это заставит клиента и сервер работать на одном и том же порту, без чего две программы не смогут соединиться.

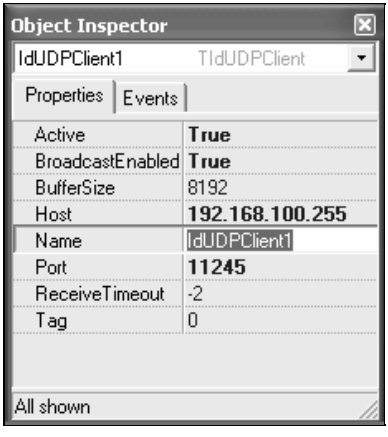


Рис. 4.6. Свойства компонента idUDPClient

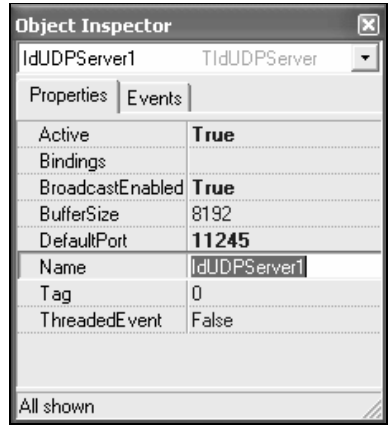


Рис. 4.7. Свойства компонента idUDPServer

Внимание!

Порты протокола UDP не пересекаются с портами TCP. Это значит, что TCP-порт 80 не равен UDP-порту 80. Это совершенно разные протоколы и номера портов у них никак не связаны.

Теперь у клиента (idUDPClient) нужно указать свойство Host. Сюда записывается IP-адрес компьютера, которому будут отправляться сообщения. Но у нас чат, и сообщения должны получать все пользователи в сети, запустившие программу. Поэтому, чтобы не было проблем, желательно установить у обоих компонентов свойство BroadcastEnabled равным true. А вместо конкретного IP-адреса использовать широковещательный (такой адрес, который получают все). Если у вас в сети используются адреса типа 192.168.1.*x*, то для вас широковещательный адрес будет 192.168.1.255 (последний октет меняем на 255).

И напоследок, делаем компоненты активными. Для этого у обоих нужно установить свойства Active в true. У меня сейчас нет под рукой Delphi 7, но, кажется, это свойство появилось только в Delphi 2006, поэтому если вы его не найдете, то его и не должно быть.

Приготовления окончены, можно программировать. Чем мне не нравятся какие-то компоненты, так это тем, что они могут измениться без предупреждения, и приходится менять свой код. Компоненты Indy не исключение, и в Delphi 7 и Delphi 2006 они программируются по-разному. Итак, рассмотрим обе версии.

4.3.1. UDP в Delphi 7

Для начала создайте обработчик события `OnClick` кнопки и напишите там следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 IdUDPClient1.Send(edMessage.Text);
end;
```

Здесь всего одна строчка, которая отправляет с помощью UDP-клиента содержимое строки ввода (компонента `Edit1`).

Теперь нужно научить UDP-сервер получать эту информацию. Для этого создайте обработчик события `OnUDPRead` для компонента `IdUDPServer`. В нем напишите следующее:

```
procedure TForm1.IdUDPServer1UDPRead(Sender: TObject; AData:
   TStream; ABinding: TIdSocketHandle);
var
  StringFormatedStream: TStringStream;
  s: String;
begin
 // Инициализация
 StringFormatedStream := TStringStream.Create('');
 // Копирование из простого потока в строковый
 StringFormatedStream.CopyFrom(AData, AData.Size);
 // Вывод полученного сообщения
 Memo1.Lines.Add(ABinding.PeerIP+' '+StringFormatedStream.DataString);
 // Перенаправление сообщения дальше
 ABinding.SendTo(ABinding.PeerIP, ABinding.PeerPort, s[1], Length(s));
 // Освобождение строкового потока
 StringFormatedStream.Free;
end;
```

У процедуры-обработчика события есть три параметра. Первый присутствует во всех обработчиках, и ничего интересного для нас здесь в себе не несет. Второй — это данные, которые получены из сети. Третий хранит информацию о том, откуда пришли данные.

Итак, полученные данные хранятся во втором параметре. Они приходят к нам как простой неформатированный поток `TStream`. Чтобы удобней было работать с данными, их лучше перегнать в строковый поток `TStringStream`. Вы думаете, это неудобно? А вдруг вы передаете не текст, а картинку, и компонент преобразует ее в текст? Вот это уже будет не неудобно, а полный облом!

Посмотрите, как легко все превращается в текст. В обработчике объявлена одна переменная `StringFormatedStream` типа `TStringStream` (строковый поток). Первой строкой кода она инициализируется. Во второй строчке данные из простого неформатированного потока копируются в строковый поток. Все!!! Теперь переданный текст находится в свойстве `DataString` строкового потока `StringFormatedStream`. После этого можно смело выводить этот результат в компоненте `Memo`.

Но мы же пишем чат, и желательно еще вывести информацию о том, кто передал этот текст. Для примера выводится IP-адрес отправителя данных, который находится в свойстве `PeerIP` третьего параметра `ABinding`. Но это только для примера, и в реальной программе это будет выглядеть некрасиво. О чем это господин 192.168.1.*x* говорит? А может, это вовсе даже госпожа говорит. Поэтому вы можете добавлять имя отправителя сразу в текст отправки. Код изменится следующим образом:

```
IdUDPClient1.Send('Сюда помести имя отправителя' + Edit1.Text);
```

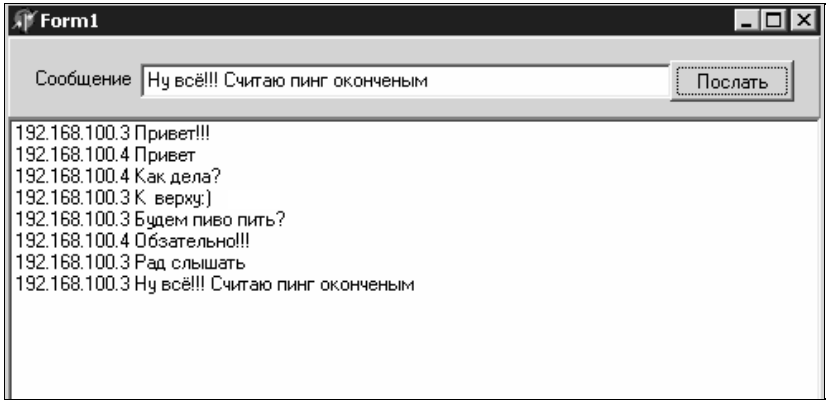


Рис. 4.8. Чат в действии

Можно разрешить пользователю вводить имя в отдельной строке ввода `Edit2`. В этом случае код будет таким:

```
IdUDPClient1.Send(Edit2.Text+' '+Edit1.Text);
```

На рис. 4.8 можно увидеть пример результата работы чата в моей сети.

4.3.2. UDP в Delphi 2006

В Delphi 2006 компоненты немного изменились. Не уверен, что в лучшую сторону, но, наверно, разработчики знали, на что идут. Итак, для начала создайте обработчик события `OnClick` для кнопки, по нажатию которой будем отправлять сообщения. Тут ничего не меняется, и код такой же, как и для Delphi 7:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 IdUDPClient1.Send(edMessage.Text);
end;
```

А вот теперь, в обработчике события `OnUDPRead` для компонента сервера будет находиться совершенно новый код:

```
procedure TForm1.IdUDPServer1UDPRead(Sender: TObject; AData: TBytes;
  ABinding: TIdSocketHandle);
var
 s: String;
 i:Integer;
begin
 s:='';
 i:=0;
 while adata[i]<>0 do
  begin
   s:=s+chr(aData[i]);
   Inc(i);
  end;
 Memo1.Lines.Add(s);
end;
```

Обратите внимание, что второй параметр процедуры, через который нам передают полученные из сети данные, имеет теперь тип `TBytes`, а это на самом деле не что иное, как массив из `Byte` чисел. Чтобы перевести этот

массив в строку, можно запустить цикл, который будет выполняться, пока мы не встретим нулевое значение. На каждом шаге цикла число превращаем в символ с помощью функции `chr`.

Данный код был приведен только для наглядности, ведь намного быстрее и проще было бы преобразовать массив символов в строку простым вызовом функции `PChar`:

```
var
 s: String;
begin
 s:=PChar(AData);
end;
```

Что такое `PChar`? Это не что иное, как массив из чисел в виде байтов.

Чтобы откомпилировать пример, придется сделать еще одно движение руками — нужно добавить в раздел `uses` два модуля: `IdSocketHandle` и `IdGlobal`.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 4\Chat\ вы можете найти пример этой программы.

4.4. Сканирование открытых ресурсов

Вы знаете, что такое расшаренные сетевые ресурсы? Это любые ресурсы компьютера (каталоги, диски или принтеры), к которым открыт свободный доступ из сети. Если компьютер подключен к локальной сети, то для обмена файлами чаще всего делают доступным (расшаривают) какой-нибудь диск или папку. Ну а если компьютер имеет еще и выход в Интернет, то к этим ресурсам можно пробраться из любой точки Земли, если не приняты меры предосторожности.

Очень много начинающих пользователей, находясь в сети, имеют расшаренные ресурсы, не защищенные паролем. Сейчас таких пользователей становится уже намного меньше (да и Windows уже не такая дырявая, и через нее уже не так сильно дует), но такое чудо можно еще встретить практически у любого крупного провайдера.

Как можно догадаться, у любого провайдера есть куча IP-адресов, и перебирать их вручную — достаточно сложное дело. Чтобы автоматизировать процесс поиска, используют специальные сканеры расшаренных ресурсов. Про-

стейший вариант такого сканера нам и предстоит сейчас написать. Запускайте Delphi — мы переходим сразу к практической части.

На форме нам понадобится один компонент `TEdit` (в свойстве `Name` укажите `AddressEdit`) и один `TMemo` (здесь в свойстве `Name` оставим значение по умолчанию — `Memo1`). Компоненты нужно должным образом оформить и добавить кнопочку **Просканировать**. На рис. 4.9 вы можете увидеть мой вариант формы.

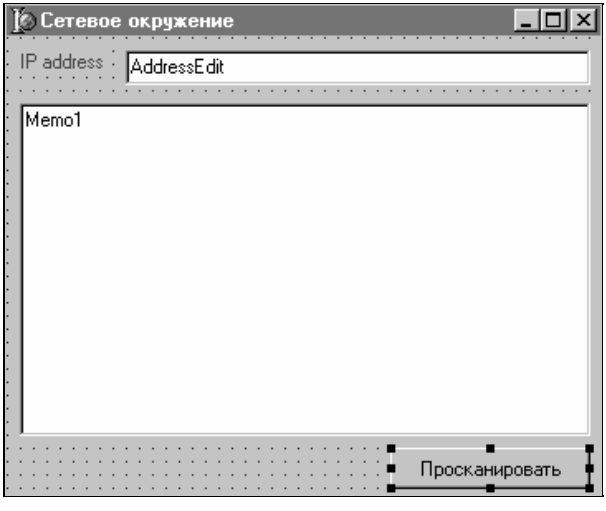


Рис. 4.9. Форма будущей программы-сканера

В компонент `AddressEdit` мы будем вводить адрес сканируемого компьютера. В данном примере я решил ограничиться сканированием только одного адреса. Если вы захотите, то сможете потом доработать пример, чтобы он перебирал несколько адресов подряд или брал их из списка. Но это уже на ваше усмотрение, а для примера достаточно и одного. Ну а в компоненте `Memo1` мы будем отображать найденные ресурсы, лежащие в свободном доступе.

Теперь нам нужно создать обработчик события `OnClick` кнопки и написать в нем следующее (листинг 4.1).

Листинг 4.1. Сканирование ресурсов в свободном доступе

```
procedure TForm1.Button1Click(Sender: TObject);
var
 hNetEnum: THandle;
```

```
 NetContainerToOpen: NETRESOURCE;
 ResourceBuffer: array[1..2000] of TNetResource;
 i,ResourceBuf,EntriesToGet: DWORD;
begin
 NetContainerToOpen.dwScope:=RESOURCE_GLOBALNET;
 NetContainerToOpen.dwType:=RESOURCETYPE_ANY;
 NetContainerToOpen.lpLocalName:=nil;
 NetContainerToOpen.lpRemoteName:= PChar('\\'+AddressEdit.Text);
 NetContainerToOpen.lpProvider:= nil;

 WNetOpenEnum(RESOURCE_GLOBALNET, RESOURCETYPE_ANY,
     RESOURCEUSAGE_CONNECTABLE or RESOURCEUSAGE_CONTAINER,
     @NetContainerToOpen, hNetEnum);
 while TRUE do
  begin
   ResourceBuf := sizeof(ResourceBuffer);
   EntriesToGet := 2000;
   if (NO_ERROR <> WNetEnumResource(hNetEnum, EntriesToGet,
                   @ResourceBuffer, ResourceBuf)) then
   begin
    WNetCloseEnum(hNetEnum);
    exit;
   end;
   for i := 1 to EntriesToGet do
    Memo1.Lines.Add(string(ResourceBuffer[i].lpRemoteName));
  end;
end;
```

Если вам листинг понятен, то можете заканчивать чтение этого раздела. Ну а если у вас возникли проблемы, то давайте разберем его подробнее.

В самом начале происходит заполнение структуры `NetContainerToOpen`, которая объявлена в разделе `var` как принадлежащая типу `NETRESOURCE`. У нее нужно заполнить следующие пять полей:

- `dwScope` — в этом параметре нужно указать рамки перечисляемых ресурсов. Я указал `RESOURCE_GLOBALNET`, чтобы поиск происходил в сети;

- `dwType` — здесь указывается тип перечисляемых ресурсов. Вы можете указать `RESOURCETYPE_DISK` для дисков, `RESOURCETYPE_PRINT` для принтеров и `RESOURCETYPE_ANY` для всего подряд;
- `lpLocalName` — этот параметр нужно обнулить;
- `lpRemoteName` — здесь нужно указать NetBIOS-имя сканируемого компьютера или IP-адрес. Если вы указываете адрес, то вначале нужно прибавить два обратных слэша `\\`, как видно из листинга;
- `lpProvider` — имя владельца ресурса. Если оно не известно, то нужно указать `nil`.

После заполнения структуры нужно открыть процесс сканирования. Для этого существует функция `WNetOpenEnum` со следующими пятью параметрами:

- область сканирования — здесь снова указываем `RESOURCE_GLOBALNET`;
- тип сканируемых ресурсов — снова указываем все подряд, `RESOURCETYPE_ANY`;
- здесь необходимо указать, какие ресурсы надо перечислять. Если нужно все подряд, то просто укажите `0`. Другие возможные значения: `RESOURCEUSAGE_CONNECTABLE` — подключаемые, и `RESOURCEUSAGE_CONTAINER` — хранимые;
- структура, которую мы заполнили;
- переменная типа `THandle`, которая будет использоваться в дальнейшем.

После того как мы открыли перечисление, можно смело приступать к его реализации. Для этого запускается бесконечный цикл:

```
while TRUE do
 begin
 end;
```

Внутри цикла постоянно вызывается функция `WNetEnumResource`. Если она возвращает ошибку (результат не равен `NO_ERROR`), то перечисление закрывается с помощью `WnetCloseEnum`, выходим из процедуры, потому что больше открытых ресурсов нет. У функции `WnetEnumResource` есть четыре параметра:

- в первом параметре нужно указать ту же переменную, которую мы задали в последнем параметре при открытии перечисления `WNetOpenEnum`;
- во втором параметре следует указать переменную, в которой хранится число необходимых к возврату ресурсов. В примере это переменная `EntriesToGet`, в которой записано число `2000`. После того как функция выполнится, в этой переменной будет не `2000`, а количество реально открытых ресурсов;

- в третьем параметре должен быть массив структур `TNetResource`. Его длина должна быть достаточной для хранения возвращенной информации об открытых ресурсах. В листинге запрашивается максимум 2000 ресурсов, значит, массив должен состоять из 2000 структур (`ResourceBuffer: array[1..2000] of TNetResource;`);
- в четвертом параметре задается размер массива, указанного в предыдущем параметре.

У функции `WnetCloseEnum` есть только один параметр, в котором мы должны указать ту же переменную, что писали в последнем параметре при открытии перечисления `WNetOpenEnum`.

Если перечисление прошло успешно, то мы можем вывести полученную информацию на экран. Для этого запустим цикл от 0 до количества возвращенных значений `EntriesToGet`:

```
for i := 1 to EntriesToGet do
 Memo1.Lines.Add(string(ResourceBuffer[i].lpRemoteName));
```

Внутри цикла добавляем в компонент `Memo1` строку, содержащую имя ресурса. Имя полученного открытого ресурса можно прочитать в переменной `lpRemoteName` структуры `ResourceBuffer[i]`. Единственное, что тут надо помнить: `ResourceBuffer[i].lpRemoteName` — это не строка, поэтому данный параметр надо превратить в строку. Для этого надо использовать функцию `String()`:

```
String(ResourceBuffer[i].lpRemoteName)
```

Итак, сканер расшаренных ресурсов готов, правда, он пока сканирует только одну указанную машину. Из-за этого использование данной программы в боевых условиях нереально. Но никто же не мешает вам дополнить программу перебором, ведь это не так уж и сложно. Результат можно увидеть на рис. 4.10.

Единственный недостаток такого алгоритма сканирования — слишком большая медлительность в работе. Это связано не с самим алгоритмом, а со скоростью исполнения используемых функций. Если сканируемого адреса нет в сети, то программа может потерять лишние 3—5 секунд в ненужных поисках. Это очень много, и это будет абсолютно бессмысленной потерей времени. Чтобы избавиться от этого недостатка, можно перед сканированием произвести операцию `ping` указанного IP-адреса. Пинг проходит достаточно быстро и сможет сказать нам, существует ли указанный адрес. Если он существует, то можно открывать перебор ресурсов, лежащих в свободном доступе, иначе нет смысла тратить время на бессмысленные поиски.

Примечание

На компакт-диске в каталоге \Примеры\Глава 4\Scan share\ вы можете найти пример этой программы.

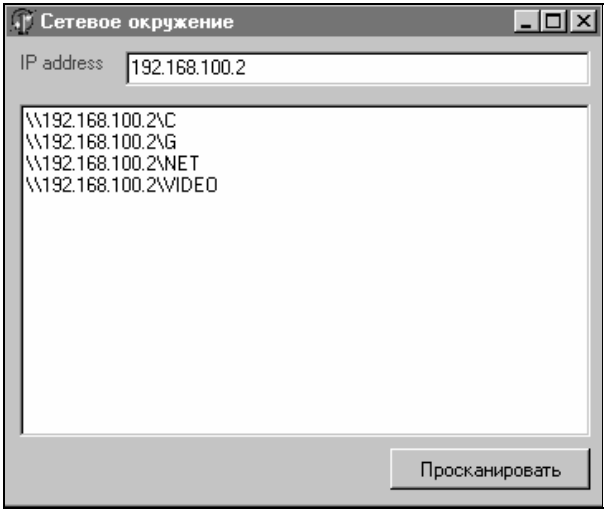


Рис. 4.10. Результат работы программы-сканера

4.5. Telnet-клиент

Теперь я хочу показать вам, как можно написать простейшего Telnet-клиента. С помощью такого клиента можно подключиться к какому-нибудь серверному порту и выполнять команды прямо на сервере. Например, если подключиться к порту Telnet, то можно запускать на сервере программы. Если подключиться к порту FTP, то можно выполнять команды FTP из командной строки.

Для тестирования нашего Telnet-клиента я буду использовать подключение к локальному FTP-серверу, который входит в поставку Windows 2000. В старых версиях Windows 9*x* можно использовать Personal Web Server, который отличается только настройками и меньшим количеством возможностей. Но об этом чуть позже.

Итак, создаем новый проект и делаем форму похожей на ту, что изображена на рис. 4.11. Здесь находятся две строки ввода: для ввода IP-адреса сервера, к которому мы хотим подключиться, и для ввода номера порта, который надо использовать. В единственном выпадающем списке `TComboBox` можно выбирать тип используемого терминала.

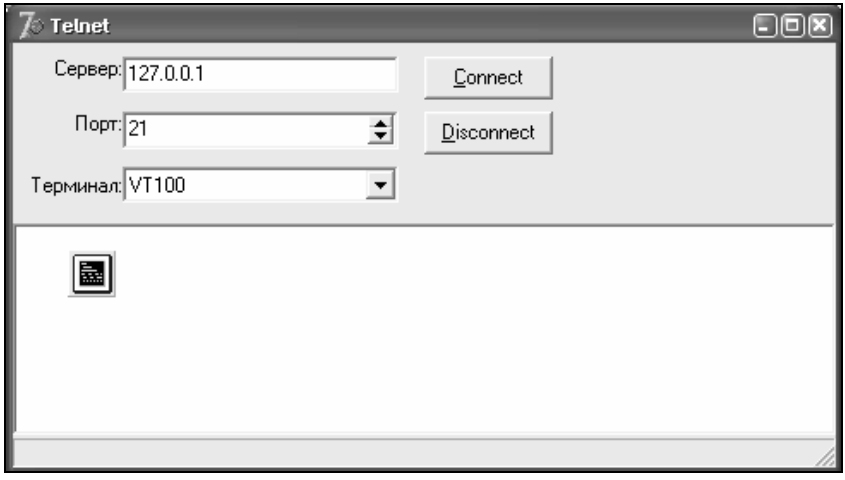


Рис. 4.11. Форма будущего Telnet-клиента

Помимо этого нам понадобятся две кнопки: **Connect** и **Disconnect** для подключения и отключения от сервера, и один компонент `TMemo`, в котором мы будем набирать команды, отправляемые серверу, и отображать результат выполнения команд.

Самым основным компонентом будет `IdTelnetDemo` со вкладки **Indy Clients** палитры компонентов. Его нужно просто перенести на форму, все его свойства оставим заданными по умолчанию.

В обработчике нажатия кнопки **Connect** пишем следующий код:

```
procedure TTelnetForm.ConnectButtonClick(Sender: TObject);
begin
 IdTelnet1.Terminal:=TerminalCB.Text;
 IDTelnet1.Host := EdtServer.Text;
 IDTelnet1.port := SpnedtPort.Value;
 IdTelnet1.Connect;
end;
```

В первых трех строках я устанавливаю свойства компонента `IdTelnet1`:

- `Terminal` — здесь указывается в виде текста тип используемого терминала;
- `Host` — адрес сервера, к которому будем подключаться;
- `Port` — номер порта, к которому нужно производить подключение.

Обработчик нажатия кнопки `Disconnect` еще проще:

```
procedure TTelnetForm.btnDisconnectClick(Sender: TObject);
begin
 IdTelnet1.Disconnect;
end;
```

Здесь просто вызывается метод `Disconnect` компонента `IdTelnet1`, что приводит к отключению от сервера.

Для отправки команд на сервер мы создадим обработчик события `OnKeyPress` для компонента `Memo1`. В этом обработчике напишем следующее:

```
procedure TTelnetForm.Memo1KeyPress(Sender: TObject;
  var Key: Char);
begin
  if IdTelnet1.Connected then
   begin
    IdTelnet1.SendCh(Key);
   end;
end;
```

Здесь происходит проверка, если компонент `IdTelnet1` подключен к серверу, то символ нажатой клавиши нужно передать на сервер. Для этого используется метод `SendCh` компонента `IdTelnet1`, а в качестве параметра этому методу передается нажатый символ. Теперь, когда мы нажимаем любую клавишу для ввода символа в компонент `Memo1`, этот символ моментально отправляется на сервер.

Теперь создадим обработчик события `OnConnected` для компонента `IdTelnet1`. В этом обработчике напишем следующий код:

```
procedure TTelnetForm.IdTelnet1Connected(Sender: TObject);
begin
 Memo1.Lines.Add('Клиент подключен.');
 Memo1.Lines.Add('Можете выполнять команды');
 Memo1.Lines.Add('');
end;
```

В принципе, вся работа этого обработчика направлена на то, чтобы проинформировать пользователя о том, что соединение произошло успешно. Я просто вывожу соответствующий текст в компонент `Memo1`, что придаст программе большую наглядность.

Теперь создадим обработчик события `OnDataAvailable` компонента `IdTelnet1`. Этот обработчик вызывается каждый раз, когда к нам с сервера поступили данные. В нем нужно написать код из листинга 4.2.

Листинг 4.2. Обработка данных, поступивших с сервера

```
procedure TTelnetForm.IdTelnet1DataAvailable(Sender: TIdTelnet;
  const Buffer: String);
const
  CR = #13;
  LF = #10;
var
  Start, Stop: Integer;
begin
 Memo1.Lines.Add('');
  Start := 1;
  Stop := Pos(CR, Buffer);
  if Stop = 0 then
    Stop := Length(Buffer) + 1;
  while Start <= Length(Buffer) do
  begin
    Memo1.Lines.Strings[Memo1.Lines.Count - 1] :=
      Memo1.Lines.Strings[Memo1.Lines.Count - 1] +
      Copy(Buffer, Start, Stop - Start);
    if Buffer[Stop] = CR then
    begin
      Memo1.Lines.Add('');
    end;
    Start := Stop + 1;
    if Start > Length(Buffer) then
      Break;
    if Buffer[Start] = LF then
      Start := Start + 1;
    Stop := Start;
    while (Buffer[Stop] <> CR) and (Stop <= Length(Buffer)) do
      Stop := Stop + 1;
  end;
end;
```

Весь написанный в листинге код направлен на вывод пришедшего текста с помощью компонента `Memo1`. Для этого ищутся символы конца строки и перевода каретки, и если они найдены, то в компонент добавляется новая строка. Для облегчения поиска заведены две константы `CR` и `LF` с шестнадцатеричными значениями `#13` и `#10`, которые являются кодами символов конца строки и перевода каретки.

На рис. 4.12 можно увидеть пример результата работы программы.

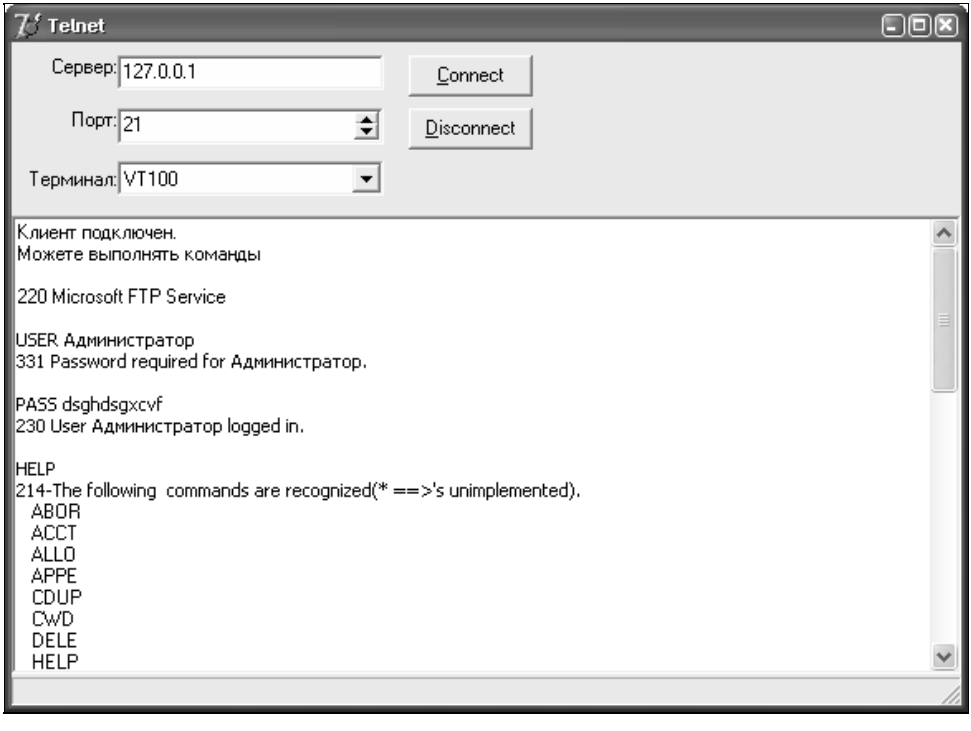


Рис. 4.12. Результат работы программы

Теперь поговорим о принципе тестирования. Для проверки работы программы я установил на компьютере сервер IIS. Если ваш компьютер работает под управлением Windows 2000/XP, то для установки этого сервера вам необходимо выполнить следующие действия.

1. Для начала необходимо вставить компакт-диск с Windows 2000 или Windows XP в устройство для чтения и запустить setup.exe или дождаться автозапуска. Передо мной открылось окно, как на рис. 4.13 (у вас оно может отличаться, но смысл будет тот же).

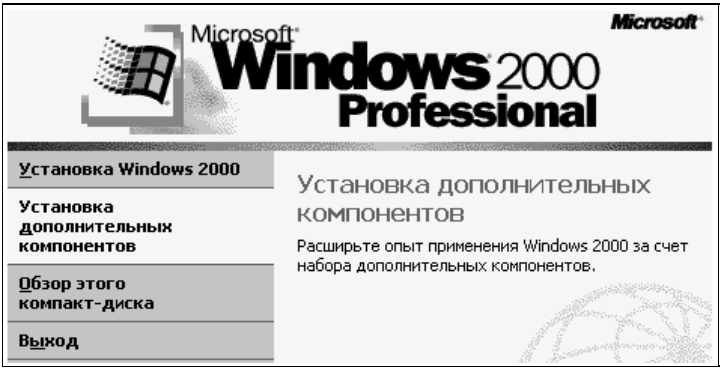


Рис. 4.13. Окно установки Windows 2000

2. Выберите пункт **Установка дополнительных компонентов** и вы окажетесь в окне, похожем на изображение на рис. 4.14. Можете выделить в нем все подряд, в хозяйстве пригодится (но только для обучения). Если вы устанавливаете компоненты на рабочий сетевой сервер, то ставьте только то, что нужно.

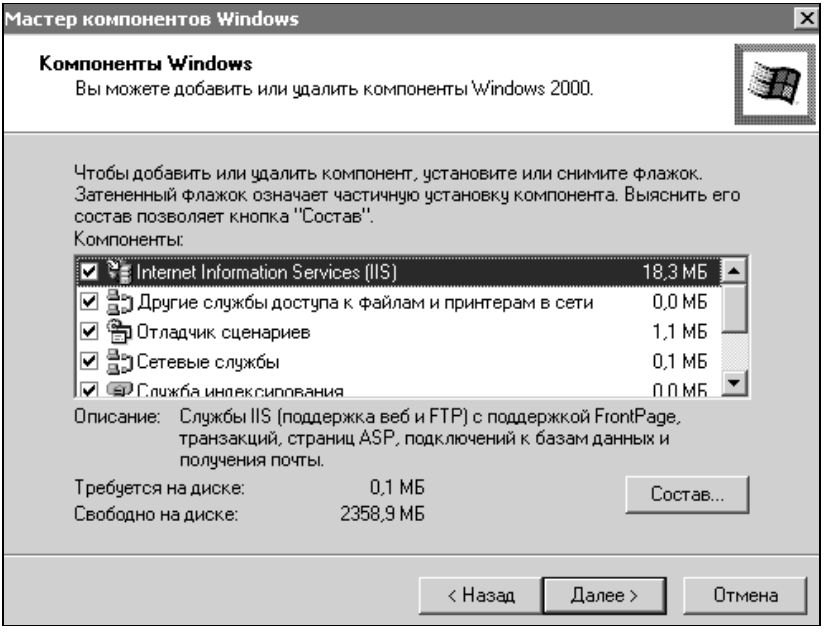


Рис. 4.14. Установка дополнительных компонентов

3. Самым важным для нас будет пункт **Internet Information Services (IIS)**. Можете дважды щелкнуть по нему, чтобы посмотреть состав IIS. Здесь вы увидите FTP-сервер, Web-сервер, документацию ("кривая", но все же) и т. д. Я оставил все, что есть, выбранным (это отнимет не так уж много места на диске) и нажал кнопку **Далее**.
4. После установки дополнительных сервисов нужно перейти в **Пуск | Панель управления | Администрирование | Управление компьютером**. Перед вами должно открыться окно, как на рис. 4.15. Если вы раскроете здесь ветку **Службы и приложения**, то сможете увидеть тут новый пункт **Internet Information Services**. Выделите его, и в правой половине окна появятся:
   - **FTP-узел по умолчанию**;
   - **Веб-узел по умолчанию**;
   - **Виртуальный SMTP-сервер по умолчанию**.

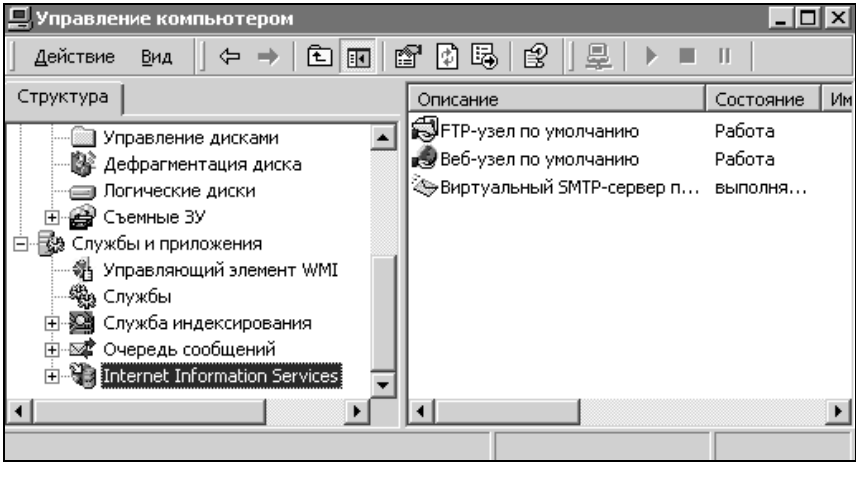


Рис. 4.15. Окно настройки IIS

Выделяя любой из этих сервисов, вы можете запускать, останавливать и приостанавливать любой из них с помощью соответствующих кнопок на панели инструментов.

5. Выберите пункт **FTP-узел по умолчанию** и щелкните на нем правой кнопкой мыши. В появившемся меню нужно выбрать пункт **Свойства**, и перед вами появится окно настроек FTP-сервера. На первой вкладке этого окна нужно обязательно выбрать в строке **IP-адрес** адрес вашего компьютера (адаптера), через который будет доступен сервер. Если в вашем компьютере две сетевые карты, то вы должны указать ту, через ко-

торую пользователи смогут получить доступ к серверу FTP. Если сетевых карт нет, то можно указать `127.0.0.1`.

На вкладке **Домашний каталог** нужно указать папку, с которой пользователи будут работать по протоколу FTP. Здесь нужно также указать вид доступа к папке.

Вот теперь перейдем к тестированию нашего Telnet-клиента. Для этого запустите его и подключитесь к своему компьютеру (IP-адрес 127.0.0.1) на 21-й порт, где по умолчанию должен ждать FTP-сервер. Как только в компоненте `Memo1` вы увидите сообщение об удачном соединении, введите в поле `Memo1` команду:

```
USER имя_пользователя
```

Нажмите клавишу <Enter>, и вы увидите ответ, в котором сервер сообщает вам, что для данного имени пользователя нужно указать пароль. Для ввода пароля выполните команду:

```
PASS пароль
```

Для того чтобы узнать, какие еще команды вы можете выполнять, можно выполнить команду:

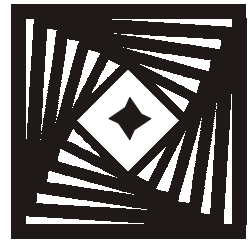
```
HELP
```

Вот и все, что я хотел рассказать про Telnet-клиент. Как видите, с его помощью вы можете соединяться с серверными программами и напрямую выполнять их команды. В этом примере я показал прямую работу с FTP-сервером, но вы можете таким образом потренироваться с 7-м портом, который выдает echo-ответы (просто возвращает все, что вы ему отослали), или с почтовым сервером, если вы знаете его команды.

Примечание

На компакт-диске в каталоге \Примеры\Глава 5\Telnet\ вы можете найти пример программы и цветные рисунки этого раздела.

Глава 5



Сеть на низком уровне

Под работой с сетью на низком уровне я буду понимать использование функций Windows библиотеки WinSock. Работать с нею сложнее, чем с компонентами, потому что многие вещи приходится делать вручную. Но, прочитав эту главу, вы увидите, что в сети нет абсолютно ничего страшного. Все очень просто и очень даже интересно.

Раньше я любил использовать компоненты, которые предоставляет нам Delphi, но потом понял, что в таком кодинге нет соединения с киберпространством. Именно поэтому в своих программах я перестал использовать сторонние компоненты, а стал писать все ручками. Ручной кодинг сетевых функций не намного дольше, но намного интереснее.

Если отказаться от использования готовых решений, то вы получаете максимальную гибкость и пластику, как у Волочковой. А это уже может дать вашей программе неоспоримые преимущества перед конкурентами (если вы занимаетесь коммерческой разработкой).

5.1. Инициализация WinSock

Библиотека Windows Sockets (WinSock) состоит из одного лишь файла Winsock.dll. Хотя нет, Winsock.dll не используется уже очень давно, потому что это 16-битный файл и оставлен в Windows только для совместимости с программами Windows 3.1. Интересно, у кого-то остались еще такие? Не пора ли очистить систему от этого мусора? Версия для 32-битных приложений находится в одном из двух файлов:

- wsock32.dll — содержит функции Winsock версии 1;
- ws2_32.dll — содержит функции Winsock версии 2.

Все эти файлы можно найти в каталоге Windows\System32.

Функции Windows Sockets очень хорошо подходят для создания простых приложений, потому что в этой библиотеке реализовано все необходимое для создания соединения и приема/передачи файлов. Зато снифер (программу, прослушивающую трафик) создавать даже не пытайтесь. В WinSock нет ничего для доступа к заголовкам пакетов. Microsoft обещала встроить эти необходимые продвинутому программисту вещи в WinSock2, но как всегда все закончилось только словами.

Чем хороша эта библиотека, так это тем, что все ее функции одинаковы для многих платформ и языков программирования. Так, например, когда мы напишем сканер портов, его легко можно будет перенести на язык C/C++ и даже написать что-то похожее в UNIX-подобных системах, потому что там сетевые функции называются так же и имеют практически те же параметры. Разница между сетевой библиотекой Windows и Linux минимальна, хотя и есть. Но так и должно быть, ведь Microsoft не может по-человечески, и их программистам обязательно надо выделиться.

Сразу же предупрежу, что мы будем изучать WinSock2, а Delphi поддерживает только первую версию. Чтобы она смогла увидеть вторую, нужно подключить заголовочные файлы для этой версии. На компакт-диске, прикладываемом к книге, вы можете найти нужные файлы в каталоге Headers\Winsock2.

Вся работа сетевой библиотеки построена вокруг понятия "сокет" — это как бы виртуальный сетевой канал. Для соединения с сервером вы должны подготовить такой канал к работе и потом можете соединяться с любым портом сервера. Лучше всего увидеть это на практике, но я попробую дать вам сейчас общий алгоритм работы с сокетами.

1. Инициализируем библиотеку WinSock.
2. Инициализируем сокет (канал для связи). После инициализации у нас должна быть переменная, указывающая на новый канал. Созданный сокет — это, можно сказать, открытый порт на вашем компьютере. Порты есть не только на сервере, но и у клиента, и когда происходит передача данных между компьютерами, то она происходит между сетевыми портами.
3. Можно присоединяться к серверу. В каждой функции для работы с сетью первым параметром обязательно указывается переменная, указывающая на созданный канал, через который будет происходить соединение.

Самое первое, что надо сделать, — инициализировать сетевую библиотеку (для UNIX-подобных ОС это не нужно делать. Для этого необходимо вызвать функцию `WSAStartup`. У нее есть два параметра:

- версия WinSock, которую мы хотим использовать. Для версии 1.0 нужно указать `MAKEWORD(1,0)`, но нам нужна вторая, значит, будем указывать `MAKEWORD(2,0)`;

- структура типа `TWSADATA`, в которой будет возвращена информация о найденном WinSock.

Если функция отработала успешно, то результатом работы функции будет 0. Иначе результатом является код ошибки, а ошибки бывают следующие:

- `WSASYSNOTREADY` — базовая сетевая система не готова к сетевому взаимодействию;
- `WSAVERNOTSUPPORTED` — запрашиваемая версия не поддерживается;
- `WSAEINPROGRESS` — выполняется блокирующая операция WinSock 1.1;
- `WSAEPROCLIM` — достигнут максимальный предел выполнений;
- `WSAEFAULT` — структура `TWSADATA`, которую вы передали во втором параметре, является некорректным указателем.

На моей практике при корректно настроенном протоколе TCP/IP ошибок при инициализации никогда не было.

Теперь узнаем, как нужно закрывать библиотеку. Для этого следует вызвать функцию `WSACleanup`, у которой нет параметров. В принципе, если вы не закроете WinSock, то ничего критического не произойдет. После выхода из программы все само закроется, просто освобождение ненужного сразу после использования является хорошим тоном в программировании.

Инициализация необходима не перед каждым использованием сокетов, а один раз в течение работы программы. Поэтому нет необходимости вызывать `WSAStartup` в каждом модуле, достаточно сделать это один раз при загрузке программы и освободить библиотеку с помощью `WSACleanup` при выходе из программы.

5.1.1. Пример инициализации

Давайте сразу напишем пример, который будет инициализировать WinSock и выводить на экран информацию о нем. Создайте в Delphi новый проект. Теперь к нему надо подключить заголовочные файлы WinSock второй версии. Для этого надо перейти в раздел `uses` и добавить туда модуль `Winsock2`.

Если вы попробуете сейчас скомпилировать этот пустой проект, то Delphi будет ругаться на добавленный модуль. Это потому, что она не может найти сами файлы. Тут можно поступить несколькими способами.

5.1.2. Подключение заголовочных файлов

Итак, выполните следующие действия:

1. Сохраните новый проект в какой-нибудь каталог и туда же скопируйте файлы winSock2.pas, ws2tcpip.inc, wsipx.inc, wsnwlink.inc и wsnetbs.inc. Не-

удобство этого способа — в каждый проект, использующий WinSock2, надо забрасывать заголовочные файлы.

2. Можно поместить эти файлы в папку Delphi\Lib, и тогда уж точно любой проект найдет их.
3. Можно положить файлы в отдельный каталог, а затем подключить его к Delphi. На всякий случай коротко напомню, как это делается, потому что я считаю этот способ наиболее удобным.

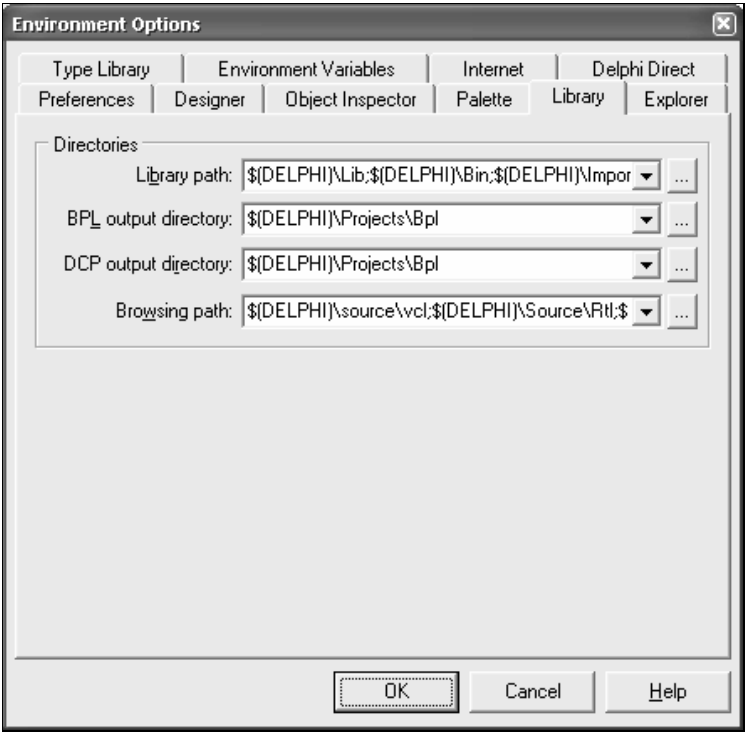


Рис. 5.1. Окно настроек **Environment Options**

Чтобы подключить каталог с заголовочными файлами в Delphi, нужно выбрать в меню **Tools** пункт **Environment options**. В появившемся окне (рис. 5.1) нужно перейти на вкладку **Library**. Здесь происходит настройка путей, с которыми работает среда разработки Delphi. Нас интересует первый выпадающий список — **Library path**. В нем перечислены пути к каталогам с заголовочными файлами и исходными кодами компонентов. Вот сюда и нужно добавить путь к каталогу с файлами WinSock2.

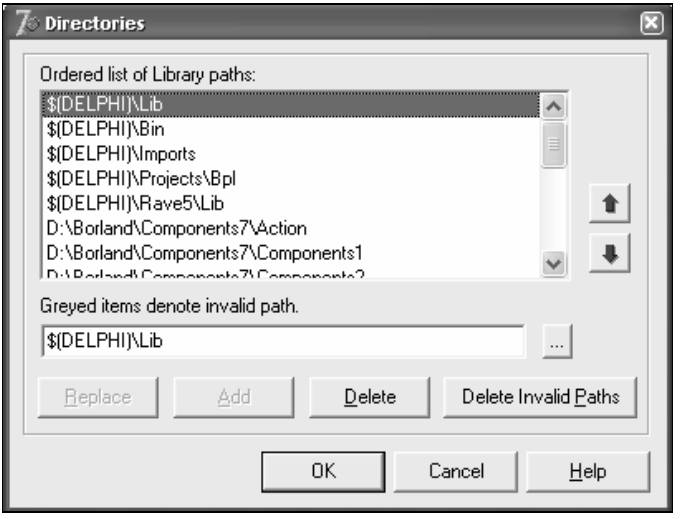


Рис. 5.2. Окно настроек путей к каталогам

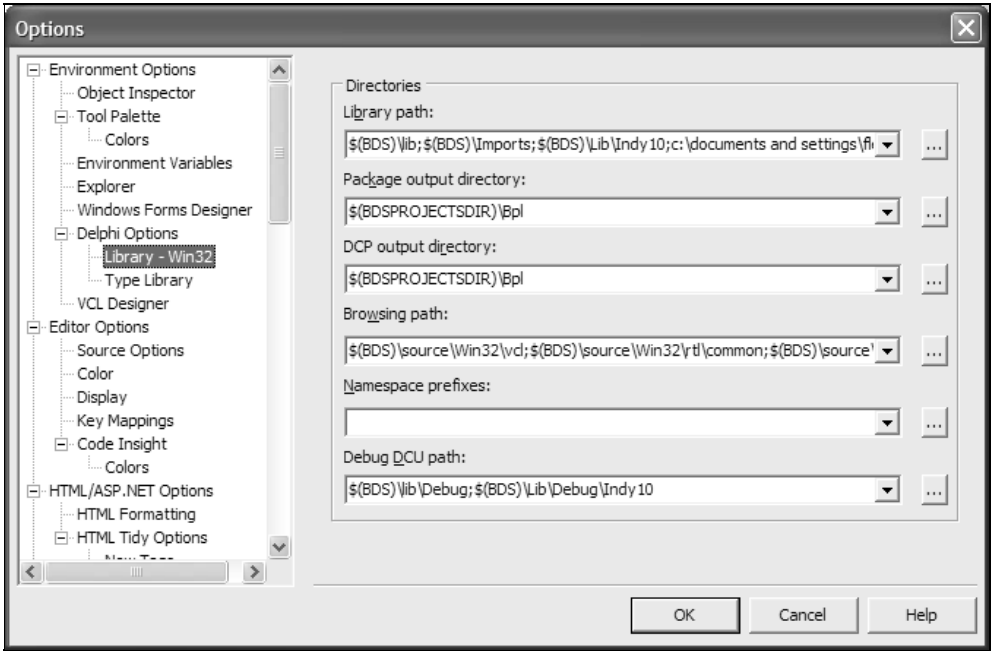


Рис. 5.3. Окно настроек в Delphi 2006

Для этого опять есть два способа:

- добавить ручками в конец строки точку с запятой ";" и после этого написать полный путь к файлам;
- щелкнуть на кнопке с тремя точками справа от строки ввода и перед вами откроется окно, как на рис. 5.2.

  Здесь сверху находится список каталогов и под ним строка ввода для нового пути. Справа от строки ввода опять есть кнопка с тремя точками. Если щелкнуть на этой кнопке, то увидите стандартное окно выбора папки. Найдите нужную папку с заголовочными файлами и нажмите кнопку **OK**. Теперь в строке ввода должен отображаться полный путь к нужному каталогу. Чтобы добавить его, нужно щелкнуть на кнопке **Add**.

Все, путь добавлен, и можно закрывать все окна нажатием кнопок **OK**.

В Delphi 2006 список настроек немного изменился. Чтобы вызвать его, выберите в меню **Tools** пункт **Options**. Слева в окне находится дерево категорий настроек, а справа выводятся параметры для данного раздела. Перейдите в дереве в раздел **Environment Options | Delphi Options | Library - Win32** (рис. 5.3).

5.1.3. Получение информации о сокетах

Вот теперь попробуем инициализировать библиотеку WinSock. Для этого перенесите на созданную нами форму три строки ввода и кнопку. После этого создайте обработчик события `OnClick` для кнопки и напишите там следующий текст:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 info:TWSADATA;
begin
 WSAStartup(MAKEWORD(2,0), info);
 VersionEdit.Text:=IntToStr(info.wVersion);
 DescriptionEdit.Text:=info.szDescription;
 SystemStatusEdit.Text:=info.szSystemStatus;
 WSACleanup;
end;
```

В самом начале запускается WinSock с помощью вызова функции `WSAStartup`. В нем запрашивается вторая версия, а информация о текущем состоянии будет возвращена в структуру `info`. После этого выводим полученную информацию из структуры в главное окно программы (рис. 5.4).

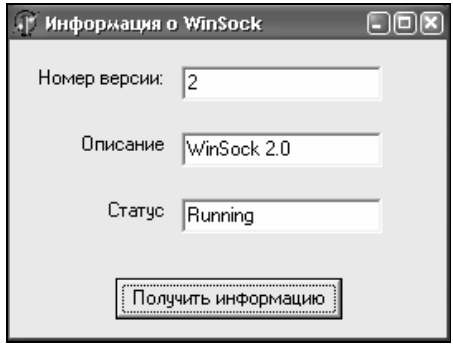


Рис. 5.4. Результат работы программы

Примечание

На компакт-диске в каталоге \Примеры\Глава 5\Initialize\ вы можете найти пример этой программы.

5.2. Обработка сетевых ошибок

Большинство сетевых функций, которые мы будем рассматривать в этой главе, в случае удачного выполнения возвращают какое-то определенное значение, а в случае ошибки возвращают −1. Есть и исключения, как, например, в *разд. 5.1.3*, где функции инициализации и закрытия сетевой библиотеки сразу же возвращали код ошибки.

Есть еще одно небольшое исключение — функция `socket` возвращает созданный сокет, который имеет тип беззнакового целого числа. Как же тогда вернуть −1 в случае ошибки? Для этого завели специальную константу ошибки `INVALID_SOCKET`, остальные же функции возвращают `SOCKET_ERROR`.

Чтобы узнать подробную информацию о происшедшей ошибке в последней сетевой функции, используется `WSAGetLastError`. Если мы увидели, что какая-то функция отработала некорректно, то мы можем вызвать `WSAGetLastError`. Эта функция не получает никаких параметров, а возвращает код последней ошибки для данного процесса.

Допустим, что мы выполнили функцию А и она отработала с ошибкой. Мы не стали проверять состояние, а выполнили функцию Б, которая также завершилась неудачей. Теперь если вызвать `WSAGetLastError`, то мы увидим код ошибки Б, потому что она выполнялась последней, а код ошибки А будет недоступен.

Проверять на ошибку желательно немедленно. Некоторые сетевые функции обнулят ошибку даже в случае удачного выполнения. На примере с функциями А и Б это означает, что даже если Б выполнится корректно, результат ошибки А может исчезнуть.

Теперь давайте рассмотрим, какие коды ошибок могут возвращать сетевые функции:

- `WSANOTINITIALIZED` — необходимо предварительно вызвать `WSAStartup`;
- `WSAENETDOWN` — сетевая система или связанный сервис-провайдер не доступен;
- `WSAEAFNOSUPPORT` — указанная система адресации не поддерживается;
- `WSAEINPROGRESS` — в данный момент выполняется блокирующая операция Windows Socket 1.1;
- `WSAEMFILE` — нет больше доступных сетевых дескрипторов;
- `WSAENOBUFS` — нет доступной памяти;
- `WSAEPROTONOSUPPORT` — указанный протокол не поддерживается;
- `WSAEPROTOTYPE` — указанный протокол не поддерживается данным типом сокета;
- `WSAESOCKTNOSUPPORT` — указанный тип сокета не поддерживается для данного семейства адресации.

Констант очень много, но каждая функция может генерировать далеко не все из этих ошибок. Ради экономии места я собрал все ошибки здесь, а когда мы будем рассматривать сами сетевые функции, я буду только показывать константы, которые могут быть возвращены для данной функции.

При разработке сетевых приложений нужно быть очень аккуратными, потому что от этого может зависеть безопасность компьютера. Я бы посоветовал не стесняться и проверять каждый вызов сетевой функции на ошибки. Если что-то пропустить и вызвать другую функцию с заведомо некорректными данными, то это может завершиться для компьютера очень плачевно.

Чтобы проще было обрабатывать ошибки, можно написать универсальную функцию, которая будет получать код последней ошибки и выводить соответствующее сообщение. Подобная функция может выглядеть примерно следующим образом:

```
procedure ShowWinSockError(FuncName:String);
var
 sErrorString:String;
begin
 sErrorString:='Неизвестная ошибка';
```

```
 case WSAGetLastError() of
  WSAENETDOWN: sErrorString:='Проблема с сетью';
  WSAEADDRINUSE: sErrorString:='Указанный адрес уже используется';
  WSAEPROTONOSUPPORT: sErrorString:='Протокол не поддерживается';
  WSAEINVAL: sErrorString:='Сокет уже связан с адресом';
  WSAENOBUFS: sErrorString:='Недостаточно памяти';
  WSAENOTSOCK: sErrorString:='Неверный дескриптор сокета';
  WSAEISCONN: sErrorString:='Сокет уже подключен';
  WSAEMFILE: sErrorString:='Больше нет доступных дескрипторов';
 end;

 MessageBox(0, PChar('Ошибка в функции '+FuncName+
    ' - '+sErrorString), 'Ошибка', 0);
end;
```

В качестве параметра функция `ShowWinSockError` получает строку, где нужно передать имя функции, которая привела к ошибке.

Теперь вызов сетевых функций может выглядеть следующим образом:

```
if socket(AF_INET, SOCK_STREAM, IPPROTO_IP)=INVALID_SOCKET then
  begin
   ShowWinSockError('socket');
   exit;
  end;
// Здесь могут быть другие сетевые функции
```

Здесь вызывается функция `socket`, которую мы рассмотрим в *разд. 5.3*. Если она вернула ошибку (результат равен `INVALID_SOCKET`), то вызывается функция `ShowWinSockError` для получения более подробной информации, и работа прерывается. В зависимости от логики приложения, вы должны корректно отрабатывать каждую ошибку.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 5\TestOnError\ вы можете найти пример этой программы.

5.3. Функции соединения с сервером

Прежде чем производить соединение с сервером, надо еще подготовить сокет к работе. Этим и займемся.

Для подготовки нужно выполнить функцию socket, у которой есть три параметра.

- Тип используемой адресации. Нас интересует Интернет, поэтому мы будем указывать PF_INET или AF_INET. Как видите, оба значения очень похожи и показывают одну и ту же адресацию. Я бы даже сказал больше: значения идентичны, потому что в файле winsock2 есть строка объявления: PF_INET = AF_INET. Так что не имеет значения, какую из констант использовать.
- Базовый протокол. Здесь мы должны указать, на основе какого протокола будет происходить работа. Если вы прочитали документы о сетях на компакт-диске или *главу 4*, то должны знать, что существуют два базовых протокола: TCP (с надежным соединением) и UDP (не производящий соединений). Для TCP в этом параметре надо указать SOCK_STREAM, а если вам нужен протокол UDP, то указывайте SOCK_DGRAM.
- Вот здесь мы можем указывать, конкретно какой протокол нас интересует. Возможных значений тут очень много (например, IPPROTO_IP, IPPORT_ECHO, IPPORT_FTP и т. д.). Если хотите увидеть все, то открывайте файл winsock2.pas и запускайте поиск по "IPPORT_", и все, что вы найдете, и будет возможными протоколами. Самое интересное, что бы вы не указали, работа сокета изменится не сильно.

Функция socket пока не устанавливает никаких соединений, но возвращает дескриптор созданного сокета и выделяет все необходимые ресурсы.

Если сокет был удачно создан, то результатом будет дескриптор нового сокета, иначе результатом будет константа INVALID_SOCKET. Чтобы более подробно узнать, какая именно произошла ошибка, следует вызвать функцию WSAGetLastError, которая может вернуть одно из следующих значений: WSANOTINITIALISED, WSAENETDOWN, WSAEAFNOSUPPORT, WSAEINPROGRESS, WSAEMFILE, WSAENOBUFS, WSAEPROTONOSUPPORT, WSAEPROTOTYPE, WSAESOCKTNOSUPPORT.

5.3.1. Синхронность/асинхронность работы порта

Теперь я хочу вас познакомить с синхронностью и асинхронностью работы порта. Разница в этих двух режимах следующая.

Синхронная работа: когда вы вызываете функцию, то программа останавливается и ждет полного ее выполнения. Допустим, что вы запросили соединение с сервером. Программа будет ждать, пока не произойдет реальное соединение или ошибка.

Асинхронная работа: в этом режиме программа не спотыкается о каждую сетевую функцию. Допустим, что вы сделали все тот же запрос на соединение с сервером. Ваша программа посылает запрос на соединение и тут же продолжает выполнять следующие действия, не дожидаясь физического контакта с сервером. Это очень удобно (но немного тяжелее в программировании), потому что можно использовать время ожидания контакта в своих целях и выполнить какие-то действия. Единственное, что вы не можете делать, — вызывать сетевые функции, пока не произойдет реальное физическое соединение. Недостаток в том, что программисту самому приходится следить за тем, когда закончится выполнение функции и можно будет дальше работать с сетью.

По умолчанию сокеты работают в синхронном режиме. В Windows еще можно встретить понятие "блокирующий режим", т. е. программа блокирует свое выполнение, пока функция не завершит выполнение.

5.3.2. Соединение с сервером

После вызова функции `socket` мы получаем готовый к употреблению сокет. Теперь мы можем соединяться с сервером. Для этого в библиотеке WinSock есть функция `connect`. Эта функция выглядит следующим образом:

```
function connect(
 const s: TSocket;
 const name: PSockAddr;
 namelen: Integer
): Integer; stdcall;
```

Здесь у нас три параметра:

- переменная-сокет, которую мы получили после вызова функции `socket`;
- структура типа `TSockAddr`;
- размер структуры, указанной во втором параметре. Для того чтобы узнать размер, можно воспользоваться функцией `SizeOf` и указать в качестве параметра структуру.

Структура `TSockAddr` очень сложная, и описывать ее полностью нет смысла. Дело в том, что она хранит адрес компьютера, с которым мы хотим подключиться. В зависимости от протокола может отличаться и адресация, поэтому мы рассмотрим структуру только с точки зрения самого популярного протокола — TCP/IP:

- `sin_family` — семейство используемой адресации. Здесь нужно указывать то же, что указывали в первом параметре при создании сокета. Мы

работаем с протоколом TCP/IP, а значит, указываем здесь `PF_INET` или `AF_INET`;

- `sin_addr` — адрес сервера, к которому мы хотим присоединиться;
- `sin_port` — порт, к которому мы хотим подключиться.

На деле это будет выглядеть так:

```
var
 addr: TSockAddr;
begin
 addr.sin_family := AF_INET;
 addr.sin_addr := ServerName;
 addr.sin_port := htons(21);
 connect(FSocket, @addr, sizeof(addr));
end;
```

В данном случае подразумеваем, что переменная `ServerName` содержит корректный адрес сервера.

5.3.3. Порты

Не забываем, что порт — это число, в котором порядок битов отличается от общепринятого. Чтобы преобразовать число в TCP/IP, используется функция `htons`, которая выглядит следующим образом:

```
function htons(
 hostshort: u_short
): u_short; stdcall;
```

В качестве параметра она получает короткое число без знака (16 битов) в стандартном порядке байтов, а в результате возвращает нам такое же число, где байты переставлены.

Сетевые порты имеют размер в 16 битов, а вот сетевой адрес в IPv4 занимает 32 бита, и его 4 байта также идут в не совсем привычной для нас последовательности. Для преобразования длинного целого числа из 32 битов используется функция `htonl`.

Для обратного преобразования применяется функция `ntohs`:

```
function ntohs(
 netshort: u_short
): u_short; stdcall;
```

Здесь в качестве параметра мы передаем число, в котором байты идут в порядке, необходимом TCP/IP, а в результате получаем число, в порядке, пригодном для Windows.

5.3.4. Закрытие соединения

Следующая функция, которую мы рассмотрим, закрывает соединение и называется `closesocket`:

```
function closesocket(
const s: TSocket
): Integer; stdcall;
```

В качестве параметра нужно указать переменную-сокет.

5.4. Сканер портов

Мы уже изучили достаточно теории, чтобы написать еще один пример и закрепить полученные знания. На реальном примере мы вдобавок узнаем, как можно определять адрес по имени.

Как сканируются порты? Для понимания этого нужно представлять процесс соединения двух компьютеров по протоколу TCP/IP. Когда двое в сети хотят пообщаться через сетевой кабель, то один из них посылает другому запрос с номером порта, на котором должно произойти соединение. По этому порту другая сторона определяет, к какой программе хотят подключиться. Если какое-то приложение действительно открыло нужный порт и ожидает соединения, то запрашиваемый получит ответ об успешности попытки. Все проверки пароля и прочие защитные механизмы проходят уже после соединения с портом удаленного компьютера, поэтому мы можем произвести соединение и узнать о доступности порта, даже если программа требует пароля.

Но сканер все же может показать нам, что порт закрыт, но реально на удаленном компьютере данный порт открыт какой-либо программой. Как это может быть? Ответ прост — это сетевой экран перекрывает нам доступ к требуемому порту. Когда мы посылаем запрос на соединение с сервером, сетевой экран проверяет свои правила, и если соединение с данным портом разрешено, то мы сможем увидеть, что порт открыт. Если правилами сетевого экрана запрещено соединяться с портом, то запрос будет отклонен. Сервис на сервере даже не узнает о нашей попытке, потому что она будет

отклонена на уровне драйвера сетевым экраном, а мы будем ошибочно думать, что порт закрыт и там ничего не работает. Но обход сетевых экранов выходит за рамки книги, поэтому эту тему мы отложим.

Теперь перейдем от слов к делу. Запустите Delphi. Перенесите на форму следующие компоненты:

- одну кнопку (имя `Button1`);
- два компонента `TLabel`;
- два компонента `TEdit`;
- компонент типа `TMemo`.

Два компонента `TEdit` необходимы нам для того, чтобы пользователь мог вводить первый и последний порт, т. е. указывать диапазон сканирования. Поле для ввода начального порта назовем `edStart`, а для конечного — `edEnd`. В `Memo`-поле будем выводить тестовое сообщение, когда соединение пройдет удачно, т. е. удаленный порт открыт.

Сканирование будет происходить по нажатию единственной кнопки на нашей форме. Создайте обработчик события `OnClick` и напишите в нем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 stClient:TSocket;
 localaddr : sockaddr_in;
 index:Integer;
begin
 Memo1.Clear;
 stClient := socket(AF_INET, SOCK_STREAM, 0);
 if stClient = INVALID_SOCKET then
  exit;

 localaddr.sin_addr := LookupName(edServer.Text);
 localaddr.sin_family := AF_INET;
 for index := StrToIntDef(edStart.Text, 1) to
                StrToIntDef(edEnd.Text, 1) do
  begin
   localaddr.sin_port := htons(index);
   if connect(stClient, @localaddr, sizeof(localaddr))=0 then
```

```
    begin
     CloseSocket(stClient);
     Memo1.Lines.Add('Порт '+IntToStr(index));
    end;
  end;
end;
```

Сначала создаем сокет. Мы будем использовать один сокет для соединения со всеми портами из указанного пользователем диапазона. Создавать для каждого подключения свой сокет не имеет смысла. Это необходимо только тогда, когда несколько соединений должны быть активными одновременно.

При создании сокета (вызове функции `socket`) указываем параметры:

- `AF_INET` — будем использовать IP-протокол;
- `SOCK_STREAM` — будем использовать протокол с установкой соединения (TCP). У UDP нет возможности подключения, а значит, мы не сможем проверить, открыт ли какой-либо порт;
- 0 — это значение соответствует константе `IPPROTO_IP`, т. е. IP-протоколу.

Теперь заполняем структуру `sockaddr_in`, в которой можно указать адрес компьютера, подвергающегося сканированию, и семейство протоколов. Мы будем последовательно подключаться к множеству портов, а значит, поле `sin_port` пока не заполняем. Его будем изменять в цикле.

Чтобы заполнить поле `sin_addr` структуры `sockaddr_in`, нужно указанный пользователем адрес привести к определенному формату. Для этого я вызываю функцию `LookupName`, которой передается текстовое имя сервера. Функция написана мной, и мы увидим ее чуть позже, но сейчас я могу сказать, что она универсальна и может принимать в качестве параметра как IP-адрес, так и текстовое имя компьютера.

Далее идет цикл сканирования. Внутри него сначала мы должны указать порт, к которому нужно подключиться, при этом конвертировать его в нужный формат с помощью `htons`.

Теперь вызываем функцию `connect` для подключения к серверу. Если результат работы функции равен нулю, то подключение произошло удачно. Можно выводить соответствующее сообщение на экран, но для продолжения работы необходимо еще закрыть текущее соединение с помощью `CloseSocket`, иначе следующая попытка соединения будет заведомо неудачной. Если соединения не было, то закрывать нечего, поэтому в данном случае я не вызываю функцию `CloseSocket`.

Теперь пришла пора познакомиться с функцией `LookupName`, код которой вы можете увидеть в листинге 5.1.

Листинг 5.1. Функция перевода адреса в нужный формат

```
function LookupName(name:String): TInAddr;
var
 HostEnt: PHostEnt;
 InAddr: TInAddr;
begin
 if name[4]='.' then
  InAddr.s_addr := inet_addr(PChar(name))
 else
  begin
  HostEnt := gethostbyname(PChar(name));
  FillChar(InAddr, SizeOf(InAddr), 0);
  if HostEnt <> nil then
   begin
    with InAddr, HostEnt^ do
     begin
      S_un_b.s_b1 := h_addr^[0];
      S_un_b.s_b2 := h_addr^[1];
      S_un_b.s_b3 := h_addr^[2];
      S_un_b.s_b4 := h_addr^[3];
     end;
   end
  end;
  Result := InAddr;
end;
```

В самом начале проверяется введенный пользователем текст. Если в тексте есть точка, то я считаю, что введен IP-адрес машины, например, `127.1.1.2`. В этом случае следует просто преобразовать его в нужный формат с помощью функции `inet_addr`.

Если точки нет, то введено имя машины. В этом случае запускается функция `gethostbyname`, которая получает IP-адрес по имени машины, и потом полученный адрес переводим в нужный формат. Эта функция выполняется небыстро, поэтому я и отсекаю IP-адреса по точке, чтобы если пользователь ввел IP, то сканирование начиналось как можно скорее.

Конечно, такая проверка может быть некорректной. Дело в том, что IP-адрес может выглядеть как 10.1.1.1, и это вполне корректный IP-адрес, но точка находится в третьей позиции. С другой стороны, пользователь может ввести адрес **abc.ru**, и здесь уже другая проблема — это не IP-адрес. Поэтому в реальных приложениях никогда не делайте такой проверки, а я ее оставил только для ускорения перевода текстового представления IP-адреса в нужный формат, потому что я привык работать с IP-адресами, а не именами.

Вот теперь рассмотрение примера можно считать законченным. Кстати, чтобы пример заработал, не забудьте добавить где-нибудь инициализацию WinSock. Я это сделал в своем примере по событию `OnCreate` для формы. Теперь можете запускать пример и тестировать. Только не советую указывать большой диапазон, потому что он будет сканироваться очень долго. Дело в том, что каждая попытка соединения отнимает очень много времени, и 10 портов будут сканироваться более минуты.

> ***ПРИМЕЧАНИЕ***
>
> На компакт-диске в каталоге \Примеры\Глава 5\ScanPort\ вы можете найти пример быстрого сканера.

5.5. Самый быстрый сканер портов

Сканер, который мы написали в *разд. 5.4*, очень медленный, и для проверки 1000 портов понадобится немало времени. Чтобы написать быстрый сканер, нужно намного больше знаний, чем мы уже имеем, поэтому я сначала не хотел затрагивать эту тему сейчас. Но потом решил, что это необходимо. Мы увидим интересный пример, а заодно познакомимся с целым вагоном хороших функций.

Алгоритм быстрого сканирования достаточно прост, но многие программисты с неохотой выдают хоть какую-нибудь информацию о нем. Я спрашивал представителей нескольких фирм, рекламирующих свои сканеры как самые быстрые, и никто не ответил на мои вопросы. А меня же просто интересовало, действительно ли сканер быстрый, или это только реклама.

Такая секретность достаточно очевидна, ведь если снять тайну с алгоритма, то они не смогут заколачивать деньги. Зайдите на **download.com** и посмотрите, сколько денег просят за быстрый сканер. Любой познавший эту тайну сразу пишет свою программу, пичкает какими-нибудь никому ненужными дополнительными возможностями и продает по 10—15 у. е. за программу. Вроде не так уж много, но небольшой капитал заработать можно.

Примечание

Алгоритм быстрого сканирования сложно было найти на момент написания первого издания книги. Сейчас уже много бесплатных программ и сканеров с открытым кодом, но и платные варианты иногда появляются. Если вы решите зарабатывать деньги на сканере портов, то приготовьтесь к лишней трате времени. На данный момент мало кто согласиться платить за это деньги.

Для начала посмотрим на то, как программисты неправильно пытаются увеличить скорость сканирования портов. Для этого многие пытаются использовать преимущества многозадачности Windows, запускают кучу потоков и в каждом из них делают попытку соединиться со своим портом. Оригинально, но это напрягает ОС и компьютер, да и увеличение в скорости получается незначительное.

Если работать с сетью в синхронном режиме, то получаются большие накладные расходы на ожидание соединения, и для реального увеличения скорости нужно запускать 30—40 потоков. Это очень неудобно и усложняет программу, а значит, она будет нестабильной и неудобной.

Для нормального сканирования не надо никаких дополнительных потоков. Тут нужно воспользоваться возможностями асинхронности сетевых функций. Это на первый взгляд выглядит сложнее в программировании, но реально такой сканер будет содержать намного меньше кода (максимум 40 строчек), а главное — реальный выигрыш в скорости и реальная параллельность сканирования. Напомню, что при синхронном режиме ОС останавливается на каждой функции и ожидает ее окончательного исполнения. При асинхронном режиме функция выполняется и не ждет ожидания ответа от сервера.

Рассмотрим реальный пример асинхронности. Для нашего сканера нужна только одна функция — `connect`. Когда мы вызываем ее в асинхронном режиме, то ОС посылает запрос на соединение серверу и, не дожидаясь реального коннекта, продолжает работать дальше, как ни в чем не бывало. Если мы шлем запрос на соединение в асинхронном режиме, то пока сервер решает, какой прислать ответ, можно послать еще кучу подобных запросов на соединение с другими портами. Потом нужно только подождать немного, чтобы сервер успел обработать все наши запросы и проверить результат.

Единственный недостаток асинхронности — надо самому проверять результат работы функции `connect`. Но это не так уж и сложно, и вы убедитесь в этом, когда увидите исходный код сканера.

Давайте теперь на бумаге прикинем, как же будет работать быстрый сканер:

1. Надо объявить кучу переменных типа `TSOCKET`, а лучше объявить массив таких переменных, например:

```
FSocket: array [0..39] of TSOCKET;
```

В этом примере я объявил 40 сокетов, значит, можно сканировать сразу по 40 портов.

2. Каждому сокету из массива назначить свой номер порта и выполнить функцию `connect`. Первый сокет будет пытаться присоединиться к 1-му порту, 2-й — ко второму и т. д.
3. Добавить все сокеты в специальный контейнер сетевых событий.
4. Запустить ожидание события.
5. Если произойдет какое-нибудь событие, значит, произошло соединение.
6. Вывести информацию об открытых портах и закрыть все сокеты.
7. Можно перейти на первый шаг и запустить сканирование следующей партии портов.

В этом случае сканирование происходит пачками по несколько портов сразу без использования каких-либо дополнительных потоков. Это позволяет не только выиграть в скорости, но и сильно разгрузить систему, лишив ее проблем с обработкой многих потоков.

Библиотека WinSock тоже может работать через события. Для этого нужно создать объект-событие с помощью функции `WSACreateEvent`. После этого добавить к нему сокеты, от которых нужно ожидать события с помощью функции `WSAEventSelect`. Потом следует дождаться события от любого из указанных сокетов и можно работать дальше, а именно проверять результат выполнения операции (в данном случае функции `connect`).

5.5.1. Работа с событиями

Функция `WSACreateEvent` не получает никаких параметров, а только создает новый объект-событие и возвращает его.

Создав пустое событие, мы можем добавлять в него нужные сокеты с помощью функции `WSAEventSelect`. Эта функция выглядит следующим образом:

```
function WSAEventSelect(
 s : TSocket;
 hEventObject : WSAEVENT;
 lNetworkEvents : LongInt
): Integer; stdcall;
```

Тут у нас три параметра:

- сокет, который нужно добавить в набор;
- объект-событие, в который нужно добавить сокет;

- события, на которые необходимо реагировать. События могут быть сочетанием из следующих флагов:
  - `FD_READ` — готов для чтения данных из сети;
  - `FD_WRITE` — готов к передаче данных;
  - `FD_ACCEPT` — запрос на входящее соединение;
  - `FD_CONNECT` — установлено соединение;
  - `FD_CLOSE` — закрыть соединение;
  - `FD_OOB` — есть внеполосные данные (out-of-band).

Мы пишем сканер портов, а значит, нас интересует событие `FD_CONNECT`, но можно добавить еще и `FD_WRITE`. Во время соединения готовность к записи может быть только тогда, когда соединение установлено.

Когда вы вызываете функцию `WSAEventSelect`, сокет автоматически переходит в асинхронный режим.

После завершения работы с событиями необходимо вызвать функцию `WSACloseEvent`. Она в качестве параметра получает объект-событие, который нужно закрыть.

5.5.2. Время и количество

Когда мы запускаем ожидание события, то должны указать максимальное время в секундах. Это необходимо, потому что если сервер не ответит ни на одну попытку соединения, то программа может ждать очень долго. Это особенно важно для сканера портов, ведь излишнее зависание нам ненужно.

Допустим, что программа проверяет соединение с 21-м портом. А если он у сервера закрыт, то она может долго и нудно ждать ответа, которого просто не будет. Раз порт закрыт, значит, ответа не будет. Вот именно поэтому обязательно нужно указывать максимальное время ожидания, после которого считается, что соединение невозможно.

В синхронном режиме за время ожидания отвечает библиотека WinSock, а в асинхронном мы сами регулируем время и можем прервать ожидание в любой момент, а можем ждать и бесконечно. Тут надо отметить, что когда мы посылаем асинхронный запрос на соединение, и долго нет ответа по причине недоступности сервера, то мы уже никогда не получим ответ, даже если сервер уже стал доступен. Это связано с тем, что попытка соединения происходит только один раз при отправке запроса, а не все время ожидания. Задержка на соединение связана только с накладными расходами на сервере, а не с ожиданием освобождения порта или сервера.

При выборе максимальной продолжительности ожидания нужно быть очень аккуратным, потому что если выбрать слишком большое число, то сканер потеряет в скорости. Ну а если указать слишком маленькое, то сервер может не успеть ответить, и вы будете думать, что порт закрыт, а на самом деле все в порядке.

Время ожидания сильно зависит от количества одновременно сканируемых портов, скорости соединения и мощности сервера. Из моей практики могу сказать, что если сканировать через модем при скорости 28,8 Кбит/с по 20—40 портов в пачке, максимальное время нужно ставить в 1—2 секунды. Если вы хотите отсканировать сразу 1024 порта, то лучше поставить 3—4 секунды. Большее значение лучше не ставить, потому что все равно из 1024 портов у интернет-серверов открыто бывает не более 10. За 4 секунды любой сервер сможет ответить на попытку соединения на 10 из его портов. Но я не советую вам сканировать такими большими пачками, лучше ограничиться максимумом в 40 портов.

Для локальных сетей это значение можно уменьшить и сканировать по 50 портов с задержкой в 1 секунду. Даже при средней загрузке сервера это вполне нормально.

При сканировании с задержкой в 1 секунду и пачками по 40 портов мой сканер отсканировал 1024 порта за 16 секунд. Но это, правда, при большом обилии открытых портов на моем локальном сервере. Если будет доступен только один порт, то сканирование будет проходить чуть более 25 секунд. Когда порты открыты, то они отвечают быстро. Если ни одного открытого порта в сканируемой пачке нет, то программа будет ожидать максимально установленное время, и сканирование будет происходить дольше.

5.5.3. Кодинг

Вот теперь перейдем к практической части рассмотрения нашего сканера портов. Для будущего примера я создал форму, как на рис. 5.5.

Здесь присутствуют следующие элементы:

- поле ввода адреса сканируемого сервера — компонент `TEdit` с именем `AddressEdit`;
- текстовое поле для ввода начального порта, с которого нужно начать сканирование — компонент `TEdit` с именем `StartPortEdit`;
- текстовое поле для ввода конечного порта, до которого нужно сканировать — компонент `TEdit` с именем `EndPortEdit`;
- кнопка, после нажатия которой будем запускать сканирование;

- область, в которой будет отображаться результат — компонент `TRichEdit` с именем `DisplayMemo`.

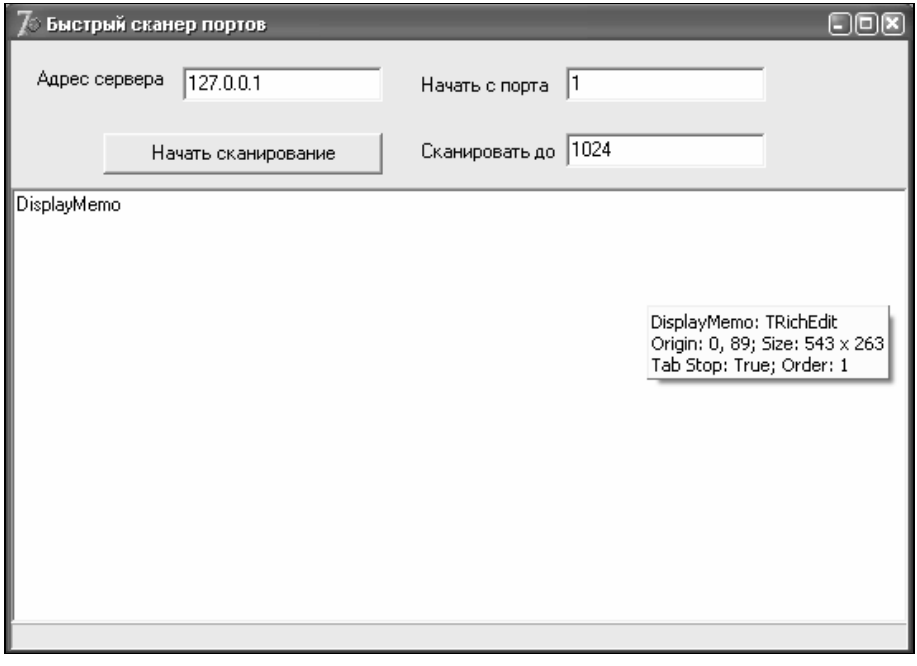


Рис. 5.5. Форма будущего сканера портов

В раздел `uses` исходного кода формы нужно добавить модуль `Winsock2`, чтобы мы могли пользоваться новыми функциями второй версии WinSock.

В обработчике нажатия кнопки **Начать сканирование** нужно написать код из листинга 5.2.

Листинг 5.2. Сканирование портов

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i,j,s, opt, index: Integer;
 FSocket: array [0..41] of TSOCKET; // массив сокетов
 busy   : array [0..41] of boolean; // массив состояний
 port   : array [0..41] of integer; // массив сканируемых портов
 addr   : TSockAddr;
```

```
 hEvent : THandle; // объект для обработки сетевых событий
 fset   : TFDset;
 tv     : TTimeval;
 tec    :PServEnt;
 PName:String;
begin
 // Устанавливаем максимальное и минимальное значения полоски
 // состояния сканирования. Минимум — начальный порт сканирования,
 // максимум — конечный порт.
 ProgressBar1.Min:=StrToInt(StartPortEdit.Text);
 ProgressBar1.Max:=StrToInt(EndPortEdit.Text);

 // Записываем в переменную i значение начального порта
 i:=StrToInt(StartPortEdit.Text);

 // Заполняем основные поля структуры addr, которая будет использоваться
 // при вызове функции connect
 addr.sin_family := AF_INET;
 addr.sin_addr.s_addr := INADDR_ANY;

 // Выводим сообщение о том, что начат поиск введенного хоста
 DisplayMemo.SelAttributes.Color:=clTeal;
 DisplayMemo.SelAttributes.Style:=
        DisplayMemo.SelAttributes.Style+[fsBold];
 DisplayMemo.Lines.Add('Поиск хоста');

 // Ищем хост и адрес сохраняем в структуре addr
 addr.sin_addr := LookupName;

 // Выводим сообщение о том, что начато сканирование
 DisplayMemo.SelAttributes.Color:=clTeal;
 DisplayMemo.SelAttributes.Style:=
       DisplayMemo.SelAttributes.Style+[fsBold];
 DisplayMemo.Lines.Add('Сканирование...');
```

```
// В index находится количество сокетов, проверяемых за один раз
index:=40;

// Создаем объект для обработки сетевых событий
hEvent := WSACreateEvent();

// цикл сканирования
while i<StrToInt(EndPortEdit.Text) do
 begin
  // Всем элементам массива busy присваиваем значение false
  for j:=0 to index do
   busy[j]:=false;

  // В этом цикле будут асинхронно посылаться запросы на соединение.
  // Переменная j будет изменяться от 0 до максимального количества
  // элементов в массиве.
  for j:=0 to index do
   begin
    // Если j-й порт превысил значение указанного максимального
    // порта, то прервать цикл
    if i>StrToInt(EndPortEdit.Text) then
     begin
      index:=j-1;
      break;
     end;

    // Инициализируем очередной j-й сокет из массива FSocket
    FSocket[j] := socket(AF_INET, SOCK_STREAM, IPPROTO_IP);

    // Добавляем j-й сокет к объекту событий с помощью WSAEventSelect
    WSAEventSelect(FSocket[j], hEvent, FD_WRITE + FD_CONNECT);

    // Указываем порт, на который надо произвести попытку соединения
    addr.sin_port := htons(i);
```

```
    // Попытка соединения на очередной порт
    connect(FSocket[j], @addr, sizeof(addr));

    // Даем ОС поработать и обработать накопившиеся события.
    // Если этого не делать, то во время сканирования будет
    // возникать эффект зависания
    Application.ProcessMessages;

    // Проверяем, были ли ошибки
    if WSAGetLastError()=WSAEINPROGRESS then
     begin
      // Если ошибка произошла, то закрываем этот порт
      closesocket (FSocket[j]);

      // Устанавливаем соответствующий элемент в массиве busy в true,
      // чтобы потом не проверять этот порт, потому что он все равно
      // уже закрыт
      busy[j]:=true;
     end;

    // Указываем в массиве port, на какой именно порт мы
    // сейчас послали запрос
    port[j]:=i;
    // Увеличиваем счетчик i, в котором отслеживаем, какой
    // порт сейчас сканируется, чтобы на следующем
    // этапе цикла for запустить сканирование следующего порта
    i:=i+1;
   end;
  // Обнуляем переменную fset
  FD_Zero(fset);

  // Заполняем сканируемый массив сокетов в переменную fset
  for j := 0 to index do
   begin
    if busy[j] <> true then
```

```
    FD_SET (FSocket[j], fset);
  end;

// Даем ОС поработать и обработать накопившиеся события
Application.ProcessMessages;

// Заполняем структуру, в которой указано время ожидания
// события от сокета
tv.tv_sec := 1; // Мы будем ждать 1 секунду
tv.tv_usec := 0;

// Ожидаем, пока произойдет хотя бы одно событие от любого из сокетов
s:=select (1, nil, @fset, nil, @tv);

// Даем ОС поработать и обработать накопившиеся события
Application.ProcessMessages;

// Запускаем массив, в котором будет проверяться, какие из сокетов в
// массиве FSocket прошли соединение успешно, а какие нет
for j := 0 to index do
 begin
  // Проверяем, был ли закрыт соответствующий порт из-за ошибки.
  // Если да, то нет смысла его проверять
  if busy[j] then continue;

  if FD_ISSET (FSocket[j], fset) then
   begin
    // В переменную s записывается размер переменной Opt
    s:=Sizeof(Opt);
    opt:=1;

    // Получаем состояние текущего j-го сокета,
    // результат состояния будет в переменной opt
    getsockopt(FSocket[j], SOL_SOCKET, SO_ERROR, @opt, s);
```

```
      // Если opt равно 0, то порт открыт, и к нему можно подключиться
      if opt=0 then
        begin
         // Пытаемся узнать символьное имя порта
         tec := getservbyport(htons(Port[j]),'TCP');
         if tec=nil then
          PName:='Unknown'
         else
          begin
           PName:=tec.s_name;
          end;
         // Выводим сообщение об открытом порте
         DisplayMemo.Lines.Add('Хост:'+AddressEdit.Text+': порт :'
             +IntToStr(Port[j])+' ('+Pname+') '+' открыт ');
        end;
      end;

     // Закрыть j-й сокет, потому что он больше уже не нужен
     closesocket(FSocket[j]);
    end;

   // Увеличиваем позицию в ProgressBar1
   ProgressBar1.Position:=i;
  end;
 // Закрываем объект событий
 WSACloseEvent(hEvent);

 // Выводим сообщение о конце сканирования
 DisplayMemo.SelAttributes.Color:=clTeal;
 DisplayMemo.SelAttributes.Style:=
      DisplayMemo.SelAttributes.Style+[fsBold];
 DisplayMemo.Lines.Add('Сканирование закончено...');
end;
```

На первый взгляд код достаточно большой, но реально тут больше места занимают комментарии. По ним вы сможете без проблем разобраться со

всем происходящим, а я не буду вас сильно путать. Код и комментарии лучше всего расскажут логику работы сканера. Я хочу добавить к этим комментариям только пару замечаний и описание используемых сетевых функций, которые мы еще не рассматривали.

Запустите пример и можете уже насладиться его результатом (рис. 5.6), а потом переходите к рассмотрению новых функций.

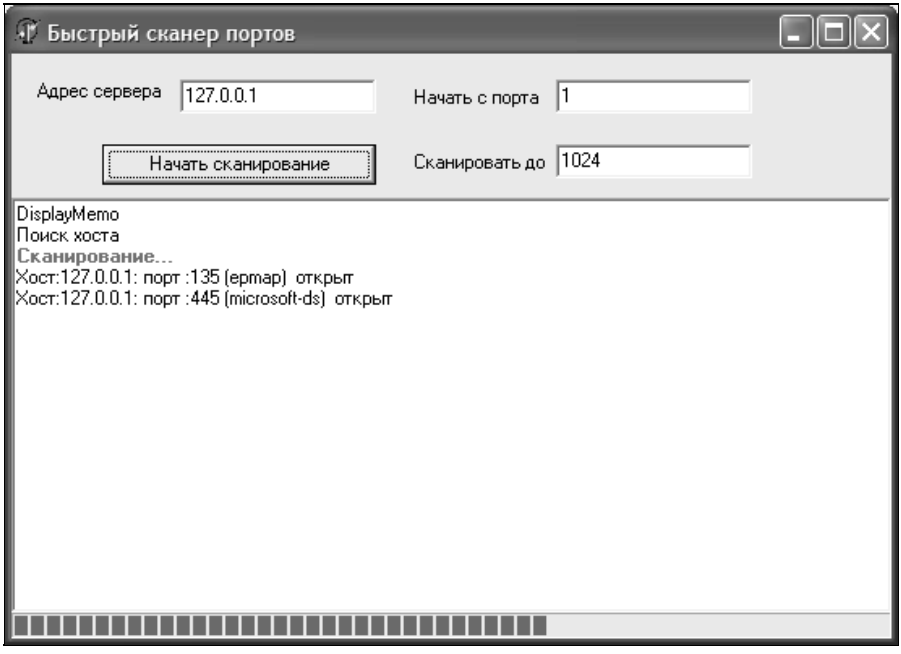


Рис. 5.6. Сканер в работе

Примечание

На компакт-диске в каталоге \Примеры\Глава 5\FastScanPort\ вы можете найти пример быстрого сканера.

5.5.4. Структура типа *TFDSet*

Отдельного разговора в данном примере заслуживает переменная `fset`, которая имеет тип `TFDSet`.

Этот тип данных вплотную связан с сетевыми событиями и на самом деле является структурой следующего вида:

```
PFDSet = ^TFDSet;
TFDSet = packed record
 fd_count: u_int;
 fd_array: array[0..FD_SETSIZE-1] of TSocket;
end;
```

Первое поле — это количество элементов в массиве, а второй параметр — это сам массив из сокетов, за которыми необходимо наблюдать. Тут нужно обратить внимание на второй параметр, ведь здесь у нас далеко не динамический массив, а вполне конечный. Максимальная длина равна `FD_SETSIZE`, а эта константа, в свою очередь, равна 64. Постарайтесь не превысить это значение, если захотите увеличить количество одновременно сканируемых сокетов.

Итак, структура `TFDSet` — это набор сокетов, от которых нужно ожидать разрешения определенной операции. Например, если вам нужно дождаться прихода данных на один из двух сокетов, то вы можете сделать следующее: добавить в набор `TFDSet` два уже созданных сокета, запустить функцию `select` и в качестве второго параметра указать набор с сокетами.

Кстати, что такое функция `select`?

Одной из первых функций проверки готовности сокета была функция `select`:

```
function select(
 nfds: Integer;
 readfds: PFDSet;
 writefds: PFDSet;
 exceptfds: PFDSet;
 timeout: PTimeVal
): Longint; stdcall;
```

Функция возвращает количество готовых к использованию дескрипторов сокета. Теперь рассмотрим параметры этой функции:

- `nfds` — игнорируется и служит только для совместимости с моделью сокетов BSD, используемой в Linux;
- `readfds` — проверить возможность чтения (структура типа `TFDSet`);
- `writefds` — проверить возможность записи (структура типа `TFDSet`);

- `exceptfds` — проверка внеполосных данных (структура типа `TFDSet`);
- `timeout` — максимальное время ожидания или `nil` для блокирования дальнейшей работы (ожидать бесконечно).

Функция `select` будет ожидать данные указанное время, после чего вы можете прочитать данные из сокета. Но данные могут прийти только на один из двух сокетов или вообще не прийти, потому что просто вышло время ожидания. Как узнать, на какой именно? Для начала нужно обязательно проверить, входит ли сокет в набор, с помощью функции `FD_ISSET`.

При работе со структурой типа `TFDSet` вам понадобятся следующие функции:

- `FD_ZERO` — прежде чем добавлять в набор новые сокеты, необходимо его очистить, чтобы проинициализировать набор. Очистка должна происходить именно этой функцией, а не удалением `FD_CLR`, потому что вторая функция не инициализирует набор, а просто удаляет его элементы. У этой функции только один параметр — указатель на переменную типа `TFDSet`;
- `FD_SET` — добавляет сокет в набор. У функции два параметра — сокет, который нужно добавить, и переменная типа `TFDSet`, в набор которой нужно добавить сокет;
- `FD_CLR` — удалить сокет из набора. У этой функции два параметра — сокет, который надо удалить, и набор, из которого будет происходить удаление;
- `FD_ISSET` — проверяет, входит ли сокет, указанный в качестве первого параметра, в набор типа `TFDSet`, указанный в качестве второго параметра.

Вот теперь мы готовы пробежаться по логике работы с сокетами в нашем примере сканера портов в асинхронном режиме:

1. Создаем пустой объект событий с помощью `WSACreateEvent`.
2. Создаем сокеты и добавляем их в объект события `WSAEventSelect`. При этом сокет переходит в неблокирующий режим. Есть еще другие способы перевода сокета в неблокирующий режим, но о них вы узнаете позже в этой книге.
3. Очищаем массив сокетов из структуры `TFDSet` с помощью `FD_Zero`.
4. Заполняем массив сокетов `TFDSet` с помощью `FD_SET`.
5. Вызываем функцию `select`, чтобы дождаться, когда наступит одно из указанных нами событий.
6. С помощью `FD_ISSET` проверяем, для какого сокета из массива `TFDSet` сокетов наступило событие.

Опции сокета

Когда мы узнали, для какого из сокетов произошло событие, то необходимо проверить, а вдруг просто произошла ошибка соединения. Для этого вызываем функцию `getsockopt`. В общем виде она выглядит так:

```
function getsockopt(
 s: TSocket;
 level,
 optname: Integer;
 optval: PChar;
 var optlen: Integer
): Integer; stdcall;
```

Здесь у нас пять параметров:

- `s` — сокет, параметры которого надо определить;
- `level` — уровень, который определяет параметр. Здесь можно указывать одну из двух констант: `SOL_SOCKET` или `IPPROTO_TCP`;
- `optname` — имя параметра, значение которого надо прочитать;
- `optval` — новое значение параметра, которое надо установить;
- `optlen` — размер данных в параметре `optval`.

Кстати, существует еще функция `setsockopt`, которая может изменять параметры сокета. Эта функция идентична и имеет те же параметры, и параметры выполняют те же функции, поэтому все, что мы будем рассматривать для `getsockopt`, далее также относится и к функции изменения параметров.

Если в качестве второго параметра `level` задана константа `SOL_SOCKET`, то в третьем параметре можно указать одно из следующих значений (в скобках приведен тип данных):

- `SO_ACCEPTCONN` (`Boolean`) — этот параметр можно только читать. Если он равен `true`, то сокет находится в режиме прослушивания порта;
- `SO_ERROR` (`Boolean`) — ошибка сокета;
- `SO_BROADCAST` (`Boolean`) — если указать `true`, то сокет сконфигурирован для отправки или приема широковещательных пакетов. Этот параметр допустим для протоколов без установки соединения (нельзя использовать с `SOCK_STREAM`);
- `SO_CONNECT_TIME` (`Integer`) — позволяет определить продолжительность времени соединения. Этот параметр только для чтения, потому что время соединения изменить вручную нельзя;

- SO_DONTLINGER (Boolean) — если установить в false, то при закрытии сокета все данные, находящиеся в очереди на отправку или прием, будут корректно отработаны. Если этот параметр установить в true, то через определенное время сокет закроется без обработки данных, находящихся в очереди;
- SO_DONTROUTE (Boolean) — если равен true, то при отправке пакетов не должна использоваться маршрутизация;
- SO_EXCLUSIVEADDRUSE (Boolean) — если равен true (значение по умолчанию), то никакое другое приложение не может использовать выбранный вами порт на локальной машине. Если параметр изменить на false, то порт не блокируется и может быть открыт другим процессом. При этом системе сложно будет определить, какое из двух приложений должно получить данные, пришедшие на порт, поэтому без особой надобности не стоит изменять этот параметр;
- SO_LINGER (структура типа LInger) — определяет время ожидания обработки очереди сообщений после закрытия сокета. Если в течение этого времени не все данные будут отработаны, сокет все равно закроется;
- SO_MAX_MSG_SIZE (Integer) — параметр только для чтения и возвращает максимальный размер сообщения;
- SO_TYPE (Integer) — возвращает тип сокета, например SOCK_STREAM;
- SO_RECVBUF (Integer) — с помощью этого параметра можно определить или задать размер буфер приема;
- SO_SNDBUF (Integer) — с помощью этого параметра можно узнать или изменить размер буфера отправки;
- SO_REUSEADDR (Boolean) — если установлен в true, то можно создать сокет только с таким адресом, который уже используется другим сокетом. Как мы уже говорили, не желательно использовать сокеты, которые уже заняты другим процессом;
- SO_SNDTIMEO (Integer) — время ожидания для отправки в блокирующем режиме;
- SO_RCVTIMEO (Integer) — время ожидания для получения данных в блокирующем режиме.

В TCP/IP нас будет интересовать еще один уровень — IPPROTO_TCP, в котором можно установить параметр TCP_NODELAY. Заголовок TCP-пакета составляет 20 байтов. Если отправлять по сети пакеты по 1 байту и при этом каждому из них будет прикрепляться заголовок в 20 байтов, то это будет слишком расточительным. По умолчанию сокет накапливает данные и старается отправлять пакеты более заполненными, для оптимизации трафика. Это хорошо, но когда нужно отправлять маленькие пакеты, возникнут не-

оправданные задержки, и в этом случае неплохо было бы установить параметр `TCP_NODELAY` в `true`.

В нашем примере быстрого сканера портов мы проверяем, не равно ли состояние порта ошибке `SO_ERROR`. Если ошибки нет, то порт точно открыт.

Имя сервиса

Что-то наш пример сканера портов потребовал намного больше функций, чем я предполагал. Ну, ничего, зато узнали достаточно много нового и интересного. Последняя функция, которую нам следует рассмотреть, — `getservbyport`. Она возвращает текстовое название порта. Этот текст мы выводим на экран, чтобы результат был более наглядным.

Функция выглядит следующим образом:

```
function getservbyport(
 port: Integer;
 proto: PChar
): PServEnt; stdcall;
```

В качестве первого параметра указывается номер порта, а во втором параметре — текстовая строка с именем протокола. Помните, что для протоколов TCP и UDP порт с одним и тем же номером на самом деле не является идентичным? Вот поэтому и нужно указывать протокол.

Несмотря на то, что функция хорошо определяет основные порты, результату доверять нельзя. Дело в том, что имена сервисов функция банально берет из файла C:\Windows\System32\drivers\etc\services. Попробуйте сейчас найти этот файл у себя на диске, откройте его в любом текстовом редакторе и найдите строку:

```
epmap    135/tcp    loc-srv     #DCE endpoint resolution
```

Эту строку можно разбить на несколько колонок: имя службы, номер порта/протокол, псевдонимы, комментарий (идет после символа `#`). В данном случае перед нами строка для службы `epmap`, которая работает на 135-м порту TCP-протокола. Попробуйте заменить имя `epmap` на `ерунда` и запустить сканирование портов своего компьютера. Если у вас порт открыт (а он, вроде бы, должен быть открыт на Windows-компьютерах), то наш сканер вернет для него имя, которое вы написали в файле.

Получается, что функция `getservbyport` возвращает имя службы, которая должна работать, но не факт, что именно она находится перед нами. Администратор может запустить Web-сервер не на 80-м порту, а на 21-м, и тогда наш сканер будет показывать некорректное имя. Но это уже из области очень редкого случая, и на него не будем обращать внимания.

5.6. Продолжаем знакомиться с WinSock

Когда в *разд. 5.1* я описывал функции WinSock, мною были рассмотрены только основные функции: инициализация и соединение. С обеими мы познакомились на практике и создали быстрый сканер портов. Теперь нам предстоит расширить свои знания о WinSock и узнать, какие функции используются для получения и отправки данных. Хотя в своей книге я буду их часто использовать, знание "внутренностей" сетевой библиотеки Windows никогда не помешает.

Первая функция, которую мы рассмотрим, — `bind`. Она используется при написании серверов и предназначена для связывания локального адреса с сокетом. Зачем это нужно? Дело в том, что на одном компьютере может быть установлено несколько сетевых карт, и у каждой из них может быть свой IP-адрес. Когда запускается сервер, то с какой сетевой карты ему необходимо ожидать соединений? С помощью `bind` вы связываете сокет с сетевым адресом той сетевой карты, с которой нужно ожидать запросы пользователей. В общем виде `bind` выглядит следующим образом:

```
function bind(
 const s: TSocket;
 const addr: PSockAddr;
 const namelen: Integer
): Integer; stdcall;
```

Функция получает три параметра:

- `s` — дескриптор, определяющий несвязанный сокет;
- `addr` — структура типа `SockAddr`, в которой указан IP-адрес сетевой карты, с которой нужно работать. Если вместо реального IP-адреса указать константу `INADDR_ANY`, то сервер будет ожидать запросов от клиентов с любой сетевой карты на данном компьютере;

> ***Примечание***
>
> Не забываем, что IP-адрес — это 32-битное число, где байты идут в отличной от привычной Intel-процессорам последовательности, поэтому константу `INADDR_ANY` нужно приводить в нужный формат с помощью функции `htonl`.

- `namelen` — длина структуры, указанной во втором параметре.

Если функция отработала успешно, то результатом будет 0, иначе результатом будет константа `SOCKET_ERROR`.

Следующая функция, необходимая при программировании серверов сетевых программ, — `listen`. Когда серверная программа открыла порт и ожидает

соединения со стороны клиента, то она должна вызвать эту функцию. Функция служит для начала прослушивания порта на случай подключения к нему со стороны клиента. Вот так выглядит эта функция в WinSock2:

```
function listen(
  s: TSocket;
  backlog: Integer
  ): Integer; stdcall;
```

У нее два параметра:

- `s` — дескриптор гнезда или сокет;
- `backlog` — максимально допустимое число запросов, ожидающих обработки. Если этот параметр равен `SOMAXCONN`, то ядро само установит максимальное возможное для него значение.

В большинстве случаев, параметр `blocklog` зависит от установленного в системе параметра "максимальное количество подключений". Если вы используете Windows 95/98, то этот параметр регулируется в настройках сети.

Следующая функция называется `accept`. Она служит для подтверждения соединения сервером. Эта функция принимает запросы на подключение, поступающие на вход процесса-сервера:

```
function accept(
  const s: TSocket;
   var addr: TSockAddr;
   var addrlen: Integer
  ): TSocket; stdcall;
```

У функции три параметра:

- `s` — это все тот же дескриптор гнезда/сокета;
- `addr` — указатель на структуру, через которую нам возвращают адрес подключаемого клиента;
- `addrlen` — размер адреса.

После завершения выполнения функции ядро записывает в переменную `addrlen` длину параметра `addr`. Функция возвращает новый дескриптор гнезда, отличный от дескриптора `s`. Процесс-сервер может продолжать слежение за состоянием объявленного гнезда, поддерживая связь с клиентом по отдельному каналу.

Вот мы и закончили рассматривать функции, необходимые вам для соединения клиента и сервера. Теперь мы начнем знакомиться с передачей данных. И первой на очереди стоит функция отправки пакетов, потому что для

того, чтобы что-то принять, необходимо сначала отправить. И поможет нам в отправке пакетов функция send.

```
function send(
  s: TSocket;
  var Buf;
  len,
  flags: Integer
  ): Integer; stdcall;
```

Рассмотрим каждый параметр в отдельности:

- s — как всегда, это дескриптор сокета;
- buf — указатель на посылаемые данные;
- len — размер данных;
- flags — флаги, установки.

Функция возвращает количество фактически переданных байтов.

Параметр flags может содержать значения: MSG_DONTROUTE определяет, что данные не должны быть подчинены маршрутизации, MSG_OOB — послать данные out-of-band ("через таможню"), если посылаемые данные не учитываются в общем информационном обмене между взаимодействующими процессами.

Длина сообщения не должна превышать значения в SO_MAX_MSG_SIZE.

Прием данных осуществляется функцией recv:

```
function recv(
  s: TSocket;
  var buf; len, flags: Integer
  ): Integer; stdcall;
```

Параметры практически те же:

- s — дескриптор сокета;
- buf — массив для приема данных;
- len — ожидаемый объем данных;
- flags — флаги, могут быть установлены таким образом, что поступившее сообщение после чтения и анализа его содержимого не будет удалено из очереди, или настроены на получение данных out-of-band:
  - MSG_PEEK — данные будут скопированы в буфер, но не удалены из входной очереди;
  - MSG_OOB — то же, что и в функции send.

Функция `recv` возвращает количество байтов, фактически переданных пользовательской программе.

Для дейтаграммных версий используются функции `sendto` и `recvfrom`. Обе функции работают так же, как и в `send` и `recv`, только в качестве дополнительных параметров указываются адреса.

Теперь мы научились получать соединения, посылать данные, осталось только научиться закрывать соединения. Функция `shutdown` закрывает соединение.

5.7. Определение локального/удаленного IP-адреса

Я уже рассказал, как определить локальный IP-адрес *(см. разд. 5.4)*, но этот способ очень сложный, хотя и очень хороший, и позволяет получить множество дополнительной информации. Чаще всего подобные действия избыточны и сложны, а нужно определить только IP и ничего больше.

В этом разделе я приведу небольшой пример определения IP-адреса, заодно мы познакомимся с еще одной API-функцией библиотеки WinSock. Возможно, она вам пригодится в будущем.

Сейчас мы напишем программу, в которой программа возвращает все установленные IP-адреса для сетевых интерфейсов. Помимо этого, вы узнаете, как обращаться к компонентам на форме не по имени, а по "индексу", это очень удобная и нужная возможность.

Для примера бросьте на форму одну кнопку и несколько компонентов `TEdit`. Для кнопки создайте обработчик события `OnClick` и напишите в нем код, показанный в листинге 5.3.

Листинг 5.3. Определение адреса

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TaPInAddr = Array[0..10] of PInAddr;
  PaPInAddr = ^TaPInAddr;
var
  phe: PHostEnt;
  pptr: PaPInAddr;
  Buffer: Array[0..63] of Char;
```

```
  I: Integer;
  GInitData: TWSAData;
begin
 // Инициализация сокетов
 WSAStartup($101, GInitData);
 // Получаем имя локального компьютера (хоста)
 GetHostName(Buffer, SizeOf(Buffer));
 // Получаем указатель на хост
 phe := GetHostByName(buffer);
 if phe = nil then Exit;
 // Получаем указатель на массив адресов
 pPtr := PaPInAddr(phe^.h_addr_list);
 I := 0;
 // Перечисляем все адреса
 while pPtr^[I] <> nil do
  begin
   // Вывести адрес
   TEdit(FindComponent('Edit'+IntToStr(i+1))).Text:=inet_ntoa(pptr^[I]^);
   Inc(I);
  end;
 // Закрываем сокет
 WSACleanup;
end;
```

Теперь рассмотрим новые функции, которые мы использовали в этом примере. Первая — это `GetHostName`, она определяет имя локальной машины. У функции есть два параметра:

- буфер, в который будет занесено имя машины;
- размер выделенной под буфер памяти.

Следующая функция — `GetHostByName`, она определяет IP-адрес компьютера по его имени. В качестве единственного параметра нужно передать имя компьютера, адрес которого нужно узнать. Мы передаем буфер, в котором находится имя локальной машины, которое мы предварительно выяснили с помощью функции `GetHostName`. Результатом выполнения функции будет массив из структур типа `PInAddr`.

Почему при определении IP-адреса мы получаем массив из структур? Это потому, что на компьютере может быть установлено несколько сетевых

карт, например, у меня на работе на одном из серверов было установлено две сетевые карты на 10 и 100 Мбит/с. Я использовал их для разделения двух сетей, с разной скоростью и с разной топологией. А если бы у меня было еще и подключение к Интернету через модем, то появился бы еще один адрес.

Вот теперь самое интересное — вывод результата:

```
TEdit(FindComponent('Edit'+IntToStr(i+1))).Text:=inet_ntoa(pptr^[I]^)
```

Здесь использована функция `FindComponent`, которая ищет компонент на форме по имени. В качестве параметра нужно передать имя компонента, например, `Edit1`. Но можно поступить хитрее и передать имя `Edit` плюс индекс, приведенный к строковому типу `IntToStr(i+1)`. В итоге на первом этапе мы будем искать `FindComponent('Edit1')`, на втором этапе `FindComponent('Edit2')` и т. д.

Найденный с помощью функции `FindComponent` компонент нужно привести к типу `TEdit` с помощью `TEdit(FindComponent('Edit'+IntToStr(i+1)))`. А далее используем всю эту конструкцию, как простой `TEdit`-компонент:

```
TEdit(FindComponent('Edit'+IntToStr(i+1))).Параметр:=Значение;
```

Рассмотрим еще пример. Допустим, у вас есть пять компонентов `CheckBox` с именами `CheckBox1`, `CheckBox2`, `CheckBox3` и т. д. Чтобы перебрать все эти компоненты и узнать, какой из них выделен, нужно сделать так:

```
for i:=1 to 5 do
 if TCheckBox(FindComponent('CheckBox'+IntToStr(i+1))).Checked then
  begin
   // i-й компонент CheckBox выделен
  end;
```

Ну и напоследок идет функция `inet_ntoa`. Она превращает переданный ей IP-адрес в строку. В качестве единственного параметра передаем адрес, а на выходе получаем его строковое представление.

> ***ПРИМЕЧАНИЕ***
>
> На компакт-диске в каталоге \Примеры\Глава 5\GetIP\ вы можете найти пример этой программы.

5.8. Пишем TCP/IP-сервер и клиента

Теперь у нас достаточно знаний для того, чтобы начать писать более интересные примеры, работающие с Windows Sockets. Сетевое общение по про-

токолу TCP/IP основано на технологии "клиент-сервер". Один компьютер является сервером, а другой клиентом. Чем это грозит нам? А тем, что программируются эти части по-разному.

5.8.1. TCP-сервер

Для создания сервера необходимо:

1. Создать сокет с помощью функции `socket`.
2. Заполнить структуру `sockaddr_in`, указав в ней:
   - в поле `sin_addr.s_addr` — IP-адрес сетевой карты, с которой будет связан сокет.
   - в поле `sin_family` — семейство адресации;
   - в `sin_port` — порт, на котором нужно ожидать соединений.
3. Запустить прослушивание порта с помощью функции `listen`.
4. Вызвать функцию `accept`, которая ожидает соединение от клиента, и когда придет запрос на подключение, функция вернет управление и передаст нам сокет `TSocket`, через который можно будет отправлять клиенту сообщения.

Не забываем, что в обоих случаях необходимо еще при старте программы запустить сетевую библиотеку с помощью `WSAStartup`.

Давайте теперь на примере посмотрим, как будет выглядеть создание TCP/IP-сервера в виде кода. Для этого создадим новый проект, на форму бросим одну кнопку и компонент `Memo`. По нажатию кнопки будем запускать сервер, а в `Memo`-компоненте будет отображаться ход работы программы. Код запуска сервера можно увидеть в листинге 5.4. Напишите его в обработчике события `OnClick` для кнопки.

Листинг 5.4. Запуск сервера

```
procedure TForm1.Button1Click(Sender: TObject);
var
 sServer, sClient : TSOCKET;
 serverAddr, clientAddr : sockaddr_in;
 iSize : Integer;
begin
 // Создаем сокет
 sServer := socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
```

```
if sServer = INVALID_SOCKET then
 begin
  MessageBox(0, 'Ошибка создания сокета', 'Ошибка', 0);
  exit;
 end;

// Заполняем структуру с адресом
serverAddr.sin_addr.s_addr := htonl(INADDR_ANY);
serverAddr.sin_family := AF_INET;
serverAddr.sin_port := htons(7777);

// Связываем сокет с адресом
if bind(sServer, @serverAddr, sizeof(serverAddr)) = SOCKET_ERROR then
 begin
  ShowMessage('Ошибка bind');
  exit;
 end;

// Запускаем прослушивание
if listen(sServer, 4)=SOCKET_ERROR then
 begin
  ShowMessage('Ошибка listen');
  exit;
 end;

memoInfo.Lines.Add('Сервер запущен');

// Ожидаем соединения
iSize := sizeof(clientaddr);
sClient := accept(sServer, clientaddr, iSize);
if sClient = INVALID_SOCKET then
 begin
  ShowMessage('Ошибка accept');
  exit;
 end;
```

```
 // Сообщение об удачном соединении
 memoInfo.Lines.Add('Подключение от: '+inet_ntoa(clientaddr.sin_addr));

 // закрываем сокет
 closesocket(sServer);
end;
```

В самом начале создаем сокет, указывая три параметра:

- `AF_INET` — будет использоваться интернет-адресация, т. е. протокола IP;
- `SOCK_STREAM` — протокол с установкой соединения, т. е. TCP;
- `IPPROTO_IP` — уточняем протокол.

После этого заполняем структуру `sockaddr_in`, в которой нужно указать адрес используемого интерфейса, семейство протокола и порт, на котором сервер будет ожидать подключения. В качестве адреса для примера я указал `INADDR_ANY`, т. е. соединение может быть доступно с любого сетевого интерфейса, конечно же, если там порт не запрещен сетевым экраном.

Заполнив эти данные, вызываем функцию `bind`, чтобы связать сокет с заполненной нами структурой.

На этом этапе сервер готов к работе, но он еще не открыл порт и не начал прослушивание. Чтобы сделать это, вызываем функцию `listen`. Если все прошло удачно, то порт свободен и может использоваться. Если не удалось выполнить эту функцию, то порт занят в блокирующем режиме другой программой, и нужно прерывать работу. Если же все прошло удачно, то выводим соответствующее сообщение в `Memo`-компонент.

Прослушивание началось, и чтобы принять соединение от клиента, нужно вызвать функцию `accept`. Если вы работаете в блокирующем режиме, а пока мы работаем только так, то функция блокирует работу программы в ожидании соединения. Когда клиент наконец-то подключится к программе, функция вернет сокет, через который мы сможем общаться с клиентом.

Чтобы пример был более интересным, когда клиент подключится, мы увидим в `Memo`-компоненте IP-адрес клиента. Чтобы получить этот адрес, нам нужно обратиться к полю `clientaddr` структуры `sin_addr`. Эту структуру мы передаем в качестве второго параметра функции `accept` и получаем в ней данные об адресе клиента.

Теперь закрываем сокет с помощью `closesocket`, потому что пока мы не будем передавать данные между клиентом и сервером.

Не спешите запускать пример, потому что он зависнет в ожидании подключения. А так как у нас пока нет клиента, то выйти из программы можно будет только с помощью прерывания процесса.

Здесь необходимо сделать небольшое отступление. Какой порт выбирать для работы сервера? Все зависит от того, какую программу вы пишите. Для общепринятых сервисов (FTP, HTTP, POP3, SMTP и т. д.) есть общепринятые номера портов, поэтому для них лучше выбирать значения, которые рекомендуются в RFC (Requests For Comments).

Если вы разрабатываете какую-то уникальную программу, которую до вас еще никто не писал, то придется выбирать порт из тех, которые не используются. Тут рекомендуется брать число, большее 1024. Меньшие значения зарезервированы для распространенных сервисов, а значения больше 1024 можно использовать без зазрения совести.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 5\TCPServer\ вы можете найти пример этой программы.

5.8.2. TCP-клиент

Теперь напишем программу-клиент, которая будет подключаться к серверу. Писать клиенты немного проще, хотя нет, намного проще, потому что нужно выполнить намного меньше действий, и это вы увидите уже на том, что код будет существенно короче.

Итак, чтобы написать программу, соединяющуюся с сервером по протоколу TCP/IP, нужно выполнить следующие действия:

1. Создать сокет, без которого в нашем деле никуда.
2. Заполнить структуру `sockaddr_in`, где нужно указать параметры сервера, к которому будет происходить подключение.
3. Вызвать функцию `connect`.

Теперь посмотрим, как это будет выглядеть в виде кода. В листинге 5.5 можно увидеть пример, который будет соединяться с сервером, написанным в *разд. 5.8.1*.

Листинг 5.5. Код клиента, подключающегося к серверу

```
procedure TForm1.Button1Click(Sender: TObject);
var
 addr : sockaddr_in;
```

```
begin
 sClient := socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
 if sClient = INVALID_SOCKET then
  begin
   MessageBox(0, 'Ошибка создания сокета', 'Ошибка', 0);
   exit;
  end;

 // Заполнение структуры адреса
 addr.sin_addr := LookupName('127.0.0.1');
 addr.sin_family := AF_INET;
 addr.sin_port := htons(7777);

 if (connect(sClient, @addr, sizeof(addr)) = SOCKET_ERROR) then
  begin
   ShowMessage('Ошибка send');
   exit;
  end;
end;
```

Посмотрим, что здесь происходит. Вызов функции `socket` ничем не отличается от того, который мы использовали в *разд. 5.8.1* при написании сервера. А вызов и не должен отличаться, мы будем использовать тот же протокол, а значит, должен быть создан точно такой же сокет.

Теперь заполняем структуру. В поле `sin_addr` необходимо указать адрес компьютера, к которому следует подключиться. У меня нет сейчас сети, где можно было бы разнести клиент и сервер по разным физическим компьютерам, поэтому тестировать приходится на одной машине. А раз сервер будет запущен локально, то для подключения в качестве адреса я указываю свой компьютер (адрес 127.0.0.1).

Порт для подключения мы должны выбрать такой же, как и у сервера. Сервер запустил прослушивание на 7777-м порту, значит, и клиент должен указать в структуре то же самое значение. Если вы захотите подключиться к FTP-серверу (большинство из них работает на 21-м порту), то в поле `sin_port` нужно будет указать число 21.

Кстати, переменная `sClient`, которую мы используем в примере, должна быть объявлена в разделе `private` нашей формы как `TSocket`. Таким образом, мы храним сокет в качестве члена класса и можем потом работать с

ним. Поместите на форму еще одну кнопку, по нажатию которой будем закрывать сокет, и напишите там следующую строку:

```
CloseSocket(sClient);
```

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 5\TCPClient\ вы можете найти пример этой программы.

5.9. Передача данных

В *разд. 5.8* мы научились создавать сервер и клиента и соединяться, но это только цветочки. Самое интересное и самое сложное — это передача данных. В этом разделе нам предстоит познакомиться с функциями и методами передачи данных.

5.9.1. Блокирующий режим

Для написания примеров из этой главы мы будем использовать базу, уже написанную нами в *разд. 5.8*. Мы наделим уже написанные примеры возможностью передачи и приема данных.

Самый ужасный метод для передачи данных — блокирующий. В этом режиме работа программы блокируется на выполнении сетевых функций в ожидании результата. Но это самый простой метод, поэтому мы начнем с него. Открываем клиент, написанный в *разд. 5.8.2*, и после соединения с сервером добавляем следующий код:

```
SendString(sClient, 'Запрос от клиента');
iSize := recv(sClient, sBuff, 1024, 0);
if (iSize > 0) then
 begin
  recievedString:=sBuff;
  ShowMessage(recievedString);
 end;
```

В первой строке вызывается функция `SendString`, которой передаются сокет и строка, которую нужно передать в сеть. Такой функции не существует, но мы ее напишем чуть позже и будем использовать во всех будущих программах.

После этого вызываем функцию `recv`, чтобы получить ответ от сервера. Эта функция уже из состава WinSock, и мы ее рассматривали в *разд. 5.6*.

Если какие-то данные получены, то через второй параметр нам вернут буфер с данными. Полученные данные отображаем на экране с помощью `ShowMessage`.

Теперь посмотрим, как должна выглядеть функция `SendString`:

```
procedure SendString(s:TSocket; str:String);
var
 sRecvBuff : array [0..255] of char;
 TempStr : AnsiString;
begin
 TempStr:=str+#0;
 CopyMemory(@sRecvBuff, PChar(TempStr), Length(TempStr));
 send(s, sRecvBuff, Length(TempStr), 0);
end;
```

Сначала добавляем к отправляемой строке нулевой байт, по которому на стороне сервера мы сможем потом определить конец данных. После этого копируем в буфер `sRecvBuff` строку, которую нужно отправить, и пересылаем в сеть данные с помощью `send`.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 5\TCPClient1\ вы можете найти пример этой программы.

5.9.2. Блокирующий TCP-сервер

Клиент готов, и теперь нам необходимо улучшить сервер так, чтобы он мог прочитать полученные данные и ответить клиенту.

Чтобы сервер мог обрабатывать сразу нескольких пользователей, можно использовать потоки. Для этого после получения соединения необходимо создать дочерний поток, который будет общаться с клиентом, а основной поток будет ожидать соединения следующего клиента.

Меньше теории — сразу переходим к делу и на примере увидим, как все будет работать, и какие мы получим недостатки. Открываем программу сервера, который мы написали в *разд. 5.8.1*, и добавляем новый класс-поток `TThread`. Назовем его `TTCPServerThread`. Код модуля вы можете увидеть в листинге 5.6. Из этого листинга я только удалил описание функции `SendString`, которую мы уже видели в *разд. 5.9.1*.

Листинг 5.6. Код потока, работающего с клиентом

```
unit ServerThreadUnit;
interface
uses Classes, Winsock2, Windows, Messages, Dialogs;

type
  TTCPServerThread = class(TThread)
  private
  protected
    procedure Execute; override;
  public
   clientSocket: TSOCKET;
  end;

implementation

uses MainUnit;

procedure TTCPServerThread.Execute;
var
 sBuff : array [0..255] of char;
 iSize:Integer;
begin
 while(true) do
  begin
   iSize := recv(clientSocket, sBuff, 1024, 0);
   if (iSize < 1) then
    break;

   SendString(clientSocket, 'OK');
  end;
 CloseSocket(clientSocket);
end;

end.
```

При запуске потока начинает работать процедура `Execute`. В этой процедуре мы запускаем бесконечный цикл, внутри которого сначала читаем данные из сокета, а потом отправляем в ответ строку `'OK'`. Если во время получения данных к нам пришло нулевое значение, то это может быть одно — клиент отключился и закрыл соединение, поэтому мы прерываем цикл.

Но в данном примере я сделал проверку немного по-другому — если результат меньше 1, то прерывается соединение. Если пришли корректные данные, то результат `recv` будет иметь положительное значение. Если соединение закрыто, то будет 0, и если произошла ошибка, то мы получим `SOCKET_ERROR`, которая равна −1. Таким образом, мы будем закрывать соединение в двух случаях — во время ошибки и при закрытии соединения.

Возвращаемся к основному потоку нашей программы. После начала прослушивания и до закрытия сокета все удаляем, а вместо этого пишем следующий код:

```
iSize := sizeof(clientaddr);
while true do
 begin
  sClient := accept(sServer, clientaddr, iSize);
  if sClient = INVALID_SOCKET then
   begin
    ShowMessage('Ошибка accept');
    exit;
   end;

  // Выводим сообщение
  memoInfo.Lines.Add('Подключение от:   '
        +inet_ntoa(clientaddr.sin_addr));

  // Создаем поток
  ServerThread:=TTCPServerThread.Create(true);
  ServerThread.clientSocket:=sClient;
  ServerThread.Resume;
 end;
```

Когда клиент подключился к серверу, мы создаем поток `TTCPServerThread` (обязательно в приостановленном режиме), записываем в него сокет клиента, через который мы можем общаться, и запускаем поток на выполнение.

Обратите внимание, что теперь наш сервер в бесконечном цикле ожидает запросов от клиента, а значит, программу можно будет завершить только снятием процесса. Это очень плохо, но в *разд. 5.9.3* мы будем рассматривать неблокирующие сокеты и сможем сделать корректное завершение программы.

Пока вроде бы все красиво и легко, так что же тут проблематичного? Дело в том, что если вы захотите полученные данные поместить в `Memo`-компонент на главной форме, то это окажется серьезным препятствием. При работе с визуальными компонентами из потока нужно быть аккуратным, чтобы не получить конфликта одновременного доступа. Логично будет создать отдельную функцию, из которой будет происходить обращение к главной форме, а функцию вызывать с помощью `synchronize`. Но в данном случае это не поможет.

Как работает функция `synchronize`? На самом деле, вызываемая в ней функция выполняется не в контексте потока `TThread`, а в контексте основного потока приложения. Таким образом, будет гарантировано, что к визуальному компоненту никто другой не будет обращаться в данный момент.

Попробуйте внимательно обдумать, с какой проблемой может столкнуться программа. Ничего не видите? Главный поток нашего сервера, увидев клиентское соединение, создаст поток для работы клиента и тут же вызовет функцию `accept` в ожидании следующего клиента. Главный поток заблокируется сетевой функцией, а дочерний поток будет обрабатывать запросы клиента. Получив данные, мы вызываем в `synchronize` функцию обновления визуального компонента, которая должна работать в контексте основного потока. Но наш основной поток заблокирован, а значит, функция не сможет быть выполнена.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 5\TCPServer1\ вы можете найти пример этой программы.

5.9.3. Неблокирующий сокет

Если сокет не будет блокировать работу потока (основного процесса или дочернего), то мы решим проблему зависания и синхронизации данных между дочерним и главным потоками. Итак, сейчас мы улучшим TCP-сервер, который пишем на протяжении *разд. 5.8* и *5.9* так, чтобы окно не блокировалось и могло реагировать на запросы пользователя.

Для начала добавим на форму сервера одну кнопку `TButton`, по нажатию которой сервер будет останавливаться и больше не будет принимать новых

сообщений от клиента. В свойстве `Name` напишем `btStop`, а по нажатию кнопки напишем всего одну строку кода:

```
btStop.Enabled:=false;
```

Теперь, перед запуском сервера будем делать кнопку останова сервера активной, а в бесконечном цикле, где у нас принимаются входящие сообщения от пользователя, мы будем проверять, если кнопка не активна, то ее нажали и нужно прервать работу сервера.

Когда мы работали в блокирующем режиме, то этот трюк был бесполезен, потому что ввод со стороны пользователя игнорировался, и никакую кнопку нельзя было нажать. Сейчас же все будет работать, нужно только сделать еще несколько телодвижений в ритме танго, а именно переписать половину кода логики работы сервера. Для нажатия кнопки запуска пишем код, который показан в листинге 5.7. По сравнению с предыдущей версией добавлено много нового, поэтому я решил привести полный код процедуры и расставил комментарии там, где что-то добавлено.

Листинг 5.7. Работа с сокетом в неблокирующем режиме

```
procedure TForm1.btStartClick(Sender: TObject);
var
 // Старые переменные опущены, а приведены только новые
 iMode        : Cardinal;
 ServerThread : TTCPServerThread;
 ReadSet      : TFDSet;
 ReadySock    : Integer;
 tv           : TTimeval;
begin
 sServer := socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
 if sServer = INVALID_SOCKET then
  begin
   MessageBox(0, 'Ошибка создания сокета', 'Ошибка', 0);
   exit;
  end;

 // Переход в неблокирующий режим
 iMode := 1;
 ioctlsocket(sServer, FIONBIO, iMode);
```

```
// Далее я опустил код заполнения структуры адреса, вызов
// функций bind и listen, потому что они не изменились
...

iSize := sizeof(clientaddr);

// Заполняем структуру TTimeval, в которой указываем время ожидания
tv.tv_sec := 1;
tv.tv_usec := 0;

btStop.Enabled:=true;
while True do
 begin
  // Обнуляем набор сокетов и добавляем наш
  FD_ZERO(ReadSet);
  FD_SET(sServer, ReadSet);

  // Запускаем select в ожидании соединения
  ReadySock := select(0, @ReadSet, nil, nil, @tv);
  if (ReadySock = SOCKET_ERROR) then
   exit;

  // Если наш сокет сработал, то пришло соединение
  if (FD_ISSET(sServer, ReadSet)) then
   begin
    sClient := accept(sServer, clientaddr, iSize);
    if sClient = INVALID_SOCKET then
     begin
      ShowMessage('Ошибка accept');
      exit;
     end;

    memoInfo.Lines.Add('Подключение от: '+
         inet_ntoa(clientaddr.sin_addr));
    ServerThread:=TTCPServerThread.Create(true);
    ServerThread.clientSocket:=sClient;
```

```
     ServerThread.Resume;
    end;
   // Обработать сообщения пользователей
   Application.ProcessMessages;
   // Если нажата кнопка останова, то прерываем работу сервера
   if btStop.Enabled=false then
    break;
  end;
 closesocket(sServer);
end;
```

Для перехода в неблокирующий режим есть несколько функций. Одна из них использована в этом примере — `ioctlsocket` и мы ее вызываем сразу после создания сокета. В общем виде `ioctlsocket` выглядит следующим образом:

```
function ioctlsocket(
 const s: TSocket;
 const cmd: DWORD;
 var arg: u_long
): Integer; stdcall;
```

Функция предназначена для управления режимом ввода/вывода сокета и имеет три параметра:

- сокет, режим которого нужно изменить;
- изменяемый параметр. Чтобы перевести сокет в неблокирующий режим, здесь нужно указать константу `FIONBIO`;
- новое значение режима. Если во втором параметре указана константа `FIONBIO`, то для перехода в блокирующий режим здесь нужно указать 0, любое другое значение выключает блокировку.

В неблокирующий режим можно перейти и с помощью вызова функций `WSAAsincSelect` и `WSAEventSelect`, которые мы использовали при написании быстрого сканера сети.

Тут необходимо сделать еще одно замечание — третий параметр функции `ioctlsocket` в первой версии WinSock имел тип `Integer`, а во второй уже `Cardinal`, поэтому будьте внимательны при выборе типа данных.

После того, как мы запустили порт на прослушивание, и до того, как запустили бесконечный цикл приема соединений, подготовим структуру `TTimeval`, чтобы указать максимальное время ожидания, изменения состоя-

ния сокета. Эту структуру мы уже тоже использовали при написании быстрого сканера.

Далее, уже знакомое нам обнуление набора сокетов, заполнение набора нашим единственным серверным сокетом и запуск ожидания с помощью `select`.

При вызове `select` мы могли бы не указывать время ожидания, но в нашем случае эта функция будет ожидать подключения клиента бесконечно, что снова приведет к блокировке. Ограничив ожидание одной секундой, функция `select` будет хотя бы раз в секунду прерывать свою работу. В этот момент мы должны проверить, если есть подключение со стороны клиента, то следует создать необходимый поток для обмена сообщениями с клиентом. Если соединений нет, то ничего этого делать не нужно.

В любом случае, на каждом этапе цикла будут выполнены еще три строки кода:

```
Application.ProcessMessages;
if btStop.Enabled=false then
  break;
```

В первой строке мы вызываем метод `ProcessMessages` нашего приложения, чтобы оно обработало сообщения в системе, которые накопились за время работы функции `select`. После этого проверяем состояние кнопки останова. Если она не активна, то прерываем цикл.

Таким образом, хотя бы раз в секунду приложение будет реагировать на события Windows, и не будет ощущения зависания.

Теперь посмотрим, какие изменения произошли в потоке обмена данными с клиентом. После чтения данных (вызова функции `recv`) добавляем следующие две строки:

```
recievedString:=copy(sBuff, 1, iSize);
synchronize(ShowBuffer);
```

Здесь мы в переменную `recievedString` (она должна быть объявлена как `String` в разделе `private` нашего потока) копируем полученные данные. После этого в синхронизированном режиме вызываем процедуру `ShowBuffer`. Эта процедура должна быть в вашем потоке и иметь следующий вид:

```
procedure TTCPServerThread.ShowBuffer;
begin
 Form1.memoInfo.Lines.Add(recievedString);
end;
```

Здесь мы просто выводим в Memo-компонент главной формы полученную строку. Так как основной процесс приложения теперь не блокируется, то эта функция сможет корректно выполниться.

Вот теперь наш пример намного лучше и интереснее. Но возникает вопрос: в каком режиме работает сокет, который мы получаем для работы с клиентом после вызова функции accept? Сокет сервера мы перевели в неблокирующий режим, а в каком будет работать этот? Ответ банален — все сокеты по умолчанию работают в блокирующем режиме, и этот тоже будет по умолчанию блокирующим. Если вы захотите работать с клиентом в неблокирующем режиме, то этот сокет нужно тоже перевести с помощью ioctlsocket.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 5\TCPServer2\ вы можете найти пример этой программы.

5.9.4. Обмен через сообщения

Следующий метод работы с сокетами в неблокирующем режиме основан на сообщениях Windows. Для иллюстрации работы с сообщениями я решил написать пример работы с FTP-протоколом, т. е. небольшой FTP-клиент.

Для тестирования примера вам понадобится FTP-сервер. Простейший сервер можно найти на компакт-диске в каталоге Документация\NetCoding\FTP Server\. Там же есть и исходные коды, а в файле Документация\NetCoding\ftpserver.pdf можно найти статью, в которой описывается, как самому написать подобную программу.

Для реализации примера я набросал небольшой класс TFTPSocket, в котором реализованы некоторые возможности, но далеко не все.

Итак, откройте файл Headers\Flenov\FTPSocket.pas на компакт-диске, и давайте рассмотрим, как в нем реализована работа с сокетом. Весь класс я не буду приводить в книге, потому что он большой, но самое интересное и необходимое мы увидим. А начнем с метода ConnectToServer, который показан в листинге 5.8. В листинге представлено самое необходимое, а некоторые информационные строчки кода я убрал.

Листинг 5.8. Метод подключения к FTP-серверу

```
procedure TFTPSocket.ConnectToServer;
var
 addr:TSockAddr;
```

```
begin
 FSocket := socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
 if FSocket = INVALID_SOCKET then
  begin
   RaiseException('Create socket error');
   exit;
  end;

 addr.sin_addr.s_addr := htonl(INADDR_ANY);
 addr.sin_family := AF_INET;
 addr.sin_port := htons(FPort);
 addr.sin_addr := LookupName;

 if (connect(FSocket, @addr, sizeof(addr)) = SOCKET_ERROR) then
  begin
   FState:=wsClosed;
   exit;
  end;
 FState:=wsConnected;
 RaiseInformation('Connection status: OK');

 ResetEvent(Event);
 WSAAsyncSelect(FSocket, ParentForm.Handle, WM_NETMESSAGE,
   FD_ACCEPT+FD_CLOSE+FD_READ+FD_WRITE);
 WaitForSingleObject(Event, 5000);

 SendStr('USER '+FUsername);
end;
```

Сначала идет знакомый нам код создания сокета, заполнения структуры адреса и подключение к серверу. Все это время сокет находится в блокирующем режиме, а значит, функция `connect` заблокирует работу программы, пока соединение с сервером не завершится.

Только после этого мы выполняем функцию `WSAAsyncSelect`, которая переводит сокет в неблокирующий режим, а заодно запрашивает у системы воз-

можность работы с сокетом через Windows-сообщения. У функции четыре параметра:

- сокет, с которым нужно работать через сообщения;
- окно, которому система будет отправлять сообщения;
- сообщение;
- события, на которые нужно реагировать. Здесь может быть сочетание из следующих флагов (я приведу только основные и часто используемые):
  - `FD_READ` — сокет готов к чтению данных;
  - `FD_WRITE` — сокет готов к записи;
  - `FD_ACCEPT` — есть входящее соединение;
  - `FD_CONNECT` — соединение с сервером завершилось;
  - `FD_CLOSE` — сокет закрыт.

Несколько замечаний по поводу второго параметра. Сообщения Windows — это числа. Вы можете взять любое число и зарегистрировать его в качестве сетевого сообщения. В случае срабатывания одного из событий, указанного в четвертом параметре, система будет направлять указанному окну событие с вашим номером.

Но ведь в системе уже есть куча зарезервированных номеров для событий Windows. Как же тогда выбрать такое число, чтобы не было проблем? Очень просто, число должно быть больше константы `WM_USER`. В данном примере мы создадим новую константу `WM_NETMESSAGE`, которая будет равна `WM_USER` плюс единица. Для этого в моем модуле в начале есть следующие две строки:

```
const
 WM_NETMESSAGE=WM_USER+1;
```

Следующая проблема заключается во втором параметре — события будут передаваться окну. В данном случае я специально написал класс, чтобы вы увидели возможную проблему. Класс — не компонент и не принадлежит окну, а значит, он не может принимать события. Чтобы получать события, можно разработать невизуальный компонент, который будет помещаться на форме и принадлежать ей, а можно поступить так, как это сделал я.

Для начала, среди членов класса `TFTPSocket` я завел переменную `ParentForm` типа `TForm`. Любое приложение, создавая экземпляр `TFTPSocket`, должно сюда прописывать форму, которая будет ловить события и потом передавать их классу. Теперь, для обработки событий в классе я объявил процедуру следующего вида:

```
procedure WM_NetMsg(Var M : TMessage);
```

Реализацию процедуры можно увидеть в листинге 5.9. Это всего лишь обработка событий, но не ловушка. Данная процедура не будет автоматически реагировать на события.

Листинг 5.9. Процедура обработки событий

```
procedure TFTPSocket.WM_NetMsg(var M: TMessage);
var
 iRet:Integer;
 sRecvBuff : array [0..255] of char;
begin
 // Проверяем, какое событие произошло
 case M.LParam of
  // Если есть данные для чтения, то читаем их
  FD_READ:
   begin
    iRet := recv(m.wParam, sRecvBuff, 1024, 0);
    if (iRet = SOCKET_ERROR) then
     begin
      RaiseException('Data receive error');
      exit;
     end;
    RaiseInformation(sRecvBuff);
    FLastCommand:=sRecvBuff;
   end;

  // Если сокет готов к записи, то выставляем событие
  FD_WRITE:
   begin
    SetEvent(Event);
   end;

  // Если сокет закрыт, то вызываем функцию, которая
  // освободит ресурсы и проинформирует всех, кого надо
  FD_CLOSE:
```

```
   begin
    DisconnectFromServer;
   end;
 end;
end;
```

А вот теперь в приложении, которое будет использовать данный класс, а точнее в форме, нужно объявить следующую процедуру:

```
procedure WM_NetMsg(Var M : TMessage); message WM_NETMESSAGE;
```

Здесь мы объявляем процедуру, которая должна реагировать на событие `WM_NETMESSAGE`. Когда в системе будет происходить это событие, то система будет передавать его окну, а оно автоматически вызовет `WM_NetMsg`.

Процедура `WM_NetMsg` должна выглядеть следующим образом:

```
procedure TCyDFTPPEForm.WM_NetMsg(var M: TMessage);
begin
 FSocket.WM_NetMsg(M);
end;
```

Переменная `FSocket` в данном случае это экземпляр класса `TFTPSocket`. Мы всего лишь функцию `WM_NetMsg` этого класса (мы ее видели в листинге 5.9), а она уже обрабатывает события.

Теперь на практике посмотрим, как можно использовать данный класс для написания FTP-клиента, который будет уметь подключаться к серверу и авторизоваться.

5.9.5. Пример работы через сообщения

Теперь попробуем написать пример, который будет использовать класс работы с сокетами, описанный в *разд. 5.9.4*. Создайте форму и поместите на нее две кнопки для соединения с сервером и отключения. Помимо этого нам понадобится компонент `TMemo` (назовем его `reMessages`), в котором будут отображаться сообщения о ходе работы с сетью.

Теперь добавим в раздел `uses` наш модуль `FTPSocket`, а в разделе `private` описания формы добавим следующее:

```
FSocket:TFTPSocket;
procedure FTPError(Sender: TObject; Msg : String);
procedure FTPInfo(Sender: TObject; Msg : String);
procedure FTPConnected(Sender: TObject);
```

В первой строке объявляем переменную типа `TFTPSocket`, через которую будем работать с классом. После этого идут три процедуры обратного вызова:

- для перехвата ошибок, в качестве второго параметра процедура будет получать текст ошибки;
- для получения информационных сообщений, в качестве второго параметра процедура будет получать текст сообщения;
- для определения, когда соединение с сервером завершилось удачно.

Все эти три процедуры будут переданы классу `TFTPSocket` и будут вызываться как обработчики событий. Точнее сказать, они даже реализованы как обработчики событий, просто перед нами не компонент, а простой класс, поэтому обработчики придется назначать программно.

Теперь, по нажатию кнопки соединения с сервером пишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 // Создаем экземпляр класса
 FSocket := TFTPSocket.Create(self);

 // Связываем обработчики событий
 FSocket.OnError:=FTPError;
 FSocket.OnInfo:=FTPInfo;
 FSocket.OnConnected:=FTPConnected;

 // Назначаем имя пользователя и пароль
 FSocket.Username:='Anonymous';
 FSocket.Password:='test@test.ru';

 // Назначаем порт и адрес сервера
 FSocket.Port := 21;
 FSocket.RemoteAddress := 'localhost';

 // Вызываем функцию соединения с сервером
 FSocket.ConnectToServer;
end;
```

Код очень прост, и с ним легко разобраться по комментариям. В обработчиках событий от класса нужно всего лишь вывести в Memo-компонент передаваемый тест сообщения/ошибки:

```
reMessages.Lines.Add(Msg);
```

Вроде бы пример готов. Ах да, вы не забыли, что перед нами класс, который не является окном, а значит, не может обрабатывать события от системы? Обработчик события мы должны реализовать в своей форме и передавать события классу. Для этого в разделе private или public описания нашей формы добавим объявление метода:

```
procedure WM_NetMsg (Var M : TMessage); message WM_NETMESSAGE;
```

А реализация его будет выглядеть следующим образом:

```
procedure TForm1.WM_NetMsg(var M: TMessage);
begin
 FSocket.WM_NetMsg(M);
end;
```

Пример готов. Мы реализовали самую маленькую часть FTP-протокола — соединение с сервером и авторизацию. Но нарастить возможности программы не так уж и сложно. Нужно только организовать возможность отправки на сервер необходимых FTP-команд и реализовать соответствующую реакцию.

Таким образом, очень легко реализовать практически любой сетевой протокол, особенно если он работает через текстовые сообщения. На компакт-диске в каталоге Документация\Protocols вы найдете несколько интересных примеров, которые реализуют такие протоколы, как POP3, SMTP и HTTP.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 5\FTPClient\ вы можете найти пример этой программы.

5.10. Как написать троян

В первом издании был один раздел, посвященный написанию собственного троянского коня. Пример был написан с помощью встроенных компонентов, а в этом издании книги мы используем готовые компоненты по минимуму. Именно поэтому данный раздел был убран из книги, и он перелетел на компакт-диск в каталог Документация\NetCoding.

Прочитав электронный вариант статьи, вы без проблем сможете написать троян и без компонентов, а на чистом Windows API, что будет намного лучше. Сейчас же мы не будем писать полноценный пример, а только ограничимся теорией.

Принцип работы достаточно простой. Пишутся две программы — клиент и сервер. Серверная часть забрасывается жертве и выполняет следующие функции:

- открывает порт и ожидает соединения пользователей;
- когда к порту произошло подключение, может проверяться пароль, чтобы никто, кроме автора троянского коня, не смог управлять жертвой;
- сервер принимает запросы клиента и выполняет соответствующие действия.

Программа-клиент остается у хакера и с ее помощью он подключается к компьютеру жертвы и управляет им.

Команды, передаваемые от клиента к жертве, могут быть простыми текстовыми строками, как в протоколе FTP. Например, чтобы выключить компьютер жертвы, можно по сети отправить `turn off`. Программа-сервер, получив такую строку, может выключить компьютер жертвы.

Таким образом, в зависимости от того, какие команды и возможности вы реализуете в троянской программе, то и сможете сделать с жертвой. Троянские программы действительно были опасны, но это было очень давно, потому что нынешние системы защищаются сетевыми экранами. Даже в Windows, начиная с версии XP, появился простой, но достаточно действенный сетевой экран. Конечно, существует множество способов обойти встроенную защиту, поэтому большинство пользователей устанавливает дополнительный уровень защиты в виде сетевого экрана стороннего разработчика.

Не знаю почему, но со средствами защиты у Microsoft серьезные проблемы. Большинство даже бесплатных утилит работают намного лучше, а про такие разработки, как Agnitum Outpost Pro, даже мухи молчат. Эта разработка обходит работу Microsoft на несколько лет вперед.

5.11. Работа с UDP

Теперь поговорим о UDP-протоколе. Как мы уже знаем, он не устанавливает никакого соединения с сервером, и мы не можем гарантировать доставку данных. Но т. к. нет соединения, то как нужно обмениваться сообщениями? Работа с UDP очень похожа на TCP, поэтому мы не будем так подробно останавливаться на этом вопросе, но напишем небольшой пример.

5.11.1. UDP-сервер

Логика работы UDP-сервера банальна, дальше некуда. Необходимо выполнить только два действия — вызвать функции: `socket`, чтобы создать сокет, и `bind`, чтобы связать сокет с определенным портом и интерфейсом. Этого достаточно. Прослушивание порта запускать не нужно. Достаточно только вызвать функцию `recvfrom`, чтобы начать считывать данные. Эту функцию мы еще не рассматривали, давайте взглянем на нее поближе:

```
function recvfrom(
 s: TSocket;
 var buf;
 len: Integer;
 flags: Integer;
 var from: TSockAddr;
 var fromlen: Integer
): Integer; stdcall;
```

Функция имеет следующие параметры:

- `s` — открытый сокет, с которого нужно читать данные;
- `buf` — буфер, для хранения полученных данных;
- `len` — длина буфера;
- `flags` — флаги. Здесь можно указать один или сочетание из следующих флагов:
  - `MSG_PEEK` — прочитать данные из сокета, но не удалить их. Это значит, что если еще раз вызвать эту функцию, то она вернет те же самые данные, потому что они останутся в системном буфере, пока мы не вызовем функцию без этого флага;
  - `MSG_OOB` — обработать внеполосные данные (out-of-band);
- `from` — здесь можно указать структуру `sockaddr`, в которой мы получим адрес хоста, приславшего данные;
- `fromlen` — если заполнен параметр `from`, то здесь нужно указать длину буфера адреса.

Отдельного обсуждения заслуживают флаг `MSG_OOB` и внеполосные данные. Этот флаг может использоваться как при отправке данных, так и при получении. Если во время отправки был указан этот флаг, то для чтения тоже нужно его указать. Чтобы проверить наличие внеполосных данных, можно вызвать функцию `select` с указанием третьего параметра (`exceptfds`) или функцию `WSAASYNCSELECT` с указанием `FD_OOB`.

Кстати, при создании сокета следует во втором параметре указать тип `SOCK_DGRAM`, что означает работу через дейтаграммы без установки соединения. В последнем параметре нужно указать константу `IPPROTO_UDP`, которая соответствует UDP-протоколу. Например:

```
socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

Теперь посмотрите, как это выглядит в виде кода. Сервер UDP представлен в листинге 5.10.

Листинг 5.10. Сервер UDP

```
procedure TForm1.Button1Click(Sender: TObject);
var
 sServer: TSOCKET;
 localaddr, clientaddr: sockaddr_in;
 sRecv: array [0..255] of char;
 iSize: Integer;
begin
 sServer := socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
 if sServer = INVALID_SOCKET then
  exit;

 localaddr.sin_addr.s_addr := htonl(INADDR_ANY);
 localaddr.sin_family := AF_INET;
 localaddr.sin_port := htons(10000);

 if bind(sServer, @localaddr, sizeof(localaddr)) = SOCKET_ERROR then
  exit;

 iSize:=sizeof(clientaddr);
 if recvfrom(sServer, sRecv, 255, 0,
     clientaddr, iSize)<>SOCKET_ERROR then
  Memo1.Lines.Add(sRecv);

 closesocket(sServer);
end;
```

Что тут сказать? Код говорит сам за себя. Добавить что-то еще просто невозможно.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 5\UDPServer\ вы можете найти пример этой программы.

5.11.2. UDP-клиент

Теперь посмотрим, как можно реализовать UDP-клиент. Тут все намного проще. Дело в том, что для клиента достаточно только создать сокет, заполнить структуру `sockaddr` и можно вызывать функцию `SendTo` для отправки данных. Эта функция выглядит следующим образом:

```
function sendto(
 s: TSocket;
 var buf;
 len,
 flags: Integer;
 var addrto:
 TSockAddr; tolen: Integer
): Integer; stdcall;
```

Посмотрим на параметры этой функции:

- `s` — открытый сокет, через который будет происходить отправка;
- `buf` — буфер, данные которого нужно отправить;
- `len` — длина буфера;
- `flags` — флаги. Здесь можно указать один или сочетание из следующих флагов:
  - `MSG_DONTROUTE` — не использовать маршрутизацию. Провайдер сокетов может игнорировать этот флаг, если для протокола по умолчанию запрещена маршрутизация;
  - `MSG_OOB` — обработать внеполосные данные (out-of-band);
- `addrto` — здесь нужно указать структуру `sockaddr`, в которой должен быть указан адрес и порт сервера получателя;
- `fromlen` — длина адреса в параметре `addrto`.

Пример UDP-клиента представлен в листинге 5.11.

Листинг 5.11. Пример UDP-клиента

```
var
 sClient : TSOCKET;
 serveraddr : sockaddr_in;
 sSendBuf : array [0..255] of char;
begin
 sClient := socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
 if sClient = INVALID_SOCKET then
  exit;

 serveraddr.sin_addr.s_addr := htonl(INADDR_ANY);
 serveraddr.sin_family := AF_INET;
 serveraddr.sin_port := htons(10000);
 serveraddr.sin_addr := LookupName(Edit1.Text);

 CopyMemory(@sSendBuf, PChar(Edit2.Text), Length(Edit2.Text));
 SendTo(sClient, sSendBuf, 16, 0, serveraddr, SizeOf(serveraddr));

 closesocket(sClient);
end;
```

Примечание

На компакт-диске в каталоге \Примеры\Глава 5\UDPClient\ вы можете найти пример этой программы.

5.11.3. Замечания

Меня пару раз спрашивали, а как может сервер ответить на запрос клиента? Как отправить ответ? В случае с TCP-протоколом происходило соединение двух машин (или соединение внутри машины, когда клиент и сервер запущены на одном компьютере), и через это соединение мы могли общаться в обе стороны. В случае с UDP соединения нет, и UDP-клиент не может получать данные. Чтобы реализовать двунаправленное соединение, необходимо, чтобы на сервере тоже был запущен UDP-сервер, который будет ожидать данные. Получается, что программа должна быть и клиентом, и сервером одновременно.

Но тут возникает серьезная проблема, если программа запущена дважды на одном компьютере и пытается обмениваться данными внутри одной машины. Две программы не могут одновременно открыть порт для ожидания, а если и откроют, то как узнать, какая из них должна прочитать полученные данные? Система запутается, кто отправил данные, а кто их ожидает?

Если общение происходит по сети и обмен данными организуется между двумя компьютерами, то тут уже проблем никаких.

5.12. HTTP-клиент

На компакт-диске в каталогах \Документация\NetCoding и \Документация\Protocols вы найдете множество примеров использования различных протоколов посредством сокетов. Но один протокол (HTTP) я решил рассмотреть более подробно. Дело в том, что он наиболее популярен и про него чаще всего спрашивают читатели. Сейчас мы напишем пример, который будет обращаться к Web-серверу и получать запрашиваемую страницу. При этом наш клиент будет работать как напрямую, так и через прокси-сервер.

Да, до настоящего Web-браузера нашему клиенту еще далеко, но мы его создадим для написания других приложений, работающих с серверами по протоколу HTTP.

Итак, для реализации примера нам понадобятся следующие компоненты на форме:

- поле ввода для указания URL к Web-странице;
- кнопка, по нажатию которой будет начинаться сканирование;
- компонент `Checkbox`, который будет определять, нужно ли работать через прокси-сервер;
- поле для ввода IP-адреса прокси-сервера;
- поле для указания порта прокси-сервера;
- компонент `Memo`, в который будет помещаться загруженная страница.

Пример моей формы вы можете увидеть на рис. 5.7.

Итак, по нажатию кнопки пишем следующий код:

```
Memo1.Lines.Assign(GetWEBPage(edURL.Text, 1));
```

По нажатию кнопки мы всего лишь вызываем функцию `GetWEBPage`, которую скоро напишем. Результат этой функции — объект `TStrings`, в котором будут находиться строки, загруженной страницы. В качестве параметров мы должны передать URL и метод получения данных. Вы, наверно, знаете, что существуют два метода получения данных от сервера — `GET` и `POST`. Мы реа-

лизуем функцию `GetWEBPage` универсально. Если во втором параметре указана 1, то будет использоваться метод `GET`, а если 2, то метод `POST`.

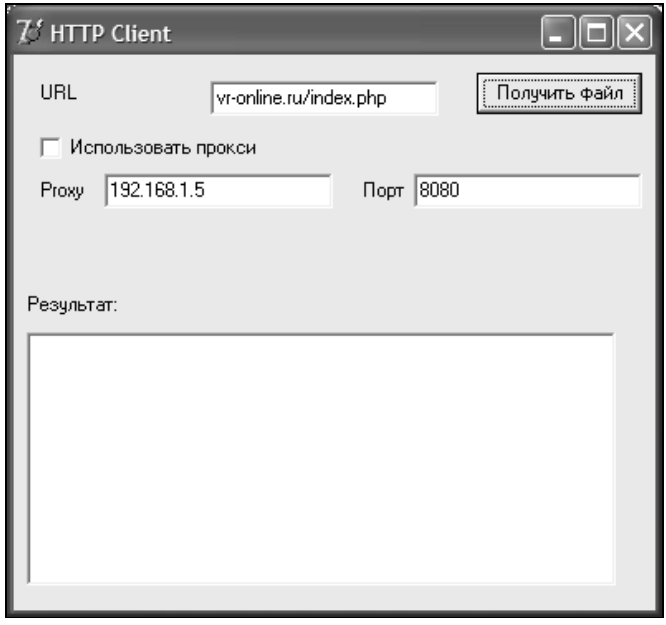


Рис. 5.7. Пример формы HTTP-клиента

Теперь пора посмотреть на функцию `GetWEBPage`. Ее реализацию вы можете увидеть в листинге 5.12.

Листинг 5.12. Функция загрузки страницы с Web-сервера

```
function TTCPClientForm.GetWEBPage(addr: String;
   Method:Integer):TStringList;
var
 localaddr : sockaddr_in;
 iMode, iSize:Integer;
 rfds : TFDSET;
 Buff: array [0..1024] of char;
 stClient:TSocket;
 testingserver, servername, portname:String;
 timeout:TTimeVal;
```

```
begin
 // Создаем объект для хранения результата
 Result:=TStringList.Create;
 // Создание сокета
 stClient := socket(AF_INET, SOCK_STREAM, 0);
 if stClient = INVALID_SOCKET then
  begin
   MessageBox(0, Ошибка', 'Error', MB_OK);
   exit;
  end;

 // Сохраняем URL в testingserver и выделяем имя сервера
 testingserver:=addr;
 if pos('// ', testingserver)>0 then
  delete(testingserver, 1, pos('// ', testingserver)+1);
 if pos('/', testingserver)>0 then
  delete(testingserver, pos('/', testingserver), 255);

 // Проверяем соединение
 if cbUseProxyServer.Checked then
  begin
   servername:=edExtProxyAddr.Text;
   portname:=edExtProxyPort.Text;
  end
 else
  begin
   servername:=testingserver;
   portname:='80';
  end;

 // Заполняем структуру адреса
 localaddr.sin_addr := LookupName(servername);
 localaddr.sin_family := AF_INET;
 localaddr.sin_port := htons(StrToIntDef(portname, 80));
 if connect(stClient, @localaddr, sizeof(localaddr))<>0 then
```

```
  begin
   MessageBox(0, 'Ошибка соединения', 'Error', MB_OK);
   exit;
  end;


 // Посылаем запрос
 if Method=1 then
  SendStr(stClient, 'GET '+addr+' HTTP/1.0'+#13+#10+'Content-Type:
text/html'+#13+#10+'User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Win-
dows NT 5.2; .NET CLR 1.1.4322; .NET CLR 2.0.50727)'+#13+#10+'Host:
'+testingserver+#13+#10+'Proxy-Connection: Keep-Alive'+#13+#10+#13+#10);
 if Method=2 then
  SendStr(stClient, 'POST '+addr+' HTTP/1.0'+#13+#10+'Content-Type:
text/html'+#13+#10+'User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Win-
dows NT 5.2; .NET CLR 1.1.4322; .NET CLR 2.0.50727)'+#13+#10+'Host:
'+testingserver+#13+#10+'Proxy-Connection: Keep-Alive'+#13+#10+#13+#10);


 // Переходим в неблокирующий режим
 iMode:=1;
 setsockopt(stClient, IPPROTO_TCP, TCP_NODELAY, @iMode, sizeof(integer));


 // Цикл получения данных
 while true do
  begin
   FD_ZERO(rfds);
   FD_SET(stClient, rfds);

   timeout.tv_sec:=10;
   if (select(0, @rfds, nil, nil, @timeout) <= 0) then
    exit;

   if(FD_ISSET(stClient, rfds)) then
    begin
     iSize := recv(stClient, buff, sizeof(buff), 0);

     if (iSize<1) then
      break;
```

```
     Result.Add(String(buff));
    end;
  end;
 CloseSocket(stClient);
end;
```

Сначала мы создаем сокет, для работы по протоколу TCP/IP, т. е. с установкой соединения. Да, HTTP использует протокол TCP/IP.

Теперь адрес URL копируем в переменную `testingserver`. Она будет хранить имя сервера, к которому будем обращаться. Если запрашиваемая пользователем страница имеет адрес **http://www.vr-online.ru/index.php**, то в переменную `testingserver` попадет имя сервера **www.vr-online.ru**. Именно к этому серверу мы должны будем подключаться.

Теперь проверяем, если нужно подключаться через прокси-сервер, то в переменные `servername` и `portname` попадают введенные пользователем адрес и порт прокси-сервера. Если необходимо подключаться напрямую, то в эти же переменные попадают имя сервера из переменной `testingserver` и 80-й порт. Именно этот порт использует большинство Web-серверов.

Получается, что при работе через прокси мы отправляем запрос не Web-серверу, а прокси-серверу, который анализирует запрос и переправляет нашу просьбу к Web-серверу.

Теперь заполняем структуру адреса. Адрес и порт у нас хранятся в `servername` и `portname`, и именно эти значения мы используем. Заполнив адрес, можно подключаться к серверу функцией `connect`.

Теперь нужно отправить серверу HTTP запрос на получение страницы. В зависимости от выбранного метода, серверу отправляется `POST`- или `GET`-запрос. Как выглядит этот запрос? Давайте рассмотрим его на примере `GET`:

```
GET http://www.vr-online.ru/index.php HTTP/1.0
Content-Type: text/html
User-Agent: Mozilla/4.0
Host: www.vr-online.ru
Proxy-Connection: Keep-Alive
```

Этот запрос отправляется построчно и в конце каждой строки должна идти пара символов `#13` и `#10`, а в конце запроса эта пара символов должна идти дважды, т. е. должна быть отправлена пустая строка.

Самое важное тут находиться в первой и четвертой строках. После ключевого слова `GET` в первой строке идет URL страницы, которую мы запраши-

ваем. В четвертой строке после слова `Host` нужно передать имя сервера, в нашем случае это **www.vr-online.ru**. В принципе, **www.** можно опустить.

После отправки запроса переводим сокет в неблокирующий режим. В принципе, моя реализация все равно блокирует приложение, когда вызывается функция `select`, но вы можете улучшить этот пример. Мы уже видели, как можно не блокировать приложение и обрабатывать сообщения Windows во время ожидания ответа от сервера.

Далее идет бесконечный цикл, который читает данные с сокета. Этот цикл будет выполняться, пока функция `select` не вернет значение меньшее или равное 0, т. е. когда передача всех данных завершится. В цикле мы ожидаем очередную порцию данных, а получив ее, добавляем в результирующий набор.

Для отправки данных на сервер используется функция `SendStr`. Ее вы можете увидеть в листинге 5.13.

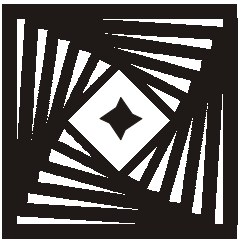
Листинг 5.13. Функции получения данных

```
procedure SendStr(s:TSocket; str:String);
var
 sRecvBuff : array [0..1024] of char;
 TempStr : AnsiString;
begin
 TempStr:=str+#13+#10;
 CopyMemory(@sRecvBuff, PChar(TempStr), Length(TempStr));
 send(s, sRecvBuff, Length(TempStr), 0);
end;
```

Как видите, ничего сложного в этом нет. Большинство популярных протоколов Интернета (FTP, POP3, SMTP) также не представляют никакой сложности. Нужно только знать, какие команды нужно отправлять серверу, а они в основном текстовые и не сложные.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 5\HTTPClient\ вы можете найти пример этой программы.

Глава 6



Железная мастерская

Все, что я буду описывать в этой главе, касается железа. Я покажу, как получать информацию о разных устройствах компьютера и как все это красиво преподнести пользователю.

Полученную информацию о железе можно использовать для защиты своих программ от копирования, как это делает Microsoft. Вы можете использовать любые уникальные данные (например, серийный номер диска) для проверки легальности копии при установке или запуске ваших программ. Я не считаю такую защиту надежной, но она очень хорошая и весьма простая.

Помимо этого, достаточно подробно будет описана работа с COM-портами (RS-232), которые часто используются на предприятиях для работы с различным оборудованием. Я сталкивался по работе с несколькими промышленными предприятиями, и на всех хоть где-то использовались программы, которые через COM-порт собирали данные с устройств или просто наблюдали за работой оборудования.

Несмотря на то, что в современных компьютерах COM-порт уже не устанавливают, и в моем ноутбуке его нет, тема остается актуальной, потому что модемы видны в системы именно как COM-устройства.

6.1. Общая информация о компьютере и ОС

Отличие данной главы от предыдущих глав состоит в том, что здесь мы будем писать не много маленьких примеров, а один, но большой. В одной программе мы объединим получение информации о различных параметрах системы.

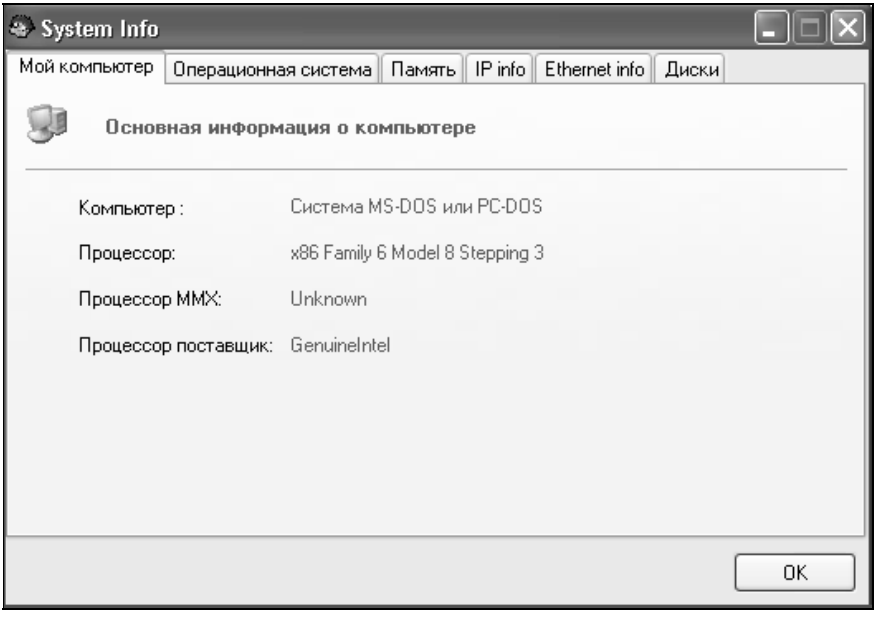


Рис. 6.1. Основная информация о компьютере

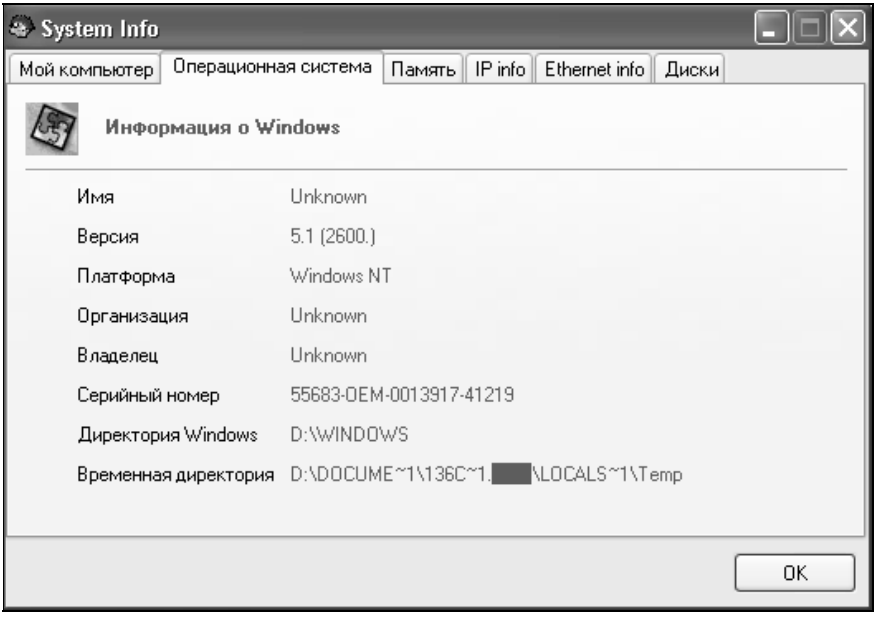


Рис. 6.2. Основная информация об операционной системе

Начнем рассмотрение с самого простого — получение общей информации о компьютере и установленной ОС. Вкладки, которые отображают эту информацию, вы можете увидеть на рис. 6.1 и 6.2.

Для заполнения данных, расположенных на этих двух вкладках, используется процедура `GetCompInfo`. Мы вызовем эту функцию по событию `OnShow` для главной формы, чтобы информация читалась при старте программы. Код процедуры можно увидеть в листинге 6.1.

Листинг 6.1. Сбор информации о системе

```
procedure TSystemInfoForm.GetCompInfo;
var
 SystemIniFile:TIniFile;
 RegFile:TRegIniFile;
 PathArray : array [0..255] of char;
 OSVersion: TOSVersionInfo;
begin
// Компьютер
 SystemIniFile:=TIniFile.Create('System.ini');
 ComputerLabel.Caption:=SystemIniFile.ReadString('boot.description',
   'system.drv', 'Unknown');
 SystemIniFile.Free;
 RegFile:=TRegIniFile.Create('Software');
 RegFile.RootKey:=HKEY_LOCAL_MACHINE;
 RegFile.OpenKey('hardware',false);
 RegFile.OpenKey('DESCRIPTION',false);
 RegFile.OpenKey('System',false);
 RegFile.OpenKey('CentralProcessor',false);
 ProcessorLabel.Caption:=RegFile.ReadString('0','Identifier','Unknown');
 MMXIdentifierLabel.Caption:=RegFile.ReadString('0','MMXIdentifier',
  'Unknown');
 VendorIdentifierLabel.Caption:=RegFile.ReadString('0',
  'VendorIdentifier','Unknown');
 RegFile.CloseKey;
// ОС
 OSVersion.dwOSVersionInfoSize := SizeOf(OSVersion);
 if GetVersionEx(OSVersion) then
```

```
  begin
   VersionLabel.Caption:= Format('%d.%d (%d.%s)',
     [OSVersion.dwMajorVersion, OSVersion.dwMinorVersion,
     (OSVersion.dwBuildNumber and $FFFF), OSVersion.szCSDVersion]);
   case OSVersion.dwPlatformID of
    VER_PLATFORM_WIN32s:VersionNumberLabel.Caption := 'Windows 3.1';
    VER_PLATFORM_WIN32_WINDOWS: VersionNumberLabel.Caption:='Windows 95';
    VER_PLATFORM_WIN32_NT: VersionNumberLabel.Caption := 'Windows NT';
   else
    VersionNumberLabel.Caption := '';
   end;
  end;
 RegFile.OpenKey('SOFTWARE',false);
 RegFile.OpenKey('Microsoft',false);
 RegFile.OpenKey('Windows',false);
 OSNameLabel.Caption:=RegFile.ReadString('CurrentVersion',
   'ProductName','Unknown');
  RegisteredOrganizationLabel.Caption:=
    RegFile.ReadString('CurrentVersion',
                       'RegisteredOrganization','Unknown');
 RegisteredOwnerLabel.Caption:=
    RegFile.ReadString('CurrentVersion','RegisteredOwner',
                       'Unknown');
 SerNumberEdit.Caption:=RegFile.ReadString('CurrentVersion','ProductId',
                       'Unknown');
 RegFile.Free;
 FillChar(PathArray, SizeOf(PathArray), #0);
 GetWindowsDirectory(PathArray,255);
 WindowsDirLabel.Caption:= Format('%s',[PathArray]);
 FillChar(PathArray, SizeOf(PathArray), #0);
 ExpandEnvironmentStrings('%TEMP%', PathArray, 255);
 TempDir.Caption:=Format('%s',[PathArray]);
end;
```

Информации очень много, поэтому будем рассматривать ее по частям.

6.1.1. Платформа компьютера

Сначала узнаем платформу компьютера, на котором запущена программа. Для этого выполняется следующий код:

```
SystemIniFile:=TIniFile.Create('System.ini');
ComputerLabel.Caption:=SystemIniFile.ReadString('boot.description',
    'system.drv',
    'Unknown');
SystemIniFile.Free;
```

Здесь инициализируется переменная `SystemIniFile`, которая объявлена принадлежащей типу `TIniFile` и предназначена для работы с ini-файлами. Мы загружаем в эту переменную файл System.ini. После этого из файла считывается параметр `system.drv` из раздела `boot.description`. Это и есть описание платформы, на которой запущена программа.

6.1.2. Информация о процессоре

Следующим этапом мы узнаем информацию о процессоре. Она находится в реестре по адресу:

```
HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcessor
```

Информация о первом процессоре находится в разделе `0`. Если есть еще процессоры, то они будут находиться в разделах 1, 2 и т. д. Здесь нас интересуют параметры `Identifier`, `MMXIdentifier`, `VendorIdentifier`.

6.1.3. Информация о платформе Windows

Теперь узнаем, на какой платформе Windows запущена программа. Существуют три основные платформы Windows:

- WIN32s — это Windows 3.1 с 32-разрядным расширением;
- WIN32_WINDOWS — это Windows 95/98/ME. Эта ветка уже прекратила свое развитие;
- WIN32_NT — это Windows NT/2000/XP и все последующие версии, основанные на ядре NT.

Для того чтобы узнать платформу, выполняется следующий код:

```
OSVersion.dwOSVersionInfoSize := SizeOf(OSVersion);
if GetVersionEx(OSVersion) then
```

```
begin
 VersionLabel.Caption:= Format('%d.%d (%d.%s)',
   [OSVersion.dwMajorVersion, OSVersion.dwMinorVersion,
   (OSVersion.dwBuildNumber and $FFFF), OSVersion.szCSDVersion]);
 case OSVersion.dwPlatformID of
  VER_PLATFORM_WIN32s:VersionNumberLabel.Caption := 'Windows 3.1';
  VER_PLATFORM_WIN32_WINDOWS: VersionNumberLabel.Caption:='Windows 95';
  VER_PLATFORM_WIN32_NT: VersionNumberLabel.Caption := 'Windows NT';
 else
  VersionNumberLabel.Caption := '';
 end;
end;
```

В первой строке заполняется свойство `dwOSVersionInfoSize` структуры `OSVersion`, которая имеет тип `TOSVersionInfo`. В этом свойстве обязательно нужно указывать размер самой структуры. После этого следует вызвать функцию `GetVersionEx` и указать ей в качестве параметра нашу структуру. Функция заполнит все остальные поля структуры корректной информацией о версии и платформе Windows. Идентификатор платформы находится в свойстве `dwPlatformID`.

6.1.4. Дополнительная информация о Windows

Далее из реестра считывается дополнительная информация о Windows, которая находится в реестре по адресу:

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`

6.1.5. Переменные окружения Windows

Переменные окружения — это самое основное, что может вам понадобиться в повседневном программировании. В переменные окружения входят каталоги Windows, которые использует ОС. Сам каталог, в который установлена Windows, узнать достаточно легко:

```
FillChar(PathArray, SizeOf(PathArray), #0);
GetWindowsDirectory(PathArray,255);
WindowsDirLabel.Caption:= Format('%s',[PathArray]);
```

В первой строке заполняется переменная `PathArray` нулевыми символами, чтобы случайный мусор не повлиял на работу программы. Эта переменная — простой массив символов. Во второй строке вызывается функция `GetWindowsDirectory`, которой нужно передать два параметра: массив символов и его размер. Размер массива символов должен быть достаточным для хранения полного пути к папке Windows. В большинстве случаев достаточно 20 символов, но на всякий случай я всегда выделяю 255.

Чтобы узнать путь к временной папке Windows, используется более прогрессивный метод:

```
FillChar(PathArray, SizeOf(PathArray), #0);
ExpandEnvironmentStrings('%TEMP%', PathArray, 255);
TempDir.Caption:=Format('%s',[PathArray]);
```

В первой строке массив символов также заполняется нулями. Во второй строке вызывается функция `ExpandEnvironmentStrings`, которой нужно передать три параметра:

- переменную окружения, значение которой нужно получить;
- массив символов, в который будет записан результат;
- размер массива символов.

Но самый прогрессивный метод, который я рекомендую использовать в своих программах, я пока не указал. Для его реализации нам понадобится две переменные:

```
P:PItemIDList;
C:array [0..1000] of char;
```

Теперь напишем код, который узнает местоположение папки, в которой хранятся ярлычки меню **Программы** главного меню:

```
if SHGetSpecialFolderLocation(Handle,CSIDL_PROGRAMS,p)=NOERROR then
 begin
  SHGetPathFromIDList(P,C);
  Переменная типа String:=StrPas(C);
 end
```

В первой строке вызывается функция `SHGetSpecialFolderLocation`, ей передаются три параметра:

- указатель на окно, которое должно будет отображаться (если это будет необходимо);

- идентификатор параметра, который мы хотим получить. Возможны следующие значения:
  - `CSIDL_BITBUCKET` — Корзина;
  - `CSIDL_CONTROLS` — Панель управления;
  - `CSIDL_DESKTOP` — рабочий стол;
  - `CSIDL_DESKTOPDIRECTORY` — физический путь к папке рабочего стола;
  - `CSIDL_DRIVES` — Мой компьютер;
  - `CSIDL_FONTS` — шрифты;
  - `CSIDL_NETHOOD` — сетевое окружение;
  - `CSIDL_NETWORK` — виртуальный каталог сетевого окружения;
  - `CSIDL_PERSONAL` — персональная папка для документов (Мои документы);
  - `CSIDL_PRINTERS` — принтеры;
  - `CSIDL_PROGRAMS` — программы (Program Files);
  - `CSIDL_RECENT` — недавно открытые документы;
  - `CSIDL_SENDTO` — папка с ярлыками **Отправить**;
  - `CSIDL_STARTMENU` — папка главного меню **Пуск**;
  - `CSIDL_STARTUP` — папка меню **Автозагрузка**;
  - `CSIDL_TEMPLATES` — папка шаблонов;
- переменная, в которую будет записан результат.

Результат мы получаем в виде переменной типа `PItemIDList`. Чтобы превратить эту переменную в строку, нужно сначала выполнить функцию `SHGetPathFromIDList`, которой передаются два параметра:

- переменная типа `PItemIDList`;
- массив символов, в который будет записан результат.

Чтобы массив символов превратить в привычную для Delphi строку `String`, можно использовать функцию `StrPas`, которой нужно передать массив символов, и на выходе получить привычную строку.

6.2. Информация о памяти

Прошли те времена, когда память считали в килобайтах, и программисты ценили каждый бит оперативной памяти компьютера на вес золота. Сейчас в настольных системах устанавливается не менее 64 Мбайт памяти, а то и

все 128, 256 или больше. Помимо этого, ОС Windows умеет использовать дисковое пространство для того, чтобы выгружать неиспользуемые в данный момент блоки оперативной памяти на диск, чтобы освободить программе необходимые мегабайты.

И все же еще остались такие приложения, где оперативная память играет достаточно большую роль, и ее нехватка может плохо отражаться на производительности или даже стабильности программы. Именно поэтому, прежде чем выделять большие куски памяти, нужно убедиться, что необходимые мегабайты существуют и свободны для использования.

В этом разделе мы познакомимся с тем, как можно узнать количество общей/свободной оперативной или дисковой памяти, которую может нам выделить Windows. Для этого в нашем глобальном примере существует вкладка **Память** (рис. 6.3). Здесь расположено несколько компонентов `Label` и два компонента `TGauge` со вкладки **Samples**. Компоненты `TGauge` будут отображать информацию о памяти в графическом виде. У обоих компонентов нужно установить свойство `Kind` равным значению `gkPie`, чтобы компонент выглядел в виде круговой диаграммы.

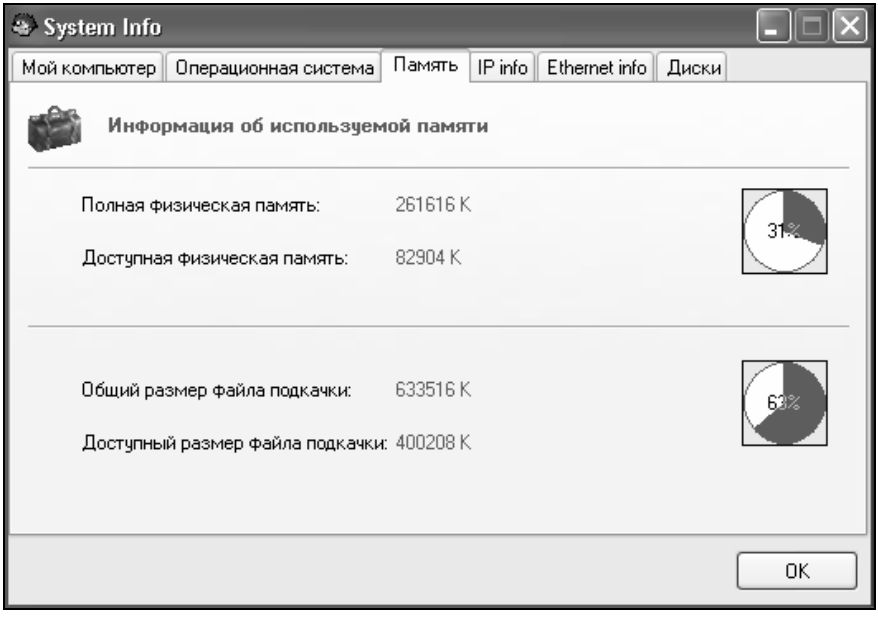


Рис. 6.3. Окно, отображающее информацию о состоянии памяти компьютера

Содержимое вкладки **Память** будет заполняться при событии `OnShow` главной формы в процедуре `GetMemoryInfo`, которая должна быть объявлена в разделе `private` следующим образом:

```
private
  { Private declarations }
  procedure GetMemoryInfo;
```

Опишите эту процедуру и нажмите комбинацию клавиш <Ctrl>+<Shift>+ +<C>, чтобы среда Delphi создала пустую заготовку. В ней нужно написать следующее:

```
procedure TSystemInfoForm.GetMemoryInfo;
var
 MemInfo : TMemoryStatus;
begin
 MemInfo.dwLength := Sizeof (MemInfo);
 GlobalMemoryStatus (MemInfo);
 TotalPhys.caption:=inttostr(MemInfo.dwTotalPhys div 1024) + ' K';
 AvailPhys.caption:=inttostr(MemInfo.dwAvailPhys div 1024) + ' K';
 TotalPage.caption:=inttostr(MemInfo.dwTotalPageFile div 1024) + ' K';
 AvailPage.caption:=inttostr(MemInfo.dwAvailPageFile div 1024) + ' K';
 RamGauge.Progress := MemInfo.dwAvailPhys div (MemInfo.dwTotalPhys div
100);
 VirtualGauge.Progress := MemInfo.dwAvailPageFile div
(MemInfo.dwTotalPageFile div 100);
 {Если значение слишком маленькое, то меняем цвет на красный}
 if (RamGauge.Progress < 5) then RamGauge.ForeColor := clRed
 else RamGauge.ForeColor := clActiveCaption;
 if (VirtualGauge.Progress < 20) then VirtualGauge.ForeColor := clRed
 else VirtualGauge.ForeColor := clActiveCaption;
end;
```

В первой строчке заполняется свойство `dwLength` структуры `MemInfo`, которая имеет тип `TMemoryStatus`. В этом свойстве нужно обязательно указать размер структуры `MemInfo`. Далее вызывается WinAPI-функция `GlobalMemoryStatus`, которой нужно передать нашу структуру. После выполнения функция заполнит все поля структуры оперативной информацией.

Теперь у нас в структуре `MemInfo` находится вся необходимая информация о состоянии памяти компьютера.

Вот свойства, которые вас могут заинтересовать:

- `dwTotalPhys` — общий объем физической оперативной памяти;
- `dwAvailPhys` — объем доступной оперативной памяти;
- `dwTotalPageFile` — общий размер файла подкачки;
- `dwAvailPageFile` — доступный размер файла подкачки.

Если из общего размера памяти вычесть доступный, то мы получаем размер занятой памяти.

В указанных свойствах данные приведены в байтах. Чтобы их перевести в килобайты, нужно разделить значение свойства на 1024. Делим с помощью операции `div`, которая возвращает целое значение результата, отбрасывая дробную часть.

Если размер оставшейся памяти слишком маленький, то нужно изменить цвет заливки компонентов `TGauge` на красный.

6.3. Информация о дисках

Информация о состоянии диска достаточно актуальна для любых приложений. Допустим, что ваша программа должна скопировать какой-то файл. Если на диске не будет достаточного объема свободного пространства, то произойдет лишняя ошибка, которая может ввести пользователя в заблуждение. Любая ошибка будет только лишним недостатком программы, поэтому от таких досадных ошибок, как нехватка ресурсов, лучше предостерегаться заранее.

Нехватка дискового пространства — одна из самых неприятных проблем, поэтому перед созданием/копированием больших файлов лучше проверять доступность необходимого пространства. Это особенно касается программ установки.

Помимо доступного пространства можно узнать серийный номер тома, на котором установлена программа. С помощью такого номера очень удобно и надежно делается защита от копирования программы без инсталляции. После установки программа может запомнить где-нибудь серийный номер тома и затем проверять его при старте. Если серийный номер изменился, то программа должна требовать переустановки, где вы можете реализовать более жесткие возможности защиты программы.

В моей программе на вкладке **Диски** вы можете узнать основные сведения о любом диске (рис. 6.4). Для этого на вкладке у меня расположен компонент `DriveComboBox` со вкладки **Win 3.1** для выбора диска и куча компонентов `TLabel`, в которых будет отображаться вся полученная информация.

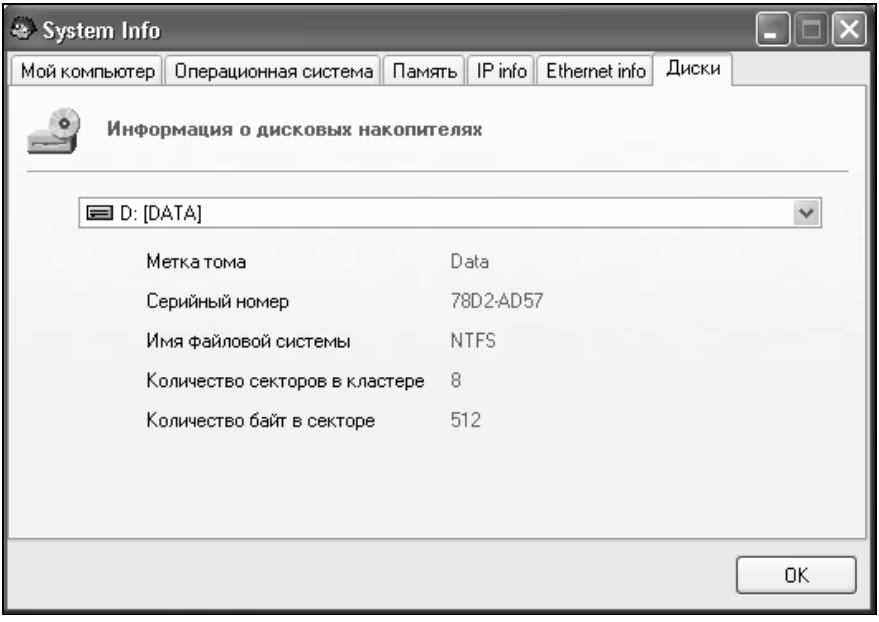


Рис. 6.4. Окно, отображающее информацию о выбранном диске

Получение информации происходит по событию `OnChange` компонента `DriveComboBox1`. Здесь вызывается процедура `UpdateDisk`. Ее нужно описать в разделе `private` следующим образом:

```
procedure UpdateDisk;
```

А сама процедура должна выглядеть так, как в листинге 6.2.

Листинг 6.2. Сканирование параметров диска

```
procedure TSystemInfoForm.UpdateDisk;
var
 lpRootPathName           : PChar;
 lpVolumeNameBuffer       : PChar;
 nVolumeNameSize          : DWORD;
 lpVolumeSerialNumber     : DWORD;
 lpMaximumComponentLength : DWORD;
 lpFileSystemFlags        : DWORD;
 lpFileSystemNameBuffer   : PChar;
 nFileSystemNameSize      : DWORD;
```

```
 FSectorsPerCluster: DWORD;
 FBytesPerSector   : DWORD;
 FFreeClusters     : DWORD;
 FTotalClusters    : DWORD;
begin
 lpVolumeNameBuffer      := '';
 lpVolumeSerialNumber    := 0;
 lpMaximumComponentLength:= 0;
 lpFileSystemFlags       := 0;
 lpFileSystemNameBuffer  := '';
 try
  GetMem(lpVolumeNameBuffer, MAX_PATH + 1);
  GetMem(lpFileSystemNameBuffer, MAX_PATH + 1);
  nVolumeNameSize := MAX_PATH + 1;
  nFileSystemNameSize := MAX_PATH + 1;
  lpRootPathName := PChar(DriveComboBox1.Drive+':\');
  if GetVolumeInformation( lpRootPathName, lpVolumeNameBuffer,
      nVolumeNameSize, @lpVolumeSerialNumber, lpMaximumComponentLength,
      lpFileSystemFlags, lpFileSystemNameBuffer, nFileSystemNameSize )
   then
     begin
      VolumeName.Caption    := lpVolumeNameBuffer;
      VolumeSerial.Caption  := IntToHex(HIWord(lpVolumeSerialNumber), 4)
             + '-' + IntToHex(LOWord(lpVolumeSerialNumber), 4);
      FileSystemName.Caption:= lpFileSystemNameBuffer;
      GetDiskFreeSpace( PChar(DriveComboBox1.Drive+':\'),
           FSectorsPerCluster, FBytesPerSector,
           FFreeClusters, FTotalClusters);
     end;
 finally
  FreeMem(lpVolumeNameBuffer);
  FreeMem(lpFileSystemNameBuffer);
  end;
 SectorsPerCluster.Caption:=IntToStr(FSectorsPerCluster);
 BytesPerSector.Caption:=IntToStr(FBytesPerSector);
end;
```

В самом начале все основные переменные lpVolumeNameBuffer, lpVolumeSerialNumber, lpMaximumComponentLength, lpFileSystemFlags, lpFileSystemNameBuffer обнуляются. После этого с помощью процедуры GetMem выделяется память для переменных lpVolumeNameBuffer и lpFileSystemNameBuffer. В переменные nVolumeNameSize и nFileSystemNameSize заносится размер выделенной памяти для предыдущих переменных.

Для получения информации о выбранном диске используется WinAPI-процедура GetVolumeInformation, у которой следующие параметры:

- имя диска, информацию о котором надо получить;
- буфер, в который будет помещено имя тома диска;
- размер буфера для имени тома;
- переменная, в которую будет записан серийный номер;
- переменная, в которую будет записано максимальное значение пути, поддерживаемое файловой системой диска;
- флаги файловой системы. Здесь может быть любая комбинация следующих флагов:
  - FS_CASE_IS_PRESERVED — указывает на то, что файловая система сохраняет регистр имен файлов, когда сохраняет имя на диске;
  - FS_CASE_SENSITIVE — файловая система чувствительна к регистру имен файлов;
  - FS_UNICODE_STORED_ON_DISK — файловая система поддерживает имена в Unicode;
  - FS_PERSISTENT_ACLS — файловая система поддерживает списки доступа (например, NTFS);
  - FS_FILE_COMPRESSION — файловая система поддерживает компрессию на уровне файлов;
  - FS_VOL_IS_COMPRESSED — файловая система поддерживает компрессию на уровне тома (например, DoubleSpace тома диска);
- буфер, в который будет помещено имя файловой системы;
- размер буфера для имени файловой системы.

После вывода на экран полученной информации вызывается еще одна функция — GetDiskFreeSpace, с помощью которой можно рассчитать свободное дисковое пространство. Но пока мы не рассчитываем этого, потому что данный вопрос мы будем обсуждать немного позже. С помощью этой функции мы узнаем количество секторов в кластере и количество байтов в секторе.

Давайте рассмотрим параметры функции `GetDiskFreeSpace`:

- имя диска, информацию которого надо получить;
- переменная, в которую будет записано количество секторов в кластере;
- переменная, в которую будет записано количество байтов в секторе;
- переменная, в которую будет записано количество свободных кластеров;
- переменная, в которую будет записано общее количество кластеров.

По размеру кластера и количеству свободных кластеров можно рассчитать общий размер свободного пространства. По размеру кластера и общему количеству кластеров можно узнать общий размер диска.

Вот теперь рассмотрим пример универсальной функции, которая рассчитывает размер диска:

```
function GetFreeDiskSize(Root: string): LongInt;
var
  SpC, BpS, NfC, TnC: DWORD;
  FreeDiskSize: Double;
begin
  GetDiskFreeSpace(PChar(Root), SpC, BpS, NfC, TnC);
  FreeDiskSize := (NfC * SpC * BpS) / 1024;
  Result := Round(FreeDiskSize);
end;
```

В первой строке узнаем размер и количество свободных кластеров. Во второй строке производим расчет с помощью перемножения количества свободных кластеров, количества секторов в кластере и количества байтов в секторе. Результат расчета делим на 1024, чтобы перевести результат из байтов в килобайты.

> ***Примечание***
>
> На компакт-диске в каталоге\Примеры\Глава 6\System Info\ вы можете найти пример программы, обсуждаемой в данном разделе.

6.4. Частота и загрузка процессора

В этом разделе речь пойдет о сердце компьютера — процессоре. Как быстро он работает? Сильно ли его загружает то или иное приложение? На эти вопросы можно ответить самим, написав программы, тестирующие работу процессора.

6.4.1. Частота процессора

В данном разделе я хочу показать, как определить частоту работы процессора. Несмотря на то, что в реальных условиях трудно найти пример программы, где это может пригодиться, но используемые в этом примере приемы программирования очень интересны и познавательны.

Давайте создадим в Delphi новый проект. На форме нам понадобятся два компонента `TLabel` и две кнопки: **Запустить** и **Стоп**. После нажатия этих кнопок будет запускаться и останавливаться процесс определения скорости процессора. Один компонент `TLabel` чисто информационный и содержит текст **Скорость процессора:**. Во втором мы будем выводить текст, содержащий значение частоты процессора. Мою форму программы вы можете увидеть на рис. 6.5.

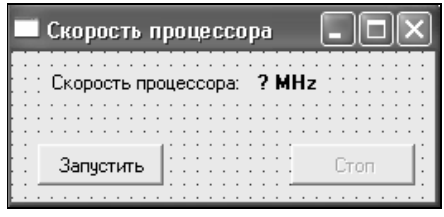


Рис. 6.5. Окно, готовое к отображению частоты процессора

Теперь в обработчике события нажатия кнопки **Запустить** напишите следующий код:

```
procedure TFormCPUSpeed.BitBtnStartClick(Sender: TObject);
begin
 BitBtnStart.Enabled := False;
 BitBtnStop.Enabled := True;
 Stop := False;
 while not Stop do
  begin
   LabelCPUSpeed.Caption := FloatToStr(GetCPUSpeed)+' MHz';
   Application.ProcessMessages;
  end;
 BitBtnStart.Enabled := True;
 BitBtnStop.Enabled := False;
end;
```

После запуска определения частоты процессора мы делаем кнопку **Запустить** неактивной, потому что второй раз нажимать ее нет смысла. Ошибки конечно не будет, но лишние активные кнопки не нужны. Кнопку **Стоп**, наоборот, делаем активной, чтобы пользователь имел возможность остановить процесс.

Далее, переменной `Stop` присваивается значение `false`. По значению этой переменной будет определяться, нужно ли остановить процесс определения частоты. Как только она станет равной `true`, мы прервем работу программы.

Вот теперь все готово для запуска цикла. Цикл определения очень прост:

```
while not Stop do
 begin
  LabelCPUSpeed.Caption := FloatToStr(GetCPUSpeed)+' MHz';
  Application.ProcessMessages;
 end;
```

Здесь работает цикл `while`, который будет выполняться, пока переменная `Stop` не станет равной значению `true`. Внутри цикла только две строки:

- в первой вызывается функция `GetCPUSpeed`. Результат ее выполнения превращаем в строку с помощью функции `FloatToStr` и присваиваем компоненту `TLabel`, который отображает частоту процессора;
- во второй вызывается метод `ProcessMessages`, который дает другим программам поработать, чтобы наша маленькая утилита не отобрала все процессорное время и не произошел эффект зависания.

Функция `GetCPUSpeed` приведена в листинге 6.3.

Листинг 6.3. Определение частоты процессора

```
function GetCPUSpeed: Double;
const
 DelayTime = 500;
var
 TimerHi, TimerLo: DWORD;
 PriorityClass, Priority: Integer;
begin
 PriorityClass := GetPriorityClass(GetCurrentProcess);
 Priority := GetThreadPriority(GetCurrentThread);
 SetPriorityClass(GetCurrentProcess, REALTIME_PRIORITY_CLASS);
 SetThreadPriority(GetCurrentThread, THREAD_PRIORITY_TIME_CRITICAL);
```

```
 Sleep(10);
 asm
  dw 310Fh
  mov TimerLo, eax
  mov TimerHi, edx
 end;
 Sleep(DelayTime);
 asm
  dw 310Fh
  sub eax, TimerLo
  sbb edx, TimerHi
  mov TimerLo, eax
  mov TimerHi, edx
 end;
 SetThreadPriority(GetCurrentThread, Priority);
 SetPriorityClass(GetCurrentProcess, PriorityClass);
 Result := TimerLo/(1000.0 * DelayTime);
end;
```

Как видите, эта функция не относится к объекту окна, а значит, должна быть описана выше того места, где мы ее используем, т. е. выше обработчика события `OnClick` кнопки **Запустить**. Перепишите ее себе, и сейчас мы рассмотрим все содержимое процедуры более подробно.

В самом начале мы узнаем приоритет класса и приоритет потока с помощью функций `GetPriorityClass` и `GetThreadPriority`. По умолчанию все программы получают нормальный приоритет и работают наравне с другими. Значения приоритетов сохраняются в отдельных переменных.

После этого значения приоритетов изменяются на максимальный с помощью функций `SetPriorityClass` и `SetThreadPriority`. Для класса устанавливается приоритет реального времени — `REALTIME_PRIORITY_CLASS`. Для потока указывается критический ко времени приоритет — `THREAD_PRIORITY_TIME_CRITICAL`. Это необходимо, чтобы получить абсолютно все ресурсы компьютера.

Изменив приоритет, делаем задержку в десять миллисекунд с помощью вызова процедуры `Sleep(10)`, чтобы Windows смогла среагировать на изменения и выделить все ресурсы.

Вот теперь начинает происходить само определение частоты. Для этого дважды вызывается ассемблерный код, а в промежутке происходит задержка на период, указанный в константе `DelayTime`. Сам ассемблерный код я расписывать не буду, потому что он выходит за рамки книги и потребует от вас дополнительных знаний. Я только скажу, что с помощью ассемблера замеряется работа таймера процессора за интервал времени, указанный в `DelayTime`. По умолчанию этот интервал равен 500 миллисекундам.

После замера работы таймера значения приоритета класса и потока восстанавливаются с помощью все тех же функций изменения приоритета и сохраненных исходных значений:

```
SetThreadPriority(GetCurrentThread, Priority);
SetPriorityClass(GetCurrentProcess, PriorityClass);
```

Если этого не сделать, то может произойти сбой, и Windows будет работать некорректно. Критичный приоритет и приоритет реального времени отдает программе все ресурсы, и могут произойти конфликты, потому что на таком приоритете работает только ядро Windows и некоторые особо критичные приложения. Если мы добавим свою программу, то она может конфликтовать с ядром и вызвать системный сбой. Именно поэтому желательно получать у Windows такие ресурсы только на короткие промежутки времени.

Результату выполнения функции я присваиваю результат расчета работы частоты процессора — `TimerLo / (1000.0 * DelayTime)`. Именно это значение мы переводим потом в строку и выводим на экран (рис. 6.6).

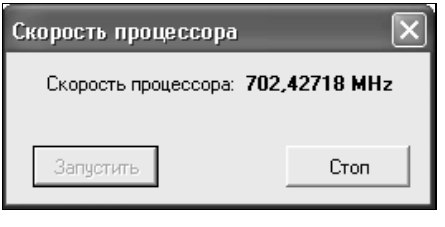


Рис. 6.6. Пример работы программы

В обработчике события нажатия кнопки **Стоп** пишем следующий код:

```
procedure TFormCPUSpeed.BitBtnStopClick(Sender: TObject);
begin
 Stop := True;
end;
```

Здесь мы только делаем переменную `Stop` равной `true`, что заставит цикл определения частоты процессора прерваться.

Если вы запустите программу и запустите определение частоты, то заметите, что она немного плавает. Это нормальная работа программы, и тут не надо волноваться. Все в порядке. Есть процессоры, которые могут сами регулировать свою частоту в зависимости от температуры. Если у вас такой экземпляр, то вы сможете программно управлять приоритетами, не используя системных средств.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 6\CPU Speed\ вы можете найти пример этой программы.

6.4.2. Загрузка процессора

Для определения загрузки процессора я воспользуюсь модулем `adCpuUsage`, который написал Alexey A. Dynnikov. Этот человек явно русского происхождения, но писать его фамилию на русском я побоялся, чтобы случайно не ошибиться в правильности написания. Этот модуль вы можете найти на компакт-диске в каталоге Headers или в том же каталоге, что и пример программы.

Для реализации примера на форме нам понадобятся один компонент `Chart` со вкладки **Additional** и таймер со вкладки **System**. У таймера нужно установить свойство `Enabled` равным `true`, чтобы после старта программы он сразу же был во включенном состоянии. Форму будущей программы вы можете увидеть на рис. 6.7.

В обработчике события `OnTimer` компонента `Ttimer` пишем следующий код:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
 i:Integer;
begin
 CollectCPUData;
 for i:=0 to GetCPUCount-1 do
  begin
   if Chart1.Series[i].Count>20 then
    Chart1.Series[i].Delete(0);
```

```
    Chart1.Series[i].AddXY(Time, GetCPUUsage(i)*100,
         Format('%5.2f%%',[GetCPUUsage(i)*100]));
  end;
end;
```

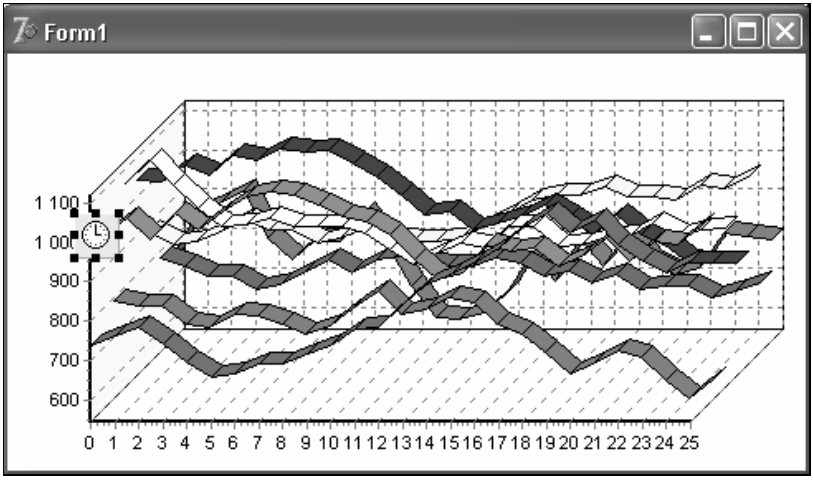


Рис. 6.7. Форма будущей программы определения загрузки процессора

Для компиляции примера нам понадобится добавить в раздел `uses` модуль `adCpuUsage`. Не забудьте поместить файл модуля в каталог, который будет доступен для компилятора Delphi, например, в тот же каталог, что и программа.

Попробуйте скомпилировать пример и запустить на выполнение. Убедитесь, что программа работает верно и без ошибок.

Теперь рассмотрим код, который мы написали. В самом начале вызывается функция `CollectCPUData`, которая получает данные о загруженности процессора. После этого запускается цикл от 0 до количества процессоров, установленных на компьютере. В большинстве случаев на компьютере установлен только один процессор, но надо учитывать и серверы, где их может быть не только 2, но и более.

Внутри цикла проверяется, если у компонента `Chart1`, который строит график по внесенным в него значениям, накопилось более 20 параметров, то нужно удалить самый старый из них, т. е. нулевой. После этого в компонент `Chart1` добавляется текущее значение загруженности, которое можно получить с помощью функции `GetCPUUsage`. Это значение дается нам в долях (от

нуля до единицы), поэтому его нужно умножить на 100, чтобы получить процентное отношение.

Результат работы программы представлен на рис. 6.8.

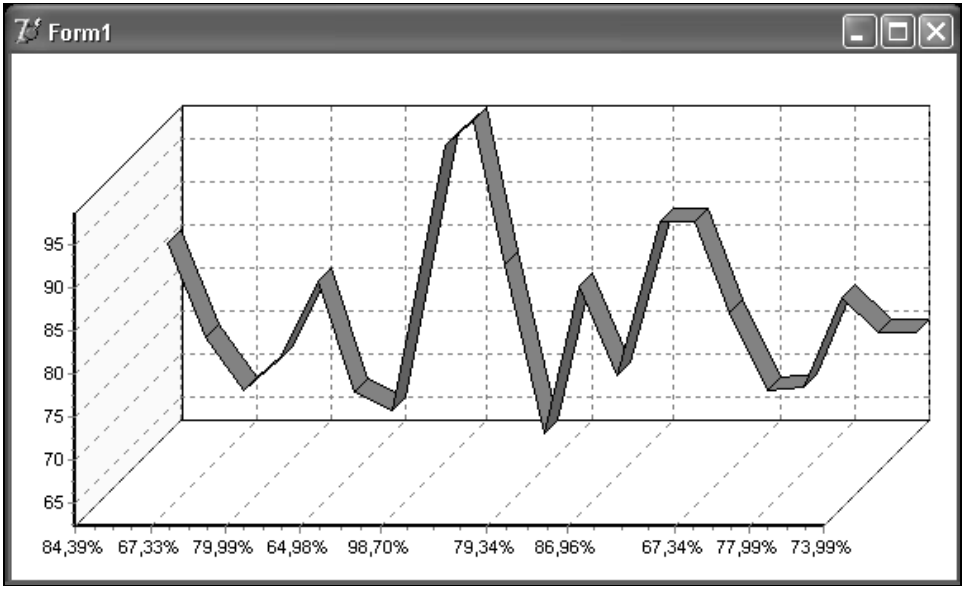


Рис. 6.8. Пример работы программы

Примечание

На компакт-диске в каталоге \Примеры\Глава 6\CPU Usage\ вы можете найти пример программы и цветные рисунки этого раздела.

6.5. Работа с COM-портом

По своему опыту могу сказать, что работа с COM-портом — одна из наиболее часто решаемых задач на предприятиях, если не считать финансовых программ. На производствах везде используют современное оборудование (контроллеры, устройства сбора информации), с которыми чаще всего можно работать через стандартный порт компьютера — COM.

Порт COM в промышленности часто называют RS-232. Вы должны знать это название, чтобы случайно не растеряться, когда увидите его. В докумен-

тации на промышленные приборы используется именно это название, хотя мы привыкли к названию COM-порт.

Итак, для работы с портами компьютера я привык использовать компонент `Comm32`. Этот компонент я уже давно нашел на просторах Интернета и немного переработал, усилив его надежность и улучшив возможности. Исходный код компонента вы можете найти на компакт-диске в каталоге Headers. Он инсталлируется в Delphi, поэтому перед использованием его нужно установить, выбрав в меню **Component** пункт **Install Component**. Остальное вам уже должно быть известно.

У компонента `Comm32` есть следующие свойства:

- `BaudRate` — скорость передачи данных (бит/с), например 9600;
- `Bits` — биты данных, например 8;
- `StopBits` — стоповые биты, например 1;
- `ComPort` — порт, с которым вы хотите работать. Имя порта нужно указывать в тестовом виде, например, COM1 или COM2;
- `ComLogFileName` — имя файла, в который будет сохраняться вся информация от работы с портом.

Помимо этого, во время работы компонент использует следующие дополнительные настройки, которые нельзя изменить с помощью свойств:

- четность — нет (с помощью четности проверяется целостность передаваемых данных);
- управление потоком — нет (может быть Xon, Xoff или аппаратное и также помогает обеспечить целостность данных).

Теперь напишем пример использования примера. Создайте новый проект в Delphi и разместите на форме:

- три компонента `TButton` с заголовками: **Открыть порт**, **Послать**, **Закрыть порт**. Кнопки **Послать** и **Закрыть порт** должны быть неактивными (свойство `Enabled` установим равным `false`);
- один компонент `TMemo`, в котором мы будем выводить полученные через порт данные;
- только что установленный компонент `Comm32` со вкладки **CyD**.

Мою форму вы можете увидеть на рис. 6.9.

У компонента `Comm32` установим значения следующих свойств:

- `BaudRate` — `9600`;
- `Bits` — `8`;
- `StopBits` — `1`.

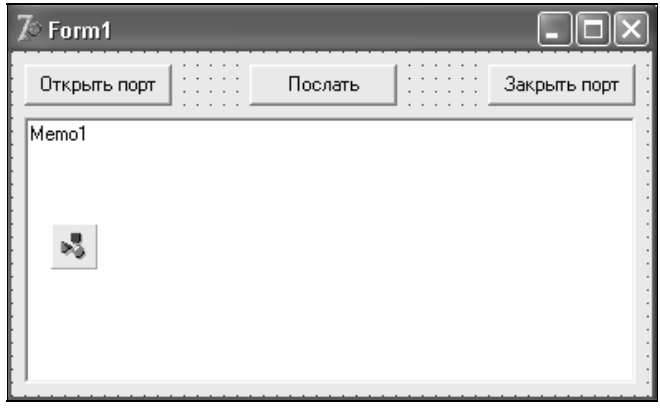


Рис. 6.9. Форма будущей программы

В свойстве `CommPort` укажите порт, с которым вы хотите работать.

Теперь займемся программированием. При нажатии кнопки **Открыть порт** мы должны открыть порт. Для этого пишем следующий код:

```
procedure TForm1.OpenButtonClick(Sender: TObject);
begin
 Comm321.StartComm;
 SendButton.Enabled:=true;
 CloseButton.Enabled:=true;
 OpenButton.Enabled:=false;
end;
```

Здесь вызывается метод `StartComm`, который открывает порт для приема/передачи данных. После этого делаем активными кнопки для отправки данных и закрытия порта. Кнопку открытия порта наоборот деактивируем, чтобы пользователь не нажал второй раз кнопку **Открыть порт**, потому что это вызовет ошибку.

Не советую сразу же после открытия отправлять данные, потому что реально данные могут не уйти. После открытия порта желательно делать задержку хотя бы в 100 миллисекунд. Обычно я добавляю в проект следующую функцию:

```
Procedure WaitT(time: integer);
var
 h:THandle;
```

```
begin
 h:=CreateEvent(nil, true, false, '');
 WaitForSingleObject(h,Time);
 CloseHandle(h);
end;
```

Теперь достаточно только вызвать в нужном месте эту процедуру и передать ей количество миллисекунд, которые надо подождать. Принцип ожидания вам уже должен быть известен, потому что мы уже не раз использовали его на протяжении всей книги.

Закрытие порта такое же простое, как и открытие. В обработчике нажатия кнопки **Закрыть порт** вы должны написать следующий код:

```
procedure TForm1.CloseButtonClick(Sender: TObject);
begin
 Comm321.StopComm;
 SendButton.Enabled:=false;
 CloseButton.Enabled:=false;
 OpenButton.Enabled:=true;
end;
```

Первым делом вызываем метод `StopComm` компонента `Comm32`, который закрывает работу порта. После этого деактивируем кнопки отправки данных и закрытия порта, потому что порт закрыт и отправлять данные уже нельзя. Кнопку открытия порта, наоборот, делаем активной, чтобы можно было снова открыть порт.

В обработчике нажатия кнопки **Послать** нужно написать следующий код:

```
procedure TForm1.SendButtonClick(Sender: TObject);
var
 SendStr:String;
begin
 SendStr:='';
 if InputQuery('Запрос данных',
               'Введите данные, которые надо отправить',
               SendStr) then
  Comm321.WriteCommData(PChar(SendStr+#13#10), Length(SendStr)+2);
end;
```

Перед отправкой выводим с помощью функции `InputQuery` на экран окно, в котором пользователь должен ввести данные для отправки. Это окно вы можете увидеть на рис. 6.10.

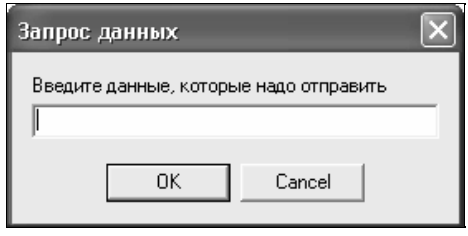


Рис. 6.10. Окно ввода отправляемых данных

Если пользователь ввел какие-нибудь данные и нажал кнопку **OK**, то отправляем их с помощью метода `WriteCommData` компонента `Comm32`. У этого метода два параметра:

- отправляемые данные в формате `PChar`. К отправляемым данным добавляем символы конца строки и перевода каретки: `#13#10`. Эти символы очень часто используются в оборудовании;
- во втором параметре указываем длину отправляемых данных. Узнаем длину введенной строки плюс два символа на перевод каретки и конец строки.

Для получения данных нужно создать обработчик события `OnReceiveData` компонента `Comm32`. В этом обработчике я написал следующий код:

```
procedure TForm1.Comm321ReceiveData(Sender: TObject; Buffer: Pointer;
  BufferLength: Word);
var
 RecivedStr:String;
begin
 RecivedStr:=PChar(Buffer);
 Memo1.Lines.Add(RecivedStr);
end;
```

Этот обработчик события будет вызываться каждый раз, когда на порт приходят какие-либо данные. В качестве первого параметра нам передается стандартный параметр для любого обработчика — объект, сгенерировавший событие. Во втором параметре передается буфер, содержащий принятые данные. Третий параметр — размер принятых данных.

Теперь разберемся с кодом обработчика. В первой строке кода преобразуем принятый буфер данных в привычную строку `String`. Во второй строчке добавляем данные в компонент `Memo1`.

Пример готов. Вы можете протестировать его на любом имеющемся оборудовании, которое может работать с компьютером через COM-порт, например модем. Таким образом, вы практически написали простейшую терминальную программу, с помощью которой можно работать с модемом или даже программировать его.

На компакт-диске в каталоге Документация вы найдете документ ПрограммированиеМодема.pdf, в котором описаны основные команды большинства модемов. С помощью этих команд можно программировать большинство модемов, потому что все современные модемы умеют обрабатывать стандартные AT-команды.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 6\COM Port\ вы можете найти пример данной программы.

6.6. Работа с LPT-портом

Точнее сказать, этот раздел будет посвящен управлению принтером, который естественно подключается к LPT-порту.

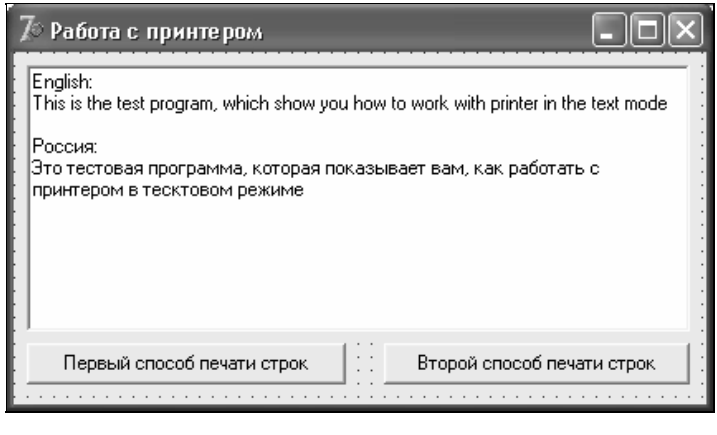


Рис. 6.11. Внешний вид будущей программы

Если вы работаете через этот порт только со сканером, то вы точно можете переворачивать страницу и не одну. С принтерами, установленными через USB, обсуждаемый пример сможет работать только наполовину. В общем, давайте рассмотрим пример, который я подготовил, и все увидим своими глазами.

В большинстве книг описывается, как работать с принтером в графическом режиме. А что если надо выводить информацию построчно? Этот вопрос почему-то опускается. Я решил исправить ситуацию и обсудить эту тему.

Запустите Delphi и в новом проекте поместите на форму две кнопки и один компонент `TMemo`. Внешний вид формы вы можете увидеть на рис. 6.11. Для правильной компиляции примера можете сразу же добавить в раздел `uses` модуль `WinSpool` и не дожидаться, когда вам сообщит об этом Delphi.

В обработчике нажатия кнопки **Первый способ печати строк** напишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 PrnHandle: THandle;
 N: DWORD;
 DocInfo1: TDocInfo1;
 i:Integer;
begin
 if not OpenPrinter('HP DeskJet 690C', PrnHandle, nil) then
   begin
    ShowMessage('Ошибка ' + IntToStr(GetLastError));
    exit;
   end;
 DocInfo1.pDocName := PChar('test doc');
 DocInfo1.pOutputFile:=nil;
 DocInfo1.pDataType := 'RAW';
 StartDocPrinter(PrnHandle, 1, @DocInfo1);
 StartPagePrinter(PrnHandle);
 for i:=0 to Memo1.Lines.Count-1 do
  WritePrinter(PrnHandle, PChar(Memo1.Lines.Strings[i]),
      Length(Memo1.Lines.Strings[i]), N);
 EndPagePrinter(PrnHandle);
 EndDocPrinter(PrnHandle);
 ClosePrinter(PrnHandle);
end;
```

Здесь в первой строке кода мы открываем принтер с помощью функции `OpenPrinter`. У этой функции указаны три параметра:

- имя принтера, установленного в системе;
- через этот параметр мы получим указатель на открытый принтер;
- указатель на структуру настроек по умолчанию `PRINTER_DEFAULTS`. Здесь вписываем нулевой указатель — `nil`.

Дальше заполняем переменную `DocInfo1`, которая объявлена в виде структуры типа `TDocInfo1`. Она понадобится нам уже на следующем этапе при открытии нового документа на принтере. Самое главное — это свойство `pDataType`. Здесь мы пишем параметр `RAW`.

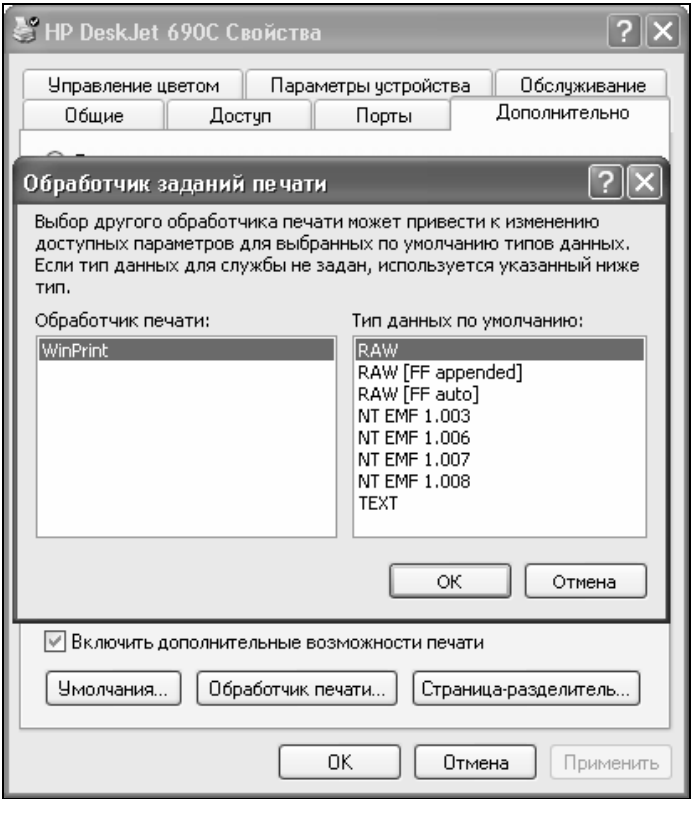


Рис. 6.12. Окно настройки типов данных по умолчанию

Таким образом, мы выбираем тип данных, с которыми будет работать наш принтер. Все типы данных вы можете увидеть в окне свойств принтера

(в Панели управления) на вкладке **Дополнительно**, если нажать кнопку **Обработчик очереди** (у вас, конечно, это может быть устроено по-другому). На рис. 6.12 вы можете увидеть окно настроек моего принтера.

Теперь мы готовы запустить новый документ на открытом принтере. Для этого вызывается API-функция `StartDocPrinter`. У нее есть три параметра:

- указатель на открытый принтер;
- версия структуры `DocInfo`, если вы хотите, чтобы ваша программа нормально работала в Windows NT, то вы можете использовать только первую версию структуры, для Windows 95/98 можно использовать и вторую версию;
- структура `DocInfo1`, которую мы недавно заполнили необходимыми параметрами.

Открыв документ, мы должны запустить на нем новую страницу. Для этого вызывается функция `StartPagePrinter`. У нее только один параметр — указатель на открытый принтер.

Теперь можно построчно выводить информацию на принтер с помощью функции `WritePrinter`, которой нужно передать четыре параметра:

- указатель на открытый принтер;
- строка, содержащая текст, который надо вывести на печать;
- длина строки;
- в этой переменной будет количество байтов, записанных в принтер.

После печати необходимо закрыть страницу с помощью функции `EndPagePrinter`, затем закрыть документ посредством `EndDocPrinter`, далее закрыть сам принтер с помощью функции `ClosePrinter`.

Этот метод построчной печати очень хорош и удобен тем, что будет работать даже там, где принтер подключен по USB или другим способом. Следующий метод, который я буду описывать, стабильно работает только через LPT-порт, а с принтерами, работающими по USB, иногда возникают проблемы. Возможно, это связано не с методом доступа, а с невозможностью принтеров работать в таком режиме, тут уже я сказать уверенно не могу, т. к. сам с такими проблемами нос к носу не сталкивался.

Для открытия принтера в текстовом режиме вторым способом используется процедура `AssignPrn`. В качестве единственного параметра этой процедуре надо передать переменную типа `TextFile`. После этого переменной оказывается назначен принтер по умолчанию. Дальше его нужно открыть с помощью процедуры `Rewrite`.

Как только файл открыт, в него можно печатать с помощью процедуры `writeln`, у которой два параметра:

- переменная типа `TextFile`, которой назначен принтер;
- текст, который надо распечатать.

После печати переменную надо освободить (закрыть файл, ассоциированный с принтером) с помощью процедуры `CloseFile`.

Итак, в обработчике события `OnClick` второй кнопки пишем следующий код:

```
var
 f:TextFile;
 i:Integer;
begin
 AssignPrn(f);
 try
  Rewrite(f);
  for i:=0 to Memo1.Lines.Count-1 do
   Writeln(f, Memo1.Lines.Strings[i]);
 finally
  CloseFile(f);
 end;
end;
```

В первой строке кода переменной `f` назначается принтер. После этого идет открытие файла, ассоциированного с принтером, и вывод на печать, заключенные между `try` и `finally`. Это необходимо, потому что если после выполнения процедуры `AssignPrn` переменной `f` не будет назначен принтер (ну, нет его в системе, не установлен или вообще отсутствует!), то при попытке открыть файл или начать печать произойдет ошибка.

Между `finally` и `end` (код, написанный здесь, будет выполняться всегда, вне зависимости от того, была ошибка или нет) происходит закрытие файла. Если не использовать `try...finally..end`, а во время печати произошла бы ошибка, то файл, ассоциированный с принтером, остался бы открытым. А это значит, что последующая нормальная работа принтера уже не гарантируется.

В предыдущем примере мы открывали принтер с помощью функции `AssignPrn`, а потом обращались к принтеру как к файлу. Для более полной иллюстрации того, что вы работаете с LPT-портом, как с настоящим текстовым файлом, попробуйте изменить строку открытия на эту:

```
AssignFile(f, 'LPT1');
```

Здесь использована функция `AssignFile` для открытия файла. Первая переменная указывает на переменную текстового файла (`TextFile`). Второй параметр должен содержать имя открываемого файла, а мы указываем `LPT1`. Если вы запустите этот пример, то сможете убедиться в том, что наш код действительно работает напрямую с портом и, соответственно, с принтером. С другими устройствами, подключенными к LPT, поддерживающими тестовой режим, работа будет происходить точно так же, т. е. как с простым текстовым файлом.

Обратите внимание, если вместо русского языка вы увидите абракадабру, то текст придется перекодировать в кодировку DOS. О переводе в различные кодировки можно прочитать в файле /Документация/Coding/Converter.pdf на компакт-диске.

Примечание

На компакт-диске в каталоге \Примеры\Глава 6\Printer\ вы можете найти пример данной программы.

6.7. Получение информации об устройстве вывода

Практически всю необходимую информацию об устройстве вывода, таком как монитор, принтер, плоттер, можно получить с помощью только одной функции — `GetDeviceCaps`.

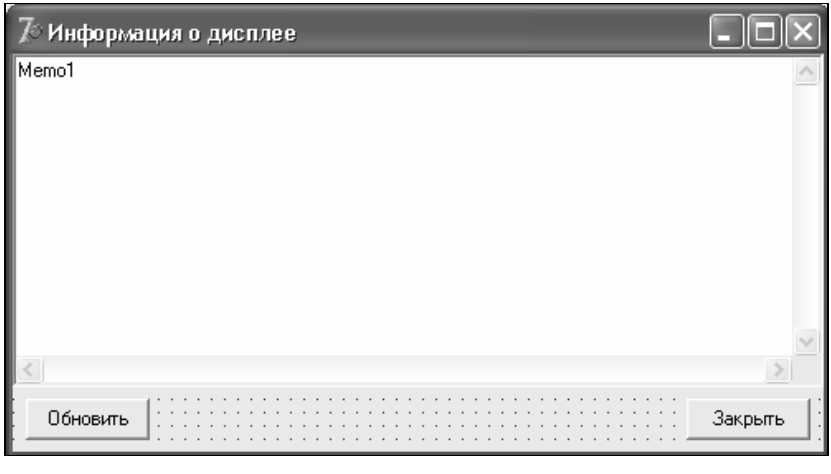


Рис. 6.13. Форма будущей программы

Это WinAPI-функция, которая универсальна для всех устройств вывода. В этом разделе книги мы рассмотрим небольшой пример, в котором получим всю необходимую информацию о мониторе.

Нам на форме обязательно понадобится одна кнопка, при нажатии которой нужно будет получать параметры дисплея. Помимо этого будет нужен компонент `TMemo`, в который будет выводиться вся информация в текстовом виде. Мою форму будущей программы вы можете увидеть на рис. 6.13.

Теперь создадим обработчик события `OnClick` кнопки **Обновить** и вставим в него содержание листинга 6.4. Полный листинг оказался слишком большим, чтобы сэкономить место, я привел только самые интересные отрывки. В примере на компакт-диске вы найдете более полный пример.

Листинг 6.4. Получение информации о мониторе

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  case GetDeviceCaps(Canvas.Handle, TECHNOLOGY) of
    DT_PLOTTER:    Memo1.Lines.Add('Тип: Векторный плоттер');
    DT_RASDISPLAY: Memo1.Lines.Add('Тип: Растровый дисплей');
    DT_RASPRINTER: Memo1.Lines.Add('Тип: Растровый принтер');
    DT_RASCAMERA:  Memo1.Lines.Add('Тип: Растровая камера');
    DT_CHARSTREAM: Memo1.Lines.Add('Тип: Поток символов');
    DT_METAFILE:   Memo1.Lines.Add('Тип: Метафайл');
    DT_DISPFILE:   Memo1.Lines.Add('Тип: Файл дисплея');
  end;
  Memo1.Lines.Add('Ширина в миллиметрах '+
     IntToStr(GetDeviceCaps(Canvas.Handle, HORZSIZE)));
  Memo1.Lines.Add('Высота в миллиметрах '+
     IntToStr(GetDeviceCaps(Canvas.Handle, VERTSIZE)));
  Memo1.Lines.Add('Ширина в пикселах '+
      IntToStr(GetDeviceCaps(Canvas.Handle, HORZRES)));

  if (GetDeviceCaps(Canvas.Handle, RASTERCAPS) and
      RC_BANDING)=RC_BANDING then
        Memo1.Lines.Add('Требуется сегментация');
  if (GetDeviceCaps(Canvas.Handle, RASTERCAPS) and
      RC_BITBLT)=RC_BITBLT then
        Memo1.Lines.Add('Может передавать Bitmaps');
```

```
  if (GetDeviceCaps(Canvas.Handle, RASTERCAPS) and
      RC_BITMAP64)=RC_BITMAP64 then
        Memo1.Lines.Add('Поддержка Bitmaps > 64K');

  if GetDeviceCaps(Canvas.Handle, CURVECAPS)=CC_NONE then
    Memo1.Lines.Add('Устройство не поддерживает кривые')
  else
   begin
    if (GetDeviceCaps(Canvas.Handle, CURVECAPS) and CC_PIE)=CC_PIE then
          Memo1.Lines.Add('Поддержка Pie Wedges');
    if (GetDeviceCaps(Canvas.Handle, CURVECAPS) and CC_WIDE)=CC_WIDE then
          Memo1.Lines.Add('Поддержка Wide Borders');
    if (GetDeviceCaps(Canvas.Handle, CURVECAPS) and
        CC_WIDESTYLED)=CC_WIDESTYLED then
          Memo1.Lines.Add('Поддержка Wide And Styled Borders');
    if (GetDeviceCaps(Canvas.Handle, CURVECAPS) and
          CC_INTERIORS)=CC_INTERIORS then
          Memo1.Lines.Add('Поддержка Interiors');
  end;

  if GetDeviceCaps(Canvas.Handle, LINECAPS)=LC_NONE then
    Memo1.Lines.Add('Device Does Not Support Lines')
  else
   begin
    if (GetDeviceCaps(Canvas.Handle, LINECAPS) and
        LC_POLYMARKER)=LC_POLYMARKER then
          Memo1.Lines.Add('Поддержка Multiple Markers');
    if (GetDeviceCaps(Canvas.Handle, LINECAPS) and LC_WIDE)=LC_WIDE then
          Memo1.Lines.Add('Поддержка Wide Lines');
    if (GetDeviceCaps(Canvas.Handle, LINECAPS) and
        LC_WIDESTYLED)=LC_WIDESTYLED then
          Memo1.Lines.Add('Поддержка Wide And Styled Lines');
    if (GetDeviceCaps(Canvas.Handle, LINECAPS) and
        LC_INTERIORS)=LC_INTERIORS then
          Memo1.Lines.Add('Поддержка Interiors');
  end;
end;
```

Несмотря на то, что кода много, он очень прост. Все здесь крутится вокруг вызова API-функции `GetDeviceCaps`. У этой функции два параметра:

- указатель на устройство, информацию о котором нужно получить;
- флаг, указывающий на ту информацию, которая нас интересует.

Я мог бы описать все флаги, но это не будет более понятно, чем сам исходный код. Именно поэтому я привел исходник полностью. Вам остается только посмотреть, какие параметры запрашиваются и какой текст выводится после этого в компонент `Memo1`. Например, самым первым идет свойство `TECHNOLOGY`, задавая в качестве первого параметра указатель на окно. Результатом будет технология устройства (для монитора — `Растровый дисплей`). Если указать в качестве первого параметра указатель на принтер (`Printer.Handle`), то в качестве результата можем получить `Растровый принтер`. По крайней мере, я получил именно это значение для своего Hewlett Packard 690C.

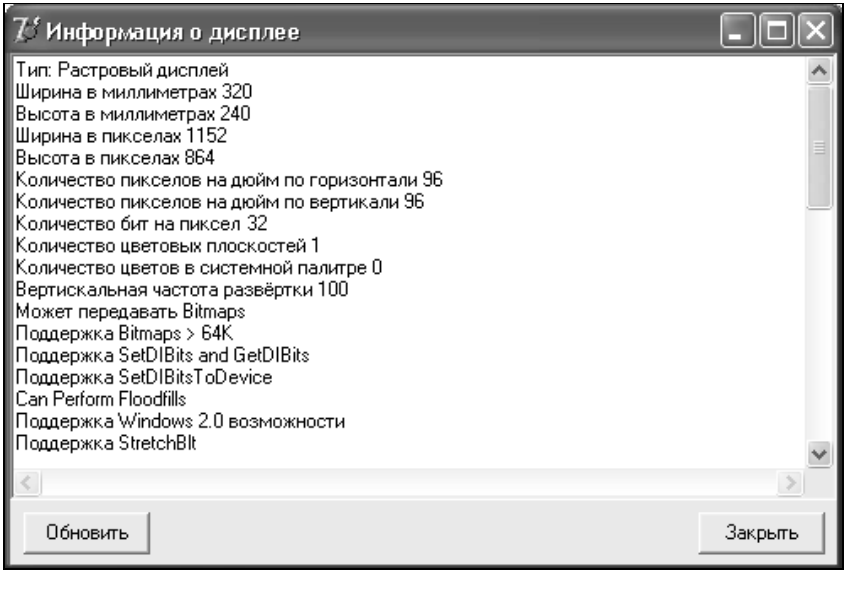


Рис. 6.14. Результат работы программы

На рис. 6.14 вы можете увидеть результат работы моей программы. Глядя на это окно, вы сможете увидеть, как работает пример. В этом примере нет выбора принтера, поэтому будет использоваться тот, который выбран принтером по умолчанию. Чтобы изменить принтер по умолчанию, нужно зайти в **Пуск | Настройки | Панель управления | Принтеры**, щелкнуть правой кноп-

кой мыши по ярлыку нужного принтера и в появившемся меню выбрать пункт меню **Использовать по умолчанию**.

> ***ПРИМЕЧАНИЕ***
>
> На компакт-диске в каталоге \Примеры\Глава 6\Display\ вы можете найти пример этой программы.

6.8. Работа с типами файлов

Если вам надо узнать, какой значок связан с определенным расширением, какая программа запускается для обработки определенного типа файлов, или вы хотите назначить свою программу для обработки определенного типа файлов, то этот раздел — для вас. Но все это мы будем узнавать постепенно.

6.8.1. Получение информации о типе файлов

Для начала научимся определять информацию и значок, связанные с определенным расширением. Для примера нам понадобится:

- поле ввода `Edit` для ввода типа файлов (например, doc);
- кнопка, нажимая которую мы будем получать необходимую информацию;
- четыре компонента `TLabel` для вывода необходимой информации:
  - **Параметры**, в свойстве `Name` укажите `ParamLabel`;
  - **Описание типа**, в свойстве `Name` укажите `DescriptionLabel`;
  - **Описание файла**, в свойстве `Name` укажите `FileDescriptionLabel`;
  - какой программой открывается указанный тип, в свойстве `Name` укажите `OpenWithLabel`;
- один компонент типа `TImage`. В этой картинке мы будем выводить изображение значка, связанного с указанным типом.

На рис. 6.15 вы можете увидеть мою форму, созданную для данного примера. Как всегда, вы можете видоизменить ее внешний вид, но необходимые для примера компоненты на форме должны быть.

Вся необходимая информация находится в реестре, и именно оттуда мы и будем ее читать, поэтому в раздел `uses` нужно сразу же добавить соответствующий модуль — `Registry`.

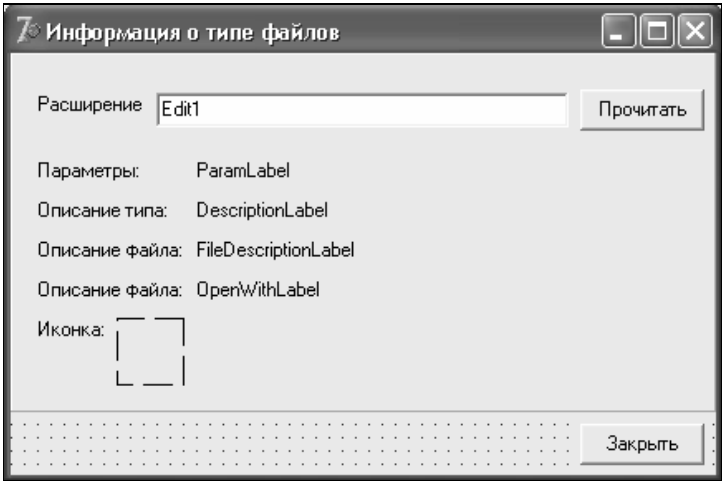


Рис. 6.15. Форма будущей программы

После нажатия кнопки **Прочитать** мы должны выяснить информацию об указанном типе файла. Для этого в обработчике события `OnClick` кнопки напишите следующее (листинг 6.5).

Листинг 6.5. Получение информации о типе файла

```
var
 Reg: TRegistry;
 IconFileName, IconIndex: String;
 PC: Array[0..255] of Char;
 i: Integer;
 ExtIcon:TIcon;
begin
 ExtIcon:=TIcon.Create;
 Reg := TRegistry.Create;
  try
   with Reg do
    begin
     RootKey := HKEY_CLASSES_ROOT;
     OpenKey(Edit1.Text, True);
     ExtDescription := ReadString('');
```

```
    OpenKey('\' + ExtDescription, True);
    FileDescription := ReadString('');
    OpenKey('DefaultIcon', True);
    IconFileName := ReadString('');
    SplitStr(',', IconFileName, IconIndex);
    StrPCopy(PC, IconFileName);
    ExtIcon.Handle := ExtractIcon(0, PC, StrToInt(IconIndex));
    Image1.Picture.Assign(ExtIcon);
    OpenKey('\' + ExtDescription + '\Shell\Open\Command', True);
    OpenWith := ReadString('');
    i := Pos('"', OpenWith);
    if i = 1 then
     begin
      OpenWith := Copy(OpenWith, 2, Length(OpenWith) - 1);
      i := Pos('"', OpenWith);
      ParamString := Copy(OpenWith, i + 3, Length(OpenWith) - i - 3);
      OpenWith := Copy(OpenWith, 0, i - 1);
     end
    else
     SplitStr(' ', OpenWith, ParamString)
   end;
 finally
  Reg.Free;
  ExtIcon.Free;
  ParamLabel.Caption:=ParamString;
  DescriptionLabel.Caption:=ExtDescription;
  FileDescriptionLabel.Caption:=FileDescription;
  OpenWithLabel.Caption:=OpenWith;
 end;
```

После открытия реестра мы сразу же переходим в раздел `HKEY_CLASSES_ROOT`. Именно здесь находится вся необходимая информация в разделах с именами в виде типа файла, например doc. В этом разделе нужно прочитать значение по умолчанию. Это значение используется в качестве имени раздела, в котором хранится дополнительная информация о типе файла. Например, для типа файлов doc у меня установлено значение

по умолчанию `Word.Document.8`. Чтобы прочитать остальную информацию, нам надо перейти в раздел `HKEY_CLASSES_ROOT\Word.Document.8` (цифра 8 является номером версии) — рис. 6.16.

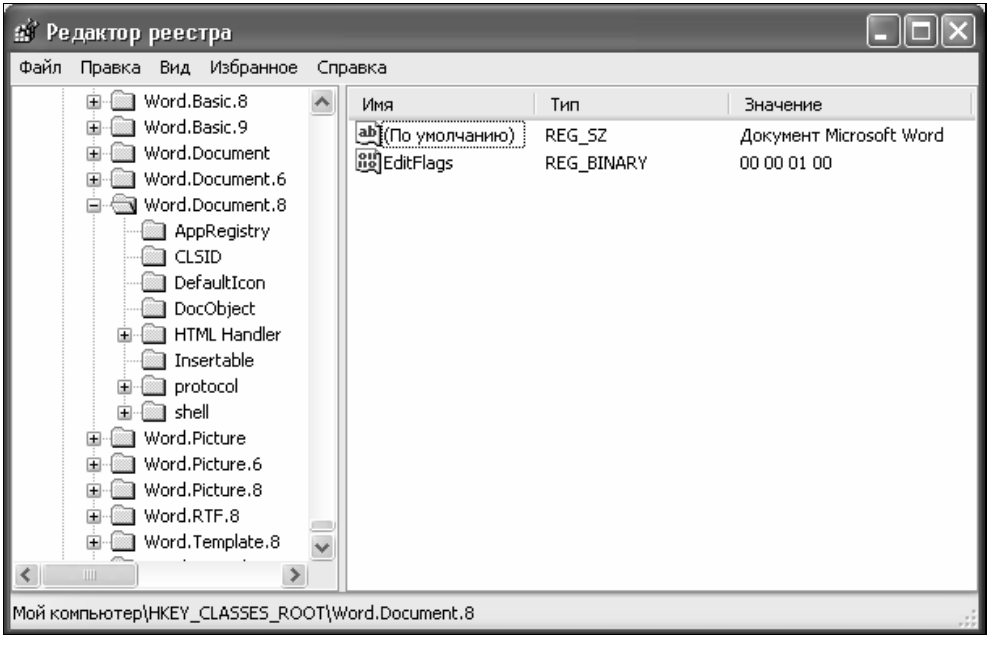


Рис. 6.16. Раздел `Word.Document.8` реестра

В этом разделе мы можем прочитать в качестве значения по умолчанию более полное описание типа файла. В подразделе `DefaultIcon` вы можете узнать значок, заданный по умолчанию для данного типа приложения, который используется для индикации в Проводнике. Изображение значка можно получить с помощью функции `ExtractIcon`. Этой функции нужно передать три параметра:

- указатель на экземпляр;
- полный путь к имени файла, значок которого нужно получить;
- индекс значка.

Функция загружает значок из файла и возвращает ее нам. Мы сохраняем ее в переменной `ExtIcon`, а потом присваиваем компоненту `Image1`.

В подразделе `\shell\Open\command` вы можете узнать путь к программе, которая используется для открытия файла данного типа.

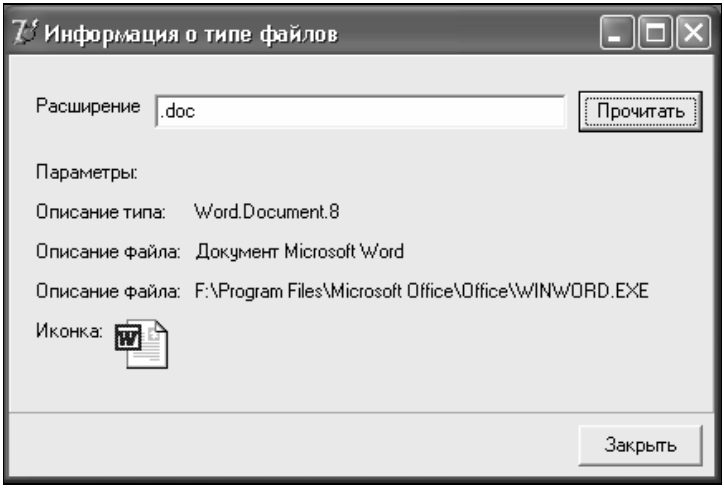


Рис. 6.17. Результат работы программы

На рис. 6.17 вы можете увидеть результат работы программы для расширения .doc. Обратите внимание, что перед именем расширения обязательно должна быть точка. Если пользователь не поставит точки, то может произойти ошибка, и мы вообще не получим никакой информации. Чтобы этого не произошло, можно модифицировать пример, добавив в самом начале следующую проверку:

```
var
 ...
 ...
 ExtStr:String;
begin
 ExtStr:=Edit1.Text;
 if ExtStr[1]<>'.' then
  Insert('.', ExtStr, 1);
 Edit1.Text:=ExtStr;
 ...
 ...
end;
```

Здесь я завел отдельную переменную, с помощью которой проверяю, если первый символ не равен точке, то она вставляется в первую позицию.

Примечание

На компакт-диске в каталоге \Примеры\Глава 6\Icon\ вы можете найти пример этой программы.

6.8.2. Связывание своей программы с неопределенным типом файлов

Теперь рассмотрим маленький пример, который будет отображать картинки в BMP-формате. Самое интересное в этой программе будет то, что мы зарегистрируем BMP-формат за нашей программой. Когда вы захотите просмотреть файл в этом формате, то будет запускаться наше приложение.

Итак, давайте создадим новый проект. На форме нам понадобится только компонент `TImage` (я растянул его по всей форме) и кнопка **Зарегистрировать**. Помимо этого, я установил у главной формы свойство `WindowsState` равным значению `wsMaximized`.

На рис. 6.18 вы можете увидеть форму нашей будущей программы.

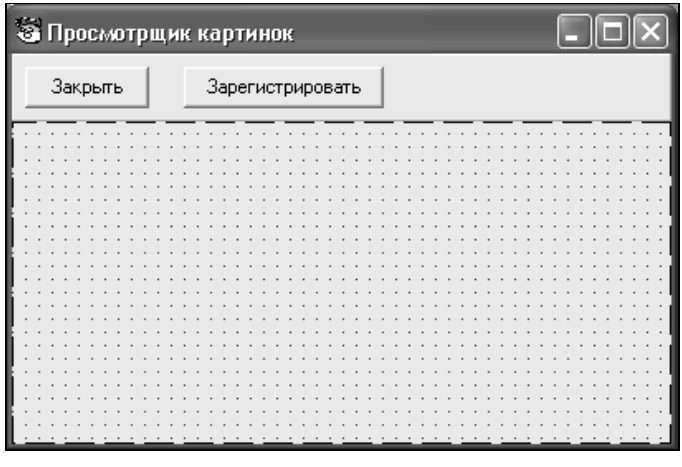


Рис. 6.18. Форма будущей программы

В обработчике нажатия кнопки **Зарегистрировать** пишем следующий код:

```
var
 Reg: TRegistry;
begin
 Reg := TRegistry.Create;
```

```
 Reg.RootKey := HKEY_CLASSES_ROOT;
 Reg.OpenKey('.BMP' , True);
 Reg.WriteString('', 'BMPfile');
 Reg.CloseKey;
 Reg.CreateKey('BMP'+'file_cyd');
 Reg.OpenKey('BMPfile\DefaultIcon', True);
 Reg.WriteString('', Application.ExeName + ',0');
 Reg.CloseKey;
 Reg.OpenKey('BMPfile\shell\open\command', True);
 Reg.WriteString('', Application.ExeName + ' "%1"');
 Reg.CloseKey;
 Reg.Free;
end;
```

В принципе, мы здесь записываем ту же информацию, которую читали в прошлом примере. Сначала открываем раздел `.BMP` и записываем туда значение по умолчанию — строку `BMPfile`. Эта строка будет восприниматься как имя раздела, в котором нужно искать информацию о программе, которая должна запускаться при запуске файла этого типа.

После этого создаем и открываем раздел `BMPfile`. Здесь записываем в раздел `DefaultIcon` значок, связанный с типом файла. В качестве значка указываем полный путь к своей программе и через запятую пишем `0`. Это значит, что для отображения файлов bmp-типа будет использоваться нулевой значок (основной) моей программы.

На рис. 6.19 показан один из моих каталогов с bmp-файлами. Посмотрите, какой значок установлен для данного типа файлов. Именно его я указал в качестве основного в своей программе.

Последнее, что мы обязательно должны сделать, — записать в раздел `BMPfile\shell\open\command` полный путь к программе, которая будет обрабатывать данный тип файлов. Сюда я записываю полный путь к своей программе.

Попробуйте скомпилировать пример, только не забудьте добавить в раздел `uses` модуль `Registry`, потому что наша программа работает с реестром. Теперь запустите программу и нажмите кнопку **Зарегистрировать**. Теперь, если попытаться запустить любой bmp-файл, то запустится не стандартный Paint или какая-либо другая программа, а именно наш пример.

Если вы уже попробовали описанное в действии, то наверно заметили, что когда вы дважды щелкаете по bmp-файлу, то наша программа стартует, но не обрабатывает выбранный файл. Это потому, что мы еще ничего не сдела-

ли для того, чтобы отображать изображения в компоненте `TImage`. Давайте исправим эту ситуацию.

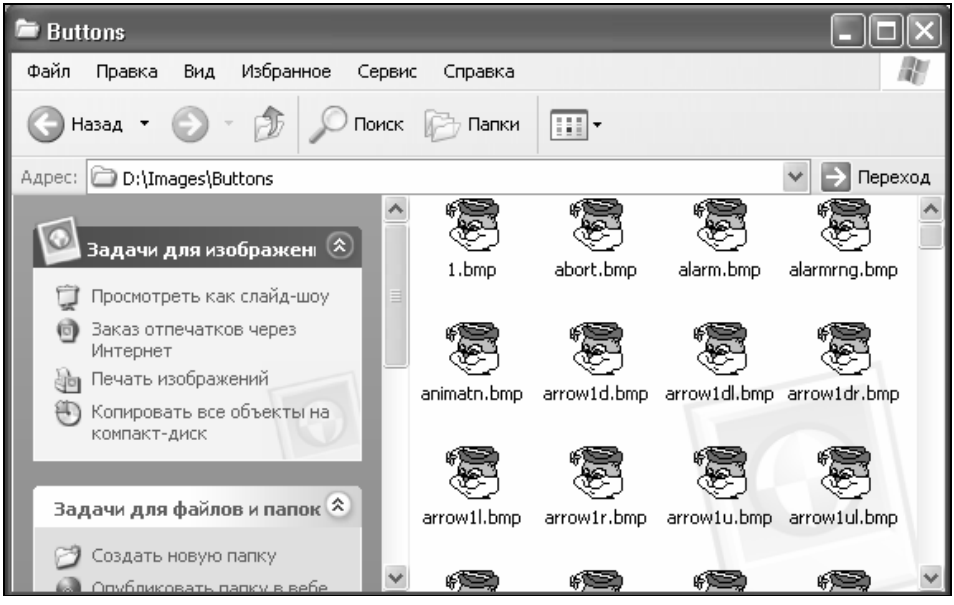


Рис. 6.19. Файлы bmp со значками от моей программы

Имя и полный путь к файлу передаются нам в качестве параметров. Чтобы считать эти параметры, мы должны создать обработчик события `OnShow` для главной формы и там написать следующий код:

```
procedure TForm1.FormShow(Sender: TObject);
var
 Str:String;
 i:Integer;
begin
 if (ParamCount > 0) then
  begin
   Str:=ParamStr(1);
   for i:=2 to ParamCount do
    begin
     Str:=Str+' '+ParamStr(i);
    end;
```

```
    Image1.Picture.LoadFromFile(Str);
  end;
end;
```

Здесь в первой строке проверяется значение переменной `ParamCount`. Эта переменная хранит количество параметров, переданных нашей программе. Если оно больше 0, то выполняется код загрузки изображения. Но для начала мы должны узнать имя файла, которое нужно загрузить. Если в имени файла нет пробелов, то его можно узнать из первого параметра — `ParamStr(1)`. Если пробелы есть, то имя может быть разбито на несколько параметров, и в этом случае мы должны их все сложить в одно целое. Для этого запускаем цикл:

```
Str:=ParamStr(1);
for i:=2 to ParamCount do
  begin
   Str:=Str+' '+ParamStr(i);
  end;
```

В этом цикле все переданные параметры объединяются в одну длинную строку.

Вот теперь мы имеем полный путь к файлу, который надо загрузить, и смело делаем это с помощью `Image1.Picture.LoadFromFile(Str)`.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 6\Image Viewer\ вы можете найти пример этой программы и цветные версии рисунков.

6.9. Работа со сканером

Еще в этой главе я хочу рассказать о работе со сканером и сканировании изображений. Сейчас сканирующие устройства стоят достаточно дешево и при этом обладают приемлемыми характеристиками. У меня достаточно много знакомых, которые имеют в своем арсенале подобные устройства и регулярно работают с отсканированными изображениями. В производстве так же люди находят применение сканирующим устройствам, поэтому умение работать со сканером прибавит вам немного неоценимого опыта.

Для работы со сканером у Delphi нет никаких компонентов или заголовочных файлов. Но на компакт-диске к этой книге вы сможете найти все необходимое в каталоге Headers\Scanner. Для компиляции примера из этого раз-

дела вы обязательно должны иметь два файла: eztwain.obj и MultiTWAIN.pas. Оба файла нужно поместить в один каталог с исходным кодом проекта.

В названии обоих файлов присутствует пять одинаковых букв — TWAIN. Это надстройка, через которую наша программа будет работать со сканером. Если в вашей системе установлен сканер, то зайдите в каталог, в который у вас установлена Windows, и вы увидите подкаталог twain или twain_32. Он устанавливается вместе с драйверами сканера и содержит библиотеки, упрощающие доступ к устройству. В этих библиотеках находится маленькая программа в виде dll-файла, которая будет вызываться, когда нам нужно что-то отсканировать. Таким образом, все программы, которые работают со сканером через этот интерфейс, будут использовать схожее окно для сканирования, и вам не надо думать над его работой — все будет происходить автономно.

Итак, создадим в Delphi новый проект. Сразу же сохраните его в каком-нибудь каталоге и поместите туда оба указанных файла. В раздел `uses` главной формы нужно добавить ссылку на модуль `MultiTwain`, чтобы вы могли использовать его возможности.

На форме нам понадобятся две кнопки (рис. 6.20):

- **Выбрать сканер** (если их несколько);
- **Сканировать**.

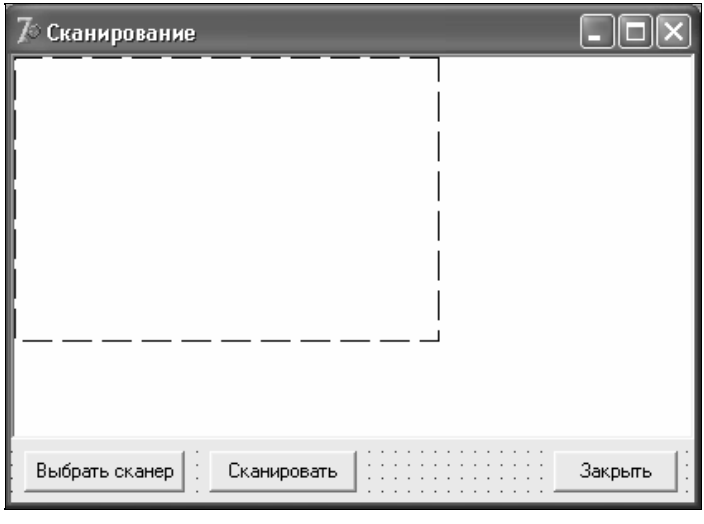


Рис. 6.20. Форма будущей программы

Помимо этого, по всей рабочей области окна я растянул компонент `TScrollBox`, который автоматически добавляет прокрутку. Внутрь этого компонента я поместил компонент `TImage`, в который мы будем помещать отсканированное изображение. У этого компонента нужно установить свойство `AutoSize` равным значению `true`. Если изображение будет больше, чем родительский компонент `TScrollBox`, то мы сможем пролистать его.

Теперь в разделе `private` объявим несколько переменных, которые нам пригодятся в процессе сканирования:

```
private
  { Private declarations }
    hdib, testdib: hbitmap;
    w,h,n: integer;
```

В обработчике события `OnCreate` формы нужно обнулить все переменные:

```
procedure TScanForm.FormCreate(Sender: TObject);
begin
 hDib := 0;
 testDib := 0;
 w := 0;
 h := 0;
end;
```

Теперь напишем код для выбора сканера. В обработчике нажатия соответствующей кнопки напишем следующий код:

```
procedure TScanForm.SetScanButtonClick(Sender: TObject);
begin
 TWAIN_SelectImageSource(0);
end;
```

Код прост, как никогда. Вам достаточно вызвать TWAIN-функцию `TWAIN_SelectImageSource`, и перед пользователем откроется стандартное окно выбора сканера.

В обработчике нажатия кнопки **Сканировать** нужно написать код, показанный в листинге 6.6.

Листинг 6.6. Код запуска сканирования

```
procedure TScanForm.ScanButtonClick(Sender: TObject);
begin
 Image1.Picture.Bitmap:=nil;
```

```
 hdib := TWAIN_AcquireNative(0, 0);
 n := TWAIN_GetNumDibs;
 if n >= 1 then
  begin
   TestDib := TWAIN_GetDib(0);
   Image1.Width:=TWAIN_DibWidth(TestDib);
   Image1.Height:=TWAIN_DibHeight(TestDib);
   CopyDibIntoImage(TestDib, Image1);
   TWAIN_FreeNative(TestDib);
   TestDib := 0;
  end;
 if n = 2 then
  begin
   TestDib := TWAIN_GetDib(1);
   Image1.Width:=TWAIN_DibWidth(TestDib);
   Image1.Height:=TWAIN_DibHeight(TestDib);
   CopyDibIntoImage(TestDib, Image1);
   TWAIN_FreeNative(TestDib);
   TestDib := 0;
  end;
end;
```

В первой строке кода мы очищаем текущее содержимое компонента `Image1`. Для этого присваиваем `Image1.Picture.Bitmap` значение `nil`.

Во второй строке вызывается TWAIN-функция `TWAIN_AcquireNative`. Этот метод отображает стандартное окно сканирования для вашего сканера. Пример моего окна для сканера Mustek 1200 ED вы можете увидеть на рис. 6.21. На этом выполнение нашего кода приостановится и продолжится только после окончания сканирования и закрытия появившегося окна.

После окончания сканирования нам необходимо выяснить, сколько реально страниц было отсканировано. Для этого вызываем TWAIN-функцию `TWAIN_GetNumDibs`. Как раз она нам и вернет необходимую информацию. Хотя я в любом случае собираюсь забирать только одну картинку, я сделал небольшой логический финт, выясняя зависимость от количества отсканированных страниц. Если отсканирован хотя бы один документ, то я буду забирать нулевой. Если отсканировано ровно 2, то я буду брать первый.

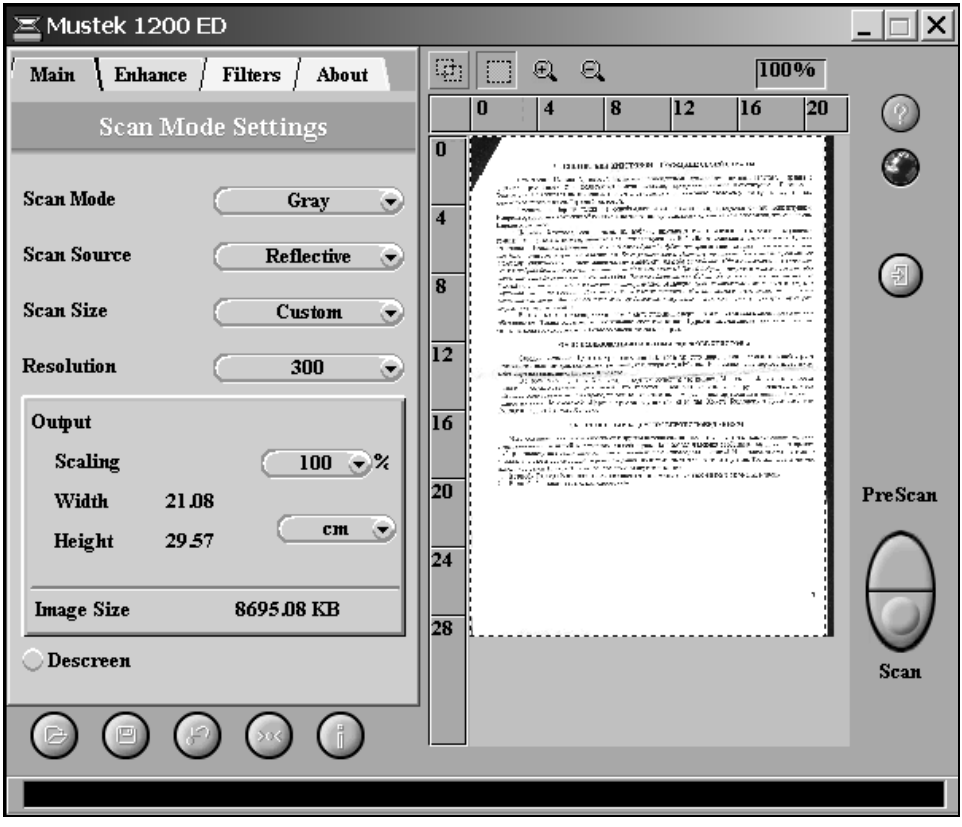


Рис. 6.21. Стандартное окно сканирования для моего сканера — Mustek 1200 ED

В любом случае, код получения изображения выглядит так:

```
TestDib := TWAIN_GetDib(Номер изображения);
Image1.Width:=TWAIN_DibWidth(TestDib);
Image1.Height:=TWAIN_DibHeight(TestDib);
CopyDibIntoImage(TestDib, Image1);
TWAIN_FreeNative(TestDib);
TestDib := 0;
```

В первой строке кода получаем нужное изображение с помощью функции `TWAIN_GetDib`. Ей нужно указать номер отсканированной картинки, которую мы хотим получить. В результате функция возвращает картинку в формате битовой матрицы — `hbitmap`. Для дальнейшей работы нам надо преобразовать этот формат и скопировать в компонент `Image1`.

Перед копированием нужно установить размеры компонента `Image1` равными размерам отсканированного изображения. Ширину битовой матрицы мы получаем с помощью функции `TWAIN_DibWidth`. В качестве параметра указываем переменную `TestDib`, а в результате получаем число — ширину изображения. Точно так же я получаю высоту с помощью функции `TWAIN_DibHeight`.

Теперь производим само копирование с помощью функции `CopyDibIntoImage`. В качестве первого параметра указываем битовую матрицу отсканированного изображения, а в качестве второго параметра — компонент `Image1`.

Вот теперь мы полностью получили необходимую картинку и можем освобождать выделенные ресурсы. Для этого сначала вызовем метод `TWAIN_FreeNative` для освобождения выделенной памяти в TWAIN-библиотеке, а затем уничтожим битовую матрицу простым обнулением ее переменной.

Результат работы программы представлен на рис. 6.22.

Рис. 6.22. Пример работы программы

После выхода первого издания книги было очень много вопросов по поводу работы с TWAIN. У некоторых читателей пример выдавал ошибку при обращении к функциям TWAIN. Я долго не мог понять, когда появляется ошибка, потому что с теми драйверами, с которыми работал я, проблем не было. Оказалось, что некоторые производители не включают в свои драйверы динамическую библиотеку twain.dll, через которую и происходит вся работа. Необходимые файлы twain.dll и twain_32.dll вы найдете на компакт-диске вместе с исходным кодом примера или в каталоге Исходники\Scanner.

ПРИМЕЧАНИЕ

На компакт-диске в каталоге \Примеры\Глава 6\Scaner\ вы можете найти пример программы работы со сканером и цветные версии рисунков данного раздела.

6.10. IP-config собственными руками

Вы, наверно, помните такую прекрасную утилиту Winipcfg.exe, которая преследовала нас на протяжении всего существования линейки Windows 9*x*. Лично мне эта утилита очень нравилась, и по моей практике могу сказать, что ею пользовалось очень много народа. Я регулярно слышу не очень приличные слова в сторону Билла за то, что в Windows NT (2000, XP) нет такой программы, и теперь получение информации о конфигурации IP немного неудобное. Привык уже народ к этой утилите, и, убрав ее из дистрибутива Windows, Билл словно отобрал у ребенка погремушку.

Ну, ничего, скоро вы сможете написать собственную утилиту, которая будет выводить подробную информацию о сетевых настройках. Чтобы получить информацию, которую нам показывала утилита Winipcfg, для Delphi нужно иметь дополнительные заголовочные файлы: IpExport.pas, IpHlpApi.pas, IpIfConst.pas, IpRtrMib.pas и IpTypes.pas. Для любителей C++ подобные заголовочные файлы можно взять из специального сетевого SDK, который легко найти на сайте Microsoft. Ну а для Delphi вы сможете найти эти файлы в каталоге Headers\IP компакт-диска. Эти файлы нужно расположить так, чтобы Delphi их нашла.

Итак, будем считать, что файлы вы скопировали, куда нужно. Запускайте Delphi и создавайте новый проект. Сразу же перейдите в окно кода и добавьте в раздел `uses` имена следующих модулей: `IpHlpApi`, `IpTypes`, `IpIfConst`.

Вот теперь перейдем к созданию формы будущей программы. Посмотрите на рис. 6.23 и попробуйте создать нечто подобное.

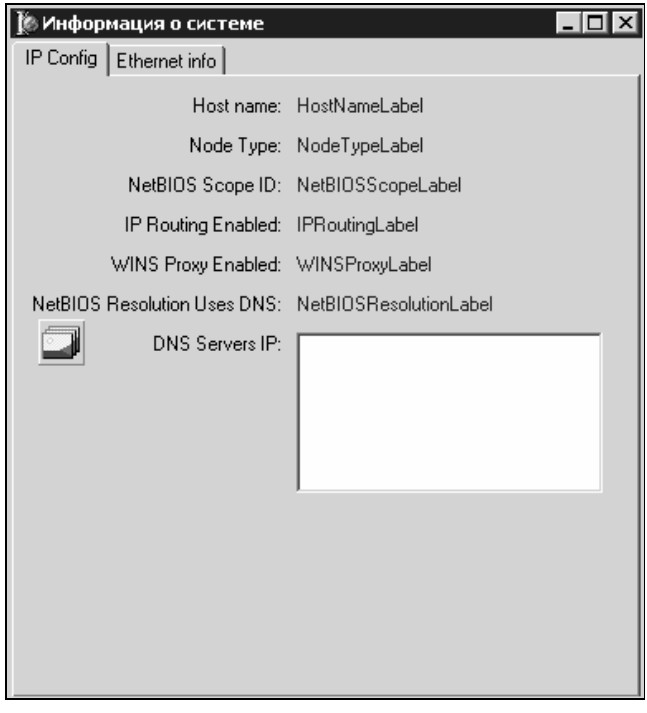


Рис. 6.23. Форма будущей программы

Несколько слов о дизайне. По всей форме у меня растянут компонент `PageControl`. На нем я создал две вкладки: **IP Config** и **Ethernet info**. Для начала мы научимся получать всю информацию, которая расположена на первой вкладке, а вторую оставим до *разд. 6.11*.

Теперь создайте обработчик события `OnChange` для компонента `PageControl1`. Когда пользователь будет менять вкладку, мы должны будем обновлять информацию о конфигурации протокола IP. Пока что напишите в этом обработчике следующий код:

```
if PageControl1.ActivePageIndex=0 then
  GetIPInfo;
```

Этот код выполняет следующее: если сейчас выделена первая вкладка, то вызвать процедуру `GetIPInfo`. Все, подготовка закончена, переходим к программированию этой загадочной процедуры `GetIPInfo`.

Добавьте в разделе `private` нашей формы следующее:

```
procedure GetIPInfo;
```

Таким образом, мы объявим эту злосчастную процедуру. Теперь нажмите комбинацию клавиш <Ctrl>+<Shift>+<C>, и Delphi подготовит для вас заготовку новой процедуры, в которую нужно вставить код из листинга 6.7.

Листинг 6.7. Процедура `GetIPInfo`

```
procedure TSystemInfoForm.GetIPInfo;
var
 FixedInfoSize, Err:DWORD;
 pFixedInfo:PFIXED_INFO;
 pAddrStr:PIP_ADDR_STRING;
begin
 FixedInfoSize:=0;
 Err:=GetNetworkParams(nil, FixedInfoSize);
 if (Err<>0) and (Err<>ERROR_BUFFER_OVERFLOW) then
  begin
   HostNameLabel.Caption:='Error';
   exit;
  end;
 pFixedInfo:=PFIXED_INFO(GlobalAlloc(GPTR, FixedInfoSize));
 GetNetworkParams(pFixedInfo, FixedInfoSize);
 HostNameLabel.Caption:=StrPas(pFixedInfo.HostName);
 DNSListBox.Items.Clear;
 DNSListBox.Items.Add(StrPas(pFixedInfo.DnsServerList.IpAddress.S));
 pAddrStr:=pFixedInfo.DnsServerList.Next;
 while (pAddrStr<>nil) do
  begin
   DNSListBox.Items.Add(StrPas(pAddrStr.IpAddress.S));
   pAddrStr:=pAddrStr.Next;
  end;
 case pFixedInfo.NodeType of
  1: NodeTypeLabel.Caption:='Broadcast';
  2: NodeTypeLabel.Caption:='Peer to peer';
  4: NodeTypeLabel.Caption:='Mixed';
  8: NodeTypeLabel.Caption:='Hybrid';
 end;
```

```
 NetBIOSScopeLabel.Caption:=pFixedInfo.ScopeId;
 if pFixedInfo.EnableRouting>0 then
  IPRoutingLabel.Caption:='Yes'
 else
  IPRoutingLabel.Caption:='No';
 if pFixedInfo.EnableProxy>0 then
  WINSProxyLabel.Caption:='Yes'
 else
  WINSProxyLabel.Caption:='No';
 if pFixedInfo.EnableDns>0 then
  NetBIOSResolutionLabel.Caption:='Yes'
 else
  NetBIOSResolutionLabel.Caption:='No';
end;
```

Начнем рассматривать весь этот код по частям, потому что сразу разобраться с чем-то громоздким очень трудно. В самом начале у нас выполняется следующее:

```
Err:=GetNetworkParams(nil, FixedInfoSize);
if (Err<>0) and (Err<>ERROR_BUFFER_OVERFLOW) then
 begin
  HostNameLabel.Caption:='Error';
  exit;
 end;
pFixedInfo:=PFIXED_INFO(GlobalAlloc(GPTR, FixedInfoSize));
GetNetworkParams(pFixedInfo, FixedInfoSize);
```

Первое, что надо сделать, — это вызвать функцию `GetNetworkParams` с двумя параметрами. Она возвращает информацию о сети. Но для получения этих данных нам нужно знать их размер. Чтобы это сделать, нужно сначала первый параметр установить равным `nil`, а второй — это переменная, куда будет записан размер необходимой памяти для получения полной информации. Результат выполнения функции записывается в переменную `Err`, чтобы потом мы могли проверить его на ошибки. Если эта переменная не равна 0 и не равна константе `ERROR_BUFFER_OVERFLOW`, то точно была какая-то ошибка, и надо вывести об этом сообщение и выйти из процедуры. Такое возможно, если сетевая карта не настроена.

После этого мы выделяем память для структуры pFixedInfo типа PFIXED_INFO, куда мы будем записывать полученную информацию. Память нужно выделить в том количестве, которое мы узнали из первого вызова GetNetworkParams.

Далее снова вызывается функция GetNetworkParams, только теперь в качестве первого параметра указана структура pFixedInfo.

В принципе, все необходимое мы получили, и оно находится в pFixedInfo. Так что остается только рассмотреть эту структуру:

- pFixedInfo.HostName — имя хоста (вашего компьютера);
- pFixedInfo.DnsServerList.IpAddress — адрес DNS-сервера. Если такой сервер не один, то нужно вызвать pFixedInfo.DnsServerList.Next, чтобы получить доступ к следующему. В качестве результата вызова Next нам вернется переменная типа PIP_ADDR_STRING, через которую и можно получить следующий адрес DNS-сервера. В общем случае код будет выглядеть так:

```
pAddrStr:=pFixedInfo.DnsServerList.Next;
while (pAddrStr<>nil) do
  begin
   pAddrStr.IpAddress.S    // Здесь находится следующий адрес
   pAddrStr:=pAddrStr.Next // Получить еще адрес, если есть
  end;
```

- pFixedInfo.NodeType — тип узла. Здесь хранится целое число. Если оно равно 1, то, значит, тип узла Broadcast, 2 — Peer to peer, 4 — Mixed, 8 — Hybrid;
- pFixedInfo.ScopeId — идентификатор NetBIOS (NetBIOS Scope ID);
- pFixedInfo.EnableRouting — включена ли маршрутизация IP. Если здесь хранится true, то маршрутизация включена, иначе отключена;
- pFixedInfo.EnableProxy — включен ли WINS Proxy. Если здесь находится true, то прокси-сервер включен, иначе отключен.

Вот и все. Достаточно только вызвать одну процедуру, и вы уже знаете так много интересного о компьютере и его сетевых настройках. В следующем разделе я покажу еще одну процедуру, с помощью которой мы узнаем еще немало о своем любимце.

На рис. 6.24 вы можете увидеть результат работы. Как видите, пример оказался достаточно простым, самое сложное еще впереди.

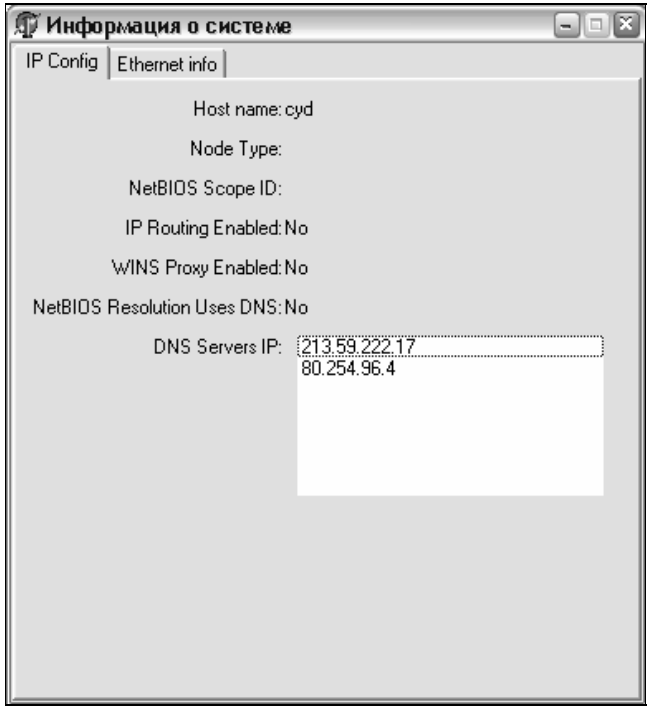


Рис. 6.24. Результат работы программы в Windows XP

6.11. Получение информации о сетевом устройстве

Здесь я закончу рассказ, начатый в предыдущем разделе. Напоминаю, что там мы создали форму и поместили на нее компонент `TPageControl` с двумя вкладками. Мы научились получать от системы все, что находится на первой вкладке, и теперь осталось только заполнить вторую вкладку полезной информацией.

Сейчас нам предстоит познакомиться с самым интересным — мы научимся получать из системы количество установленных сетевых устройств и их свойства. Среди этих свойств будет определение IP-адреса и маски сети. Только что я заглянул в свой почтовый клиент и понял, что вопрос о том, как определить IP-адрес, — чуть ли не самый популярный. Но я эту тему пока не затрагивал, потому что способ, который я покажу сейчас, будет дос-

таточно сложным. Зато он позволяет определить IP и маску любого сетевого устройства, установленного в системе.

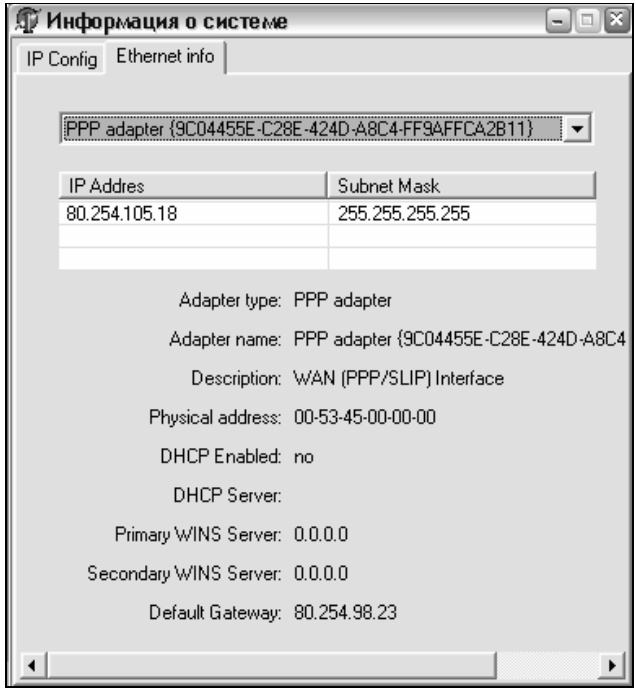


Рис. 6.25. Форма второй вкладки программы IP Config

Откройте проект, который мы создали в предыдущем разделе, и подкорректируйте вторую вкладку в соответствии с рис. 6.25. На этой вкладке у меня вверху расположен компонент `TComboBox`, в котором будет выводиться список установленных в системе сетевых устройств. Чуть ниже находится компонент `TListView`, у которого нужно установить следующие свойства:

- `Name` — измените имя компонента на `IPListView`;
- `ViewStyle` — здесь нужно указать `vsReport`;
- `Columns` — здесь нужно создать две колонки с именами `IP Address` и `Subnet Mask` (IP-адрес и маска сети). Я люблю писать такие вещи на английском языке, но если вы хотите, то можете писать хоть на китайском.

Дальше идут компоненты `TLabel`, с помощью которых мы будем выводить полученную из системы информацию.

Теперь научимся получать список установленных в системе сетевых устройств. Я думаю, что это нужно делать по событию `OnShow` для формы (когда форма появляется на экране). В этом случае, проверка будет происходить только при появлении формы, а потом мы будем только получать информацию о выбранном устройстве.

Где-то я видел утилиту, которая обновляла этот список при каждом обращении пользователя к любому элементу. Лично я не вижу в этом смысла. Если вы считаете, что за время выполнения программы список может измениться (например, включили новое USB-устройство), то для таких случаев лучше добавить кнопку **Обновить** и не мучить пользователя бесполезными задержками и нагрузками.

Итак, в обработчике события `OnShow` главной формы пишем код из листинга 6.8.

Листинг 6.8. Получение списка сетевых устройств

```
procedure TSystemInfoForm.FormShow(Sender: TObject);
var
 pAdapterInfo, pAdapt:PIP_ADAPTER_INFO;
 pAddrStr:PIP_ADDR_STRING;
begin
 // Очищаем список устройств
 AdapterCB.Items.Clear;
 // Получить количество устройств
 AdapterInfoSize:=0;
 Err:=GetAdaptersInfo(nil, AdapterInfoSize);
// Если произошла ошибка, то...
 if (Err<>0) and (Err<>ERROR_BUFFER_OVERFLOW) then
  begin
   AdapterCB.Items.Add('Error');
   exit;
  end;
// Получить информацию об устройствах
 pAdapterInfo := PIP_ADAPTER_INFO(GlobalAlloc(GPTR, AdapterInfoSize));
 GetAdaptersInfo(pAdapterInfo, AdapterInfoSize);
 pAdapt := pAdapterInfo;
// Проверяем тип полученного адаптера
 while pAdapt<>nil do
```

```
  begin
   case pAdapt.Type_ of
    MIB_IF_TYPE_ETHERNET:
     AdapterCB.Items.Add('Ethernet adapter '+pAdapt.AdapterName);
    MIB_IF_TYPE_TOKENRING:
     AdapterCB.Items.Add('Token Ring adapter '+pAdapt.AdapterName);
    MIB_IF_TYPE_FDDI:
     AdapterCB.Items.Add('FDDI adapter '+pAdapt.AdapterName);
    MIB_IF_TYPE_PPP:
     AdapterCB.Items.Add('PPP adapter '+pAdapt.AdapterName);
    MIB_IF_TYPE_LOOPBACK:
     AdapterCB.Items.Add('Loopback adapter '+pAdapt.AdapterName);
    MIB_IF_TYPE_SLIP:
     AdapterCB.Items.Add('Slip adapter '+pAdapt.AdapterName);
    MIB_IF_TYPE_OTHER:
     AdapterCB.Items.Add('Other adapter '+pAdapt.AdapterName);
   end;
   pAdapt := pAdapt.Next;
  end;
 GlobalFree(Cardinal(pFixedInfo));
end;
```

Я постарался снабдить код подробными комментариями, чтобы вы смогли разобраться с тем, что здесь происходит. Самое сердце этой процедуры — вызов функции `GetAdaptersInfo`. Первый раз она вызывается с первым параметром, равным 0. Это заставляет ОС сообщить нам, сколько памяти необходимо для хранения информации об установленных устройствах. Эту информацию мы получаем через переменную, указанную во втором параметре.

После этого выделяем необходимое количество памяти. Это происходит с помощью функции `GlobalAlloc`, которая выделяет память в глобальной памяти машины. Желательно использовать именно эту функцию. Я пробовал выделять память в другом месте (не в глобальной области), и в этом случае программа выдавала ошибку или вообще ничего не выводила.

Во второй раз функция `GetAdaptersInfo` вызывается уже со всеми нормальными параметрами. В первом мы указываем на выделенную память, а второй параметр указывает на количество этой памяти.

После получения необходимой информации об установленных устройствах я заполняю выпадающий список `ComboBox` именами найденных сетевых плат (эти имена находятся в `pAdapt.AdapterName`). Я также прибавляю к имени адаптера его тип, который можно определить по свойству `Type` структуры `pAdapt`. Это свойство может принимать следующие значения:

- `MIB_IF_TYPE_ETHERNET` — сетевой адаптер Ethernet;
- `MIB_IF_TYPE_TOKENRING` — адаптер Token Ring;
- `MIB_IF_TYPE_FDDI` — адаптер FDDI;
- `MIB_IF_TYPE_PPP` — PPP-адаптер;
- `MIB_IF_TYPE_LOOPBACK` — адаптер LoopBack;
- `MIB_IF_TYPE_SLIP` — Slip-адаптер;
- `MIB_IF_TYPE_OTHER` — другое.

Теперь у нас в выпадающем списке `ComboBox` находятся имена всех найденных устройств, и нам надо отслеживать событие, когда пользователь выбрал какой-то элемент из списка, и выводить информацию, относящуюся к этому элементу. Чтобы поймать такое событие, мы должны создать обработчик `OnChange` для выпадающего списка. В этом обработчике мы должны получить количество установленных устройств. Для этого вызываем функцию `GetAdaptersInfo` с нулевым первым параметром, чтобы узнать количество необходимой памяти, для хранения информации об устройствах:

```
AdapterInfoSize:=0;
Err:=GetAdaptersInfo(nil, AdapterInfoSize);
```

Некоторые в этом месте скажут, что мы уже знаем количество устройств и выполняли этот код, когда заполняли выпадающий список. А я на это скажу, что это было слишком давно. Если бы я писал этот код лет пять назад, то я действительно не запрашивал лишний раз количество устройств, а узнал его по количеству элементов в списке. Сейчас, во времена правления USB, я ни в чем не могу быть уверенным. Только что могло быть только 2 сетевых устройства, а через минуту их может быть три. Поэтому лучше лишний раз перестраховаться. Главное — не делать это слишком часто.

Далее я снова получаю список устройств:

```
pAdapterInfo := PIP_ADAPTER_INFO(GlobalAlloc(GPTR, AdapterInfoSize));
GetAdaptersInfo(pAdapterInfo, AdapterInfoSize);
pAdapt := pAdapterInfo;
```

После этого мы запускаем цикл `while`, в котором происходит проверка всех названий из вновь полученного списка устройств с тем именем, которое вы-

брал пользователь. Как только эта запись найдена, нужно считать информацию из структуры pAdapterInfo:

```
IP_ADAPTER_INFO = record
Next: PIP_ADAPTER_INFO;
 ComboIndex: DWORD;
 AdapterName: array [0..MAX_ADAPTER_NAME_LENGTH + 3] of Char;
 Description: array [0..MAX_ADAPTER_DESCRIPTION_LENGTH + 3] of Char;
 AddressLength: UINT;
 Address: array [0..MAX_ADAPTER_ADDRESS_LENGTH – 1] of BYTE;
 Index: DWORD;
 Type_: UINT;
 DhcpEnabled: UINT;
 CurrentIpAddress: PIP_ADDR_STRING;
 IpAddressList: IP_ADDR_STRING;
 GatewayList: IP_ADDR_STRING;
 DhcpServer: IP_ADDR_STRING;
 HaveWins: BOOL;
 PrimaryWinsServer: IP_ADDR_STRING;
 SecondaryWinsServer: IP_ADDR_STRING;
 LeaseObtained: time_t;
 LeaseExpires: time_t;
end;
```

Здесь вы видите следующие свойства:

- ComboIndex — индекс устройства;
- AdapterName — с этим мы уже встречались и это имя сетевого адаптера;
- Description — текстовое описание адаптера;
- AddressLength — длина физического (MAC) адреса;
- Address — массив символов, в котором хранится сам адрес. Длина массива определяется предыдущим параметром;
- Type — тип адаптера. Это свойство я расписал немного выше;
- DhcpEnabled — указывает на использование DHCP;
- CurrentIpAddress — текущий IP-адрес;
- GatewayList — список шлюзов;
- DHCPServer — IP-адрес DHCP-сервера;

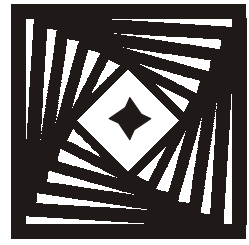
- `HaveWins` — используется ли WINS-сервер;
- `PrimaryWinsServer` — главный WINS-сервер;
- `SecondaryWinsServer` — подчиненный WINS-сервер.

Это основные и часто используемые свойства, которые вы можете прочитать из структуры `P_ADAPTER_INFO`.

Вот теперь у вас есть полноценный и собственный `IPConfig`. Вы можете модифицировать его на свой вкус и цвет, а я надеюсь, что дал вам достаточно подробную информацию для того, чтобы вы могли продолжить самостоятельное улучшение этого примера.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 6\IP Config\ вы можете найти пример данной программы.

Глава 7



Полезное

Эта глава в основном будет посвящена сети, но не протоколу TCP/IP, который мы уже рассмотрели в *главе 5*. Здесь мы будем обсуждать протоколы NetBIOS, ARP, а также различные утилиты, которые так или иначе связаны с сетью.

Нам предстоит узнать немного нового о системе, написать пару программ-шуток и снова вернуться к сетям. В *главе 5* мы подробно рассмотрели протокол TCP/IP, а в этой главе нам предстоит окунуться в мир NetBIOS и искупаться в ARP. Почему мы всего лишь окунемся в NetBIOS? Да потому, что этот протокол не пользуется такой популярностью, как TCP, и с каждым днем имеющаяся популярность падает.

В общем, читайте, будет интересно. Особенно тем, кто не видел первого издания, но даже если вы уже знакомы с ним, я надеюсь, что вы найдете что-то интересное в обновленном варианте книги.

7.1. Работа с NetBIOS

Я не собираюсь описывать все возможности NetBIOS, потому что эта тема заслуживает отдельного разговора продолжительностью не в одну сотню страниц, но основные сведения постараюсь дать. Описанного в этом разделе не хватит для создания профессионального приложения, работающего по протоколу NetBIOS, но будет достаточно для продолжения дальнейшего самостоятельного обучения, если вам вдруг понадобится этот доисторический и мало используемый протокол.

Протокол NetBIOS достаточно прост, потому что API протокола состоит только из одной функции NetBIOS. Несмотря на то, что функция одна, она

умеет больше чем любая функция из других сетевых библиотек. Именно поэтому я ее не смогу описать всю, но небольшие начальные сведения постараюсь предоставить.

В Windows за работу NetBIOS отвечает библиотека netapi32.dll. Это значит, что поклонникам языка C++ нужны будут заголовочные файлы и файл библиотеки netapi32.lib. Нам, программистам на Delphi, немного легче и достаточно только заголовочного файла nb.pas. Его вы найдете на компакт-диске вместе с исходным кодом примера из этого раздела или в каталоге Headers\nb.

Как я уже сказал, вся библиотека NetBIOS посвящена одноименной функции, и ее объявление выглядит вот таким образом:

```
function NetbiosCmd(var NCB: TNCB): Word;
```

Как видите, у нее только один параметр, но зато какой... — структура `NCB`. Это достаточно сложная структура, в которой и заключена вся работа. С ее помощью мы будем говорить библиотеке, что от нее требуется, а также принимать и получать любые данные.

Теперь перейдем к более конкретному изучению структуры `NCB`. Ее описание для Windows выглядит так:

```
TNCB = packed record
  Command: byte;
  RetCode: byte;
  LSN: byte;
  Num: byte;
  Buf: ^byte;
  Length: word;
  CallName: TNBName;
  Name: TNBName;
  RTO: byte;
  STO: byte;
  PostPrc:  TNCBPostProc;
  Lana_Num: byte;
  Cmd_Cplt: byte;
  Reserved: array[0..9] of byte;
  Event: THandle;
end;
```

Рассмотрим поля структуры.

- Первый параметр (Command) указывает на команду, которую необходимо выполнить. Я не смогу их описать все, но советую заглянуть в заголовочный файл и поискать константы, начинающиеся на NCB_. Все это (кроме NCB_ASYNC) и есть имена констант, указывающих на определенные команды. Константа NCB_ASYNC имеет собственное особое значение. Если вы просто укажете необходимую команду, то она будет выполнена синхронно. Но если ее поразрядно логически сложить (для этого вместо знака + указывают and, хотя и простое сложение тоже сработает) с константой NCB_ASYNC, то команда будет уже выполнена асинхронно.
- Второй параметр (RetCode) содержит код результата выполнения команды. Если вы выполняете ее асинхронно, то NetBIOS не сможет сразу вернуть результат. Поэтому в данном случае сюда будет помещено значение $ff или константа NRC_PENDING, которая означает, что асинхронная команда еще не выполнена. Константы возвращаемых значений можно найти в заголовочном файле, и начинаются они с NRC_.
- Параметр LSN — номер локального сеанса, который вы можете получить после выполнения команд NCB_CALL (открыть сессию) и NCB_LISTEN (ждать вызова).
- Num — номер сетевого имени. Такие номера получаются после вызова команд NCB_ADDNAME (добавить уникальное имя в локальную таблицу) и NCB_ADDGRPNAME (добавить имя группы в локальную таблицу).
- Следующий параметр (Buf) — это буфер, в котором нужно разместить данные, подлежащие отправке в сеть, или получить данные, принятые из сети.
- Length — длина буфера. По этому числу библиотека сможет узнать, сколько данных вы хотите отправить в сеть или сколько хотите получить.
- Параметр CallName — это имя удаленного приложения.
- Противоположность предыдущему Name — имя вашей программы.
- Далее идет RTO — время ожидания (time-out) при получении данных. Учтите, что вы указываете число единиц времени, а одна единица равна 500 миллисекундам. Одна секунда равна 1000 миллисекундам, а значит, если указать число 2, то мы попросим ожидать приема ровно 1 секунду.
- Противоположностью предыдущему является STO — время ожидания отправки данных по сети. Так же указывается в единицах, где одна единица равна 500 миллисекундам.

- `PostPrc` указывает на процедуру, которую необходимо выполнить после выполнения команды в асинхронном режиме. Такая процедура должна иметь вид:

  ```
  TNCBPostProc = procedure(P: PNCB);
  ```

  Это значит, что она обязана иметь один и только один параметр в виде переменной типа `PNCB`, т. е. она — указатель на структуру `TNCB`.

  Если вы работаете в доисторической Windows 3.11 (примите мои соболезнования), то этот параметр будет состоять из пары параметров `Post_Offs` и `Post_Seg` (указатель на сегмент и смещение).
- `Lana_Num` — номер адаптера, с которым необходимо работать.
- `Cmd_Cplt` — это код выполнения команды. Здесь также при асинхронной работе будет стоять значение `$ff` или константа `NRC_PENDING`.
- `Reserved` — зарезервированный параметр и должен равняться нулю.
- `Event` — удобная фишка Win32 — событие. Его удобно использовать при работе в асинхронном режиме, когда необходимо узнать момент окончания выполнения асинхронной операции.

Рис. 7.1. Форма программы определения MAC-адреса

Напишем пример, который будет определять MAC-адрес. Для этого создайте новое приложение, а главную форму сделайте такой, как показано на рис. 7.1. В обработчике события нажатия кнопки **Найти** напишите следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 L_Enum : TLana_Enum;
 i:Integer;
begin
 L_Enum := NbLanaEnum;
 for i := 0 to (L_Enum.Length - 1) do
```

```
  begin
   NbReset(L_Enum.Lana[i]);
   NetBIOSLabel.Caption:=NbGetMacAddr(Edit1.Text,i);
   if NetBIOSLabel.Caption<>'?' then break;
  end;
end;
```

Прежде чем использовать протокол NetBIOS, вызывается функция `NbLanaEnum`, в которой происходит перечисление всех доступных в компьютере сетевых устройств. Для этого используется NetBIOS-команда `NCB_ENUM`. Функция `NbLanaEnum` написана нами и выглядит вот так:

```
function TForm1.NbLanaEnum: TLana_Enum;
var
 NCB: TNCB;
 L_Enum: TLana_Enum;
 RetCode: Word;
begin
 FillChar(NCB, SizeOf(NCB), 0);
 FillChar(L_Enum, SizeOf(TLana_Enum), 0);
 NCB.Command := NCB_ENUM;
 NCB.Buf := @L_Enum;
 NCB.Length := Sizeof(L_Enum);
 RetCode := NetBiosCmd(NCB);
 if RetCode <> NRC_GOODRET then
  begin
   L_Enum.Length := 0;
   L_Enum.Lana[0] := Byte(RetCode);
  end;
 Result := L_Enum;
end;
```

В первой строке заполняется структура `NCB` с помощью функции `FillChar`. В следующей строке происходит то же самое с переменной `L_Enum`. Далее идет заполнение структуры `NCB`. Указан тип команды `NCB_ENUM` (говорит, что необходимо перечислить сетевые устройства), а также буфер (свойство `Buf`) и его размер (свойство `Length`). После этого выполняется NetBIOS-команда с помощью функции `NetBiosCmd`. Если результат нормальный, то моя функция возвращает количество найденных устройств.

После получения информации о сетевых устройствах запускается цикл, в котором перебираем эти устройства и ищем у них MAC-адрес:

```
for i := 0 to (L_Enum.Length - 1) do
  begin
   NbReset(L_Enum.Lana[i]);
   NetBIOSLabel.Caption:=NbGetMacAddr(Edit1.Text,i);
   if NetBIOSLabel.Caption<>'?' then break;
  end;
```

Далее запускается цикл для всех найденных устройств (`LANA`). Внутри цикла по идее мы должны просто определить MAC-адрес устройства, но не тут то было. В NetBIOS, прежде чем использовать любой номер адаптера `LANA`, его надо обнулить. Для этого вызываем свою процедуру `NbReset`, в которой выполняем NetBIOS-команду `NCB_RESET`. Не пренебрегайте этим действием, даже если уверены, что программа сработает без этой команды. Никогда нельзя быть уверенным на 100%. Вот так выглядит функция `NbReset`:

```
function TForm1.NbReset(l: Byte): Word;
var
 NCB: TNCB;
begin
 FillChar(NCB, SizeOf(NCB), 0);
 NCB.Command := NCB_RESET;
 NCB.Lana_Num := l;
 Result := NetBiosCmd(NCB);
end;
```

Ну а теперь, после перечисления и обнуления можно смело вызывать функцию `NbGetMacAddr`, которая как раз и определит соответствующий устройству MAC-адрес:

```
function TForm1.NbGetMacAddr(Name: String; LanaNum: Integer): String;
var
 NCB: TNCB;
 AdpStat: TAdpStat;
 RetCode: Word;
 i:Integer;
begin
 FillChar(NCB, SizeOf(NCB), 0);
 FillChar(AdpStat, SizeOf(AdpStat), 0);
 NCB.Command := NCB_ADPSTAT;
```

```
 NCB.Buf := @AdpStat;
 NCB.Length := Sizeof(AdpStat);
 FillChar(NCB.CallName, Sizeof(TNBName), $20);
//  NCB.CallName[0] := Byte('*');
 for i:=0 to Length(Name)-1 do
  NCB.CallName[i] := Byte(Name[i+1]);
 NCB.Lana_Num := LanaNum;
 RetCode := NetBiosCmd(NCB);
 if RetCode = NRC_GOODRET then
  begin
   Result := Format('%2.2x:%2.2x:%2.2x:%2.2x:%2.2x:%2.2x',
             [AdpStat.ID[0], AdpStat.ID[1], AdpStat.ID[2],AdpStat.ID[3],
              AdpStat.ID[4], AdpStat.ID[5]]);
  end
 else
  begin
   Result := '?';
  end;
end;
```

Обратите внимание на одну закомментированную строку:

```
NCB.CallName[0] := Byte('*');
```

Если в параметре `CallName` указать звездочку, то вы укажите на локальную машину, и не обязательно знать ее NetBIOS-имя. Если нужна удаленная тачка, то делайте именно так, как в примере.

В этом разделе я дал вам теорию, которую нельзя преподнести с помощью примера. Без понимания всего описанного невозможно изучать NetBIOS.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 7\NB\ вы можете найти пример программы.

7.2. Работа с ARP

Прежде чем приступить к программированию, мне надо дать некоторые пояснения относительно сетевого протокола ARP. Не все знают, что это такое, и тем более, как с ним работать. ARP (Address Resolution Protocol) — это

протокол сопоставления адреса. RARP — это протокол, выполняющий действия, обратные ARP.

Любой пакет, передаваемый по сети, должен содержать в себе MAC-адрес (аппаратный адрес сетевого устройства). Этот адрес прошит производителем в сетевом устройстве. Если вы хотите узнать MAC-адрес своей карты и у вас стоит Windows 9*x*/ME, то запустите Ipconfig.exe или winipconfig.exe из каталога Windows. Для winipconfig.exe нажмите кнопку **Сведения**, и вы сможете увидеть окно, как на рис. 7.2.

В выпадающем списке вы можете увидеть PPP-адаптер (если вы подключены к Интернету) и имя своей сетевой карты (если она есть). Выбирая одно из них, вы можете увидеть их свойства.

Для Windows 2000/XP этот адрес можно увидеть, если запустить программу **Сведения о системе**, выбрав **Пуск | Все программы | Стандартные | Служебные**. В этой программе нужно в левой части окна открыть ветви дерева **Компоненты | Сеть | Адаптер** (рис. 7.3).

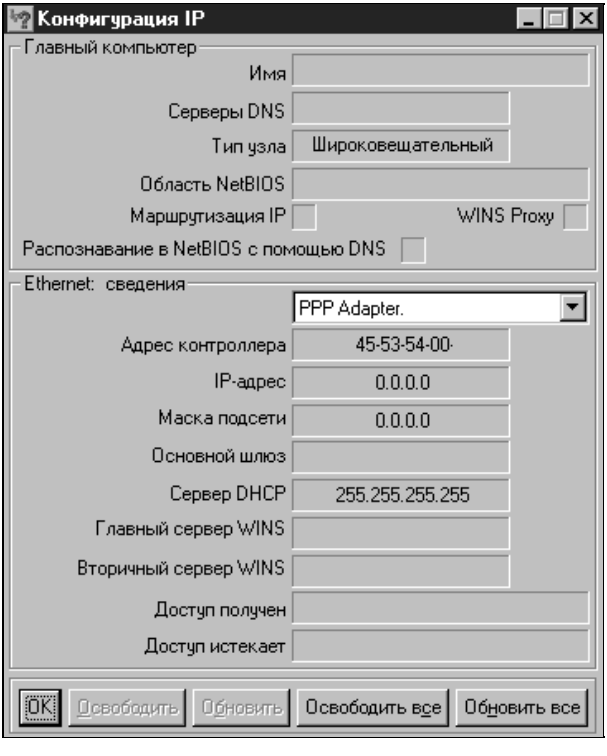


Рис. 7.2. Просмотр MAC-адреса в Windows 9*x*/ME

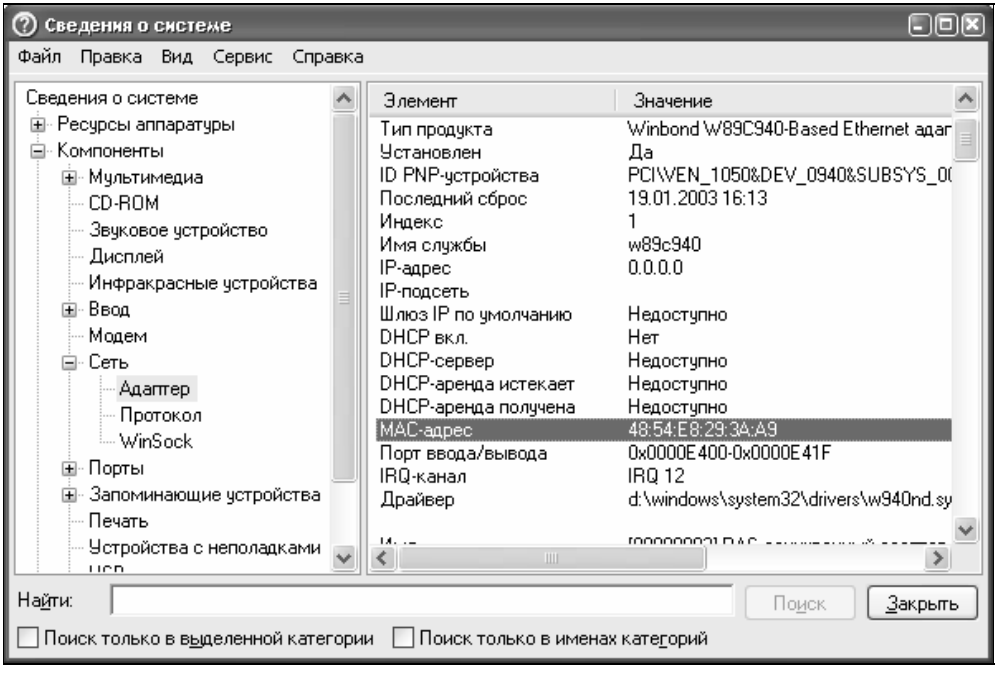


Рис. 7.3. Просмотр MAC-адреса в Windows 2000/XP

Итак, прежде чем пакет будет отправлен, машина должна знать адрес получателя. Протокол ARP занимается поиском этого адреса. В общем случае поиск MAC-адреса выполняется следующим образом:

1. Сначала происходит поиск в кэше. Если адрес не найден, то переходим дальше.
2. Посылается широковещательный ARP-запрос. В этом запросе устанавливается MAC-адрес FF-FF-FF-FF-FF и указывается IP-адрес нужной машины. Если какая-нибудь машина в сети знает о существовании этого IP-адреса и знает его MAC-адрес, то она возвращает ответ с MAC-адресом нужной машины. Полученный адрес помещается в кэш.
3. Если и после этого не найден адрес, то пакет отправляется в шлюз.
4. Если IP-адрес найден в локальной сети, то компьютер получает реальный MAC-адрес. Если нет, то запрос отправляется маршрутизатору, который ищет MAC-адрес в удаленной сети. Когда он найдет MAC-адрес, то возвращает компьютеру не требуемый, а свой MAC-адрес. Таким образом, компьютер будет посылать пакеты на IP, а указывать MAC-адрес маршрутизатора, а тот будет переправлять пакет куда надо.

Когда компьютер получает MAC-адрес, то он сохраняет его в кэше. Адреса в кэше сохраняются в течение определенного времени (по умолчанию 10 минут). Если компьютер в течение 10 минут еще раз обращается по этому IP-адресу, то начнется отсчет с начала. Но такое бывает не во всех системах.

Для просмотра ARP-кэша в Windows можно воспользоваться командой `ARP` с параметром `-g` или `-a`. Но мы в этой главе напишем собственную небольшую утилиту, которая будет работать с этим интересным протоколом — ARP. Пример будет достаточно сложный, поэтому мы будем его изучать постепенно.

Запустите Delphi и создайте форму, похожую на ту, что изображена на рис. 7.4. В верхней части окна расположена панель с кнопками:

- **Обновить** — при нажатии этой кнопки мы будем перечитывать информацию о ARP-таблице из кэша;
- **Добавить** — чуть позже мы поместим в программу возможность добавления новых ARP-записей вручную. Эта функция нужна очень редко;
- **Удалить** — по этой команде мы будем удалять строки из кэша;
- **Очистить** — по этой команде мы будем полностью очищать ARP-кэш.

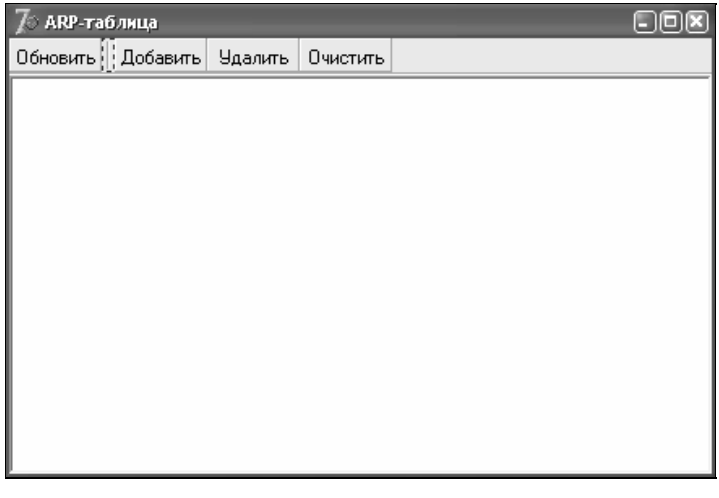


Рис. 7.4. Форма будущей программы

В центре окна находится компонент `TRichEdit`, который будет служить для отображения таблицы. Его задача только отображать, поэтому можно установить свойство `ReadOnly` равным `true`, чтобы не смущать пользователя лишними возможностями.

В раздел `uses` нужно добавить уже знакомые вам модули `IpRtrMib`, `IpHlpApi`, `iptypes` и `IpIfConst`. Без этих модулей программа не будет компилироваться, поэтому их присутствие обязательно.

Теперь приступим к программированию. Для начала напишем код, который будет выполняться после нажатия кнопки **Обновить** (листинг 7.1).

Листинг 7.1. Получение ARP-кэша

```
procedure TARPForm.UpdateButtonClick(Sender: TObject);
var
 Size: ULONG;
 I: Integer;
 NetTable: PMibIpNetTable;      // ARP-таблица
 NetRow: TMibIpNetRow;          // Строка ARP
 CurrentIndex: DWORD;           // Используется для показа заголовка
 IpAddrTable: PMibIpAddrTable;  // Таблица адресов
begin
 DisplayMemo.Clear;
 Size := 0;
 GetIpNetTable(nil, Size, True);
 NetTable := AllocMem(Size);
 try
  if GetIpNetTable(NetTable, Size, True) = NO_ERROR then
   begin
    // Получаем таблицу IP-адресов
    IpAddrTable := GetIpAddrTableWithAlloc;
     try
      // Запоминаем первый интерфейс
      CurrentIndex := NetTable^.table[0].dwIndex;
      DisplayMemo.SelAttributes.Color:=clTeal;
      DisplayMemo.SelAttributes.Style:=
         DisplayMemo.SelAttributes.Style+[fsBold];
      DisplayMemo.Lines.Add(Format('Интерфейс: %s на интерфейсе 0x%u',
         [IntfIndexToIpAddress(IpAddrTable, CurrentIndex),
          CurrentIndex]));
      DisplayMemo.SelAttributes.Color:=clTeal;
```

```
    DisplayMemo.SelAttributes.Style:=
       DisplayMemo.SelAttributes.Style+[fsBold];
    DisplayMemo.Lines.Add
       ('  IP-адрес           Физический адрес     Тип');
    // Для каждой записи ARP
    for I := 0 to NetTable^.dwNumEntries - 1 do
     begin
      NetRow := NetTable^.table[I];
      if CurrentIndex <> NetRow.dwIndex then
       begin
        // Определяем интерфейс
        CurrentIndex := NetRow.dwIndex;
        DisplayMemo.SelAttributes.Color:=clTeal;
        DisplayMemo.SelAttributes.Style:=
         DisplayMemo.SelAttributes.Style+[fsBold];
        DisplayMemo.Lines.Add(Format
           ('Интерфейс: %s на интерфейсе 0x%u',
         [IntfIndexToIpAddress(IpAddrTable, CurrentIndex),
          CurrentIndex]));
        DisplayMemo.SelAttributes.Color:=clTeal;
        DisplayMemo.SelAttributes.Style:=
          DisplayMemo.SelAttributes.Style+[fsBold];
        DisplayMemo.Lines.Add('IP-адрес      Физический адрес   Тип');
       end;
      // Отображаем строки
      DisplayMemo.Lines.Add(Format('%-20s %-30s %s',
         [IpAddrToString(NetRow.dwAddr),
         PhysAddrToString(NetRow.dwPhysAddrLen,
         TPhysAddrByteArray(NetRow.bPhysAddr)),
         ArpTypeToString(NetRow.dwType)]));
      DisplayMemo.Lines.Add('');
     end;
    finally
     FreeMem(IpAddrTable);
    end;
  end
```

```
  else
   begin
    // Если таблица не найдена, то выводим сообщение...
    DisplayMemo.SelAttributes.Color:=clRed;
    DisplayMemo.SelAttributes.Style:=
        DisplayMemo.SelAttributes.Style+[fsBold];
    DisplayMemo.Lines.Add('ARP-таблица не найдена.');
   end;
 finally
  FreeMem(NetTable);
 end;
end;
```

Самое интересное находится в начале процедуры и спрятано под вызовом функции `GetIpNetTable`. Она возвращает нам в первом параметре ARP-таблицу. Но когда она вызывается в первый раз, мы указываем `nil`. Если указать нулевое значение, то функция возвращает размер необходимой памяти для хранения ARP-таблицы. После получения размера ARP-таблицы мы выделяем память с помощью функции `AllocMem` для переменной `NetTable`.

После получения ARP-таблицы нужно узнать IP-адреса, которые принадлежат компьютеру. Возможно, что на компьютере установлены две сетевые карты, и тогда мы должны будем отсортировать записи из таблицы ARP по соответствующим сетевым интерфейсам. Интерфейс будет определяться по IP-адресу. IP-адреса мы узнаем с помощью функции `GetIpAddrTableWithAlloc`, которая выглядит следующим образом:

```
function GetIpAddrTableWithAlloc: PMibIpAddrTable;
var
 Size: ULONG;
begin
 Size := 0;
 GetIpAddrTable(nil, Size, True);
 Result := AllocMem(Size);
 if GetIpAddrTable(Result, Size, True) <> NO_ERROR then
  begin
   FreeMem(Result);
   Result := nil;
  end;
end;
```

Когда мы получим все необходимые данные, то будем готовы приступить к процессу вывода информации об ARP-таблице. В самом начале выводим информацию о первом найденном интерфейсе, для которого есть записи в кэше ARP:

```
CurrentIndex := NetTable^.table[0].dwIndex;
DisplayMemo.SelAttributes.Color:=clTeal;
DisplayMemo.SelAttributes.Style:=
            DisplayMemo.SelAttributes.Style+[fsBold];
DisplayMemo.Lines.Add(Format('Интерфейс: %s на интерфейсе 0x%u',
            [IntfIndexToIpAddress(IpAddrTable, CurrentIndex),
            CurrentIndex]));
DisplayMemo.SelAttributes.Color:=clTeal;
DisplayMemo.SelAttributes.Style:=
            DisplayMemo.SelAttributes.Style+[fsBold];
DisplayMemo.Lines.Add('  IP-адрес          Физический адрес     Тип');
```

После этого запускается цикл, в котором перебираем все записи кэша:

```
for I := 0 to NetTable^.dwNumEntries - 1 do
```

Внутри цикла первым делом получаем текущую строку ARP-записи:

```
NetRow := NetTable^.table[I];
```

После этого проверяем, изменилось ли значение свойства `dwIndex` текущей строки по сравнению с предыдущей. Если нет, то строка принадлежит тому же интерфейсу. Если там другое значение, то текущая ARP-строка относится к другому интерфейсу (не к тому, с которого мы начинали), поэтому нужно вывести информацию о следующем интерфейсе, найденном с помощью функции `GetIpAddrTableWithAlloc`.

```
if CurrentIndex <> NetRow.dwIndex then
```

Вот теперь уж точно можно выводить информацию о текущей ARP-записи на экран:

```
// Отображаем строки
DisplayMemo.Lines.Add(Format('  %-20s %-30s %s',
      [IpAddrToString(NetRow.dwAddr),
      PhysAddrToString(NetRow.dwPhysAddrLen,
      TPhysAddrByteArray(NetRow.bPhysAddr)),
      ArpTypeToString(NetRow.dwType)]));
DisplayMemo.Lines.Add('');
```

После вывода информации для всех строк освобождаем всю выделенную память под хранение таблиц с помощью функции `FreeMem`. Несмотря на то, что эта переменная локальная и должна уничтожаться автоматически, я явно уничтожаю переменную, чтобы уж точно быть уверенным в том, что из-за моей программы не происходит утечка памяти. И вам советую освобождать всю выделенную память самостоятельно и не надеяться на чужого дядю.

7.3. Изменение записей ARP-таблицы

Протокол ARP работает автономно, и все записи в нем появляются автоматически и без нашего участия. Записи, появляющиеся в ARP-таблице, называются динамическими.

Судя по спецификации протокола, у нас есть возможность самостоятельно создавать записи в таблице ARP, и такие записи называются статическими. Зачем это нужно? Динамические записи хранятся в таблице недолго, и если вы некоторое время не обращались по определенному адресу, то его запись уничтожается. Это связано с тем, что компьютеры могут иметь динамические IP-адреса даже в локальных сетях (выделение адресов по протоколу DHCP) и в любую минуту у компьютера с определенным MAC-адресом может измениться IP-адрес. Чтобы это несоответствие не создавало конфликтов в сети, динамические записи в ARP-таблице хранятся только определенное время.

Если в вашей сети используются только постоянные IP-адреса, и вы хотите, чтобы ARP-записи, соответствующие этим адресам, хранились все время, то можно добавить в ARP-таблицу статичные записи. В этом случае такие записи не будут удаляться, и при обращении к компьютерам не будет тратиться время на поиск MAC-адреса.

7.3.1. Добавление ARP-записей

Давайте организуем в нашей программе возможность добавления таких записей. Для этого сначала создадим новое окно, в котором пользователь должен будет вводить параметры новой записи. Внешний вид моего окна вы можете увидеть на рис. 7.5.

Теперь в обработчике события нажатия кнопки **Добавить** главной формы напишем следующий код:

```
procedure TARPForm.AddButtonClick(Sender: TObject);
begin
 InputIPForm.ShowModal;
```

```
 if InputIPForm.ModalResult<>mrOK then exit;
 SetArpEntry(InputIPForm.AddressEdit.Text, InputIPForm.MACEdit.Text);
end;
```

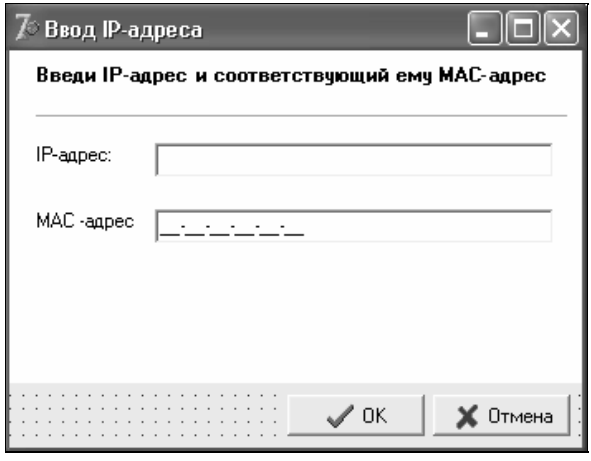


Рис. 7.5. Окно для ввода нового адреса

В первой строчке кода отображаем окно для ввода параметров новой записи. Во второй строке проверяется: если возвращаемое окном значение не равно `mrOK`, значит, была нажата кнопка **Отмена**, и запись добавлять не нужно, поэтому будет выполнен выход из процедуры.

Если была нажата кнопка **OK**, то выполнится третья строка, в которой вызывается процедура `SetArpEntry`. У этой процедуры два параметра:

- IP-адрес записи;
- MAC-адрес записи.

А теперь посмотрим, как выглядит сама процедура `SetArpEntry`. Ее нет среди API-функций, и мы должны ее написать сами. Для этого в разделе `private` добавьте для нее следующее описание:

```
private
  { Private declarations }
  procedure SetArpEntry(const InetAddr, EtherAddr: string);
```

Теперь нажмите комбинацию клавиш <Ctrl>+<Shift>+<C>, и Delphi создаст для этой процедуры заготовку, в которой нужно написать следующее (листинг 7.2).

Листинг 7.2. Добавление статичных записей

```
procedure TARPForm.SetArpEntry(const InetAddr, EtherAddr: string);
var
  Entry: TMibIpNetRow;
  IpAddrTable: PMibIpAddrTable;
begin
 // Обнуляю структуру
 FillChar(Entry, SizeOf(Entry), 0);
 // Назначаю IP-адрес
 Entry.dwAddr := StringToIpAddr(InetAddr);
 Assert(Entry.dwAddr <> DWORD(INADDR_NONE));
 // Назначаю физический адрес
 Entry.dwPhysAddrLen := 6;
 StringToPhysAddr(EtherAddr, TPhysAddrByteArray(Entry.bPhysAddr));
 Entry.dwType := MIB_IPNET_TYPE_STATIC;
 // Указываю интерфейс
 IpAddrTable := GetIpAddrTableWithAlloc;
 Assert(IpAddrTable <> nil);
 Entry.dwIndex := FirstNetworkAdapter(IpAddrTable);
 FreeMem(IpAddrTable);
 DisplayMemo.SelAttributes.Color:=clTeal;
 DisplayMemo.SelAttributes.Style:=
      DisplayMemo.SelAttributes.Style+[fsBold];
 // Добавляю запись, выводя результат работы
 DisplayMemo.Lines.Add(SysErrorMessage(SetIpNetEntry(Entry)));
end;
```

Процедура достаточно сложная, и чтобы ее понять, придется немного постараться. В первой строке заполняется нулями структура `Entry`, которая объявлена принадлежащей типу `TMibIpNetRow`, чтобы в ней случайно не оказалось никакого мусора. Для этого использована функция `FillChar`.

Во второй строке у структуры `Entry` заполняется свойство `dwAddr`, в котором указывается IP-адрес для добавляемой записи. Адрес IP у нас хранится

в строковой переменной `InetAddr` и его нужно преобразовать в числовой, что и делается с помощью функции `StringToIpAddr`, которая выглядит так:

```
function StringToIpAddr(const Addr: string): DWORD;
begin
  Result := inet_addr(PChar(Addr));
end;
```

Здесь для преобразования используется WinAPI-функция `inet_addr`. В принципе, можно было бы вызывать ее напрямую, но я сделал отдельную функцию, на случай если вы захотите добавить в нее возможность преобразования и символьных имен.

После преобразования происходит проверка с помощью функции `Assert` на правильность адреса. Если `Entry.dwAddr` не равен `INADDR_NONE`, то все нормально, иначе генерируется ошибка.

Дальше нужно указать физический адрес. Сначала указываем длину физического адреса `Entry.dwPhysAddrLen`, вписывая значение `6`. После этого присваиваем свойству `bPhysAddr` структуры `Entry` значение физического адреса с помощью функции `StringToPhysAddr`, которая одновременно переводит строковое представление MAC-адреса в нужный формат. У этой функции два параметра:

- строковое представление MAC-адреса;
- переменная, в которую нужно записать приведенный адрес.

Саму функцию нужно еще написать. Я не стал ее делать частью объекта окна, поэтому где-нибудь выше нашего обработчика напишите код из листинга 7.3.

Листинг 7.3. Перевод строки в физический адрес

```
procedure StringToPhysAddr(PhysAddrString: string;
         var PhysAddr: TPhysAddrByteArray);
var
  C: Char;
  I, V: Integer;
begin
  Assert(Length(PhysAddrString) = 17);
  Assert(
    (PhysAddrString[3] = '-') and
    (PhysAddrString[6] = '-') and
    (PhysAddrString[9] = '-') and
```

```
    (PhysAddrString[12] = '-') and
    (PhysAddrString[15] = '-'));
  PhysAddrString := UpperCase(PhysAddrString);
  for I := 0 to 5 do
  begin
    C := PhysAddrString[I * 3];
    V := CharHex(C) shl 4;
    C := PhysAddrString[(I * 3) + 1];
    V := V + CharHex(C);
    PhysAddr[I] := V;
  end;
end;
```

Здесь сначала проверяется обязательное присутствие знака "-" в позициях 3, 6, 9, 12, 15. После этого строка преобразовывается к верхнему регистру и запускается цикл преобразования.

Теперь, когда мы указали длину физического адреса и сам адрес, нужно указать, что он статичный. Для этого в свойство `dwType` структуры `Entry` задаем константу `MIB_IPNET_TYPE_STATIC`.

Следующим этапом нужно указать интерфейс, для которого мы создаем запись. В вашем компьютере может быть несколько сетевых карт, и компьютер должен знать, для какой из них будет действовать ARP-запись. Все это делается в следующем коде:

```
// Указываем интерфейс
IpAddrTable := GetIpAddrTableWithAlloc;
Assert(IpAddrTable <> nil);
Entry.dwIndex := FirstNetworkAdapter(IpAddrTable);
FreeMem(IpAddrTable);
```

В первой строке мы получаем таблицу IP-адресов с помощью уже знакомой вам функции `GetIpAddrTableWithAlloc`. Если полученная таблица равна нулю (эту проверку делает вторая строка), то произойдет ошибка.

В третьей строке мы получаем первый адаптер (IP-адрес) из таблицы с помощью функции `FirstNetworkAdapter` и присваиваем его свойству `dwIndex` структуры `Entry`. Функция `FirstNetworkAdapter` выглядит следующим образом:

```
function FirstNetworkAdapter(IpAddrTable: PMibIpAddrTable): Integer;
var
  I: Integer;
  IfInfo: TMibIfRow;
```

```
begin
  Result := -1;
  for I := 0 to IpAddrTable^.dwNumEntries - 1 do
  begin
    {$R-}IfInfo.dwIndex := IpAddrTable^.table[I].dwIndex;{$R+}
    if GetIfEntry(@IfInfo) = NO_ERROR then
    begin
      if IfInfo.dwType in [MIB_IF_TYPE_ETHERNET, MIB_IF_TYPE_TOKENRING]
       then
      begin
        Result := IfInfo.dwIndex;
        Break;
      end;
    end;
  end;
end;
```

Как видите, она не является частью нашего объекта-окна, поэтому ее нужно дописать где-нибудь выше кода, который ее использует.

В примере все записи будут всегда создаваться для первого интерфейса из таблицы, но вы можете улучшить код, чтобы записи можно было создавать для любого интерфейса. Я даю вам только основу, чтобы вы потом могли создать именно то, что вам нужно, а заранее предугадать потребности всех читателей я не в силах. Но если вы захотите добавить возможность выбора интерфейса, то вам нужно изменить код на такой:

```
if Интерфейс <> '' then
 Entry.dwIndex := StrToInt(Интерфейс)
else
 begin
  IpAddrTable := GetIpAddrTableWithAlloc;
  Assert(IpAddrTable <> nil);
  Entry.dwIndex := FirstNetworkAdapter(IpAddrTable);
  FreeMem(IpAddrTable);
 end;
```

После получения первого адаптера таблицу адресов можно удалять, что и делается с помощью вызова функции `FreeMem(IpAddrTable)`.

Теперь структура `Entry` окончательно готова, и для добавления записи достаточно вызвать API-функцию `SetIpNetEntry`, которая сделает все необходимое. Эта функция вернет нам результат выполнения команды, который потом преобразовывается в строковое представление с помощью `SysErrorMessage`. Это строковое представление ошибки добавляется в качестве строки компонента `DisplayMemo`.

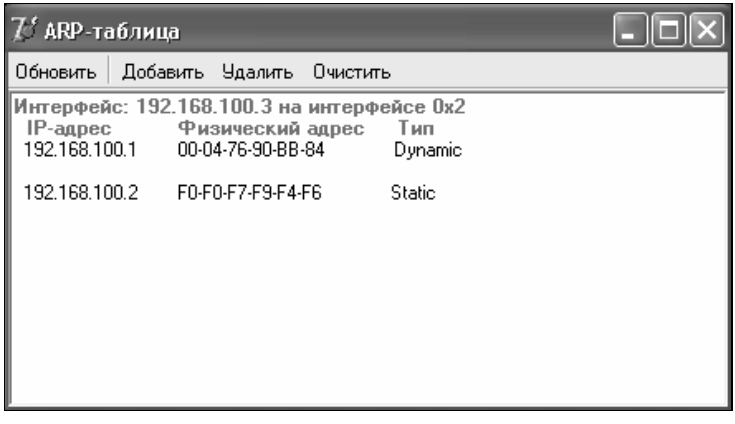


Рис. 7.6. Результат работы программы

На рис. 7.6 вы можете увидеть результат работы примера. В моей ARP-таблице две записи:

- первая запись указывает на MAC-адрес компьютера `192.168.100.1` и является динамической (`Dynamic`), о чем говорит последняя колонка;
- вторая запись указывает на компьютер `192.168.100.2` и является статической (`Static`), потому что я создал ее вручную.

7.3.2. Удаление ARP-записей

Теперь добавим в нашу программу возможность удаления записей ARP. Для этого в обработчике нажатия кнопки **Удалить** пишем следующий код:

```
procedure TARPForm.DeleteButtonClick(Sender: TObject);
begin
 InputIPForm.Label4.Visible:=false;
 InputIPForm.Label2.Visible:=false;
 InputIPForm.MACEdit.Visible:=false;
```

```
 InputIPForm.ShowModal;
 InputIPForm.Label4.Visible:=true;
 InputIPForm.Label2.Visible:=true;
 InputIPForm.MACEdit.Visible:=true;
 if InputIPForm.ModalResult<>mrOK then exit;
 DeleteArpEntry(InputIPForm.AddressEdit.Text,'');
end;
```

Для указания удаляемой записи используется то же окно, что и для добавления, только прежде чем его отобразить, делаются невидимыми все компоненты, которые относятся к MAC-адресу. Удаляемую запись мы будем определять по IP-адресу, поэтому мне достаточно только одного поля ввода для него (рис. 7.7).

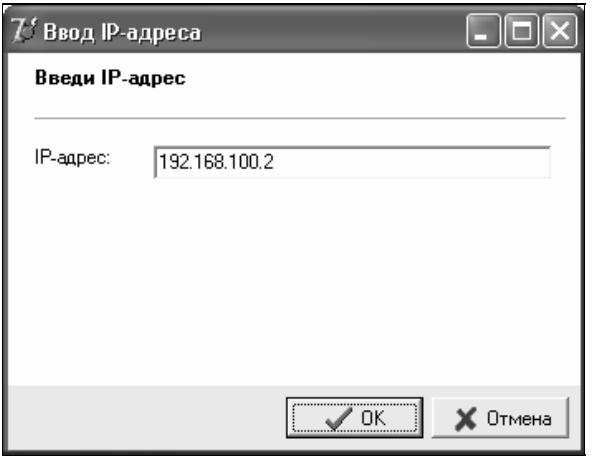


Рис. 7.7. Окно удаления ARP-записи

После отображения окна снова делаем видимыми все спрятанные компоненты, чтобы, если пользователь сразу же захочет добавить новую запись, ему были доступны все необходимые поля. На первый взгляд, этот процесс неудобен и требует лишних строчек кода, а с другой стороны, экономится память и размер программы за счет того, что мы не создаем лишних окон. А главное, наша программа не тратит время при загрузке на создание лишних окон. Этот трюк с экономией очень прост и эффективен, поэтому старайтесь его использовать почаще, чтобы сэкономить на ресурсах и повысить скорость своих приложений.

Для удаления ARP-записи мы вызываем процедуру `DeleteArpEntry`. У нее два параметра:

- IP-адрес, запись для которого нужно удалить;
- этот параметр не используется, но вы можете внести в него возможность введения номера интерфейса, для которого удаляется запись. Это то, что я опустил в процедуре создания новой ARP-строки.

Чтобы создать процедуру `DeleteArpEntry`, в разделе `private` добавьте ее описание и нажмите комбинацию клавиш <Ctrl>+<Shift>+<C>. Описание должно выглядеть так:

```
procedure DeleteArpEntry(const Host, Intf: string);
```

В полученной заготовке напишите код из листинга 7.4.

Листинг 7.4. Удаление записи в таблице ARP

```
procedure TARPForm.DeleteArpEntry(const Host, Intf: string);
var
  Entry: TMibIpNetRow;
  HostAddr, IntfAddr: DWORD;
  Size: ULONG;
  Adapters, Adapter: PIpAdapterInfo;
begin
  FillChar(Entry, SizeOf(Entry), 0);
  HostAddr := 0;
  if Host <> '*' then
  begin
    HostAddr := inet_addr(PChar(Host));
    if HostAddr = DWORD(INADDR_NONE) then Exit;
  end;
  if Intf <> '' then
  begin
    IntfAddr := inet_addr(PChar(Intf));
    if IntfAddr = DWORD(INADDR_NONE) then Exit;
  end;
  // Удалить только один адрес
  if (Host <> '*') and (Intf <> '') then
```

```
begin
  Entry.dwIndex := IpAddressToAdapterIndex(Intf);
  Entry.dwAddr := HostAddr;
  DisplayMemo.SelAttributes.Color:=clTeal;
  DisplayMemo.SelAttributes.Style:=
       DisplayMemo.SelAttributes.Style+[fsBold];
  if DeleteIpNetEntry(Entry) = NO_ERROR then
    DisplayMemo.Lines.Add('Удаление прошло успешно')
  else
    DisplayMemo.Lines.Add('Ошибка');
  Exit;
end;
// Удалить все адреса
if (Host = '*') and (Intf <> '') then
begin
  FlushIpNetTable(IpAddressToAdapterIndex(Intf));
  Exit;
end;
Size := 0;
if GetAdaptersInfo(nil, Size) <> ERROR_BUFFER_OVERFLOW then Exit;
Adapters := AllocMem(Size);
try
  if GetAdaptersInfo(Adapters, Size) = NO_ERROR then
  begin
    Adapter := Adapters;
    while Adapter <> nil do
    begin
      // Удалить все адреса из всех интерфейсов
      if (Host = '*') and (Intf = '') then
      begin
        FlushIpNetTable(Adapter.Index);
      end;
      // Удалить указанный адрес из всех интерфейсов
      if (Host <> '*') and (Intf = '') then
```

```
        begin
          FillChar(Entry, SizeOf(Entry), 0);
          Entry.dwIndex := Adapter.Index;
          Entry.dwAddr := HostAddr;
          DeleteIpNetEntry(Entry);
        end;
        Adapter := Adapter^.Next;
      end;
    end;
  finally
    FreeMem(Adapters);
  end;
 DisplayMemo.SelAttributes.Color:=clTeal;
 DisplayMemo.SelAttributes.Style:=
    DisplayMemo.SelAttributes.Style+[fsBold];
 DisplayMemo.Lines.Add('Удаление прошло успешно')
end;
```

Процедура получилась достаточно большая, зато универсальная, и на все случаи жизни. Попробуйте с ней подробно разобраться и понять, что она умеет делать. У вас должны быть уже все необходимые знания.

У функции два параметра:

- адрес хоста, который надо удалить;
- интерфейс, из которого нужно удалить запись.

Обратите внимание, если в качестве имени хоста будет указан знак звездочки (*), то будут удалены все записи для указанного интерфейса. Если интерфейс не указан, то удаляются все записи из всех интерфейсов.

Теперь напишем код обработчика события нажатия кнопки **Очистить**:

```
procedure TARPForm.ClearButtonClick(Sender: TObject);
begin
 DeleteArpEntry('*', '');
end;
```

Здесь вызывается функция удаления записи `DeleteArpEntry`, у которой первый параметр равен звездочке, а второй пустой, чтобы очистить все ARP-записи.

Примечание

На компакт-диске в каталоге \Примеры\Глава 7\ARP\ вы можете найти пример данной программы.

7.4. Работа с сетевыми ресурсами

В *разд. 4.4* мы уже познакомились с работой сетевых ресурсов, когда написали соответствующий сканер. В этой части я опишу этот процесс более подробно. Мы напишем программу, которая будет сканировать всю локальную сеть на предмет открытых ресурсов, как это делает Сетевое окружение. В дальнейшем мы добавим в нее возможность подключения сетевых дисков.

Итак, наша программа вытаскивает сведения о структуре сети. На рис. 7.8 вы можете увидеть главную форму нашей будущей программы.

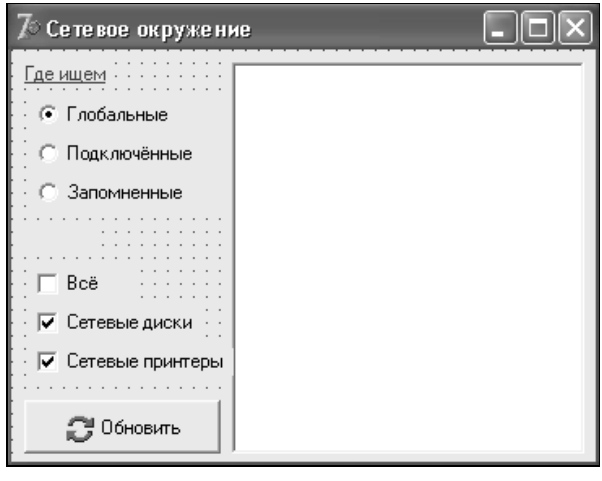


Рис. 7.8. Главная форма будущей программы

На форме расположены три переключателя `TRadioButton`, которые управляют работой программы. В зависимости от их установки будет изменяться список искомых ресурсов. Помимо этого есть три компонента `TCheckBox`, с помощью которых можно выбрать что искать: все, сетевые диски или принтеры. Большую часть окна занимает компонент `TTreeView`, в котором мы будем отображать дерево найденных сетевых устройств.

Процесс работы программы начинается с нажатия кнопки **Обновить**. Как только пользователь щелкнет на ней, мы должны просканировать всю сеть

на наличие открытых ресурсов. В обработчике события `OnClick` кнопки **Обновить** пишем следующий код (листинг 7.5).

Листинг 7.5. Поиск ресурсов в свободном доступе

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ResScope,
  ResType:DWORD;
begin
 TreeTreeRes.Items.Clear;
 if RadioButton2.Checked then
  ResScope := RESOURCE_GLOBALNET
 else
  if RadioButton1.Checked then
   ResScope := RESOURCE_CONNECTED
  else
   ResScope := RESOURCE_REMEMBERED;
 ResType := 0;
 if CBTypeAny.Checked then
  ResType := ResType or RESOURCETYPE_ANY;
 if CBTypeDisk.Checked then
  ResType := ResType or RESOURCETYPE_DISK;
 if CBTypePrint.Checked then
  ResType := ResType or RESOURCETYPE_PRINT;
 EnumNet(TreeTreeRes.Items.Add(nil, 'Сетевое окружение'),
    ResScope, ResType, nil);
end;
```

В самом начале нужно очистить компонент `TtreeView` от текущих данных, потому что сейчас мы будем заполнять его новыми данными.

После этого происходит проверка, какой компонент `RadioButton` выбран. Если это переключатель **Глобальные**, то в переменную `ResScope` заносим константу `RESOURCE_GLOBALNET` для поиска глобальных сетевых устройств. Если выбран пункт **Подключенные**, то в эту переменную заносится константа `RESOURCE_CONNECTED`. Если не то и не другое, то указывается константа `RESOURCE_REMEMBERED` для поиска запомненных в кэше ресурсов.

Следующим этапом мы проверяем, что нужно искать. Если установлен флажок **Все**, то в переменную `ResType` заносится константа `RESOURCETYPE_ANY`. Если выбран пункт **Сетевые диски**, то добавляется константа `RESOURCETYPE_DISK`. Если установлен флажок **Сетевые принтеры**, то нужно добавить `RESOURCETYPE_PRINT`. Здесь переменная `ResType` заполняется методом добавления выбранных констант. Например, если пользователь выбрал поиск сетевых принтеров и сетевых дисков, то мы должны занести в эту переменную две константы: `RESOURCETYPE_DISK` и `RESOURCETYPE_PRINT`.

Заполнив необходимые переменные начальными значениями, вызываем процедуру `EnumNet`, которая выглядит следующим образом:

```
procedure TForm1.EnumNet(const ParentNode: TTreeNode;
   ResScope, ResType: DWORD;
   const NetContainerToOpen: PNetResource);
var
  hNetEnum: THandle;
begin
 hNetEnum := OpenEnum(NetContainerToOpen, ResScope,
       ResType, RESOURCEUSAGE_CONNECTABLE or RESOURCEUSAGE_CONTAINER);
 if (hNetEnum = 0) then exit;
 EnumResources(ParentNode, ResScope, ResType,
       RESOURCEUSAGE_CONNECTABLE or RESOURCEUSAGE_CONTAINER, hNetEnum);
 if (NO_ERROR <> WNetCloseEnum(hNetEnum)) then
  ShowMessage('WNetCloseEnum Error');
end;
```

На входе процедура получает следующие параметры:

- `ParentNode` — родительский объект в дереве `TreeView`, к которому будут добавляться имена найденных ресурсов;
- `ResScope` — указание, где надо будет искать ресурсы (глобальные, подключенные или запомненные);
- `ResType` — тип ресурсов, которые надо искать (все, принтеры, диски);
- `NetContainerToOpen` — переменная, используемая при перечислении.

В самой первой строке мы вызываем функцию `OpenEnum`, которая объявлена в разделе `public` следующим образом:

```
public
  procedure EnumNet(const ParentNode: TTreeNode;
              ResScope, ResType: DWORD;
              const NetContainerToOpen: PNetResource);
```

```
  function OpenEnum(const NetContainerToOpen: PNetResource;
              ResScope, ResType, ResUsage: DWORD): THandle;
  function EnumResources(const ParentNode: TTreeNode;
              ResScope, ResType, ResUsage: DWORD;
              hNetEnum: THandle): UINT;:
```

Здесь описаны три функции, хотя мы увидели пока только одну. Вы тоже должны объявить все три функции, а их содержимое мы будем писать постепенно. Нажмите комбинацию клавиш <Ctrl>+<Shift>+<C>, чтобы Delphi создала заготовку для всех трех функций. Теперь нужно найти функцию `OpenEnum`, с которой мы уже столкнулись, и написать в ней следующее:

```
function TForm1.OpenEnum(const NetContainerToOpen: PNetResource;
             ResScope, ResType, ResUsage: DWORD): THandle;
var
 hNetEnum: THandle;
begin
 Result := 0;
 if (NO_ERROR <> WNetOpenEnum(ResScope, ResType,
         RESOURCEUSAGE_CONNECTABLE or RESOURCEUSAGE_CONTAINER,
         NetContainerToOpen, hNetEnum)) then
  ShowMessage('Ошибочка вышла')
 else
  Result := hNetEnum;
end;
```

Весь смысл этой функции — запустить перечисление ресурсов с помощью функции `WNetOpenEnum`. После этого она проверяет результат выполнения запущенного перечисления и возвращает его. Функция `WNetOpenEnum` выглядит следующим образом:

```
function WNetOpenEnum(
dwScope, dwType, dwUsage: DWORD;
lpNetResource: PNetResource;
var lphEnum: THandle
): DWORD; stdcall;
```

Эта функция открывает перечисление сетевых устройств в локальной сети.

Рассмотрим передаваемые ей параметры.

- `dwScope` — какие ресурсы будут включаться в перечисление. Возможны комбинации следующих значений:
  - `RESOURCE_GLOBALNET` — все ресурсы сети;
  - `RESOURCE_CONNECTED` — подключенные;
  - `RESOURCE_REMEMBERED` — запомненные.
- `dwType` — тип ресурсов, включаемых в перечисление. Возможны комбинации следующих значений:
  - `RESOURCETYPE_ANY` — все ресурсы сети;
  - `RESOURCETYPE_DISK` — сетевые диски;
  - `RESOURCETYPE_PRINT` — сетевые принтеры.
- `dwUsage` — тип использования ресурсов, включаемых в перечисление. Возможны значения:
  - `0` — все ресурсы сети;
  - `RESOURCEUSAGE_CONNECTABLE` — подключаемые;
  - `RESOURCEUSAGE_CONTAINER` — контейнерные.
- `lpNetResource` — указатель на структуру `NETRESOURCE`. Если этот параметр равен нулю, то перечисление начнется с самой верхней ступени иерархии сетевых ресурсов. Ноль ставится для того, чтобы получить самый первый ресурс. Замем в качестве этого параметра передается указатель на уже найденный ресурс. Тогда перечисление начнется с найденного и продолжится дальше. Так повторяется, пока не найдутся все ресурсы.
- `lphEnum` — это указатель, который понадобится в функции `WnetEnumResource`.

Теперь нужно рассмотреть структуру `NETRESOURCE`. В Delphi есть три разновидности этой функции: `NETRESOURCE`, `NETRESOURCEA` и `NETRESOURCEW`. Отличаются они последней буквой в названии и типом строк.

```
_NETRESOURCEA = packed record
  dwScope: DWORD;
  dwType: DWORD;
  dwDisplayType: DWORD;
  dwUsage: DWORD;
  lpLocalName: PAnsiChar;
  lpRemoteName: PAnsiChar;
  lpComment: PAnsiChar;
  lpProvider: PAnsiChar;
end;
```

Что такое `dwScope`, `dwType` и `dwUsage` вы уже знаете, поэтому мы их опустим. А вот остальные — рассмотрим:

- `dwDisplayType` — как должен отображаться ресурс:
  - `RESOURCEDISPLAYTYPE_DOMAIN` — это домен;
  - `RESOURCEDISPLAYTYPE_GENERIC` — нет значения;
  - `RESOURCEDISPLAYTYPE_SERVER` — сервер;
  - `RESOURCEDISPLAYTYPE_SHARE` — разделяемый ресурс;
- `lpLocalName` — локальное имя;
- `lpRemoteName` — удаленное имя;
- `lpComment` — комментарий;
- `lpProvider` — хозяин ресурса. Параметр может быть равен `0`, если хозяин неизвестен.

После открытия перечисления с помощью функции `OpenEnum` вызывается функция `EnumResources`. Ее объявление мы уже написали, и заготовка у нас уже есть. Осталось только написать код, который выглядит так, как представлено в листинге 7.6.

Листинг 7.6. Перечисление ресурсов

```
function TForm1.EnumResources(const ParentNode: TTreeNode;
                              ResScope, ResType, ResUsage: DWORD;
                              hNetEnum: THandle): UINT;
 function ShowResource(const ParentNode: TTreeNode;
    Res: TNetResource): TTreeNode;
 var
  Str:String;
  index:Integer;
 begin
  Result:=ParentNode;
  if Res.lpRemoteName=nil then exit;
  Str:=string(Res.lpRemoteName);
  index:=Pos('\',Str);
  while index>0 do
   begin
    Str:=Copy(Str,index+1,Length(Str));
```

```
    index:=Pos('\',Str);
   end;
  Result := TreeTreeRes.Items.AddChild(ParentNode, Str);
 end;
var
 ResourceBuffer: array[1..2000] of TNetResource;
 i, ResourceBuf, EntriesToGet: DWORD;
 NewNode: TTreeNode;
begin
 Result := 0;
 while TRUE do
  begin
   ResourceBuf := sizeof(ResourceBuffer);
   EntriesToGet := 2000;
   if (NO_ERROR <> WNetEnumResource(hNetEnum, EntriesToGet,
         @ResourceBuffer, ResourceBuf)) then
    begin
     case GetLastError() of
       NO_ERROR: break;
       ERROR_NO_MORE_ITEMS: exit;
     else
      ShowMessage('Ошибочка вышла');
      Result := 1;
      exit;
     end;
    end;
   for i := 1 to EntriesToGet do
    begin
     NewNode := ShowResource(ParentNode, ResourceBuffer[i]);
     if (ResourceBuffer[i].dwUsage and RESOURCEUSAGE_CONTAINER) <> 0 then
      EnumNet(NewNode, ResScope, ResType, @ResourceBuffer[i]);
    end;
  end;
end;
```

Самая интересная здесь функция — это `WNetEnumResource`. Она перечисляет ресурсы сетевых объектов. В Delphi она описана следующим образом:

```
function WNetEnumResource(
         hEnum: THandle;
         var lpcCount: DWORD;
         lpBuffer: Pointer;
         var lpBufferSize: DWORD
         ): DWORD; stdcall;
```

У нее такие параметры:

- `hEnum` — указатель на возвращенное функцией `WNetOpenEnum` значение;
- `lpcCount` — максимальное количество возвращаемых значений. Не стесняйтесь, ставьте `2000`. Если вы укажете `0xFFFFFFFF`, то будут перечислены все ресурсы. После выполнения функция поместит сюда фактическое число найденных ресурсов;
- `lpBuffer` — указатель на буфер, в который будет помещен результат;
- `lpBufferSize` — размер буфера.

После вызова этой функции найденные, ресурсы добавляются с помощью:

```
NewNode := ShowResource(ParentNode, ResourceBuffer[i]);
```

Функция `ShowResource` как бы входит в состав функции `EnumResources` и выглядит вот так:

```
function ShowResource(const ParentNode: TTreeNode;
  Res: TNetResource): TTreeNode;
 var
  Str:String;
  index:Integer;
 begin
  Result:=ParentNode;
  if Res.lpRemoteName=nil then exit;
  Str:=string(Res.lpRemoteName);
  index:=Pos('\',Str);
  while index>0 do
   begin
    Str:=Copy(Str,index+1,Length(Str));
    index:=Pos('\',Str);
   end;
```

```
  Result := TreeTreeRes.Items.AddChild(ParentNode, Str);
end;
```

После перечисления всех ресурсов вызывается функция `WnetCloseEnum`, которая закрывает перечисление ресурсов.

Попробуйте скомпилировать программу и запустить ее. Посмотрите, как она ищет ресурсы, если выбирать различные параметры поиска. На рис. 7.9 вы можете увидеть результат выполнения программы в моей локальной сети.

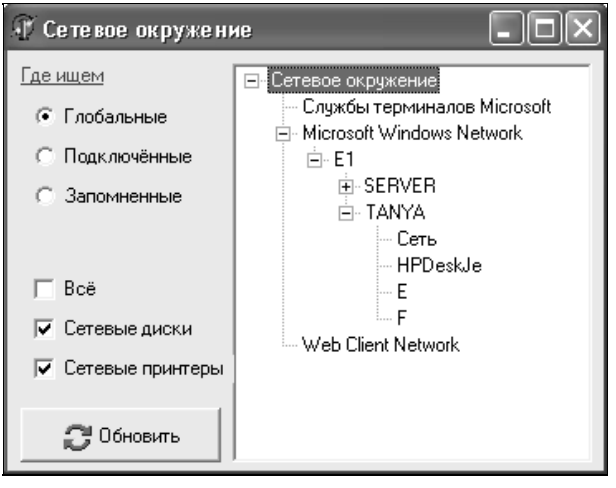


Рис. 7.9. Результат работы программы

Если во время выполнения появится окно с ошибкой, то это не значит, что программа работает неверно. Возможно, в перечисление попал компьютер, который требует авторизации и без пароля не пускает. В этом случае генерируется ошибка перечисления, но программа продолжает искать другие компьютеры в локальной сети.

Примечание

На компакт-диске в каталоге \Примеры\Глава 7\Net Resource1\ вы можете найти пример этой программы и цветные версии рисунков из этого раздела.

Теперь добавим в нашу программу возможность подключения и отключения сетевых дисков, для этого на нашу форму поместим еще две кнопки: **Присоединиться** и **Отсоединиться**. На рис. 7.10 можно увидеть внешний вид обновленной программы.

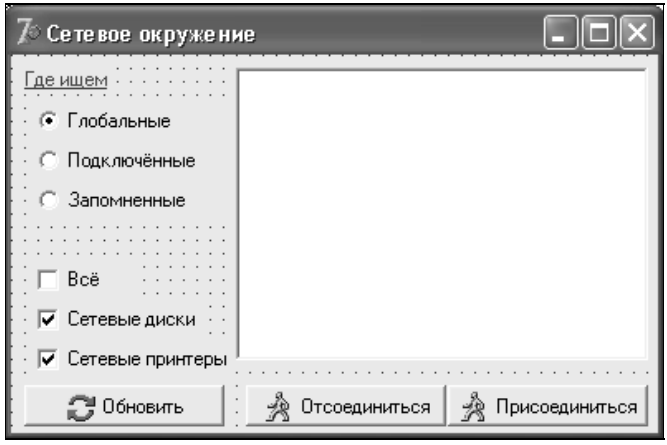


Рис. 7.10. Обновленная форма программы

После нажатия кнопки **Присоединиться** мы будем выводить стандартное окно подключения сетевого диска. Создайте обработчик события `OnClick` для этой кнопки и напишите в нем следующий код:

```
procedure TForm1.ConnectBtnClick(Sender: TObject);
begin
 WNetConnectionDialog(Handle,RESOURCETYPE_DISK);
end;
```

Здесь используется функция `WNetConnectionDialog`, которая выглядит в Delphi следующим образом:

```
function WNetConnectionDialog(
         hwnd: HWND;
         dwType: DWORD):
         DWORD; stdcall;
```

В качестве первого параметра передается указатель на окно владельца. Второй параметр — это флаги, которые могут принимать значения:

- `RESOURCETYPE_DISK` — отображать в выпадающем списке диалога сетевые диски;
- `RESOURCETYPE_PRINT` — отображать в выпадающем списке диалога сетевые принтеры;
- `RESOURCETYPE_ANY` — отображать в выпадающем списке диалога все, что попадется под руку.

Вид стандартного окна присоединения дисков в Windows XP вы можете увидеть на рис. 7.11.

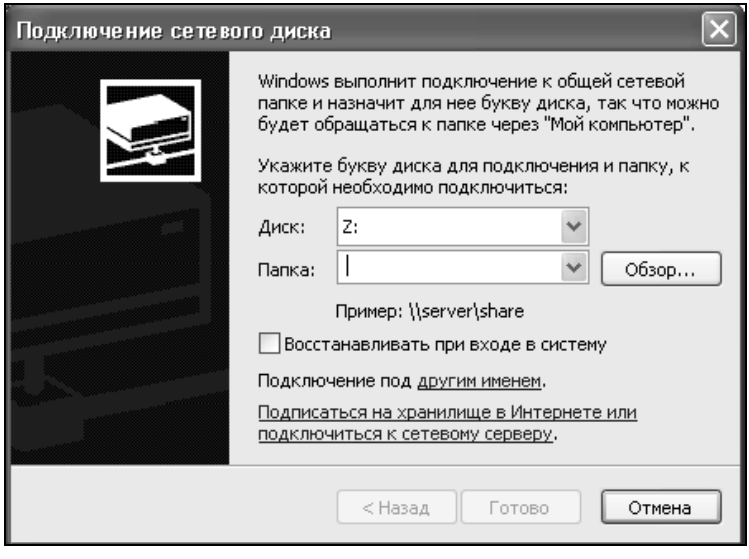


Рис. 7.11. Стандартное окно подключения дисков

В обработчике события нажатия кнопки **Отсоединиться** пишем следующий код:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
 WNetDisconnectDialog(Handle, RESOURCETYPE_DISK);
end;
```

Здесь мы используем функцию `WnetDisconnectDialog`, которая выводит на экран стандартное окно отключения дисков. В Delphi эта функция объявлена следующим образом:

```
function WNetDisconnectDialog (
         hwnd: HWND;
         dwType: DWORD):
         DWORD; stdcall;
```

Здесь параметры те же, что и у функции подключения сетевых дисков. Такое окно из Windows XP вы можете увидеть на рис. 7.12.

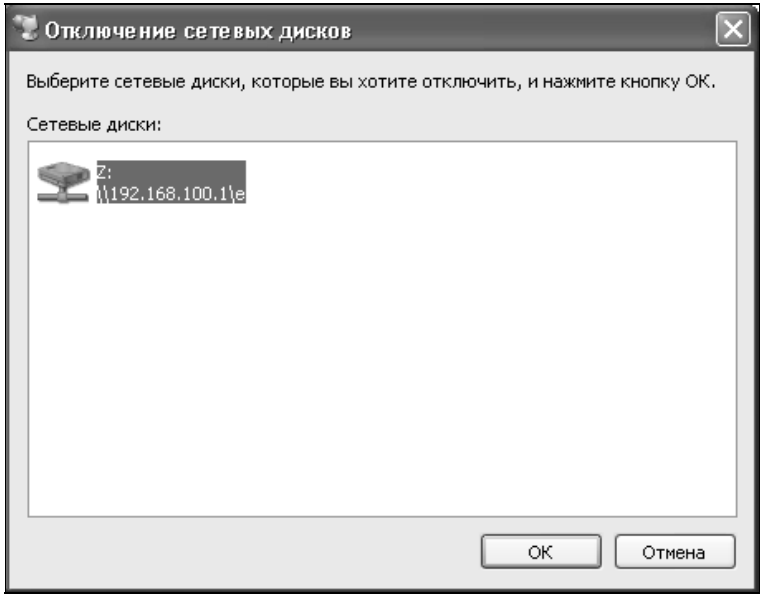


Рис. 7.12. Стандартное окно отключения сетевых дисков

Примечание

На компакт-диске в каталоге \Примеры\Глава 7\Net Resource2\ вы можете увидеть модифицированный пример из этого раздела и цветные рисунки, относящиеся к нему.

7.5. Быстрая проверка состояний портов: вариант 1

Тот, кто знаком с сетью не первый день, должен знать утилиту TCPView великого Марка Русиновича и как она помогает хакерам в нашем нелегком труде. На рис. 7.13 можно увидеть главное окно этой программы, отображающее состояние портов на моем компьютере. Для тех, кто не слышал о TCPView и не знает, что такое NET Stat, поясню: эти утилиты удобны тем, что позволяют быстро определить состояния портов на локальном компьютере. Используемые функции возвращают более подробную информацию, чем это можно получить с помощью сканера портов, но обладают одним серьезным недостатком — нельзя просмотреть таблицу портов для удаленного компьютера.

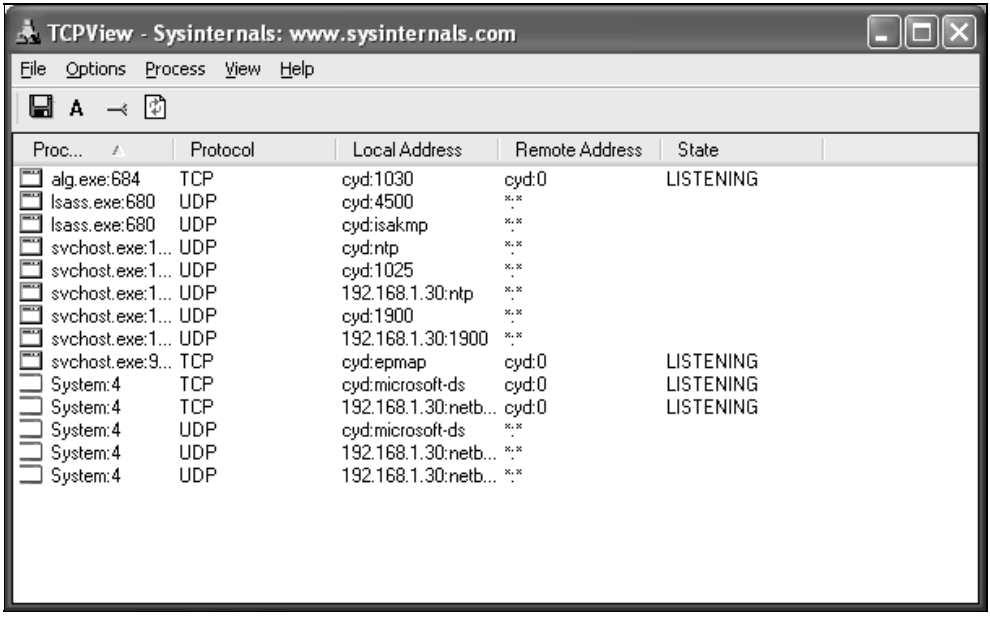


Рис. 7.13. Отображение состояния портов с помощью TCPView

Сегодня мы начнем писать собственную утилиту подобного плана, и вы увидите, что ничего великого в ней нет. Все очень просто, если знать необходимые функции и уметь ими пользоваться. Пять минут работы за клавиатурой, и у нас будет готов собственный TCPView.

Существуют два набора функций для получения состояния TCP/UDP-портов. Первый поддерживается всеми ОС, начиная с Windows 95, а второй набор появился, если я не ошибаюсь, в Windows 2000 или даже в Windows XP, т. е. достаточно свеженький. Новые функции намного лучше и позволяют получить больше информации, в том числе и имя процесса, который открыл порт, но они будут работать только в Windows XP (за Windows 2000 я не ручаюсь). Старый вариант немного проще, но работает везде.

Оба варианта функций реализованы в динамической библиотеке iphlpapi.dll. Так как мы хотим, чтобы наша программа работала в любой версии ОС Windows, то рассмотрим оба набора и создадим универсальную утилиту.

В стандартной библиотеке Delphi ни один из наборов функций не описан, поэтому придется писать собственный заголовочный файл, чтобы Delphi знала, где найти необходимые функции. Мы напишем такой заголовочный файл и сделаем это правильно — функции будут подключены не статически, а динамически, т. е. библиотека iphlpapi.dll будет загружаться с помощью `LoadLibrary`, а потом мы будем получать адреса необходимых функций. Это

очень важно при использовании нового, расширенного набора функций, ведь если связать заголовочный файл статически, то при старте программы функция автоматически будет искать связь с библиотекой, и если пользователь запустит программу в Windows 95, то произойдет ошибка, ведь новые функции не будут найдены.

Сейчас мы рассмотрим только старый набор функций, а в следующий раз двинемся дальше. Итак, усаживаемся поудобнее — мы погружаемся в код.

Для получения таблицы состояния TCP-портов необходимо использовать функцию `GetTcpTable`. Она выглядит следующим образом:

```
GetTcpTable: function(
 pTcpTable: PMIB_TCPTABLE;
 var pdwSize: DWORD;
 bOrder: BOOL
): DWORD; stdcall;
```

Здесь мы объявляем переменную типа "функция" с именем `GetTcpTable`. Чуть позже, когда мы напишем код загрузки библиотеки, в эту переменную будет записан адрес системной функции `GetTcpTable`. В качестве параметров она получает три значения:

- указатель на структуру `MIB_TCPTABLE`. Это и есть таблица состояний, которую мы получим на выходе;
- размер выделенной памяти, для хранения таблицы. О том, сколько памяти выделять, мы поговорим, когда будем рассматривать реальный пример;
- булево значение, которое определяет, нужно ли сортировать таблицу.

Самое интересное здесь — это, конечно же, первый параметр, где мы видим структуру `MIB_TCPTABLE`. Эта структура выглядит следующим образом:

```
MIB_TCPTABLE = record
 dwNumEntries: DWORD;
 table: array [0..0] of MIB_TCPROW;
end;
```

Уже по именам можно понять, для чего нужны поля структуры. Первый — это количество записей в таблице состояний, а второй параметр — это массив структур типа `MIB_TCPROW`, где содержатся сами записи.

Структура `MIB_TCPROW` выглядит вот так:

```
MIB_TCPROW = record
 dwState: DWORD;
 dwLocalAddr: DWORD;
```

```
 dwLocalPort: DWORD;
 dwRemoteAddr: DWORD;
 dwRemotePort: DWORD;
end;
```

Вот тут вся необходимая нам информация, которую и отображает TCPView:

- `dwState` — состояние;
- `dwLocalAddr` — локальный адрес;
- `dwLocalPort` — локальный порт;
- `dwRemoteAddr` — адрес удаленной машины, которая запросила соединение;
- `dwRemotePort` — удаленный порт.

Идентичная функция есть для получения состояния UDP-портов — `GetUdpTable`. Мы ее не будем рассматривать так подробно, потому что все идентично. Разница только в том, что структура состояния содержит лишь локальный порт и локальный адрес. У UDP нет возможности создавать соединения, поэтому удаленного адреса и порта просто не может быть. Полный код описания процедур `GetTcpTable`, `GetUdpTable` и необходимых структур можно увидеть в листинге 7.7. Ради экономии места я привожу только само описание. На самом деле вы должны создать модуль и добавить в него код из листинга 7.7.

Листинг 7.7. Описание необходимых функций и структур

```
type
// Описание отдельной записи TCP
PMIB_TCPROW = ^MIB_TCPROW;
MIB_TCPROW = record
 dwState: DWORD;
 dwLocalAddr: DWORD;
 dwLocalPort: DWORD;
 dwRemoteAddr: DWORD;
 dwRemotePort: DWORD;
end;

// Структура для хранения массива записей TCP
PMIB_TCPTABLE = ^MIB_TCPTABLE;
MIB_TCPTABLE = record
```

```
 dwNumEntries: DWORD;
 table: array [0..0] of MIB_TCPROW;
end;

// Описание отдельной UDP-записи
PMIB_UDPROW = ^MIB_UDPROW;
MIB_UDPROW = record
 dwLocalAddr: DWORD;
 dwLocalPort: DWORD;
end;

// Структура для хранения массива записей UDP
PMIB_UDPTABLE = ^MIB_UDPTABLE;
MIB_UDPTABLE = record
 dwNumEntries: DWORD;
 table: array [0..0] of MIB_UDPROW;
end;

var
// Функция получения таблицы TCP
GetTcpTable: function (pTcpTable: PMIB_TCPTABLE;
 var pdwSize: DWORD; bOrder: BOOL): DWORD; stdcall;
 {$EXTERNALSYM GetTcpTable}

// Функция получения таблицы UDP
GetUdpTable: function (pUdpTable: PMIB_UDPTABLE;
 var pdwSize: DWORD; bOrder: BOOL): DWORD; stdcall;
 {$EXTERNALSYM GetUdpTable}
```

Теперь необходимо написать код загрузки библиотеки. Давайте создадим для этого процедуру `LoadAPIHelpAPI`:

```
procedure LoadAPIHelpAPI;
begin
 if HIphlpapi = 0 then
 HIpHlpApi := LoadLibrary('iphlpapi.dll');
```

```
 if HIpHlpApi > HINSTANCE_ERROR then
 begin
  @GetTcpTable := GetProcAddress(HIpHlpApi, 'GetTcpTable');
  @GetUdpTable := GetProcAddress(HIpHlpApi, 'GetUdpTable');
 end;
end;
```

Тут можно заметить переменную `HIpHlpApi`. Что это? Это дескриптор загруженной библиотеки. Эту переменную нужно объявить где-нибудь в модуле чуть ранее:

```
var
HIpHlpApi: THandle = 0;
```

Теперь вернемся к `LoadAPIHelpAPI`. Сначала проверяется переменная `HIpHlpApi`. Если она равна нулю, то загружаем сетевую библиотеку. Кстати, имя этой библиотеки iphlpapi.dll. Теперь проверяем результат. Если он удачен, то получаем адреса необходимых функций с помощью `GetProcAddress`.

Чтобы функция выполнялась автоматически при использовании модуля, вызовем ее в разделе `initialization` нашего модуля:

```
initialization
 LoadAPIHelpAPI;
```

Будем следовать правилам хорошего тона и напишем функцию, которая будет освобождать дескриптор загруженной библиотеки:

```
procedure FreeAPIHelpAPI;
begin
 if HIpHlpApi <> 0 then FreeLibrary(HIpHlpApi);
 HIpHlpApi := 0;
end;
```

Чтобы эта функция вызывалась автоматически при завершении работы программы, напишем вызов `FreeAPIHelpAPI` в разделе `finalization`:

```
finalization
 FreeAPIHelpAPI;
```

Полный код модуля с описанием функций, загрузки и выгрузки можно найти на компакт-диске в файле HorrificHelpAPI.pas.

Теперь переходим непосредственно к примеру получения таблицы состояний TCP-портов с использованием функций, которые мы только что рассмотрели. Создаем новый проект и добавляем созданный ранее заголовоч-

ный файл. На форме нам понадобится только компонент типа `TListView`, которому установим следующие свойства:

- `Name` — установим в `lwTCP`;
- `ViewStyle` — будет `vsReport`, чтобы видеть сетку.

Дважды щелкните по созданному компоненту `ListView` и добавьте 7 колонок: протокол, процесс, локальный адрес, локальный порт, удаленный адрес, удаленный порт, состояние. Мы будем заполнять все колонки, кроме "процесс". Старыми функциями, которые мы сегодня рассматриваем, процесс получить нельзя. Не даром же Microsoft ввела расширенные функции работы с таблицами TCP и UDP, которые мы рассмотрим позднее. Может быть, и можно каким-то другим способом, но я не знаю — как, и особо не копался в этой теме.

После всех манипуляций у вас должна получиться форма, похожая на рис. 7.14.

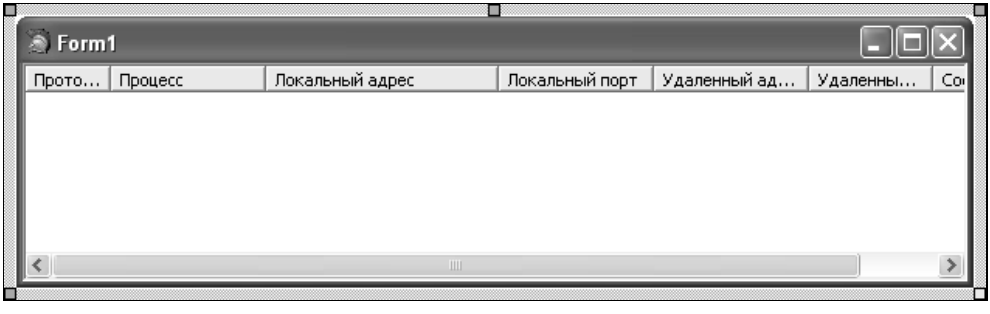


Рис. 7.14. Форма будущей программы

Теперь посмотрите на листинг 7.8, где показан пример кода, получающего TCP-таблицу.

Листинг 7.8. Получение таблицы состояний TCP-портов

```
var
 error, dwSize:DWORD;
 tcpTable:PMIB_TCPTABLE;
 i:Integer;
begin
 lwTCP.Items.Clear;
 dwSize:=0;
```

```
 // Первый вызов для определения количества записей
 error := GetTcpTable(nil, &dwSize, TRUE);
 if (error <> ERROR_INSUFFICIENT_BUFFER) then
 exit;
 // Выделяем необходимую память и получаем таблицу TCP
 try
 ReallocMem(tcpTable, dwSize);
 error := GetTcpTable(tcpTable, &dwSize, TRUE);
 if (error>0) then
 exit;

 // Перебираем все записи и добавляем их в ListView
 for i := 0 to tcpTable.dwNumEntries - 1 do
 begin
 with lwTCP.Items.Add do
  begin
   Caption:='TCP';
   SubItems.Add(inet_ntoa(TInAddr(tcpTable^.table[i].dwLocalAddr)));
   SubItems.Add(IntToStr(ntohs(tcpTable^.table[i].dwLocalPort)));
   SubItems.Add(inet_ntoa(TInAddr(tcpTable^.table[i].dwRemoteAddr)));
   SubItems.Add(IntToStr(ntohs(tcpTable^.table[i].dwRemotePort)));
   SubItems.Add(TCPState[tcpTable^.Table[I].dwState]);
   SubItems.Add('');
  end;
 end;
 finally
  FreeMem(tcpTable);
 end;
end;
```

Примечание

При отображении номера порта не забываем, что его значение имеет интернет-формат последовательности байтов, поэтому необходимо превратить число в общепринятое с помощью функции `ntohs`. Если вы забудите сделать это, то в окне будет отображаться некорректное значение.

После очистки компонента `ListView` вызываем функцию `GetTcpTable`. Обратите внимание, что первый параметр равен `nil`. Вместо того чтобы указать переменную, куда функция запишет результат, передается нулевое значение. Почему? Дело в том, что мы не знаем, сколько записей вернет функция и сколько памяти надо выделить под хранение результата.

Так как мы указали нулевое значение в первом параметре и нулевой размер во втором, функция должна завершиться ошибкой `ERROR_INSUFFICIENT_BUFFER`, а во втором параметре вернуть количество записей в системе. Выделяем необходимую память с помощью `ReallocMem`.

Теперь уже по-человечески вызываем `GetTcpTable` и получаем нормальную таблицу. Если результат не равен нулю, то все закончилось успешно. Иначе нужно сваливать, потому что произошла ошибка и никаких данных функция не вернула.

Теперь все банально: количество записей находится в `tcpTable.dwNumEntries`. Доступ к каждой отдельной структуре, описывающей состояние TCP-порта, можно получить так:

```
tcpTable^.table[номер записи]
```

Дальнейшие комментарии мне кажется излишни. Все и так понятно из кода. Хотя нет, нужно еще сказать о состоянии. Если с адресом и портом все понятно, то вот состояние — это вопрос. Судя по справке MSDN, состояние может принимать значения от 1 до 12, и для этого заведены следующие константы: `MIB_TCP_STATE_CLOSED`, `MIB_TCP_STATE_LISTEN`, `MIB_TCP_STATE_SYN_SENT`, `MIB_TCP_STATE_SYN_RCVD`, `MIB_TCP_STATE_ESTAB`, `MIB_TCP_STATE_FIN_WAIT1`, `MIB_TCP_STATE_FIN_WAIT2`, `MIB_TCP_STATE_LAST_ACK`, `MIB_TCP_STATE_CLOSE_WAIT`, `MIB_TCP_STATE_CLOSING`, `MIB_TCP_STATE_TIME_WAIT`, `MIB_TCP_STATE_DELETE_TCB`. Уже по названию можно понять, для чего нужны эти константы. Исходя из личного опыта, могу сказать, что иногда состояние может быть равным 0, хотя соответствующей константы нет. Видимо, это состояние нужно воспринимать как ошибку, а может быть, как что-то неопознанное. Чтобы удобнее было превращать числовое значение состояния в строку, можно создать константу в виде массива строк, например, вот так:

```
TcpState: array [0..12] of String = (
 '???', 'CLOSED', 'LISTENING', 'SYN_SENT',
 'SYN_RCVD', 'ESTABLISHED', 'FIN_WAIT1',
 'FIN_WAIT2', 'CLOSE_WAIT', 'CLOSING',
 'LAST_ACK', 'TIME_WAIT', 'DELETE_TCB');
```

Ради экономии места я не буду рассматривать код получения UDP-таблицы. Вы его можете написать самостоятельно, потому что он идентичен работе с TCP. Просто заменяем функцию `GetTCPTable` на `GetUDPTable`, а перемен-

ную типа PMIB_TCPTABLE на PMIB_UDPTABLE. Попробуйте реализовать код самостоятельно, не заглядывая сейчас в листинг 7.9. Ну а на рис. 7.15 вы можете увидеть результат работы программы.

Листинг 7.9. Получение таблицы состояний UDP-портов

```
var
error, dwSize:DWORD;
udpTable:PMIB_UDPTABLE;
i:Integer;
begin
  // Определяем количество UDP-записей
  dwSize:=0;
  error := GetUDPTable(nil, &dwSize, TRUE);
  if (error <> ERROR_INSUFFICIENT_BUFFER) then
    exit;
  // Выделяем память и получаем UDP-таблицу
  ReallocMem(udpTable, dwSize);
  error := GetUdpTable(udpTable, &dwSize, TRUE);
  if (error>0) then
    exit;

  // Просматриваем таблицу и добавляем записи в ListView
  for i := 0 to udpTable.dwNumEntries - 1 do
  begin
    with lwTCP.Items.Add do
    begin
      Caption:='TCP';
      SubItems.Add('');

      SubItems.Add(inet_ntoa(TInAddr(udpTable^.table[i].dwLocalAddr)));
      SubItems.Add(IntToStr(ntohs(udpTable^.table[i].dwLocalPort)));
    end;
  end;
  FreeMem(udpTable);
end;
```

Как видите, ничего сверхъестественного мы не написали. Никаких сверхсложных алгоритмов, а только две функции, о которых просто нужно знать и понимать смысл работы. В *разд. 7.6* вы узнаете про расширенные функции работы с TCP- и UDP-таблицами, а также как с ними работать.

На этом завершаем рассказ. Спокойной ночи, дорогие друзья. Тфу, насмотрелся с детьми "Спокойной ночи". Оксана Федорова рулит! В детской передаче снимать "Мисс мира" — гениальное решение. Мой сын уже с детства засматривается на красивых женщин в самом расцвете сил.

ПРИМЕЧАНИЕ

На компакт-диске в каталоге \Примеры\Глава 7\NetView1\ вы можете найти модифицированный пример этого раздела и цветные рисунки, относящиеся к нему.

7.6. Быстрая проверка состояний портов: вариант 2

В *разд. 7.5* мы познакомились со старыми функциями определения состояния портов. В данном разделе мы двинемся дальше и познакомимся с новыми функциями Windows XP, которые делают то же самое и еще немного интересного. Новый вариант программы будет отображать не только открытые порты, но и процессы, которые их открыли. В этом нам помогут функции, которые не описаны в заголовочных файлах Delphi и даже VC++, поэтому мы будем их загружать динамически.

Итак, работу начнем с заголовочного файла, потому что необходимые сегодня функции в Delphi не описаны. Нам понадобятся следующие функции: `AllocateAndGetUdpExTableFromStack`, `AllocateAndGetTcpExTableFromStack`, `Process32First`, `CreateToolhelp32Snapshot` и `Process32Next`. Первые две из них реализованы в библиотеке iphlpapi.dll и необходимы для получения из стека таблицы открытых TCP- и UDP-портов соответственно. Какая из функций какую таблицу возвращает, нетрудно догадаться по имени. Остальные три реализованы в kernel32.dll и пригодятся нам для определения процесса, который открыл порт.

В предыдущем разделе мы уже писали программу TCPView с запасом на будущее и даже в главном окне подготовили отдельную колонку для отображения имени процесса. Сейчас с помощью нескольких волшебных телодвижений мы заполним эту колонку.

Итак, открываем заголовочный файл, созданный в *разд. 7.5*, и начинаем добавлять в него описания функций. Как и в прошлый раз, будем объявлять функции в виде переменных, чтобы загружать их динамически.

Двинемся по порядку, а значит, начнем с рассмотрения функции AllocateAndGetTcpExTableFromStack:

```
AllocateAndGetTcpExTableFromStack: function (
 pTCPTable: PMIB_TCPEXTABLE;
 bOrder: BOOL;
 heap: THandle;
 zero: DWORD;
 flags: DWORD
): DWORD; stdcall;
```

Этим мы объявляем переменную AllocateAndGetTcpExTableFromStack типа "функция", которая принимает, если я правильно посчитал, пять параметров. До пяти я, вроде, считать умею, а если что-то не так, то простите старика инвалида клавиатурного труда. Так, функция получает следующие параметры:

- указатель типа PMIB_TCPEXTABLE, через который нам вернут массив состояния TCP-портов;
- булево значение, определяющее, нужно ли сортировать таблицу;
- куча (heap), в которой нужно выделить память для хранения результирующей таблицы. Вполне логично хранить результат в куче своего процесса, указатель на которую можно получить с помощью функции GetProcessHeap;
- флаги, определяющие поведение с кучей. В утилите Руссиновича здесь указывается зачем-то двойка, и если опять же запустить поиск по Интернету, то все примеры тоже будут автоматом указывать в этом параметре это же число. Зачем? Видимо, код копируется даже без осмысливания того, что он делает. Нам никакие специфические поведения кучи не нужны, поэтому будем указывать 0;
- последний флаг определяет IP-адреса, для которых нужно получить таблицу. Здесь можно указать флаги AF_INET или AF_INET6 для IP-протокола 6-й версии. Опять же, все копируют код один к одному и явно указывают число 2, а это значение константы AF_INET. Обе константы объявлены в заголовочном файле WinSock, хотя, нет — константа AF_INET6 есть только в заголовочном файле второй версии, ведь первый WinSock ничего не знал о IPv6.

Запустите поиск в Рунете по названию функции AllocateAndGetTcpExTableFromStack и в большинстве случаев вы узнаете, что функция недокументирована. Кем недокументирована? В MSDN есть подробное описание, нужно просто уметь искать и знать, где искать. Све-

жий MSDN всегда можно найти по адресу **msdn.microsoft.com**. Да, он обновляется с задержкой и уже после выхода ОС, но чтобы быть впереди планеты, просто нужно подписаться на рассылку, которая стоит немалых денег. А если новой функции нет в старой версии справки, то это не значит, что описания нет вовсе.

Кстати, если верить MSDN, то эта функция устарела и больше не поддерживается в новоиспеченной Windows Vista! Я не проверял, потому что еще даже не ставил "свистуна", но раз так, то наш универсальный пример будет как раз кстати. Если вы посмотрите в SDK для Vista, то функция там будет объявлена, но это только для совместимости. Так что не пытайтесь вызвать напрямую, иначе вас ждет крах программы. Что будет в качестве замены, я пока не знаю, а Microsoft молчит, или я не нашел. Если найду, то обязательно расскажу, ведь полноценная замена необходима, нужно же как-то определять не только состояние портов, но и процессы, которые инициировали соединение.

Таблицу состояний UDP-портов можно узнать с помощью функции `AllocateAndGetUdpExTableFromStack`, которую необходимо объявить следующим образом:

```
AllocateAndGetUdpExTableFromStack: function (
 pUDPTable: PMIB_UDPEXTABLE;
 bOrder: BOOL;
 heap: THandle;
 zero: DWORD;
 flags: DWORD
): DWORD; stdcall;
```

Параметры практически идентичны функции работы с TCP-портами. Разница только в первом параметре, который имеет тип `PMIB_UDPEXTABLE`. Порты UDP не имеют соединений, поэтому их таблица состояний немного отличается.

Теперь поговорим о структурах данных, через которые мы будем получать результирующие таблицы. Начнем с TCP-портов. Функция принимает в качестве первого параметра тип данных `PMIB_TCPEXTABLE`. На самом деле это структура следующего вида:

```
PMIB_TCPEXTABLE = ^TMIB_TCPEXTABLE;
TMIB_TCPEXTABLE = packed record
 dwNumEntries: DWORD;
 Table: array[0..0] of TMIB_TCPEXROW;
end;
```

Тут всего два поля: количество элементов в таблице и массив элементов таблицы состояний портов. Каждый элемент массива — это тоже структура, которая имеет тип TMIB_TCPEXROW и выглядит следующим образом:

```
PMIB_TCPEXROW = ^TMIB_TCPEXROW;
TMIB_TCPEXROW = packed record
 dwState: DWORD;
 dwLocalAddr: DWORD;
 dwLocalPort: DWORD;
 dwRemoteAddr: DWORD;
 dwRemotePort: DWORD;
 dwProcessID: DWORD;
end;
```

Функция GetTcpTable (мы ее рассматривали в предыдущем разделе) возвращает примерно такую же структуру. Здесь так же есть локальный адрес, локальный порт, удаленный адрес и удаленный порт. Самое последнее поле является новым и определяет идентификатор процесса, который открыл порт.

Теперь посмотрим на структуру PMIB_UDPEXTABLE, которая передается в качестве первого параметра функции для получения состояний UDP-портов:

```
PMIB_UDPEXTABLE = ^TMIB_UDPEXTABLE;
TMIB_UDPEXTABLE = packed record
 dwNumEntries: DWORD;
 Table: array[0..0] of TMIB_UDPEXROW;
end;
```

Тут снова нас ожидает количество элементов в таблице состояний и массив из структур типа TMIB_UDPEXROW. Эта структура выглядит так:

```
PMIB_UDPEXROW = ^TMIB_UDPEXROW;
TMIB_UDPEXROW = packed record
 dwLocalAddr: DWORD;
 dwLocalPort: DWORD;
 dwProcessID: DWORD;
end;
```

В структуре мы видим локальный адрес, локальный порт и идентификатор процесса. Информации об удаленной машине нет и не может быть.

Для реализации примера нам понадобятся еще три системные функции из Windows-ядра kernel32.dll:

```
CreateToolhelp32Snapshot: function (dwFlags,
    th32ProcessID: DWORD): THandle; stdcall;
{$EXTERNALSYM CreateToolhelp32Snapshot}

Process32First: function (hSnapshot: THandle;
    var lppe: TProcessEntry32): BOOL; stdcall;
{$EXTERNALSYM Process32First}

Process32Next: function (hSnapshot: THandle;
    var lppe: TProcessEntry32): BOOL; stdcall;
{$EXTERNALSYM Process32Next}
```

Давайте кратко пробежимся по этим функциям:

- `CreateToolhelp32Snapshot` — создает снимок указанного процесса;
- `Process32First` — возвращает первый процесс из снимка;
- `Process32Next` — возвращает следующий процесс из снимка.

За более подробной информацией по этим функциям обращайтесь к MSDN. У них достаточно много возможностей, различных флагов, поэтому описать их в одной книге проблематично.

Мы объявили переменные, через которые будем обращаться к системным функциям, но все они являются указателями и на данном этапе указывают в никуда. Чтобы через эти переменные можно было непосредственно обращаться к функциям системы, необходимо получить соответствующие адреса. В прошлый раз для этого мы создали в нашем модуле функцию `LoadAPIHelpAPI`. Давайте расширим ее и добавим следующие строки:

```
@AllocateAndGetTcpExTableFromStack:=
   GetProcAddress(HIpHlpApi, 'AllocateAndGetTcpExTableFromStack');
@AllocateAndGetUdpExTableFromStack:=
   GetProcAddress(HIpHlpApi, 'AllocateAndGetUdpExTableFromStack');

@CreateToolhelp32Snapshot :=
  GetProcAddress(GetModuleHandle('kernel32.dll'),
  'CreateToolhelp32Snapshot');
@Process32First :=
  GetProcAddress(GetModuleHandle('kernel32.dll'), 'Process32First');
```

```
@Process32Next :=
  GetProcAddress(GetModuleHandle('kernel32.dll'), 'Process32Next');
```

Теперь после загрузки каждая переменная будет указывать на соответствующую функцию в системе. Если какая-то функция не будет найдена, то соответствующая переменная будет равна нулю. Эту особенность мы будем использовать для того, чтобы определить, поддерживает ли ОС данные функции, или необходимо использовать универсальный код, который мы рассматривали в прошлый раз.

Вот мы и подошли к самому интересному — реализации универсального примера. В прошлый раз мы написали пример, в котором по событию `OnShow` вызывалась функция `GetConnections`, в которой и происходило определение состояний портов. Улучшим пример, и теперь безусловный вызов заменим на условие:

```
procedure TForm1.FormShow(Sender: TObject);
begin
 if @AllocateAndGetTcpExTableFromStack=nil then
  GetConnections
 else
  GetExConnections;
end;
```

Если указатель `AllocateAndGetTcpExTableFromStack` равен нулю, то соответствующей функции нет в системе, и вызываем `GetConnections`. Если он не равен нулю, то функция найдена в системе, и можно использовать расширенные функции, которые мы рассмотрели сегодня. Для этого вызываем расширенный вариант `GetExConnections`. Код этой функции показан в листинге 7.10.

Листинг 7.10. Новый вариант получения состояния портов

```
procedure TForm1.GetExConnections;
var
 TCPExTable: PMIB_TCPEXTABLE;
 UDPExTable: PMIB_UDPEXTABLE;
 hSnapshot: THandle;
 i:Integer;
 local_name:array[0..255]of char;
 ExeName:String;
```

```
begin
 lwTCP.Items.BeginUpdate;
 lwTCP.Items.Clear;
 // Получаем снимок процессов
 hSnapshot := CreateToolhelp32Snapshot($2, 0);
 try
  // Определяем таблицу состояний TCP-портов
  if AllocateAndGetTcpExTableFromStack(@TCPExTable, False,
        GetProcessHeap, 2, 2) = NO_ERROR then
   begin
    for i := 0 to TCPExTable.dwNumEntries - 1 do
     begin
      with lwTCP.Items.Add do
       begin
        Caption:='TCP';
        if (hSnapshot = INVALID_HANDLE_VALUE) then
    // Если не удалось получить снимок, то имя процесса оставить пустым
         SubItems.Add('')
        else
         begin
          // Снимок процессов был получен удачно, поэтому
          // переводим ID в человеческое имя
          ExeName:=ProcessPidToName(hSnapshot,
             tcpExTable.Table[i].dwProcessId);
          SubItems.Add(ExeName);
         end;

        SubItems.Add(inet_ntoa(
              TInAddr(TCPExTable^.Table[I].dwLocalAddr)));
        SubItems.Add(IntToStr(TCPExTable^.Table[I].dwLocalPort));
        SubItems.Add(
              inet_ntoa(TInAddr(TCPExTable^.Table[I].dwRemoteAddr)));
        SubItems.Add(IntToStr(TCPExTable^.Table[I].dwRemotePort));
        SubItems.Add(TCPState[TCPExTable^.Table[I].dwState]);
       end;
     end;
   end;
```

```
  // Определяем таблицу состояний UPX-портов
  if AllocateAndGetUdpExTableFromStack(@UdpExTable, False,
          GetProcessHeap, 2, 2) = NO_ERROR then
   begin
    for i := 0 to UDPExTable.dwNumEntries - 1 do
     begin
      with lwTCP.Items.Add do
       begin
        Caption:='UDP';

        if (hSnapshot = INVALID_HANDLE_VALUE) then
         // Если не удалось получить снимок,
         // то имя процесса оставить пустым
         SubItems.Add('')
        else
         // Снимок процессов был получен удачно, поэтому переводим
         // ID в человеческое имя
         SubItems.Add(ProcessPidToName(hSnapshot,
            UDPExTable.Table[i].dwProcessId));
        gethostname(local_name, 255);
        SubItems.Add(inet_ntoa(
            TInAddr(UDPExTable^.Table[I].dwLocalAddr)));
        SubItems.Add(IntToStr(UDPExTable^.Table[I].dwLocalPort));
       end;
     end;
   end;
 finally
  lwTCP.Items.EndUpdate;
 end;
end;
```

Логика `GetExConnections` практически не изменилась по сравнению с написанной в *разд. 7.5* функцией `GetConnections` (см. листинг 7.8). Мы точно так же получаем таблицу состояний и выводим ее содержимое, просто другими API-функциями. Единственное, что заслуживает отдельного внимания, — это вызов функции `ProcessPidToName`, которая должна переводить

идентификатор процесса в удобочитаемое имя. Эта функция с подробнейшими комментариями представлена в листинге 7.11.

Листинг 7.11. Определение имени процесса по идентификатору

```
function TForm1.ProcessPidToName(hProcess: THandle; ProcID: DWORD):
String;
var
 procEntry:TPROCESSENTRY32;
 ProcessName:String;
begin
procEntry.dwSize := sizeof(procEntry);
ProcessName:='???';
// Получаем первый процесс из снимка,
// который мы сделали в функции GetExConnections
if (Process32First(hProcess, procEntry)) then
 begin
  // Запускаем цикл перебора всех процессов
  repeat
  // Если текущий процесс равен искомому, то возвращаем его имя
  if (procEntry.th32ProcessID = ProcId) then
   begin
    ProcessName:=procEntry.szExeFile;
    Result:=ProcessName;
    exit;
   end;
  // Цикл выполнять, пока функция Process32Next не вернет nil,
  // т. е. не достигнем конца снимка
  until (not Process32Next(hProcess, &procEntry));
 end;
Result:=ProcessName;
end;
```

Смысл функции очень прост — сначала функция получает с помощью `Process32First` первый процесс из снимка, который мы сделали с помощью `CreateToolhelp32Snapshot`. После этого запускаем цикл, в котором проверяем: если идентификатор текущего процесса равен искомому, то бе-

рем его имя, иначе берем следующий процесс с помощью `Process32Next` и переходим к следующему шагу цикла.

Как видите, все гениальное — в простоте и умении искать нужные функции. Можете улучшить этот пример, чтобы он обновлял таблицу по таймеру и подсвечивал записи, состояние портов которых изменилось или появились новые записи в таблице. Можно добавить возможность уничтожения выделенного процесса, ведь соответствующий идентификатор мы научились определять. Только не забывайте, что при использовании старых функций процесс не определен. Если я узнаю, как это сделать, то обязательно расскажу.

Чтобы реализовать удаление процесса, можно идентификатор помещать в 6-ю колонку `ListView`. У нас на форме всего пять колонок, поэтому 6-я отображаться не будет, но содержать значение может. Итак, следующий код берет 6-е поле для уничтожения процесса:

```
var
 procID:Cardinal;
 procHandle:THandle;
begin
 if lwTCP.Selected<>nil then
  begin
   Screen.Cursor:=crHourGlass;
   try
    procID:=StrToInt(lwTCP.Selected.SubItems[6]);
    procHandle:=OpenProcess(PROCESS_TERMINATE, false, procID);
    TerminateProcess(procHandle, 0);
    CloseHandle(procHandle);
    Sleep(3000);
   finally
    Screen.Cursor:=crDefault;
   end;
  end;
```

Кстати, написанный пример получает только состояние IP-портов старой версии. Вы можете еще добавить состояния IPv6 простым изменением последнего флага при вызове функций `AllocateAndGetTcpExTableFromStack` и `AllocateAndGetUdpExTableFromStack`.

Примечание

На компакт-диске в каталоге \Примеры\Глава 7\NetView2\ вы можете найти модифицированный пример этого раздела и цветные рисунки, относящиеся к нему.

7.7. Работа с ICMP на примере *ping*

В первом издании данной книги был пример написания собственной утилиты `ping` на примере библиотеки ICS. Пример обладал несколькими проблемными местами:

- библиотека устанавливалась с проблемами и после нескольких шаманских действий;
- библиотека неуклюжая, и банальный `ping` реализован в виде двух классов, а на самом деле можно обойтись небольшой функцией.

На самом деле, реализовать `ping` без использования сторонних компонентов не так уж и сложно. Достаточно написать несколько строк кода и результат в кармане, а точнее, на экране. Но в Delphi есть небольшая проблема — необходимые ICMP-функции не описаны в Delphi. Придется описывать функции самостоятельно, но вы должны были уже привыкнуть к этому.

Итак, создадим новый модуль и назовем его `PingUnit`. В этом модуле будут описаны все необходимые функции и структуры работы с ICMP, а так же функция `PingHost`, которая будет использовать функции и предоставит нашему приложению удобный способ работы с протоколом. Код можно увидеть в листинге 7.12.

Листинг 7.12. Модуль `PingUnit`

```
unit PingUnit;

interface

uses
 windows, winsock2, math, SysUtils;

function Pinghost(host:String):String;

type

  PIPOptionInformation = ^TIPOptionInformation;
  TIPOptionInformation = packed record
     TTL: Byte;          // Время жизни
     TOS: Byte;          // Тип сервиса
     Flags: Byte;        // Флаг
```

```
    OptionsSize: Byte; // Размер данных для опций
    OptionsData: PChar;// Буфер опций, максимальный размер - 40
  end;

  PIcmpEchoReply = ^TIcmpEchoReply;
  TIcmpEchoReply = packed record
    Address: LongInt;
    Status: DWord;
    RTT: DWord;
    DataSize: Word;
    Reserved: Word;
    Data: Pointer;
    Options: TIPOptionInformation;
  end;

var
  IcmpCreateFile:function(): THandle; stdcall;
  IcmpCloseHandle:function(IcmpHandle: THandle): Boolean; stdcall;
  IcmpSendEcho:function(IcmpHandle:THandle; DestinationAddress: LongInt;
        RequestData:Pointer; RequestSize: Word;
        RequestOptions: PIPOptionInformation;
        ReplyBuffer: Pointer; ReplySize: DWord;
        Timeout: DWord): DWord; stdcall;

implementation

var
 IcmpDll: THandle = 0;

function Pinghost(host:String):String;
var
 hICMP:THandle;
 pReqData, pData: Pointer;
 Msg: String;
 BufferSize: Integer;
```

```
 pIPE: PIcmpEchoReply;
 IPOpt: TIPOptionInformation;
 Reply:TIcmpEchoReply;
begin
 Result:='Ошибка';

 // Открываем дескриптор для работы
 hICMP := IcmpCreateFile();
 if hICMP = INVALID_HANDLE_VALUE then
  exit;

 // Выделяем память для буфера
 BufferSize := SizeOf(TICMPEchoReply) + 56;
 GetMem(pReqData, 56);
 GetMem(pData, 56);
 GetMem(pIPE, BufferSize);

 try
  // Заполняем буфер символом $20
  FillChar(pReqData^, 56, $20);
  // Копируем в буфер сообщение, хотя можно этого и не делать
  Msg := 'Flenov Michael ping';
  Move(Msg[1], pReqData^, min(56, Length(Msg)));

  pIPE^.Data := pData;
  FillChar(pIPE^, SizeOf(pIPE^), 0);

  FillChar(IPOpt, SizeOf(IPOpt), 0);
  IPOpt.TTL   := 64; // Время жизни
  IPOpt.Flags := $2; // Включить фрагментацию

  // Отправляем запрос
  IcmpSendEcho(hICMP, LookupName(host).S_addr, pReqData, 56,
            @IPOpt, pIPE, BufferSize, 5000);
```

```
  // Читаем результат
  Reply := pIPE^;
  Result:='Получено ' + IntToStr(Reply.DataSize)+' байт '+
      inet_ntoa(TInAddr(Reply.Address))+' (' +
      IntToStr(Reply.RTT)+' мс)';

 Finally
  // Очищаем буферы
  FreeMem(pIPE);
  FreeMem(pData);
  FreeMem(pReqData);
  // Закрываем дескриптор
  IcmpCloseHandle(hICMP);
 end;
end;


function LoadAPIHelpAPI: Boolean;
begin
 Result := False;
 if IcmpDll = 0 then
  IcmpDll := LoadLibrary('iphlpapi.dll');

 if IcmpDll > HINSTANCE_ERROR then
  begin
   IcmpCreateFile  := GetProcAddress(IcmpDll, 'IcmpCreateFile');
   IcmpCloseHandle := GetProcAddress(IcmpDll, 'IcmpCloseHandle');
   IcmpSendEcho    := GetProcAddress(IcmpDll, 'IcmpSendEcho');
   Result:=true;
  end;
end;

procedure FreeAPIHelpAPI;
begin
 if IcmpDll <> 0 then FreeLibrary(IcmpDll);
```

```
 IcmpDll := 0;
end;

initialization
 LoadAPIHelpAPI;

finalization
 FreeAPIHelpAPI;
end.
```

Начнем рассмотрение модуля с функций, а что касается ICMP, то в модуле объявлено три функции:

- `IcmpCreateFile` — функция открывает дескриптор, через который можно работать с ICMP. Эта функция должна вызываться первой;
- `IcmpSendEcho` — отправить эхо-запрос, т. е. `ping`;
- `IcmpCloseHandle` — закрыть дескриптор. После работы с ICMP необходимо закрыть дескриптор с помощью этой функции.

Все функции объявлены как переменные, а их реализация находится в библиотеке iphlpapi.dll. У функций одноименные имена, и чтобы получить их адреса, я написал функцию `LoadAPIHelpAPI`, которая вызывется из раздела `initialization`. Она загружает библиотеку iphlpapi.dll и с помощью `GetProcAddress` получает необходимые нам адреса.

Первая функция (`IcmpCreateFile`) не получает никаких параметров, а только возвращает дескриптор, который мы будем потом использовать. Последняя функция (`IcmpCloseHandle`) получает в качестве параметра открытый дескриптор и закрывает его. Самая интересная функция — это `IcmpSendEcho` и выглядит она следующим образом:

```
IcmpSendEcho:function(
 IcmpHandle:THandle;
 DestinationAddress: LongInt;
 RequestData:Pointer;
 RequestSize: Word;
 RequestOptions: PIPOptionInformation;
 ReplyBuffer: Pointer;
 ReplySize: DWord;
```

```
 Timeout: DWord
): DWord; stdcall;
```

Тут у нас 8 параметров:

- `IcmpHandle` — заранее открытый с помощью `IcmpCreateFile` дескриптор;
- `DestinationAddress` — адрес получателя, т. е. компьютера, который мы хотим пингануть. Обратите внимание, что этот параметр имеет тип `LongInt`. Если посмотреть на файл справки или MSDN, то вы там увидите тип `IPAddr`, но это одно и то же. Мы привыкли работать со структурой `TInAddr` для адресации. В данном случае нужно использовать поле `S_addr` данной структуры;
- `RequestData` — буфер, содержащий данные, которые необходимо отправить;
- `RequestSize` — размер отправляемых данных;
- `RequestOptions` — это структура типа `TIPOptionInformation`, в которой задаются параметры запроса. Эту структуру мы рассмотрим чуть позже, но сейчас достаточно знать, что здесь находятся такие опции, как время жизни пакета, флаги и т. д.;
- `ReplyBuffer` — буфер, в который будет помещен результат работы. Для эхо-запроса `ping`, в этом буфере будет структура типа `TIcmpEchoReply`. Для хранения буфера необходимо выделить память в размере отправляемых данных, плюс размер структуры `TICMPEchoReply`;
- `ReplySize` — размер буфера в параметре `ReplyBuffer`;
- `Timeout` — время ожидания ответа.

У нас есть практически все, что необходимо, осталось только рассмотреть структуры `TIPOptionInformation` и `TIcmpEchoReply`. Первая из них задает параметры запроса и выглядит так:

```
TIPOptionInformation = packed record
 TTL: Byte;
 TOS: Byte;
 Flags: Byte;
 OptionsSize: Byte;
 OptionsData: PChar;
end;
```

Рассмотрим поля этой структуры:

- `TTL` — время жизни пакета, максимальное количество маршритизаторов, через которые может пройти пакет;

- `TOS` — тип сервиса;
- `Flags` — флаги IP-заголовка;
- `OptionsSize` — размер необязательных данных;
- `OptionsData` — необязательные данные.

Для эхо-запроса необходимо только указать `TTL`, для большинства случаев вполне достаточно 64. Можно еще указать флаг `$2`, что означает необходимость фрагментации пакетов.

Теперь посмотрим на структуру, через которую мы получаем результат. А она выглядит так:

```
TIcmpEchoReply = packed record
 Address: LongInt;
 Status: DWord;
 RTT: DWord;
 DataSize: Word;
 Reserved: Word;
 Data: Pointer;
 Options: TIPOptionInformation;
end;
```

У этой структуры следующие поля:

- `Address` — адрес компьютера, с которого пришел ответ;
- `Status` — статус;
- `RTT` — Round Trip Time, или время, затраченное на прохождения пакета туда и обратно;
- `DataSize` — размер полученных данных;
- `Reserved` — зарезервировано для использования системой;
- `Data` — полученные данные;
- `Options` — опции ответа в форме структуры `TIPOptionInformation`.

Теперь у вас есть вся необходимая информация, чтобы понять, как работает и что делает функция `Pinghost`. В качестве параметра она получает адрес хоста, которому нужно отправить эхо-запрос, а в результате возвращается строка, содержащая информацию о результате запроса.

Чтобы использовать модуль и написать реальный пример:

- создайте новое приложение;
- подключите модуль `PingUnit`;

- поместите на форму поле ввода `edPingAddress`, где будет указываться адрес компьютера, которому нужно отправить запрос;
- добавьте поле `Memo` с именем `DisplayMemo` для отображения результата;
- разместите кнопку, по нажатию которой будет отправляться запрос.

Теперь, по нажатию кнопки пишем следующую строку кода:

```
DisplayMemo.Lines.Add(Pinghost(edPingAddress.Text));
```

Вот и все. Пример готов и можно отпраздновать это бутылкой шампанского.

> ***Примечание***
>
> На компакт-диске в каталоге \Примеры\Глава 7\Ping\ вы можете найти модифицированный пример из этого раздела и цветные рисунки, относящиеся к нему.

7.8. Trace Route

Еще одна интересная утилита, которую можно построить на основе ICMP-пакетов — Trace Route. Чтобы написать такую программу, нужно понять, как она работает. А тут все достаточно просто.

Trace Route — это программа, которая показывает, через какие маршрутизаторы прошел пакет. Как получить список маршрутизаторов? Все очень просто, если пакет не может достигнуть получателя за указанное TTL число маршрутизаторов, то ответ пришлет последний из маршрутизаторов, который получил пакет. Отправив пакет, в котором TTL будет равен 1, ответ пришлет первый маршрутизатор, который встретится на пути следования пакета. Указав 2, ответ будет от второго маршрутизатора. А что если во второй раз пакет пойдет другим маршрутом? Как показывает практика, такое встречается очень редко. Чаще всего, маршрут только один.

Итак, чтобы написать собственную утилиту Trace Route, нужно последовательно увеличивать TTL, пока ответ не поступит от нужного нам хоста. Мы не будем писать полноценную программу, оставлю эту тему в качестве домашнего задания. Попробуйте сами реализовать все, что описано выше. Необходимые знания у вас уже есть. Я дам только пару советов.

- Как проверить, кто нам ответил? Очень просто, нужно перевести адрес в число с помощью `LookupName` (эту функцию мы уже не раз использовали) и сравнивать именно числовое значение.

- Сервер получатель или маршрутизатор может не отвечать на ICMP-пакеты, поэтому вы должны учитывать, что в определенный момент может не поступить ответа. Вполне возможно, что это маршрутизатор настроен так, чтобы не отвечать, и нужно продолжать увеличивать TTL и отправлять эхо-запросы. А если сервер уже достигнут, но он не отвечает? В этом случае можно сколько угодно увеличивать TTL, ответа не последует. Чтобы обезопасить себя от такой ситуации, нужно определить максимальное значение времени жизни пакета, после которого будем считать, что увеличение бессмысленно. Таким числом может быть 64. Этого вполне достаточно, чтобы пакет добрался до самого дальнего сервера в мире. Возможно, что где-то и есть пути длиннее, но это уже большая редкость.

Приложение

Описание компакт-диска

Папки	Описание
\Headers	Все необходимые заголовочные файлы, которые нужно будет подключать к Delphi для компиляции некоторых примеров
\Source	Исходные коды своих простых программ, чтобы вы могли ознакомиться с реальными приложениями. Их немного, но посмотреть стоит
\Soft	Инсталляционный пакет программы Adobe Acrobat Reader 5.0. Если у вас нет этой программы, то вы должны ее установить, чтобы можно было читать документацию, расположенную на диске
\Vr-online	Полная копия сайта автора, а это 100 Мбайт документации, полезной информации, исходных кодов и компонентов. Здесь же вы можете найти мою книгу "Библия Delphi" в электронном виде. В ней вы найдете все необходимые для понимания этого материала основы, и если вы еще ни разу не видели Delphi, то после прочтения этой книги вы сможете понять все описанное здесь
\Документация	Дополнительная документация, которая может понадобиться для понимания каких-то глав
\Иконки	В этом каталоге вы найдете большую коллекцию значков, которые можете использовать в своих программах. Эту коллекцию я подбирал достаточно долго, и все значки хорошего качества
\Компоненты	Дополнительные компоненты, которые будут использоваться в примерах книги
\Программы	Программы, которые пригодятся в программировании. Среди них Header Convert — программа, которая конвертирует заголовочные файлы с языка C на Delphi, и ASPack — программа сжатия исполняемых файлов

Литература

1. Журнал "Хакер". Рубрика "Кодинг" всегда содержит множество интересных примеров.
2. Фленов М. DirectX и C++. Искусство программирования. — СПб.: БХВ-Петербург, 2006.
3. Фленов М. DirectX и Delphi. Искусство программирования. — СПб.: БХВ-Петербург, 2006.
4. Фленов М. PHP глазами хакера. — СПб.: БХВ-Петербург, 2005.
5. Фленов М. Transact-SQL. — СПб.: БХВ-Петербург, 2005.
6. Фленов М. Искусство программирования игр на C++. — СПб.: БХВ-Петербург, 2006.

Предметный указатель

A

ASPack 17

C

CreateWindow 34

E

ernational Organization for
Standardization (ISO) 238

K

Key Objects Library (KOL) 37
KOL 37
KOL+MCK:
Создание проекта 39
Установка 39

L

LoadLibrary 58

M

Mirror Classes Kit (MCK) 38
MCK 38

N

NetBIOS 244

O

Open Systems Interconnection
(OSI) 238

S

Socket 246
Spoofing 243

U

UPX 20
UPX_Control 21

A

Ассемблер 63
внешний 66
встроенный 63
Использование внешней
процедуры 67
Комментарии 65
Компиляция 67
Регистр 64
Функция 65

Б

Библиотека:
netapi32.dll 406
VCL 40
WinSock 271

Библиотека ключевых объектов 37
Буфер обмена 123

Д

Дескриптор безопасности 165, 172
Динамическая библиотека 40, 56
  Определение адреса функции 58
Диск 353

К

Кнопка Пуск 76, 81
Код, безопасный 71
Комплект зеркальных классов 38
Компонент 15

М

Менеджер сервисов 193
Мониторинг 140
Мышь 109

Н

Наследование 16

О

Обои 105
Окно 105, 107, 108, 114, 119, 121
  дочернее 220
  нестандартной формы 155
  прозрачное 232
Операция сдвига 50
Оптимизатор
  встроенный 70

П

Панель задач 87
Пароль 133
Переключение экранов 150
Переменная окружения 348
Порт 247, 280, 282, 283, 287
  COM 364
  LPT 369
  асинхронная работа 281
  синхронная работа 280
Права доступа 182
Принтер 370
Протокол 239
  ARP 241, 411
  HTTP 336
  IP 240
  IPX 246
  NetBEUI 246
  RARP 242
  SPX 246
  TCP 243
  UDP 242, 331
Процессор 357

Р

Рабочий стол 90
Разрешение экрана 95

С

Сервис 189
Система 343
Сканер 386
  портов 283, 287
Служба 189
Соединение с сервером 281, 283
Сокет 246, 276
Спуфинг 243

У

Управление пользователями 208

Ф

Файл:
  сжатие 17
  тип 378

Ч Я

Чат 252
Ярлык 222