ЮРИЙ РЕВИЧ

bhv

НЕСТАНДАРТНЫЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ НА DELPHI

РЕЗИДЕНТНЫЕ ПРОГРАММЫ В WINDOWS

ИНСТАЛЛЯЦИЯ И ДЕИНСТАЛЛЯЦИЯ ПРОГРАММ

ПОИСК В ДОКУМЕНТАХ

РАБОТА С ГРАФИКОЙ В WINDOWS

ЛЮБИТЕЛЬСКАЯ КРИПТОГРАФИЯ

РАБОТА С COM- И USB-ПОРТАМИ

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

Юрий Ревич

НЕСТАНДАРТНЫЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ НА DELPHI

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.06
ББК 32.973.26-018.2
Р32

Ревич Ю. В.

Р32 Нестандартные приемы программирования на Delphi. — СПб.: БХВ-Петербург, 2005. — 560 с.: ил.

ISBN 5-94157-686-2

Книга призвана помочь программистам разрабатывать полноценные, профессиональные Windows-приложения в Delphi. Показано, как предотвращать повторный запуск приложения, работать с нестандартными окнами, перехватывать нажатие клавиш, создавать резидентные программы в Windows, а также инсталляторы и деинсталляторы программ, осуществлять поиск в документах, работать с COM- и USB-портами, шифровать текст и многое другое. Рассмотрены примеры решения этих и многих других проблем, которые встают при создании программы, ориентированной на долговременное использование и распространение. Приведены приемы работы с Windows API. Изложение ведется на примерах поэтапного создания реально работающих практических приложений. Компакт-диск содержит исходные тексты разобранных в книге примеров.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

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

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 21.09.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 45,16.
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 5-94157-686-2

Оглавление

Введение

О чем и для кого написана эта книга

> Я вообще всю жизнь полагал, что ничего хорошего из исполнения чаяний неких гипотетических потребителей, "народа", никогда не выходило. Писать (стихи, романы, пьесы, сценарии и... программы) надо исключительно для себя. Ну, и для близкого круга друзей.
>
> *Е. Козловский, "Ниоткуда с любовью"*

По классификации ученого-химика А. Шкроба, создателя хорошего сайта о науке под названием "VivosVoco", программисты делятся на любителей, дилетантов и профессионалов[1]. Любители пишут программы для развлечения, дилетанты пишут программы по необходимости, профессионалы пишут программы для заработка. Вероятно (кто бы провел такое исследование?), любителей и дилетантов больше, чем профессионалов. Мало того, с распространением Интернета и появлением онлайновых сообществ грани между любителями, дилетантами и профессионалами при разделении их по признаку цели все больше и больше стираются — к какой категории, например, отнести добровольных членов сообщества создателей Linux? Сейчас самодеятельное (объединим таким названием область деятельности любителей и дилетантов) программирование у периодических компьютерных изданий несколько не в моде — просто в силу того, что значительную часть ниши, которую раньше занимали программы непрофессионалов (точнее, программирующих специалистов), ныне занимают универсальные фирменные продукты. Однако, как заявляют авторы одного старого пособия по программированию: *"...прикладные программы, созданные программирующим профессионалом (т. е. "дилетантом" в нашей классификации — Ю. Р.), с точки зрения профессионального программиста зачастую выглядят неуклюжими и неизящными. Но зато они обладают одним общим достоинством — они действи-*

[1] А. Шкроб. Я не любитель, я другой... — Компьютерра, № 24—25, 1998 (http://www.computerra.ru/offline/1998/252/1439/).

тельно работают..."[2]. Дополним данную мысль — это происходит потому, что у дилетантов нет выхода: они пишут программы для себя, и плохо работающие им просто не нужны.

Но согласно хорошему определению, услышанному автором этих строк от одного профессионального фотографа, профессионал отличается от любителя тем, что любитель всегда ищет ответ на вопрос "как", а профессионал — "зачем". Любой любитель или дилетант в конце концов доходит до той стадии, когда ему позарез требуются некоторые функции, которые Delphi (а большинство непрофессионалов использует именно Delphi) сама по себе дать либо не может, либо их осуществление не описано в обычных учебниках и пособиях. Причем среди таких функций есть очень распространенные и необходимые. Вот таким любителям и дилетантам и адресована эта книга. Не ждите от нее последовательного изложения основ объектно-ориентированного программирования (ООП) или построения Windows API. Подобно тому, как можно грамотно писать по-русски, не понимая разницы между существительным и сказуемым, создавать вполне работоспособные программы можно без глубокого знания ООП. Правда, как и в случае грамотности, ваши умения будут ущербными в том смысле, что выйти за рамки конкретных образцов вам будет сложновато, но на основе изложенного в этой книге материала вполне можно научиться делать программы не хуже фирменных — ну, а если вас программирование заинтересует, как самостоятельный предмет, то для этого нужно читать совсем другие пособия и, как правило, не на русском языке.

Зачем все это?

Индия планирует довести экспорт программного обеспечения к 2008 году до 50 миллиардов долларов — почти в два раза больше объема российского экспорта нефти. И, хотя официальные данные по экспорту ПО из России отличаются от этой цифры примерно на два порядка (полмиллиарда в 2004 году, по официальным данным), на самом деле есть основания полагать, что официальная статистика врет — некоторое представление о реальности может дать тот факт, что около 10% shareware в мире делается в России и Украине. К тому же индусам завидовать вообще не очень хочется: они не программисты, а кодеры, рабочие, которые лишь кладут кирпичи в здание, возводимое другими. Самостоятельно в Индии не создано, вероятно, ни одной хоть сколько-нибудь известной программы, мне, по крайней мере, о таких программах слышать не приходилось.

---

[2] Сташин В. В. и др. Проектирование цифровых устройств на однокристальных микроконтроллерах. — М.: Энергоатомиздат, 1990.

"Большое" программирование — индустрия, и без вот таких кодеров-пролетариев обойтись не в состоянии. Но как существование индустрии звукозаписи не отменяет музыкальное творчество, так и производство коммерческого программного обеспечения не противоречит существованию программиста-творца. На западе таких обычно, не особенно разбираясь, именуют "хакерами", но как мы увидим позже, это не совсем точное определение. Но дело не в терминах — такой программист вполне может добиться успеха и признания не будучи коммерсантом: свидетельств тому множество. Главному разработчику нашумевшего браузера FireFox Блейку Россу было всего 19 лет, когда версию 1.0 этого достойного конкурента Internet Explorer скачали за первый же месяц 5 миллионов человек. Большинство вошедших в обиход программных новинок последнего времени сделаны молодыми людьми в том же возрасте или чуть постарше, причем отнюдь не в рамках плановых разработок крупных софтверных компаний. Файлообменные сети Шона Фэннинга, сервис LiveJournal Брэда Фитцпатрика, проигрыватель WinAmp Джастина Франкеля, поисковая система Google Сергея Брина и Ларри Пейджа — список можно продолжать и продолжать. Придется сделать вывод, что время "компьютерных гениев" и "гаражных стартапов" (start up) отнюдь не прошло. Кстати, один из самых успешных отечественных программных продуктов — распознаватель текстов FineReader — также был создан под руководством совсем молодого тогда физика Давида Яна. Кто знает, может ваше имя недолгое время спустя будет звучать в этом ряду?

Но занятие под названием "программирование" — отнюдь не прерогатива только молодого поколения, с пеленок привыкшего держаться за мышку. Существует достаточное количество профессий, в которых молодые имеют преимущество благодаря своей высокой активности и стремлению сделать карьеру, и, наоборот, есть области деятельности, куда молодым до поры до времени дорога заказана. Программирование — как раз то занятие, где возраст не имеет никаких преимуществ, ни в ту, ни в другую сторону — было бы желание. Молодежь легко и быстро осваивает новое, зато человек постарше берет свое обстоятельностью и стремлением докопаться до глубин.

Единственное, что нужно заметить по этому поводу — особенности отечественного технического образования и, главное, содержание деятельности советского инженера были таковы, что он привык все делать сам, и даже "доводить напильником" то, что не было доделано разработчиками. Это с одной стороны хорошо — кругозор намного шире, и такой человек в отдельных случаях может создавать много лучше работающие программы, чем обычный среднестатистический западный специалист. Но в подавляющем большинстве случаев стремление сделать все самому от начала до конца, не затрудняя себя изучением того, что уже сделано до тебя, может только навредить — современные ОС чаще всего этого просто не позволят и правильно сделают. А почему — вы узнаете уже из первой главы этой книги.

Что можно найти в книге?

Предполагается, что читатель уже обладает некоторыми навыками создания приложений в среде Delphi и хочет расширить функциональность своих программ, придать им удобный интерфейс, законченность и профессиональные скоростные качества. Просто поместить на форму компонент Delphi, к примеру, `MainMenu` и написать обработчик щелчка мыши для пункта **Файл | Открыть** может каждый после десятиминутного ознакомления с соответствующим разделом в учебнике. Поэтому мы посвятим одну главу созданию проекта "с нуля", а в дальнейшем сосредоточимся на реализации функций, которые вы часто встречаете в различных программах, но приемы программирования для них в учебниках по Delphi не описаны вообще или вызывают трудности при практической реализации. Некоторые из таких функций могут служить просто для красоты или удобства, другие необходимы, если вы пишете программу не на один раз, а собираетесь ею часто пользоваться и тем паче распространять. Как предотвратить повторный запуск программы? Что такое потоковое чтение файлов и зачем оно нужно? Как правильно написать программу без окна, в виде иконки? Как перехватить нажатие клавиш? Как создать инсталляционный пакет? На подобные простые и не очень простые вопросы мы постараемся ответить в дальнейшем. Причем не удивляйтесь, если вы встретите подробное обсуждение некоей проблемы, которая может показаться совершенно второстепенной, скажем, как убрать с глаз долой текстовый курсор в компонентах-редакторах — хорошо сделанная программа отличается от плохо сделанной в первую очередь подобными мелочами.

В процессе создания программ мы постараемся избежать недозволенных приемов и недокументированных возможностей. В Windows 9*x* вам в принципе никто не запрещает, например, печатать документ прямым обращением к LPT-порту или использовать прямое обращение к регистрам последовательного порта. Но разработчики из MS не рекомендуют этого делать, и поступают совершенно правильно — с нашей точки зрения, хотя бы потому, что при переходе к семейству NT такие программы окажутся совершенно неработоспособными и их придется заново переписывать. В случае же, если мы все делаем, как рекомендовано инструкцией, мы, по крайней мере, снимаем с себя ответственность за происходящее. Это, конечно, шутка — с точки зрения пользователя, программы должны работать, а кто за это несет ответственность — дело десятое. И мы увидим, что при всех наших стараниях, нам все же не удастся следовать рекомендациям разработчиков в полной мере.

Кроме этого, я постараюсь, чтобы при использовании в своих программах примеров из этой книги вам не потребовалось бы использовать ничего, выходящего за рамки свежеустановленной Delphi. За некоторыми отдельными ис-

ключениями — это касается, во-первых, использования готового компонента AsyncFree для работы с COM-портом (но будет также и подробно рассказано, как все сделать без него), во-вторых, я для сокращения записи буду часто пользоваться собственной процедурой преобразования числа в HEX-форму (также будет показано, как ее можно легко заменить стандартными средствами).

В книге вы нередко встретите изложение личных предпочтений автора и его взглядов на ту или иную проблему, включая некоторые особенности Windows вообще. По мнению автора, в Windows есть много неоправданно сложных и просто ошибочных механизмов, для которых можно и нужно обсуждать целесообразность и границы их применимости. Сам автор отчетливо понимает, что его воззрения не есть истина в последней инстанции, и относится к этому в стиле "это мое мнение, и я его разделяю", но старается предоставить читателю самому судить о том, насколько автор прав в тех или иных случаях. В частности, реконструкция логики разработчиков языковых интерфейсов Windows *(см. главу 8)* представляет именно такой случай — вполне возможно, что среди профессионалов найдутся люди, которые считают эту систему стройной и логичной, находя в ней положительные стороны, ускользнувшие от автора этих строк. Прошу читателя иметь в виду, что дискутировать с автором в подобных вопросах совершенно не возбраняется.

Следуя за автором в его критике отдельных сторон Windows, следует иметь в виду одно обстоятельство — как и в любом другом деле, голоса критикующих здесь всегда звучат громче. В защите Windows не нуждается — ее доминирующая роль на рынке ПК сама по себе есть надежная защита. Однако справедливости ради следует отметить, что в продуктах Microsoft есть много положительных качеств — это и реализации некоторых функций Windows (например, буфера обмена), очень неплохая и изначально весьма продвинутая система работы с таблицами в Word, удобная работа с макросами в Office (кажется, вообще не имеющая аналогов), и т. п. Большинство же недостатков Windows (исключая откровенные недоработки) проистекает из того факта, что эту систему создавали и продвигали на рынок в условиях, весьма далеких от тепличных, в которых создавалась и продвигалась, например, MacOS. Обеспечить беспроблемную работоспособность ОС на миллионах *различных* конфигураций "железа" от неизвестных производителей — задача в принципе не решаемая, и, надо сказать, в Microsoft с ней справились не так уж и плохо. Это доказывает тот факт, что Linux при установке создает на порядок больше проблем (пока?), и с этим спорить трудно. Главный недостаток Windows — ее негибкость и ориентация на некоего "среднего" пользователя (так, как его себе представляют в Microsoft) — есть продолжение этих достоинств. Не надо забывать и другое — если бы не Windows, собиравшая и аккумулировавшая в себе все чужие передовые идеи, иногда искажая их до неузнаваемости,

но всегда доводя до состояния, относительно пригодного к использованию даже последними "чайниками", никакой компьютерной революции бы не произошло, компьютер так и остался бы дорогой игрушкой для инженеров и бизнесменов. Мало кто помнит, что правило, согласно которому последняя модель PC всегда стоит $1500, стало соблюдаться только с началом эпохи Windows. Первый IBM PC (с весьма ограниченными возможностями) был выпущен на рынок по запредельной для 1981 года цене примерно $3000, а продвинутая модель IBM PC AT (на основе 286-го процессора) в середине 80-х стоила порядка $5000—7000.

Есть и еще несколько обстоятельств, во многом извиняющих разработчиков из MS. Например, подавляющее большинство современных программ вполне может выполняться на "железе" трех-, пятилетней давности, и уже почти забыто, что еще лет пять-шесть назад в балансе аппаратных и программных средств аппаратные средства решительно отставали — очередные версии ОС требовали последних и достаточно дорогих конфигураций компьютера. Так как никому не хотелось тратить лишние деньги, был период, когда в эксплуатации одновременно находился почти весь парк компьютерного "железа", начиная с клонов IBM PC XT и до первых моделей Pentium MMX. Поэтому гораздо более остро стоял вопрос совместимости софта — пересаживаясь за новые компьютеры, пользователь хотел более комфортабельной работы, но вовсе не желал расставаться с любимыми Norton Commander и Lexicon (автор своими глазами наблюдал этот мучительный процесс). И в этом вопросе корпорации Microsoft, которая не стала ломать ситуацию "через колено", можно сказать только большое-большое спасибо. Но эти требования совместимости всего и вся имели и другую сторону — первые версии Windows оказались весьма далеки от идеала, который обычно обозначается словами "многозадачная операционная система".

Некоторые подробности о Windows и Delphi будут излагаться отдельно, в виде "заметок на полях", но хочется еще раз подчеркнуть, что книга эта ни в коем случае не призвана заменить учебник программирования, справку по Windows API или по работе в Delphi. Вам как минимум потребуется учебник-справочник по языку Object Pascal, приемам работы в Delphi IDE (Integrated Development Environment, интегрированная среда разработки) и основным компонентам. Некоторые подобные издания, а также ссылки на отдельные удачные, по субъективному мнению автора, интернет-ресурсы приведены в списке литературы. Отмечу, что и печатных пособий, и тем более сайтов, посвященных Delphi, так много, что никакой подобный список не может претендовать на полноту, так что эта задача даже и не ставилась. В качестве основного источника сведений по Windows API вполне достаточно официального сайта Microsoft MSDN [14] или даже встроенной в Delphi справки (файл win32.hlp, который располагается в папке C:\Program Files\Common Files

\Borland Shared\MSHelp\; может быть вызван и через меню **Пуск | Программы | Borland Delphi 7 | Help | MS SDK Help files | Win32 Programmer's Reference** — правда, в некоторых отношениях эта справка устарела). Есть и множество русскоязычных ресурсов, той или иной степени полноты, представляющих собой простой или комментированный перевод материалов с сайта Microsoft MSDN. Наиболее полные и систематизированные переводы принадлежат Владимиру Соковикову [16,18], другие вы без труда отыщете сами, просто набрав в Яндексе название той или иной функции API.

Знания и умения

Есть знания и умения. Обучение чтением теоретических курсов — приобретение знаний, как таковых — изобретение нового времени и составляет основное содержание классной системы образования, возникшей в период Реформации. До этого обучение происходило исключительно на личных примерах и в процессе реальной работы. И сейчас в тех областях, где закончивший некий курс обучения должен выйти в мир, обладая именно практическими умениями (как, например, в медицине), основное содержание обучения составляет практика. А есть области, в которых теоретических знаний не требуется вообще или нужно настолько мало, что без специального изучения теории можно обойтись — например, вождение автомобиля или, скажем, плотницкие работы. С другой стороны, такие области, как теоретическая физика или математика, наоборот, в значительной степени основаны именно на знаниях (хотя тоже далеко не полностью). Любопытно, что программирование, при том, что оно опирается на самые абстрактные области человеческого знания, по сути своей — типичное умение, ремесло. Можно спокойно начинать программировать, обладая математическим багажом в пределах средней школы — необходимые знания приобретаются в процессе отработки умений. А умения приобретаются только на практике.

Автора этой книги часто спрашивали: "Как можно научиться работать на персональном компьютере?" Ответ на этот вопрос единственный: надо его приобрести. Остальное сделает ваше желание научиться — если оно имеется, конечно. Точно так же и навыки программирования нельзя приобрести при чтении даже самых умных книжек. Включите компьютер, подберите себе кресло поудобнее и начинайте работать. Только если вы хотите стать Настоящим Программистом, не забывайте, что это очень особая профессия. Сама специфика деятельности программиста такова, что здесь не могут удержаться люди, привыкшие к делению жизни на работу с 9 до 17 и развлечения все остальное время. Этой особенности профессии посвящена добрая половина программистского фольклора:

У жены программиста спросили:
— А как он за тобой ухаживал?
Жена, после минутного раздумья:
— Компьютер показал...

Но, чтобы писать вполне работоспособные программы, Настоящим Программистом становиться вовсе необязательно. В этой книге я ориентировался на тех, кто делает программы, предназначенные для практической работы, а не для развлечения, постаравшись сосредоточиться на том, чтобы все "навороты" служили одной цели: сделать программу удобной и приятной для пользователя. Впрочем, грань между серьезным и развлекательным провести трудно. Никому сами по себе не нужные "приколы", несомненно, иногда неплохо помогают иллюстрировать тот или иной прием. Да и одно из основных наших приложений — SlideShow, которое мы будем создавать на протяжении почти всей книги, по своему назначению есть программа в основном развлекательная.

Кто такие хакеры?

Несколько слов об упомянутом ранее понятии "хакер". В компьютерной прессе не устают подчеркивать, что настоящий хакер — это не преступник, который рыщет по Интернету в поисках способов взломать сайт Пентагона или своровать базу кредитных карт клиентов онлайнового аукциона eBay. Точнее, часть таких преступников-"крякеров" — и любителей и профессионалов, занимающихся промышленным шпионажем — может относиться к хакерам, но уже обратное неверно: далеко не все хакеры, и даже не заметная их часть — преступники. Однако, когда доходит до объяснений, а кто же собственно такой хакер, дело обычно ограничивается следующей характеристикой: это человек, помешанный на компьютерах, желающий досконально знать, как все в них устроено, который отличается виртуозным владением соответствующими инструментами, позволяющими ему выделывать с компьютерами и сетями разные необычные штуки. Между тем, это определение является недостаточным: кроме виртуозного владения компьютером, принадлежность к хакерскому течению предполагает еще и определенный склад ума и особую социальную философию левацкого толка. Развитие хакерской субкультуры связано с Массачусетским технологическим институтом и появлением в 1961 году компьютеров PDP-1. Позднее, с появлением первой сети ARPAnet и, в дальнейшем, Internet (Интернет), хакерство интернационализировалось. Главными практическими результатами деятельности представителей хакерской субкультуры для всего остального мира стало рождение ОС Unix и языка C, а позднее сообщества "свободного софта" и соответствующего мировоз-

зрения. Характерной особенностью Unix (как и ее знаменитого клона — Linux) было то, что она была сделана полностью в рамках частной инициативы, без какого-то заказа — в дальнейшем это стало основным отправным пунктом идеологии "свободного софта" вообще, даже когда подобные проекты делаются в коммерческих целях, дух все равно тот же: "потому что интересно".

Основным в хакерстве является именно вот это либертарианское, отчасти анархистское мировоззрение, совпавшее по времени возникновения с расцветом движения хиппи со всеми сопутствующими атрибутами — цветочками в волосах, драными джинсами и способностью работать когда хочется, а не когда этим занимаются все остальные. Хакеры — это технологические хиппи, те из них, кто обрел, как ему показалось, именно в компьютерах желанную свободу от этого кошмарного мира "власти чистогана". Хакерское движение, как и молодежные бунты 60-х вообще, вне зависимости от практических результатов, сделало очень много хорошего. Прежде всего, в политическом смысле они стали действительно реальной оппозицией консервативной бизнес-верхушке — заняли то место, которое ранее безуспешно пытались занять коммунисты-догматики. У них это получилось потому, что, в отличие от политиков, они не болтали и не устраивали революций, а делали реальные дела ("Не пишите манифесты, пишите код!"), и чуть ли не впервые в истории показали всем, что, оказывается, серьезные вещи можно делать и так — без корпоративных структур, без формального подчинения, на основе добровольности и открытости. Современный компьютерный — и не только компьютерный — мир очень и очень многое у них заимствовал, как в плане методологии ведения разработок, так и в плане конкретных идей.

Эту небольшую политинформацию я провел вот зачем: хакер — совсем не синоним программиста-виртуоза, хакерство — течение социальное, а не профессиональное, поэтому термин этот без нужды примерять к себе не следует. Далеко не все профессиональные программисты и даже взломщики программ называют себя хакерами и являются ими. Вообще большинство из тех, кто профессионально "ломает" программы и средства защиты, являются серьезными учеными-криптографами, обладателями всяких академических званий и степеней, или, по крайней мере, составили себе имя публикациями на соответствующие темы. И взломом они занимаются не просто так, а потому что без их труда невозможно правильно оценить эффективность работы тех или иных алгоритмов. Согласитесь, в этом подходе есть некоторые отличия от очередных "подвигов" взломщиков сайтов, по какой-то странной причине называющих себя "хакерскими группами". Тем более, совсем не каждый, кто считает, что хорошо владеет компьютером, вправе считать себя хакером. Добавлю специально для тех, кто усматривает некую романтику в образе "крякера"-подпольщика: все громкие деяния, типа кражи исходного кода

Windows или остроумно написанного вируса, заразившего очередной миллион компьютеров, относятся к настоящему хакерству, примерно так же, как поведение водителя-лихача к манере вождения шофера-профессионала.

Как пользоваться книгой

Но вернемся к тому, с чего начали — играть лучше в практически полезные дела. Исходя из этих соображений, книга эта построена так, чтобы каждый пример, по возможности, являлся неким законченным проектом (или его частью), который может пригодиться на практике. Некоторые примеры (но не все, конечно) будут даны в развитии — вы увидите, как приложение постепенно отлаживается, с промежуточными экспериментами по использованию того или иного приема или функции. Все примеры (по главам) имеются на прилагаемом диске в том виде, в котором они описаны в тексте. Эти примеры в идеале нужно изучать следующим образом — вы берете исходный текст примера, такой, какой он есть к началу соответствующего раздела, и повторяете все, что описано в этом разделе по ходу мысли автора. А потом сравниваете то, что получилось, с окончательным вариантом, который имеется на диске. Если варианты совпали — замечательно, если нет — при сравнении легко установить, что именно (и у кого именно) не так. Все примеры тестировались на совместимость с Windows 98 и Windows XP (в последнем случае без установленных сервис-паков). При обнаружении каких-то особенностей в работе под указанными ОС, это оговаривается в тексте. Отразить в оглавлении все имеющиеся в книге приемы возможности, конечно, не было, поэтому в конце книги размещен предметный указатель в форме FAQ, где перечислены по возможности все имеющиеся в книге ответы на вопросы "Как?".

Запускать проекты прямо с диска неудобно, вам придется долго настраивать среду и вы не сможете вносить изменения, поэтому их нужно сначала перенести на жесткий диск вашего компьютера. При переносе, возможно, придется изменить путь к папке с проектом, который указан в исходном DSK-файле. Если вы устанавливали Delphi в режиме по умолчанию и папка Program Files находится на диске C:, то можно ничего не менять — просто скопируйте папку, относящуюся к данной главе, в папку Projects. В этом случае путь будет такой: C:\Program Files\Borland\Delphi7\Projects, и он зафиксирован в DSK-файлах проектов на диске. В противном случае можно вовсе не копировать DSK-файл в новую папку, но при этом вы также потеряете и все настройки. Лично я открываю DSK-файл в Блокноте и вношу исправления через пункт **Заменить | Заменить все**. Перенос можно осуществить и раздельным копированием через пункт **Save as** сначала проекта, потом каждого модуля программы по очереди.

Перед тем как приступить к делу, автор считает себя обязанным выразить благодарности:

- Валерию Васильевичу Фаронову — за его замечательные учебники по Pascal и Delphi;
- Евгению Сергеевичу Голомину, глубоко верующему человеку, за выложенные им для всеобщего ознакомления исходные тексты к прекрасной программе "Опечатка" (**http://come.to/golomin**);
- всем без исключения посетителям форумов по программированию в Рунете — без них эта книга вообще бы не увидела света.

Со всеми вопросами и пожеланиями пишите на **revich@homepc.ru**.

ГЛАВА 1

Ликбез

Некоторые сведения о программировании, Windows и Delphi

> Создайте систему, которой сможет пользоваться каждый дурак, и только дурак захочет ею пользоваться.
>
> *Принцип Шоу*

> Компьютерная программа выполняет то, что вы приказали ей делать, а не то, что вы бы хотели, чтобы она делала.
>
> *Третий закон Грида*

Современный персональный компьютер — устройство необычайно сложное. Причем рядовой ПК образца 2004 года отличается от IBM PC двадцатилетней давности гораздо больше, чем, к примеру, последняя модификация Ford Focus от легендарного Ford T образца 1913 года. И дело тут не в самой по себе скорости работы, которая возросла примерно на три порядка. Гораздо важнее принципиально возросшая функциональность — ни о какой 3D- и даже обычной многоцветной 2D-графике тогда и речи не шло, голосовые интерфейсы существовали разве что в фантастических романах, а предсказать нечто подобное Интернету, да еще и мобильному, фактически не смог никто. Если продолжить аналогию с первыми автомобилями, то современный ПК в сравнении с IBM PC скорее следует уподобить, если и не реактивному истребителю, то, по крайней мере, пассажирскому лайнеру.

Казалось бы, управлять такой сложной машиной — учиться и учиться. Но ПК не обосновались бы настолько прочно в наших домах и офисах, если бы разработчики не придумали программные средства, сводящие сверхсложные операции к двум-трем щелчкам мыши. Причем на сегодняшний момент сложилась довольно парадоксальная ситуация: разнообразие компьютерного "железа" настолько велико, что *правильно* подобрать комплектующие и *правильно* настроить современный ПК невозможно без определенного багажа специальных знаний. А вот технологии программирования ПК, наоборот, предельно упростились. Правда, это в полной мере справедливо только для

случая создания пользовательских программ — системные программы, разумеется, требуют для своего создания специальных знаний. Показательно, что в популярной книге С. В. Зубкова "Assembler для DOS, Windows и Unix" [13], выдержавшей несколько переизданий, программированию под Windows и Unix посвящено менее 100 страниц из почти 700 — настолько это простое по сути занятие.

Заметки на полях

Попробую пояснить эту парадоксальную мысль. Для начала хочу подчеркнуть разницу между понятиями "сложный" и "громоздкий" — очень простая по сути программа может быть очень громоздкой и наоборот. Например, расшифровать заголовок файла, содержащего изображение в формате BMP, и воспроизвести на экране содержащуюся в нем картинку — достаточно сложное занятие. Программа, которая это делает из-под DOS, окажется и сложной и достаточно громоздкой, учитывая еще тот факт, что для воспроизведения True Color придется где-то искать и присоединять к программе драйвер имеющейся в наличии видеокарты. Излишне говорить, что в силу большого разнообразия "железа" последняя задача может доставить и программисту и пользователю массу хлопот. А вот в Windows ни один из этих вопросов не стоит вообще — все драйверы уже установлены заранее, а для вывода на экран картинки формата BMP не требуется даже знать, что у нее есть какой-то там заголовок — даже на ассемблере это делается вызовом одной-двух функций. Вместе с тем, если простейшая ассемблерная программа под DOS может состоять из одной-единственной команды процессора, то любая программа под Windows обязана содержать некий минимум команд, так что решение упомянутой ранее задачи отображения картинки по объему кода может даже превысить DOS-вариант. Что делает Windows-программы более громоздкими, но отнюдь не более сложными, потому что этот необходимый минимум повторяется из программы в программу и заново его "изобретать" каждый раз не требуется (см. [13]).

Так получилось потому, что большинство функций в современных операционных системах скрыто за оболочкой, носящей название *пользовательские программные интерфейсы* (Application Programming Interface, API). Собственно процесс программирования сводится к тому, чтобы вовремя и правильно вызвать нужную функцию API. Поэтому такие крайне сложные для "обычного" программирования операции, как, к примеру, вывод на экран полноцветной графики или создание текстового редактора с поддержкой форматирования, шрифтов, операций с буфером обмена и прочих необходимых свойств, сводится к вызову нескольких функций API. Довольно большой прогресс по сравнению с прерываниями DOS, которые программисты в большинстве случаев старались обойти, не видя необходимости в лишних прослойках между программой и BIOS или "железом", не так ли?

Однако на этом разработчики современных систем не остановились. С помощью средств так называемого *визуального программирования* процесс вызова функций API максимально автоматизирован. Вам уже не нужно выполнять рутинную работу по написанию кода программы целиком. Достаточно

перетащить мышью нужный компонент на форму, и соответствующий код включается в текст программы автоматически. В результате процесс программирования стал воистину творческим занятием: всю "грязную" работу берет на себя компьютер (точнее, выбранный пакет программирования), а вам остается только правильно обработать возникающие в программе события. Специальных знаний об устройстве компьютера и построении ОС здесь требуется не больше, чем для обычного квалифицированного пользователя.

Разумеется, как в любом другом деле, за удобства приходится платить: вы полностью привязаны к существующим программным интерфейсам, которые писали для вас добрые дяди из Microsoft и других фирм-производителей системного софта, и шаг вправо-влево тут означает если и не расстрел, то, во всяком случае, необходимость приобретения *очень специальных* знаний. Однако такой выход за пределы стандартных функций в подавляющем большинстве случаев и не требуется: "добрые дяди" постарались предусмотреть по максимуму все, что требуется среднестатистическому пользователю. Другое дело, что поиск нужной функции и способа ее правильного использования может быть очень непростым занятием — ведь далеко не все можно сделать, перетаскивая компоненты мышью на форму. Именно в этом деле и призвана помочь книга, которую вы держите в руках.

О Delphi и Windows

Кстати, а почему именно Delphi? В принципе все современные пакеты визуального программирования позволяют делать одно и то же, а если какие-то вещи делать удобнее в одном пакете, а другие — в другом, то никто не мешает использовать их совместно. Скажем, библиотеки VCL (Visual Components Library) от Borland являются общими для Visual C++ и Delphi и написаны в основном на Object Pascal, а Windows API, наоборот, большей частью написаны на C (или C++), что не мешает использовать их в любой среде. Но Delphi, безусловно, является на сегодняшний день наиболее универсальной средой программирования, которая позволяет без лишних сложностей создавать как самые простые пользовательские программы, так и навороченные профессиональные пакеты.

Эта книга написана с расчетом на выполнение примеров в среде Delphi 7.0. Седьмая версия — последний релиз Delphi для платформы Win32, на которой основываются все версии Windows от 95-й до XP. В настоящее время Microsoft переходит на платформу .NET — на ней будет полностью основана Windows Longhorn, которую планируется выпустить на рынок в 2006 году. Частично на .NET переходит уже обновленная 64-разрядная Windows XP, готовящаяся к выходу в 2005 году. Между прочим, базовый язык программирования для платформы .NET под названием C# разрабатывает Андерс

Хейлсберг — человек, которому корпорация Borland во многом обязана самим своим существованием.

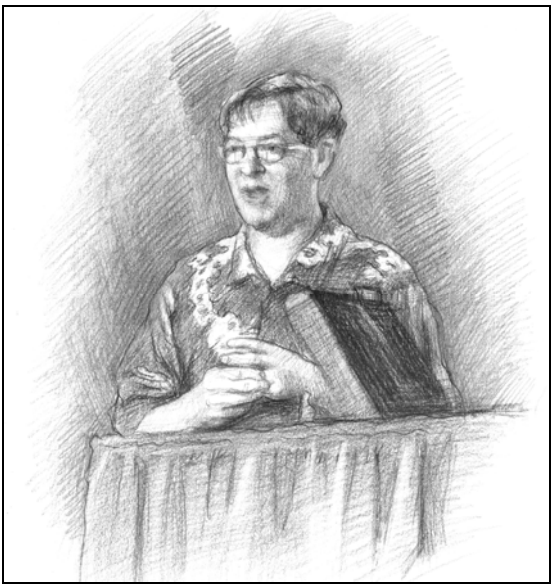


Рис. 1.1. Андерс Хейлсберг (рисунок Александры Дрофиной)

Андерс Хейлсберг (Anders Heilsberg) — создатель Turbo Pascal и один из главных архитекторов Borland Delphi с момента ее возникновения, человек, разработавший продукты, выведшие корпорацию Borland в ряды ведущих поставщиков программного обеспечения. Также главный архитектор языка C# для платформы Microsoft .NET. О себе Андерс рассказывает так (из интервью журналу "Домашний компьютер", #1, 2004):

Я родился в 1960 году в Копенгагене, Дания. Свое тихое и милое детство я провел в пригороде Копенгагена. Потом учился на инженера по электротехнике в Техническом университете Дании. В 1987 году переехал в США, а в 1994 — женился. Живу в Сиэтле. С 1983 по 1996 я работал в компании Borland, а теперь в Microsoft. В 1979 я основал компьютерную компанию в Дании под названием PolyData. Это было время, когда персональных компьютеров еще не существовало. Мы продавали компьютерные комплексы и писали для них программное обеспечение. Я написал такие вещи, как ассемблер, дизассемблер, небольшую операционную систему и несколько расширений для Microsoft ROM-Basic. Моим самым первым большим проектом стал компилятор с языка Pascal и редактор, который мог заменить ROM-Basic. После этого я написал еще одну реализацию Pascal для операционной системы CP/M. Она называлась PolyPascal. В 1983 году мы объединились с

ребятами, которые только что основали компанию Borland, они лицензировали наш компилятор Pascal, добавили туда свой собственный редактор и назвали все это Turbo Pascal. Я помню, как думал, что они сумасшедшие: эти парни продавали новый продукт по цене 49 долларов 95 центов, в то время как он стоил 500 долларов! Но достаточно быстро выяснилось, что я ошибался — Turbo Pascal стал очень популярным. Мы продали его столько, что в начале было невозможно представить.

Для того чтобы читатель немного сориентировался, приведем краткую сравнительную характеристику различных сред обработки от Borland, базирующихся на языке Pascal.

В начале всего был Turbo Pascal 1.0, вышедший на рынок 20 ноября 1982 г. Об искусстве Андерса Хейлсберга может говорить тот факт, что интегрированная среда разработки, встроенный редактор и библиотека времени выполнения умещались в файле turbo.com размером 33 280 байт. Правда, эта версия сама могла делать только COM-программы (если кто помнит, то в DOS существовал такой формат исполняемых файлов, отличавшийся от привычного EXE тем, что занимал только один 64-килобайтный сегмент памяти), но зато работала она на медленных ПК того времени очень быстро и давала очень компактный код, благодаря чему сразу вывела компанию Borland в лидеры. В дальнейшем эволюция Turbo Pascal привела к версии 7.0 (1992) — до сих пор популярному средству разработки программ под DOS. Благодаря непревзойденной по удобству среде разработчика (Integrated Development Environment, IDE) и простому для освоения языку, Turbo Pascal стал особенно популярным в непрофессиональной и полупрофессиональной среде. Строго говоря, последняя версия Turbo Pascal, как такового, называлась 6.0, седьмая версия называлась Borland Pascal и включала в себя две разных среды: Borland Pascal for DOS 7.0 (базирующийся на шестой версии с некоторыми существенными доработками) и Borland Pascal for Windows. Если вам удастся достать какое-нибудь пособие по последнему (например, раритетную книгу В. В. Фаронова "Паскаль и Windows"), то вы увидите, что разработка приложений под Windows "вручную" — довольно громоздкое занятие, требующее хорошего знания приемов объектно-ориентированного программирования (ООП). Чтобы облегчить это занятие, в Borland (под руководством все того же Хейлсберга) к 1995 году была разработана Delphi 1.0 — первая визуальная среда программирования, ориентированная на разработку 16-разрядных приложений под Windows 3*x*.

Начиная же с версии 2.0, Delphi ориентировалась на 32-разрядные версии Windows (95 и выше), использующие платформу Win32. Все использующиеся на практике на момент выхода этой книги версии Windows ориентируются

именно на эту платформу, поэтому в подавляющем большинстве случаев вам вообще не нужно думать о том, какая версия Windows у вас установлена. И все же многие API могут быть специфичными для того или иного семейства ОС либо той или иной версии. Поэтому работу программ, особенно тех, которые предназначены для распространения, следует проверять под различными версиями. Для этого удобно установить на компьютере сразу несколько систем, благо XP это позволяет без лишних сложностей.

На момент написания данной книги уже имеются версии Delphi 8.0 и Delphi 2005, ориентированные на .NET. Для совместимости в комплект поставки Delphi 8.0 входит Delphi 7.1, подобно тому, как в Borland Pascal входил Turbo Pascal, а Delphi 2 комплектовалась версией 1. Разница только в том, что вторая версия Delphi в свое время вышла с опозданием, уже после выхода Windows 95, а восьмая версия отчасти опережает события. Заметим, что наименование "Delphi 8.0" следует признать неудачным — это другой продукт для другой платформы (изменили же когда-то марку Turbo Pascal на Borland Pascal — почему бы и не следовать этому принципу и в дальнейшем?). Поэтому большинство примеров, которые вы встретите в этой книге, в среде Delphi 8.0 (и выше), скорее всего, просто не будут компилироваться — но сами по себе готовые программы будут пригодны еще долгие и долгие годы, в будущей Longhorn декларирована совместимость с Win32 (подобно тому, как версии Win95/98/ME были совместимы с DOS-программами).

Что же касается более ранних версий, то большинство примеров из этой книги совместимы с версиями Delphi, начиная с 4.0, в крайнем случае 6.0. При установке следует позаботиться, чтобы у вас оказались установленными файлы справки по Win32. Эта справка, к сожалению, практически не обновлялась с версии 3.0, так что можно просто скопировать файл win32.hlp в отдельный каталог и пользоваться им автономно, независимо от версии пакета.

Примечание специально для тех, кто только осваивает программирование: на пиратских развалах якобы появилась русифицированная версия Delphi. Ее устанавливать *не следует*, и не только потому, что она неизвестного происхождения, но и потому, что все без исключения пособия и интернет-ресурсы оперируют с английской версией, и в результате вы потратите гораздо больше времени на обратный перевод русских названий пунктов меню и текстов сообщений, чем на то, чтобы один раз выучить английские.

> Несколько слов о том, каковы особенности работы программ под теми или иными версиями Windows. Главное отличие семейства 9*x* (95/98/ME) от семейства NT (NT/2000/XP) заключается в реализации многозадачности. В забытых ныне 16-разрядных версиях Windows 3.*x* многозадачность называлась кооперативной — фактически весь процесс работы заключается в выполнении последовательного ряда событий Windows, и пока одно событие

не обработается, все остальные вынуждены ждать своей очереди. Таким образом, малейшая ошибка в одном приложении полностью подвешивает всю систему. Все программы, в том числе служебные, видимы друг для друга, модуль, содержащий ошибки обращения к памяти, может легко испортить ее содержимое, принадлежащее другому процессу.

В версиях семейства NT реализована настоящая — вытесняющая — многозадачность (напомним, что NT появилась раньше, чем 95-я). Такая система распределяет процессорное время и память между процессами (process) и потоками (thread)[1], как будто бы каждый из них выполняется в отдельном компьютере, поэтому "зависшая" программа теоретически не оказывает влияния на все остальные. Но за это приходится платить — в первую очередь тем, что прямой доступ к "железу" практически исключен, а если вы попытаетесь этот запрет обойти тем или иным способом, то такая попытка ничем хорошим ни для вашей программы, ни для системы не кончится. То есть требования к качеству программного кода сильно повышаются и можно смело утверждать, что все случаи "обрушения" Windows XP связаны именно с неправильно (как говорят программисты — "криво") написанными приложениями.

А в версиях 9*x* в целях совместимости с 16-разрядными приложениями (включая DOS-программы) реализовано что-то вроде промежуточного варианта. Все 32-разрядные прикладные программы выполняются в соответствии с моделью вытесняющей многозадачности. На уровне таких приложений формально все работает независимо, как и NT. Однако в области модели памяти (в основном из-за требований совместимости с 16-разрядными приложениями DOS и Win3*x*) есть серьезные дыры. Как и в 3*x*, ничто не может помешать программе, содержащей ошибку обращения к памяти, произвести запись в адреса, принадлежащие системным DLL, и вызвать крах всей системы.

В Windows 9*x* ситуация, когда выполняющийся поток все никак не освободит ресурсы компьютера, случается не так уж редко. Даже формально правильное приложение может быть источником неприятностей: так, многим знакома картинка, когда ресурсоемкая программа (архиватор, файловый менеджер при копировании больших объемов информации, или, например, поисковая программа) настолько "оккупирует" все ресурсы компьютера, что даже картинка рабочего окна не успевает прорисовываться. Один из механизмов таких задержек — но не единственный, конечно — связан с тем, что многие подобные процессы выполняются через 16-разрядные функции, а

[1] Сами по себе процессы ничего не делают, а лишь предоставляют ресурсы и контекст для выполнения потоков. Любой процесс должен иметь, по крайней мере, один поток, который и выполняет код процесса. Причем планирование переключения задач в Windows осуществляется на уровне потоков, а не процессов.

только один поток может обращаться к 16-разрядным DLL в каждый момент времени, потенциально затормаживая другие процессы, которым нужен к ним доступ. Другой — с неправильным распределением приоритетов одновременно выполняющихся потоков. Бороться с такими неприятностями можно только внутри самой программы, периодически искусственно вызывая обработчик системных сообщений, но самое правильное — по возможности не использовать потенциально "тормозных" процедур вообще.

О пользовательских интерфейсах компьютерных программ

Крупнейший в мире специалист по компьютерным интерфейсам Джеф Раскин (Jef Raskin, 1943—2005 гг.) говорил примерно так (за точность цитаты не ручаюсь): *"Пользователь обычно не понимает, насколько ему неудобно, механизм обратной связи от потребителя к производителю не работает, и единственное, что может заставить проектировщика создавать по-настоящему хороший интерфейс — это его совесть"*. Я не могу тут не остановиться на истории пользовательских интерфейсов, потому что непонимание, насколько неудобно работать с современными программами, к сожалению, характерно для программистов в еще большей степени, чем для пользователей. Невозможно делать удобные программы, если вы не очень хорошо представляете, с чем вам придется иметь дело.

Точкой роста почти всех идей, реализованных впоследствии в так называемом графическом интерфейсе пользователя (Graphic User Interface, GUI), стал Xerox Palo Alto Research Center (Xerox PARC), возникший на рубеже 60—70 годов. Любопытно, что одним из его основателей, а с сентября 1970 года — и руководителей, стал Боб Тейлор (Robert W. Taylor), который пришел в Xerox из DARPA, где в 1966—69 гг. возглавлял проект ARPAnet, предшественника Интернета. Одной из идей, родившихся в этом центре, была так называемая парадигма WIMP (Windows, Icons, Menus, Point-and-Click — "окна, пиктограммы, меню, укажи и щелкни"), которая переросла позже в концепцию GUI и продолжает эксплуатироваться в настоящее время. Эти разработки связывают с именем Алана Кея (Alan Key), также известного, как автора SmallTalk (первого объектно-ориентированного языка программирования). В 1975 году в Xerox была начата разработка нового проекта, закончившегося в апреле 1981 года представлением системы Xerox 8010 Professional Workstation, более известной под торговой маркой Xerox Star. Именно с нее и началось победное шествие GUI, внедрение которого, как промышленного стандарта, было произведено в 1988 году усилиями Sun Microsystems, Xerox и AT&T. Xerox Star мы также обязаны такими вещами, как иконки, окна, меню, двухбайтовые многоязычные шрифты (современный Unicode — см. *главу 8*), режим

WYSIWYG и др. За подробностями я отсылаю читателя к [25], а здесь нам важно, что распущенный после рыночного провала Xerox Star примерно в 1983 году коллектив ее разработчиков в основном оказался в Apple — за важными исключениями, о которых поговорим отдельно.

В числе прочих инноваций в интерфейсе Star было впервые использовано представление экранного пространства в виде "рабочего стола" (Desktop). Эта метафора основывается на том, что пользователь якобы не имеет представления о существовании, скажем, программы под названием "текстовый редактор", а просто "открывает документ". Альтернативой является концепция "инструментов" (Tools) или "приложений" (Applications), где пользователь запускает нужный инструмент (приложение), и с его помощью открывает документ, причем тип его идентифицируется обычно по расширению имени файла. Ответственность за результат, например, загрузки файла, содержащего изображение, в текстовый редактор при этом целиком ложится на пользователя. Разработчики Star сознательно шли на потерю универсальности, присущую инструментальной модели, в то же время предупреждая об ограниченности сферы применимости метафоры "рабочего стола" исключительно офисными системами. (Дуглас Энгельбарт, изобретатель мышинного интерфейса, позднее иронизировал: *"Весь мир был увлечен идеей "офисной автоматизации", будучи уверен, что "настоящие пользователи" компьютеров — это секретари, чьи задачи необходимо автоматизировать."*)

В 1978 году упомянутый ранее Джеф Раскин работал в Apple, где начал новый проект под названием Macintosh, однако уже в 1982 г. разругался с Джобсом и покинул фирму. Впрочем, это неудивительно — ни один из его проектов так и не стал успешным в коммерческом смысле, притом что выдвинутые им теории обязательно изучают на всех соответствующих курсах в мире. Основная заслуга Раскина в том, что он одним из первых осознал: самое важное ментальное ограничение человека — ограниченность внимания. Фокус внимания у человека один. Интерфейсы, переключающие на себя внимание человека, — это одна из ключевых причин неэффективности взаимодействия с машинами.

Заметки на полях

Вот пример: когда мне диктуют телефонный номер, я хочу сразу записать его, пока он не вылетел из памяти. В типичной современной ОС для этого требуется запустить специальное приложение (по крайней мере, переключиться на него), после чего создать новый контакт (это же надо еще знать, что то, что вам нужно, называется именно "контактами", см. *главу 16*), найти на появившейся форме подходящее поле, вписать туда номер... Автор этих строк и в бумажной-то телефонной книжке не вел записи по алфавиту, предпочитая для ускорения записывать их "внавал", и сейчас фиксирует номера, адреса и прочие полезные сведения в маленькой программке в стиле картотеки из Windows 3*x*. И даже забросил ведение адресной книги в The Bat! — слишком много усилий приходится

предпринимать для систематизации записей и извлечения оттуда нужного адреса, проще найти корреспондента через "Поиск".

Раскин выступил с радикальным предложением: отказаться от зоопарка разных приложений и заменить его на единую рабочую среду, которая всегда ведет себя одинаково. В такой среде, например, нажатие клавиш всегда приводило бы к вводу текста, поэтому человеку, желающему записать телефон, достаточно было бы просто его набрать. Разумеется, определить введенный номер в "правильное" место нужно было бы и в этой среде, но в ней это можно сделать после ввода номера, что принципиально удобнее. В результате Раскин придумал интерфейс, суть которого состоит в следующем: все, что хранится в компьютере, представляет собой единый документ, отдельные приложения — это команды и модули, запускаемые пользователем или подключающиеся автоматически. Причем все управление осуществляется с помощью клавиш — мыши Раскин не признавал, действительно, ведь для работы с текстом мышь в принципе не требуется, так зачем лишние сущности? Несмотря на понятные ограничения (а кроме инструментальной модели, предоставляющей пользователю полный доступ к "потрохам" системы, вплоть до ее перепрограммирования, любая другая модель является ограниченной и пригодной лишь в определенной области) эта система для тех, кто работает именно с документами, была бы, вероятно, более удобной. И в конце 80-х она даже была осуществлена на практике в компьютере Canon Cat, который разошелся в количестве 20 тыс. экземпляров, но затем проект был свернут.

Подход Раскина можно интерпретировать так: вся среда в компьютере есть одно универсальное приложение. Легко заметить, что черты этого подхода можно встретить, например, в MS Office, в попытках интегрировать все действия пользователя — или большинство их — в единую среду. Я не знаю, что могла бы представлять доведенная до логического завершения концепция Раскина (в Canon Cat был осуществлен фактически лишь текстовый редактор с функциями электронных таблиц), но есть сильное подозрение (основанное на практическом опыте), что ни один программист, даже такой корифей, как Джеф Раскин, не сумел бы сделать интегрированную среду во всех нюансах так, чтобы в ней было бы удобно работать всем без исключения.

Но вернемся к существующим интерфейсам. Для платформы PC (уже превратившейся к тому времени в Wintel) "рабочий стол" был реализован только спустя пятнадцать лет — с появлением Windows 95. Казалось бы, было время все продумать и учесть недоработки Xerox и Apple, но в Windows была предпринята эклектичная попытка сохранить все преимущества инструментальной модели, как наиболее гибкой, но рассчитанной на специально обученных пользователей, и в то же время склонить на свою сторону армию "чайников", обучаться не желающих или не имеющих возможности. В результате пророческие слова Раскина о том, что *"пользователь обычно не понимает, на-*

сколько ему неудобно" относятся к Windows, как ни к чему другому. Буквально любое незнакомое действие в Windows, несмотря на все попытки унификации и стандартизации, требует от "чайника" консультаций с более продвинутым собратом по несчастью[2]. Непоследовательное использование Desktop-метафоры может приводить к просто катастрофическим последствиям: т. к. первичным признаком для отнесения файла к тому или иному типу в Windows по-прежнему служит расширение его имени, то при неправильном переименовании файл может быть просто потерян для непросвещенного пользователя. При попытке скрыть расширения, как это происходит по умолчанию в Windows, часто может возникать забавная ситуация, когда в одной папке оказываются несколько абсолютно одинаковых с виду значков, что приводит к недоразумениям — одно дело послать по электронной почте многомегабайтный TIFF, другое — компактный JPEG, а ведь все иконки, относящиеся к изображениям (если вы не зададите специально обратного), будут связаны с одним приложением, и потому и обозначатся одинаково.

Рискну предположить, что правильным решением всех этих проблем был бы выпуск множества относительно автономных модификаций одной и той же ОС с интерфейсами, "заточенными" под нужды бухгалтера, секретаря, технического писателя, ученого, фотохудожника, журналиста, ребенка, наконец, с возможностью установки их в любой комбинации и переключения между ними. Глядишь, и модель Раскина вполне тогда вписалась бы в коммерческие продукты, и любителям прыгающих по экрану кнопок или "скрепок-помощников" тоже было бы где развернуться...

> Резюмируем то, что сказано ранее, попробовав сформулировать основную цель любого интерфейса — не только компьютерного. Подавляющее большинство действий совершается человеком бессознательно. Часть подобных действий человек умеет совершать от рождения, таких, как глотательные рефлексы или отдергивание руки от горячего предмета. Среди этих действий есть очень сложные *программы поведения* — например, половое поведение или стайные инстинкты. Другая часть — инстинкты и программы, приобретенные в процессе взросления и обучения, это навыки ходьбы, речи, письма и счета, или социальное поведение: стыдливость, способность к состраданию и т. п. Наконец, сравнительно небольшую часть действий составляют те, что контролируются сознанием. Однако именно эти действия требуют повышенного внимания, они человека, пользуясь расхожим выражением, "поглощают целиком". Ни один человек не может обдумывать две

[2] Причем по мере развития одни неудобства устраняются (так, реализация Plug&Play в Windows XP принципиально лучше всех предыдущих версий), зато множатся другие (в той же XP — непонятные проблемы с языком, атрибутами папок, активацией и т. п.).

вещи сразу. Поэтому любое обучение всегда преследует одну и ту же цель — перевести некие действия в область бессознательного, сделать так, чтобы они выполнялись автоматически. Именно на это направлены бесконечные тренировки спортсменов, балерин, специалистов по рукопашному бою, водителей автомобилей. Более того, на это также направлены и процессы умственного обучения — студентов заставляют решать учебные задачи с тем, чтобы в процессе практической деятельности нужное решение находилось как бы "само", на основе приобретенного опыта.

Интерфейс GUI менее всего приспособлен к эффективному обучению. Производители затвердили словосочетание "интуитивно понятный" применительно к интерфейсам, как будто это вообще что-то означает — интуитивно понятным является только тот интерфейс, к которому вы привыкли, и он не заставляет вас задумываться над каждым телодвижением. Обратите внимание, как ловко подростки манипулируют предельно "неинтуитивным" интерфейсом мобильников, и вы поймете, что это совершенно пустое понятие. Интерфейс может и должен быть *логичным* — хотя и это, в общем, имеет значение только для удобства обучения, но уж к "интуитивности" точно не имеет никакого отношения. Является ли "интуитивно понятным" интерфейс управления автомобилем? Отнюдь нет — обратите внимание, скажем, что сцепление нужно *нажимать*, чтобы его *выключить*. Или: если вы замрете при ходьбе, то остановитесь, а если вы все бросите в процессе езды, то автомобиль вовсе не остановится, как можно было бы предположить, исходя из *интуитивного* представления. Но никто ведь не протестует, правда? Потому что в принципе неважно, какие именно действия нужно вызубрить, "заложить в подкорку" (на самом деле — скорее в мозжечок) — важно, чтобы они не требовали в дальнейшем напряжения сознания. Если вы будете все время в процессе езды раздумывать над вопросом, какую педаль нужно нажать для снижения скорости — вряд ли вы вернетесь из поездки живыми.

А вот для существующего WIMP-интерфейса вызубрить действия даже в одной отдельно взятой программе нельзя. Кажущаяся простота действий при указании мышью пункта меню на самом деле оборачивается тем, что это действие невозможно автоматизировать — сам процесс таков, что требует полного переключения внимания. Нужно найти это меню зрительно, направить на него мышь, щелкнуть, потом найти нужный подпункт — все это настоящая полноценная задача, требующая обдумывания, занимающая сознание целиком, как требует обдумывания вопрос, куда именно направить автомобиль при движении. Но направление движения — это и есть настоящая цель управления автомобилем, а выбор пункта меню — вовсе нет. Отличный пример на эту тему приводит известный программист Михаил Донской: *"Такой интерфейсный элемент, как линейка прокрутки, находится в противоречии с одним из основных принципов психологии восприятия: у человека может быть только одна точка активного внимания. При использовании же линейки прокрутки приходится смотреть в две совершенно*

различные точки — на прокручиваемое изображение (не пора ли остановиться?) и на линейку. Всем знакомые неприятности с непопаданием мышью в нужную точку при прокрутке или с соскакиванием курсора мыши с линейки — очевидное следствие вышеуказанного противоречия."

Наглядность Windows, которая так привлекает начинающих, в сравнении с интерфейсом командной строки, оборачивается тем, что они и в дальнейшем вынуждены тратить огромное количество времени на обдумывание действий, которые в принципе никакого обдумывания не требуют. Это все равно, как если бы педаль тормоза в автомобиле каждый раз оказывалась на новом месте и ее нужно было бы искать, например, по тому, что на ней имеется табличка: "тормоз". Задача программиста — сделать таким образом, чтобы в идеале *каждое* действие можно было бы выполнять автоматически, "спинным мозгом", сосредоточившись именно на целевой задаче, а не над тем, в каком углу экрана в данный момент находится меню, и каким щелчком мыши — двойным или одинарным — оно в этой программе вызывается. Означает ли это, что надо вернуться к интерфейсу командной строки? Вовсе нет, нужно только четко себе представлять, что запомнить раз и навсегда последовательность нажатия двух-трех клавиш в конечном итоге намного проще, чем каждый раз целиться в меню. А использовать при этом WIMP, инструментально-командную модель, метафоры "рабочего стола", "вселенной" или что-то другое — это уже дело, в принципе, десятое. Главное не забывать, для чего все это делается вообще.

Страна советов

Конечно, в реальности программисту приходится идти на компромисс, подстраиваясь под существующие стандарты. Точно так же, как архитектор или скульптор при своей работе не может не принимать во внимание законы сопромата, программист должен соблюдать некие правила, которые в совокупности делают его творение программным продуктом. Это, разумеется, в первую очередь относится к ПО, которое предназначено для распространения — неважно, за деньги или просто так. Но даже если вы делаете некую программу для собственных нужд, и это не просто единовременная "проба пера", а некий продукт, которым вы собираетесь пользоваться в течение долгого времени, эти стандарты нужно соблюдать. Грубую ошибку делает тот, кто решает за пользователя (даже если этот пользователь — вы сами), в какой последовательности ему нужно нажимать экранные кнопки. Рассчитывать нужно на тот случай, что пользователь справку не читает (даже если она есть) и обязательно первым делом нажмет именно ту комбинацию клавиш, которая "повесит" вашу программу. И вы сами через пару месяцев гарантированно забудете, что и в какой последовательности надо нажимать. Есть несколько общих правил

составления программ — и далее вы увидите яркие примеры того, как многие из этих правил нарушаются даже в самых известных продуктах.

Но прежде, чем перейти к советам по соблюдению этих правил, замечу, что есть одно главное правило, справедливое для всех почти без исключения программ, по крайней мере таких, которые не осуществляют настолько уникальные функции, что им невозможно найти замену. Это эмпирически найденное правило гласит: если пользователь не может без обращения к справке в течение максимум получаса разобраться с основными функциями программы, он ее бросает. Имея это в виду, перейдем к советам по составлению программ более подробно.

Совет 1 — о справке

Программа должна иметь справку. Совершенно необязательно составлять разветвленный Help в неудобном до крайности стандартном стиле Windows. В простейшем случае достаточно просто описать необходимые действия в текстовом файле, а в соответствующей главе мы покажем, как легко и просто без всяких дополнительных инструментов можно создавать достаточно навороченную справку в HTML-формате. Должен отметить, что только создание справки и написание комментариев (наиболее ненавидимые "настоящими программистами" этапы создания программы), займет у вас почти столько же времени, сколько написание самой программы, а иногда даже больше. Но нужно иметь в виду вот какое обстоятельство. Процесс написания справки позволяет взглянуть на вашу программу со стороны, обобщить все, что вы о ней знаете, и увидеть такие возможные упущения и ошибки, которые в противном случае выявились бы только после долговременной эксплуатации. (Задумайтесь, почему хорошие лекторы заставляют писать студентов конспекты собственноручно?) Так что не надо относиться к этому только, как к раздражающей "обязаловке".

Совет 2 — о комментариях и именах переменных

Текст программы должен быть как можно подробнее комментирован. Это тот самый случай, когда требования ГОСТ'а полностью отвечают реальному положению вещей. Той же цели повышения читаемости программ служит требование, чтобы наименования идентификаторов переменных были как можно более осмысленными: программа не рухнет, если вы назовете несколько переменных типа **`file`** именами `f1`, `f2` и т. д. Но вам не придется каждый раз искать в тексте место с их инициализацией, чтобы вспомнить, что `f1` — это файл установок, а `f2` — журнал (log-файл), если вы с самого начала присвоите им имена, например, **`file`**`_set` и **`file`**`_log`. Хорошим методом изобретения

идентификаторов является присвоение "говорящих" имен по звучанию (iks, igrek) или записанных транслитом по-русски (stroka). При присвоении имен использование фантазии по максимуму не возбраняется, но тут главное — не переборщить. Так, для именования простых численных переменных удобно использовать имена, аналогичные обозначениям в обычной алгебре: x, y, i, n и т. д. Вряд ли целесообразно затемнять смысл текста программы чем-то принципиально более навороченным.

Заметки на полях

В Windows используется так называемая "венгерская нотация", названная в честь программиста Чарльза Саймони (Charles Simonyi), венгра по происхождению. Он начинал свою карьеру еще на советских "Уралах" в начале 60-х, а в 70-е годы в упоминавшемся Xerox PARC занимался разработкой WYSIWYG-редактора Bravo. Позднее он стал главным архитектором MS Office (отсюда понятно, почему сам по себе MS Word, в отрыве от системы и поздних наслоений — вполне приличная вещь). Придуманная им нотация (т. е. правила записи имен переменных) заключается в том, что каждый идентификатор должен содержать префикс, информирующий о типе переменной. Само же наименование любой переменной должно отражать ее назначение и смысл. Тогда названия переменных — при надлежащем опыте — легко расшифровать. Так, имя "wm_KeyDown" говорит, что это сообщение Windows (Windows message), означающее, что нажата некая клавиша, а имя "faAnyFile" — что это набор атрибутов (file attributes), идентифицирующий любой (any) файл. Никто не запрещает вам придумывать и использовать свои собственные префиксы. Так, автор этих строк за некоторыми понятными исключениями старается начинать идентификаторы файловых переменных с буквы "f", строковых — с букв "st", массивов — с буквы "a" (array) и т. п. Некоторые Unix-программисты резонно замечают, что венгерская нотация ведет к излишним сложностям (например, если в процессе перехода на другую платформу тип переменной изменится, то придется переписывать весь код), что в значительной степени справедливо.

Совет 3 — об исключениях

Нужно тщательно просматривать программу на предмет возможного возникновения исключительных ситуаций. Приведу один простой пример: предположим, вы открываете некий файл, который только что сами создали. То есть он, казалось бы, гарантированно существует — ну какая тут может быть исключительная ситуация? Но если вы перед открытием не проверили, действительно ли он существует, то в сложной программе с множеством событий вы легко можете попасть в совершенно дурацкую ситуацию: представьте себе, что некий "ламер" взял и удалил этот файл в промежутке между созданием и обращением к нему. "Сам виноват" — скажете вы, и будете категорически неправы. В вашей воле прервать выполнение программы с сообщением типа "Файл ... не существует", но если программа при этом виснет или выдает нечто вроде невнятного "Access denied" — "ламер" даже не поймет, где и что

он сделал не так. А такого допускать нельзя. Я специально заостряю ваше внимание на данном примере, потому что на практике таких экзотических проверок, конечно, никто не делает, полагаясь на системные обработчики исключений. И мы также этим заниматься не будем, но вы должны понимать, что в принципе это неправильно.

Одна из грубейших ошибок практически всех производителей софта, включая разработчиков Windows и самой Delphi — когда сообщение о возникновении исключительной ситуации не содержит никакой внятной информации. В лучшем случае создатели текстов таких сообщений рассчитывают на специалиста. Автор не видит никаких причин, по которым сообщение "Ошибка 103" не могло бы выглядеть, как "Ошибка доступа к файлу <*имя_файла*>" (то же относится и к известной "Ошибке 404" в серверном ПО для Интернета, про которую, правда, все уже все выучили). И тем более недопустимы сообщения такого рода, если они возникают перед глазами конечного пользователя. Это одна из причин, по которым все возможные исключительные ситуации нужно обрабатывать в программе, не полагаясь на системные обработчики.

Совет 4 — о функциональности

Программа не должна делать ничего лишнего. Типичный пример — запуск второго экземпляра программы. Если это специально не предусмотрено (подобно тому, как Internet Explorer можно запустить в любом количестве экземпляров и это отвечает его назначению), то программа обязана, по крайней мере, информировать пользователя о том, что он запускает второй экземпляр.

Программа не должна уметь ничего лишнего. Часто встречающейся концептуальной ошибкой, которой подвержены даже самые крупные и известные софтверные фирмы, является неудержимое стремление как можно больше расширить функциональность продукта. Это далеко не всегда означает, что разработчики так уж глупы — просто в этой среде *принято* выпускать каждый год по новой версии продукта в целях поддержания конкурентоспособности. И в ситуации, когда некая программа и так уже отлично делает все, что нужно и даже сверх того, разработчики просто вынуждены в чисто маркетинговых целях увеличивать ее функциональность — очень часто вместо того, чтобы просто поправить ошибки и довести программу до ума. Типичный пример — Word for Windows, который уже в 6-й версии (еще под Windows 3*x*), и даже в почти забытой ныне 2-й, содержал практически всю нужную самому продвинутому пользователю функциональность. Когда же его пытаются декларировать, например, как средство создания HTML-страничек, ничего, кроме улыбок, это вызвать не может — это совсем *другой* продукт и предназначен он для иных целей. Единственным разумным спосо-

бом как-то объединить под одной крышей разные функции является не нагромождение функций в новой версии "все в одном", а модульный принцип построения, например, использование плагинов — кому надо, тот и установит. Но на самом деле и это не всегда требуется — вполне нормально, если каждая программа будет работать в своей области, но зато делать это на пять баллов. Ну не получается в прокрустовых рамках Windows делать универсальные программы в стиле Раскина! Я не знаю примеров программ, которые осуществляли бы разные действия лучше, чем отдельные программы, "выглаженные" каждая для одного своего действия.

Заметки на полях

Здесь можно привести аналогию с производителями принтеров, сканеров и факсов — почти 10 лет они не могли понять, почему это пользователи настолько глупы, что не желают понимать своей выгоды и приобретать устройства "все в одном" (многофункциональные устройства, МФУ) в полтора-два раза дешевле, чем по сумме каждого из агрегатов в отдельности. Количество проданных МФУ в России в 2004 году составило всего чуть больше 10% от количества принтеров, но тенденции этого рынка очень показательны: он вырос за год на 140% (а продажи принтеров — всего на 12%). Что же произошло? Очень просто: в течение всего периода с момента выхода первого МФУ, каждая из его функций сама по себе была заведомо хуже по качеству, чем отдельные устройства. И только теперь, когда все функции делаются на основе лучших отдельных продуктов той же фирмы, пользователи начинают соглашаться — да, так выгоднее и удобнее. Применяя эти соображения к нашим программам, можно выразиться так: пока Word в роли HTML-редактора будет хоть в чем-то хуже, чем DreamWeaver, эта функция в нем никому не нужна. Так что если вы решили написать текстовый редактор для создания программ на ассемблере, совершенно необязательно заставлять его проигрывать MP3-файлы — WinAmp с этим справится однозначно лучше.

Совет 5 — об интерфейсе

То, что мы говорили об интерфейсах ранее, можно конкретизировать так: программа не должна заставлять вас делать что-то лишнее. Меню программы, а также все операции должны быть простыми и понятными и по возможности осуществляться стандартным способом. Надо четко представлять, что именно вы хотите сделать, и идти к цели наиболее коротким и по возможности стандартным путем. Сама Microsoft злостно нарушила это требование еще много лет назад, когда по совершенно необъяснимой причине вдруг задействовала клавишу <Alt> для входа в меню (кстати, в *главе 7* вы узнаете, как написать утилиту, которая эту функцию отключает). Хочу предостеречь читателей от подобных извращений — системные клавиши (в первую очередь клавиши-модификаторы <Alt>, <Ctrl> и <Shift>, но также и такие, как <Esc>, <Del>, <End> и пр.) предназначены для выполнения совершенно определенных действий и не должны использоваться в вашей программе ни для чего другого.

В противном случае вы обязательно создадите большие трудности пользователям. Единственное исключение — нестандартное использование правых (дополнительных) клавиш <Alt>, <Ctrl> и <Shift>, изредка — второго <Enter>, которое стало уже традиционным по той причине, что для осуществления основных функций они оказались просто лишними. Допустимо также задействовать для какой-то оригинальной операции практически неиспользуемую системную клавишу <ScrollLock>, если вас не раздражает лампочка, а вообще горячие клавиши должны быть только из набора специально для этой цели предназначенных функциональных клавиш <F*x*> или буквенно-цифровых в сочетании с клавишами-модификаторами (достаточный набор их имеется прямо в меню соответствующего компонента Delphi и обычно ничего специально придумывать не надо). Не следует также вслед за разработчиками некоторых графических редакторов и CAD использовать в качестве горячих клавиш "голые" буквенные или цифровые клавиши — просто по той причине, что их легко случайно задеть и что-то при этом испортить. Излишне говорить, что любая достаточно солидная программа обязана дублировать все пункты меню горячими клавишами.

Заметки на полях

В качестве типичного примера, как делать *не надо*, можно привести программы для записи на CD, например, WinOnCD или Nero. Когда мне случилось столкнуться с этими программами первый раз, я был просто ошарашен. Ладно, к тому, что просто записать файл не получится, а надо обязательно создавать некий "project", я был готов. Но WinOnCD (речь идет о версии 6), помимо идиотских вопросов про какие-то "мультисессии", встретила меня четырьмя смежными окнами, снабженными в общей сложности 19 пунктами меню (многие из которых имеют иногда с десяток подпунктов). Плюс еще семь кнопок без подписей, но и этого разработчикам показалось мало, и на экране наблюдается еще 8 (восемь) закладок. "Боже мой, — растерянно думает пользователь в моем лице — а я-то хотел всего лишь файл на диск записать...". Не думаю, что сильно ошибусь, если предположу, что программисты фирмы Roxio (или те, что продали все это Roxio) решили, что они сделают интерфейс "интуитивно понятным", если продублируют одну и ту же функцию во всех местах, где только осталось свободное место. Какая уж тут психология, эргономика и прочие премудрости...

Поползновения "настоящих программистов" сделать как можно сложнее наблюдаются, например, в таких популярных и необходимых программах, как WinZip, и, к еще большему сожалению, сделанному по ее образцу WinRar — программах, в основе которых лежат чудесные алгоритмы, но с точки зрения удобства интерфейса они заслуживают огромный жирный кол. WinZip (в версии 6.3) имеет в сумме 47 пунктов меню и сверх того панель из восьми кнопок, и к тому же представлена в двух почти ничем не различающихся вариантах ("классический" и еще какой-то, не помню). Легко сообразить, что для программ типа WinZip и WinRar в их основной функциональности меню не

требуется вообще (как оно не требуется для окна папки в Проводнике) — все спокойно можно реализовать через правую кнопку мыши. Если разработчик хотел добавить что-то еще, он был просто обязан спрятать это с глаз долой — для функций типа создания самораспаковывающихся архивов лично я бы просто сделал отдельную программу (кстати, сама по себе она реализована в WinRar просто отлично, и мы непременно этим воспользуемся, см. *главу 17*).

Заметки на полях

В WinRar есть одна концептуальная ошибка, которая может показаться мелкой, но мы остановимся на ней подробнее, т. к. она очень показательна. В диалоге распаковки файлов в строке с указанием пути указано одно, а в рядом расположенном окне с деревом папок — совершенно другое. Учитывая, что строка пути сделана на основе компонента Edit с неубранным выделением (к компоненту Edit с его синим выделением мы не раз вернемся), и по умолчанию там путь высвечивается абсолютно нечитаемым белым по синему фону шрифтом 8 кегля, господин Рошал простит меня за то, что я принципиально не пользовался графическим RAR лет шесть — только по крайней необходимости. (WinZip я вообще никогда не пользовался — в Disco Commander входит свой Zip, обращение с которым на много порядков быстрее.) И только потом я обнаружил, что эта программа, оказывается, запоминает последнюю папку и наверху еще есть кнопочка "Показать"! Если это не есть типичное "умножение сущностей без необходимости", то что тогда? Подумайте, сколько *в принципе* щелчков надо сделать для того, чтобы просто извлечь файл из архива?

Еще один отрицательный пример — демонстрация сокращенного пути к файлу вместо полного. Так, известный антивирус Касперского при сканировании диска может продемонстрировать путь к проверяемому файлу в таком, например, виде: C:\...\svhost.exe. Скажите, на кого рассчитана такая запись и зачем это демонстрировать вообще? Допустимо лишь сокращать путь *до* папок, местоположение которых очевидно: запись "..\Borland\Delphi7\Projects" или "..\Program Files\Adobe\Photoshop" в большинстве случаев не создаст неудобств. Такая запись, наоборот, может быть полезна, т. к. начальные папки ("Borland" и "Program Files" в данном примере) могут быть расположены в самых разных местах, и мы в этой книге, например, таким образом подчеркиваем, что нам это безразлично.

И еще одно замечание по поводу иконок. Их часто по-русски именуют пиктограммами, но это неправильно — пиктограмма есть то, что нарисовано на иконке, а не сама иконка. Более правильно именовать иконки значками, но это слово в русском языке имеет достаточно много значений, и чтобы сразу было понятно, о чем речь, в этой книге мы будем использовать кальку с английского "icon". Но вне зависимости от названия, главное — при использовании иконки надо отчетливо представлять, что пиктограмма на ней ни в коем случае не может заменить текстовую подпись. Информативность пиктограммы зависит даже не от художественной квалификации того, кто ее составлял — она зависит от того, насколько художник и пользователь настроены

"на одну волну". А *после того*, как вы выучите, что именно обозначает тот или иной рисунок, становится уже неважно, что именно там нарисовано — пусть это хоть ровный квадрат определенного цвета, и это будет ничуть не менее информативно, чем сложный многоцветный художественный образ (возьмите в качестве примера дорожный светофор). Поэтому основная задача при составлении иконок — сделать так, чтобы они максимально *отличались* друг от друга. Не пытайтесь нарисовать, как это делают в большинстве случаев, похожие иконки, отличающиеся в деталях — не столь эффектно, но куда лучше с точки зрения функциональности будет выглядеть простейший рисунок, но зато имеющий бросающиеся в глаза отличия от соседних.

Совет 6 — о пользовательских установках

Еще один вопрос, который может показаться второстепенным, но на самом деле он очень важен. Это вопрос запоминания пользовательских установок. Конечно, если у вас и запоминать-то нечего, то и думать об этом не нужно. Но если вы, к примеру, предусмотрели в меню возможность изменения цветов интерфейса, нужно обязательно позаботиться о том, чтобы данные установки сохранялись к следующему запуску — иначе это останется совершенно бесполезной "фичей". Это очевидный пример, но есть и не столь очевидные — типичным примером "как делать не надо" будут некоторые функции все тех же программ от Microsoft. Они никогда не запоминают рабочей папки дольше, чем на один сеанс, и всегда обращаются в "Мои документы" (или то, что ее заменяет). Редчайшее исключение — Internet Explorer, который запоминает папку, куда сохранялись файлы через пункт **Сохранить как**. Объяснение этому упрямству простое и кроется в упомянутой парадигме "рабочего стола" — когда пользователь о "приложениях" понятия не имеет, а запускает исключительно "документы", ему нормальный диалог открытия и не нужен. Сложнее понять, почему была напрочь утрачена функция открытия файла именно на том месте, на котором вы с ним прошлый раз расстались. Как показывает пример Delphi и множества других программ, это совсем несложно сделать в любом редакторе. Казалось бы, можно привести аргументы в пользу того, чтобы установки, которых может быть достаточное количество, сбрасывались по окончании текущего сеанса — пользователь может просто не помнить, что он там такое установил и как вернуть все обратно. Но это порочное рассуждение: если вас это волнует, возьмите пример с разработчиков BIOS и поставьте отдельную кнопочку "Set default" (каковая, кстати, в программах от MS обычно также отсутствует). На практике возможность автоматически запоминать все установки, типы файлов, папку, куда производилось последнее обращение, место в тексте документа, на котором вы остановились последний раз, намного сокращает количество пустых операций и общее время работы с программой. В идеале для каждого такого пункта

должна быть предусмотрена отдельная возможность выбора "запоминать — не запоминать", но в принципе достаточно и общей настройки "Запоминать установки".

Совет 7 — об украшательствах

Наконец, позвольте расширить сказанное ранее об интерфейсах пользователя и дать совет любителям разукрасить программу, как новогоднюю елку, чтобы все мигало, переливалось и бегало по экрану. Пожалуй, верно, что вкус не воспитывается, он либо есть, либо его нет, но что можно тут сделать, так это посоветовать всегда помнить, для кого и для чего вы все затевали. Украшательства допустимы и уместны, если вы делаете, например, свою версию медиаплеера для младшего школьного возраста, но будут смотреться достаточно дико в программе, предназначенной для профессиональной обработки музыкальных файлов. Найдите мне хоть одного человека, который использует MS Word для серьезной работы и которого при этом не раздражает пресловутая "скрепка-помощник". Заметим попутно, что все то же самое относится к злоупотреблениям Flash-роликами при построении сайтов[3]. Как нельзя более тут подходят слова классика:

Когда в делах — я от веселий прячусь,
Когда дурачиться — дурачусь.
А смешивать два этих ремесла
Есть тьма искусников — я не из их числа.

(А. С. Грибоедов, "Горе от ума")

Но, с другой стороны, украшательства иногда просто необходимы по делу. Пример: вы делаете программу, которая имитирует функции осциллографа. Тут имеет смысл приложить все усилия, чтобы "осциллограф" выглядел "как настоящий" — с сеточкой на экране, с зелененькой кривой, с "ручками" управления, с кнопками-переключателями. Так вы сильно облегчите задачу освоения программы теми, кто настоящий осциллограф знает, как свои пять пальцев, и время, потраченное на отработку интерфейса, уйдет не просто на упражнения в программировании с целью продемонстрировать (в основном своим коллегам-"ламерам") какой вы крутой, а по делу.

[3] По степени навязчивости и отсутствия элементарного вкуса в использовании Flash авторы некоторых сайтов и рекламных баннеров переплюнули даже телевизионную рекламу, создание которой по крайней мере предполагает какой-никакой уровень профессионализма. Автор этих строк однажды просто взял и избавился от Flash-плагина раз и навсегда, перешел на браузер FireFox (чтобы не тормозило) и с тех пор чувствует себя значительно лучше — хотя и понимает, что лишил себя доступа к редким удачным опытам в этом направлении.

Совет 8 — об автоматизации

Основная цель, для которой изначально были созданы компьютеры — автоматизировать и ускорить те операции, которые человеку самому проводить сложно. Это никогда не следует упускать из виду. Описанные ранее программы, типа WinOnCD, выглядят глупо именно потому, что их разработчики об этом напрочь забыли. Давайте подумаем: ведь CD-ROM есть не что иное, как просто еще одна разновидность памяти. Формально он ничем не отличается от, к примеру, дискет. Разве вы видели программу записи на дискету, которая вынуждала бы создавать какие-то "project"? Обращение к дискете происходит совершенно прозрачно для пользователя, единственное, что ему требуется делать — следить за тем, чтобы она была отформатирована (что, если вдуматься, тоже лишнее, вполне можно было бы обойтись и без перекладывания этого действия на плечи пользователя). Программы записи на CD — типичные образчики *программистского мышления*.

Все, что может быть автоматизировано, должно быть автоматизировано, и неважно, рассчитана ли ваша программа на виртуоза-профессионала или на "ламера", привыкшего использовать компьютер исключительно как печатную машинку. Возьмем упомянутую ранее обработку исключений. Корректно составленная программа в случае обнаружения ошибки при каких-то действиях пользователя должна выводить сообщение об этом — но при действиях именно пользователя, а ни в коем случае не самой программы! Если ошибку можно безболезненно обойти, то это исключительно внутреннее дело программиста. Пользователя о таких ситуациях нужно информировать только, если от него требуется принятие какого-то решения, в противном случае он устанет нажимать на кнопку **OK**, закрывая окна с совершенно бесполезной для него информацией.

Но и впадать в противоположную крайность, делая все действия программы полностью скрытыми от пользователя, тоже не стоит — помните, что решения, которые принимаете вы, мягко говоря, не всегда являются оптимальными, вы просто не можете предусмотреть всего. Поэтому настройки автоматизации нужно спрятать от "чайника", но сделать так, чтобы они были доступны квалифицированному пользователю по максимуму.

Заметки на полях

Вот пример: пользователи скажут вам большое спасибо, если вы не поленитесь сделать автозаполнение для полей какой-либо формы так, что их нужно только немного подправить. Это касается и диалогов и интернет-форм, и вообще любых действий, когда от пользователя требуется ввести с клавиатуры что-то конкретное. Например, если пользователю требуется ввести дату, то *обязательно* нужно, чтобы в соответствующем поле появлялся образец, иначе он будет долго гадать, что именно от него хотят: вводить ли дату, как "21.06.05", "21.06.2005", "06.21.05", "21/06/05", или, может быть, как "21 june 2005"? Но нет

ничего печальнее опыта разработчиков Word, которые включали автозаполнение даты в тексте по умолчанию — само по себе это нужно крайне редко, а такая функция может создать кучу неприятностей. Представьте себе, что вы открыли созданное полгода назад служебное письмо и вдруг все даты в тексте автоматически поменялись на текущую. Неопытная секретарша может даже и не заметить этого, пока не получит выговор от шефа — это в лучшем случае. Еще хуже выглядит "автоматизация" в стиле Excel, когда вы обнаруживаете, что вместо введенного вами числа 3005,97 в ячейке вдруг оказывается записано "30 мая 1997". Как говорится, это было бы смешно, если бы не было так грустно...

В общем, все рекомендации по этому поводу вполне укладываются в лозунг: "Сервис не должен быть навязчивым!".

Это не все, что мне бы хотелось отметить в качестве советов далеко не только начинающим программистам, но для начала достаточно. Ранее я замечал, что одно выполнение первых двух пунктов может значительно увеличить время вашей работы над программой, а если учесть все остальное, то можно прийти к выводу, что основное время будет потрачено на такие вещи, которые к собственно главной функциональности программы не относятся. Да, это так, и добавлять "к сожалению" я не считаю нужным. Потому что даже такая утилитарная вещь, как приготовление и употребление пищи, предполагает создание определенного комфорта, и хозяйке приходится тратить иногда гораздо больше времени на то, чтобы приготовить не как-нибудь, а вкусно, и еще красиво оформить стол, а потом помыть посуду и вычистить кухню. Все это — неотъемлемая часть процесса приготовления еды. И программирование тут ничем не отличается от любой другой области человеческой деятельности. Можно только помочь хозяйке, например, подарив ей посудомоечную машину — именно в роли подобного помощника и выступают современные среды визуального программирования.

Немного о стилях программирования

Принцип, который автор этих строк всегда старался соблюдать, как при составлении программ, так и при разработке электронных схем — "не умножать сущностей без необходимости". Должна быть четко поставлена цель, и она должна быть достигнута минимально возможными средствами. Однако помните, что на пути упрощения вас всегда подстерегает опасность что-то не предусмотреть или забыть, и если есть такая опасность, то лучше, как говорится, "перебдеть, чем недобдеть". Так, в *главе 20* при чтении из последовательного порта для сокращения текста программы мы пренебрежем рекомендуемыми проверками типа:

```
if not ClearCommError(hPort, dwError, @ComStat) then
raise Exception.Create('Error clearing port');
```

Автор этих строк много раз пытался искусственно вызвать ситуацию, когда указанная функция ClearCommError и аналогичные ей возвращают ошибку, и ему это не удалось. Поэтому он сделал вывод, что на практике такая ошибка крайне маловероятна. И все же, если вы делаете программу для широкого распространения, этими проверками ни в коем случае пренебрегать нельзя: мало ли что может случиться, если программа выйдет из-под вашего контроля. Однажды мне пришлось столкнуться со случаем (единственный, правда, раз в жизни), когда заведомо существующий коммуникационный порт не поддавался инициализации никакими способами. И если в программе не было нужных проверок, то она просто "висла", а это недопустимо.

Несколько слов об организации циклов. Те, кто хоть раз пробовал программировать на низкоуровневом ассемблере, знают, что никаких операторов **while**, **for**, **if**...**then** и других подобных атрибутов языков высокого уровня для процессора не существует. Любое подобное действие осуществляется на самом деле с помощью условных операторов перехода на метку, а еще точнее — на нужную команду по ее адресу (в этом смысле канонический Basic, столь нелюбимый профессионалами, был даже ближе к ассемблеру, чем C или тем более Pascal). Покажем, как, например, выглядит на Intel-ассемблере цикл **if**...**then**...**else**. Пусть есть переменная **var**_x, константа **const**_1 и две процедуры: pr_1 и pr_2. Нам требуется выполнить следующую часто встречающуюся задачу: ЕСЛИ переменная **var**_x больше **const**_1, ТО обнулить переменную **var**_x и перейти к процедуре pr_1, ИНАЧЕ выполнить процедуру pr_2. После выполнения той или иной процедуры требуется, естественно, продолжить выполнение основной программы. Соответствующий программный код может быть, например, таким:

```
. . . . .
cmp var_x,const_1 ;сравниваем
ja metka1 ;если больше, то на метку 1
call pr_2 ;иначе вызываем процедуру 2
jmp metka2 ;после нее на продолжение
metka1: xor ax,ax ;обнуляем регистр ax
mov var_x,ax ;переменная = 0
call pr_1 ;вызываем процедуру 1
metka2: <продолжение основной программы>
. . . . .
```

А вот та же самая задача, но реализованная на языке Pascal:

```
if (var_x>const_1) then begin var_x:=0; pr_1 end else pr2;
```

На этом примере очень хорошо видны преимущества языков высокого уровня. Знаменитый лозунг Эдгара Дейкстры "программирование без **goto**", таким образом, предполагает фактически использование эмуляторов действий, свя-

занных с использованием условных и безусловных переходов, но зато следование ему значительно повышает читаемость программ. И все же оператор **goto** включен во все современные языки, и никто вам не может запретить его использовать. Для людей, которые имеют образно-аналитический склад мышления, иногда проще разобраться в хитросплетениях условных и безусловных переходов, чем вникать, к примеру, в семантические различия между циклами типа **while**...**do** и **repeat**...**until**.

То же самое относится и к использованию флагов. Прием, когда по некоторому событию устанавливается некий флаг (в простейшем случае — всего один бит в некоторой переменной или регистре, но можно использовать и переменные целиком), является основным способом организации взаимодействия отдельных команд, целых процедур и даже узлов аппаратуры при низкоуровневом программировании. Так, в представленном ранее примере тоже неявно используется флаг — специальный бит в регистре флагов (бит переноса CF) устанавливается или сбрасывается в результате операции сравнения cmp (фактически — операции вычитания) и служит сигналом для последующей команды перехода. В высокоуровневых языках, как правило, можно обойтись без специальных флагов и институционально такой прием вообще "вне закона", но, как мы увидим, иногда его использование не только значительно облегчает составление программ, но и применяется самими разработчиками системы (скажем, булевская переменная CanClose, используемая в методе **Close** формы, есть не что иное, как типичный флаг по смыслу и назначению).

Немного о локальных и глобальных переменных. Автор этих строк предпочитает везде, где только можно, для передачи параметров пользоваться глобальными переменными — вместо того, чтобы организовывать процедуры с формальными параметрами. Это, разумеется, не руководство к действию, а просто привычка — на мой взгляд, так повышается читаемость и прозрачность программы. Разумеется, при использовании глобальных переменных для организации, например, циклов вы легко можете наделать ошибок, в причинах которых иногда непросто разобраться, недаром Delphi выводит даже специальное предупреждение на этот счет. Нет никаких особых причин, кроме указанных, для того чтобы следовать тому или иному стилю программирования, только при использовании локальных переменных автор настоятельно рекомендует по возможности присваивать им имена, не совпадающие ни с другими локальными переменными, ни с глобальными, иначе в тексте вашей программы не разберется и сам Валерий Васильевич Фаронов.

Еще одно замечание относится к собственно структурированию. Если текст процедуры состоит из одного-единственного оператора, то для повышения читаемости, на мой взгляд, лучше повторить нужный фрагмент несколько раз в основном тексте, каждый раз с новой переменной, чем выделять его в от-

дельную процедуру — хотя это и противоречит принципам структурного программирования. Не забывайте, что структурирование по процедурам и функциям удобнее с точки зрения программирования, но часто затрудняет чтение текста программы и к тому же ведет к раздуванию скомпилированного кода за счет организации стека при вызове процедуры.

ГЛАВА 2

Начинаем работу
Создаем типичное приложение

> При ремонте окон ни в коем случае нельзя вытаскивать шурупы клещами и забивать их молотком. Следует пользоваться только отверткой.
>
> *"1000 советов любителю мастерить"*

В качестве базового приложения мы создадим просмотрщик слайдов. Это, с одной стороны, функционально достаточно богатая вещь, и содержит в себе примеры многих типичных задач, возникающих при создании пользовательских программ. С другой стороны — само по себе приложение несложное, что позволит нам не отвлекаться на частности. В иллюстративных целях я опишу процесс создания достаточно подробно. Хотя в этой главе вы ничего, выходящего за рамки обычной работы в среде Delphi, не встретите, но подробности позволят даже самому неискушенному читателю избежать потерь времени на разрешение многих частных проблем, возникающих при создании приложений. А тем, кто процесс создания приложения уже знает достаточно хорошо, я все же рекомендую не просто скопировать проект с прилагаемого диска, а просмотреть эту главу хотя бы по диагонали — возможно, вы встретите здесь некоторые вещи, о которых ранее не знали.

Два слова по предварительной подстройке IDE (пункт меню **Tools**). Перед тем, как приступить к работе, я рекомендую сделать следующие установки:

- в пункте **Tools | Environment Options** на закладке **Preferences** установить флажки в обоих пунктах группы **Autosave options** и в пункте **Show compiler progress**. Первое даст вам возможность автоматически сохранить последний вариант проекта в случае, если Delphi при пробном запуске приложения "повиснет" (а в некоторых случаях, особенно при прямом обращении к API, это совсем не исключено), второе сделает процесс компиляции более наглядным;

- в пункте **Tools | Editor properties** на закладке **General** установить флажок в пункте **Undo after save**. Особенно актуально иметь такую возможность отмены изменений в тексте программы после неудачной компиляции.

Есть еще один пункт, на который стоит обратить внимание. Он находится на вкладке **Tools | Debbuger Options | Language Exceptions** и называется **Stop on Delphi Exceptions**. Если оставить его отмеченным (режим по умолчанию), то Delphi после запуска приложения будет "тормозить" на всех процедурах обработки исключений, что не слишком удобно на практике. Однако в некоторых случаях это может оказаться необходимым — не только, когда процедура обработки исключения работает не так, как запланировано, но и в случае необходимости отследить ошибки времени выполнения, которые иначе поймать не удается. Целесообразнее всего этот флажок снять, но на всякий случай помнить о такой возможности.

Теперь можно приступать к созданию проекта. Сначала надо кратко набросать его план — что именно мы хотим получить? В крупных проектах эта стадия — написание технического задания — может занять достаточно длительное время, а мы можем просто записать план на бумажке. Итак, наш простейший просмотрщик слайдов будет:

- открывать JPEG-файлы, находящиеся в нужной папке;
- демонстрировать картинки из этой папки по очереди от первой до последней по команде пользователя;
- демонстрировать их в режиме автоматического слайд-шоу, с неким интервалом времени, в закольцованном режиме.

Конечно, для того чтобы создать нормальную программу такого направления, следовало бы еще ввести:

- возможность просмотра картинок в виде набора "превьюшек" (preview);
- регулировку времени между демонстрацией отдельных слайдов;
- сортировку имен файлов по алфавиту;
- поиск файлов не только с расширением jpg, но и jpeg, а также, по возможности, и других форматов;
- отображение имени файла и размера изображения;

и т. п.

Все это не так уж и сложно, но в результате получится довольно громоздкая конструкция, а наша задача сейчас в том, чтобы создать законченное и компактное приложение-основу, которое мы потом будем постепенно дорабатывать. Все перечисленные функции, и даже несколько сверх того, мы введем в дальнейшем по ходу изложения *(законченный проект SlideShow см. в главах 15 и 16)*.

Начало

Запустим Delphi и создадим заготовку нового проекта (**File | New | Application**). Самое первое, что нужно сделать, — сохранить еще не рожденный проект, для чего выбираем пункт **File | Save project as**. У вас появится обычное окно сохранения файла с заголовком **Save Unit As**. Не забывайте, что введенное здесь имя — еще не имя проекта (программы), это только имя главного модуля с текстом, поэтому можно вести любое имя файла, и даже оставить предлагаемое по умолчанию — Unit1.pas, хотя это и неудобно (представляете, сколько у вас этих одинаковых "юнитов-1" потом наберется?). Введем имя, например, slide.pas. Перед тем, как нажимать кнопку **Сохранить**, в текущей папке Projects следует создать специальную папку для проекта, иначе потом файлы этого и других проектов перепутаются. Выбор папки и наименований — очень ответственный момент, потому что в Delphi переименовать что-то в проекте с перемещением его в другую папку — процедура не совсем тривиальная. Если вы в дальнейшем будете создавать еще модули, то обращайте внимание на то, в какой папке они будут сохраняться — не исключено, что они по умолчанию окажутся, например, в папке Мои документы, и вы потом потратите кучу времени на то, чтобы корректно собрать проект воедино.

Итак, создаем папку с именем Glava2, открываем ее и, наконец, сохраняем модуль. У вас тут же появляется аналогичное окно, но с заголовком **Save Project As** — тут уже надо вводить то имя, которым потом будет называться исполняемый файл программы. Пусть это так и будет SlideShow (пока этим именем будет называться файл проекта — SlideShow.dpr).

Компоненты

Заготовка будущего проекта, которую мы видим, представляет собой пока просто пустую форму `Form1` — будущее окно программы. Процесс программирования будет заключаться в том, что мы наполним форму компонентами, расставим их по местам, придадим им в Object Inspector необходимые свойства, затем напишем обработчики нужных событий. В процессе работы нам придется неоднократно запускать на выполнение еще недоделанный проект, так удобнее искать ошибки — *отлаживать программу*. При запуске на выполнение проект будет каждый раз *компилироваться* — в выбранной нами папке появится исполняемый файл программы SlideShow.exe, который, собственно, и будет запускаться по команде **Run** из меню Delphi (для этого нужно выбрать пункт меню **Run | Run** <**F9**>). Напомним, что если в программе есть критические ошибки периода компиляции, то компиляция не пройдет до конца, а на ошибки вам укажут. Если вы просто хотите проверить программу на отсутствие ошибок, не запуская ее, то нужно нажать клавиши

<Ctrl>+<F9> — тогда программа скомпилируется, не запускаясь (почему-то в 7-й версии Delphi этот пункт перенесен из меню **Run** в меню **Project**). При этом нужно учитывать два обстоятельства: во-первых, то, что при наличии ошибок EXE-файл не создается, а старый окажется стерт — т. е., если вы хотите сохранить промежуточный вариант скомпилированного файла, это следует делать до внесения изменений.

Первый компонент, который нам понадобится, — `MainMenu`, главное меню программы. Он расположен в палитре компонентов на закладке **Standard**, второй слева. Щелкаем по иконке, потом щелкаем на форме — иконка переносится на форму. Подобные компоненты можно располагать в любом месте формы (меню после компиляции все равно окажется там, где надо), но удобнее его поместить ближе к верхнему обрезу формы, иначе иконка будет "путаться под ногами" при установке других компонентов.

Затем нам понадобится диалог открытия файла. Он расположен на закладке **Dialogs**. Для того чтобы до нее добраться, нужно прокрутить палитру компонентов с помощью стрелочки у правого края верхнего окна. Компонент `OpenDialog`[1] самый первый слева. Переносим его на форму описанным ранее способом и располагаем где-нибудь рядом с меню. Далее таким же способом переносим на форму компонент `Timer` (нам ведь надо будет чем-то отмерять время при автоматическом показе) — он находится на закладке **System**.

Из визуальных компонентов нам в первую очередь понадобится кнопка, с помощью которой мы будем менять слайды. Используем также стандартную кнопку (компонент расположен на закладке **Standard**, иконка в виде маленькой кнопочки "Ok"). Переносим ее на форму и располагаем там, где она будет находиться в программе — внизу посередине. Кнопке можно придать любые размеры, растягивая ее мышью, но здесь нас устраивает и стандартный размер — если не нравится, можно подправить. Пока на ней написано `Button1`, но потом мы изменим надпись.

Теперь мы добавим один компонент для красоты — рамку будущего экрана для демонстрации. Картинки на экране несравненно лучше выглядят на черном фоне. Одно из крупнейших упущений составителей почти всех без исключения подобных программ под Windows — то, что они (скорее всего, даже не подозревая об этом) пытаются следовать рекомендациям для демонстрации именно картин, т. е. изображений на твердом носителе в отраженном свете. Такие изображения следует располагать на нейтральном сером или близком к нему фоне (именно так оформляют картинные галереи). А вот самосветящиеся изображения нужно помещать на черный и только на черный

[1] Напомню, что есть и специальный компонент-диалог открытия изображений (всем знакомый по MS Word: когда нам приходится вставлять рисунок в текст, именно этот диалог и используется), но в данном случае мы, исходя из нашей общей установки поступать как можно проще, будем использовать универсальный диалог.

фон (вспомните интерьеры кинотеатров), иначе они зрительно потеряются. Причем дополнительным достаточно сильно искажающим восприятие фактором будут окружающие картинку интерфейсные элементы — меню, кнопки, бордюр и т. п.; особенно это критично для "развлекательного" стиля Windows XP. В идеале нужно демонстрировать изображение "во весь экран", но это не всегда удобно для пользователя, поэтому придется хотя бы отделить изображение от интерфейсных элементов черным фоном-рамкой.

Рамку удобнее всего сделать из компонента `Panel` (находится на закладке **Standard**). Переносим ее на форму (она получит имя `Panel1`) и растягиваем за уголки так, чтобы от формы по боковым краям осталась только узенькая рамочка, а вверху и внизу оставалось небольшое пространство для меню и кнопки. Последний и главный компонент, который нам потребуется — собственно экран для отображения картинок. Этот компонент расположен на закладке **Additional**, носит незатейливое название `Image`, и его иконка выглядит, как сине-голубой квадратик (который, видимо, означает деревенский пейзаж на фоне голубого неба). Переносим его на форму, щелкнув на уже имеющейся там панели `Panel1` (компонент обозначится просто пунктиром и получит имя `Image1`), и тоже растягиваем, оставив поля, которые потом будут черными. К этому моменту наша форма должна иметь вид, показанный на рис. 2.1.

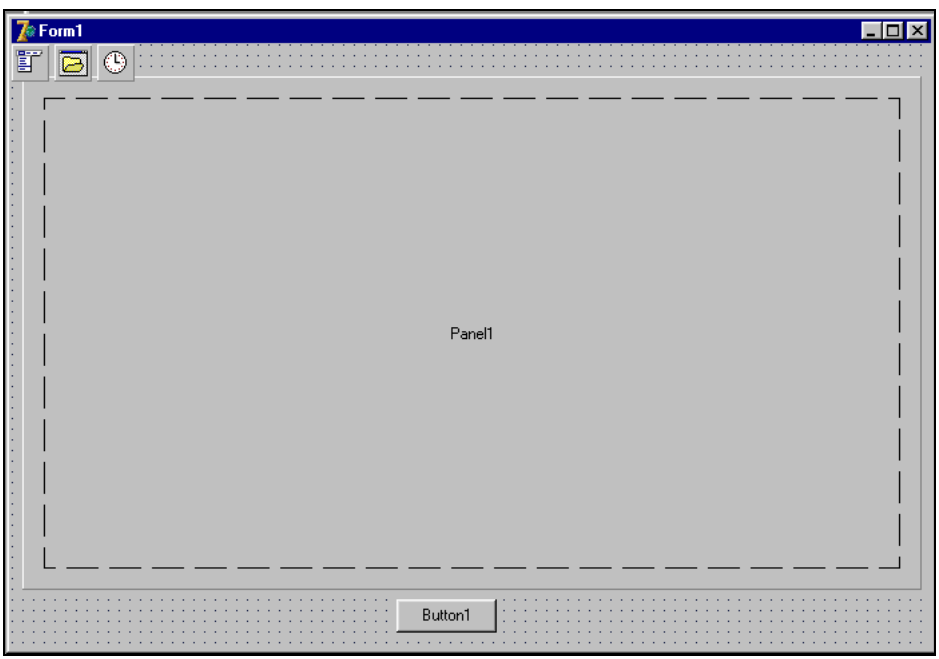


Рис. 2.1. Заготовка проекта SlideShow

Первый этап можно считать законченным — компоненты мы на форму "накидали". Теперь перейдем ко второму этапу — будем придавать им нужные свойства.

Свойства

Начнем с панели, которую мы обещали сделать черной. Для этого выделите панель (нужно щелкнуть на ней с краю, вне пределов компонента `Image1`, иначе выделится именно `Image1`). В окне Object Inspector появится имя компонента (`Panel1`), а ниже — перечень его свойств. Во-первых, очистим свойство `Caption`, где написано `Panel1` — надпись на панели пропадет и она станет чистой. Теперь щелкните на пункте `Color` и затем на стрелочке справа — у вас появится длинный список цветов, среди которых нужно выбрать `clBlack`. Другой способ, который позволяет задать цвет визуальным выбором — дважды щелкнуть на поле с названием цвета, а в появившейся таблице нажать кнопку **Определить цвет**. У вас появится таблица цветов (рис. 2.2 — не изумляйтесь, что там все по-русски — это функция не Delphi, а Windows), из которой можно выбрать любой из 16 миллионов оттенков, так что если вам моя идея насчет черного цвета, несмотря на все высказанные аргументы, не нравится, то выбирайте по вкусу. Кроме этого, придайте свойству `BorderStyle` значение `bsSingle` — края панели вместо выпуклых станут вогнутыми, как будто у нас настоящий экран.

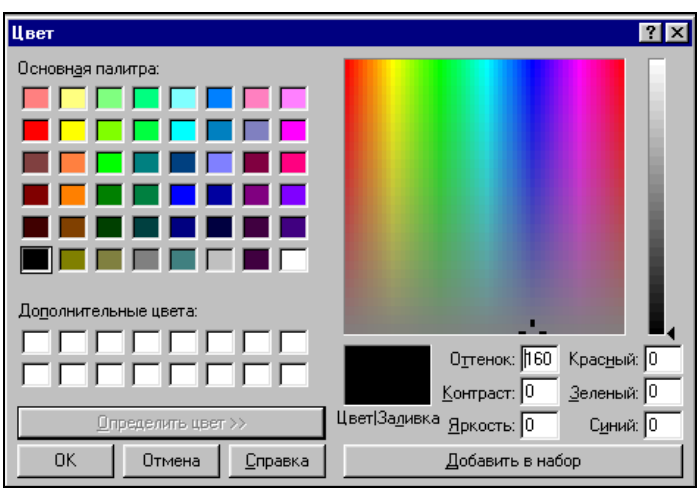


Рис. 2.2. Окно выбора цвета

Пунктир, обозначавший компонент `Image1`, у нас теперь пропал из виду, но мы помним, где компонент находится, а если забыли — нужный компонент

всегда можно выделить, выбрав его из выпадающего списка компонентов в верхнем окне Object Inspector. Выделим Image1 и прежде всего установим в True значения свойств Stretch (лучше всего это перевести, как "растяжимость") и Proportional — тогда загруженная картинка будет всегда растягиваться до размеров окна компонента Image, не искажаясь в пропорциях. А чтобы картинка располагалась по центру нашего экрана, надо установить в True свойство Center этого компонента. Да, а вдруг пользователь по ходу просмотра захочет распахнуть окно программы на максимум? Тогда, для того чтобы наши компоненты тоже соответственно поменяли размеры, нужно для компонентов Panel1 и Image1 выбрать свойство Anchors и установить в значение True все четыре пункта (akLeft, akTop, akRight и akBottom — эти пункты выравнивают компонент соответственно по левому, верхнему, правому и нижнему краям), которые раскрываются, если нажать на маленький плюсик слева от надписи "Anchors" (два из этих пунктов уже установлены в True по умолчанию).

Заметки на полях

Когда компонентов на форме много, процедура установки их свойств через Object Inspector (Инспектор объектов) может утомлять. На этот предмет в Delphi предусмотрена красивая возможность клонирования одинаковых компонентов со всеми их свойствами обычным методом Copy-Paste. Можно даже выделить все компоненты на форме через обычный пункт **Edit | Select All (<Ctrl>+<A>)** (или несколько из них — щелчком мыши при нажатой клавише <Shift>) и перенести их через буфер обмена в новый проект, новую форму или на другое место без изменений, кроме имени, которое присваивается автоматически добавлением к старому порядкового номера.

Прежде чем двигаться дальше, неплохо бы запустить нашу программу и проверить, как это все выглядит. Но проверить хорошо бы прямо с какой-нибудь картинкой! Для загрузки картинки найдите для компонента Image1 свойство Picture. Нажмите на кнопочку с тремя точками справа, и затем в открывшемся окне — на кнопку **Load**. Выберите какую-нибудь картинку типа JPEG — она загрузится в нашу форму (на диске файл 0klen2.jpg расположен в папке с проектом). Если вы ее потом не уберете, она внедрится в исполняемый файл (точнее в файл ресурсов проекта DFM, откуда и будет перенесена в исполняемый файл при компиляции) и будет всегда появляться при запуске программы. Можно изготовить какой-нибудь логотип с вашим портретом.

Итак, запускаем программу на выполнение (**Run** или <F9>) Сейчас, конечно, критических ошибок быть не может, поэтому она запустится наверняка. Распахнув окно программы на полную, мы увидим нечто, похожее на рис. 2.3.

Здóрово, но куда же подевалась кнопка? Да вот она — посередине фотографии! И заголовок окна какой-то невнятный: "Form1"... Немедленно исправлять! Закроем программу, выделим на форме кнопку Button1 и в ее свойстве Anchors установим в False все пункты, кроме akBottom. Теперь перейдем к

самой форме `Form1` (чтобы ее выделить, нужно щелкнуть на узенькой полоске между панелью и заголовком, или просто найти ее в Object Inspector). Там мы меняем заголовок (в свойстве `Caption`) на название нашей программы: `SlideShow 1.0`. Номер версии нужен не только для солидности, но и потому, что когда вы в дальнейшем будете улучшать программу, его следует увеличивать, как положено — если номера нет, вы и сами запутаетесь в том, какой именно вариант у вас перед глазами. Основной номер обычно соответствует капитальным переделкам, а цифра(ы) после запятой — косметическим улучшениям или просто вариантам. Следует отметить, что в Delphi имеется штатная возможность присвоения номера версии (а также ряда других "опознавательных" параметров — через пункт **Project | Options | Version Info**), но в этом случае номер версии будет автоматически доступен только через контекстное меню приложения. Для того чтобы его оттуда извлечь и поместить, например, в заголовок формы или в меню **About** (с целью автоматического изменения текста в них), нужно вызвать соответствующую структуру через функции API, мы вернемся к этому вопросу позже *(см. главу 11)*.

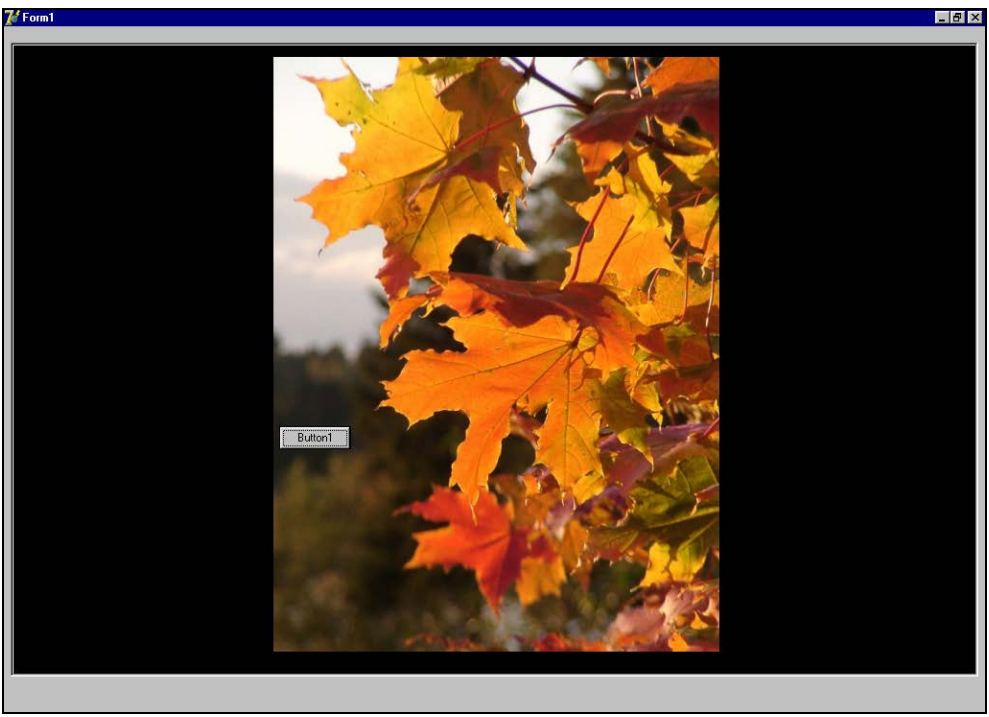


Рис. 2.3. Окно программы SlideShow

Кроме этого, для формы нужно установить положение на экране монитора, в котором она будет появляться после запуска. Это делается в свойстве

Position — по умолчанию там стоит poDesigned, т. е. окно программы будет появляться там, где вы его расположили при проектировании. Это неудобно, потому из многочисленных вариантов выбираем poDesktopCenter — тогда окно будет появляться в центре рабочего стола. Проверяем еще раз (**Run** или <F9>) и убеждаемся, что теперь все на месте. Да, а надпись на кнопке? Возвращаемся к Button1 (не забыв закрыть запущенную программу) и в свойстве (угадали каком?) Caption меняем надпись на ------> (имитацию стрелочки, ведь кнопка у нас будет перелистывать картинки). Для того чтобы нарисовать на кнопке красивую стрелочку, есть известные специальные способы *(см. главу 15)*, но описание того, как это делается, отвлекло бы нас сейчас от более насущных вещей.

Меню, таймер и диалог

Теперь сформируем главное меню. Дважды щелкнем на иконке меню, у нас появится окно с заготовкой. В свойствах выберем пункт Caption, ставший для нас уже родным, и напишем заголовок первого пункта: Файл (по-русски, естественно), а также сменим имя N1 (свойство Name) на более внятное File1. После того как вы нажмете клавишу <Enter>, у вас заголовок перенесется на заготовку, а рядом с ним пунктиром обозначится следующий пункт. Выделим его, щелкнув по нему мышью, и назовем Демонстрация, а имя пусть будет — Show1[2]. Теперь вернемся к первому пункту Файл и выделим появившийся квадратик ниже — заготовку для подпункта. Его мы назовем Открыть (имя — Open1). Чтобы было все сделано грамотно, для подпунктов следует установить горячие клавиши. Это делается в свойстве ShortCut — установим для открывания файла сочетание клавиш <Ctrl>+<O>. Перейдем опять к пункту Демонстрация и назовем его подпункт Запуск (имя — Run1), а горячую клавишу для него "своруем" у Delphi — <F9>. Пока пунктов нам достаточно. Окончательно меню в отладочном окне будет выглядеть, как показано на рис. 2.4. Закроем окно заготовки меню, запустим программу и убедимся, что меню появилось на нужном месте.

Для таймера Timer1 нужно установить его свойство Interval в значение 3000, тогда таймер будет срабатывать каждые три секунды (разумеется, можно в любое другое по выбору — идея установки ясна). Кроме того, для него нужно установить свойство Enable в False, т. к. сразу после запуска он работать не должен.

---

[2] Нельзя присваивать пунктам меню названия, совпадающие с уже определенными идентификаторами. Если это зарезервированное слово (например, File), то Delphi просто не позволит вам этого сделать, но в остальных случаях проверки не производятся. Так, в первоначальном варианте автор бездумно назвал данный пункт просто Show, что вызвало в дальнейшем неприятности при попытке вызвать метод Form1.Show.

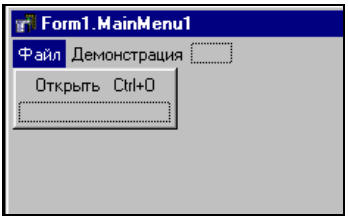


Рис. 2.4. Главное меню программы SlideShow

А для диалога `OpenDialog1` нужно сразу установить фильтр по расширению файлов, которые он будет открывать. Для этого открываем его свойство `Filter` и в открывшемся окне вносим в первую строку название фильтра (`Картинки JPEG`) и маску файлов: `*.jpg` (рис. 2.5).

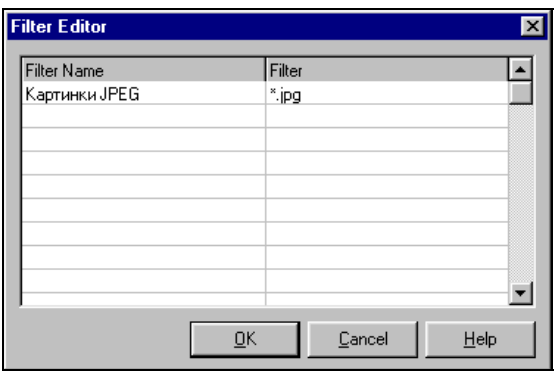


Рис. 2.5. Маска файлов для программы SlideShow по умолчанию

Открытие файла

Наконец-то можно приступить непосредственно к программированию. Чтобы программа начала работать, нам нужно по крайней мере указать папку, в которой хранятся картинки. В нашей простейшей программе мы будем загружать какую-нибудь картинку через диалог открытия файла и заодно тем самым указывать, какую папку считать текущей. На самом деле это не совсем правильно — профессионально сделанные программы с возможностью просмотра некоторого набора файлов предполагают, что указание папки и просто загрузка одного файла разные операции, но к этому вопросу мы вернемся несколько позднее *(см. главу 9)*.

На форме переходим к главному меню и щелкаем по пункту **Файл** | **Открыть**, как будто мы действительно собираемся открыть файл. На первый план выдвигается окно с заготовкой текста модуля slide.pas. Напоминаю, что

все необходимые служебные записи в текст Delphi вносит автоматически, при указании интересующих нас событий, и попытка внести правку в эти служебные записи вручную должна предприниматься с полным пониманием того, как и что именно делается (далее мы изучим некоторые такие операции). В данном случае у нас появилась заготовка процедуры обработки события `Open1Click`, которое состоит в том, что мы обратились к пункту **Открыть**. Что должна программа делать, если мы щелкнули по этому пункту? Разумеется, вызвать диалог открытия файла.

Прежде чем писать обработчик, сначала объявим глобальную переменную `n` типа `integer` — флаг, который будет показывать, первая это загрузка из данной папки при обращении к пункту **Открыть** или нет. Без этого нам пришлось бы просматривать папку каждый раз заново, что значительно удлиняло бы время загрузки. Использование флага в данном случае — наиболее простой и очевидный способ, хотя, разумеется, и не единственный. В данном случае мы еще и инициализируем эту переменную начальным значением, равным 0.

Заметки на полях

Формально в языке Pascal (точнее, его варианте от Borland) так записывалась только типизированная константа, а не переменная, но разница между ними всегда была настолько гомеопатическая, что разработчики Delphi не сочли нужным даже изменить синтаксис. Типизированные константы, как таковые, также остались, но в Delphi, начиная с 6-й версии, в режиме по умолчанию их динамически изменять нельзя, для этого нужно установить флажок **Assignable typed constants** на вкладке **Compiler** в пункте **Project | Options**.

На самом деле все переменные в Delphi по умолчанию вроде бы и так инициализируются нулевым значением, но с идеологической точки зрения было бы категорически неправильно опустить этот момент — а вдруг область памяти, отведенная под переменную, уже ранее использовалась? Любые критичные для дела начальные значения нужно всегда устанавливать явно — либо в свойствах компонента, либо динамически, например, при запуске программы.

Кроме флага `n`, нам в дальнейшем понадобится переменная типа `TSearchRec` для поиска файлов, назовем ее `srf`, и строковая переменная `st`. Соответствующий фрагмент текста тогда будет выглядеть так:

```
. . . . .
var
Form1: TForm1;
n: integer=0;
srf: TSearchRec;
st: string;
implementation
. . . . .
```

Корректно составленная программа в случае обнаружения ошибки при каких-то действиях пользователя (в данном случае — если при нажатии кнопки загрузка файла не прошла) должна выводить сообщение об этом. Еще раз подчеркнем — при действиях именно пользователя, а не самой программы *(см. главу 1)*. Это можно сделать самыми разными способами, и мы используем вызов окна сообщения через Delphi-трансляцию соответствующей функции API под названием `MessageBox`. Нельзя сказать, что с дизайнерской точки зрения это идеальное решение — функция выполнена довольно примитивно. Не так уж сложно сделать и красивее — организовать, например, невидимую изначально панельку, раскрасить ее, ввести хорошо читаемый шрифт и т. п. Главное — собственную функцию можно заставить автоматически убираться с экрана, как только мы начинаем делать что-то еще, и это будет наиболее правильно с точки зрения интерфейса, т. к. щелчок на кнопке **OK** для простого сообщения есть, в общем, лишнее действие. Но это достаточно усложняет программу: при дальнейшем усовершенствовании все время придется следить за тем, чтобы вся автоматика работала, как надо, так что в большинстве случаев мы будем обходиться без этого.

Синтаксис вызова функции `MessageBox` очень хорошо иллюстрирует обращение с функциями API вообще. Напомним, что в Object Pascal строковые переменные фактически представляют собой массив значений типа **`char`**, индекс которого отсчитывается с 1, а по нулевому значению индекса хранится фактическое число элементов массива. В то же время во всех современных языках, сделанных на основе C, строки являются нуль-терминированными (подробную информацию об этом см. в *главе 21*). Поэтому паскалевские строки **`string`** при обращении к API приходится преобразовывать в нуль-терминированные (в языке Object Pascal такой тип называется `Pchar`). Проще всего это сделать с помощью обычной процедуры прямого преобразования типов (а обратный перевод, кстати, вообще не требуется — можно переменной типа **`string`** просто присвоить значение переменной типа `Pchar`).

Заметки на полях

При задании строк в Delphi для функций API рекомендуется делать либо явное преобразование типов:

```
var st:string;
. . . . .
st:='Неправильный размер файла';
Application.MessageBox(Pchar(st),'Error',MB_OK);
```

Либо прямо указывать строку внутри функции:

```
Application.MessageBox('Неправильный размер файла','Error',MB_OK);
```

Не рекомендуется присваивать непосредственное значение строке типа Pchar:

```
var pst:Pchar;
. . . . .
pst:='Неправильный размер файла';
Application.MessageBox(pst,'Error',MB_OK);
```

В данном конкретном примере, возможно, все будет нормально, но у автора бывали случаи, когда подобная конструкция вызывала исключение ("Access violation..."). Вообще со строками типа Pchar, как с не "родным" для Delphi форматом, следует работать с большой осторожностью (см. также замечание на эту тему в *главе 14* и подробности в *главе 21*).

Таким образом, между словами **begin** и **end** (так называемые операторные скобки) в заготовку процедуры обработки события вносим следующий текст:

```
procedure TForm1.OpenClick(Sender: TObject); {открытие файла}
begin
If OpenDialog1.Execute then
{если диалог открытия файла завершился удачно}
Image1.Picture.LoadFromFile(OpenDialog1.FileName)
{загружаем картинку в Image1}
else
begin
      {иначе, если диалог завершился неудачно, выводим сообщение}
st:='Неправильный формат файла '+ OpenDialog1.FileName;
Application.MessageBox(Pchar(st),'Error',MB_OK);
exit; {выходим из процедуры}
end;
n:=0; {n=0 - признак первой загрузки из данной папки}
end;
```

Здесь Error — то, что будет демонстрироваться в заголовке сообщения, а в переменной st содержится собственно текст. Заголовок, к слову, также имеет тип Pchar — если вы захотите его формировать отдельно. В принципе мы могли бы сразу объявить переменную типа Pchar, но st мы потом можем использовать и в каких-нибудь других целях.

Запускаем программу и пробуем открыть какую-нибудь картинку. Если запуск прерывается и вам сообщают нечто вроде "jpg — unknown extension for files containing image" (jpg — неизвестное расширение для файлов, содержащих изображения), то это означает, что отсутствует ссылка на модуль, который умеет обрабатывать изображения, сжатые с помощью алгоритма JPEG. Скорее всего, вы в данном случае при установке свойств компонентов вообще не загружали начальную картинку в свойство Pucture компонента Image1,

или загруженное тогда изображение имеет тип не JPEG, а BMP — иначе бы ссылка вставилась автоматически. Теперь придется это делать вручную. В начале текста модуля slide.pas после слова **uses** идет перечисление всех используемых модулей, куда мы через запятую добавляем название нашего модуля — JPEG:

```
uses
. . . . ., StdCtrls, Menus, JPEG;
```

Если теперь все запускается нормально, и файлы открываются как надо, идем дальше.

Перелистывание

Следующее событие, которое нас интересует, — нажатие кнопки Button1, по которому происходит загрузка очередного рисунка. Сначала мы отдельно оформим процедуру поиска файлов и их загрузки — она еще пригодится при работе по таймеру. При этом неплохо бы как-то узнать, что мы пролистали все файлы и дошли до конца каталога. В данном случае у нас сделано так, что последний файл в папке будет просто показываться дважды — и мы этим на данном этапе ограничимся. Если продолжать нажимать, то все будет начинаться сначала. Процедуру Loadfile (текст приведен далее) надо вставить в текст программы выше обработчиков событий, сразу после слова **Implementation** и зелененькой надписи с директивой {$R *.dfm} — расположение своих процедур в начале (а не вперемешку с обработчиками событий) удобнее для чтения текста программы. Обратите внимание, что в своих процедурах наименование свойств и методов компонентов надо либо начинать с самого начала по всей цепочке их принадлежности (Form1 и т. п.), либо использовать оператор **with**.

```
procedure Loadfile; {процедура поиска и загрузки файлов JPEG}
begin
if n=0 then {если это первая загрузка}
begin
 if FindFirst ('*.jpg',faAnyFile,srf)<>0 then exit;
{ищем самый первый JPEG-файл в папке, если не находим - выход из процедуры}
 try {пробуем его загрузить в Image1}
 Form1.Image1.Picture.LoadFromFile(srf.Name);
 except {это вовсе не JPEG}
                        {выводим сообщение}
st:='Неправильный формат файла '+ srf.Name;
Application.MessageBox(Pchar(st),'Error',MB_OK);
       exit {выход из процедуры}
 end;
```

```
 n:=1; {следующая загрузка будет уже не первой}
end else {если эта загрузка уже не первая}
begin
 if FindNext(srf)=0 then
{если =0, то найден следующий файл}
 try {пробуем его загрузить в Image1}
 Form1.Image1.Picture.LoadFromFile(srf.Name)
 except {это вовсе не Jpeg}
                {выводим сообщение}
st:='Неправильный формат файла '+ srf.Name;
Application.MessageBox(Pchar(st),'Error',MB_OK);
       exit {выход из процедуры}
 end
 else  {а если не 0 - значит файлов больше нет}
 begin
  FindClose (srf); {закрываем поиск}
  n:=0; {в следующий раз начнем сначала}
 end;
end;
end;
```

Заметки на полях

Текст этой процедуры хорошо иллюстрирует использование операторных скобок **begin...end.** Они нужны для того, чтобы выделить группу операторов программы, которые будут выполняться совместно. Например, они обязательно обрамляют все операторы процедуры, чтобы компилятор знал, где в тексте эта процедура заканчивается. Кроме этого, скажем, в данном случае ими выделяются те операторы, что будут выполняться по одному и тому же условию (оператор **if...then...else**). Если условие не выполняется, они будут пропущены и выполнится другая группа операторов. Если в группе всего один оператор, операторные скобки можно не ставить.

Оформлять текст программы принято так, чтобы внутри каждой группы, обрамленной операторными скобками **begin...end**, текст сдвигался вправо. Осуществлять сдвиг текста полагается клавишей <Tab> или пробелами. Если сдвига не делать, то в сложных выражениях очень трудно бывает найти, к какому **begin** какой **end** относится. Пожалуй, единственная ошибка, при наличии которой компилятор может не указать на настоящую ее причину, а реагировать самым причудливым образом — несоответствие количества **begin** количеству **end**. Знайте, если сообщение об ошибке содержит откровенную чушь, то, скорее всего, дело именно в этом. Тем более, что служебное слово **end** используется не только как закрывающая операторная скобка, но и в других случаях. Один из таких случаев мы также видим здесь: это пример использования обработки исключений. Имеется в виду конструкция **try...except...end**. Работает она так: если процедура после **try** выполнилась правильно, то программа перейдет к оператору после **end**, а если нет — выполнит процедуру после слова **except**. Есть и более сложный формат процедур обработки исключений.

Теперь нужно создать обработчик события щелчка по кнопке Button1. Для этого просто дважды щелкнем по ней на форме, а затем впишем в появившуюся заготовку для процедуры обработчика события Button1Click строку с единственным оператором, который есть имя нашей процедуры:

```
. . . . .
begin
  Loadfile;
end;
. . . . .
```

Запустим программу и проверим, как это работает.

Осталось оформить процедуру автоматической демонстрации. Для чего щелкните в главном меню на пункте **Демонстрация | Запустить**. Появится заготовка обработчика соответствующего события Run1Click. Текст обработчика приведен далее.

```
procedure TForm1.Run1Click(Sender: TObject); {запуск и остановка
демонстрации слайдов}
begin
if Timer1.Enabled=False then {если демонстрация была остановлена}
begin
  Run1.Caption:='Стоп'; {меняем название пункта меню на "Стоп"}
  Timer1.Enabled:=True; {запустили таймер}
end else {иначе, если демонстрация уже идет}
begin
  Run1.Caption:='Запуск';
{меняем название пункта меню обратно на "Запуск"}
       Timer1.Enabled:=False;  {таймер остановлен}
end;
end;
```

Но это не все: нужно еще создать обработчик события, которое наступает при срабатывании таймера, иначе ничего происходить не будет. Для этого надо дважды щелкнуть на иконке таймера на форме, и в появившуюся заготовку события Timer1Timer вписать ту же самую единственную строку:

```
. . . . .
begin
  Loadfile;
end;
. . . . .
```

Кстати, а как закрывать программу? Традиция предписывает введение специального подпункта в главном меню формы (обычно внутри пункта **File**), но

автору кажется это излишним. Windows и так предоставляет достаточно удобных возможностей (через системное или контекстное меню, нажатием на системную кнопку-крестик, наконец, клавиатурной комбинацией клавиш <Alt>+<F4>), и дублировать эти функции совершенно ни к чему. Единственно, когда это целесообразно — расчет на совершенных "чайников", которые не имеют представления о системных возможностях Windows. Другой случай — если вы делаете настолько оригинальное окно, что системную кнопку там найти непросто или ее вообще нет. Во всех этих случаях для закрытия приложения следует вызвать по нужному событию метод `Form1.Close`.

Теперь можно считать, что программа закончена. Запустите ее для проверки. Находится она в одном-единственном файле SlideShow.exe и для переноса ее в другое место достаточно переписать только этот файл.

Заметки на полях

По умолчанию Delphi присваивает всем новым проектам свой значок — нельзя ли его изменить? Чтобы получить оригинальную иконку, ее можно нарисовать (для этого в комплект пакета входит специальный графический редактор Image Editor, с которым мы еще не раз будем иметь дело), а можно просто подобрать из какой-нибудь коллекции. Для того чтобы сменить иконку, надо обратиться к пункту меню **Project | Options**, открыть там закладку **Application** и нажать на кнопочку **Load Icon**. Новый значок внедрится в исполняемый файл. Для образца программы, которая располагается на прилагающемся диске, иконка была заменена именно таким образом.

ГЛАВА 3

Окна настежь

Нестандартное закрытие и восстановление окна программы. Иконка в Tray Bar

> Оконные коробки могут быть цельными или отдельными. Если два переплета открываются внутрь, то цельную или отдельные коробки делают таким образом, чтобы летние переплеты были меньше зимних и свободно открывались.
>
> *А. М. Шепелев, "Ремонт квартиры своими силами"*

Вы все неоднократно встречали программы, которые при нажатии на системную кнопку-крестик (в правом верхнем углу окна) не закрываются, а сворачиваются в иконку в System Tray[1] (это обычно переводят как "системное меню" или "системная панель", но во избежание семантической путаницы — слишком много всего "системного" получается — мы будем называть его просто Tray). Некоторые программы делают то же самое и при нажатии на кнопку минимизации. Разберемся сначала, зачем и когда это может понадобиться, в предположении, что создается серьезное приложение для работы, а не просто так.

Сворачивание в иконку вместо закрытия может потребоваться, если мы хотим, чтобы приложение всегда было под рукой со всеми сохраненными настройками и данными, но при этом не "светилось" в списке задач (доступных при перелистывании через <Alt>+<Tab>) и не мешалось на панели задач. Получается нечто вроде так называемой "резидентной программы". В DOS название "резидентная программа" означала просто постоянное присутствие ее в памяти, но для многозадачной среды это название во многом обессмысли-

[1] Строго говоря, термин "сворачиваются в иконку" неправильный — окно никуда не "сворачивается", просто в Tray помещается иконка, а окно скрывается, но выражение довольно точно описывает то, как это выглядит внешне, поэтому мы им будем пользоваться.

вается. Типичный признак "настоящей" резидентной программы в Windows, когда она что-то делает полезное в фоновом режиме, а не просто ожидает, пока к ней обратятся. С этой точки зрения резидентной, например, будет программа с использованием ловушек (hook, см. *главу 6*). Но с точки зрения пользователя чем хуже программы, просто реагирующие на некую горячую клавишу? Так что это есть лишь вопрос терминологии.

Отметим, что злоупотреблять присутствием вашей программы в Tray не следует. Сейчас модно чуть ли не любую программу запускать в Tray и ничего, кроме раздражения, это вызвать не может. Лично у меня там более десятка *нужных* мне приложений (и далеко не всем им позарез надо светиться в Tray, просто выхода нет), и если бы я позволял обустраиваться там еще и MP3-плееру, интернет-пейджеру, программам для сканера, принтера, цифрового фотоаппарата, и прочим далеко не ежеминутно нужным мне вещам, у меня просто не осталось бы свободного места. Но ввести такую возможность *опционально* — для многих задач весьма полезная вещь. Единственное, что при этом требуется — в контекстном меню иконки обязательно должен быть пункт **Удалить** или **Закрыть**, чтобы не приходилось копаться в настройках и перезапускать компьютер для избавления от какого-нибудь назойливого "агента" или службы онлайнового обновления.

Что же касается минимизации в Tray (т. е. выполнения соответствующего действия при сворачивании, а не закрытии программы), то это для обычной программы требуется нечасто. Если вы не хотите, чтобы программа вообще когда-нибудь оказалась в списке задач, проще совсем убрать (точнее, дезактивировать) кнопку минимизации еще на этапе проектирования. И, тем не менее, позже мы покажем, как это делается, а в дальнейшем — как это можно использовать. И начнем с наиболее простого случая — сворачивания в иконку при потере приложением фокуса, для того чтобы проиллюстрировать, как работает механизм размещения иконки в Tray и восстановления окна, а после перейдем к более практическим применениям.

Сворачивание приложения в Tray Bar при потере фокуса

Наметим план действий в этом простейшем случае. Нам надо: при потере фокуса свернуть программу в иконку, при щелчке на иконке вновь распахнуть окно. Сразу заметим, что последнее сделать очень просто — достаточно вызвать метод `Show` формы. А вот первое действие раскладывается как минимум на два: собственно размещение иконки и скрытие окна.

Перед тем как вносить изменения, скопируйте проект SlideShow из папки Glava2 в новую папку, на прилагаемом диске это папка с именем "1" внутри

папки Glava3 (с соответствующими исправлениями в файле DSK, как описано во введении). Не забудем изменить номер версии — пусть это будет 1.10 (еще раз напомню, что изменения вносятся в двух местах: в заголовке и в пункте меню **Project | Options | Version Info**).

Заметки на полях

Далее мы часто будем употреблять термин "дескриптор", что требует комментариев. Термин handle, который имеется в виду, дословно переводится как "приводная ручка". В русскоязычной литературе по программированию его переводят как "описатель" [1], "дескриптор" [2,3], иногда "идентификатор". На взгляд автора, подкрепленный авторитетом автора книги [13], первые два случая только затемняют смысл термина. Основное предназначение "хендла" — однозначно идентифицировать объект, к которому производится обращение, и тут лучше всего подходит по смыслу слово "идентификатор", "указатель", тем более, что, как и обычный указатель (pointer) в языке Pascal, он есть просто число (индекс в обслуживаемой Windows таблице указателей на физические адреса памяти). Но оба указанных термина уже заняты. В свою очередь и слово "дескриптор" имеет свой отдельный более узкий смысл — как некая структура данных, действительно "описатель", например, блока в памяти (segment handle) или файла (file handle), или физического устройства (device handle), в этом смысле наш handle есть указатель на подобную структуру. Однако не будем нарушать традиций, просто помните, что употребление этого термина с семантической точки зрения неоднозначно.

Разберем сначала самое сложное: манипуляции с иконкой. Для этого в Win32 API имеется функция `Shell_NotifyIcon` с двумя параметрами: сообщением о том, что собственно нужно делать с иконкой при размещении в Tray, и дескриптором структуры `TNotifyIconData` (`PNotifyIconData` в нотации Windows API). Сообщение (как и вообще любое сообщение Windows) имеет тип "двойное слово" (`DWord`), что вполне можно заменить на `integer` (`Longint`) в Object Pascal. В таких случаях надо быть внимательным — несмотря на формальное совпадение типов, Delphi иногда выдает сообщение об ошибке, если вы указали не тот тип при обращении к API. Само сообщение может иметь всего три предопределенных значения: `NUM_ADD`, `NUM_DELETE` и `NUM_MODIFY`, смысл которых понятен без особых пояснений.

Заметки на полях

Отметим тут еще одно преимущество языка Pascal перед C-ориентированными языками: в нем заглавные и строчные буквы в идентификаторах заведомо не различаются, и мы не задумываясь можем писать мнемонические наименования, как нам удобно. В то время, как в C это зависит от воли разработчиков конкретной программы и часто служит источником неудобств: так, все разработчики сайтов знают, что index.html, Index.html или INDEX.HTML есть совершенно разные файлы, при совершенной бессмыслице этого с практической точки зрения. В тексте данной книги я буду придерживаться примерно того же написания, которое предлагается в справке по Win32, просто чтобы не смущать читателя, но указанную особенность следует иметь в виду.

Структура TNotifyIconData содержит несколько составляющих. Вначале (как обычно для подобных структур) идет размер самой структуры cbSize. Затем последовательно: дескриптор Wnd типа hWnd окна, с которым связана иконка, идентификатор самого значка uID типа integer (значков может быть несколько), переменная флагов UFlags, идентификатор определяемого нами сообщения от иконки приложению uCallBackMessage (и то и другое типа integer), наконец, дескриптор иконки HIcon (типа hIcon — извините за тавтологическое использование идентификаторов, но у разработчиков API подобное встречается сплошь и рядом) и текст всплывающей подсказки SzTip (подробности см. в справке по Win32, а также в [1]). Те переменные, которые участвуют в нашем процессе, следует объявить в качестве глобальных:

```
var
. . . . .
       Ico_Message: integer=wm_User; {сообщение}
       noIconData: TNotifyIconData; {дескриптор структуры}
       HIcon1: hIcon; {дескриптор иконки}
       FHandle: HWnd; {дескриптор окна (формы)}
```

Обратите внимание на то, что мы сразу инициализировали сообщение от иконки приложению значением wm_User. В принципе можно и другое значение, но нам важно, чтобы оно не совпало с уже имеющимися в данном приложении значениями, а инициализация его значением 0 (как было бы, если бы мы оставили его вовсе без инициализации) недопустима. Подробнее о пользовательских сообщениях мы еще поговорим.

Для того чтобы вызвать какую-нибудь функцию API, нужно в предложение **uses** в начале текста программы включить модуль ShellApi. После этого у нас все готово для написания процедуры размещения иконки, осталось только решить, к какому событию ее привязать. При потере фокуса формой должно произойти событие onDeactivate, однако на самом деле не все так просто — происходит лишь событие, связанное с приложением в целом, и чтобы происходило именно событие деактивации конкретного окна, нам требуется это явно указать. Удобно сделать это в самом начале программы: создать обработчик события onCreate и включить в него нужный оператор:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
Application.onDeactivate := FormDeactivate;
end;
```

Нам еще нужно разместить в памяти дескриптор окна, чтобы было куда посылать сообщения и получить дескриптор иконки. Первое делается с по-

мощью функции AllocateHWnd, а второе заключается в том, что мы скопируем иконку приложения через функцию CopyIcon. Теперь мы спокойно можем писать обработчик события onDeactivate формы:

```
procedure TForm1.FormDeactivate(Sender: TObject);
begin
FHandle := AllocateHWnd(WndProc); {получаем дескриптор окна}
HIcon1:=CopyIcon(Application.Icon.Handle);
                                        {получаем дескриптор иконки}
with noIconData do begin
cbSize:=Sizeof(TNotifyIconData); {размер структуры}
Wnd:=FHandle; {дескриптор окна}
uID:=0; {единственная иконка}
UFlags:=NIF_MESSAGE or NIF_ICON or NIF_TIP; {взводим все флаги}
SzTip:='SlideShow'; {всплывающая подсказка}
HIcon:=HIcon1; {дескриптор иконки}
uCallBackMessage:=Ico_Message;
                              {Определяемое пользователем сообщение}
end;
Shell_NotifyIcon(NIM_ADD,@noIconData); {создали иконку}
Form1.Hide; {скрыли окно}
end;
```

Для параметра UFlags мы указали все возможные флаги, т. е. мы будем посылать пользовательское сообщение (NIF_MESSAGE), рисовать иконку (NIF_ICON) и привязывать к ней всплывающую подсказку (NIF_TIP).

Но это только полдела — если мы теперь попробуем запустить приложение, то при потере фокуса оно свернется в иконку и останется там навсегда. Надо его как-то оттуда извлечь. Для этого нам нужно обработать сообщение Ico_Message, которое посылается приложению каждый раз, когда в области иконки в Tray происходит что-то, связанное с мышью. Как его поймать? Для этого можно перекрыть или переопределить метод WndProc формы, который служит для вызова оконной процедуры (т. е. главной процедуры окна по обработке сообщений — подробнее см. в *главе 5*). Для переопределения достаточно объявить свою процедуру по такому шаблону:

```
type
TForm1 = class(TForm)
       . . . . .
 procedure WndProc(var Message: TMessage);
       . . . . .
```

Включим вручную соответствующую строку в секцию объявления типов в интерфейсной части модуля slide, туда же, где находятся все остальные про-

цедуры. Запись **Message** типа TMessage основана на определенной в Windows структуре Msg (TMsg в нотации Delphi — подробнее со структурой Msg можно ознакомиться в любой справке по WinAPI, см. также *главу 5*), которая содержит параметры сообщения, для каждого их типов определенных в Windows сообщений эти параметры имеют свой особенный смысл. Из них нас сейчас будет интересовать lParam (нам еще предстоит очень близко познакомиться и с этим, и некоторыми другими параметрами настоящей структуры). Для данного типа сообщения lParam будет содержать номер, соответствующий идентификатору события, вызвавшего наше сообщение от иконки к окну. Теперь также полностью вручную (или через нажатие клавиш <Ctrl>+<Shift>+<C>) напишем текст самой процедуры в исполняемой части модуля (в любом месте после слова **implementation**):

```
procedure TForm1.WndProc(var Message: TMessage); {обработка
пользовательских сообщений}
begin
  if Message.Msg = Ico_Message then
  begin
   if Message.lParam=WM_LBUTTONUP then
              {если была отпущена кнопка}
   begin
    Form1.Show; {восстанавливаем окно}
    Application.BringToFront; {помещаем его поверх всех окон}
    DeallocateHWnd(FHandle);
              {убираем из памяти дескриптор окна}
    Shell_NotifyIcon(NIM_Delete,@noIconData); {удаляем иконку}
    Application.ProcessMessages;
              {на всякий случай обрабатываем системные сообщения}
   end;
  end;
end;
```

Вот теперь мы закончили — можно запустить приложение (<F9>). Как только оно потеряет фокус, окно просто исчезнет, а в Tray появится иконка со всплывающей подсказкой "SlideShow". При щелчке на этой иконке она пропадет, а окно восстановится поверх всех открытых на данный момент окон. В этом случае восстановление произойдет при отпускании левой кнопки (сообщение WM_LBUTTONUP). Отметим, что всегда, кроме каких-то специальных случаев, при обработке событий мыши принято использовать событие именно отпускания, а не нажатия (в отличие от событий клавиатуры).

Имея такой базовый набор отлаженных действий, мы можем теперь перейти к случаям, чаще встречающимся на практике.

Сворачивание приложения в Tray Bar вместо закрытия

Процесс закрытия приложения заключается в вызове метода Close главной формы — неважно, закрываете ли вы его через нажатие на системную кнопку-крестик или каким-то другим путем. Для того чтобы управлять процессом закрытия, в методе предусмотрена булевская переменная CanClose (по умолчанию равная True) и ряд последовательных событий, в которых эта переменная участвует. Первым возникает событие onCloseQuerry. Если в его обработчике указать CanClose:=False, то все последующие события не произойдут — приложение не закроется. Ясно, что это нужно делать по какому-то условию, иначе приложение вообще нельзя будет закрыть. В качестве примера можно привести часто встречающийся случай закрытия программы с предварительным запросом (см. [6]; о функции MessageBox см. предыдущую главу):

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
var st:string;
begin
 st:='Вы действительно хотите закрыть программу?';
 if Application.MessageBox(Pchar(st),'Warning',MB_YESNO) = IDNO
 then CanClose:=False else CanClose:=True;
end;
```

Нам же нужно, чтобы вместо закрытия исполнялась та процедура, которая описана ранее — в Tray возникала иконка, а главное окно формы исчезало. К сожалению, переменная CanClose не является глобальной, потому ей нельзя присвоить значение False заранее, так что придется изворачиваться. Мы поступим следующим образом: объявим глобальную булевскую переменную-флаг mayClose (типа boolean) и инициализируем ее значением False:

```
var
. . . . .
mayClose: boolean=False;
```

Чтобы сохранить все варианты на будущее, перед тем как вносить изменения, снова скопируйте проект SlideShow из папки Glava3\1 в новую папку (на прилагаемом диске это папка Glava3\2, не забудьте исправить файл DSK — больше я об этом упоминать не буду) и изменить номер версии (пусть это будет 1.11). После чего удалите из текста обработчики событий onDeactivate и OnCreate (они нам здесь не понадобятся).

Обработчик события onCloseQuerry тогда будет выглядеть так:

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

```
 if mayClose=False then
 begin
  CanClose:=False; {не закрываем программу}
  mayClose:=True; {а при следующем запросе закроем}
  FHandle := AllocateHWnd(WndProc); {получаем дескриптор окна}
  HIcon1:=CopyIcon(Application.Icon.Handle);
                                    {получаем дескриптор иконки}
  with noIconData do begin
    cbSize:=Sizeof(TNotifyIconData); {размер структуры}
    Wnd:=FHandle; {дескриптор окна}
    uID:=0; {единственная иконка}
    UFlags:=NIF_MESSAGE or NIF_ICON or NIF_TIP;
                                    {взводим все флаги}
    SzTip:='SlideShow'; {всплывающая подсказка}
    HIcon:=HIcon1; {дескриптор иконки}
    uCallBackMessage:=Ico_Message;
                    {определяемое пользователем сообщение}
  end;
  Shell_NotifyIcon(NIM_ADD,@noIconData); {создали иконку}
  Form1.Hide; {скрыли окно}
 end else {если можно закрыть, т. е. если mayClose=True}
 begin
       DeallocateHWnd(FHandle); {убираем из памяти дескриптор окна}
       Shell_NotifyIcon(NIM_Delete,@noIconData); {удаляем иконку}
   CanClose:=true; {можно закрывать}
 end;
end;
```

Логика этой процедуры такова: сначала флаг `mayClose` имеет значение `False`, и потому при попытке закрытия будет выполнена процедура по этому условию, т. е. создастся иконка, закроется окно, а само приложение не закроется (`CanClose=False`), но при следующем запросе на закрытие выполнится уже процедура по условию `mayClose=True`.

Нам теперь нужно сделать две вещи: во-первых, организовать процедуру восстановления окна, во-вторых, обеспечить возможность закрытия программы.

Самый простой путь — например, модифицировать процедуру `WinProc` так, чтобы при щелчке левой кнопкой окно восстанавливалось, а при щелчке правой — приложение закрывалось. Но это неочевидное и потому неграмотное решение — правильный путь лежит через создание контекстного меню. В палитре Standard находим компонент `PopupMenu`, переносим его на форму и создаем в нем два пункта: `Восстановить` (название его пусть будет `PopupRestore1`) и `Закрыть` (`PopupClose1`). Теперь переписываем процедуру `WinProc` следующим образом:

```
procedure TForm1.WndProc(var Message: TMessage); {обработка
пользовательских сообщений}
begin
  if Message.Msg = Ico_Message then
  begin
   if Message.lParam=WM_LBUTTONUP then
                              {была отпущена левая кнопка}
   begin
    mayClose:=False; {теперь снова нельзя закрыть программу}
    Form1.Show; {восстанавливаем окно}
    Application.BringToFront; {помещаем его поверх всех окон}
    DeallocateHWnd(FHandle);
                              {убираем из памяти дескриптор окна}
    Shell_NotifyIcon(NIM_Delete,@noIconData); {удаляем иконку}
    Application.ProcessMessages;
              {на всякий случай обрабатываем системные сообщения}
   end;
   if Message.lParam=WM_RBUTTONUP then
                              {была отпущена правая кнопка}
   Form1.PopupMenu1.Popup(Screen.Width-32,Screen.Height-32);
                                        {вызываем всплывающее меню}
  end;
end;
```

Здесь при восстановлении окна флаг `mayClose` опять принимает значение `False`, так что при новой попытке закрытия все опять повторится сначала. Всплывающее меню для простоты мы расположили на фиксированном расстоянии в 32 пиксела от нижнего и правого края экрана — примерно там, где находится наша иконка. И наконец, напишем два обработчика для пунктов **Восстановить** и **Закрыть**:

```
procedure TForm1.PopupRestore1Click(Sender: TObject);
begin
  mayClose:=False; {теперь снова нельзя закрыть программу}
  Form1.Show; {восстанавливаем окно}
  Application.BringToFront; {помещаем его поверх всех окон}
  DeallocateHWnd(FHandle); {убираем из памяти дескриптор окна}
  Shell_NotifyIcon(NIM_Delete,@noIconData); {удаляем иконку}
  Application.ProcessMessages;
              {на всякий случай обрабатываем системные сообщения}
end;

procedure TForm1.PopupClose1Click(Sender: TObject);
begin
Form1.Close; {закрываем программу - для этого все готово}
end;
```

Первый из них, как видите, просто дублирует то, что происходит при щелчке левой кнопкой на иконке в Tray и сделан больше "для порядка", чем для каких-то практических нужд. Теперь наша программа при попытке штатного закрытия будет сворачиваться в иконку, откуда ее можно восстановить сразу двумя путями, а закрыть ее можно будет только через контекстное меню. Попробуйте самостоятельно модифицировать программу так, чтобы при щелчке правой кнопкой (при ее нажатии) выскакивало всплывающее меню, а при ее отпускании срабатывал один из пунктов этого меню (**Восстановить** или **Закрыть**), в зависимости от того, где именно расположен курсор при отпускании — это сократит необходимое количество щелчков, но потребует некоторого усложнения программы с использованием функций определения координат мыши. Подробнее я здесь не буду на этом останавливаться — потом мы будем говорить о некоторых нюансах использования мыши и клавиатуры. Только еще одно замечание: наше всплывающее меню не оказалось истинно "всплывающим": при потере фокуса оно останется на экране и будет "торчать" там до тех пор, пока мы не щелкнем какой-либо из его пунктов. А это не очень корректно, т. к. щелкнуть на иконке можно просто по ошибке. Как сделать, чтобы оно в таком случае убиралось автоматически? Создадим обработчик события onPopup этого меню, и впишем туда одну-единственную строку:

```
procedure TForm1.PopupMenu1Popup(Sender: TObject);
begin
 SetForegroundWindow(Form1.Handle);
 {меню убирается автоматически при потере фокуса}
end;
```

Кстати, присваивание горячих клавиш пунктам всплывающего меню в данном случае просто ничего не даст — их нажатие будет проигнорировано, управлять можно только мышью. Для того чтобы клавиши работали тоже, их нажатие нужно перехватывать — этим мы займемся позже.

Сворачивание приложения в Tray Bar вместо минимизации

Это действие нам понадобится в дальнейшем *(см. главу 10)*. Реализация его сложнее, здесь нет столь же простого пути, как при дефокусировке и при закрытии формы — отдельно события onMinimize для формы не существует. Существует только событие Application.onMinimize для приложения в целом — как же его использовать?

Возьмем наш проект SlideShow, опять скопируем из папки Glava3\2 его в новую папку (папка Glava3\3 на прилагаемом диске), изменим номер версии

на 1.12, и дополним его так, чтобы сворачивание в иконку происходило при нажатии кнопки минимизации. Для того чтобы "отловить" событие минимизации, сначала создадим процедуру, которая будет выполняться при минимизации. Назовем ее `OnMinimizeProc` и объявим в секции **`private`** интерфейсной части модуля:

```
private
 { Private declarations }
 procedure OnMinimizeProc(Sender:TObject);
```

Опять создадим обработчик события `onCreate` со следующим содержимым:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 Application.onMinimize := OnMinimizeProc;
end;
```

Наконец, создадим саму процедуру `OnMinimizeProc`, которая будет просто вызывать метод закрытия и ничего более (т. е. функциональность кнопок закрытия и минимизации совпадет):

```
procedure TForm1.OnMinimizeProc(Sender: TObject);
begin
Form1.Close;
end;
```

Казалось бы, все в порядке, но это не совсем так. Если вы выполните написанную программу по шагам, то убедитесь, что перед тем, как вызывается наша функция `OnMinimizeProc`, минимизация все же происходит, только затем окно исчезает. А восстановление окна вместо штатного щелчка на заголовке в панели задач делается совсем другим способом, поэтому окно просто "не знает", что его восстановили. И если вы после восстановления попробуете еще раз щелкнуть на кнопке минимизации, события `onMinimize` не произойдет — окно и не шелохнется. Не будем ругаться на программистов, которые не предусмотрели такую ситуацию, а просто имитируем восстановление: в процедуре вызова окна на экран (и по щелчку мыши, и по пункту **Восстановить**) вызовем стандартный метод `Application.Restore`. Его можно вставить после вызова метода `Form1.Show`. Кстати, если вы попробуете, наоборот, убрать последний — ситуация станет аналогичной: восстанавливаться и минимизироваться окно будет, а вот метод `Hide` работать перестанет.

ГЛАВА 4

Погрузочно-разгрузочные работы

Предотвращение повторного запуска и загрузка с заставкой

> Окна требуют постоянного ухода и периодического ремонта. В плохо открывающихся створках оконных переплетов расшатываются соединения, уменьшается плотность прилегания стекол к фальцам, что увеличивает потери тепла.
>
> *С. Иванчиков, "Учись делать сам"*

В этой главе мы займемся вещами, которые относятся к начальному моменту загрузки приложения. Вы часто встречали программы, которые при загрузке сразу сворачиваются в Tray. Вообще это имеет смысл делать только для упомянутых в предыдущей главе резидентных программ — то есть таких, которые должны работать либо лишь иногда, либо непрерывно, но в фоновом режиме. Для программ общего назначения, типа нашего просмотрщика слайдов, устраивать сворачивание в иконку непосредственно после запуска не имеет смысла — вся работа протекает при развернутом окне. Тем не менее для простоты изложения мы продемонстрируем это действие на примере нашего приложения, чтобы не отвлекаться сейчас на реализацию других важных свойств резидентных программ, которыми мы займемся позже.

Но начнем мы издалека. Для приложения, которое находится в свернутом состоянии, особенно критичен момент, связанный со случайной повторной загрузкой — велика вероятность, что пользователь просто забудет о том, что приложение уже запущено и попробует запустить его заново.

Предотвращение повторного запуска приложения

Повторный запуск приложения иногда может наделать много неприятностей — например, если вы работаете с одним конкретным файлом или с од-

ной конкретной базой данных. Позже, когда мы будем говорить о различных способах доступа к файлам из приложения, мы приведем пример такого случая. Чаще же всего повторный запуск просто раздражает, отнимая время и ресурсы компьютера. Разумеется, есть случаи — и их достаточно много — когда возможность повторного запуска предусмотрена специально (так, вы можете запустить сразу несколько экземпляров Delphi, или несколько Word), а иногда и жизненно необходима — скажем, для Internet Explorer работа с несколькими ресурсами параллельно производится только через запуск все новых и новых его экземпляров. Но во всех случаях, когда запуск второго, третьего и т. д. экземпляров программы возможен, это должно быть осмысленным решением, а не наоборот — когда предотвращением повторного запуска занимаются только в самых критичных случаях.

Для того чтобы предотвратить повторный запуск, проще всего в самом начале загрузки программы определять, какие приложения уже запущены. Если в списке встретится наше приложение, мы делаем вывод, что оно уже было запущено и предпринимаем какие-либо действия — выводим сообщение или просто прерываем загрузку. Тут сразу возникает несколько вопросов. Как составить список запущенных приложений? Что конкретно означают слова "встретится наше приложение", как его идентифицировать? И, наконец, где именно располагать нашу процедуру? Ответим на них в порядке поступления.

Список запущенных приложений можно получить с помощью функции `GetWindow`[1]. Она имеет два параметра: дескриптор окна, от имени которого производится обращение (знакомого нам типа `hWnd`) — т. е. в данном случае это дескриптор нашего приложения, и параметр (типа `uCmd`), указывающий на отношения между нашим приложением и тем, которое мы ищем. Нас будут интересовать три возможных значения этого параметра: `GW_HWNDFIRST` — идентифицирует окно того же самого типа, что и наше, которое является самым верхним в списке запущенных приложений, `GW_HWNDNEXT` — следующее в списке окно, и `GW_OWNER` — идентифицирует дочерние окна, которые нас не интересуют. Возвращает функция `GetWindow` дескриптор найденного окна.

Идентифицировать приложение мы будем по его имени. Это не то же самое, что имя файла приложения или заголовок окна формы. Хотя имя приложения по умолчанию будет соответствовать имени файла, но оставлять это в таком виде неправильно (файл ведь можно и переименовать). Специально установить имя можно еще на стадии проектирования (через пункт **Project | Options | Application**, там же, где устанавливается иконка приложения). Если

[1] В терминологии Windows "окно" и "приложение" есть синонимы, поэтому не удивляйтесь тому, что мы далее будем употреблять эти термины вперемешку. Чтобы отличить приложение от собственно его окна в прямом смысле слова, для последнего в Delphi употребляется термин "форма".

вы установите имя "SlideShow" через указанный пункт меню, то увидите, что в тексте основной программы (файл SlideShow.dpr) автоматически появится строка:

```
Application.Title := 'SlideShow';
```

А получить имя приложения (оно же название главного окна), зная дескриптор его окна, можно через функцию GetWindowText. Имейте в виду, что возвращаемое название чувствительно к регистру букв — "SlideShow" и "Slideshow" будут идентифицировать совершенно разные приложения!

Наконец, разберем вопрос о том, где располагать нашу процедуру. Очевидно, что располагать ее в тексте модуля slide.pas (по какому-либо событию) невыгодно — тогда, чтобы ее выполнить, нам надо по крайней мере начать создание формы, а ради чего все затевалось? Поэтому нам придется вмешаться в основную программу.

Заметки на полях

Неопытные программисты часто путают приложение в целом и принадлежащую ему форму (главный модуль). Даже если форма всего одна (Form1), она формально представляет собой лишь один из модулей приложения, которых по идее может быть много, а может и не быть вовсе. Текст основной программы — приложения в целом — формируется автоматически и содержится в файле DPR, на который начинающие программисты обычно просто не обращают внимания. Но это не значит, что вы не можете его модифицировать, чем мы и займемся. *(Подробную информацию о программах и принадлежащих им окнах см. в следующей главе.)*

Текст, который нужно выполнить, мы вставим в основную программу до того, как начнется выполнение процедуры создания формы, т. е. перед строкой Application.CreateForm(TForm1, Form1). Как всегда, скопируем последний вариант проекта SlideShow из папки Glava3\3 в новую папку (на диске это папка с именем "1" внутри папки Glava4) и изменим номер версии (пусть это будет 1.13). Кстати, из изложенного ранее следует, что если вы хотите, чтобы версии воспринимались при запуске, как разные программы, нужно внести изменения и в название приложения (и, желательно, также сменить иконку) — однако здесь все версии делают одно и то же, поэтому так поступать мы не будем. Модифицированная программа (файл SlideShow.dpr) у нас будет выглядеть так (обратите внимание, что нам пришлось вставить ссылку на модуль Windows в предложение **uses**) — листинг 4.1.

Листинг 4.1. Модифицированная программа SlideShow.dpr

```
program SlideShow;

uses
Forms, Windows,
slide in 'slide.pas' {Form1};
```

```
{$R *.res}
var pbuff:array[0..127] of char; {буфер для текста}
var dWin: HWnd; {дескриптор}
var st:string; {вспомогательная строка}

begin
Application.Initialize;
Application.Title := 'SlideShow';
dWin:= GetWindow(Application.Handle, GW_HWNDFIRST);
while dWin <> 0 do
begin
 if (dWin <> Application.Handle) and
        {собственное окно игнорируем}
  (GetWindow(dWin, GW_OWNER) = 0) and
        {дочерние окна игнорируем}
  (GetWindowText(dWin, pbuff, sizeof(pbuff)) <> 0)
        {без названия игнорируем}
 then
 begin
   GetWindowText(dWin, pbuff, sizeof(pbuff));
       {получаем текст названия приложения}
   st:=string(pbuff); {переводим его в строку}
   if st='SlideShow' then
   begin
    st:='"Двое пернатых в одной клетке не живут." @A.Лебедь';
    Application.MessageBox(Pchar(st),'',MB_OK);
        {выводим предупреждение}
    exit; {прерываем программу}
   end;
 end;
 dWin:= GetWindow(dWin, GW_HWNDNEXT);
        {ищем следующее приложение из списка}
end;
Application.CreateForm(TForm1, Form1);
Application.Run;
end.
```

При попытке повторного запуска программа найдет в списке запущенных приложений свое имя и выдаст сообщение, представляющее собой знаменитое высказывание покойного генерала. После нажатия **Ok** запуск программы будет прерван. Вывод сообщения `'"Двое пернатых в одной клетке не живут." @A.Лебедь'` здесь сделан только для иллюстрации: разумеется, никаких сообщений можно и не выдавать, а сразу выполнить команду `exit`; с другой сто-

роны, можно просто организовать выбор, запускать второй экземпляр программы или нет.

Delphi не позволит вам запустить программу из своей среды повторно. Для того чтобы проверить работу программы в этом режиме, надо произвести компиляцию, полученный исполняемый файл SlideShow.exe скопировать в любую другую папку, запустить его оттуда, а затем попробовать еще раз запустить приложение из среды Delphi. Отметим также, что использование сразу двух видов представления одной и той же строковой переменной (как массив символов pbuff и как паскалевская строка st) не является необходимым — можно было бы и сократить программу в этом отношении, просто работа с паскалевскими строками нагляднее.

Несколько слов о том, какие еще действия можно предпринять при обнаружении повторного запуска вашего приложения. Если это самый простой случай обычного окна, которое не сворачивается ни в какие иконки, то целесообразно вместо вывода сообщения MessageBox вставить следующую пару операторов:

```
ShowWindow(dWin,SW_SHOWNORMAL);
BringWindowToTop(dWin);
```

По вызову этих функций первый экземпляр приложения будет максимизирован (если он был свернут) и — по идее — выведен вперед всех окон, а попытка запуска второго будет пресечена последующим оператором exit (по этому образцу работает, например, TheBat!). Аналогичную функциональность предоставляют функции SetWindowPos, SetForegroundWindow и др. (см. справку по Win32). Здесь функция BringWindowToTop добавлена для надежности — в принципе и ShowWindow и другие указанные функции в одиночку должны активировать окно, но механизм вывода приложений в активное состояние в Windows настолько запутан, что в некоторых случаях и двух совместно используемых функций может не хватить — окно восстанавливается, но вперед не выводится. Запутанность имеет оправдания (редкий случай!): в ранних версиях Windows разработчики прикладных программ настолько злоупотребляли возможностью вывести свое приложение поверх всех окон, что создателям Windows пришлось ограничить их активность на этом поприще. Вместо активизации — или вместе с ней — можно заставить заголовок окна на панели задач мигать — если вы хотите воспользоваться этой возможностью, то ищите в справке описание функции FlashWindow. В нашем же случае, когда окно свернуто в иконку, ни одна из подобных функций не сработает, потому мы не будем далее задерживаться на этом вопросе.

Заметки на полях

В ряде источников (см. например, [38]) можно встретить более простой метод решения проблемы повторного запуска, состоящий из вызова всего одной функции FindWindow(<*класс окна*>,<*заголовок окна*>). Мы будем использо-

вать в дальнейшем эту функцию для поиска приложения *(например, см. главу 7)*, однако приведенный метод хотя и более громоздкий, зато не требует знания класса окна, потому является более простым для понимания и применения. Заметим, что в таком виде данная процедура может пригодиться и для других надобностей *(например, см. главу 14)*. Во избежание возможных ошибок при ее использовании, напомню еще раз, что любая процедура поиска окна по его названию чувствительна к регистру букв. Есть и не столь "лобовые" методы решения задачи идентификации своей программы — выбранный нами не слишком хорош тем, что в принципе может потребоваться менять по ходу дела имена приложения и тем более отдельного его окна. Другой метод предлагает, скажем, Юрий Зотов (**http://www.delphikingdom.ru/asp/viewitem.asp?catalogid=903**), и состоит он в использовании memory mapped files — механизма отображения файлов на память. То есть в самом начале программы вы создаете в общей памяти некий уникальный файл, а затем просто проверяете — существует ли он уже. Но я считаю, что во всех случаях нужно выбирать самое простое и понятное решение.

Демонстрация заставки

Сворачивание в иконку сразу при запуске, как правило, должно сопровождаться демонстрацией окна-заставки — иначе пользователь может просто не заметить того, что программа запущена. Это правило не абсолютно — демонстрация заставки с точки зрения программы, как правило, есть совершенно пустое времяпрепровождение, потому что она замедляет запуск, а это может раздражать. Совершенно лишней является демонстрация заставки для программ, которые всегда работают в фоновом режиме и загружаются через автозапуск — например, файрволов (firewall) или антивирусов. С другой стороны, многие грамотно написанные крупные приложения (Photoshop, MS Word, сама Delphi) используют демонстрацию заставки с пользой — чтобы иметь время загрузить многочисленные дополнительные библиотеки и модули (плагины). В остальных случаях это чистый "декор" и для особо нервных пользователей неплохо предусмотреть возможность отключать демонстрацию заставки при запуске. Разберемся сначала с собственно демонстрацией.

Окно следует разукрасить, для чего сначала нужно создать подходящую картинку в формате BMP. Можно воспользоваться входящим в комплект Delphi редактором Image Editor (пункт меню **Tools**), но это не очень удобная программа для данной цели, и к тому же достаточно "глючная". Через нее можно работать непосредственно с файлами ресурсов, и этим свойством мы в дальнейшем не раз воспользуемся, однако для редактирования иконок и тем более простых bitmap-изображений удобнее воспользоваться специальным редактором. Забегая вперед, отметим, что очень удобной программой для редактирования иконок является LiquidIcon (**www.x2studios.com**), которая, кроме всего прочего, поддерживает "прозрачный" цвет. А нашу картинку-bitmap можно соорудить, воспользовавшись вообще любым графическим редакто-

ром (автор предпочитает Paint Shop Pro). Картинку разместим в файле zastavka.bmp. Размеры картинки надо задавать абсолютные, в пикселах (масштаб в пикселах на дюйм не имеет значения) и для обычного окна-заставки подойдет размер, например, 500×400 точек (рис. 4.1).

Рис. 4.1. Пример оформления окна-заставки

Скопируем опять проект SlideShow из папки Glava4\1 в новую папку (папка Glava4\2), изменим номер версии на 1.20 (изменения на этот раз более существенны) и добавим к нему еще одну форму — `Form2` (пункт **File | New | Form**). Это и будет окно-заставка, соответствующий файл с текстом модуля назовем zastavka.pas. Файл с картинкой (zastavka.bmp) также перенесем в папку с проектом. Теперь в свойствах `Form2` установим для `BorderStyle` значение `bsNone`, для `Position` значение `poDesktopCenter`, а размеры установим равными размерам нашей картинки: `Width = 500` и `Height = 400`. Кроме этого, следует обратиться к пункту **Project | Options** и на закладке **Forms** с помощью кнопки-стрелочки перенести `Form2` из списка автоматически создаваемых форм (**Auto-Create forms**) в список доступных форм (**Available forms**). На эту форму установим компонент `Timer`, в свойствах которого установим `Interval`, равный 5000 (или любой другой по вкусу) — это будет задержка при демонстрации заставки.

А как загрузить картинку в форму? Самый простой путь, которым непременно воспользовался бы каждый начинающий, — разместить на форме компонент `Image` и загрузить в него картинку по умолчанию — так, как мы делали

для компонента Image на главной форме *(см. главу 2)*. Но у любой формы есть свойство Canvas, а это значит, что мы без каких-то дополнительных компонентов можем рисовать прямо на форме — в том числе и размещать на ней готовое изображение. Есть разные способы для того, чтобы это сделать, и мы пойдем следующим путем (вероятно, не самым простым из возможных). Сначала мы в тексте модуля zastavka в секции **public** объявим объект типа BitMap:

```
public
 { Public declarations }
 BmpImage: TBitMap; {объявляем объект типа BitMap}
. . . . .
```

Затем объявим переменную aRect (также в модуле zastavka) — прямоугольник, ограничивающий нашу картинку:

```
var
Form2: TForm2;
aRect: TRect;
. . . . .
```

И, наконец, создадим для Form2 следующий обработчик события onPaint:

```
procedure TForm2.FormPaint(Sender: TObject);
begin
   Form2.BmpImage:=TBitMap.Create; {создаем экземпляр BitMap}
   Form2.BmpImage.LoadFromFile('zastavka.bmp');
         {загружаем в него картинку}
   aRect:=Rect(0,0,500,400); {размеры картинки}
   Form2.Canvas.StretchDraw(aRect,Form2.BmpImage);
         {загружаем картинку в канву формы}
end;
```

Созданный объект типа BitMap надо уничтожать после использования:

```
procedure TForm2.FormDestroy(Sender: TObject);
begin
 BmpImage.Destroy;
end;
```

При таком подходе картинка не внедряется в исполняемый файл, а будет загружаться динамически, поэтому файл с картинкой-заставкой придется "таскать" вместе с исполняемым файлом программы. Рассмотрение вопроса о том, как внедрить картинку в число ресурсов приложения и тем самым избавить нас от необходимости следить за размещением файла, мы отложим, а пока заметим, что такой подход имеет и свои преимущества: картинку всегда

можно заменить без необходимости перекомпиляции проекта — лишь бы сохранилось имя файла. Более того, вы можете поэкспериментировать и убедиться, что собственные размеры картинки не имеют значения — она всегда будет растягиваться до размеров 500×400, независимо от исходных.

Теперь осталось написать собственно процедуру демонстрации. Идея ее состоит в том, чтобы показать окно-заставку, а по первому же событию onTimer удалить ее, и продолжить загрузку основной формы. Сначала мы напишем обработчик события onTimer (для того таймера, разумеется, который находится на Form2):

```
procedure TForm2.Timer1Timer(Sender: TObject);
begin
  Timer1.Enabled:=false;
end;
```

После этого мы перейдем опять к файлу SlideShow.dpr и внесем в него следующие изменения. После процедуры предотвращения повторного запуска у нас идут две строки:

```
Application.CreateForm(TForm1, Form1);
Application.Run;
```

Заменим их на следующий текст:

```
Form2:=TForm2.Create(Application);
{создаем экземпляр формы-заставки - автоматически она не создается}
Form2.Show; {показываем заставку}
while Form2.Timer1.Enabled do Application.ProcessMessages;
      {пустой цикл, пока таймер активен}
Application.CreateForm(TForm1, Form1); {создаем главную форму}
Form2.Free; {уничтожаем заставку}
Application.Run; {запускаем приложение}
```

Запустим приложение и проверим в работе. В дальнейшем мы покажем, как можно создавать задержку без использования компонента-таймера, и, кроме того, заставить заставку не просто исчезнуть, а красиво свернуться или, наоборот, расшириться во весь экран. Итак, заставка работает, но мы пока не добились главного — как сворачивать приложение в иконку сразу при запуске приложения?

Сворачивание в Tray Bar при запуске

Для этого после демонстрации заставки нам нужно сделать два действия: во-первых, сразу поместить иконку в Tray, во-вторых, создать главное окно, но так, чтобы оно оставалось невидимым. Проще всего осуществить второе дей-

ствие: для этого нужно сразу после создания формы (процедура CreateForm) вставить строку Application.ShowMainForm:=False. Но если мы этим ограничимся, то создадим в прямом смысле слова программу-фантом: после демонстрации заставки она исчезнет с глаз долой и ее не останется даже в списке задач, доступных через <Alt>+<Tab>. Прервать ее можно будет только через **Run | Program Reset** (**<Ctrl>+<F2>**), если вы запускали ее из среды Delphi. А вот вопрос для знатоков из клуба "Что? Где? Когда?": что произойдет, если вы попытаетесь прервать эту фантомную программу, разыскав ее в списке всех выполняющихся задач, доступном через <Ctrl>+<Alt>+<Del>? Внимание, правильный ответ: разумеется, она появится в виде иконки в Tray — действие, которое ОС пытается совершить над прерываемой программой, равносильно вызову метода Close, а в прошлой главе мы подробно расписали, что именно наша программа делает в таком случае. Конечно, тогда уже закрыть ее или распахнуть можно будет штатным методом, через всплывающее меню.

Чтобы избежать создания фантомной программы, осталось только включить процедуру создания иконки с самого начала программы (изменения внесем в той же версии 1.20). Для этого мы поступим так: чтобы не повторять текст, относящийся к созданию иконки, мы выделим его в отдельную процедуру (разместив ее в модуле slide.pas, например, перед процедурой Loadfile):

```
procedure CreateMyicon;
begin
FHandle := AllocateHWnd(Form1.WndProc); {получаем дескриптор окна}
HIcon1:=CopyIcon(Application.Icon.Handle);
          {получаем дескриптор иконки}
with noIconData do begin
cbSize:=Sizeof(TNotifyIconData); {размер структуры}
Wnd:=FHandle; {дескриптор окна}
uID:=0; {единственная иконка}
UFlags:=NIF_MESSAGE or NIF_ICON or NIF_TIP; {взводим все флаги}
SzTip:='SlideShow'; {всплывающая подсказка}
HIcon:=HIcon1; {дескриптор иконки}
uCallBackMessage:=Ico_Message;
          {определяемое пользователем сообщение}
end;
Shell_NotifyIcon(NIM_ADD,@noIconData); {создали иконку}
Form1.Hide; {скрыли окно}
end;
```

А процедура FormDeactivate, откуда мы скопировали текст, теперь будет выглядеть так:

```
procedure TForm1.FormDeactivate(Sender: TObject);
begin
CreateMyicon; {создаем иконку в Tray}
end;
```

Для того чтобы основная программа "знала" о существовании такой процедуры, мы должны объявить ее в интерфейсной части модуля slide.pas обычным способом (без всяких этих игр в "private-public"), т. е. прямо перед служебным словом **var**:

```
. . . . .
procedure CreateMyicon;
var
Form1: TForm1;
. . . . .
```

Ссылка на модуль из основной программы у нас есть (**uses** .... slide **in** 'slide.pas'), так что теперь, казалось бы, осталось только включить вызов процедуры CreateMyicon в основной текст:

```
. . . . .
Application.CreateForm(TForm1, Form1); {создаем главную форму}
CreateMyicon; {создаем иконку в Tray}
Application.ShowMainForm:=False; {не показываем главную форму}
Form2.Free; {уничтожаем заставку}
Application.Run; {запускаем приложение}
end.
```

И все вроде бы работает, но с одним изъяном: если попытаться сразу после запуска, ни разу не вызывая окно программы на экран, просто закрыть приложение через всплывающее меню иконки, то вместо закрытия у нас образуется вторая иконка. Как это исправить? Очень просто: если помните, у нас флаг разрешения закрытия программы mayClose инициализирован значением False — это было сделано для того, чтобы изначально предотвратить закрытие распахнутого окна при вызове метода Close. Флаг принимал значение True, когда иконка расположена в Tray, и окно при этом отсутствует. Но ведь теперь это происходит с самого начала — окна программы нет, и мы спокойно исправляем строку в секции объявлений переменных модуля slide.pas на mayClose: boolean=True. Проверьте работу программы и убедитесь, что теперь все в порядке.

ГЛАВА 5

Чертик из табакерки

Как установить и использовать горячую клавишу

— Как перезагрузить компьютер, пользуясь только мышкой?
— Нажать на "завершение работы".
— Настоящий программист так не поступает. Он поднимет мышку со стола и нажмет ею на <Reset>.

Как мы видели в *главе 3*, присваивание горячей клавиши пунктам всплывающего меню ничего не дает — вызов не работает. Для того чтобы проиллюстрировать, как же можно осуществить такую операцию, мы воспользуемся все тем же нашим проектом SlideShow — опять скопируем его последнюю версию из папки Glava4\2 в новую папку (Glava5\1), присвоим проекту промежуточный номер версии 1.20.1 (т. к. в данном случае это делается скорее для примера) и назначим ему горячую клавишу (пусть это будет <Ctrl>+<F12>), по которой, например, будет вызываться всплывающее меню.

Горячая клавиша с вызовом всплывающего меню

Сначала горячую клавишу надо зарегистрировать в системе. Это делается с помощью функции `RegisterHotKey`, параметрами которой являются знакомый нам дескриптор окна типа `hWnd`, идентификатор клавиши типа `integer` (клавиш может быть несколько), `fsModifiers` — модификатор, который определяет, какие клавиши должны быть нажаты совместно с указанной и, наконец, виртуальный код самой клавиши типа `word` или `integer`. Для модификатора допустимы значения: `MOD_ALT` (дополнительная клавиша <Alt>), `MOD_CONTROL` (<Ctrl>) и `MOD_SHIFT` (<Shift>). Функция возвращает `True`, если клавиша зарегистрирована успешно, но мы будем считать для простоты, что никто другой наше сочетание заведомо не использует. Если есть подозрение, что это не

так, следует сделать проверку на возвращаемое значение функции и предпринять какие-то действия, но в любом случае катастрофы не произойдет — просто не будет работать либо наша функция, либо ранее зарегистрированная. Зарегистрировать клавишу можно, например, по событию onCreate главной формы, добавив к уже имеющемуся тексту следующую строку:

```
RegisterHotKey(Handle,1,MOD_CONTROL,vk_F12);
{регистрируем клавишу Ctrl-F12}
```

Теперь нужно написать обработчик события по нажатию этой клавиши. Сначала регистрируем в интерфейсной части, в разделе **private**, саму процедуру обработчика (одна запись — **procedure** OnMinimizeProc — у нас там уже есть):

```
private
 { Private declarations }
 procedure WMHotKey(var Mess:TWMHotKey);message WM_HOTKEY;
```

С этой процедурой будет связано сообщение WM_HOTKEY. Наконец, собственно процедура обработчика:

```
procedure TForm1.WMHotKey(var Mess: TWMHotKey);
begin
{Нажата горячая клавиша Ctrl+F12}
if mess.HotKey=1 then {если это клавиша с идентификатором №1, то вызываем
всплывающее меню}
  Form1.PopupMenu1.Popup(Screen.Width-32,Screen.Height-32);
end;
```

При закрытии программы надо не забыть снять регистрацию горячей клавиши. Добавим в процедуру по событию onCloseQuery, где у нас производится окончательное закрытие окна, следующую запись:

```
UnregisterHotKey(Handle, 1);
```

Теперь управлять приложением с клавиатуры просто: нажмите комбинацию клавиш <Ctrl>+<F12>, затем нажмите клавишу управления курсором — стрелочку вверх, фокус перейдет на нижний пункт всплывающего меню **Закрыть**, и после нажатия клавиши <Enter> приложение закроется. Если стрелочку нажать два раза, то в фокусе будет пункт **Восстановить**. Таким образом мы получаем возможность быстрого управления приложением с клавиатуры без необходимости долго прицеливаться куда-то мышью. Разумеется, можно пойти и дальше — зарегистрировать несколько разных горячих клавиш, каждую для своего действия, но злоупотреблять такими вещами не следует: необходимость запоминать многочисленные нестандартные сочетания клавиш приведет к тому, что этими возможностями никто не будет пользоваться и все ваши усилия пропадут даром.

Есть и другой, более универсальный метод регистрации горячих клавиш с помощью так называемых "ловушек" (hook), и мы об этом поговорим в следующей главе, но пока достаточно использовать описанный механизм. Сейчас мы используем его для создания программы, которая вообще не имеет видимого окна.

Простая программа в виде иконки — отладочный пример

Попробуем отвлечься, наконец, от изрядно надоевшего SlideShow и создать программу, которая всегда находится в виде иконки: в Tray и лишь по некоторым событиям производит какие-то определенные действия. Главного окна-формы такая программа может и не иметь.

Об окнах и сообщениях

Любая графическая программа в Windows имеет по крайней мере одно окно. Если вы создавали стандартную программу Delphi, то это окно главной формы. Дескриптор окна имеет тип `hWnd`. Но окно может и не быть связано с конкретной формой, т. е. не отображаться на экране, и все-таки оно есть. Именно через окно происходит взаимодействие программы с другими программами. Для этого с любым (даже невидимым) окном связана так называемая оконная процедура — внутренний цикл обработки системных сообщений.

Каждая программа в Windows имеет свой главный дескриптор (application handle), указывающий на область памяти, в которой эта программа расположена. Этот дескриптор имеет такой же тип, как и у окна программы (`hWnd`), и в текстах программ часто называется `hInstance` (Instance — пример, экземпляр), чтобы подчеркнуть факт, что этот дескриптор свой для каждого конкретного экземпляра запущенной программы. Напомню *(см. главу 2)*, что дескриптор (handle) — вообще-то просто число, поэтому во многих случаях любые дескрипторы совместимы с типом `integer` или `longint`.

Приложения, у которых графическое окно отсутствует, но, тем не менее, они могут вести диалог с пользователем, называются *консольными* (от слова console — устройство ввода-вывода), т. к. взаимодействуют с клавиатурой и экраном напрямую, без посредства визуальных компонентов. Так как консольные приложения графического окна не имеют связанной с ним оконной процедуры, то сами по себе они исключены из потока системных сообщений Windows. При необходимости что-то вывести на экран для консольного приложения открывается текстовое окно, подобное окну DOS-программы. И само создание консольных приложений напоминает написание DOS- или

Windows-программ в невизуальной среде, например, для ввода и вывода используются знакомые по Turbo Pascal функции **read**, **write**, **readln**, **writeln**. Консольные программы проще по строению и не занимают столько ресурсов, сколько графические программы. Примеры консольных приложений вы могли встречать не раз — это всем известный файловый менеджер Far, многие стандартные системные утилиты, или, например, клиентские программы на рабочих местах операционисток на почтах, в банках или обменных пунктах валюты. Для того чтобы создать консольное приложение, можно обратиться к пункту **File | New | Other** и затем выбрать из многочисленных предлагаемых вариантов **Console Application**. В этой книге консольные приложения вам почти не встретятся, но запомните, что так, например, удобно создавать одноразовые утилитки для каких-то расчетов без необходимости обращаться к Turbo Pascal.

О сообщениях Windows. Обработчики событий, имеющиеся у каждого потомка `TWinControl`, обрабатывают примерно 20% типов сообщений, которые могут поступать в главную оконную процедуру, что соответствует примерно 80% процентам жизненных ситуаций. Для перехвата остальных сообщений нужно либо переопределить метод `WndProc` (мы это делали в *главе 3*), либо перекрыть его. В первом случае оригинальный обработчик вызывается автоматически. Перекрытие осуществляется таким, например, образом:

```
TForm1 = class(TForm)
procedure FormDblClick(Sender: TObject);
. . . . .
private
procedure WMLButtonDblClick(var Msg:TWMLButtonDblClk);
message WM_LBUTTONDBLCLK ;
end;
. . . . .
procedure TForml.WMLButtonDblClickfvar Msg:TWMLButtonDblClk);
begin
 ShowMessage(Format('Click at %d,%d',[Msg.XPos,Msg.YPos]));
end;
. . . . .
procedure TForm1.FormDblClick(Sender: TObj ect) ;
begin
 ShowMessage('Right Dbl Click') ;
end;
```

Здесь мы определяем свой обработчик для события двойного щелчка левой кнопкой. Само имя процедуры обычно делают совпадающим с наименованием перехватываемого сообщения (`WMLButtonDblClick`), однако это совершенно необязательно, имя может быть любым удобным, а вот идентифика-

тор самого сообщения должен совпадать с определенной в Windows или ранее объявленной (для собственного сообщения, см. далее) константой. Параллельно я тут показал обычный обработчик двойного щелчка на форме (`FormDblClick`), который создается средствами Delphi. Приведенный код будет делать следующее: при двойном щелчке левой кнопкой мыши событие будет поступать в наш обработчик, а правой — в стандартный. Чтобы вызвать стандартный обработчик после нашего (или перед ним), нужно добавить в созданную процедуру слово `inherited` (соответственно, в конце или в начале ее).

Можно также определить в системе и свое собственное сообщение, для чего нужно объявить сначала свою собственную константу, обычно это делается через базовую константу `wm_User`, например:

```
const wm_MyMessage= wm_User+1;
```

Послать такое сообщение можно через функцию `SendMessage`, например, вот так:

```
SendMessage(SomeForm.Handle, wm_MyMessage,1,0);
```

"Поймать" такое сообщение в форме-адресате можно точно такой же функцией перехвата, как была сделана ранее для двойного щелчка, только объявив, разумеется, точно такую же константу. Бояться, что номер, объявленный через `wm_User`, случайно совпадет с объявленным в другой программе, не нужно, если сообщение действует в пределах одного приложения (константа тогда действует подобно локальным переменным). В *главе 7* мы покажем, как в иных случаях образовывать уникальный номер сообщения с помощью генератора случайных чисел.

Кроме этого, сообщения можно перехватывать через обработчик `Application.onMessage` [3]. Мы будем пользоваться всеми этими способами в дальнейшем. У каждого сообщения есть параметры `wParam` и `lPapam`, которые имеют разные значения в каждом отдельном случае. Подробное описание структуры `MSG` с этими параметрами, функции `SendMessage` и других функций, связанных с сообщениями, см. в [16,18].

Здесь мы создадим новый проект (**File | New | Application**), а затем уберем из него заготовку формы `Unit1` (и, соответственно, одноименного модуля) через меню **View | Project Manager** (выделить в пункте **Project1** подпункт **Unit1** и нажать **Remove**). После этого у вас все, относящееся к проекту, исчезнет вообще, но не пугайтесь: надо зайти опять в меню **View**, найти там пункт **Units** и выделить единственный оставшийся пункт **Project1**. Тогда у вас появится заготовка самой простой Delphi-программы (файл Project1.dpr). Сохраним проект под этим же именем (на диске он находится в папке Glava5\2).

На этом примере я хочу вам продемонстрировать процесс создания программы "по образцу". Это очень удобный и часто встречающийся прием — зачем изобретать велосипед и отлаживать то, что уже отлажено? Если бы вы создавали крупное коммерческое приложение, предназначенное для продаж по всему миру, вас могли бы осудить за нарушение права собственности, но и то только лишь в том случае, если бы вы использовали проприетарные участки кода. Нормальные программисты часто сами публикуют исходный код своих программ для повторения и модернизации — или просто так, или, как в случае "свободного софта" (распространяемого по лицензии GPL), с условием доступности исходного кода и упоминания авторов. Независимо от лицензии, упоминать источник есть просто дань вежливости, и мы именно так и поступим. Мы возьмем подходящую готовую и заведомо работающую программу из главы 1 книги [5], добавим нужную и уберем лишнюю функциональность.

А как зафиксировать имя автора исходного образца так, чтобы упоминание о нем сохранилось в программе? Обычный способ — включить его в меню **About** — нам здесь не подходит, т. к. программа простейшая и загромождать ее всякими меню нет никакого смысла. Можно использовать окно-заставку, но и это уже будет не простейшая программа — от использования одного только BMP-файла размер программы увеличится раз в пятнадцать. Однако Delphi предоставляет способ включения в исполняемый файл произвольной текстовой информации (до 255 символов). Если вы просмотрите любой "фирменный" EXE-файл в текстовом режиме (такую возможность предоставляют, например, просмотрщики, встроенные в некоторые из клонов Norton Commander), вы обязательно найдете подобную информацию о фирме-производителе, пакете программирования или фамилиях разработчиков. Этим всегда пользуются настоящие (без кавычек) программисты — был даже забавный скандал, когда Microsoft уличили в поддержке пиратства, обнаружив в некоторых файлах из комплекта Windows незамеченный ее программистами текст, свидетельствующий об использовании ими "крякнутого" (crack) софта. Альтернативный штатный способ включения информации в исполняемый файл предоставляют поля структуры `VersionInfo`, хранящиеся в ресурсах исполняемого файла. При наличии формы для заполнения этих полей мы пользуемся соответствующим пунктом в меню **Project**, а способ ввести эту структуру вручную мы будем рассматривать в *главе 11*.

Итак, для того чтобы включить в нашу программу текст с именем автора, найдем в меню **Project | Options** закладку **Linker**, а там пункт **EXE Description**, и впишем (естественно, по-английски) в поле этого пункта что-нибудь вроде: `"Programmer Revich Y. Based on template by Flenov M.V."` (Программист Ревич Ю. Основано на шаблоне Фленова М. В.). Не забудьте после компиляции заглянуть в EXE-файл — в конце его вы найдете этот текст в кодировке Unicode, т. е. каждый символ текста будет предваряться

байтом со значением 0, означающим английский язык (о кодировках мы еще поговорим в *главе 8*).

Сейчас мы на основе этой заготовки создадим прототип программы — приложение с пустым окном (представляющим собой просто серый прямоугольник), воспользовавшись образцом, но сразу модернизировав его таким образом, чтобы сначала создавалась иконка в Tray, а уже через нее происходили всякие нужные нам действия. Для того чтобы проще было отлаживать программу, мы будем отступать от образца постепенно: в первом варианте пусть по щелчку на созданной иконке показывается это самое пустое окно, уничтожающееся по нажатию клавиши <Esc> (последнее так, как в образце — процедура обработки соответствующего сообщения там уже отработана).

После этого заменим весь текст в заготовке программы Project1.dpr после заголовка (не забыв сохранить указание компилятору {$R *.RES}) на следующий (я сразу модернизировал текст образца, сократив ненужные структуры и вставив процедуру создания иконки) — листинг 5.1.

Листинг 5.1. Заготовка программы Project1.dpr

```
program Project1;

uses
windows, Messages, ShellApi, SysUtils;

{$R *.RES}

var
Instance: HWnd; {дескриптор модуля приложения}
WindowClass: TWndClass; {класс окна приложения}
FHandle: HWnd; {дескриптор окна приложения}
mesg: TMsg; {сообщения приложения}
 Ico_Message: integer=wm_User; {сообщение от иконки окну}
 noIconData: TNotifyIconData; {дескриптор структуры
                NotifyIconData}
 HIcon1: hIcon; {дескриптор иконки}

function WindowProc (Hwn,msg,wpr,lpr: longint): longint; stdcall;
{обработка сообщений}
begin
result:=defwindowproc(hwn,msg,wpr,lpr);
              {ищет стандартный обработчик}
if Msg = wm_Destroy then {если команда на закрытие}
begin
   Shell_NotifyIcon(NIM_Delete,@noIconData); {удаляем иконку}
```

```
   halt; {закрываем программу}
end;

if msg=wm_KeyDown then
if wpr=VK_ESCAPE then {при нажатии Esc удаляем программу}
begin
  Shell_NotifyIcon(NIM_Delete,@noIconData); {удаляем иконку}
  halt;
end;
if Msg = Ico_Message then {если это сообщение от иконки}
begin
  if lpr=WM_LBUTTONUP then {была отпущена левая кнопка}
  begin
    ShowWindow(FHandle, SW_SHOW); {демонстрируем окно}
    UpdateWindow (FHandle);
  end;
end;
end;

procedure CreateMyicon;
begin
HIcon1:=LoadIcon(Instance,'MAINICON'); {получаем дескриптор
                    иконки}
with noIconData do begin
cbSize:=Sizeof(TNotifyIconData); {размер структуры}
Wnd:=FHandle; {дескриптор окна}
uID:=0; {единственная иконка}
UFlags:=NIF_MESSAGE or NIF_ICON or NIF_TIP; {взводим все флаги}
SzTip:='Project1'; {всплывающая подсказка}
HIcon:=HIcon1; {дескриптор иконки}
uCallBackMessage:=Ico_Message;{Определяемое пользователем
                сообщение}
end;
Shell_NotifyIcon(NIM_ADD,@noIconData); {создали иконку}
end;

begin
instance :=GetModuleHandle(nil); {получаем дескриптор модуля}
with WindowClass do {задаем свойства окна}
begin
  style:=CS_HRedraw or CS_VRedraw;
  Lpfnwndproc:=@windowproc;
  Hinstance:=Instance;
  HbrBackground:= color_btnface;
```

```
  LpszClassName:='DX';
  Hcursor:=LoadCursor(0,IDC_ARROW);
  hIcon:=LoadIcon(Instance,'MAINICON');
end;

RegisterClass(WindowClass); {регистрируем класс}

FHandle:=CreateWindowEx (0,'DX','',WS_POPUP, 5,5, 200, 200, 0,0,instance,
nil); {получаем дескриптор окна}

CreateMyicon; {создаем иконку}

while (GetMessage(mesg, 0, 0, 0)) do {цикл обработки сообщений}
begin
 TranslateMessage(mesg);
 DispatchMessage (mesg);
end;
end.
```

Здесь процедура обработки сообщения `wm_Destroy` нужна для того, чтобы при выключении компьютера программа "самоуничтожалась" — иначе выход из Windows при запущенной программе может неоправданно затянуться с получением различных грозных сообщений.

Запустим приложение из среды Delphi и убедимся, что все работает — при запуске создается иконка, при щелчке на ней левой кнопкой в верхнем левом углу экрана возникает пустое окно-прямоугольник 200×200, при нажатии клавиши <Esc> программа закрывается, и иконка при этом уничтожается. Если мы исследуем свойства полученной программы немного подробнее, то увидим, что при запуске вне среды Delphi программа не будет реагировать на нажатие <Esc>, пока мы не сфокусируемся на ее пустом окне мышью. Это и понятно — пока окно не сфокусировано, оно не знает, что нажатие клавиши <Esc> предназначено именно ему. Можно исправить это, заставив окно при вызове сразу располагаться поверх всех окон (вызовом функции `BringWindowToTop`), но это будет полумера для данного конкретного случая: нам окно вообще-то само по себе не требуется, наша задача сделать так, чтобы программа работала по нажатию клавиши вообще без всякого (видимого) окна. Поэтому мы воспользуемся регистрацией горячей клавиши, как делали раньше. Вставим вызов регистрационной функции в нашу программу перед созданием иконки (клавишу возьмем ту же самую <Ctrl>+<F12>):

```
. . . . .
RegisterHotKey(FHandle, 1 ,MOD_CONTROL, vk_F12); {регистрируем
                        клавишу Ctrl+F12}
CreateMyicon; {создаем иконку}
. . . . .
```

А в процедуру обработки сообщений добавим следующий текст:

```
. . . . .
if Msg = WM_HOTKEY then {если нажата горячая клавиша}
if wpr=1 then {и ее номер 1}
begin
 Shell_NotifyIcon(NIM_Delete,@noIconData); {удаляем иконку}
 UnregisterHotKey(FHandle, 1); {убираем горячую клавишу}
 halt; {закрываем программу}
end;
. . . . .
```

Запустим программу и убедимся, что она работает как надо: при запуске иконка создается, а при нажатии комбинации клавиш <Ctrl>+<F12> программа закрывается. Теперь отладка прототипа закончена. Кстати, обратите внимание на размер исполняемого файла — чуть больше 40 Кбайт. Вот что значит не использовать графические ресурсы!

Резидентная программа для исправления текста в неправильной раскладке

Попробуем на основе нашего прототипа создать программу, которая по нажатию комбинации клавиш <Ctrl>+<F12> исправляет текст, набранный в неправильной раскладке клавиатуры. С целью как можно больше рассказать о методах обращения с функциями API, я покажу процесс поэтапного создания разных вариантов все более работоспособной программы.

Заготовка

Перенесем проект из папки Glava5\2 в другую папку (на диске — Glava5\3), на этот раз через пункт **File | Save Project as** под новым именем: Layout ("раскладка") . Так как мы не используем модулей, кроме стандартных, то больше ничего в проекте изменять не придется (в том числе и редактировать DSK-файл). Затем удалим из обработчика сообщений ненужные действия (обработка нажатия клавиши <Esc>), а обработку нажатия кнопки мыши перепишем следующим образом:

```
if Msg = Ico_Message then {если это сообщение от иконки}
begin
 if lpr=WM_RBUTTONUP then {была отпущена правая кнопка}
 begin
  if MessageBox(FHandle,'Вы хотите закрыть программу
     Layout?','Warning',MB_YESNO)=idYES then
```

```
  begin
    Shell_NotifyIcon(NIM_Delete,@noIconData); {удаляем иконку}
    UnregisterHotKey(FHandle, 1); {убираем горячую клавишу}
    halt; {закрываем программу}
  end;
 end;
end;
```

Левую кнопку мыши мы освободили — она нам еще понадобится, а при щелчке правой кнопкой на иконке у нас теперь будет возникать соответствующее окно с выбором: закрывать программу или нет. В принципе можно было бы и сократить текст, убрав ненужные пункты при заполнении структуры `WindowClass` (все равно окно показываться не будет), но смысла заниматься этим, рискуя что-то нарушить в программе, нет — структура так или иначе будет занимать место в памяти. Не забудем только заменить текст всплывающей подсказки `Project1` на что-то более осмысленное, например `Language Layout`.

Теперь изменим иконку приложения — раньше мы делали это через меню в свойствах главной формы, но теперь, поскольку ее нет, запустим Image Editor и вручную отредактируем файл ресурса Layout.res, относящийся к нашему проекту. Всю эту операцию надо проделывать при закрытой Delphi, т. е. запускать Image Editor надо не из среды через меню **Tools | Image Editor**, а через меню **Пуск | Программы**, папка Borland Delphi 7. Открыв Layout.res в редакторе (пункт **File | Open**), найдем там иконку (она только одна там и будет), у которой должно быть имя **MAINICON**, и откроем ее в отдельном окне (для это надо дважды щелкнуть по ее названию). Заранее подберем какую-нибудь подходящую иконку из коллекции, символизирующую, например, редактирование текста (на диске я ее расположил в папке с текущим проектом под названием edit1.ico). Откроем файл с этой иконкой в другом окне Image Editor (пункт **File | Open** с установкой фильтра для файлов ICO — см. рис. 5.1) и заменим одну иконку на другую простым методом Copy-Paste. После этого измененный файл ресурсов надо сохранить (**File | Save**), закрыть окна с иконками, и закрыть окно Image Editor.

Итак, после того как мы отладили весь антураж, нам остается оформить саму процедуру исправления раскладки, заменив в ней закрытие программы в обработчике нажатия горячей клавиши. Действия должны быть простыми: выделяем неправильно набранный (например, в Word) текст, нажимаем комбинацию клавиш <Ctrl>+<F12>, после чего текст автоматически заменяется на набранный в противоположной раскладке. Никакой автоматики (определения, действительно ли это бессмысленный текст) вводить здесь не требуется — за исключением того, что раскладка исходного фрагмента должна опре-

деляться автоматически. Что, собственно, следует делать программе? Во-первых, забрать выделенный текст в буфер обмена, вывести его, например, в строку, определить раскладку, исправить текст в строке, заменить им то, что в буфере обмена, и снова вставить на место.

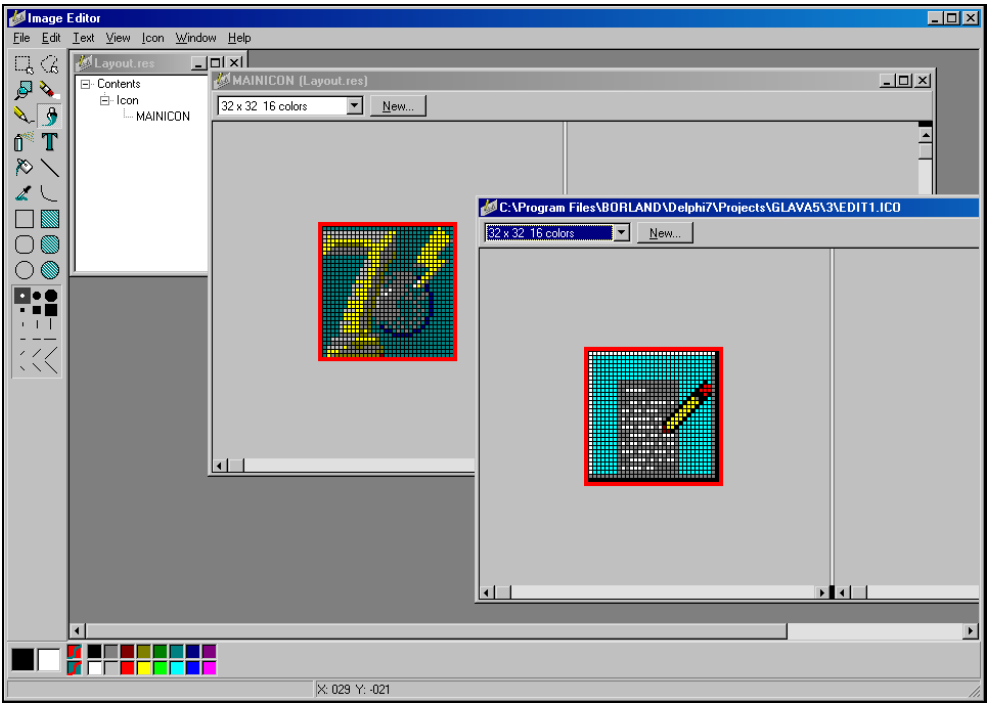


Рис. 5.1. Редактор ресурсов

Попытка первая — в лоб

Начнем с первой операции и сначала попробуем пойти самым простым путем. Специально для работы с текстом в буфере есть команды `WM_CUT`, `WM_COPY` и `WM_PASTE`, которыми можно воспользоваться, например, через функцию `SendMessage` (для других, не текстовых, форматов в буфере обмена все гораздо сложнее). Но сначала нужно определить, куда именно эти команды посылать — а именно, в то окно, которое в данный момент активно. Для этого есть функция `GetForegroundWindow`, которая возвращает дескриптор активного окна. Однако у этого окна могут быть дочерние окна, потому воспользуемся дополнительно функцией `GetTopWindow`, которая возвращает дескриптор дочернего окна (условно назовем его `Edit`) по известному родительскому. Предварительно в секции описаний надо объявить переменные:

```
var
 WindowHandle, EditHandle: HWnd;
. . . . .
```

Процедура теперь будет выглядеть так:

```
if Msg = WM_HOTKEY then {если нажата горячая клавиша}
if wpr=1 then {и ее номер 1}
begin
 WindowHandle:=GetForegroundWindow; {активное окно}
 EditHandle:=GetTopWindow(WindowHandle); {активный элемент}
 SendMessage(EditHandle, WM_COPY, 0, 0);
        {Посылаем сообщение для копирования}
 {пока больше ничего не делаем}
end;
```

Запустите программу и попробуйте — все это будет работать только с Блокнотом, в остальных случаях — полный ноль. Возможно, это связано с тем, что команды эти предназначены для работы в рамках одного приложения, вероятно — с множественным форматом буфера обмена — точно я сказать не могу. Хотя, как видим, можно использовать такой способ для обмена только чисто текстовыми сообщениями.

Вариант второй — посложнее

Остается "нажать мышью на <Reset>" *(см. эпиграф)*, а именно: эмулировать нажатие стандартных клавиш, в данном случае <Ctrl>+<C>. Для этого заменим строку, содержащую вызов SendMessage, на следующий текст:

```
. . . . .
 PostMessage(EditHandle,WM_KEYDOWN,VK_CONTROL,$001D0001);
 PostMessage(EditHandle,WM_KEYDOWN,ord('C'),$002E0001);
 SendMessage(EditHandle,WM_COMMAND, $00010043, $00000000);
. . . . .
```

Функция PostMessage вместо SendMessage используется тогда, когда нам безразлично, что именно произойдет с сообщением — она посылает сообщение без ожидания его обработки, во многих практических случаях они взаимозаменяемы, но не во всех. Дополнительная строка SendMessage с соответствующими параметрами — это так называемый accelerator (ускоритель), иначе PostMessage может не сработать как надо. Параметр wparam, сопровождающий команду WM_KEYDOWN, содержит виртуальный код клавиши (в данном случае, например, vk_Control). Цифровой параметр (lParam) содержит в двух младших байтах число повторений нажатия клавиши, которое в данном случае равно 1, а в третьем байте (шестнадцатеричные разряды 4 и 5) — скан-код

клавиши, т. е. то, что выдает клавиатура *(о скан-кодах и виртуальных кодах подробнее см. главу 6)*. Надо заметить, что, судя по всему, стандартный обработчик сообщения WM_KEYDOWN скан-код все равно игнорирует, но порядка ради будем делать все, как положено.

Заметки на полях

При необходимости послать сообщение WM_KEYUP, lparam должен иметь точно такое же значение, но самые старшие два бита должны при этом быть равны 1, т. е., например, для клавиши <C> он будет тогда выглядеть, как $C02E0001. И если внимательно вчитаться в то, что написано по поводу этих команд в Win32.hlp, а также на сайте MSDN [14][1], то становится ясно, что сама команда WM_KEY*xx* вместе с параметром wparam (виртуальный код клавиши, в данном случае ord('C')), и приведенный ранее lparam попросту дублируют друг друга. Хотя старшие два бита в lparam определяют команду (нажата клавиша или отпущена, и ее предыдущее состояние), но скан-код, как и wparam, полностью определяет саму клавишу. Отсюда становится понятно, почему обработчик игнорирует скан-код — но никто не дает гарантии, что какой-то другой обработчик тоже это будет делать!

Запустим программу несколько раз, меняя эмулируемую клавишу с <Ctrl>+<C> на <Ctrl>+<V> (скан-код $2F) и обратно, и убедимся, что копирование и вставка выделенного текста через буфер обмера работает с разными приложениями, включая даже адресную строку браузера — но не со всеми! Например, приведенная процедура не работает с "самым главным редактором", MS Word.

Вариант третий — ура!

Чтобы решить эту проблему, попробуем эмулировать нажатие клавиш по-иному — с помощью функции Keybd_event. Выгодное ее отличие от PostMessage заключается в том, что ее использование много проще — не нужно определять никаких дескрипторов, событие посылается "в никуда", т. е. оно будет автоматически перехвачено тем окном, которое находится в данный момент в фокусе. Однако функцию-ускоритель мы все же используем (иначе программа может сбоить), так что определение дескрипторов придется оставить. Кроме того, хотя в числе параметров присутствует скан-код клавиши (второй параметр), в описании функции (см. [14]) разработчики на этот раз честно признались, что он не используется (дальше мы увидим, что они слукавили — это не всегда так). Озаботимся также, чтобы эмулировалось и нажатие клавиш, и их отпускание.

[1] Я неоднократно буду ссылаться на разные страницы этого сайта в дальнейшем, но точных ссылок приводить не буду ввиду их громоздкости — нужная страница легко находится встроенным в MSDN поиском по указанным в тексте названиям функций или заголовкам, как ключевым словам.

Итак, перенесем проект Layout из папки Glava5\3 в новую папку (на диске Glava5\4) и заменим наши процедуры в обработчике на следующие:

```
. . . . .
WindowHandle := GetForegroundWindow; {активное окно}
EditHandle := GetTopWindow(WindowHandle); {активный элемент}
keybd_event(VK_CONTROL,0,0,0); { нажатие клавиши Ctrl}
keybd_event(ord('C'),0,0,0); {нажатие клавиши C}
keybd_event(ord('C'),0,KEYEVENTF_KEYUP,0); {отпускание C}
keybd_event(VK_CONTROL,0,KEYEVENTF_KEYUP,0); {отпускание Ctrl }
SendMessage(EditHandle,WM_COMMAND, $00010043, $00000000);
            {ускоритель}
. . . . .
```

Соответственно, чтобы эмулировать вставку из буфера, надо только заменить `'C'` на `'V'` в соответствующем месте. Убедимся, что все теперь работает, и пойдем дальше.

Чтобы вывести полученное в строку для дальнейших манипуляций, мы объявим в предложении **uses** модуль `Clipbord`, и тогда получим право вписать после приведенной ранее эмуляции нажатия такую строку:

```
stClb:=Clipboard.AsText; {забрали из буфера в строку}
```

Разумеется, переменную `stClb` типа **string** нам придется заранее объявить. Заодно вместе с ней допишем в секции объявлений две целые переменные-счетчика, которые нам понадобятся чуть позже:

```
var
. . . . .
stClb:string;
i,n:integer; {счетчики}
```

Прежде чем двигаться дальше, надо понять, в чем, собственно, состоит процедура перекодировки. Посмотрим на клавиатуру, которая перед вами. Так как никакой системы, связывающей символы в русской и в английской раскладке, нет, то придется соответствие между ними устанавливать табличным способом. Проще всего это сделать так: взять номера символов ASCII подряд и привязать к ним соответствующие по клавиатуре русские символы.

Таблица ASCII, а также коды русских клавиш в различных кодировках приведены в *приложении 3* (нам нужна, конечно, Win1251). Для того чтобы охватить все различающиеся символы (т. е. те, что присутствуют на клавиатуре), достаточно взять фрагмент таблицы ASCII с 34 по 126 символ, но для удобства мы возьмем также и совпадающие в обоих раскладках пробел (символ 32 = $20) и восклицательный знак (33 = $21). Глядя поочередно в таблицу

ASCII, на клавиатуру и затем в таблицу русских кодов, составим таблицу соответствия и занесем ее в виде константы в секции объявлений программы:

```
. . . . .
const CharEngRus: array [32..126] of byte =
($20,$21,$DD,$B9,$3B,$25,$3F,$FD,$28,$29,$2A,{*}
$2B,$E1,$2D,$FE,$2E,$30,$31,$32,$33,$34,$35, {5}
$36,$37,$38,$39,$C6,$E6,$C1,$3D,$DE,$2C,$22,{@}
$D4,$C8,$D1,$C2,$D3,$C0,$CF,$D0,$D8,$CE,$CB,{K}
$C4,$DC,$D2,$D9,$C7,$C9,$CA,$DB,$C5,$C3,$CC, {V}
$D6,$D7,$CD,$DF,$F5,$5C,$FA,$3A,$5F,$B8,$F4,{a}
$E8,$F1,$E2,$F3,$E0,$EF,$F0,$F8,$EE,$EB,$E4,{l}
$FC,$F2,$F9,$E7,$E9,$EA,$FB,$E5,$E3,$EC,$F6,{w}
$F7,$ED,$FF,$D5,$2F,$DA,$A8 );
var
. . . . .
```

В конце каждой строки закомментирован знак ASCII, на котором эта строка заканчивается, чтобы было проще контролировать процесс создания. Теперь если мы имеем в английской раскладке символ, скажем, номер 60 (в десятичном представлении, т. е. "<"), то по таблице мы определим, что ему соответствует русский символ с номером $C1, т. е. заглавное "Б".

Заметки на полях

Таблица ASCII устанавливает только первые 128 знаков, причем первые 32 позиции в этой таблице (от 0 до 31) представляют собой не собственно символы, а управляющие коды (соответствующие в интерпретации для клавиатуры переводу строки, команде <Esc>, <Delete> и т. п.). Поэтому в таблице ASCII, приведенной в *приложении 3*, вы видите только символы с номерами $20—$7E (32—126), плюс важный для практики код клавиши <Enter> ($0D или 13), минус практически неиспользуемый код с номером 127. Обратите внимание, что виртуальные коды клавиш *(см. приложение 2)* не всегда совпадают с номерами символов в таблице ASCII. Символы же с номерами 128 и выше зависят от национальной кодовой страницы и от, собственно, кодировки, поэтому в том же *приложении 3* приведены коды русских букв в различных кодировках. В интересующей нас здесь кодировке Win1251 русские буквы расположены по порядку в самом конце таблицы (кроме "Ё" и "е"), что облегчает задачу определения раскладки. В последней колонке приведены коды русских букв для двухбайтовой кодировки Unicode, к которой мы еще вернемся позже. Напомню, что любой символ, даже тот, для которого не предусмотрено отдельной клавиши, можно ввести с клавиатуры, если "задавить" клавишу <Alt>, а затем набрать номер символа на цифровой клавиатуре при включенной клавише <Num Lock>.

Для того чтобы не сочинять отдельную таблицу для обратного перевода (что гораздо сложнее, т. к. русские буквы и специальные символы расположены в беспорядке), мы воспользуемся следующим приемом: будем просматривать таблицу сначала, пока не наткнемся на номер, соответствующий символу в

русской раскладке, и определив, какой ячейке таблицы это соответствует, тем самым узнаем нужный в английской раскладке. Не очень экономичный алгоритм, но при современных скоростях это микросекунды, к тому же Windows тратит гораздо больше времени на куда более пустячные задачи.

Ну, а для того чтобы определить, в какой раскладке у нас набран текст, извлеченный из буфера обмена, мы поступим просто: просмотрим строку символ за символом до тех пор, пока не наткнемся на заведомую букву, а по ней и решим, русская раскладка была или английская. Все сказанное ранее реализовано в следующей процедуре, которой мы заменим обработчик нажатия горячей клавиши окончательно:

```
. . . . .
if Msg = WM_HOTKEY then {если нажата горячая клавиша}
if wpr=1 then {и ее номер 1}
begin
 WindowHandle := GetForegroundWindow; {активное окно}
 EditHandle := GetTopWindow(WindowHandle); {активный элемент}
 keybd_event(VK_CONTROL,0,0,0); { нажатие клавиши Ctrl}
 keybd_event(ord('C'),0,0,0); {нажатие клавиши C}
 keybd_event(ord('C'),0,KEYEVENTF_KEYUP,0); {отпускание C}
 keybd_event(VK_CONTROL,0,KEYEVENTF_KEYUP,0); {отпускание Ctrl}
 SendMessage(EditHandle,WM_COMMAND, $00010043, $00000000);
                        {ускоритель}
   stClb:=Clipboard.AsText; {забрали в строку}
   i:=1; {ищем первый буквенный символ}
   while (not (stClb[i] in ['A'..'Z'])) and (not (stClb[i] in
   ['a'..'z'])) and (not (stClb[i] in ['А'..'я'])) do
   begin i:=i+1; if i>length(stClb) then exit end;
    {если ни одного буквенного символа, то на выход}
   {определяем раскладку по найденному буквенному символу:}
   if (stClb[i] in ['A'..'Z']) or (stClb[i] in ['a'..'z'])
   then {то это английский}
   begin
    for n:=1 to length(stClb) do
    if (ord(stClb[n])>=32) and (ord(stClb[n])<=127) then
    stClb[n]:=chr(CharEngRus[ord(stClb[n])]);
       {заменяем символом из таблицы}
   end
   else {это русский}
   for n:=1 to length(stClb) do
   for i:=32 to 126 do
   begin  {обратный поиск по таблице}
     if ord(stClb[n])=CharEngRus[i] then
```

```
      begin stClb[n]:=chr(i); break; end;
    end;
    Clipboard.Open;
    Clipboard.AsText:=stClb; {забрали обратно в буфер}
    Clipboard.Close;
  {вставляем поверх выделенного:}
  keybd_event(VK_CONTROL,0,0,0); { нажатие клавиши Ctrl}
   keybd_event(ord('V'),0,0,0); {нажатие клавиши V}
   keybd_event(ord('V'),0,KEYEVENTF_KEYUP,0); {отпускание V}
   keybd_event(VK_CONTROL,0,KEYEVENTF_KEYUP,0); {отпускание Ctrl }
  SendMessage(EditHandle,WM_COMMAND, $00010043, $00000000);
                        {ускоритель}
end {HotKey};
. . . . .
```

После того как вы воспроизведете все, что здесь написано, или просто попробуете запустить этот вариант программы с диска, вы увидите, что он безупречно работает в среде Windows 98 и MS Word 97, однако в Windows XP возникают проблемы — во-первых, при работе с множественным буфером обмена в Word 2000 и выше. Во-вторых, что существеннее, в XP в любом редакторе вместо русского появляются "кракозябры", т. е. компонент, куда происходит вставка из буфера, "не понимает", что перед ним русский. Происходит это потому, что Win9*x* основана на DOS и однобайтовой кодировке, и переключение второй половины таблицы однобайтовых символов на русскую происходит еще при загрузке системы (и, конечно, только если установлена русская версия, иначе могут быть те же самые проблемы), а XP — на Unicode, и по умолчанию использует однобайтовую кодовую страницу с поддержкой европейских языков. Причем мы с нашей самодельной программой вправе не считать себя какими-то неполноценными: в точности те же проблемы возникают в XP при обмене текстами между "официальными" редакторами от MS (Word, WordPad, Notepad) и каким-нибудь редактором, который заведомо Unicode не поддерживает *(подробнее об этом см. главу 8)*. Ко всем этим делам мы еще вернемся позже, когда будем обсуждать Unicode, а сейчас сделаем попытку решить проблему в лоб: попробуем дать понять программе, что перед ней русский. Для этого надо переключить раскладку. Как мы увидим в дальнейшем, переключение кодовой страницы (языка) и переключение раскладки в Windows связаны, поэтому мы, забегая вперед, и попытаемся переключить раскладку перед тем, как интерпретировать содержимое буфера.

Мы еще поговорим подробно в следующих главах о переключателе раскладки, как отдельной программе, а здесь просто попробуем применить эту функцию — если программа определила, что текст был английский и перекодирует его в русский, то она автоматически будет переключать и раскладку на

русский и наоборот. Сделать это очень просто с помощью функции LoadKeyboardLayout, которая загружает нужную раскладку по ее названию. Название имеет вид строки, соответствующей 32-битному шестнадцатеричному значению, младшие два байта которого есть идентификатор языка в системе Windows, а старшие могут указывать на тип клавиатуры (раскладка Дворака и т. п.), для обычной клавиатуры они равны 0. Список всех идентификаторов языка можно посмотреть, например, в справке по Win32, на сайте [14] или даже просто в реестре, но практически нас интересуют, естественно, только два: русский (0419h) и английский (0409h). Кроме этого, вторым параметром нужно указать флаг, который означает в данном случае просто замену текущей раскладки на свою, что соответствует константе KLF_ACTIVATE[2]. С учетом всего сказанного фрагмент процедуры-обработчика горячей клавиши будет у нас выглядеть следующим образом:

```
. . . . .
if (stClb[i] in ['A'..'Z']) or (stClb[i] in ['a'..'z'])
then {то это английский}
begin
 LoadKeyboardLayout('00000419', KLF_ACTIVATE);
       {переключаем раскладку на русский}
 for n:=1 to length(stClb) do
 if (ord(stClb[n])>=32) and (ord(stClb[n])<=127) then
 stClb[n]:=chr(CharEngRus[ord(stClb[n])]);
       {заменяем символом из таблицы}
end
else {это русский}
begin
  LoadKeyboardLayout('00000409', KLF_ACTIVATE);
       {переключаем раскладку на английский}
  for n:=1 to length(stClb) do
. . . . .
```

Попробуйте, и вы убедитесь, что в Windows 98 все работает нормально — при перекодировке заодно переключается и раскладка. А вот в Windows XP опять трудности — во-первых, сама по себе раскладка (если судить по индикатору) не переключается[3]. Однако текст с английского на русский перекоди-

[2] Если вы хотите узнать подробности о функциях переключения языка и раскладки, то смотрите на сайте [14] — в отношении всех языковых функций справка по Win32 из Delphi несет устаревшую информацию.

[3] Как мы увидим в следующих главах, нам этого и не обещали. Скорее надо удивляться, что в Windows 98 все работает. Многие подробности о путанице в Windows с понятием "язык" будут раскрыты в *главе 8*.

руется нормально, а если установить вручную (клавишами, предназначенными для этого в системе) основную раскладку русскую, то все начинает нормально работать — текст перекодируется туда и обратно (хотя в Word XP все равно трудности с буфером, но это уже другие проблемы). Разумеется, такой результат нас не устраивает — будем считать попытку неудачной. Оставим пока программу в том виде, в котором она есть, и успокоимся на том, что сделали нормальный вариант для платформы Win9*x*. Приступим сначала к программе-переключателю раскладки, а позднее попробуем довести до ума и перекодировщик.

ГЛАВА 6

Давим на клаву

Некоторые особенности работы с клавиатурой. Клавиатурный шпион и использование hook

> Для удара по клавише поставьте руку на стол в таком же положении, как она ставилась на основную позицию пишущей машины. Удар будет производиться указательным пальцем. Для этого: слегка опереться на этот палец, остальные приподнять, затем слегка взмахнуть рукой, произвести удар и поставить все пальцы на место.
>
> *К. К. Соловьева, "Курс современной машинописи"*

Мышь и клавиатуру обычно рассматривают совместно, но о мыши говорить намного проще, чем о клавиатуре — прежде всего потому, что у нее всего-то две (ну ладно, три) основные кнопки, а если их больше — все равно этим управляет исключительно драйвер конкретного производителя и ваша программа "изнутри" их использовать не может. Поэтому ввиду тривиальности обращения с мышью (к любому визуальному компоненту в Delphi привязаны события `onClick`, `onMouseDown` и подобные) отдельно на этом мы останавливаться не будем, а что касается особых событий, связанных с перемещением курсора (типа технологии Drag&Drop, приемов рисования и т. п.), а также использования колесика прокрутки, то об этом разговор будет отдельный и серьезный. Здесь же мы остановимся только на некоторых подробностях и приемах использования клавиатуры, которых вы в учебниках по Delphi, вероятнее всего, не встретите.

В ходе дальнейшего изложения нам понадобится много работать с числами в двоичном и шестнадцатеричном представлении. Для того чтобы читателю, не получившему специального образования, было легче ориентироваться в этих вопросах, в *приложении 1* приведены сведения о системах счисления и обращении с числами в различной форме. Там же описан модуль Ariphm, который позволяет осуществлять вывод чисел в шестнадцатеричном представлении в виде строки.

Как все это устроено

Чтобы эффективно использовать клавиатуру, хорошо бы понимать, как ведется обработка нажатия клавиш в системе. С клавиатурой связано как минимум три разновидности кодов: символьные коды, виртуальные коды и скан-коды, и не всем понятна разница между ними. Попробуем внести ясность в этот запутанный вопрос.

Самая большая путаница со скан-кодами — для начала, их также существует две разновидности. Есть "настоящие" скан-коды — это то, что получает система прямо из клавиатуры. Прочитать их можно, если напрямую обратиться в порт 60h, но мы здесь не будем копать так глубоко, а посмотрим, что с ними происходит дальше. А дальше они попадают в цепкие лапы BIOS, которая через прерывание INT9h их интерпретирует: например, вызывает прерывания при нажатии клавиш <Ctrl>+<Alt>+<Del>, <Ctrl>+<Break> и т. п., а также справляется с текущей таблицей кодов символов (когда надо, с учетом нажатых клавиш-модификаторов <Alt>, <Ctrl>, <Shift> или <Caps Lock>) и помещает все это в буфер клавиатуры. Буфер находится в начале памяти по адресу 0040h:001Eh и может занимать максимум 32 байта, что соответствует 16 нажатым клавишам. Если программа-обработчик не успевает очищать буфер по мере того, как клавиши нажимаются, то все сверх 16 нажатий пропадает (а компьютер начинает жалобно пищать динамиком). Читать буфер можно напрямую (указатели на "голову" и "хвост" находятся по адресам 0040h:001Ah и 0040h:001Ch, если они равны, то буфер пуст), но смысла в этом особенного нет — позже мы увидим, как это можно делать "законным" путем.

В буфере клавиатуры отводится два байта на каждый символ — один из них представляет собой скан-код (но уже, как вы поняли, не совсем тот, что поступал в порт 60h), второй — символьный код (для управляющих клавиш он равен нулю). Символьный код зависит от нажатых одновременно с данной клавишей клавиш-модификаторов, а его экранная интерпретация — также от текущей кодовой страницы. Именно страницы (т. е. таблицы однобайтных кодов), которая загружается на уровне BIOS, а не раскладки клавиатуры. Это концептуально разные вещи — в DOS (и основанных на ней Win9*x*) кодовая страница загружается обычно при загрузке системы, в зависимости от локализации. Может даже не загружаться вообще, локализация в DOS часто вообще не требовалась — переключение кодовой страницы производила резидентная программа-переключатель раскладки. Подчеркнем еще раз: переключение раскладки и переключение кодовой страницы — в DOS два разных действия. Есть DOS-программы, которые переключают только страницу (т. е. фактически экранные шрифты — известная программа Evafont Петра Квитека позволяет автоматически создавать подобные утилиты), не переключая раскладку, теоретически можно сделать и наоборот, хотя вряд ли кому это нужно. Вы спокойно можете разработать свою собственную кодовую таблицу

(что равносильно разработке нового экранного шрифта) и "подсунуть" ее системе — автор этих строк так заставлял свои DOS-программы отображать надстрочные и подстрочные индексы.

В Windows же все это сделано настолько запутанно, что дальше некуда, причем еще и организовано несколько различным образом в 9*x* и в NT (XP). Там было введено понятие текущего "языка" (кроме отдельно "раскладки" и "кодовой страницы") — изначально это просто означало совокупность кодовой страницы и собственно раскладки, что логично. Но с переходом к Unicode понятие "языка" фактически стало жить самостоятельной жизнью, а к чему это приводит — мы подробнее обсудим в *главе 8*. Там же мы более подробно поговорим и о различных кириллических кодировках.

Операционная система (DOS или Windows) может использовать поверх всего этого и другие таблицы интерпретации, параллельно с кодовой страницей. Под "другими таблицами" имеются в виду не только национальные таблицы символов, но и реализация, например, такой функции, как ввод символа по его номеру с нажатой клавишей <Alt> (напомню, работает только левый <Alt>, только с цифровой клавиатурой, и только при включенном <Num Lock>), или популярного в DOS-программах представления управляющих кодов (символьные коды 01—31), которые невозможно набрать на клавиатуре прямо, через нажатие <Ctrl>+<*буква_алфавита*>. Скажем, действие, аналогичное нажатию <Enter>, т. е. ввод кодов 10 + 13, при этом выглядит, как последовательность нажатий <Ctrl>+<J>, <Ctrl>+<M>[1].

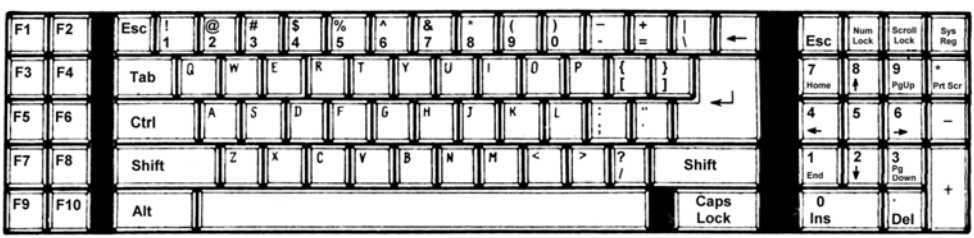


Рис. 6.1. Оригинальная 84-кнопочная клавиатура IBM PC

По всем этим причинам до оригинальных скан-кодов добраться непросто, но, по счастью, скан-коды в чистом виде на практике могут потребоваться нечас-

[1] Отметим, что само нажатие клавиши <Enter> возвращает код только 13 ("возврат каретки", <CR>), а вставку дополнительно кода 10 ("перевод строки", <LF>) для обозначения конца строки в нужных случаях производит ОС, причем только ОС от Microsoft. Другие системы используют один из этих символов: в MacOS употребляется CR, в UNIX — LF. По этой причине Web-документы в принципе могут использовать любой из таких вариантов, что следует учитывать при просмотре HTML-документов в двоичном представлении. Об особенностях действия символов перевода строки в Delphi-компонентах см. далее по тексту.

то, а для точной идентификации клавиш хватит и того, что предоставляет система. Скан-коды присваивались разработанной одновременно с IBM PC 84-кнопочной клавиатуре подряд (слева-направо, сверху-вниз), поэтому, например, клавише <Esc> присвоен скан-код 01, а следующий код (02) присвоен не клавише <F1>, как бы следовало ожидать, глядя на современную стандартную клавиатуру, а клавише <1>, которая на той клавиатуре следовала сразу за <Esc> (рис. 6.1). Еще большая путаница с системными клавишами на дополнительной клавиатуре (слева от цифровой), которые на 101/104-кнопочной клавиатуре дублируют функции цифровой клавиатуры — это <Home>, <End>, клавиши управления курсором и т. п. Дублирующими являются и клавиши около цифровой клавиатуры: <точка>, <умножение>, <плюс> и т. д. Общий принцип назначения скан-кодов состоит в следующем: основные скан-коды (они 7-битные, наличие 8-го бита означает код отпускания этой же клавиши) назначались тем клавишам, которые использовались в 84-кнопочном варианте, а вновь введенным дублирующим присваивалось то же самое значение с добавлением второго (старшего) байта, равного $E0 (это не относится к таким дублирующим клавишам, как, к примеру, правый <Shift> — она изначально была на 84-кнопочной клавиатуре). Однако, как вы увидите из таблицы, помещенной в *приложении 2*, фактически доступные скан-коды дополнительной клавиатуры повторяют аналогичные скан-коды клавиатуры цифровой, причем при выключенном <Num Lock> идентичны также и виртуальные коды, а вот при включенном виртуальный код клавиш на цифровой клавиатуре особый и отличается от цифр на основной клавиатуре. Особый код (и виртуальный, и скан-код) и у клавиш <+>, <–> и т. п., расположенных рядом с цифровой клавиатурой, потому для них отдельно и определены константы — идентификаторы виртуальных кодов.

Заметки на полях

Эта картина наблюдается и в DOS, и в Windows, т. е. обработка дополнительных клавиш ведется еще на уровне BIOS. В общем, сделано все, чтобы запутать честного программиста, который наивно предполагает, что все должно происходить согласно здравому смыслу. Если такой программист хочет найти логику в этой картине, то ему придется вернуться к истокам. На самом деле тут все прозрачно: упорная неразличимость левых-правых клавиш-модификаторов была, очевидно, заложена еще при создании первых моделей 101-кнопочной клавиатуры (а зачем, простите, их различать, если они задумывались именно как дублирующие?), и тогда же опрометчиво была заложена в BIOS, а дальше это требовалось соблюдать просто для совместимости. Аналогично произошло и с системными клавишами на дополнительной клавиатуре. В то же время отличающийся виртуальный и скан-код цифровых клавиш на цифровой и основной клавиатурах (а также левого и правого <Shift>) был заложен в систему с самого начала (еще во времена IBM PC, когда клавиатуры были только 84-кнопочные), а тогда системные программисты были другими, и не стали решать за пользователей, как им интерпретировать ту или иную клавишу. Вот все наслоения этих эпох мы и вынуждены теперь расхлебывать. С другой стороны,

нельзя не отметить, что неразличимость кодов на уровне ОС во многом оправдана, например, можно спокойно проектировать любую удобную конфигурацию клавиатуры (как, к примеру, в случае ноутбуков), и все будет совместимо без лишних сложностей.

Остановимся теперь подробнее на виртуальных кодах. Виртуальные коды — это то, что использует система для идентификации клавиш. С соответствующими предопределенными константами, действующими в Delphi, можно ознакомиться в файле Windows.pas (..\Source\Rtl\Win\Windows.pas). Сама Windows определяет в том числе константы для буквенно-цифровых клавиш, и в *приложении 2* они приведены, но в файле Windows.pas их нет, т. к. виртуальные коды буквенно-цифровых клавиш совпадают с кодами соответствующих символов *(см. приложение 3)*. В программах Delphi их надо определять через функцию `ord` или просто использовать цифровое значение кода символа. В таблице *приложения 2* суммированы сведения о предопределенных константах, виртуальных и скан-кодах для всех клавиш обычной 104-кнопочной клавиатуры, полученные из различных источников и проверенные автором по методике, которая излагается далее. По этой таблице можно определить разницу, например, между `vk_Add` и `vk_OEM_Plus` (угадайте с первого раза, какая из констант относится к "плюсу" на цифровой клавиатуре, а какая — на основной?) — нигде больше этих сведений в полном объеме вы не найдете. Отметим, что сейчас в моде расширенные клавиатуры с дополнительными функциональными клавишами (<F13>...<F24>), с клавишами управления браузером, медиаплеером и т. п., и хотя многие виртуальные коды таких клавиш также стандартизированы в Windows, пользоваться ими в прикладных программах общего назначения по понятным причинам не рекомендуется — кто знает, на каком компьютере ваша программа будет исполняться?

Заметки на полях

В связи с этим расскажу одну историю, которая врезалась мне в память, как типичный пример бездумного и безответственного отношения программиста к своим обязанностям. История, правда, больше относится к мышам, а не клавишам, но это не принципиально. В одном крупном провинциальном научно-производственном центре была разработана уникальная аппаратура для гидрографических исследований, а именно — для картографирования океанского дна. Это было еще во времена системы DOS, в которой, как известно, мыши разных производителей требовали непременно "родных" драйверов, в противном случае отказывались работать. Программист, который делал основную программу картографирования, решил, что будет очень здорово, если он возьмет и напишет собственный драйвер мыши конкретно для данного случая. Причины, которые его подвигли на такой шаг, остались неизвестными — то ли он просто не подозревал о существовании прерывания int33h, которое предоставляет "мышиный" интерфейс, то ли очень хотел показать, что лично без него в рейсе не обойдутся. В общем, на берегу программа отлично работала, потому

что все мыши в упомянутом НПЦ были закуплены единовременно и централизованно и тем самым, естественно, оказались одного производителя. А в море работать ничего не стало — на корабле мыши были другой системы. В результате все же нашлась одна мышь, которая работала, ее главный гидрограф каждый раз во время работы втыкал в компьютер, а после уносил к себе в каюту и запирал в личном сейфе. И все же в какой-то момент в ней сломалась левая кнопка, и только усилиями вашего покорного слуги, который ее починил "на коленке", программа исследований не была сорвана. Хочу предостеречь читателей от подобных "приколов" — вместо того, чтобы удивляться вашему искусству, пользователи скорее всего будут вас проклинать на чем свет стоит.

Теперь о том, как самому определить виртуальный и скан-код клавиши. Прочитать то, что находится в буфере BIOS, можно из DOS простой программкой на обычном Pascal, вызывающей прерывание int16h:

```
uses crt,dos;
var
xah,xal:byte;
begin
asm
mov ah,10h {функция чтения из клавиатуры}
int 16h
mov xah,ah
mov xal,al
end;
writeln('Symbol= ',xal,' Scan= ',xah);
end.
```

То есть после нажатия какой-нибудь клавиши в регистре `al` окажется код символа, а в `ah` — скан-код (при этом для управляющих клавиш в `al` будет 0). Недостатком этого способа является то, что не распознается нажатие некоторых системных клавиш (типа <PrintScrn>), и, главное, клавиш-модификаторов (<Alt>, <Ctrl> или <Shift>). Но мы же живем в эпоху Windows! А там есть замечательная функция `MapVirtualKey`, которою мы и используем — через нее мы можем узнать скан-код клавиши по известному виртуальному коду (или наоборот). Создадим новый проект (на диске в папке Glava6\1), поместим на форму компонент `Memo`, установим для формы параметр `KeyPreview` в `True`, и напишем следующий обработчик события `onKeyDown`:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
Shift: TShiftState);
var
st:string;
xscan:integer;
begin
xscan:=MapVirtualKey(Key,0);
```

```
st:=' Scan ='+IntToStr(xscan)+ ' '+hexb(byte(xscan))+'
   Key='+IntToStr(Key)+' '+hexb(byte(Key));
Memo1.Lines.Clear;
Memo1.Lines.Add(st);
end;
```

Для удобства вывод производится сразу в двух представлениях: в десятичном и шестнадцатеричном, для этого используется описанный в *приложении 1* модуль Ariphm, который следует объявить в интерфейсной части модуля Unit1, кроме того, в папке с проектом необходимо, разумеется, разместить файл Ariphm.pas.

В компоненте `Memo` при нажатии клавиш нам будут возвращаться виртуальные коды и скан-коды тех клавиш, которые мы нажимаем. Такая программа нам позволит отследить скан-код и виртуальный код любой клавиши (кроме опять же <PrintScrn>), и даже узнать скан-код, если мы вручную подставим в функцию `MapVirtualKey` вместо `Key` соответствующую константу из *приложения 2*. Впрочем, узнать разницу в скан-кодах между правым и левым <Ctrl> и <Alt> она не позволит — в среде Windows 9*x* этого и не обещают, но и в XP все равно не получается. Подчеркнем, что не дает результатов и подстановка констант (типа `vk_RControl`), а процедура Delphi `onKeyDown` об этих различиях уж тем более "не знает". В *приложении 2* для таких констант приведены только "официальные" виртуальные коды (которые на практике все равно нигде не работают), а скан-коды указаны одинаковые, причем для этих клавиш работает и общий код с константами `vk_Control` (<Ctrl>) и `vk_Menu` (<Alt>). Еще одна пара клавиш, которые трудно разделить — основная и "цифровая" клавиши <Enter>. Не получается также узнать разницу в скан-кодах между упомянутыми ранее системными клавишами (<Home>, <End> и т. п) на цифровой клавиатуре и на дополнительной — как уже говорилось, все дополнительные клавиши имеют "настоящий" (на уровне "железа") скан-код с дополнительным байтом $E0 и "в лоб" их различить действительно нельзя, если только не писать соответствующий кусок кода на ассемблере с перехватом нажатия непосредственно в порту клавиатуры. Отметим, что клавиши левый-правый <Shift>, в отличие от указанных, функция различает — по крайней мере в Windows XP.

Но простой способ различить в Windows (любой версии) правый-левый <Ctrl> или <Alt>, дополнительный <Enter> от основного и вообще любую дополнительную клавишу с "настоящим" скан-кодом, содержащим второй байт $E0, все же имеется, и он состоит в том, чтобы проанализировать `LParam` в системном сообщении `wm_KeyDown`. 16—23 биты (третий байт) этого параметра содержит скан-код нажатой клавиши (указанный в таблице), а младший бит следующего байта (бит номер 24) определяет, какая из клавиш нажа-

та: если этот бит равен 1, то нажата расширенная (правая) клавиша, а если 0 — стандартная (левая)[2].

Если хотите, можно считать, что скан-код расширенной клавиши на величину 256 больше, чем основной, так, если основной скан-код <Ctrl> равен $1D, то код правой клавиши <Ctrl> будет равен $11D. Отсюда понятно, почему функция `MapVirtualKey` выдает одни и те же коды для левой и правой клавиши: она анализирует только один третий байт, опуская следующий по старшинству бит. Подчеркнем, что величина $11D ни в коем случае не может рассматриваться как именно значение скан-кода и служит лишь для наглядности представления.

Эти сведения мы используем в следующей главе для того, чтобы создать резидентный переключатель раскладки клавиатуры — специальную программу для замены стандартных клавиатурных комбинаций Windows с возможностью выбора клавиши. А сейчас попробуем создать механизм, позволяющий отслеживать указанный ранее параметр `LParam` в системном сообщении `wm_KeyDown`. И тут уж нам не обойтись без использования механизма ловушек — Hook, который позволяет перехватывать любые сообщения (мы о нем упоминали в предыдущей главе).

Клавиатурный шпион

Для того чтобы отработать создание ловушек и заодно создать заготовку для переключателя клавиатуры, мы сначала сделаем простой вариант ловушки, который будет ничем иным, как типичным Spyware — т. е. программой-шпионом, которая перехватывает нажатия всех системных клавиш и отправляет их в текстовый файл. Настоящий "шпион" должен еще регистрироваться в реестре для автоматического запуска и периодически отправлять созданный файл по электронной почте, а нам это ни к чему — мы будем создавать такой перехватчик, разумеется, не для шпионажа, и даже не просто для тренировки, а с вполне прагматической целью — изучение описанного ранее параметра `LParam` для всяких дополнительных клавиш, которые нельзя различить через `MapVirtualKey`.

Для того чтобы создать такой перехватчик, мы используем заготовку программы Layout из предыдущей главы (из папки Glava5\3). Перенесем проект в новую папку (Glava6\2 на диске) под другим именем (через пункт **File | Save**

[2] Подчеркнем, что определения "правый-левый" относятся только к <Alt>, <Ctrl>, <Enter> или <Shift>, но не к случаю дополнительной клавиатуры — скажем, расширенная клавиша <Home> на дополнительной клавиатуре физически находится левее основной на цифровой. Правильно называть такие клавиши дополнительными или расширенными.

Project As) — пусть он теперь называется KeySpy. Раз такое название (spy — шпион), то и иконка должна быть соответствующая (на диске она называется Eye.ico). Заменим ею MAINICON в файле Keyspy.res через Image Editor способом, уже описанным в *главе 5 в разд. "Заготовка"*. Затем внесем в проект следующие изменения: уберем из проекта все, что относится к горячей клавише (в том числе обработчик сообщения `WM_HOTKEY`, функции `RegisterHotKey` и `UnregisterHotKey`), а также сразу заменим неудобное сообщение, которое появляется при нажатии правой кнопки мыши на всплывающем меню — оно все равно нам понадобится в дальнейшем. Пока меню это будет состоять из одного пункта **Закрыть**. И не забудем заменить в процедуре `CreateMyicon` всплывающую подсказку на `Key Spy`.

Для создания всплывающего меню сделаем следующее. Сначала в секции объявления переменных вставим строки:

```
PopMenu: hMenu; {всплывающее меню}
Pos:TPoint; {позиция курсора}
```

А выше, перед словом **var**, добавим константу, которая будет идентифицировать пункт (пока единственный) нашего меню:

```
const
idmEXIT =1;
```

Теперь в конце программы (после оператора `CreateMyicon`) создадим всплывающее меню и добавим в него этот самый пункт:

```
. . . . .
popMenu := CreatePopupMenu;
AppendMenu(PopMenu,MF_STRING,idmEXIT,'Закрыть');
. . . . .
```

Осталось, во-первых, вовремя вызвать меню на экран, во-вторых, создать обработчик сообщения (`wm_COMMAND`), которое оно посылает. Вызываем его мы, естественно, нажатием правой кнопки мыши, как и раньше `MessageBox`:

```
. . . . .
if Msg = Ico_Message then {если это сообщение от иконки}
begin
if lpr=wm_RBUTTONUP then {была отпущена правая кнопка}
begin
 GetCursorPos(Pos); {узнаем позицию курсора мыши}
 SetForegroundWindow(FHandle);
 {установим окно вперед - так рекомендует Microsoft}
 TrackPopupMenu(PopMenu,TPM_RIGHTALIGN+TPM_RIGHTBUTTON,
 Pos.x,Pos.y,0,FHandle,nil); {показываем меню}
```

```
 PostMessage(FHandle, WM_NULL, 0, 0);
   {это тоже рекомендует Microsoft}
end;
end;
. . . . .
```

Теперь обработчик (ниже, внутри той же процедуры WindowProc):

```
. . . . .
if Msg = wm_COMMAND then
if wpr =idmEXIT then
begin
Shell_NotifyIcon(NIM_Delete,@noIconData); {удаляем иконку}
halt; {закрываем программу}
end;
. . . . .
```

Если кому интересно — подробности об использовании этих функций и сообщения wm_COMMAND см. в [14, 16]. Перед созданием ловушки в программе надо еще подготовить файл, куда будем сбрасывать перехваченные коды. Для этого в начале добавим переменную ft: textfile, а в конце программы (после создания меню, но перед циклом сообщений) будем его создавать:

```
. . . . .
assignfile(ft,'c:\Keyhook.txt'); {создаем файл в корневой
       директории}
try
reset(ft); {пробуем открыть}
except
rewrite(ft); {если не получается, то создаем}
end;
closefile(ft); {закрываем файл}
. . . . .
```

Обратите внимание, что при указании имени файла нужно указывать весь путь, т. е. размещать его в конкретной папке — иначе наша ловушка его может не найти. Мы поместили файл туда, где его проще всего потом разыскать, но настоящие "шпионы", разумеется, прячут его куда подальше — в недра системных папок.

Приложение мы подготовили, теперь перейдем к созданию собственно ловушки. Ловушки бывают локальные и глобальные. *Локальная ловушка* — это фактически параллельный дополнительный поток в нашем приложении, и перехватывать нажатия клавиш извне она не умеет, так что придется сооружать глобальную ловушку. *Глобальные ловушки* помещаются в DLL — при-

дется создавать отдельную DLL-библиотеку и, главное, потом всюду ее "таскать" вместе с приложением. Поморщимся, но что поделаешь?

Не удивляйтесь, но, несмотря на громкое название, "шпионская" ловушка — одна из самых простых, потому что она только перехватывает клавиши и записывает их в файл (когда мы перейдем к переключателю, все будет несколько сложнее). Для создания DLL выполним команду **File | New | Other | DLL Wizard** и в получившуюся заготовку впишем текст из листинга 6.1 (заменив им все, что нам предлагает Delphi).

Листинг 6.1. Библиотека для "шпионской" ловушки

```
library Spyhook;

uses
Messages, Windows, Ariphm;

var
HookHandle:hHook;
ft:textfile;

function KeyboardProc(Code: Integer; wParam: wParam; LParam: LParam): in-
teger; stdcall;
begin
if code<0 then Result:=CallNextHookEx(HookHandle, code, WParam, LParam)
else
if byte(LParam shr 24)<$80 then {только нажатие}
begin
try
append(ft); {пробуем открыть файл для добавления}
except
exit; {если не получается - выход}
end;
writeln(ft,'Key(wParam) =$',hexw(wParam),' lParam
    =$',hexlong(lParam));
closefile(ft);
Result:=CallNextHookEx(HookHandle, code, WParam, LParam);
end;
end;

procedure SetHook; stdcall; {установка ловушки}
begin
HookHandle := SetWindowsHookEx(WH_KEYBOARD,KeyboardProc, hInstance, 0);
end;
```

```
procedure DelHook; stdcall; {удаление ловушки}
begin
UnhookWindowsHookEx(HookHandle);
end;

exports
SetHook, Delhook;
begin
assignfile(ft,'c:\Keyhook.txt');
end.
```

Заметим, что запись в дисковый файл прямо из ловушки — дело довольно рискованное с точки зрения устойчивости программы и системы в целом. Прежде всего, потому что жесткий диск — устройство медленное, и вся процедура будет тормозить, к тому же мало ли что может случиться с дисковым файлом. Поэтому правильно организованный "шпион" должен передавать нажатие в основную программу, а она уже — писать в файл, что намного безопаснее. На том, как именно передаются параметры, мы остановимся позже. А здесь не будем себе морочить этим голову, т. к. в данном случае программа делается исключительно в утилитарных целях.

Заметки на полях

По правилам, изложенным в справке от Microsoft, программа при обработке системной ловушки должна проверять параметр code на условие <0 и в этом случае немедленно вызвать функцию CallNextHookEx с выходом из процедуры, а обрабатывать сообщение (от клавиатуры) только, если code = HC_ACTION. Какими только вариантами реализации этого условия не полнятся исходные тексты на соответствующих форумах! От полного игнорирования до сложных конструкций **if... then... else... else... else**. На самом деле Win32 никогда не возвращает значения code меньше 0 (это было только в Windows 3*x*), и именно поэтому любые варианты работают практически одинаково, но мы все же будем поступать по правилам. А параметр HC_ACTION (только для сообщений от клавиатуры) показывает, должна ли функция обработать сообщение. На практике он не оказывает никакого влияния на обработку события — по крайней мере, мне такого влияния установить не удалось.

Другое дело, что установленные ловушки никак не сортируются — это официальная информация! — потому никто не гарантирует, что сообщение до вашей ловушки дойдет. И все же на практике этот механизм работает на удивление прилично. Хотя, например, автором при работе в среде Windows 98 были замечены редкие проскоки событий нажатия клавиш мимо ловушки. В другой (но, несомненно, сделанной по тому же механизму) программе для переключения раскладки, скачанной из Сети, наблюдался тот же эффект. Надежность перехвата клавиши зависит от установленного в системе софта, которого у автора в Windows 98 не просто много, а очень много, и на некоторых связанных с этим особенностях использования ловушек мы еще остановимся.

Пояснять подробно использование функций SetWindowsHookEx и KeyboardProc я не буду — это занятие долгое, и вы прекрасно можете найти все подробности в Интернете. Выражение byte(LParam **shr** 24)<$80 означает следующее: внутри скобок длинное слово LParam сдвигается на три байта вправо, т. е. младшим оказывается старший байт, затем полученное число усекается до байта непосредственным преобразованием типов (последнее, кстати, в данном случае необязательно и сделано только для наглядности). Как мы знаем из вышеизложенного, если старший бит в полученном байте равен 1 (т. е. байт больше 127 = $7F), то это код отпускания, а нам нужен код нажатия, отсюда и условие. Если бы мы его не включили, то каждая клавиша отслеживалась бы дважды: на нажатие и на отпускание. Заметим, что есть, разумеется, и иные способы ограничить действие только одним из сообщений.

Перехваченные коды клавиш (виртуальный код в wParam и все остальные подробности, включая скан-код и признак расширенной клавиши — в LParam) сохраняются в файле в Hex-форме благодаря использованию модуля Ariphm, как и ранее. Между **begin** и **end** выполняется код инициализации — в данном случае ассоциация с именем файла. Имейте в виду, что DLL нужно компилировать отдельно от использующей ее программы. Формально это совершенно разные программы, что, конечно, отвечает настоящему положению вещей, поэтому сначала создайте проект библиотеки отдельно, а при отладке удобно запускать два экземпляра Delphi — каждый со своим проектом (только не запутайтесь в плавающих окнах!), т. к. открыть два проекта в одном окне Delphi не позволит. Проект с DLL назовем Spyhook, и получим в результате компиляции файл Spyhook.dll.

Теперь нам осталось запускать ловушку вместе с нашим приложением и затем ее как-то удалять. Для этого используем экспортируемые из DLL процедуры SetHook и DelHook. Сначала допишем в самом начале (после предложения **uses**) ссылки на эти функции (предполагаем, что DLL находится либо в той же папке, что и программа, либо в системной папке, скажем, в Windows или Windows\System):

```
. . . . .
procedure SetHook; stdcall; External 'Spyhook.dll';
procedure DelHook; stdcall; External 'Spyhook.dll';
. . . . .
```

Заметки на полях

Обратите внимание, что обычно в настоящее время при написании DLL индексы экспортируемых функций не используются, как рекомендует, например, [3] — начиная, по крайней мере, с Windows 95 эти функции ищутся по имени, а не по индексу. Хотя строковый поиск несколько дольше, чем численный — какое это имеет значение в современных компьютерах с гигагерцовыми процес-

сорами? Практиковавшееся ранее использование индексов — когда каждую функцию вызывали по ее номеру, присвоенному программистом при написании DLL — гораздо менее удобно. Единственное затруднение, которое может при этом возникнуть при вызове функций из чужих библиотек, — то, что в C и C++ разрешены имена с использованием знака "@", а Delphi их "не понимает". Это ограничение можно обойти, организовав вызов функции из библиотеки, например, CppMade.dll с именем `CppCorrectName@0` по следующему образцу:

```
function DelphiCorrectName:integer external 'CppMade.dll' name
'CppCorrectName@0';
```

Такой способ переименования через **name** можно использовать и для придания видимых извне имен в самой DLL. Правда, внешние имена при этом будут чувствительны к регистру букв — а нужны ли нам такие сложности? Но следует иметь в виду, что другие могут так не посчитать и потому при вызове функций из чужих DLL лучше соблюдать тот регистр, который указан в авторском описании библиотеки.

Теперь, наконец, включим вызовы этих функций в нужные места: устанавливаем ловушку мы при запуске программы, поэтому включаем вызов в конце, после создания файла:

```
. . . . .
closefile(ft); {закрываем файл}
SetHook; {запускаем шпиона}
. . . . .
```

А удаление включим перед вызовом `halt` (он у нас повторяется два раза):

```
. . . . .
Shell_NotifyIcon(NIM_Delete,@noIconData); {удаляем иконку}
DelHook; {удаляем шпиона}
halt; {закрываем программу}
. . . . .
```

Заметки на полях

При отладке в Delphi программ, использующих ловушки или другие функции, находящиеся в отдельно запускаемой DLL, обязательно закрывайте запущенную в отладочном режиме программу перед выходом из Delphi или перед загрузкой другого проекта — особенно при работе в Windows XP. Попытка автоматически прервать процесс может привести к полному зависанию всей системы.

Все готово — можете запускать программу и отслеживать клавиши. Только перед просмотром файла Keyhook.txt ловушку желательно каждый раз выгружать — если используемый просмотрщик при открытии файла монопольно им завладевает (как это делает MS Word), то любое случайное нажатие на клавиши, скорее всего, обрушит открывшую программу, несмотря на наши предосторожности. Это не относится к "правильно" написанным редакторам,

которые каждый раз после открытия файла загружают его содержимое, затем закрывают его и оставляют в покое (при необходимости отслеживая сторонние изменения в нем при активации их окна). К таким редакторам, в частности, относится и встроенный редактор Delphi. Автор рекомендует для работы с текстовыми файлами (в частности, с HTML) вообще использовать не Блокнот, а какой-нибудь из подобных редакторов — например, Edit Plus.

Полученный "шпион" перехватывает абсолютно все клавиши, как вы можете убедиться сами, и возвращает при этом различный скан-код (третий байт + младший бит четвертого в параметре `LParam`). Именно с его помощью автор обнаружил у своей клавиатуры интересную особенность, о которой раньше и не подозревал — правый <Alt>, оказывается, у нее отсутствует вообще, а находящаяся на его месте клавиша имитирует два нажатия: код обычной (левой) клавиши <Alt> + код также обычной (левой) <Ctrl>. Не очень понимаю, зачем пришло такое в голову разработчикам этой замечательной клавиатуры (обычной конфигурации, но с "ноутбучными" клавишами) — ведь одновременное нажатие <Alt>+<Ctrl> требуется разве что, если вы в отчаянии посылаете систему на известные "три клавиши", а из знакомых автору продуктов употребляется только в Borland ModelMaker (далее мы, впрочем, тоже это сочетание используем). То, что это делает именно клавиатура сама, а не система — гарантированно, т. к. никаких драйверов я не ставил. Но данная особенность — хороший пример того, почему нельзя назначать переключателю раскладки какую-то определенную клавишу, необходимо предоставить пользователю выбор, иначе на каких-то компьютерах (особенно на ноутбуках) может оказаться, что ничего не работает.

ГЛАВА 9

Vis-a-vis

Организация диалогов, операции "один обработчик — много действий", передача фокуса ввода и другие хитрости

> Он остановился. Лицо его перекосилось ужасом. Панель с клавишами управления изогнулась, снова выпрямилась и бесшумно соскользнула на пол.
>
> *А. и Б. Стругацкие, "Путь на Амальтею"*

Эта глава есть естественное продолжение темы работы с клавиатурой, т. к. большая часть самостоятельно написанных обработчиков нажатия клавиш, за исключением, может быть, сложных приложений для работы с текстом, относится именно к организации диалогов. Windows автоматически предоставляет все возможности для управления диалогами, но поскольку они ориентированы в основном на мышь, то их едва ли можно признать удовлетворительными, и, как правило, более-менее сложные диалоги приходится дорабатывать — или вообще конструировать "с нуля" вручную.

Особенности работы с клавиатурой в Delphi

Один из самых простых вариантов работы с клавишами — через событие `onShortCut` формы — мы рассмотрим в *главе 15*. Этот вариант отличается тем, что он перехватывает все клавиши, в том числе и стрелки управления курсором, идентифицируя виртуальный код нажатой клавиши. При этом работа происходит, что немаловажно, независимо от установки параметра `KeyPreview` — т. е. все нажатия клавиш, кроме перехваченных, будут доходить куда надо. Здесь же мы остановимся на стандартных событиях Windows.

Виртуальные коды служат параметром в процедуре Delphi по событию `onKeyDown`, при этом можно еще отдельно проанализировать, нажата ли одновременно какая-нибудь из клавиш-модификаторов, и таким образом искусственно определить, какой именно символ имел в виду пользователь, когда на-

жимал на клавишу. Но вот определить при этом, нажата ли клавиша <Caps Lock>, не удастся. Зато в процедуре onKeyPress возвращается именно символ, с учетом национальной раскладки и регистра, однако системные клавиши там не отслеживаются. Кстати, если вы попробуете использовать функцию MapVirtualKey *(см. главу 6)* в обработчике onKeyPress (разумеется, с параметром ord(Key) вместо просто Key), то увидите, что она выдает правильное значение скан-кода только при нажатии буквы в английской раскладке в верхнем регистре, в остальных случаях выдается чепуха.

Поэтому, если нужно отследить нажатие системных клавиш и одновременно получить различия между русскими и английскими буквами, приходится выворачиваться — например, включать оба обработчика одновременно. Типичный пример такой дурацкой ситуации — когда вы самостоятельно пишите диалог с кнопками **Да** (Yes) и **Нет** (No) и при этом еще желательно, чтобы сфокусированная кнопка срабатывала при нажатии <Enter>. Дело в том, что английская буква "Y" (ответ "Yes") и русская буква "н" (ответ "Нет") расположены на одной клавише, поэтому использовать событие onKeyDown не получится, на все варианты ("Y", "y", "Н", "н") обработчик выдаст одно и то же значение Key=59 (vk_Y). А если использовать событие onKeyPress, то не удастся отследить нажатие <Enter>. Один из вариантов выхода из этой ситуации без того, чтобы загромождать программу множеством обработчиков, состоит в том, чтобы использовать функцию API GetKeyboardLayout (возвращающую тот самый национальный идентификатор языка, с которым мы мучались в предыдущих главах — число, в двух младших байтах которого содержится собственно идентификатор) для определения того, какая раскладка действует в данный момент. Но подобное требуется все же крайне редко — обычно удается обойтись процедурами onKeyDown и onKeyPress для того, чтобы опознать любой вариант результата диалога. Давайте несколько искусственно проиллюстрируем использование этих процедур для организации диалога на примере проекта SlideShow (в варианте из *главы 4*, версия 1.20). Сначала мы приведем пример того, как можно создать собственное диалоговое окно, которое заменит нам невыразительный MessageBox. В данном случае — для отклика на попытку запустить программу повторно.

Диалог типа MessageBox

Для этого перенесем, как всегда, проект из папки Glava4\2 в новую папку (Glava9\1), присвоим ему опять промежуточный номер версии 1.20.2 и присоединим к проекту еще одну форму (Form3 — она и будет окном диалога). Перенесем ее, как и Form2, в список **Available forms** (пункт **Project | Options | Forms**), установим для BorderStyle значение bsNone, для Position значение poDesktopCenter, а соответствующий модуль назовем fmessage.pas.

Размеры подгоним на глаз и установим для формы красный цвет (пункт Color в свойствах). Затем поместим на форму компонент Label, для которого установим в свойствах Font — цвет шрифта белый, размер 10 и стиль "полужирный" (можно и сам шрифт заменить, например, на Arial — будет еще красивее, только шрифты надо выбирать из тех, что заведомо установлены на любом компьютере). Текст в Label пока не пишем. Кроме этого, установим на форму две стандартные кнопки, назвав их ButtonYes и ButtonNo (их цвет изменить не получится, для этого нужно взять кнопки типа BitBtn с закладки **Additional**, но так даже пикантней). В результате получим заготовку формы, показанную на рис. 9.1.

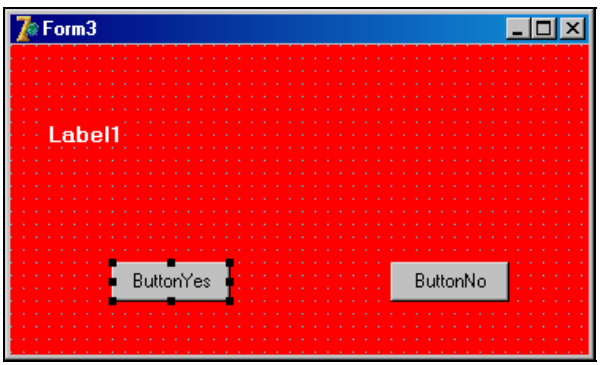


Рис. 9.1. Заготовка формы-сообщения

Окончательно выровняем компоненты позже, в процессе отладки, а сейчас еще установим свойство TabOrder у кнопки ButtonYes равным 1, а у кнопки ButtonNo, соответственно, 0. В этом случае первой при запуске сфокусируется кнопка ButtonNo, а при нажатии клавиши <Tab> фокус перейдет на ButtonYes. О таких вещах следует всегда думать, если вы собираетесь сделать программу удобной для работы. Если разнообразных компонентов много и вы не уверены, какой именно компонент будет фокусироваться первым, то можно не возиться с TabOrder, а в момент показа компонента принудительно установить фокус "куда надо" с помощью процедуры SetFocus (подробнее о передаче фокуса мы поговорим позже).

Теперь напишем следующий текст, заменив им все, что находится в основном модуле (SlideShow.dpr) там, где программа определяет, что один экземпляр уже запущен:

```
if st='SlideShow' then
begin
 Form3:=TForm3.Create(Application); {создаем экземпляр формы}
 Form3.Show; {показываем заставку}
```

```
 while FlagExit=0 do Application.ProcessMessages; {пустой цикл,
                        пока ждем ответа}
 if FlagExit=1 then {была нажата кнопка ButtonNo}
 begin
  Form3.Free; {уничтожаем Form3}
  exit; {выходим из программы}
 end else break {была нажата кнопка ButtonYes -
     выходим из цикла и запускаем программу вторично}
end;
```

Переменную-флаг FlagExit мы будем использовать для того, чтобы определить, каков был результат диалога. Естественно, надо ее добавить в число переменных в модуле fmessage:

```
var
Form3: TForm3;
FlagExit:byte=0;
```

Вообще-то мы начали немного с другого конца — логично сначала бы обеспечить диалог (ради которого все затевалось), а потом уже писать процедуру-результат его окончания. Но в данном случае это безразлично — одно без другого все равно не заработает. Переходим к форме Form3 и первым делом заполним компонент Label1 и заголовки кнопок. Для этого создаем обработчик события onCreate, в который вставляем следующие строки:

```
procedure TForm3.FormCreate(Sender: TObject);
begin
 if word(GetKeyboardLayout(0)) = $419
        {т. е. если язык по умолчанию русский}
 then
 begin
  Label1.Caption:='Вы действительно хотите
         запустить'+#10+#13+'программу повторно?';
  ButtonYes.Caption:='Да';
  ButtonNo.Caption:='Нет';
 end else
 begin
  Label1.Caption:='You really want to start program next time?';
  ButtonYes.Caption:='Yes';
  ButtonNo.Caption:='No';
 end;
end;
```

В этой процедуре мы делаем следующий фокус: определяем текущую раскладку клавиатуры (как мы знаем из *главы 7*, при запуске она соответствует

той, которая установлена в системе по умолчанию) и в зависимости от этого определяем язык диалога. Вставка в строку символьных кодов #10 и #13 (<LF> и <CR>, о них см. сноску 1 в *главе 6*) позволит разорвать строку и не делать форму слишком длинной. Если бы мы формировали заголовок статически на этапе создания формы, то разорвать строку нам бы не удалось. Английская строка короче, потому ее мы не разрываем. В принципе в свойствах `Captions` или `Text` компонентов Delphi можно использовать только один символ — либо <LF>, либо <CR> — результатом все равно будет перенос текста на следующую строку, но порядка ради мы всегда будем использовать их вместе. Причем перенос будет работать только в тех компонентах, где это возможно — например, в заголовке формы или в однострочном редакторе `Edit` либо проигнорируется перенос, либо вторая половина строки исчезнет вообще.

Теперь создаем в модуле fmessage обработчики щелчков на соответствующих кнопках:

```
procedure TForm3.ButtonYesClick(Sender: TObject);
begin
 FlagExit:=2; {обеспечит повторный запуск}
end;

procedure TForm3.ButtonNoClick(Sender: TObject);
begin
 FlagExit:=1; {обеспечит выход}
end;
```

Запустите и проверьте — все работает. Так мы сформировали стандартный диалог, который будет работать при нажатии кнопок мыши и от клавиши <Enter>. Но нам еще хочется использовать горячие клавиши. Для этого создадим такой обработчик события `onKeyPress`:

```
procedure TForm3.FormKeyPress(Sender: TObject; var Key: Char);
begin
 if (key='Y') or (key='y') or (key='д') or (key='Д') then FlagExit:=2;
 if (key='N') or (key='n') or (key='т') or (key='Т') then FlagExit:=1;
end;
```

Запустите и проверьте — все работает, причем надписи зависят от раскладки по умолчанию (которая устанавливается в свойствах клавиатуры из панели управления). На самом деле пример этот вполне искусственный, т. к. раскладка по умолчанию есть вещь независимая (слава богу!) от языковой версии Windows — у многих, включая автора этих строк, версия русская, а раскладка по умолчанию английская. И именно языковая версия нам точнее укажет на то, кто именно будет читать наши сообщения, но определение вер-

сии — отдельный вопрос. На самом же деле все это вообще не так делается — в Delphi есть специальный механизм интернационализации, который позволяет полностью перевести ваше приложение на тот или иной язык и даже автоматически запускать его в зависимости от версии ОС — как раз именно то, что недавно мы имели в виду. По всем этим причинам мы в дальнейшем повторять эту часть программы не будем — посчитаем, что это была просто иллюстрация.

А вот другой диалог, который нам будет устанавливать время задержки при демонстрации (пока оно у нас постоянно и равно 3 сек, см. *главу 2*), мы постараемся сделать как надо — это действительно необходимо в программе.

Диалог для установки таймера в SlideShow

Заново перенесем старый проект SlideShow в версии 1.20 из *главы 4* (Glava4\2) в новую папку (Glava9\2), присвоим ему номер 1.21 и несколько изменим его: восстановим в модуле slide.pas для переменной `mayClose` значение `False` по умолчанию (в секции объявлений переменных) и удалим из основной программы строки:

```
CreateMyicon; {создаем иконку в Tray}
Application.ShowMainForm:=False; {не показываем главную форму}
```

Теперь наша программа будет запускаться сразу после демонстрации заставки, а не сворачиваться в иконку — для подобного приложения это логичнее.

Нам нужен диалог с возможностью ввода чисел. Установим на форму еще один компонент `Panel` (он получит имя `Panel2`), а на него — `Label`, однострочный редактор `Edit` и компонент `UpDown` (он находится на вкладке **Win32** палитры компонентов), который пристыкуем к `Edit1` справа. Кроме этого, установим на панель кнопку `Button2` с заголовком `Ok`. В свойство `Caption` компонента `Label1` впишем `Установка интервала, с`. В свойство `Text` компонента `Edit1` поместим `3`, и свяжем между собой `Edit` и `UpDown`. Для этого в свойство `Associate` компонента `UpDown` необходимо вставить `Edit1` (через выпадающее меню)[1]. После подгонки размеров и положения компонентов наша панель должна выглядеть так, как показано на рис. 9.2, и располагаться посередине окна приложения.

Не забудем также установить для панели свойство `Visible` в `False` — сначала при запуске она должна быть невидимой. А в компоненте `UpDown1` установим минимальный (1) и максимальный (100) пределы изменений. Надо сказать, что в этом отношении Delphi несколько подкачала (точнее, не столько Delphi,

[1] Как мы уже говорили, в современных версиях Delphi есть и готовый специальный компонент для ввода чисел — `TSpinEdit`, и в дальнейшем я покажу, как его использовать.

сколько Windows) — обратной связи между компонентами Edit и UpDown нет, т. е. если вы вручную запишите в Edit число, большее максимума или меньшее минимума, установленного в UpDown, то программа это спокойно проглотит (спасибо уж и на том, что отслеживаются ошибки, связанные с вводом нечислового значения). По этой причине нам придется создать соответствующее ограничение вручную. Другой момент, связанный с компонентом Edit, мы уже упоминали в *главе 8* — при его фокусировке текст, находящийся в нем по умолчанию, выделяется, и последующий ввод заменит данный текст. Это далеко не всегда удобно, не говоря уж о том, что выделенный текст плохо виден и его непросто прочитать (насколько это неудобно, вы можете видеть на примере обращения к полям Object Inspector в Delphi, да и вообще почти любого диалога в Windows). Как мы уже знаем, для отключения выделения достаточно установить свойство AutoSelect в False.

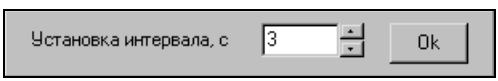


Рис. 9.2. Заготовка панели диалога для установки таймера

Теперь перейдем к реализации диалога. Для этого сначала в пункте главного меню (компонент MainMenu1) **Демонстрация**, ниже имеющегося подпункта **Запуск**, создадим разделитель (для этого надо создать пункт с заголовком "-"), а потом еще ниже введем пункт **Интервал**. Назовем его Int1 и присвоим ему горячую клавишу <Shift>+<F9>. Обработчик события по обращению к этому пункту будет таким:

```
procedure TForm1.Int1Click(Sender: TObject);
begin
{обращение к пункту меню Интервал}
  Panel2.Visible:=True; {показываем панель установки времени}
 Edit1.SetFocus; {фокусируемся на редакторе}
end;
```

После ввода значения времени мы нажмем кнопку **Ok** (Button2), и для этого события нужно написать следующий обработчик:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
{нажатие на кнопку Ok на Panel2}
if (StrToInt(Edit1.Text)<1) then Edit1.Text:='1';
if (StrToInt(Edit1.Text)>100) then Edit1.Text:='100';
Timer1.Interval:=StrToInt(Edit1.Text)*1000;
       {устанавливаем интервал таймера}
Panel2.Visible:=False; {закрываем панель}
end;
```

В первом операторе мы учитываем "тупость" компонента Edit: при вводе величины больше верхнего предела и ниже нижнего она принудительно приравнивается к соответствующему порогу, причем мы учли, что ошибка эта редкая, а если бы она встречалась чаще или была бы критичной, то следовало бы дополнительно выводить сообщение для оповещения пользователя о том, что он ошибся.

Но это еще далеко не все. Во-первых, если мы работаем с клавиатуры, то прямо нажать на клавишу <Enter>, чтобы закончить диалог, не получится — фокус находится на компоненте Edit и придется манипулировать с клавишей <Tab> либо, как большинство и поступает, хвататься за мышь. Это и неудобно, и, если можно так выразиться, "неконцептуально". Поэтому придется делать отдельный обработчик, чтобы отследить нажатие клавиши <Enter>. Так как в фокусе при работе с клавиатурой находится компонент Edit1, то обработчик мы сделаем для него:

```
procedure TForm1.Edit1KeyDown(Sender: TObject; var Key: Word;
            Shift: TShiftState);
begin
{нажатие на Enter после редактирования интервала}
if key=vk_Return then
begin
   if (StrToInt(Edit1.Text)<1) then Edit1.Text:='1';
   if (StrToInt(Edit1.Text)>100) then Edit1.Text:='100';
   Timer1.Interval:=StrToInt(Edit1.Text)*1000;
        {устанавливаем интервал таймера}
   Panel2.Visible:=False; {закрываем панель}
end;
end;
```

Но и это еще не все! Если пользователь работает не с клавиатурой, а с мышью через компонент UpDown, то нажать на кнопку **Ok** ему ничего не стоит, но еще лучше дать ему возможность также выйти из диалога по двойному щелчку просто на редакторе или на панели — это будет совсем грамотно (компонент Panel не отслеживает события клавиатуры, но события мыши отслеживают все без исключения визуальные компоненты). Казалось бы, кнопку **Ok** вообще можно было бы тогда не вводить, но это рассуждение справедливо только теоретически — выход из ситуации должен быть указан явно (и мы почти нарушаем данное положение, когда не вводим в меню специальный пункт для выхода из программы — просто его никто не станет искать, т. к. кнопка-крестик всегда перед глазами).

Для того чтобы сделать, как написано, придется создать еще два обработчика — процедуры Edit1DblClick и Panel2DblClick. Таким образом, у нас по-

лучится целых три идентичных процедуры — уже есть процедура Button2Click (процедура по нажатию клавиши <Enter> отличается от них). Нельзя ли как-нибудь упростить и сократить это дело? Запросто: надо в закладке **Events** для соответствующего компонента (Edit1 и Panel2) выбрать нужное событие (onDblClick) и через выпадающий список по кнопочке справа выбрать процедуру из имеющихся — в нашем случае Button2Click. Тогда по двойному щелчку на панели или в поле редактора будет выполняться та же самая процедура, что и по нажатию кнопки. Этот фокус типа "один обработчик — много действий" не пройдет, если процедуры разного типа — так, назначить событию нажатия клавиши обработчик щелчка мышью не получится. Это ограничение все же можно преодолеть, если использовать невизуальный компонент TActionList, но, на мой вкус, предлагаемый механизм там слишком сложен, чтобы имело смысл его использовать в таких простых случаях.

Но и это еще не все! Если бы мы оставили все, как есть, то поступили бы совершенно в русле "программистского мышления", против которого я так страстно выступал в предыдущих главах. В самом деле — а что будет, если пользователь вызовет созданный диалог на экран, а делать ничего не захочет: например, окажется, что диалог вызван по ошибке — мышь соскользнула при обращении к меню? "Программист" непременно скажет на это, что надо просто нажать на **Ok** — и ничего не изменится, но будет прав только формально, ибо пользователь-то априорно не знает, изменится или не изменится, мало того — он не хочет и не обязан терять время на обдумывание этой ситуации. Поэтому нам надо придумать простой механизм, который убирал бы вызванную панель диалога без каких-либо специальных действий. Самый простой механизм такого рода — убирать панель при осуществлении любого другого действия, доступного в программе. То есть во все процедуры-обработчики событий (кроме события по таймеру — ведь диалог может быть запущен во время демонстрации), в самом их начале, надо добавить строку:

```
Panel2.Visible:=False; {закрываем панель}
```

В том числе надо добавить эту строку и в обработчик события OnDeactivate формы Form1 — тогда у нас диалог будет пропадать и при сворачивании формы в иконку. Причем учтем, что самое естественное действие в случае, если диалог вызван по ошибке — щелкнуть вне его на поле окна. Поэтому мы не поленимся, и специально на этот случай создадим обработчик события onClick для компонента Image1 (он у нас занимает большую часть площади окна и расположен как раз под панелькой):

```
procedure TForm1.Image1Click(Sender: TObject);
begin
Panel2.Visible:=False; {закрываем панель}
end;
```

А почему бы не создать точно такой же обработчик для щелчка по Panel1 или, к примеру, по самой форме (рамке вокруг нашего "экрана") — вдруг пользователь промахнется? Проделаем тот же фокус, что и раньше — перейдем к закладке **Events** нужного компонента (Panel1 и Form1), выберем событие onClick и в выпадающем списке укажем нужный обработчик (теперь уже Image1Click). Проверьте — все работает!

Диалог с установкой нескольких параметров и сохранение установок через INI-файлы

Пусть мы хотим в одном диалоге устанавливать сразу несколько вещей — в нашем приложении, кроме интервала таймера, можно устанавливать цвет фона "экрана" (т. е. компонента Panel1). Кроме того, будем все это запоминать в качестве пользовательских установок. Как простейшим способом сохранять установки пользователя с использованием INI-файла, мы уже знаем из *главы 7*, но на этот раз для разнообразия поступим по правилам: для работы с INI-файлом используем класс TIniFile. (Между прочим, DSK-файл проекта Delphi, который я предлагаю при переносе править вручную, именно такой INI-файл и есть.) Заодно провернем следующую операцию — введем пункт выбора папки в дополнение к открытию файла (указывать конкретный файл или папку целиком — концептуально разные действия), причем опять же для разнообразия используем компоненты Delphi. Но мало того, покажем язык в сторону Редмонда и будем в том же INI-файле запоминать еще и последнюю просмотренную папку, с тем, чтобы при загрузке она устанавливалась автоматически.

Перенесем созданный проект SlideShow из папки Glava9\2 в новую папку (Glava9\3) и изменим версию на 1.30. Начнем с задания папки. В секции объявления переменных добавляем переменную-строку stpath и сразу инициализируем ее значением 'C:\':

```
stpath:string = 'C:\';
```

Теперь добавим в процедуру Form1.Create такую строку:

```
ChDir(stpath); {устанавливаем текущую папку}
```

Если сделать диалог выбора папки на невидимой панели, как установку интервала таймера, нам придется вручную отключать доступ к меню и кнопке — иначе пользователь может предпринять еще какие-то действия, не закрывая диалога, что в данном случае неправильно. Поэтому нам придется расположить диалог на отдельной форме с вызовом ее в модальном режиме — хотя это и проще, но у автора имеется хроническая нелюбовь к модальным формам, которые выскакивают, как чертик из табакерки, и обязывают

пользователя что-то предпринять, о чем он думать вовсе не хочет. Диалог на панели куда дружелюбнее, но запутаться, когда и что именно включать и отключать, очень просто, поэтому способ с формой надежнее и не столь хлопотен.

Присоединим к проекту новую форму (**File | New | Form**), она получит наименование `Form3`, а модуль для нее назовем papka.pas. У нее установим свойства `BorderStyle` в `bsDialog`, `Position` в `poDesktopCenter`, а `Caption` в `Выбор папки`. Что же касается довольно многочисленных компонентов для задания папки, о которых мы говорили в предыдущей главе, то наименее проблематичным, с точки зрения автора, является использование сочетания `ShellComboBox` с закладки **Samples** и старого `DirectoryListBox` с закладки **Win3.1** (в новом аналогичном компоненте `DirectoryOutline` есть досадные "фичи", которые приходится преодолевать шаманскими способами, к тому же достаточно лишних свойств, которые еще надо устанавливать). Вместо `ShellComboBox` логичнее было бы использовать также старый `DriveComboBox`, но он имеет настолько неэстетичный вид, что лучше потерпеть присутствие явно лишних для нашей цели пунктов вроде **Панели управления** в раскрывающемся меню `ShellComboBox`. Для выхода из диалога в данных компонентах не предусмотрено штатных методов (по щелчку мыши это делать нельзя, т. к. есть неоднозначность в использовании одинарных и двойных щелчков — например, `DirectoryListBox` по праву рождения использует только двойной щелчок для перехода на уровень ниже, в то время как в Windows 95 и выше это может устанавливаться системно). По такой причине мы внизу присовокупим кнопку **Ok**, роль кнопки **Отмена** будет играть единственная оставшаяся в соответствии со стилем `bsDialog` системная кнопка-крестик (рис. 9.3).

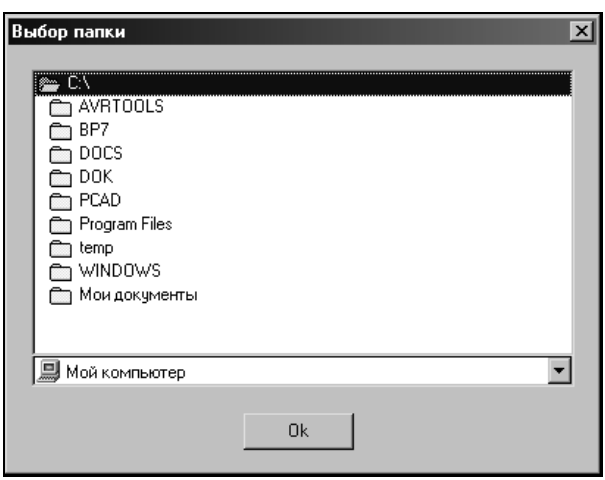


Рис. 9.3. Форма диалога для выбора папки проекта SlideShow (вид в работающей программе)

Свойство Root для ShellComboBox1 установим в rfMyComputer. После слова **implementation** в модуле papka.pas, чтобы получить доступ к глобальным переменным основного модуля, допишем строку **uses** slide; и создадим такой обработчик события onCreate:

```
procedure TForm3.FormCreate(Sender: TObject);
begin
{для первого раза}
 DirectoryListBox1.Directory:=stpath; {текущая директория}
 ShellComboBox1.Path:=stpath[1]+stpath[2]+'\'; {текущий диск}
end;
```

Добавим в меню **Файл** на форме Form1 пункт OpenFolder (**Перейти к папке**, горячая клавиша для него пусть будет <Ctrl>+<Alt>+<0>) и создадим следующий обработчик события щелчка на этом пункте:

```
procedure TForm1.OpenFolderClick(Sender: TObject);
begin  {открыть папку}
 Form3.ShowModal;
end;
```

И, наконец, в модуле papka.pas создадим следующие обработчики по щелчку на ShellComboBox и на кнопке Button3 (**Ok**):

```
procedure TForm3.ShellComboBox1Click(Sender: TObject);
begin {изменили диск}
try
 DirectoryListBox1.Directory:=ShellComboBox1.Path;
except
 Application.MessageBox('Диск недоступен','Ошибка',mb_OK);
 ShellComboBox1.Path:=stpath[1]+stpath[2]+'\'; {текущий диск}
end;
end;

procedure TForm3.Button3Click(Sender: TObject);
begin {Ок для выбора папки}
stpath:=DirectoryListBox1.GetItemPath(DirectoryListBox1.ItemIndex);
ChDir(stpath); {устанавливаем текущую папку}
DirectoryListBox1.Directory:=stpath; {на следующее открытие}
Form3.Close;
end;
```

Для диалога открытия файла OpenDialog1 ничего специально устанавливать не надо — он будет открываться на текущей папке (установленной у нас через процедуру ChDir).

Теперь займемся остальными установками. Перенесем пункт Int1 (**Интервал**) из меню **Демонстрация** в самый низ меню **Файл** (это можно сделать, просто перетащив его мышью в окне установок MainMenu1), отделим его от двух имеющихся пунктов промежутком (т. е. пунктом с заголовком "-") и назовем по-иному: Set1. В компонент Label1 на панели установок введем новый более вразумительный текст, который для компактности запишем в две строки и потому будем его устанавливать динамически, вставив в процедуру по событию onCreate формы Form1 такую строку:

```
Label1.Caption:='Установить интервал для'+#13+'показа слайдов, с';
```

Саму панельку растянем по вертикали, перенесем кнопку **Ok** вниз, установим компонент ColorBox и справа добавим Label2 с заголовком Установить цвет рамки. Для свойства Style компонента ColorBox1 установим в True пункт cbStandartColor, остальные пункты — в False. Как это должно выглядеть в программе, показано на рис. 9.4.

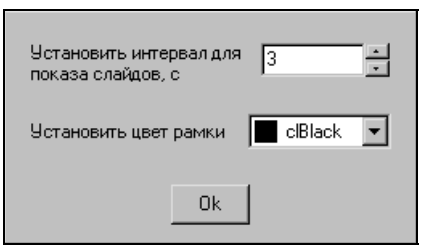


Рис. 9.4. Панель установок для SlideShow (вид в работающей программе)

Заметки на полях

Компонент ColorBox представляет собой типичный пример того, насколько формально можно подойти к разработке компонентов. Этот компонент используется в самой Delphi, в пункте выбора цветов для свойства Color любого компонента. Отличается он, однако, тем, что произвольно формировать список цветов вам не позволят, в крайнем случае можно использовать не очень удобный механизм задания цвета для пункта, определяемого флагом clIncludeDefault в свойстве Style (флаг clSystemColors также должен быть установлен), который сам теряется во множестве других вариантов, а, например, только среди трех-четырех или даже шестнадцати (основных) цветов его включить нельзя, не говоря уж о том, чтобы поставить его, например, в начало или конец списка. Логика примерно та же, как если бы мы в своих установках клавиш для переключения раскладки *(см. главу 7)* распахивали бы полный список всех доступных клавиш, в то время как пользователю нужно всего-то четыре-пять реально удобных вариантов, остальное он может и вручную расстараться ввести при необходимости. Нельзя здесь и поставить нормальные русские названия цветов. Для того чтобы сделать аккуратный компонент для удобного на практике выбора цветов, следует самому формировать список с картинками, либо на основе стандартного ComboBox (там можно, не выводя

квадратиков, например, просто покрасить текст с названием цвета или фон ячейки), либо сделать свой ColorBox на основе ComboBoxEx (вкладка **Win32**). Но наша задача сейчас не в том, чтобы демонстрировать работу со списками, поэтому мы ограничились 16 основными цветами, которых для фона картинки на практике более чем достаточно.

Создадим обработчик события ColorBox1Click и вставим в него пока единственную строку:

```
Panel1.Color:=ColorBox1.Selected;
```

Сделаем еще одно усовершенствование в диалоге-панельке. Чтобы работа с клавиатуры действительно, как и положено, ускоряла процесс, удобно действовать так: пусть при первом запуске фокус ввода устанавливается в Edit1, затем по нажатию <Enter> переносится в ColorBox (которым можно управлять стрелками вниз-вверх), потом по следующему нажатию <Enter> переносится на кнопку **Ok** для выхода из диалога. Даже в нашем случае всего трех позиций, когда, казалось бы, можно обойтись и клавишей <Tab>, это намного удобнее, а в случае наличия большого количества однотипных полей-редакторов переключение именно через <Enter> принципиально облегчает жизнь. Для этого мы отменим ввод значения интервала через нажатие <Enter> на редакторе Edit1 (здесь это уже неактуально), и заменим текст в обработчике события onKeyDown для него на следующий:

```
procedure TForm1.Edit1KeyDown(Sender: TObject; var Key: Word;
            Shift: TShiftState);
begin
{нажатие на Enter после очередного ввода}
if key=vk_Return then
FindNextControl(Sender as TWinControl, true, true,
                false).SetFocus;
end;
```

Объявим для обработчика события onKeyDown компонента ColorBox1 ту же самую процедуру Edit1KeyDown, как мы это делали раньше, и позаботимся о том, чтобы свойства TabOrder всех трех компонентов шли по порядку (у Edit1 должно стоять TabOrder=0). Теперь все будет работать так, как мы задумали.

Проверим наши усовершенствования (при выборе цвета мышью или клавишами-стрелками в ColorBox сразу должен меняться цвет поля вокруг картинки) и перейдем к вопросу о создании и формировании INI-файла. Нам надо запоминать всего три пункта, поэтому секция у нас пусть будет всего одна, назовем ее Main, а сам файл, естественно, SlideShow.ini (если указать просто имя, то INI-файл будет создан, как глобальный, в папке Windows, поэтому его надо указывать с полным путем). В список глобальных переменных добавим переменную IniFile:TIniFile, а в предложение **uses** модулей slide.pas и

papka.pas модуль IniFiles. Обработчик onCreate формы Form1 будет тогда выглядеть так (я сразу устанавливаю и параметры нужных компонентов, считанные из INI-файла, если он уже существовал):

```
procedure TForm1.FormCreate(Sender: TObject);
var x:integer;
begin
 Application.onMinimize := OnMinimizeProc;
 Label1.Caption:=
    'Установить интервал для'+#13+'показа слайдов, с';
 IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini'));
          {если не было - создаем, иначе открываем}
 with IniFile do
 begin
  if SectionExists('Main') then
    {если файл и секция в нем уже есть}
  begin
   Timer1.Interval:=ReadInteger('Main','Timer interval',3000);
           {по умолчанию 3с}
   x:=ReadInteger('Main','Color',0); {по умолчанию цвет 0}
   if x>15 then {вдруг кто-то вручную исправил}
   begin
    x:=0; {исправляем ошибку}
    WriteInteger('Main','Color',0);
   end;
   ColorBox1.Selected:=ColorBox1.Colors[x];
        {устанавливаем цвет}
   Panel1.Color:=ColorBox1.Selected;
   stpath:=ReadString('Main','Path','C:\');
        {по умолчанию диск C:}
   try
    ChDir(stpath); {устанавливаем текущую папку}
   except
    stpath:='C:\';
    ChDir(stpath); {если не получается - диск C:}
   end;
  end else {если секция еще не создана}
  begin
   WriteInteger('Main','Timer interval',Timer1.Interval);
   WriteInteger('Main','Color',0);
    {под номером 0 в списке ColorBox - черный цвет}
   WriteString('Main','Path',stpath);
  end;
```

```
  Destroy;
 end; {IniFile}
end;
```

Обратите внимание, что при установке в качестве текущей папки той, что задана в INI-файле, мы обрабатываем возможную исключительную ситуацию — за время между запусками программы папка могла исчезнуть (например, вытащили диск из CD-ROM).

Функция `ChangeFileExt(ParamStr(0),'.ini')` сразу формирует нам имя INI-файла с полным путем к нему из названия исполняемого файла программы. Для формирования имени существует несколько способов получить папку, где находится программа, и это один из них — в следующей главе мы применим для получения текущей папки функцию `ExtractFileDir(Application.ExeName)`. Если вы хотите организовать глобальный INI-файл в директории Windows, то нужно указать сразу конкретное имя `'SlideShow.ini'` без пути к нему. Кстати, никто не запрещает и просто создать собственную секцию в одном из заведомо существующих INI-файлов (например, win.ini или system.ini, которые имеются во всех версиях Windows).

Теперь добавим в обработчик `Button2Click` в конец следующие строки:

```
. . . . .
IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini')); {открываем
ini}
IniFile.WriteInteger('Main','Timer interval',Timer1.Interval);
IniFile.Destroy;
. . . . .
```

Созданный ранее обработчик события `onClick` для компонента `ColorBox` будет выглядеть так:

```
procedure TForm1.ColorBox1Click(Sender: TObject);
begin
 Panel1.Color:=ColorBox1.Selected;
 IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini'));
                        {открываем ini}
 IniFile.WriteInteger('Main','Color',ColorBox1.ItemIndex);
 IniFile.Destroy;
end;
```

А в модуле papka.pas добавим при его закрытии (перед строкой `Form3.Close`), соответственно:

```
. . . . .
IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini'));
                        {открываем ini}
```

```
IniFile.WriteString('Main','Path',stpath);
IniFile.Destroy;
. . . . .
```

Как видите, получилось лишь немногим сложнее, чем если бы мы создавали файл по собственному усмотрению, зато удобно и единообразно. На самом деле заниматься самодеятельностью имеет большой смысл только, если INI-файл уж очень простой, или если вы хотите скрыть его содержимое от посторонних взоров — в последнем случае не поможет даже переход к расположению параметров в реестре, т. к. опытный человек все равно их найдет. А вот если ваш файл будет содержать невнятный текст типа "$A0 $C2..." или вообще бинарные числа, то вероятность, что его расшифруют, резко уменьшается. Позже мы увидим, как можно создавать текстовые файлы, зашифрованные собственным шифром — правда, для серьезного криптографа расколоть наш шифр скорее всего будет, что рюмку чая проглотить, но 9999 из 10 000 пользователей даже не поймут, как к этому подступиться.

В *главе 11* мы продолжим эту тему, выполнив одно давнее обещание — дадим пользователю выбрать, демонстрировать заставку или нет, а также сворачивать ли приложение в иконку при минимизации и при закрытии.

ГЛАВА 7

Язык мой — враг мой

Резидентный переключатель раскладки

> Вопрос: Что произойдет, если нажать сразу две клавиши Shift?
> Ответ: На экране может произойти увеличение всех символов в два раза. Так не следует поступать, поскольку чрезмерное увеличение размера символов приводит к преждевременному выгоранию экрана.
>
> **forum.winall.ru**

Отметим для начала, что в Windows понятие "раскладка" достаточно размытое, и об этом подробнее мы будем говорить в следующей главе. Здесь мы только установим, что под "раскладкой" будем понимать то, что имелось в виду традиционно — а именно переключение вида отображаемых экранных символов при нажатии клавиш на клавиатуре.

Пользоваться предлагаемыми в Windows вариантами переключения раскладки при хоть сколько-нибудь профессиональной работе просто невозможно — во-первых, есть опасность, что пальцы навсегда останутся скрюченными, во-вторых, если без шуток, то стандартные комбинации клавиш <Alt>+<Shift> и <Ctrl>+<Shift> задействованы и в некоторых других важных операциях. Например, при использовании первой из этих комбинаций всегда есть опасность, что вы отпустите клавишу <Shift> раньше, и тогда по <Alt> сработает пресловутая функция обращения к меню, а вторая комбинация задействована в операциях с выделением текста и вводом специальных символов в Word. Поэтому использовать отдельный переключатель раскладки — просто жизненная необходимость. Лучшая из готовых подобных программ по всеобщему признанию — Punto Switcher, однако, по личному мнению автора, она злостно нарушает принцип "программа не должна делать ничего лишнего" *(см. главу 1)* — слишком много в ней наворочено функций, которые при ближайшем рассмотрении только мешаются. Поэтому мы создадим свою собственную "переключалку", которая только это и будет уметь.

Можно попробовать все сделать совсем просто: во-первых, использовать уже отработанный механизм горячих клавиш (можно даже не создавать новое приложение — просто зарегистрировать еще одну горячую клавишу в программе Layout из *главы 5*, присвоив ей номер 2) и по ее нажатию осуществлять процедуру, состоящую всего из одной строки:

```
ActivateKeyboardLayout(0,HKL_NEXT);
```

Эта функция будет переключать раскладку циклически. Можно усложнить дело, и переключать с помощью попеременных вызовов функции `LoadKeyboardLayout(<идентификатор языка>, KLF_ACTIVATE)`, подставляя в качестве идентификатора каждый раз либо `'00000409'` , либо `'00000419'` — так, как мы делали в примере из *главы 5*. Результат будет один — в Windows XP это работать не будет. Более того, оно не должно работать и в Windows 98, т. к. в "пособии" — т. е. на том же сайте MSDN — про обе эти функции ясно написано "This function only affects the layout for the current process or thread" (Эта функция воздействует только на раскладку клавиатуры текущего процесса или потока). Просто реализация данных функций в Windows XP и 98 разная: в первой текущим процессом, очевидно, считается тот, что вызывает функцию, а во второй — тот, что в данный момент активен (это отчасти логично — вспомните про различную реализацию многозадачности в данных системах).

Так что если вам потребуется переключать раскладку для текущего приложения, можно использовать указанный способ, он является общепринятым и официально рекомендованным к использованию. А для резидентного переключателя мы обратимся к первоисточнику, чтобы попытаться дополнительно понять, что происходит, и в какую сторону нам надо направить свои мысли. Открываем сайт MSDN Library [14] и на странице "About Keyboard Input" под заголовком "Languages, Locales, and Keyboard Layouts" читаем (привожу сразу в переводе на русский):

"Обычно не предполагается, что прикладные программы будут непосредственно управлять языками ввода. Вместо этого пользователь устанавливает комбинацию языка и раскладки, а затем переключается между ними. Когда пользователь щелкает мышью по тексту, имеющему знаки из другого языка, прикладная программа вызывает функцию ActivateKeyboardLayout, чтобы активизировать заданную по умолчанию раскладку пользователя для этого языка. Если пользователь редактирует текст на языке, которого нет в действующем списке, прикладная программа может вызывать функцию LoadKeyboardLayout с [этим] *языком, чтобы получить раскладку, базирующуюся на этом языке."*

Как видите, нам не доверяют управлять языком — вдруг мы еще какой-нибудь не тот язык включим в свою программу и осмелимся заговорить на

суахили без санкции от Microsoft. С другой стороны, отсюда же следует, что в принципе пользовательские приложения в Windows могут устанавливать свою комбинацию языка и раскладки. Но реализовано это нагромождение понятий настолько запутанно, очень часто даже фирменные приложения выкидывают весьма забавные коленца *(примеры см. в следующей главе)*. Но нам это не надо, зато нужно переключать раскладку для чужого приложения — а этого, как мы видим, "не предполагается". Так что придется опять "нажимать мышью на <Reset>" и имитировать нажатие клавиш переключения раскладки, зарегистрированных в системе — слава богу, надежный способ мы уже знаем!

Надумав такое, отвлечемся от собственно переключения: а так ли хорошо здесь использовать механизм горячих клавиш? Для того чтобы переключать раскладку, нам хотелось бы использовать какую-нибудь заведомо неиспользуемую в других программах клавишу, т. е. как раз здесь удобно выбрать либо из правых <Alt>, <Ctrl>, <Shift> или <Enter>, либо из почти неиспользуемых клавиш типа <Scroll Lock> или <Pause>. Механизм горячих клавиш в принципе не позволяет зарегистрировать в их качестве клавиши-модификаторы, но даже если и удастся (есть сведения, что это можно обойти вот таким хитрым способом: `RegisterHotKey(FHandle, 1, MOD_CONTROL, 0)`, нам это мало поможет: правый-левый <Ctrl> тут все равно не отличишь.

Самый простой переключатель раскладки

Перенесем наш проект из папки Glava6\2 под новым именем Keyswitch в другую папку (Glava7\1) и туда же перенесем проект DLL, также под новым именем KeyHook. Не забудьте, что при переносе надо отредактировать DSK-файлы для двух проектов (и для ловушки, и для основной программы).

Заменим главную иконку приложения в ресурсном файле Keyswitch.res на другую подходящую (в папке с проектом это letters.ico) все тем же способом *(см. главу 5)*. Для того чтобы перейти теперь к созданию непосредственно "переключалки", нам сначала понадобится заголовок окна программы, который у нас до сих пор оставался пустым — см. процедуру `CreateWindowEx`. Вставим в нее, наконец, настоящее имя окна "KeySwitch" (третий параметр). Так как мы создаем не учебный пример и не рабочую утилитку, а настоящее приложение всерьез и надолго, то озаботимся тем, чтобы оно не запускалось второй раз. Для этого внесем наш рабочий дескриптор `dWin` в список переменных (можно через запятую перед `Instance`), добавим туда же переменные `st:string;` и `pbuff:array[0..127] of char;` и сразу после `CreateWindowEx` вставим нашу процедуру предотвращения повторного запуска:

```
. . . . .
FHandle:=CreateWindowEx (0,'DX','KeySwitch',WS_POPUP, 5,5, 200,
200,0,0,instance, nil); {получаем дескриптор окна}
dWin:=FHandle;
while dWin <> 0 do
begin
 if (dWin <> FHandle) and {собственное окно игнорируем}
  (GetWindow(dWin, GW_OWNER) = 0) and
        {дочерние окна игнорируем}
  (GetWindowText(dWin, pbuff, sizeof(pbuff)) <> 0)
        {без названия игнорируем}
 then
 begin
   GetWindowText(dWin, pbuff, sizeof(pbuff));
       {получаем текст названия приложения}
   st:=string(pbuff); {переводим его в строку}
   if st='KeySwitch' then exit; {прерываем программу}
 end;
  dWin:= GetWindow(dWin, GW_HWNDNEXT);
       {ищем следующее приложение из списка}
end;
. . . . .
```

Теперь вернемся к перехватчику — нам нужно отработать процедуру передачи события перехваченной клавиши вызывающей программе, ради чего все и затевалось. Для этого надо как-то определять окно, куда идет передача. Самый грамотный путь — воспользоваться уже упоминавшимся в *главе 4* механизмом отображения файлов на память (memory mapped files) — в этом случае программы (ловушка и собственно переключатель) получают общий кусок памяти, через который все и передается. А нельзя ли попроще? Ведь пока нам никаких особенных параметров передавать не надо, достаточно только зафиксировать сам факт события. Мы знаем уже, что есть такая функция SendMessage, и она должна что-то передавать окну по известному дескриптору. У нас в *главе 6* это более-менее работало, но в таком случае поиск приемного окна по дескриптору и даже прямая передача дескриптора ловушке при запуске не сработают — событие передастся только, если окно вызывающей программы активно (или ловушка локальная). А у нас оно не только не активно, но еще и спрятано в иконку. Так что вместо того, чтобы пытаться передать дескриптор, мы поступим примерно так, как в процедуре предотвращения повторного запуска — найдем дескриптор окна программы по его имени. Правда, если мы захотим позднее сделать выбор горячей клавиши при запуске программы, то механизм передачи параметров через memory mapped files так или иначе использовать придется. Но пока не будем на этом сосредо-

тачиваться, а попробуем применить самое простое и к тому же уже отработанное решение — как мы увидим в дальнейшем, оно вполне работоспособно.

А какое сообщение передавать? Вставим и в текст ловушки, и в текст самого перехватчика в секции объявлений переменных такую константу: `HookMsg =wm_User+$111;`.

Тут мы число выбрали "от фонаря" — в надежде, что никому другому такое в голову и не придет. Для единообразия я заменил и `Ico_Message` (хотя это необязательно), а потом поместил их обе в секцию констант. Теперь будет всего три константы:

```
const
 idmEXIT=1; {пункт всплывающего меню "Закрыть"}
 Ico_Message = wm_User+$110; {сообщение от иконки окну}
 HookMsg =wm_User+$111; {сообщение от ловушки}
```

Далее, когда мы будем писать программу для предоставления ее во всеобщее пользование, я покажу, как зарегистрировать уникальное сообщение через функцию `RegisterWindowMessage` (получив перед этим для него уникальный номер через <Ctrl>+<Shift>+<G>).

Теперь перейдем, наконец, к ловушке. Пусть она пока будет реагировать, к примеру, на правый (цифровой) <Enter>. Вычистим из DLL все упоминания о записи в файл — она нам больше не понадобится (а в проекте Keyswitch это место можно пока временно закомментировать — запись в файл нам потом понадобится для другой цели). Исходный текст DLL должен в результате быть таким — листинг 7.1.

Листинг 7.1. Модифицированная библиотека для "шпионской" ловушки

```
library Keyhook;

uses
Messages,
Windows;

var
HookHandle:hHook;

function KeyboardProc(Code: Integer; wParam: wParam; LParam:
         LParam): integer; stdcall;
begin
if code<0 then Result:=CallNextHookEx(HookHandle, code, WParam,
                  LParam)
```

```
else
begin
 if byte(LParam shr 24)<$80 then {только нажатие}
 begin
  Result:=CallNextHookEx(HookHandle, code, WParam, LParam);
  if byte (LParam shr 16)= $1C then {клавиша Enter}
  if byte(LParam shr 24)=1 then {конкретно - дополнительная}
  begin
   SendMessage(FindWindow('DX','KeySwitch'), HookMsg, WParam,
         LParam);
  end;
 end;
end;
end;

procedure SetHook; stdcall; {установка ловушки}
begin
 HookHandle := SetWindowsHookEx(WH_KEYBOARD,KeyboardProc,
               hInstance, 0);
end;

procedure DelHook; stdcall; {удаление ловушки}
begin
 UnhookWindowsHookEx(HookHandle);
end;

exports
SetHook, DelHook;
begin
end.
```

Не забудем заменить и ссылки на DLL в объявлении процедур `SetHook` и `DelHook` в начале программы на `Keyhook.dll` и в процедуре `CreateMyicon` всплывающую подсказку на `KeySwitch`.

Теперь обратимся к программе и подумаем — а что мы, собственно, будем делать по событию `HookMsg`? Однозначно — имитировать нажатие клавиш переключения раскладки, но ведь здесь может быть два варианта: <Ctrl>+<Shift> или <Alt>+<Shift>. Какой именно установлен в системе, можно узнать из реестра, но надо ли? Давайте просто сымитируем оба варианта — при этом мы ничего не нарушим, т. к. какой-нибудь сработает, а второй просто пропадет впустую. Итак, пишем такую процедуру (текст нужно расположить до обработчика сообщений `WindowProc`):

```
procedure LayoutChange;
begin
  keybd_event(VK_SHIFT,$2a,0,0); {<Ctrl>+<Shift>}
  keybd_event(VK_CONTROL,$1d,0,0);
  keybd_event(VK_CONTROL,$1d,KEYEVENTF_KEYUP,0);
  keybd_event(VK_SHIFT,$2a,KEYEVENTF_KEYUP,0);

  keybd_event(VK_MENU,$38,0,0); {<Alt>+<Shift>}
  keybd_event(VK_SHIFT,$2a,0,0);
  keybd_event(VK_SHIFT,$2a,KEYEVENTF_KEYUP,0);
  keybd_event(VK_MENU,$38,KEYEVENTF_KEYUP,0);
end;
```

Теперь вставляем обработчик нашего сообщения в конец процедуры `WindowProc`:

```
if Msg = HookMsg then LayoutChange;
```

Обратите внимание на один момент — ранее мы утверждали (не как-нибудь, а руководствуясь официальной информацией с сайта MSDN), что второй параметр функции `keybd_event`, который должен представлять собой скан-код, все равно игнорируется, и потому может быть проставлена любая величина. И все работало именно так! А вот в данном конкретном случае, если вы немного поэкспериментируете, то обнаружите, что под Windows XP все по-прежнему работает в любом виде, а вот под Windows 98 — нет! Приходится вводить настоящий скан-код. Вот они, недокументированные особенности Windows во всей красе!

Прежде чем пробовать эту программу, удалите из системы другой подобный переключатель раскладки, если он у вас установлен — почти 100% гарантии, что он устроен аналогичным образом, и тогда конфликты неизбежны. А теперь запустите программу (предварительно, естественно, откомпилировав ловушку Keyhook) и проверьте — по правой клавише <Enter> раскладка переключается, по обычной — нет. Но! Клавиша <Enter>, кроме всего прочего, потому и называется так (ввод), что еще и делает много других действий: переводит строку в текстовых редакторах, запускает программы и т. п. И ведь точно так же будет со всеми остальными клавишами, это только <Ctrl> и <Shift> ничего сами по себе не делают, но нельзя же ограничиться таким скудным выбором! К счастью, выход из этой ситуации имеется — в случае, если нажата "наша" клавиша, надо просто "не пускать" это событие дальше нашей ловушки. Это очень просто: нужно, чтобы функция `KeyboardProc` возвращала значение, отличное от 0. Перепишем этот фрагмент в тексте ловушки Keyhook так:

```
function KeyboardProc(Code: Integer; wParam: wParam; lParam: lParam):
integer; stdcall;
begin
if code<0 then Result:=CallNextHookEx(HookHandle, code, WParam, LParam)
else
begin
 if byte(LParam shr 24)<$80 then {только нажатие}
 begin
  Result:=CallNextHookEx(HookHandle, code, WParam, LParam);
  if byte (LParam shr 16)= $1C then {клавиша Enter}
  if byte(LParam shr 24)=1 then {конкретно - дополнительная}
  begin
   SendMessage(FindWindow('DX','KeySwitch'),
           HookMsg, WParam, LParam);
   Result:=1;
  end;
 end;
end;
end;
```

Теперь обычная клавиша <Enter> будет работать по-прежнему, а цифровая — только переключать раскладку. Проверьте и убедитесь, что со всеми клавишами дело обстоит точно так же, за одним исключением: те клавиши, обработка которых ведется на уровне BIOS, все равно отключить не удастся, т. е. <Caps Lock>, к примеру, по-прежнему будет переключать регистр. Этим фактом мы можем воспользоваться по-своему и дополнительно ко всему осуществить одну красивую операцию — отключить <Alt> как способ входа в меню. Для отключения этой ненавистной функции достаточно добавить к тексту KeyboardProc одну-единственную строку:

```
if byte (LParam shr 16)= $38 then Result:=1;
```

$38 — общий скан-код для обеих клавиш <Alt>. Произойдет следующее: системная функция <Alt>, как модификатора, будет работать, как ни в чем не бывало, а функция входа в меню исчезнет. Однако не все, конечно, разделяют мою ненависть к этой особенности Windows[1], поэтому в данном варианте можете включить эту возможность в программу по желанию, а позднее, когда мы будем делать расширенный вариант переключателя, придется ее сделать опциональной — т. е. устанавливаемой по желанию пользователя. К сожалению, так же просто отключить функции системных клавиш Windows (тех, что

[1] К тому же клавиша <Alt> перестанет работать и в других программах, если к ней привязана несистемная функциональность — например, в клонах Norton Commander это переключение нижнего меню.

для вызова меню **Пуск** и контекстного), чтобы их использовать в своей программе, не удается.

Попробуем и убедимся, что все работает (могут наблюдаться "глюки" в Office XP, но далее мы укажем, как от этого недостатка избавиться). Простейший переключатель готов! Заметим, что если вы захотите на этой основе сделать, например, просто программу выключения функции <Alt>, а раскладку переключать другой программой, то они могут конфликтовать. Конфликта можно избежать очень просто: надо сделать так, чтобы наша программа запускалась раньше фирменной "переключалки", тогда все сообщения дойдут куда надо: чем раньше запущена ловушка, тем позже до нее доходит сообщение в очереди. Осуществить это несложно — надо включить нашу программу в автозапуск через реестр (как это делается, мы увидим в дальнейшем), а чужую — через папку Автозапуск, тогда порядок будет наверняка именно такой, какой надо. Если же включать в автозапуск программу целиком, то другую "переключалку" придется удалить (да и зачем она нужна?), но в некоторых случаях могут быть иные конфликты.

Уж больно переключатель получился примитивный: во-первых, не позволяет пользователю выбирать по своему усмотрению горячую клавишу для переключения. Во-вторых, иконка, с помощью которой Windows сообщает нам о действующей раскладке, довольно невыразительная — в нее надо долго вглядываться, чтобы понять, какой язык установлен, и неплохо бы ее заменить, например, на красную и синюю, которые будут меняться в зависимости от раскладки. А это повлечет за собой целый ворох проблем: для начала нужно обеспечить двустороннюю связь между системным переключателем и нашим (вовсе отключить системный мы не можем, т. к. работаем через него). Иначе, если кто-то переключит раскладку по привычке двойными клавишами или мышью, у нас ничего не переключится. Еще сложнее обеспечить автоматическое распознавание текущей раскладки активного окна с тем, чтобы она переключалась синхронно с системной при переходе от одного приложения к другому.

Заметки на полях

Не откажем себе в удовольствии отвлечься, чтобы очередной раз попинать Microsoft: вполне можно было бы и не делать свою раскладку для каждого приложения, а оставить общую раскладку для всей системы или хотя бы предоставить пользователю выбор. Одна текущая раскладка на все была бы даже лучше — переключаясь между приложениями, сейчас нужно все время поглядывать на индикатор, что очень отвлекает. В противном случае это откладывалось бы в голове и никакой индикатор вообще бы не понадобился (еще в Windows 3.*x* автор этих строк никогда индикатором не пользовался). А если уж делать, то довести идею до ума, и предоставить возможность *запоминания* раскладки по умолчанию для каждого приложения отдельно.

Займемся сначала второй задачей. Ясно, что придется переписать DLL, и настолько существенно, что мы перенесем наш проект из папки Glava7\1 в новую папку (Glava7\2), дабы не потерять достигнутое. Так как простейшая программа тоже вполне работоспособна, то проект я переименовал в LangSwitch, а проект DLL назвал Langhook.dll. Заголовок окна и всплывающую подсказку мы также заменим, но во избежание конфликтов следует помнить, что одновременно запускать эти "переключалки" нельзя.

Переключатель с заменой системной иконки — промежуточный вариант

Начнем с иконок. Главную иконку приложения заменим на Keyboard.ico (находится в папке с проектом). Для того чтобы добавить свои иконки, нам придется несколько забежать вперед и включить их в ресурсы приложения (о ресурсах мы подробнее поговорим в следующих главах).

Сначала создадим две иконки размером 16×16 (для Tray этого более чем достаточно), одну красного цвета с надписью "Ru", вторую — синего с надписью "En". Я их создавал с помощью упоминавшегося в *главе 4* редактора LiquidIcon, но можно использовать и Image Editor, тем более что он нам все равно понадобится. После создания иконок я их не компилировал в ресурс (RES-файл) через LiquidIcon, а оставил, как есть (в каталоге проекта они называются Rus.ico и Eng.ico). Затем открываем Image Editor, создаем новый файл ресурсов (**File | New | Resource file**) и называем его Icon.res. В нем создаем две иконки с названиями "MAINICONRUS" и "MAINICONENG". Названия обязательно писать заглавными буквами (по крайней мере, все пособия это рекомендуют), а не так, как это делает Image Editor по умолчанию. Напоминаю, что все изменения в ресурсы нужно вносить при закрытой Delphi (правда, от всех "глюков" Image Editor вы все равно не гарантированы).

Дальше откроем созданные иконки (**File | Open | Icons**) и просто переносим изображение методом Select All-Copy-Paste поверх заготовок иконок в ресурсном файле (чтобы открыть окно с заготовкой, надо дважды щелкнуть по названию в перечне ресурсов — см. рис. 5.1). Разумеется, можно было бы нарисовать иконку и прямо в ресурсном файле, используя заготовку, просто в редакторе LiquidIcon работать намного удобнее, чем в Image Editor. В данном случае я сделал иконки круглыми (темное поле в углах означает "прозрачный" цвет).

Теперь созданный ресурс надо, во-первых, присоединить к проекту, во-вторых, извлечь из него полученные иконки. Для первого мы просто в тексте

главной программы LangSwitch.dpr к имеющейся строке {$R *.res} добавим ниже еще одну: {$R Icon.res}. Наконец, заменим оператор

```
HIcon1:=CopyIcon(Application.Icon.Handle)
```

в функции CreateMyicon на следующий текст:

```
. . . . .
Instance :=GetModuleHandle(nil); {получаем дескриптор модуля}
if word(GetKeyboardLayout(0)) = $419 {если язык по умолчанию
                  русский}
then HIcon1:=LoadIcon(instance,'MAINICONRUS')
       {получаем дескриптор русской иконки}
else HIcon1:=LoadIcon(instance,'MAINICONENG');
       {получаем дескриптор английской иконки}
. . . . .
```

Теперь программа при запуске будет сворачиваться в одну из иконок: русскую или английскую. Функция GetKeyboardLayout возвращает, как видим, сразу идентификатор языка, а не его название типа используемой в LoadKeyboardLayout строки (идентификатор — только часть названия, см. *главу 5*). Параметр 0 в вызове функции означает раскладку для текущего процесса — т. е. эта функция возвращает либо ту раскладку, которая установлена в используемом файловом менеджере, откуда запускается программа, либо, если загрузка осуществляется через Автозапуск, ту раскладку, которая установлена в системе по умолчанию (в среде Delphi перед пробным запуском нужно устанавливать именно тот язык, что по умолчанию, иначе будет путаница). Использовать в данном случае функцию GetKeyboardLayoutName, которая возвращает название системной раскладки по умолчанию, было бы идеологически неправильно. Проверьте работу программы в этой части, сменив раскладку по умолчанию несколько раз.

Так, теперь ловушки. Нам их понадобится целых три: одна, как и раньше, будет отслеживать клавишу, вторая — системное сообщение о том, что раскладка переключилась, и третья — о смене активного окна с его собственной раскладкой. Теоретически все указанное должно бы делаться через одну ловушку типа WH_GETMESSAGE (см. описание hook в [17]), но на практике по ряду причин все будет надежно работать, только если сделать три специализированные: первую, как раньше, типа WH_KEYBOARD, вторую — WH_GETMESSAGE, и третью — WH_CBT. Продвинутые читатели немедленно заметят, что, по крайней мере, нажатие клавиш можно также отслеживать через ловушку WH_GETMESSAGE, но я их спешу успокоить — попробуйте сами, и вы убедитесь, что отслеживаются далеко не все нажатия, и, видимо, это так и должно быть (процедура GetMsgProc обрабатывает только оконные сообщения, системные

сообщения ею не обрабатываются [14]). Полный текст DLL приведен в листинге 7.2.

Листинг 7.2. Библиотека для "шпионской" ловушки с учетом раскладки клавиатуры

```
library Langhook;

uses
Messages,
Windows;

const
HookMsgKey =wm_User+$111; {сообщение о нажатии клавиши}
HookMsgLang1 =wm_User+$112; {язык - английский}
HookMsgLang2 =wm_User+$113; {язык - русский}

var
HookHandleKey,HookHandleLang,HookHandleWin:hHook;

function KeyboardProc(Code: Integer; wParam: wParam; lParam:
          lParam): integer; stdcall;
begin  {перехват нажатия клавиатуры}
if code<0 then Result:=CallNextHookEx(HookHandleKey, code, WParam,
                 LParam)
else
begin
 Result:=CallNextHookEx(HookHandleKey, code, WParam, LParam);
 if byte(LParam shr 24)<$80 then {только нажатие}
 begin
  if byte (LParam shr 16)= $1D then {клавиша RControl}
  if byte(LParam shr 24)=1 then {конкретно - дополнительная}
  begin
   SendMessage(FindWindow('DX','LangSwitch'), HookMsgKey,
         WParam, LParam);
   Result:=1;
  end;
 end;
 if byte (LParam shr 16)= $38 then Result:=1;
       {отключение Alt, как входа в меню}
end;
end;

function LangProc(Code: Integer; wParam: wParam; lParam: lParam):
integer; stdcall;
```

```
begin   {перехват системного сообщения}
if code<0 then Result:=CallNextHookEx(HookHandleLang, code, wParam,
                  LParam)
 else
 begin
  Result:=CallNextHookEx(HookHandleLang, code, wParam, LParam);
   {запрос на изменение языка:}
  if (TMsg(Pointer(LParam)^).message = wm_INPUTLANGCHANGEREQUEST)
  then
  begin
   {если раскладка русская:}
   if word(TMsg(Pointer(LParam)^).lParam)=$419 then
   SendMessage(FindWindow('DX','LangSwitch'), HookMsgLang2,
         WParam, LParam);
   {если раскладка английская:}
   if word(TMsg(Pointer(LParam)^).lParam)=$409 then
   SendMessage(FindWindow('DX','LangSwitch'), HookMsgLang1,
         WParam, LParam);
  end;
 end;
end;

function WinProc(Code: Integer; wParam: wParam; lParam: lParam):
integer; stdcall;
begin {перехват перехода фокуса ввода}
if code<0 then Result:=CallNextHookEx(HookHandleWin, code, wParam,
                  LParam)
 else
 begin
  Result:=0;
  CallNextHookEx(HookHandleWin, code, wParam, LParam);
  if code = HCBT_SETFOCUS then
  {если окно собирается получить фокус ввода}
  if (wParam<>FindWindow('DX','LangSwitch'))
  then        {кроме окна LangSwitch}
  begin
   {если раскладка русская:}
   if
    word(GetKeyboardLayout
         (GetWindowThreadProcessId(wParam,nil)))=$419
   then
   SendMessage(FindWindow('DX','LangSwitch'), HookMsgLang2,
           WParam, LParam);
```

```
   {если раскладка английская:}
   if
    word(GetKeyboardLayout
         (GetWindowThreadProcessId(wParam,nil)))=$409
   then
   SendMessage(FindWindow('DX','LangSwitch'), HookMsgLang1,
           WParam, LParam);
  end;
 end;
end;

procedure SetHook; stdcall; {установка ловушек}
begin
 HookHandleKey := SetWindowsHookEx(WH_KEYBOARD,KeyboardProc,
         hInstance, 0);
 HookHandleLang := SetWindowsHookEx(WH_GETMESSAGE,LangProc,
         hInstance, 0);
 HookHandleWin := SetWindowsHookEx(WH_CBT,WinProc,hInstance, 0);
end;

procedure DelHook; stdcall; {удаление ловушек}
begin
 UnhookWindowsHookEx(HookHandleLang);
 UnhookWindowsHookEx(HookHandleKey);
 UnhookWindowsHookEx(HookHandleWin);
end;

exports
SetHook,
DelHook;
begin
end.
```

Заметьте, что клавиша для переключения раскладки заменена на правый <Ctrl> по той причине, что я сам пользуюсь программой при написании текста этой главы, а ранее установленный окончательный вариант (с пользовательской установкой клавиши, см. далее) пришлось, естественно, из системы удалить, иначе было бы невозможно проверять описываемые. Ну, а сам я привык пользоваться именно <Ctrl>.

Все подробности расписывать долго: в ловушке `WH_GETMESSAGE` (в процедуре `LangProc`) отслеживается событие `wm_INPUTLANGCHANGEREQUEST` (запрос на переключение языка), определяется, какой язык собирается быть загруженным, и посылается соответствующее сообщение нашей программе. А в ловушке `WH_CBT` отслеживается переход фокуса ввода от окна к окну и определяется,

какая раскладка действует в том окне, которое этот фокус получит. Характеристики ловушек, типы процедур, которые выполняются, и что означают в каждом случае параметры wParam и lParam, вы можете посмотреть в [17, 18], а также, разумеется, на официальном сайте [16].

Теперь осталось внести изменения в основную программу LangSwitch.dpr. Сначала объявим константы, те же, что и в ловушке:

```
. . . . .
const
 idmEXIT=1; {пункт всплывающего меню "Закрыть"}
 Ico_Message = wm_User+$110; {сообщение от иконки окну}
 HookMsgKey = wm_User+$111; {сообщение о нажатии клавиши}
 HookMsgLang1 = wm_User+$112; {язык - английский}
 HookMsgLang2 = wm_User+$113; {язык - русский}
. . . . .
```

Процедуры по обработке этих сообщений будут выглядеть так (вставить их надо, естественно, внутрь процедуры WindowProc):

```
. . . . .
if Msg = HookMsgKey then LayoutChange; {переключаем раскладку
                по сообщению о нажатии клавиши}
if Msg = HookMsgLang1 then {переключение при смене фокуса ввода}
begin
  HIcon1:=LoadIcon(instance,'MAINICONENG');
       {получаем дескриптор английской иконки}
  noIconData.hIcon:=HIcon1;
  Shell_NotifyIcon(NIM_Modify,@noIconData); {поменяли иконку}
end;
if Msg = HookMsgLang2 then {переключение при смене фокуса ввода}
begin
  HIcon1:=LoadIcon(instance,'MAINICONRUS');
       {получаем дескриптор русской иконки}
  noIconData.hIcon:=HIcon1;
  Shell_NotifyIcon(NIM_Modify,@noIconData); {поменяли иконку}
end;
. . . . .
```

Безупречной работы от нашей простой программы во всех случаях ожидать не следует. В Windows 98 не всегда отслеживается переключение в диалоговых окнах; в XP с этим все в порядке, зато наблюдаются "глюки" в Office XP с наиболее подходящими для нашей цели расширенными системными клавишами (правый <Ctrl>, правый <Enter>) — с обычными клавишами все отлично работает, а вне Office нормально работает и с указанными. Этот недос-

таток преодолеть очень просто — надо зайти в **Панель управления**, запустить сервис **Языки и региональные стандарты**, и на вкладке **Языки | Подробнее | Параметры клавиатуры | Смена сочетания клавиш** снять галочку в пункте **Переключать раскладки клавиатуры** (оставив отмеченным пункт **Переключать языки ввода**)[2]. Тогда везде все заработает нормально, причем другой очевидный, казалось бы, путь решения проблемы — выбрать для имитации из двух возможных сочетаний клавиш только одно, реально установленное в системе для переключения языка — к желаемому результату не приводит. О том, что именно вы потеряете с отключением этого пункта, мы еще будем говорить в следующей главе.

Кроме того, в среде Windows 98 наблюдались конфликты с некоторыми автоматически запускаемыми сервисами (у автора — с файрволом Sygate), однако это происходило только, если включить программу в автозапуск, и тем самым она окажется в очереди сообщений позже указанных сервисов. Программа надежно работает, если она запускалась последней, однако "в лоб" через автозапуск это реализовать не удастся, все системные сервисы здесь запускаются в самую последнюю очередь[3]. А вот в XP очередь сообщений работает иначе — но не будем в это углубляться, способы обойти проблему у нас есть, а заниматься отладкой можно до бесконечности, чтобы убедиться в этом, достаточно заглянуть в историю версий любой подобной программы, хоть Punto, хоть RusLat. Кстати, казус с конфликтами ловушек встречается даже в фирменных приложениях — как пример можно привести конфликт очень удобного в использовании переводчика Socrat от фирмы "Арсенал" с программами, написанными на Delphi.

Учитывая указанные обстоятельства, придется обязательно реализовать установки, какую именно клавишу использовать и заодно стоит ли выключать системную функцию <Alt> для входа в меню. На выбранном нами пути реализовать их не получится. Так что ловушки мы отладили, а теперь приступим к изучению отображения файлов на память (memory mapped files) .

Заметки на полях

Почему глобальные ловушки не позволяют прямо передавать параметры в программу и обратно? Дело в особенностях функционирования динамически загружаемых библиотек (DLL) в Windows. Код установленной нами ловушки отображается в память всех процессов в программе, но только код — а не данные!

---

[2] Тут же целесообразно изменить сочетание клавиш на <Alt>+<Shift> — если вы используете свой переключатель, то сочетание <Ctrl>+<Shift> будет очень мешать работе с выделениями, неразрывными пробелами, дефисами и т. п. — в устранении этого недостатка как раз и заключается основной смысл введения отдельного переключателя раскладки.

[3] Обходной путь решения проблемы — привлечение софта, позволяющего регулировать порядок автозапуска (Start Group Optimize, Heliesoft ActiveStartup, Startup Organizer и т. п.).

Поэтому параметры и будут действительны только для передавшей их программы, а ее нам как раз и не надо отслеживать. Механизм отображения файлов на память позволяет получить общее пространство данных для всех отображаемых процессов, и тогда передача будет работать. Есть и другие способы передачи данных между процессами, например, через DDE, через сообщение `WM_COPYDATA`, через так называемые "атомы", но в случае ловушек традиционно используются именно memory mapped files. Подробнее об этом см. [19, 20]. "Маппинг" нам еще понадобится в дальнейшем для другой цели — ускоренного чтения дисковых файлов *(см. главу 14)*.

Переключатель с установками

На всякий случай перенесем проект LangSwitch из папки Glava7\2 в новую папку (Glava7\3). Для того чтобы ввести механизм передачи параметров, проделаем следующее. Начнем с создания общей разделяемой области. Для этого добавим к проекту основной программы LangSwitch, как это советует автор [17], отдельный модуль IniHook (**File | New | Unit**) с секциями **`initialization`** и **`finalization`**, определим в нем нужные структуры, перенесем в него в том числе и определение наших сообщений. Здесь мы не будем присваивать им выдуманные из головы значения, а используем процедуру регистрации `RegisterWindowMessage` с присвоением уникального (в вероятностном смысле) номера. Когда сообщение действует внутри одного оконного класса, делать это необязательно, но у нас оно посылается от DLL конкретному окну, и есть опасность, что кто-то еще пошлет такое же сообщение. Для того чтобы получить уникальный номер, есть два пути. Первый заключается в том, чтобы использовать встроенный генератор случайных чисел Delphi, который вызывается клавишами <Ctrl>+<Shift>+<G>, в результате чего генерируется уникальная строка символов, хорошо знакомая по внешнему виду всем, кто занимался поиском по реестру (так еще получают уникальные имена файлов и классов для регистрации в системе). Второй путь — вызов функции `CreateGUID`, которая делает то же самое, но каждый раз заново при запуске приложения. Второй путь несколько более надежен (им часто пользуются авторы вирусов и троянов), но в данном случае его применить нельзя, т. к. нам нужно сообщить полученные номера ловушкам, а DLL компилируется отдельно. Так что мы применим первый способ, и имейте в виду, что сначала нужно компилировать проект программы вместе с модулем IniHook, а уж затем — проект DLL, именно в таком порядке. Сам модуль будет выглядеть так — листинг 7.3.

Листинг 7.3. Модуль IniHook

```
unit IniHook;

interface
```

```
uses Windows,Messages;

type
PHookInfo = ^THookInfo;
THookInfo = packed record
   FormHandle: THandle; {дескриптор окна приложения}
   HookHandleKey: THandle; {дескрипторы ловушек}
   HookHandleLang: THandle;
   HookHandleWin: THandle;
   Key:byte; {скан-код}
   KeyExt:byte; {расширенная=1, иначе 0}
   Alt:boolean; {отключать/не отключать}
 end;

var
DataArea: PHookInfo = NIL;
hMapArea: THandle = 0;
HookMsgKey,HookMsgLang1,HookMsgLang2:integer;

implementation

initialization
{создаем файл в памяти:}
hMapArea := CreateFileMapping($FFFFFFFF, NIL, PAGE_READWRITE, 0,
              SizeOf(DataArea), 'KeyWinLangHook');
DataArea := MapViewOfFile(hMapArea, FILE_MAP_ALL_ACCESS, 0, 0, 0);

{регистрируем свои уникальные сообщения}
HookMsgKey:=RegisterWindowMessage('{296EDA43-8114-11D9-BF43-
                  444553540000}');
HookMsgLang1:=RegisterWindowMessage('{D4EF4AA0-81C6-11D9-BF43-
                   444553540000}');
HookMsgLang2:=RegisterWindowMessage('{BE5421C0-81C7-11D9-BF43-
                   444553540000}');

finalization
{убираем файл из памяти}
if Assigned(DataArea) then UnmapViewOfFile(DataArea);
if hMapArea <> 0 then CloseHandle(hMapArea);
end.
```

Чтобы сгенерировать три уникальных строки через встроенный генератор Delphi, нужно выждать между нажатиями <Ctrl>+<Shift>+<G> некоторое время (генератор основан на работе системного таймера). Однако в принципе это необязательно, эксперименты показали, что функция `RegisterWindowMessage`

устроена так, что строки могут быть и одинаковыми, а идентификаторы сообщений при этом различаются, и для них сгенерируются все равно разные значения. А вот для сообщения от иконки (Ico_**Message**) в основной программе проводить подобную операцию совершенно ни к чему, т. к. она зарегистрирована в пределах одного приложения. Ее сообщение никуда дальше не пойдет, и может быть любым в пределах от wm_USER до $7FFF.

Поля Key и KeyExt типа Byte в записи, на которую указывает переменная DataArea, будут идентифицировать клавишу, а поле Alt (булевского типа) сигнализировать, нужно ли отключать системную функцию клавиши <Alt>. Ссылка на модуль IniHook автоматически вставится в тексте LangSwitch.pas. А в предложении **uses** ловушки тоже надо сослаться на этот модуль, секцию же объявления переменных и констант удалить вообще.

Теперь изменим процедуру SetHook в ловушке:

```
procedure SetHook; stdcall; {установка ловушек}
begin
 DataArea^.HookHandleKey :=
  SetWindowsHookEx(WH_KEYBOARD,KeyboardProc, hInstance, 0);
 DataArea^.HookHandleLang :=
  SetWindowsHookEx(WH_GETMESSAGE,LangProc, hInstance, 0);
 DataArea^.HookHandleWin :=
  SetWindowsHookEx(WH_CBT,WinProc,hInstance, 0);
end;
```

Во всех функциях надо заменить вызов FindWindow на DataArea^.FormHandle, а дескрипторы отдельных ловушек на соответствующие поля в записи, как ранее в процедуре установки (не забывая и процедуру DelHook). Текст, например, функции KeyboardProc теперь будет выглядеть так:

```
function KeyboardProc(Code: Integer; wParam: wParam; lParam:
          lParam): integer; stdcall;
begin  {перехват нажатия клавиатуры}
if code<0 then Result:=CallNextHookEx(DataArea^.HookHandleKey,
                    code, WParam, LParam)
else
 begin
 Result:=CallNextHookEx(DataArea^.HookHandleKey, code, WParam,
            LParam);
 if byte(LParam shr 24)<$80 then {только нажатие}
 begin
  if byte(LParam shr 16)= DataArea^.Key then {клавиша}
  if byte(LParam shr 24)= DataArea^.KeyExt then
        {конкретно - дополнительная}
```

```
  begin
   SendMessage(DataArea^.FormHandle, HookMsgKey,
         WParam, LParam);
   Result:=1;
  end;
 end;
 if (byte (LParam shr 16)= $38) and (DataArea^.Alt=True)
 then Result:=1; {отключение Alt, как входа в меню}
end;
end;
```

Теперь в основной программе LangSwitch.dpr перед установкой ловушки заполним нужные поля в записи:

```
. . . . .
FillChar(DataArea^, SizeOf(DataArea^), 0); {очищаем разделяемую
                    область}
DataArea^.FormHandle:=FHandle; {передаем дескриптор нашего окна}
DataArea^.Key:=$1D; {по умолчанию клавиша Ctrl}
DataArea^.KeyExt:=1; {расширенная - т. е. правая}
DataArea^.Alt:=True; {Alt отключать}
SetHook; {запускаем ловушки}
. . . . .
```

В принципе мы могли бы вместо того, чтобы передавать разные сообщения при переключении языка, добавить в запись `THookInfo` еще поля для обратной — от ловушки к программе — передачи параметров активизируемого процесса и анализировать язык непосредственно в программе. Но на двух языках мы ничего особо не выиграем, а те, кто захочет расширить программу и ввести поддержку нескольких раскладок, могут сделать это самостоятельно — при трех языках и более удобнее передавать один параметр, чем множить сообщения.

Кстати, теоретически не мешало бы сделать также еще одну вещь: это мы с вами люди, привычные к работе с клавиатуры, а на свете есть немало пользователей, которые привыкли хвататься за мышь при первой возможности (есть даже чудаки, которые пользуются исключительно экранной клавиатурой). Обратите внимание, что сама Microsoft предусмотрела возможность переключения раскладки мышью, причем в случае, когда языков несколько, выбрать их из меню действительно проще, чем переключаться скрюченными пальцами по кругу. Однако здесь мы не будем этим заниматься — для двух языков это просто нецелесообразно, а упомянутым чудакам наша программа вовсе без надобности. Если же кому-нибудь захочется доделать "переключалку", то укажу возможный путь решения этой проблемы. Основная сложность состоит в том, что при щелчке мышью по иконке окно, для которого

раскладка переключается, теряет фокус ввода. Нужно сделать следующее: перехватить в ловушке событие потери фокуса при его переходе конкретно к нашей программе и определить, какое окно его потеряло. Все это в принципе умеет наша ловушка `HookHandleWin`, если ее немного доработать. Затем через специальное поле в `THookInfo` передать дескриптор этого окна в основную программу. Дальше нужно сделать отдельное меню, как и в Microsoft, возникающее по нажатию левой кнопки мыши, и по событию выбора соответствующего пункта устанавливать нужный язык для окна по полученному дескриптору и возвращать тому же окну фокус ввода.

Осталось организовать собственно установку параметров и ее запоминание. Для этого мы добавим к меню еще один пункт **Установки** (пункт **Закрыть** должен быть в самом низу):

```
. . . . .
popMenu:=CreatePopupMenu;
AppendMenu(PopMenu, MF_STRING, idmSets, 'Установки...');
AppendMenu(PopMenu, MF_SEPARATOR, 0,'' );
AppendMenu(PopMenu, MF_STRING, idmEXIT, 'Закрыть');
. . . . .
```

Не забудем в нашу осиротевшую секцию определений констант добавить константу `idmSets=2`. Так, а что мы будем делать по пункту **Установки**? Для того чтобы организовать соответствующий диалог, проще всего добавить к нашему проекту форму и сделать все "по-человечески", средствами Delphi.

Добавим форму (**File | New | Form**), назовем ее `Sets` и переместим, как и раньше *(см. главу 4)*, в список **Available Forms**. В предложение **uses** основной программы добавим модуль Forms. Кстати, теперь будут доступны и различные поля в пункте **Project | Options**, так что можно присвоить номер версии, только поле `Title` заполнять не следует! Установим для формы параметры, как в заставке из *главы 4* (Object Inspector у нас изначально отсутствует, поэтому его придется вызвать через пункт **View**): `BorderStyle` в `bsNone`, а `Position` в `poDesktopCenter`. Расставим на форме компоненты в соответствии с рис. 7.1, причем сначала нужно установить на форму компонент `Panel`, а уже на него — все остальные, иначе форма совсем без рамочки получится уж больно некрасивая. Для редактора `Edit1` установим свойство `Enabled` в `False` (его мы будем делать доступным при выборе переключателя **Другая**). По умолчанию, как видите, я установил все параметры в соответствии с выбором правой <Ctrl>, как у нас и есть на самом деле. Двухстрочную надпись **Правая (расширенная)** пришлось делать двумя компонентами, добавив к `CheckBox` еще и `Label`, т. к., в отличие от текста в компонентах `Memo`, `Label` и др., заголовки таких компонентов, как `CheckBox` или `RadioButton`, многострочными быть не могут.

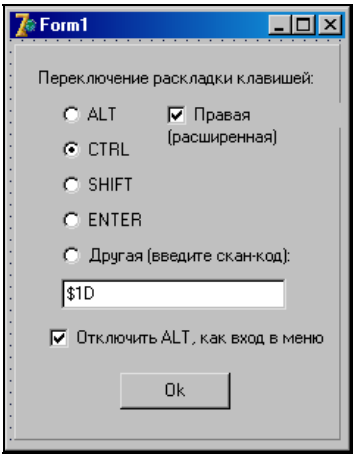


Рис. 7.1. Форма для установок переключателя раскладки

Теперь опять перейдем к основной программе LangSwitch.dpr и вставим после создания всплывающего меню строку:

```
Form1:=TForm1.Create(Application); {создаем экземпляр формы}
```

Соорудим обработчик события по выбору пункта **Установки** из всплывающего меню.

```
if Msg = wm_COMMAND then
begin
  if wpr =idmEXIT then CloseAll;
  if wpr =idmSets then Form1.Show; {показываем форму}
end;
```

Выход из программы я выделил в отдельную процедуру `CloseAll`. Теперь надо еще и уничтожать форму, кроме того, у Windows XP есть "дурная" особенность не выгружать автоматически из памяти неиспользуемые DLL — видимо на всякий случай, вдруг кому пригодится! Чтобы не напрягать никого замусориванием памяти, будем принудительно удалять библиотеку. Так что процедура стала раздуваться, а все время править в двух местах и неудобно и может привести к ошибкам. Выглядеть она будет так:

```
procedure CloseAll;
begin
 if Form1 <> nil then Form1.Free; {если форма существует -
                уничтожаем}
 Shell_NotifyIcon(NIM_Delete,@noIconData); {удаляем иконку}
 DelHook; {удаляем ловушки}
 dWin:=GetModuleHandle('Langhook.dll'); {дескриптор DLL}
```

```
 FreeLibrary(dWin); {удаляем DLL}
 halt; {закрываем программу}
end;
```

Естественно, в обработчике события wm_Destroy тоже надо заменить текст на вызов этой процедуры.

Заметки на полях

Стоит ли показывать форму в модальном режиме? Лично я терпеть не могу модальных диалогов и всячески стараюсь их избегать, кроме случаев, когда от пользователя действительно требуется принятие решения. А если разработчик заставляет пользователя обязательно принимать решение в случае, когда его можно не принимать вовсе — перед вами типичный образчик "программистского мышления". Раз уж вы открыли окно, значит просто *обязаны* что-то делать, считает такой "программист", а не хотите — заставим. Хороший пример подобного подхода: в прекрасной программе Disco Commander, из всех клонов наиболее близкой к оригиналу в лице Norton Commander, в том числе воспроизведена опция, выдающая предупреждение при попытке закрыть программу. Однако разработчики забыли, что Windows — не DOS, в которой компьютер выключали просто кнопкой питания. И, если вы эту опцию установили, при выходе из Windows у вас это предупреждение обязательно вылезет. Причем не сразу, естественно, а через некоторое время. В результате, если вы торопитесь, и сразу после нажатия кнопки в меню **Завершение работы** покинули рабочее место, то, придя обратно, можете обнаружить, что компьютер вовсе и не выключался, а терпеливо ждет, когда вы ему сообщите, что действительно желаете выйти из программы и вообще все выключить. А ведь желание тут было уже высказано пользователем совершенно недвусмысленно — когда он выключал Windows, и задача программиста — обеспечить исполнение этого желания — решена не была. Причем в той же ситуации выводить предупреждение о несохраненном файле — как раз совершенно правильное действие.

Прежде чем перейти к собственно установкам, заранее решим еще одну задачу — их сохранения. Для этого придется создавать дисковый файл (можно сохранять данные и в реестре, но в *главе 17* мы окончательно решим этим не заниматься). Восстановим нашу процедуру создания файла из примера про "шпиона" в конце программы, только назовем его, естественно, иначе. В Delphi (точнее, в Windows) есть механизм работы с INI-файлами и со строками типа <*параметр*>=<*значение*> *(об обращении с INI-файлами штатным способом пойдет речь в главе 9)*, но, честно говоря, я не вижу причин, почему бы здесь нам не организовать все это самим, записав туда просто числа. Мы теряем в наглядности представления, но здесь она не только не требуется, но и вредна, т. к. провоцирует пользователя что-то отредактировать самостоятельно. А зачем ему это может понадобиться, если и сами изменения он будет вносить в установки в лучшем случае пару раз сразу после инсталляции программы? По той же причине мы не будем предельно "вылизывать" сам диалог, хотя и с безразличием тоже отнестись к этому нельзя — работа с диалогом может быть не очень удобной, но по крайней мере не должна создавать проблем.

Установки по умолчанию запишем в файл, если он создается первый раз, а если нет — прочитаем их (установки значений полей записи DataArea, которые делались ранее, кроме дескриптора окна, можно, естественно, удалить).

```
. . . . .
assignfile(ft,'LangSwitch.ini'); {файл установок}
try
 reset(ft); {пробуем открыть}
 readln(ft,DataArea^.Key); {и прочитать}
 readln(ft,DataArea^.KeyExt);
 readln(ft,st);
 if st='TRUE' then DataArea^.Alt:=True else DataArea^.Alt:=False;
except {если не получается, то создаем все заново}
 DataArea^.Key:=$1D; {по умолчанию клавиша Ctrl}
 DataArea^.KeyExt:=1; {расширенная - т. е. правая}
 DataArea^.Alt:=True; {Alt отключать}
 rewrite(ft);
 writeln(ft,DataArea^.Key);
 writeln(ft,DataArea^.KeyExt);
 writeln(ft,DataArea^.Alt);
end;
closefile(ft); {закрываем файл}
. . . . .
```

Заметим, что записывать булевскую переменную в текстовый файл Object Pascal умеет, а вот читать — нет, потому при чтении пришлось "извращаться" со строкой. Далее нужно установить параметры формы-диалога в соответствии с прочитанными значениями:

```
. . . . .
with Form1 do
begin
 if DataArea^.KeyExt=1 then CheckBox1.Checked:=True
 else CheckBox1.Checked:=False;
 if DataArea^.Key=$36 then CheckBox1.Checked:=True;
          {для правой клавиши Shift - особое условие}
 RadioButton5.Checked:=True; {всегда выбран пункт 5 "Другая"}
 if CheckBox1.Checked=True then {кроме случаев}
 case DataArea^.Key of {перечисленных здесь}
   $38: RadioButton1.Checked:=True; {Alt}
   $1D: RadioButton2.Checked:=True; {Ctrl}
   $36: RadioButton3.Checked:=True; {Shift}
   $1C: RadioButton4.Checked:=True; {Enter}
 end;
```

```
 Edit1.Text:='$'+hexb(DataArea^.Key);
end;
. . . . .
```

Здесь мы использовали модуль Ariphm для вывода текста в HEX-форме, его нужно добавить в **uses**. Теперь перейдем, наконец, собственно к установкам и создадим в новом модуле процедуру по нажатию клавиши **Ok**. Выглядеть процедура нажатия клавиши **Ok** может примерно так:

```
procedure TForm1.Button1Click(Sender: TObject); {кнопка Ok}
begin
 DataArea^.Key:=StrToInt(Edit1.Text); {обходим правую Shift}
 if Edit1.Text='$36' then DataArea^.KeyExt:=0
 else
 begin
  if CheckBox1.Checked=True then DataArea^.KeyExt:=1
                    {расширенная}
  else DataArea^.KeyExt:=0; {обычная}
 end;
 DataArea^.Alt:=CheckBox2.Checked; {Alt, как меню}
 assignfile(ft,'LangSwitch.ini'); {файл установок}
 rewrite(ft);
 writeln(ft,DataArea^.Key);
 writeln(ft,DataArea^.KeyExt);
 writeln(ft,DataArea^.Alt);
 closefile(ft); {закрываем файл}

 Form1.Hide; {скрываем форму}
end;
```

Для того чтобы это правильно работало, нужно обеспечить смену текста в редакторе Edit1 при выборе различных пунктов (кроме **Другая**, когда текст можно сменить только вручную). Нужно также обеспечить активацию редактора (свойство Enabled) при выборе пункта **Другая** и его деактивацию при выборе иного пункта (если помните, он у нас изначально деактивирован). Для этого надо добавить обработчики событий onClick для каждой из кнопок RadioButton вот по такому шаблону:

```
procedure TForm1.RadioButton5Click(Sender: TObject);
begin
 Edit1.Enabled:=True;
end;

procedure TForm1.RadioButton1Click(Sender: TObject);
begin
 Edit1.Text:='$38';
```

```
 Edit1.Enabled:=False;
end;
. . . . . {и т. д.}
```

Для <Ctrl> (RadioButton2) значение должно быть '$1D', для <Shift> (RadioButton3) — '$36' и для <Enter> (RadioButton4) — '$1C' *(см. приложение 2)*. Обратите внимание, что для правой клавиши <Shift> условие особое, т. к. она отличается от остальных дополнительных клавиш тем, что имеет оригинальный скан-код *(см. ранее)*. Так что для нее устанавливать значение KeyExt в 1 нельзя — не будет работать никакая клавиша. И чтобы не морочить голову "чайникам" такими тонкостями, я принудительно устанавливаю для нее "галку" в пункте **Правая (расширенная)**, и особым образом обхожу это в программе.

Мы обещали не "вылизывать" диалог, и в данном случае это также выразится в том, что мы не будем отслеживать запрещенные комбинации "скан-код + бит расширенной клавиши" — это довольно громоздкая задача, проще будет сделать соответствующее пояснение в справке. Там же надо не забыть отразить момент, связанный с неоднозначностью определения "правый-левый" для расширенных клавиш, которое, как мы помним, относится только к <Alt>, <Ctrl>, <Enter> или <Shift>, но не к случаю дополнительной клавиатуры.

Еще необходимо делать в редакторе проверку на ввод числового значения и на их диапазон (не более 127) — подобные проверки нужно делать всегда. Мы сделаем самым простым способом — объявим переменную OldText:**String** и будем через нее возвращать старый текст при ошибке во вводе:

```
procedure TForm1.Edit1Change(Sender: TObject);

begin
 if Edit1.Text<>'' then
 try
  if StrToInt(Edit1.Text) > 127
  then Edit1.Text:=OldText; {пробуем преобразовать в число
                и заодно проверяем диапазон}
 except
  Edit1.Text:=OldText; {если не получается - восстанавливаем}
  Edit1.SelStart:=length(Edit1.Text);
  Edit1.SelText:=''; {курсор в конец текста}
  exit;
 end;
  OldText:=Edit1.Text;
end;
```

```
procedure TForm1.FormShow(Sender: TObject);
begin
  OldText:=Edit1.Text; {запоминаем текст}
end;
```

Способ не слишком удобный для пользователя — автомат есть автомат, попробуйте сами и убедитесь. Использовать же компонент `SpinEdit`, который представляет собой конструкцию `Edit` + `UpDown`, специально предназначенную для ввода чисел, тут не получится — мы хотим вводить числа также и в шестнадцатеричном виде, а `SpinEdit` нам этого не позволит. Но наш механизм вполне работает, а "наворачивать" всякие диалоги с объяснением, что именно пользователь сделал неправильно, в данном случае нецелесообразно просто из-за редкости обращения к этому пункту.

Кстати, наша программа имеет один минус — т. к. она работает через сканкоды, то с ее помощью нельзя делать различия между клавишами цифровой клавиатуры при выключенном и включенном <Num Lock> (посмотрите на таблицу в *приложении 2*). Поэтому использовать удобную возможность, которую дает цифровая "пятерка" при выключенном <Num Lock>, в полной мере не удастся — клавиша также перестанет работать и для ввода цифры "5". Для исправления этой ситуации нужно дополнительно либо отслеживать состояние <Num Lock>, либо анализировать также и виртуальные коды — но такие "навороты" я оставляю читателю в качестве домашнего задания. Неплохо бы также ввести отдельный пункт по выбору режима возвращать/не возвращать обработку клавиши системе — для кого-нибудь опция может оказаться полезной.

Полный текст всех модулей программы вы можете посмотреть на прилагаемом диске. На данном же этапе программа еще не закончена. Это мы сделаем позднее — добавим справку, пункт **О программе**, инсталляционный пакет, с помощью которого программу можно будет регистрировать в реестре, и т. п. В частности, при установке целесообразно "убить" системный индикатор раскладки (зачем нам сразу два индикатора?) и выключить пункт **Переключать раскладки клавиатуры** в Windows XP — сейчас это приходится делать вручную, а в процессе настоящей инсталляции мы предложим делать это автоматически. Данный проект — хороший пример того, что обеспечение главной функциональности программы (собственно переключения раскладки) очень часто занимает малую часть всего времени, потраченного на ее написание. Сравните наш первый вполне работоспособный проект простейшего переключателя и гораздо более сложный, но и гораздо более "фирменный" окончательный вариант.

ГЛАВА 8

Unicode и другие звери

Как работать с документами в различных кодировках

> Если пациент сразу решил закодироваться, то он элементарно может "пролететь": метод кодировки не всегда действует мгновенно и безотказно. Тогда приходится кодироваться снова.
>
> *Газета "Новая Сибирь", 12 ноября 1999 г.*

В компьютерах все — и данные, и команды, и буквы-символы — представляется одинаково, в виде чисел. И это уже дело наше (точнее, программного обеспечения), как эти числа интерпретировать: компьютеру решительно все равно, какие именно символы какими именно числами кодировать. Мы можем договориться, что в данной программе число 13 обозначает команду на перевод строки (или создание нового абзаца, если форматирование строк производится редактором автоматически), число 48 обозначает знак цифры "0", а число 65 — знак строчной латинской буквы "A". Разумеется, если бы каждая программа имела бы свое собственное представление о том, как интерпретировать подобные *коды*, то сложилась бы ситуация строителей вавилонской башни — текст, созданный в одной программе, нельзя было бы прочесть больше нигде. Поэтому естественно возникло стремление к созданию единого стандарта, желательно общемирового, который позволил бы одинаково воспроизводить на экране и при печати тексты на любых языках. Все прекрасно знают, что подобный идеал не достигнут и по сей день, хотя история кодировок насчитывает уже четыре с лишним десятилетия. А в чем проблема, собственно?

У тех, кому пришлось решать нелегкую проблему разработки кодировок разноязычных текстов, было две разные, вообще говоря, задачи. Первая задача: отобразить на экране то, что было когда-то кем-то набрано, причем именно так, как этот "наборщик" задумал (в широком смысле отобразить — например, в том числе дать понять средствам проверки орфографии, какой перед ними язык, или принтеру — каким шрифтом печатать). Вторая задача — переключение раскладки и экранных шрифтов при вводе и, возможно, сохране-

ние этой комбинации в документе во имя выполнения первой задачи. Рассмотрим сначала, как решались эти задачи в исторической перспективе.

О кодировках

Надо сказать, что обеспечить выполнение указанных задач в общем случае непросто, и все DOS-тексты тому ярчайший пример: если отображать нечто, набранное в DOS-редакторе французом или испанцем, т. е. с использованием символов с номерами 128—255, то на компьютере с включенной поддержкой русской кодовой страницы вы получите нечитаемый текст из бессмысленного набора кириллических символов вперемешку со специальными знаками. То же самое будет, если исходный текст вполне русский, но текущая и заложенная в документ кодировки различаются.

Так как же дать понять компьютеру, что перед ним некий конкретный язык (кодировка) с тем, чтобы он подставил в нужные места нужные начертания символов (кодовую страницу)? Решение задачи — даже два разных решения — на самом деле было известно давно, со времен слепого французского врача Луи Брайля (Louis Braille, 1809—1852 гг.), который еще в мальчишеском возрасте придумал последовательную систему рельефных точек, кодирующих буквы алфавита, цифры и знаки препинания для слепых — несколько дополненная, азбука Брайля используется и в наши дни. Базовый элемент системы Брайля содержит 6 позиций-ячеек, каждая из которых может быть выпуклой или плоской — т. е. его кодировка основана на двоичном коде. Всего в ней можно закодировать $2^6 = 64$ символа, однако Брайлем был предусмотрен механизм практически неограниченного расширения количества кодируемых символов — для этого употребляются коды-переключатели (Shift-коды). Наличие такого переключателя означает, что все последующие знаки надо читать определенным образом (например, как цифры, а не буквы), до тех пор, пока не встретится другой подобный знак. Другая разновидность переключающих кодов (Escape-коды) действует только на один знак после такого кода, и тоже впервые введена в азбуке Брайля. Забегая вперед, заметим, что вы уже, несомненно, узнали Shift-переключатели в современных тегах HTML.

> Предшественником современных компьютерных кодировок принято считать коды, разработанные еще в 1874 г. французом Эмилем Бодо (Jean-Maurice-Emile Baudot 1845—1903 гг.), усовершенствованные позднее Дональдом Мюрреем и принятые в качестве международного стандарта в 1931 году. Официальное название кода Бодо — International Telegraph Alphabet #2, ITA-2. Он применялся в телетайпных аппаратах и, соответственно, в первых компьютерах, некоторые из них эти аппараты использовали

в качестве входных/выходных консолей. Коды Бодо – пятибитовые, поэтому количество представленных символов ограниченно 32, чего для всех необходимых символов не хватает. Для увеличения этого числа, как и в коде Брайля, использовались Shift-переключатели. Основная проблема была в том, что переключатель типа Shift действует до тех пор, пока не встретится другой переключатель, отменяющий первый. Поэтому если какое-то сообщение заканчивается цифрой, то начало следующего за ним текста до первого отменяющего переключателя будет представлять бессмысленный набор цифр вместо букв (в HTML эта проблема решена тем, что каждому подобному тегу поставлен в соответствие закрывающий тег). С этим мог сталкиваться каждый, кто пытался печатать "красиво" на матричных принтерах, скажем, на популярном некогда FX-800 или еще более древнем Robotron 6329. Если некий текст заканчивался курсивом без отмены этого режима в конце документа, то следующая попытка что-то распечатать приводила к тому, что документ тоже начинался с курсива.

В 1963 году возник, а в 1967 году был утвержден в качестве стандарта American Standard Code for Information Interchandge — ASCII *(см. приложение 3)*, который и является до сих пор основой всех кодовых таблиц, устанавливая символы с номерами 0—127 (точнее, 32—127, т. к. символы с номерами 0—31 представляют собой команды и могут интерпретироваться поразному, хотя формально входят в ASCII). У истоков ASCII стояла фирма AT&T, а также IBM. Ради полноты картины следует упомянуть, что для мэйнфреймов IBM вплоть до начала 80-х годов был принят другой код — EBCDIC, довольно громоздко устроенный, и ведущий свое происхождение от систем перфокарточного ввода.

Что касается русскоязычных кодировок, то в середине 70-х гг. возникла KOI, в которой русские буквы во второй половине таблицы ставились на такие места, чтобы при вычитании числа 128 (т. е. при обнулении старшего бита) из кода получалась соответствующая по звучанию (но не всегда — по написанию) английская буква, причем в противоположном регистре, чтобы отличить английский текст от русского. Скажем, "Русский Текст" превратилось бы в "rUSSKIJ tEKST". Сделано это было потому, что первые почтовые серверы были семибитовыми, т. е. могли передавать только ASCII. KOI-8 существовала в виде общесоюзного стандарта (ГОСТ-19768-74) и даже чуть было не была утверждена в качестве международного (ISO-IR-111 или ECMA-Cyrillic). Упомянутый ГОСТ впоследствии был заменен на мертворожденный ГОСТ 19768-93[1], вообще к нам — пользователям ПК — никакого отношения не имеющий, т. к. устанавливает ни с чем не совпадающие кодовые таблицы для ЕС ЭВМ, выпуск которых почти полностью к момен-

[1] Часто упоминающегося в этой связи ГОСТ'а под номером 19768-87, судя по официальному их перечню, вообще не существовало в природе.

ту создания этого ГОСТ'а был прекращен. Таким образом, надо понимать, KOI-8r в настоящее время не описывается никаким ГОСТ'ом. Тем не менее KOI-8r (вообще кодировок с общим названием KOI существует по меньшей мере семь) как была, так и осталась самым распространенным стандартом для электронной почты и в славные времена создания Релкома, в конце 80-х гг., была возведена в ранг интернет-стандарта под названием RFC-1489. Тут, конечно, сыграло свою роль то, что она и была к тому времени стандартом де-факто (подкрепленным авторитетом того самого ГОСТ'а от 1974 года) для UNIX-систем, которые доминировали в сетевых делах.

С распространением компьютерного дела в нашей стране и в родственных странах было разработано множество разных кириллических кодировок (в том числе "украинская", "польская", "болгарская" и др. — последняя и до сих пор используется в болгарской Linux). Среди них наибольшую известность получили "основная" (ISO/IEC 8859-5-88) и "альтернативная" (MS-DOS, CP866) кодировки ГОСТ, на текущий момент стандартизированные в ГОСТ Р 34.303-92, который в основной своей части (касающейся таблицы 8859-5) является простым переводом стандарта ISO 4873-86. "Основная кодировка ГОСТ", несмотря на солидную поддержку со стороны разработчиков стандартов, широко не использовалась на практике, видимо, никогда[2] — придумать хоть какое-то рациональное обоснование ее преимуществ невозможно, сам Госстандарт сейчас спокойно приветствует посетителей на своем сайте в кодировке Win1251.

Сложившимся стандартом для DOS стала "альтернативная" кодировка, которая отличается от всех остальных наибольшей, пожалуй, продуманностью (на своих местах остались символы псевдографики и многие другие спецсимволы из второй половины ASCII — т. е. оформленный с их использованием английский текст абсолютно не менялся независимо от текущей кодовой страницы). Но в этой кодировке символы располагаются не подряд (как будто это имеет хоть какое-нибудь принципиальное значение), потому с развитием Windows придумали еще одну кодировку, в обычном стиле Microsoft проигнорировавшую все традиции и стандарты. Ее создатели отчасти справедливо рассудили, что символы псевдографики при наличии графических WYSIWYG-редакторов как-то ни к чему. Кириллическая азбука волюнтаристски заняла последние 64 ячейки таблицы, кроме букв "Ё" и "е", которые вставили куда-то в середину. Новая кодировка получила название "Windows 1251" (Win1251). Интересно, что внедрением и CP866 в 1989 году и Win1251 в 1995 году занимался один и тот же человек — россиянин Петр Квитек. На самом деле на настоящий момент действуют па-

[2] Единственный известный автору пример ее использования, как основной кодировки по умолчанию — русский Nescape Composer. В те времена 8859-5 традиционно всегда включалась в меню выбора кодировки браузеров и мейлеров (стандарт все-таки), но уже в Internet Explorer 6 и других современных браузерах она отсутствует.

раллельно пять однобайтовых кириллических кодировок, вы с ними можете ознакомиться в *приложении 3*.

Истоки еще одной отдельной разновидности кодировок надо искать в устройстве первых почтовых серверов — это пересылка вложений. Казалось бы, в чем проблема — передавай себе байты по Сети, и все тут. Но, как мы говорили ранее, почта изначально была "заточена" исключительно под обмен английскими алфавитными символами, т. е. кодами ASCII 32—127. Коды в таблице ASCII, меньшие чем 32, т. е. команды, вообще не должны отображаться на экране, а коды, большие 127, могли обрезаться семибитными серверами (или подвергаться двойной перекодировке в российских пенатах — хоть эта проблема, к счастью, почти изжита). Как же отличить текст письма от вложения? Для этого последние, в которых по определению могут содержаться байты с любым значением, передают довольно сложным путем: коды преобразовываются так, чтобы они содержали только байты со значением из интервала 33—127. Так устроены распространенные системы кодирования вложений, например, base64.

Таким образом, чтобы решить проблему отображения многоязычных документов, напрашивается следующий путь: просто стандартизировать ряд Shift-кодов ("тегов"), определенных для каждого языка, и проблема решена — по крайней мере, в принципе. Правда, то, что называется "чистый текст", все равно осталось бы читаемым неоднозначно, но для продвижения многоязыковых систем была бы хорошая отправная точка. По этому пути попытались пойти разработчики из ISO и, вслед за ними, из Госстандарта. ГОСТ 27463-87 (он же ISO 646-83, устанавливающий национальные таблицы 7-битных кодов), ГОСТ 27465-87 (устанавливающий наборы русскоязычных символов) и обширный ГОСТ 27466-87 (ISO 2022-86, устанавливающий правила расширения кодовых таблиц) представляют собой довольно стройную систему, являющуюся ничем иным, как альтернативой Unicode. Их принцип — это переключение между наборами символов с помощью Shift-переключателей, которые в этих документах называются "последовательности АР2". Система эта не получила никакого практического применения, видимо, по двум причинам: в силу своей громоздкости и, главное, ограниченности — уже для алфавитов, включающих более чем 256 символов, она становится настолько сложной, что двухбайтовая Unicode выглядит на ее фоне много симпатичнее. Нечто похожее, правда, с меньшим размахом, пытались придумать и в ANSI (American National Standarts Institute) — в DOS можно было подключить некий ANSI-драйвер, которым, однако, на практике никто все равно не пользовался.

В HTML идея Shift-переключателей в конце концов была доведена до некоторого логического завершения. В части языковой поддержки соответствующие теги в HTML просто указывают, какой именно шрифт нужно в данный

момент использовать (а также его начертание, размер, цвет и т. п. — см. *главу 16*). Совершенно аналогично устроен, например, формат RTF. А уж личное дело пользователя — есть у него на компьютере такой шрифт или нет. Вот на этом последнем обстоятельстве здравая в принципе идея "тормознулась": проблема наличия нужных шрифтов оказалась настолько сложной, что редакторы интернет-ресурсов обычно справедливо запрещают использовать длинные тире, фигурные кавычки или символы типа "№" в сетевых публикациях — с высокой степенью вероятности они у пользователя будут выглядеть одинаковыми знаками вопроса[3]. В перемещаемых же документах все несколько проще — по идее, можно внедрить шрифт в сам документ, хотя это и увеличивает его размер, но зато гарантирует от неправильного отображения.

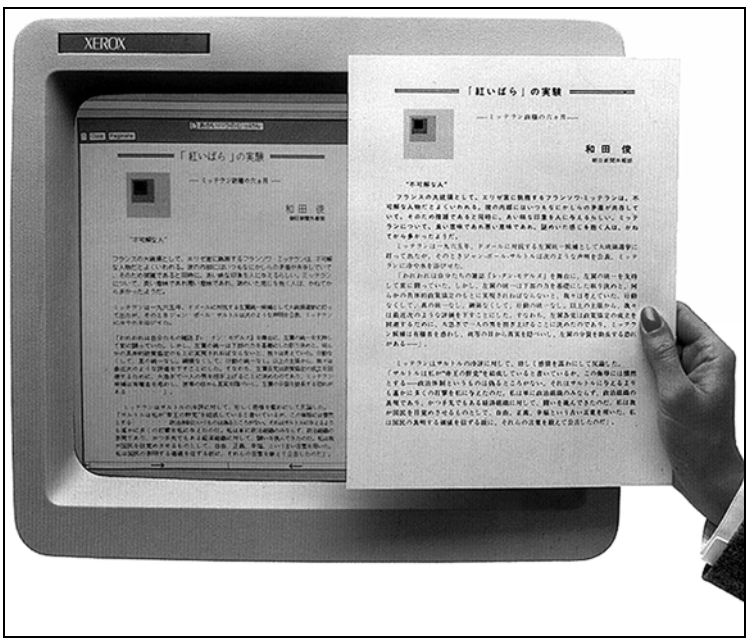


Рис. 8.1. Демонстрация WYSIWYG-возможностей Xerox Star с отображением текста на корейском языке, 1981 год

Так что, проблема отображения решена? Не тут-то было. Во-первых, остается тяжелейшая проблема совместимости "чистых текстов", от которых, понятно,

[3] Для Windows-пользователей проблема легко решается объявлением обязательно присутствующих в системе шрифтов типа Times New Roman, как и поступают текстовые редакторы от Microsoft при преобразовании документа в HTML. Однако достаточно многочисленные пользователи *nix-систем оказываются при этом, как правило, не у дел. Правильным решением этой проблемы служит вставка вместо шрифтовых символов специальных знаков *(см. главу 16)*.

никто отказываться не собирается по очень многим существенным причинам. В Word, особенно последних версий, кстати, проблема именно однобайтных кодировок решена отлично, но этого нельзя сказать о многих других редакторах. Беспроигрышным путем пошли разработчики редактора Edit Plus, когда пользователь сам подбирает шрифт в нужной кодировке — безупречно работающим, но крайне неудобным на практике. Но наилучший способ применили еще в начале 80-х гг. разработчики упоминавшегося в *главе 1* эпохального компьютера Xerox Star. Они ввели двухбайтную кодировку, которая должна была охватить все существующие языки (рис. 8.1). Эта идея и оказалась наиболее жизнеспособной, а если вернуться к Брайлю — она есть не что иное, как модернизированная идея Escape-переключателей, т. е. переключателей, действующих не до отмены, а только на единственный следующий символ.

Unicode

Введение двухбайтной кодировки было задумано хорошо, а выполнено (по крайней мере, в Windows) — "как всегда". Читаем в официальном пособии [1, стр. 519]: *"Архитектура многоязыковой поддержки базируется на раскладках клавиатуры"*. Привязка переключения раскладки к переключению языка была, конечно, ошибкой: на практике следовало эти функции жестко и безоговорочно разделить. А зачем, однако, разработчикам Windows вообще понадобилось отдельно понятие "языка", если он при отображении на экране однозначно определяется текущей кодовой страницей, а при вводе — текущей раскладкой клавиатуры? Все дело в том, что через это разработчики Microsoft еще со времен Windows 3.*x* пытались обеспечить поддержку многоязычных документов и, надо сказать, сначала немало в том преуспели.

Вернемся к Shift-переключателям и попробуем воспроизвести логику работы Word 6.0 с его однобайтными шрифтами при отображении многоязычных документов. Напомним, что есть две задачи: при вводе надо правильно скомбинировать раскладку клавиатуры и текущую кодовую страницу, что в совокупности разумно объединить под неким понятием "язык" (как и было сделано в локализованных версиях Windows 3.*x*). А при отображении уже написанного текста раскладка клавиатуры в принципе не участвует: можно пометить разноязыкие участки текста некими "тегами", и тогда независимо от раскладки документ воспроизведется правильно. Понятие языка в том же смысле, как и ранее, здесь вообще не требуется — установленный "тег", как и в HTML, означает всего лишь шрифт нужного начертания, или, если хотите, нужной кодовой страницы (как мы видели в *главе 6*, понятия текущей кодовой страницы и текущего экранного шрифта в принципе равнозначны). Именно поэтому вплоть до Windows 98 существовал отдельно Arial, Arial CE и Arial Cyr. Но понятие "языка", объединяющего раскладку и экранный

шрифт (кодовую страницу), тут еще не мешает: можно переключить шрифт с помощью соответствующего "тега", можно — отдельно раскладку соответствующей командой, а можно "язык", т. е. и то и другое. Но суть дела в том, что при переходе от однобайтных шрифтов к Unicode этих ухищрений уже вообще не требуется!

В Unicode все организовано элементарно просто и логично: если первый байт равен нулю, то второй соответствует начертанию символа из таблицы ASCII (или, что то же самое, английскому языку). Если же, к примеру, первый байт равен 4, то это кириллица. Русские буквы в таблице располагаются подряд, начиная со значения младшего байта, равного 16, "Ё" и "е", как всегда, отдельно *(см. приложение 3)*. Всякие фигурные кавычки и прочие длинные тире выбираются из отдельных таблиц со своим значением первого байта. В принципе, если вы имеете некий Unicode-шрифт со всеми мыслимыми начертаниями знаков, то можете писать многоязычный текст на всех языках вперемешку. Правда, такой шрифт известен пока только один: это Arial Unicode MS, который имеет объем более 13 Мбайт, и оттого может еще и заработать не на всяком компьютере. Но на практике выше головы хватает и десятка заполненных национальных страниц — что и наблюдается в ходовых TTF-шрифтах.

Unicode и Win32

Однако в Windows 32-разрядных версий весь механизм объявления языка стал настолько запутанным, что здравая идея двухбайтных Unicode-символов, однозначно определяющих язык, оказалась полностью выхолощенной. Вот как это происходит в логике разработчиков, например, Word. Вы устанавливаете текущую раскладку (в Windows 98 это и называется именно так — "раскладка", но в Windows XP, как мы увидим, будет уже "язык"), в соответствии с чем происходит переключение кодовой страницы и символы вводятся в нужном начертании. Как мы говорили ранее, сверх этого никакая установка языка отдельно уже не требуется — ведь мы имеем Unicode, в котором каждый символ однозначно сам определяет свою языковую принадлежность. Как бы не так! Наберите в Word русский текст, выделите его, и через меню **Сервис | Язык | Выбрать язык** объявите его хоть английским, хоть албанским — и такая операция у вас спокойно пройдет! Это кому и в каком страшном сне могло присниться, что текст, который вы читаете сейчас, может быть объявлен английским (причем "английскими" также станут цифры, пробелы, кавычки, точки, запятые...)? Я, как автор этого текста, однозначно и недвусмысленно указал, выбрав раскладку, что перед вами именно русский текст. Иногда в нем встречаются английские слова и отдельные буквы, и для них также однозначно указано в первом байте символа, что именно вот это есть

символ английский. Отсюда кошмарная неразбериха с проверкой правописания, когда вы не знаете, подчеркнуто красным слово потому, что вы его неправильно написали, или потому, что оно "английское". Особенно забавно выглядит, когда вы можете написать английское слово, объявить его русским, потом вернуться в его середину, переключиться на английский и вставить одну букву: все, что до этой буквы и после нее, будет подчеркнуто красным, как ошибочно набранное, а сама буква — нет.

Это даже не "программистское мышление", это гораздо хуже, ведь на самом деле никакой операции "присвоения языка" в старом смысле — раскладка там, экранные шрифты — не происходит. Все, что вы проделали, есть чисто фиктивная акция, которая действует разве что на средства проверки орфографии и не самым лучшим, как мы видим, образом. И вы не можете таким образом воздействовать ни на раскладку, ни на переключение экранных шрифтов (кодовой страницы), которое по-прежнему оказывается привязано к раскладке, по самому смыслу относящейся только к вводу, а не к отображению символов. Получается, что разработчики Word фактически ввели еще одно понятие "языка" поверх всех существующих. Раскладки, Unicode, кодовые страницы существуют где-то в ином измерении. Если программа ошиблась в определении кодовой страницы при отображении (демонстрирует то, что удачно окрестили "кракозябрами"[4]), то исправить эту ошибку средствами, доступными пользователю (как в браузерах — просто выбором кодировки вручную), не удастся — хотя это самая прямая и, пожалуй, единственная обязанность операции выбора "языка", которая могла бы оправдать существование самого этого понятия отдельно (и в однобайтных кодировках оправдывала!). Логично было бы просто взять текст с "кракозябрами", выделить его, и упомянутым ранее способом объявить язык русским. А система бы тогда аккуратно объяснила пользователю, что в использованном им шрифте поддержки русского нет, а потому нужно либо оставить все, как есть, либо шрифт заменят на другой подходящий. Видно, как люди решали сиюминутные проблемы, наращивая слой за слоем функциональность, спущенную им из маркетингового отдела.

А что касается Windows XP с декларированной поддержкой Unicode на уровне системы, то там картина еще печальнее — возможность различать однобайтные кодовые страницы окончательно привязана к раскладке. Проведите следующий эксперимент: откройте Word (подразумевается, что у вас Word также версии 2000 или выше) и Блокнот, затем наберите в Word любое русское слово и перенесите его через буфер обмена в Блокнот. Получается следующий поразительный результат: если в Блокноте вставлять текст при

[4] Или, не менее удачно, "арабской вязью". Интересно, что американцы в аналогичных случаях говорят нечто вроде "опять эти японцы кириллицей замучали".

включенной (для него) русской раскладке, то все вставляется правильно, а если при английской — получаются "кракозябры"! Для пущего эффекта можно вставить два раза подряд, переключив только раскладку. Еще интереснее получается при вставке в редактор Delphi: если вставлять из Word чистый русский текст, то все вставляется правильно (независимо от раскладки), а если перемешанный (текст программы с комментариями на русском) — вместо русских букв одни знаки вопроса! Мало того, даже в рамках самого Office можно попасть в безвыходную ситуацию: попробуйте скопировать из редактора Visual Basic что-нибудь русское, и вставить в текст документа Word, или наоборот.

Заметки на полях

Есть совершенно неочевидный "народный" рецепт того, как отчасти исправить в Windows XP ситуацию с русским языком[5]: для этого в **Панели управления** надо щелкнуть на иконке **Языки и региональные стандарты**, зайти на вкладку **Дополнительно**, и отметить в окне **Кодовые страницы таблиц преобразования** абсолютно все пункты (включая **IBM EBCDIC...**, **MAC...** и т. п.). Система затребует дистрибутивный диск и после перезапуска, по крайней мере обмен между Блокнотом и Word заработает нормально. Естественно, кодовые страницы типа, например, "TeleText тайваньская" никак не могут оказывать влияния на русский язык, но, видимо, при этой операции подгружается какой-то модуль, который ранее отсутствовал. Другой рецепт — вручную (или с помощью соответствующего REG-файла) внести изменения в реестр:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
\FontSubstitutes]
   "Arial,0"="Arial,204"
   "Courier New,0"="Courier New,204"
   "Courier,0"="Courier New,204"
   "Times New Roman,0"="Times New Roman,204"
   "Times,0"="Times New Roman,204"
```

Тем не менее проблемы с не-Unicode программами останутся и нет оснований полагать, что в платформе .NET ситуация не будет еще хуже. В мае 2005 года появилась пре-бета-версия будущей Longhorn. Разумеется, окончательных выводов делать нельзя, но по свидетельству тех, кто ее пробовал, добиться нормальной поддержки русского в не-Unicode редакторах практически невозможно.

Наиболее запутывает пользователей, но и многое объясняет в логике разработчиков появление в Windows XP дополнительной функции "переключения языка" и изменение в этой связи смысла понятия "раскладки" (**Панель управления**, запустить сервис **Языки и региональные стандарты** и зайти на вкладку **Языки | Подробнее | Параметры клавиатуры | Смена сочета-**

[5] Этот способ самостоятельно обнаружил мой шестнадцатилетний племянник.

ния клавиш). После экспериментов выясняется, что пункт **Переключать языки ввода** как раз раскладку-то и переключает, а пункт **Переключать раскладки клавиатуры**, судя по всему, не делает ничего. На самом деле с ним связана вот какая операция: для любого языка в принципе можно ввести несколько разных раскладок (для русского, например, машинописную или обычную компьютерную, они различаются регистром для отображения цифр и еще некоторыми особенностями). Таким образом, понятие "языка" обретает некоторую законченность. Путаница же тут происходит от того, что с точки зрения системы "русский машинописный" и "русский компьютерный" — такие же разные языки, как греческий с французским, и вводить отдельные понятия было совершенно ни к чему (а если уж и вводить, то, как мы говорили, до конца последовательно). Поскольку на практике о таком разделении труда между "языком" и "раскладкой" чаще всего никто и не подозревает (даже зная о такой возможности, очень немногие будут ее реально использовать), то отключение данного пункта ничего в системе не меняет, и идет только на пользу в случае использования имитатора нажатия клавиш переключения раскладки, как мы это делали в предыдущей главе. Напомним, что в этой книге слово "раскладка" употребляется в изначальном смысле, как переключение экранных шрифтов для операции ввода символов.

Программа преобразования Unicode в чистый текст

Попробуем создать одну небольшую утилитку, которая иногда может очень пригодиться на практике. Существует множество программ, позволяющих осуществлять перекодировку русскоязычного текста из произвольной восьмибитной кодировки в любую другую. Однако прочтение текста в кодировке Unicode, если это не штатный формат какой-либо Unicode-программы, может стать проблемой. Такая задача может возникнуть, например, если вы хотите прочесть скрытый текст документа Word — дело в том, что популярнейший редактор имеет интересную привычку сохранять изменения, которые внесены в документ. Уничтожить следы вашего творческого процесса, чтобы ими не воспользовались супостаты, легко: нужно выделить весь текст готового документа и перенести его через буфер обмена в новый документ и сохранить под старым именем. Заодно это позволит сильно сократить размер файла, особенно если вы по ходу вставляли и убирали из него картинки. Но иногда просто необходимо восстановить удаленные по ошибке фрагменты текста, а это не так-то просто сделать, если вы уже вышли из программы. Для старых не-Unicode версий (Word 6.0) достаточно было изменить расширение файла с doc на txt и открыть его как "просто текст", но в Unicode-форматах (Word 97, 2000, XP) вы ничего не разберете — приходится применять про-

грамму перекодировки. Кроме того, это единственный известный автору способ спасения DOC-файлов, испорченных в результате, например, некорректных дисковых операций (или просто нечаянным ручным редактированием).

Преобразование "вручную"

Вот короткий алгоритм перевода Unicode в "чистый текст" (который, впрочем, в случае DOC-файлов окажется все равно довольно "грязным", хотя и читабельным), который мы воплотим на практике — разумеется, ограничимся только русской и английской кодовыми страницами. Как мы уже говорили, в Unicode для английского языка первый (старший) байт двухбайтного символа равен 0, а для кириллицы — 4. Русские буквы начинаются со значения младшего байта, равного 16, т. е. прописная "А" соответствует коду 04 16, прописная "Б" — 04 17 и т. д. Всего 64 знака, описывающих прописные и строчные буквы русского алфавита, кроме "Ё" и "е", которые мы опустим. Для того чтобы раскодировать такую запись, нужно проделать следующие операции:

1. Прочесть значение первого байта.
2. Если оно равно 0, то следующий байт записать в новый файл без изменений, если только он тоже не равен нулю (в формате DOC довольно много резервных полей с нулевым значением байт, но они нас не интересуют), перейти к чтению следующей пары.
3. В противном случае, если значение первого байта равно 4, то прибавить к значению второго байта 176 (в кодировке Win1251 русские буквы начинаются с символа номер 192) и записать полученное значение в новый файл, перейти к чтению следующей пары.
4. В противном случае (не 0 и не 4) можно записать оба байта, как есть, а можно сразу перейти к чтению следующей пары (чтобы миновать, скажем, коды картинок и служебные поля). Во втором случае мы потеряем многие служебные символы, но они все равно будут отображаться неправильно — чтобы в конечном тексте отображалось все корректно, над алгоритмом нужно очень потрудиться.

В результате вы получите текстовый файл в кодировке Win1251, который содержит достаточно мусора, но главная задача будет выполнена — просмотрев его, вы можете открыть для себя много интересного.

Итак, создадим новый проект, назовем его Unicode, модуль с формой назовем просто Code, и сохраним все это дело в отдельной папке (Glava8\1). Заголовок формы переделаем в `Unicode reader`, для порядка можно заменить и иконку. На форму поместим главное меню, диалог открытия файла и компонент `Memo`, который нам в принципе не нужен, но будет служить для контроля.

Для него мы установим свойство `ReadOnly` в `True` (нам редактировать ничего не надо), и введем вертикальную линейку прокрутки (`ScrollBars` установить в `ssVertical`). Текстовый курсор в `Memo` нам тоже не требуется (только жутко раздражает), но, разумеется, просто взять и отменить его нельзя (от введения столь немудреной операции, несомненно, пострадала бы репутация разработчиков Windows), поэтому создадим обработчики:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  HideCaret(Memo1.Handle); {скрываем курсор}
end;

procedure TForm1.Memo1Change(Sender: TObject);
begin
  HideCaret(Memo1.Handle); {скрываем курсор}
end;
procedure TForm1.Memo1Click(Sender: TObject);
begin
  HideCaret(Memo1.Handle); {скрываем курсор}
end;
```

Первый будет скрывать курсор при запуске, второй — при изменениях в `Memo1` и третий — при щелчке на него мышью (проще не получается!).

В диалоге открытия файла я установил фильтр по расширению `*.doc`, хотя, разумеется, это необязательно. Теперь введем переменные:

```
var
Form1: TForm1;
ff,fo:file;
xs,xofs:byte;
xw:word;
xz,i:integer;
st:string;
```

Мы выбрали нетипизированные файлы, потому что так удобно определять точный размер файла, и также читать численные значения в формате `word` (т. е. по два байта), а писать — в формате `byte` (по одному). В главном меню создадим единственный пункт **Открыть**, и обработчик соответствующего события для него будет выглядеть так:

```
procedure TForm1.Open1Click(Sender: TObject);
begin
If OpenDialog1.Execute then
begin
  assignfile(ff,OpenDialog1.FileName); {открываем исходный файл}
```

```
reset(ff,2); {для чтения двухбайтными словами}
xz:=filesize(ff); {размер файла в двухбайтных словах}
st:=OpenDialog1.FileName; {преобразуем название}
delete(st,length(st)-2,3);
st:=st+'txt';
assignfile(fo,st); {создаем txt}
rewrite (fo,1);
st:=''; {строка понадобится для вывода в Memo}
for i:=1 to xz do
begin
  blockread(ff,xw,1); {читаем слово}
  xofs:=hi(xw); {делим его на байты}
  xs:=lo(xw);
  if (xofs=0) then {наш алгоритм}
  begin
   if (xs<>0) then
   begin
    blockwrite(fo,xs,1);
    if xs=13 then
    begin
     Memo1.Lines.Add(st); {выводим в Memo каждую строку}
     st:='';
    end
    else st:=st+chr(xs); {дополняем строку}
   end;
  end else
  if (xofs=4) then
  begin
  if xs<>0 then
  begin
     xs:=xs+176;
     blockwrite(fo,xs,1);
     st:=st+chr(xs);
  end;
  end else
  begin
   blockwrite(fo,xs,1); {если не 0 и не 4 - пишем все}
   blockwrite(fo,xofs,1); {в файл, но не в строку}
  end;
end;
closefile(ff);
closefile(fo);
Memo1.SelStart:=Memo1.Perform(EM_LINEINDEX,1,0)+1;
```

```
  Memo1.Perform(EM_SCROLLCARET,0,0);
  Memo1.SetFocus;
end;
end; {Open}
```

Последние три оператора устанавливают наш невидимый курсор в начало текста, чтобы не потребовалось его сразу прокручивать — после загрузки строки указатель в Memo станет в самый конец файла, но не знаю, как вы, а я привык рассматривать файлы с начала. Остальное в принципе здесь должно быть все понятно — алгоритм мы описали раньше. Разумеется, писать на диск или нет по условию "старший байт не 0 и не 4" — дело ваше. Если не будете, то потеряете все символы типа фигурных скобок, но не это главное — в файле может оказаться и обычный, не Unicode, скрытый текст (вот так вот устроен Word!). Если вам это неважно, то при исключении этих операторов мусора в файле будет существенно меньше.

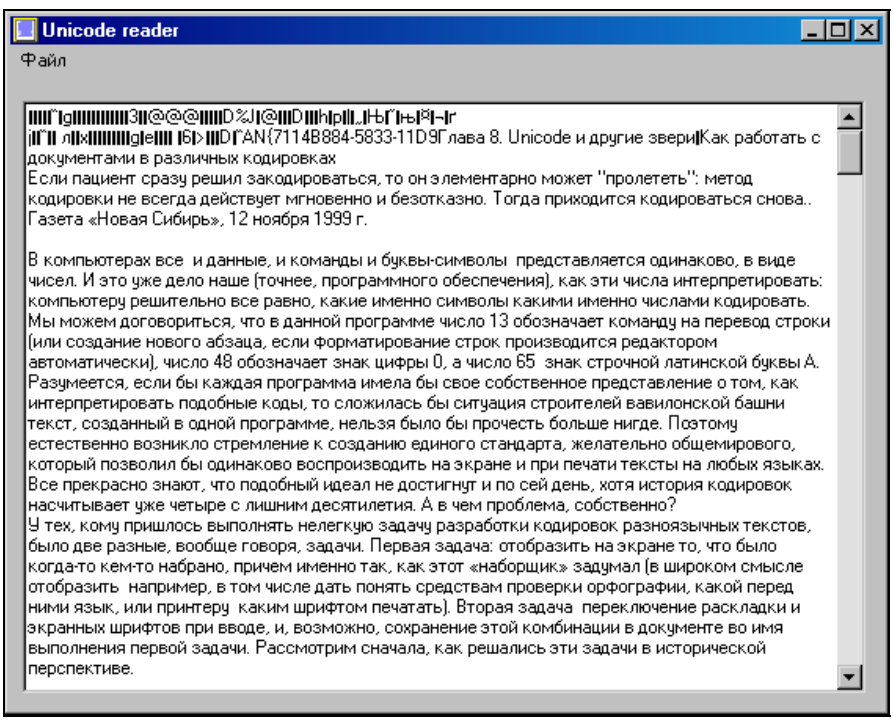


Рис. 8.2. Результат работы Unicode reader над текстом этой главы

Кстати, в Word, в отличие от текстовых форматов Microsoft, перевод строки (который здесь представляет собой конец абзаца) обозначается одним символом с номером 13 (CR), а не двумя (CR + LF), поэтому, чтобы выводить текст

в Memo постепенно, мы поставили условие вывода именно по этому знаку. Как будет выглядеть начало этой главы после загрузки ее в нашу программу, представлено на рис. 8.2. Одновременно создастся соответствующий текстовый файл. В конце текста можно найти много любопытного: отброшенные варианты, сведения об авторе и программе и пр.

Преобразование через WideString

А почему бы не воспользоваться средствами Windows и Object Pascal, которые позволяют читать двухбайтные символы? Вопрос только в том, как их преобразовать в однобайтные, и это придется делать уже на уровне строк — обычная (ANSI) строка может прямо приравниваться к "широкой" (WideString). Простой механизм преобразования на уровне одиночных символов мне не известен (приравнять тип Char к WideChar, как в случае строк, компилятор не позволит) — если читатели слышали о таком, пусть поделятся. Но и со строками в принципе получается даже проще, чем "вручную" — другое дело, что для отсеивания мусора все равно придется принимать отдельные меры.

Добавим к меню пункт **Файл | Unicode**, и напишем вот такой обработчик щелчка по этому пункту:

```
procedure TForm1.Unicode1Click(Sender: TObject);
var wch:WideChar;
 wst:WideString;
begin
If OpenDialog1.Execute then
begin
  Memo1.Lines.Clear;
  wst:='';
  assignfile(ff,OpenDialog1.FileName); {открываем исходный файл}
  reset(ff,sizeof(wch)); {для чтения двухбайтными символами}
  xz:=filesize(ff); {размер файла в символах}
  for i:=1 to xz do
  begin
   try
    blockread(ff,wch,1);
   except
    continue;
   end;
    if wch>#31 then wst:=wst+wch else
    if wch=#13 then
    begin
     st:=wst;
```

```
      Memo1.Lines.Add(st); {выводим в Memo каждую строку}
    end;
  end;
  closefile(ff);
end;
end;
```

Если мы выполним эту процедуру, мы увидим, что в результирующей строке теперь будет много "мусорных" символов в виде "?", которые придется — при желании — отсеивать отдельно. Главное же в том, что протекает все это много медленнее, чем раньше. Отмечу, что мы вообще тут читаем файл неправильно — побайтно (или даже по два байта) читать дисковый файл в Windows, где памяти всегда более чем достаточно, есть совершеннейшее "ламерство", и допустимо только, если мы читаем небольшой объем данных. Позже мы вернемся к этому вопросу подробнее, но боюсь, что в данном случае замедление происходит совсем по иной причине — ведь в обоих случаях методика чтения одинаковая.

Заметки на полях

Я еще со времен Turbo Pascal заметил, что "штатная" реализация многих простых функций в продуктах Borland тормозит программу по совершенно необъяснимым причинам. Базовая для многих графических операций функция `PutPixel`, которая окрашивает заданную точку DOS-экрана в нужный цвет, выполняется много медленнее, чем прямой ее аналог, написанный даже не на ассемблере, а обычными паскалевскими способами прямой записи в регистры портов и в память:

```
procedure outpix(x,y:word;col:byte);
var m:byte;
begin
   port[$3CE]:=5;
   port[$3CF]:=2;
   port[$3CE]:=8;
   port[$3CF]:=128 shr (x mod 8);
   m:=mem[$A000:y*80+x div 8];
   mem[$A000:y*80+x div 8]:=col;
end;
```

Я не буду пояснять здесь текст данной процедуры, это довольно долго, просто привожу ее как пример того, что самостоятельно написанный код не так уж и редко может работать лучше "штатного". Это, видимо, касается и использованных ранее манипуляций с "широкими" строками. Несложно попробовать написать собственную процедуру преобразования типа `WideChar` в `ANSIChar`, исключив преобразование "широкой" строки в обычную, а потом посмотреть, что именно тормозит в приведенном ранее алгоритме. Я не стал этого делать по одной простой причине: сама процедура чтения тогда получится вырожденной — фактически окажется, что мы используем тот же "ручной" алгоритм, что и раньше. Зачем тогда вообще связываться с "широкими" символами?

Проблема автоматического переключения раскладки в RichEdit

Создадим тестовый проект с именем RichEditText (модуль формы назовем edit.pas — на диске Glava8\2), положим на пустую форму компонент `RichEdit`, запустим его и наберем в нем несколько разноязычных строк. Если вы теперь будете клавишами управления курсором или мышью перемещать текстовый курсор между этими строками, то раскладка у вас также будет меняться. Интересно, что баг (bug) функционирует непоследовательно: если окно теряет фокус ввода, а потом его восстанавливает, то раскладка устанавливается та, что была по умолчанию установлена системным переключателем, а стоит курсор сдвинуть с места, если он на строке с другой раскладкой, она тут же переключится. Потерей-возвращением фокуса можно вернуть раскладку на место. Заметим, что этот баг широко известен, начиная по крайней мере с версии Delphi 2.0, корпорация Borland успела с тех пор пару раз сменить название, но он так и кочует из версии в версию[6] (уверен, что и в Delphi .NET он также наверняка имеется — просто не проверял). Вроде бы винить Borland особенно не за что, т. к. компонент этот есть просто ретрансляция класса того же названия (`RichEdit`) из Windows API. Но ведь справиться с этим багом квалифицированный программист может минут за пятнадцать, и, к тому же, это не единственное, что в `RichEdit` сделано достаточно "криво" — так, что без прямого обращения к API сколько-нибудь приличную программу на его основе сделать просто нельзя.

Для того чтобы понять, что происходит в `RichEdit` при навигации по многоязычному тексту, достаточно воспользоваться программой WinSight ("официальным инструментом хакера", как ее часто характеризуют), которая входит в поставку Delphi. "Поковырявшись" в сообщениях, мы обнаружим, что разница между событиями переключения языка самопроизвольно и по нашей команде заключается в том, что в первом случае появляется только сообщение `WM_INPUTLANGCHANGE`, во втором — сначала еще и наш любимый запрос *(см. главу 7)* `WM_INPUTLANGCHANGEREQUEST`. Это логично: запросу при автоматическом переключении просто неоткуда взяться, а сообщение

[6] Цитирую из статьи в одном компьютерном издании (название программы я изменил на нейтральное "Редактор"): *"Еще одна фирменная "фишка" — автоматическое переключение раскладки клавиатуры. "Редактор" определяет, на каком языке (русском или английском) вы пишете, и самостоятельно переключает клавиатуру. …К слову, "Редактор" — единственный в мире редактор, имеющий подобную встроенную функцию"*. Так вот, описанная там программа, видимо, потому и единственная в мире, что ошибку в реализации компонента выдает за фирменную особенность. В известном Punto Switcher аналогичная функция реализована сознательно и целенаправленно, и работает совершенно иначе (все равно крайне неудобно получилось, но это уже личное мнение автора).

WM_INPUTLANGCHANGE отправляется уже после того, как был изменен язык ввода. Изменен-то изменен, но в нашем праве пропустить это сообщение "мимо ушей" и ничего не переключать. Алгоритм исправления ситуации понятный: если запрос был — переключаем, если не было — игнорируем.

События можно в принципе отследить любым из способов: либо, как мы уже умеем, с помощью ловушки, причем тут дело значительно облегчается тем, что мы действуем в пределах окна программы и ловушка может быть локальной. Либо воспользоваться штатным способом переопределения метода OnMessage приложения. Однако на практике все это отладить довольно сложно: запрос WM_INPUTLANGCHANGEREQUEST перехватывается легко и непринужденно, а вот с последующим сообщением WM_INPUTLANGCHANGE могут возникнуть трудности. Однако есть изящное решение, которое, в том числе, позволяет перехватывать и фильтровать только нужные сообщения, а не все на свете. Заключается оно в том, чтобы создать свой собственный RichEdit на основе имеющегося, изменив в нем только реакцию на сообщение WM_INPUTLANGCHANGE[7]. Назовем его RichEditInt (international).

Так как "люди мы не местные", мы будем создавать компоненты вручную (точнее, в среде Delphi), а не с помощью ModelMaker (для чего его еще надо отдельно установить с диска Delphi). Имейте в виду, что компонент можно создавать и отлаживать либо "ручками", либо уж полностью через ModelMaker — мешать данные способы не получится. Это в Delphi разрешается править исходники в каком угодно редакторе, а ModelMaker — такая странная штука, которая в результате *компиляции* проекта выдает *исходный текст* (используемый потом в Delphi). И если вы внесете свои изменения отдельно, то придется делать это уже до конца, потому что при попытке загрузки исправленного варианта проекта ModelMaker эти ваши изменения даже не просто проигнорирует, а немедленно уничтожит, ничего не спрашивая. Кроме того, в некоторых отношениях использование ModelMaker, по личному мнению автора, не упрощает, а усложняет задачу — например, на первом этапе, когда надо определить класс-родителя. Но, конечно, у этой программы есть и свои преимущества (чем сложнее проект — тем больше преимуществ), так что, как говорится, на вкус и цвет... Если хотите ознакомиться с процессом использования ModelMaker получше — вам сюда [3].

Во всех деталях я описывать процесс создания компонента не буду (см., например, [21, 22], где все это описано очень подробно). Сначала выберем пункт **Component | New component** и заполним поля — в качестве родителя укажем TRichEdit, название присвоим, как договаривались, RichEditInt, а в

[7] Основная идея принадлежит Максиму Гуменюку (**http://delphiworld.narod.ru/base/richedit_change_lang.htm**.

пункте **Palette Page** установим вкладку **Win32** (хотя никто не запрещает завести и свою собственную — она потом создастся автоматически). Получим заготовку, которая после добавления надлежащего кода будет выглядеть так (обратите внимание на дополнительные модули в **uses**) — листинг 8.1.

Листинг 8.1. Модуль RichEditInt

```
unit RichEditInt;

interface

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

type
TLanguageMessage = procedure(Sender: TObject; Lang: HKL)
 of object; {процедура по событию смены языка}
TRichEditInt = class(TRichEdit) {регистрируем сам компонент}
private
 FOnLangChange: TLanguageMessage;
         {регистрируем событие компонента}
 procedure LangRequest(var Mesg: TMessage);
       message WM_INPUTLANGCHANGEREQUEST;
 procedure LangChange(var Mesg: TMessage);
        message WM_INPUTLANGCHANGE;
protected
 { Protected declarations }
public
 { Public declarations }
published
 property OnLangChange: TLanguageMessage
    read FOnLangChange write FOnLangChange;
     {свойство компонента отслеживать смену языка}
end;

procedure Register;

implementation

procedure TRichEditInt.LangRequest(var Mesg: TMessage);
{если пришел запрос}
begin
if assigned(FOnLangChange) then {на дальнейшую обработку его}
FOnLangChange(self, Mesg.LParam);
```

```
inherited;
end;

procedure TRichEditInt.LangChange(var Mesg: TMessage);
begin
{если только сообщение о смене языка}
Mesg.Result := 1; {игнорируем}
end;

procedure Register;
begin
RegisterComponents('Win32', [TRichEditInt]);
end;

end.
```

Для того чтобы проверить, все ли работает, временно добавить компонент к проекту RichEditText можно динамически. Для этого надо убрать с формы "настоящий" RichEdit и внести в модуль edit.pas вот такие изменения (также, конечно, следует внести модуль RichEditInt в предложение **uses**):

```
. . . . .
var
Form1: TForm1;
RichEditInt:TRichEditInt;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
 RichEditInt:=TRichEditInt.Create(Self);
 RichEditInt.Parent:=Self;
end;
. . . . .
```

После запуска на пустой форме появится квадратик редактора, в который вы можете вписать текст и убедиться, что теперь раскладка никак не зависит от передвижения курсора. Но, чтобы все было "по-взрослому", добавлять компонент можно было бы выбором из палитры на этапе конструирования формы, нам надо теперь зарегистрировать компонент в Delphi по-настоящему.

По умолчанию Delphi создала новый компонент в папке ..Delphi7\Lib, там мы его и оставим (на прилагаемом диске он находится в папке Glava8\2\Lib). Для регистрации необходимо, прежде всего, создать ресурсный файл с иконкой

компонента в виде Bitmap 24×24. Для этого откроем наш любимый Image Editor, создадим новый ресурсный файл, а в нем — ресурс типа BITMAP 24×24 в 16 цветов и раскрасим его (я сделал иконку, похожую на обычную для RichEdit, только попроще). BITMAP (внимание!) обязательно нужно назвать именем класса, причем прописными буквами (TRICHEDITINT), а ресурсный файл — тем же именем, что и модуль компонента, но с расширением dcr (RichEditInt.dcr). Сохранить ресурсный файл надо в той же папке, что и исходный текст модуля компонента.

Теперь можно регистрировать, причем имейте в виду, что при любых ошибках в этом процессе все нужно будет делать заново (так, если вам не понравится иконка, или она будет отображаться как-то не так, то недостаточно ее перерисовать в ресурсном файле, придется заново компилировать компонент, возможно, также и удалять зарегистрированный компонент из палитры и заново устанавливать). Запустите Delphi, войдите в меню **Component | Install Component** и перейдите на закладку **Into New Package** (впрочем, никто не мешает и включить его в существующий пакет, по умолчанию это dcluster.dpk). При создании нового пакета нельзя ему присваивать имя, совпадающее с именем модуля (оно у нас RichEditInt.pas), иначе получите в ответ кучу ругательств на дельфийском языке. После ввода имени (вместе с путем к файлу) надо нажать на **Ok**, и в появившемся окне с новым пакетом (package) нажать сначала на **Compile**, а затем — если все пройдет успешно — на **Install**. Уверяю, что вся процедура даже по первому разу больше двух часов у вас не отнимет.

Все, компонент создан, зарегистрирован и находится на вкладке **Win32** — можете наслаждаться обновленным RichEdit. Уберите из проекта RichEditText ранее вставленные строки для тестирования (в проекте на диске они закомментированы), поместите на форму RichEditInt и изучите закладку **Events** в Object Inspector — у вас появилось ранее не существовавшее событие OnLangChange, которое, кстати, можно использовать при надобности.

Автоматическое определение кодировки текстовых файлов

Так получилось, что проекты в этой главе — пока чисто демонстрационные (кроме, конечно, создания компонента RichEditInt). Просто трудно придумать достаточно короткий проект реальной программы с тем, чтобы не отвлекаться от сути излагаемой проблемы. Мы еще совершим в этой главе полезное дело — доделаем, насколько получится, убогий перекодировщик из *главы 5*. А пока вот еще один пример, который мы оформим в виде законченной программы, но до поры до времени он ничего особенно практически полезного делать не будет.

Вы не раз встречали программы, которые автоматически определяют кодировку текстового документа (здесь речь идет только об однобайтных кодировках, потому что в чистых текстах Unicode фактически не употребляется). На практике для русскоязычных текстов достаточно распознавать три вида кодировок: CP866, Win1251 и KOI-8 (будем их так сокращенно называть, в *приложении 3* даны более официальные названия), а также "до кучи" отличать английский текст от русского. Как же это делается?

Принцип заключается в том, что производится статистическая проверка текста на встречаемость различных сочетаний букв, и он отлично изложен в статье [23], на которую мы и будем опираться[8]. Автор статьи, однако, на мой взгляд, чересчур усложнил алгоритм, разбирая, например, редко встречающиеся случаи наличия текста с псевдографикой (вполне можно допустить приемлемый процент ошибок при определении кодировки), а также в стремлении заставить алгоритм работать быстрее. Ускорение процесса достигается за счет того, что просматривается не весь файл, а лишь достаточно большой фрагмент его. Что такое "достаточно большой"? На этот вопрос точного ответа нет, а т. к. мы будем иметь дело лишь с текстовыми документами, которые обычно имеют небольшой объем, то на современных машинах это не очень актуально, и ведет лишь к увеличению риска определить что-то неправильно. Мегабайтный текстовый файл даже при "ламерском" побайтном чтении с диска современные машины "перелопачивают" за доли секунды. А когда в дальнейшем пойдет речь о создании реально работающей программы, мы найдем "правильный" способ ускорения процесса *(см. об этом далее)*.

Поэтому мы сделаем так: будем просто читать каждый файл, как бинарный, т. е. содержащий только коды символов, затем преобразовывать эти коды в символы в соответствии с таблицами кодов, и вести статистику для каждой разновидности кодировок — соответствуют ли двухбуквенные сочетания допустимым в русском языке. При этом, естественно, какое-то количество недопустимых сочетаний все равно встретится — не только из-за орфографических ошибок, но и в каких-нибудь, например, аббревиатурах ("***ПБ***ОЮЛ"), кальках с иностранных языков ("к***эб***", "лао ***цз***ы", "Давидо***фф***") и т. п. Далее мы будем сравнивать вычисленные проценты появления допустимых сочетаний друг с другом и без всяких ухищрений решать, что кодировка та, в которой процент этот больше. Ну, а английский текст отличить проще простого — у него подавляющая часть кодов будет меньше 128. Все это не до конца корректно — например, может встретиться какая-нибудь неизвестная кодировка, или вообще бессмысленный набор знаков (скажем, письмо при двой-

[8] Описываемый принцип, хотя и требует известной предварительной подготовки, более надежен и эффективен, чем изложенный на официальном русскоязычном сайте Microsoft MSDN, см. **http://www.microsoft.com/Rus/Msdn/Activ/MSVB/Archive/VBA/Decoder/254.mspx**.

ной перекодировке Win-KOI-Win), и тогда корректно было бы признаться, что кодировку мы не знаем. Но, во-первых, если мы текст будем отображать, то и так будет видно по результату, что программа не справилась. А во-вторых, "перелопатив" по этому алгоритму все текстовые файлы, которые встретились на диске C: автора (более 10 тысяч), программа ошиблась только в нескольких случаях, когда русского текста было немного среди большого количества английского текста и числовых значений.

Создадим новый проект под названием Kodirovka.dpr (модуль формы назовем code.pas — на диске в папке Glava8\3). Программа, которую мы сделаем, будет просматривать все файлы в заданном каталоге, фильтровать их на предмет принадлежности к текстовому формату и определять кодировку. Прежде, чем приступать к делу, надо составить таблицы кодировок — хотя, конечно, для Win1251 и cp866 можно использовать и функции API `OEMtoANSI` и `ANSItoOEM`, но уже для KOI-8 такой возможности нет, поэтому для единообразия я делаю все вручную (а вдруг вы захотите еще и другими кодировками дополнить?). Для этого пронумеруем русские буквы подряд от 1 до 32 (отбросив "Ё") и поставим их в соответствие номерам символов от 128 до 255 в различной кодировке. Заглавные и строчные варианты будем считать за одну и ту же букву. Если номеру не соответствует никакая буква, то ставим в этом месте 0 (см. далее исходный текст модуля code.pas).

Но самое главное — составить таблицу недопустимых сочетаний букв. К счастью, мы живем в эпоху компьютеров и заниматься лингвистическими изысканиями долго нам не придется — необходимо просто обработать на этот предмет достаточно большое по объему количество текстов, желательно литературных. Автор статьи [23] утверждает, что обработал некий 800-мегабайтный файл, ну а автор этих строк проверил его результаты на нескольких текстах из Библиотеки Мошкова и убедился, что все практически совпадает. Результирующая таблица представляет собой матрицу 32×32 (буква "Ё" опущена), в ячейках которой стоит 1 — если буквы на пересечении сочетаются, и 0 — если нет (причем ведущие буквы сочетаний расположены в строках матрицы, а ведомые — в столбцах).

Задача отнесения к текстовому формату формально не решается, но на практике можно применить следующий алгоритм: следует отсеять все файлы, в которых хоть один раз встретится символ с нулевым значением. Картинки, программы, базы данных, DOC-файлы в формате Word 97/200/XP, XLS-файлы и т. п. чаще всего содержат поля с нулевым значением байт, а в текстовом формате это исключено. В достаточно больших архивах нулевые байты также обязательно встретятся. Но для надежности и ускорения работы мы дополнительно введем, во-первых, фильтр по часто встречающимся расширениям заведомо не текстовых форматов (.doc .rtf .exe .dll .drv .sys .ax .bin

.acm .vxd .bmp .jpg .jpeg .gif .tiff .psd .psp .pdf .zip .rar .pcb .pdb .sch .tbb .tbi .msi .wav .mp3 .avi .ovl — сколько фантазии хватит), во-вторых, будем рассматривать только файлы размером не более 500 Кбайт. Практика показывает, что при таком подходе ошибки практически исключены и алгоритм работает достаточно быстро.

Итак, установим на форму компонент `Panel`, установим его свойство `BorderStyle` в `bsSingle`, а `Color` — в `clWhite`, очистим заголовок `Caption`. Растянем ее на всю форму. Затем на нее установим `RichEdit` (хотя было бы здóрово испытать наш вновь созданный `RichEditInt`, но здесь это безразлично, т. к. мы вручную ничего вводить не будем, и в целях воспроизводимости примера у всех читателей я поставил обычный `RichEdit`). У него нужно сделать следующие установки: свойство `BorderStyle` в `bsNone`, `ParentColor` в `True` (тогда редактор сольется с фоном), `ScrollBars` в `ssVertical`, `ReadOnly` в `True`. Растянем `RichEdit1` так, чтобы вверху панели оставалось свободное пространство. Далее установим на этом пространстве однострочный редактор `Edit`, растянув его так, чтобы справа и слева оставалось место, и не забудем установить его свойство `AutoSelect` в `False` — мелкий белый текст на синем фоне совершенно неразборчив, и убирать выделение по умолчанию лучше во всех случаях, кроме самых необходимых *(см. об этом также в главе 9)*. Правда, мы к тому же будем убирать выделение программно, устанавливая курсор на конец строки, но и специальная установка также не помешает. В свойство `Text` введем начальную строку `C:\`.

Справа от `Edit1` поставим кнопку `Button1`[9] и изменим ее заголовок на "**...**", установив шрифт (свойство `Font`) пожирнее (Arial 18 кегля полужирный). Ниже поставим еще одну кнопку `Button2`, растянем ее подлиннее, также установим шрифт побольше (Arial 12 кегля полужирный), и изменим заголовок на **Искать!**. Левее установим еще одну кнопку `Button3` с тем же шрифтом (можно просто клонировать `Button2` через Copy-Paste), сделаем ее покороче, установим заголовок **Отмена** и свойство `Enabled` в `False`. В свойствах формы установим `BorderStyle` в `bsSingle`, а также удалим кнопку для распахивания окна на весь экран (свойство `BorderIcon | biMaxsimize` в `False`), чтобы не возиться с "якорями", и при этом у пользователя не возникло бы желания изменять размеры формы. Позицию формы при запуске (`Position`), как всегда, поменяем на `poDesktopCenter`, заголовок заменим на **Определение кодировки**, и заменим иконку на свою (folder04.ico на диске).

---

[9] Имейте в виду, что позиционирование компонентов относительно друг друга с помощью мыши может быть слишком грубым. Для того чтобы точно выровнять положение и подогнать размеры компонентов, необходимо вручную подбирать значения полей `Top` и, возможно, `Height` (`Width`) в Object Inspector.

Вам может показаться, что я слишком много вожусь с установками для демонстрационного примера, однако позднее мы используем этот проект для более практической цели, и потому лучше все сделать сразу. Мало того, для украшения программы (и не лень ведь!) поставим в левый верхний угол панели компонент `Image` и вставим в него картинку, пародирующую стиль заголовка поисковика Google (google.bmp на диске). Для полноты ощущений ниже установим компонент `Label`, покрасим его в бледно-голубой цвет, и установим его свойство `AutoSize` в `False`, очистим заголовок, на будущее установим шрифт пожирнее (также Arial 18 кегля полужирный) и растянем компонент во всю длину экрана. То, что получилось в результате, вы можете видеть на рис. 8.3.

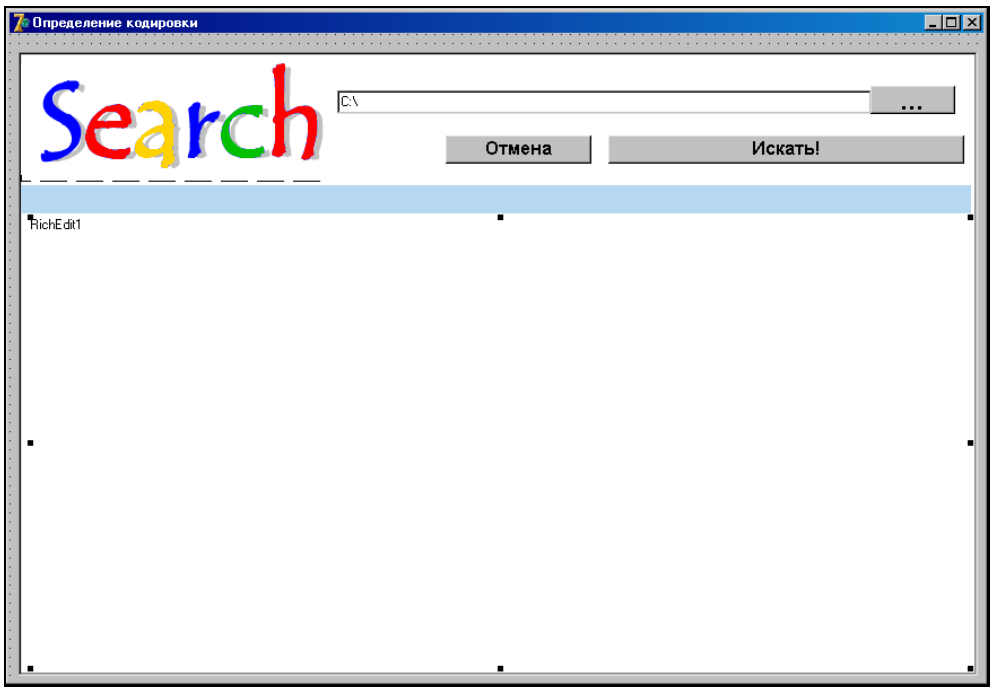


Рис. 8.3. Форма для поиска текстовых файлов и определения их кодировки

Заметки на полях

"Приколы", подобные нашему заголовку а-ля Google, всегда лучше делать с помощью картинок, а не с использованием различных компонентов для вывода фигурного текста, просто по той причине, что компонент использует системные шрифты, и вы ограничены выбором из трех-пяти обязательных их разновидностей, остальных в системе пользователя просто может не оказаться. Да и выбор эффектов несравненно больше и их проще реализовать.

Далее напишем обработчик события показа формы для установки "правильного" начального фокуса ввода, начальных свойств RichEdit, а также добавим процедуры скрытия текстового курсора для RichEdit:

```
procedure TForm1.FormShow(Sender: TObject); {при запуске}
begin
  RichEdit1.Lines.Clear; {очистка RichEdit}
  with RichEdit1.DefAttributes do
  begin
   Name:='Times New Roman'; {начальные атрибуты}
   Style:=[fsBold];
   Size:=12;
  end;
  Edit1.SetFocus; {фокус на Edit1}
  Edit1.SelStart:=length(Edit1.Text);
  Edit1.SelText:=''; {курсор в конец текста Edit1}
end;

procedure TForm1.RichEdit1Change(Sender: TObject);
begin
  HideCaret(RichEdit1.Handle); {скрываем курсор}
end;

procedure TForm1.RichEdit1MouseDown(Sender: TObject; Button:
                TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
  HideCaret(RichEdit1.Handle); {скрываем курсор}
end;
```

Общий план действий такой: пользователь жмет на кнопку "**...**", устанавливает папку, в которой будет производиться поиск (или вводит ее вручную), потом жмет на **Искать!**, и дальше программа работает по нашему алгоритму, выводя результаты построчно в RichEdit. Для этого нам надо сначала решить проблему ввода имени папки через вызов диалога — тот самый вопрос, который мы в проекте SlideShow из *главы 2* временно решали только через открытие конкретного файла. Есть много разных способов это сделать: так, Delphi предлагает компоненты типа DirectoryOutline, ShellTreeView и аналогичные (закладка **Samples** — кстати, во встроенной справке о них ни слова, так что если хотите ознакомиться, читайте [3, 4]). Кто хочет, может использовать и их, однако автора раздражают некоторые "фичи" — так, упомянутый ShellTreeView при запуске программы, и даже при запуске Delphi с соответствующим проектом, упорно дребезжит флоппи-дисководом, пытаясь прочесть несуществующую дискету (справедливости ради надо отметить, что в

Windows XP не так назойливо, как в Windows 98). Есть и другие неоправданные сложности. Конечно, их можно, подобно RichEdit, модернизировать (и сделать это в данном случае проще простой правкой исходного текста и последующим перекомпилированием, т. к. в папке C:\Program Files\BORLAND\Delphi7\Demos\ShellControls лежит полный исходный код модуля ShellCtrls), и автор [4] даже уже отчасти провел за нас эту полезную работу. Но мы в самом начале договаривались, что будем использовать стандартные средства Delphi до последней возможности, так что от модернизированных компонентов откажемся. Я покажу оптимальный, на мой взгляд, вариант использования "родных" компонентов для задания папки в следующей главе. До кучи следует упомянуть и способ самостоятельного формирования списка папок с помощью ListBox, которым, очевидно, и пользуются все продвинутые программисты, но мне представляется, что выполнение этой задачи для такой цели, как просто задать текущую папку, — чересчур уж громоздкая процедура.

А здесь мы применим более простой способ: в модуле FileCtrl (его исходный текст находится в папке C:\Program Files\BORLAND\Delphi7\Source\Vcl) определены целых две функции с одинаковым названием SelectDirectory, различающиеся синтаксисом. Функция, вызываемая по варианту

```
SelectDirectory (const Caption: string; const Root: WideString; out
Directory: string): Boolean;
```

показывает специальное окно для выбора именно папки и во всем удобна, кроме того, что автору так и не удалось заставить окно это возникать в нужном месте экрана. А вторая функция (синтаксис см. далее) вызывает общий стандартный диалог Windows (хорошо знакомый еще по Windows 3.*x*) с несколько более расширенной функциональностью, чем нам требуется — кроме собственно окна для выбора папки, она показывает еще и окно с файлами, которые в этой папке содержатся. Если есть желание отполировать программу до предела, можно и самим вызывать диалог типа SHBrowseForFolder прямо через API, но это довольно громоздкое занятие, которое, на мой взгляд, ничем не оправдывается — посмотрите описание этой функции с примером использования в первоисточнике [14][10]. Я просто применял эту вторую функцию, которая к тому же сразу комплектует диалог полезным окошком выбора диска (рис. 8.4).

Крупнейшим недостатком данного способа является то, что вызываемый диалог англоязычный, и исправить это положение, к сожалению, трудно — тогда проще использовать все же указанные ранее компоненты Delphi.

---

[10] Кстати, в Windows.pas этой функции нет, так что придется еще и вызывать ее прямо из shell32.dll, формировать нужные структуры — в общем очень трудоемкий процесс.

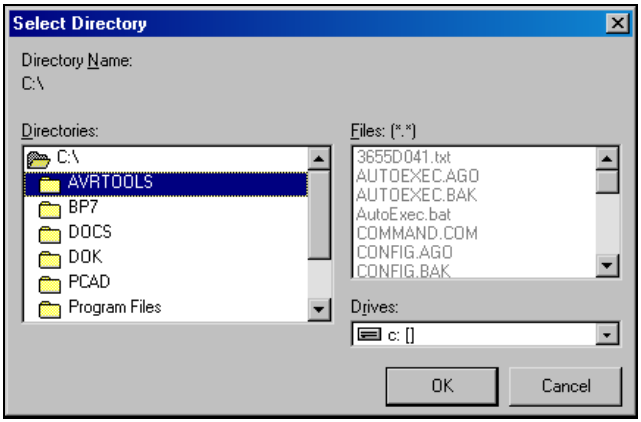


Рис. 8.4. Диалог Windows для выбора папки

Принципиальные вопросы мы решили, так что приступим: добавим в предложение **uses** модуль FileCtrl и объявим переменную `stpath:string`. После этого напишем обработчик щелчка на кнопке `Button1`:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
ChDir(Edit1.Text); {устанавливаем текущую директорию и
          вызываем диалог установки директории:}
if SelectDirectory(stpath,[],0) then Edit1.Text:=stpath;
end;
```

Осталось самое главное: собственно реализовать алгоритм поиска. Для этого мы объявим следующие константы и переменные (сверху закомментированы номера русских букв подряд — так легче составлять таблицы перекодировки):

```
. . . . .
{А=1 Б=2 В=3  Г=4 Д=5 Е=6 Ж=7 З=8  И=9 Й=10 К=11  Л=12 М=13 Н=14 О=15
П=16 Р=17 С=18 Т=19  У=20 Ф=21 Х=22 Ц=23  Ч=24 Ш=25 Щ=26 Ъ=27  Ы=28 Ь=29
Э=30 Ю=31 Я=32}

const alt: array [128..255] of byte =
(1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,
17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,
1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
```

```
const win: array [128..255] of byte =
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,
17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,
1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,
17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32);

const koi: array [128..255] of byte =
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
31, 1, 2,23, 5, 6,21, 4,22, 9,10,11,12,13,14,15,
16,32,17,18,19,20, 7, 3,29,28, 8,25,30,26,24,27,
31, 1, 2,23, 5, 6,21, 4,22, 9,10,11,12,13,14,15,
16,32,17,18,19,20, 7, 3,29,28, 8,25,30,26,24,27);

const chardouble: array [1..32,1..32] of byte =
((1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1),
(1,1,1,1,1,1,1,0,1,0,1,1,1,1,1,0,1,1,1,1,0,1,1,1,1,1,1,1,1,0,1,1),
(1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,0,0,1),
(1,1,1,1,1,1,0,0,1,0,1,1,1,1,1,0,1,1,1,1,0,0,0,1,1,0,0,0,0,0,0,0),
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,0,1,1),
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1),
(1,1,0,1,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,0,1,0,0,0,1,0,0,0,0,1,0,0,0),
(1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,0,1,1,1,1,1,1),
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1),
(0,0,0,1,1,1,0,1,0,0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,1),
(1,0,1,0,0,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,1,0,0,1,0,1,0,0,0,0,0,1,0),
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,0,1,0,1,1,1,0,1,1,0,1,1),
(1,1,1,1,1,1,0,0,1,0,1,1,1,1,1,1,1,1,1,0,1,0,1,1,1,1,1,0,0,1,1,1,1,0,1),
(1,1,1,1,1,1,1,1,1,0,1,0,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1),
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1),
(1,0,0,0,0,1,0,0,1,0,1,1,0,1,1,1,1,1,1,1,1,0,0,1,1,1,1,0,0,1,1,1,1,1,1),
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1),
(1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1),
(1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1),
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,0,0,0,1,1,1),
(1,0,0,0,0,1,0,0,1,0,0,1,1,1,1,0,1,1,1,1,0,0,0,0,0,0,0,1,0,0,1,0),
(1,0,1,1,1,1,0,0,1,0,1,1,1,1,1,1,1,1,1,1,0,0,0,0,1,0,0,0,0,1,0,0),
(1,0,1,0,0,1,0,0,1,0,1,1,1,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0),
```

```
(1,0,1,0,0,1,0,0,1,0,1,1,1,1,1,0,1,0,1,1,1,0,0,0,1,0,0,0,1,0,0,0),
(1,0,1,0,1,1,0,0,1,0,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,0,0,0,0,1,0,1,0),
(1,0,0,0,0,1,0,0,1,0,0,0,0,1,1,0,1,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0),
(0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1),
(0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,0,0,0,0,1),
(0,1,1,1,1,1,0,1,1,0,1,0,1,1,1,0,0,1,1,0,1,1,1,1,1,1,0,0,1,0,1,1),
(0,0,0,1,0,1,0,1,0,0,0,1,1,1,0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0),
(0,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,0,1,1,1,1,1,0,0,0,0,1,0),
(0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1,0,0,0,0,1,1));


const stExt: string =
('.doc .rtf .exe .dll .drv .sys .ax .bin .acm .vxd .bmp .jpg .jpeg .gif
.tiff .psd .psp .pdf .zip .rar .pcb .pdb .sch .tbb .tbi .msi .wav .mp3
.avi .ovl');

var
Form1: TForm1;
stpath,stsearch,fname,st,st1:string;
fd:file;
sf:TsearchRec;
nfile,ns1,ns2,nx,altn,winn,koi8:integer;
xw:word;
xs,xofs:byte;
FlagCancel:byte=0;
. . . . .
```

Константа chardouble представляет собой ту самую таблицу допустимых сочетаний букв, которую мы обсуждали ранее. Обработчик щелчка на кнопке Button2 (**Искать**) приведен далее. В нем мы учли, что имя папки пользователь может ввести вручную или через буфер обмена, следовательно, с возможными ошибками, которые мы постараемся исправить по максимуму.

```
procedure TForm1.Button2Click(Sender: TObject);
{ищем файлы и обрабатываем их}
begin
FlagCancel:=0; {флаг отмены}
Button3.Enabled:=True; {активируем Отмену}
Button2.Enabled:=False; {дезактивируем Поиск}
stpath:=Edit1.Text; {название папки}
if stpath='' then exit; {если оно пустое - сразу на выход}
while stpath[1]=' ' do delete(stpath,1,1);
      {удаляем возможные пробелы в начале}
if stpath='' then exit; {если пусто — сразу на выход}
```

```
if stpath[length(stpath)]='\' then
delete(stpath,length(stpath),1);
{удаляем концевой знак '\', если он есть}
try
 ChDir(stpath); {проверяем, есть ли такая папка}
except
  Application.MessageBox('Несуществующая
           директория','Error',MB_OK);
  exit; {если нет - на выход}
end;
RichEdit1.Lines.Clear; {очищаем поле результатов}
Form1.Label1.Caption:=''; {и заголовок Label}
Application.ProcessMessages; {чтобы сразу сработало}
stsearch:='\*';
stsearch:=stpath+stsearch; {строка для поиска файлов}
nfile:=0;
if FindFirst(stsearch,$23,sf)=0 then
{ $23 = не просматриваем системные файлы, тома и каталоги}
repeat
  if FlagCancel<>0 then break; {если флаг отмены не 0, то
                прерываем}
  fname:=stpath+'\'+sf.Name; {полное имя файла}
  {все к одному регистру:}
  if pos(ExtractFileExt(ANSIUpperCase(fname)),
        ANSIUpperCase(stExt))<>0 then continue;
   {если расширение совпадает с запрещенным, то на выход}
  if not ReadFileFormat then continue;{читаем файл}
  nfile:=nfile+1; {что-то получили}
  Form1.Label1.Caption:='Найдено:';
  st:=' кодировка:'+st;
  st1:=IntToStr(nfile)+'. '+fname; {номер найденного файла}
  with RichEdit1 do {выводим красиво в RichEdit}
  begin
   Lines.Add(st1);
   SelAttributes.Style:=[fsItalic];
   SelStart:=perform(EM_LINEINDEX,Lines.Count,0);
   SelText:=st;
   perform(EM_SCROLLCARET,0,0);
   SelAttributes.Style:=[fsBold];
  end;
  Application.ProcessMessages; {чтобы все прокрутилось}
until FindNext(sf)<>0; {пока файлы не закончатся}
with RichEdit1 do {заключающую строку}
```

```
begin
   Lines.Add(' ');
   st:='Просмотрено '+IntToStr(nfile);
   Lines.Add(st);
   SelStart:=length(Text);
   SelText:='';
   SetFocus; {иначе скроллинга не будет}
end;
  Form1.Edit1.SetFocus; {возвращаем фокус в Edit1}
  Edit1.SelStart:=length(Edit1.Text);
  Edit1.SelText:=''; {курсор в конец текста Edit1}
FindClose(sf); {конец поиска}
Button2.Enabled:=True; {активируем Поиск}
Button3.Enabled:=False; {дезактивируем Отмену}
end; {Button2}
```

Самодеятельную процедуру удаления возможных пробелов в начале строки я использовал тут в иллюстративных целях "как это делается вообще" — в последних версиях Delphi есть три универсальные функции: `Trim`, `TrimLeft` и `TrimRight`, которые удаляют пробелы и служебные символы (читай: знаки табуляции и концы строк) с обоих концов, в начале и в конце строки соответственно. Мы их используем в дальнейшем.

Собственно статистику мы будем вычислять в функции `ReadFileFormat`, текст которой приведен далее, а пока несколько комментариев. Функция `ExtractFileExt` возвращает расширение с ведущей точкой, поэтому у нас в строке с запрещенными расширениями они также представлены с ведущей точкой. Все процедуры со строками, в отличие от процедур с папками и файлами, чувствительны к регистру, поэтому при сравнении мы использовали на всякий случай приведение к одному регистру (верхнему) с помощью `ANSIUpperCase`. Все, что относится к `RichEdit`, я подробно комментировать не буду, потому что эта "песня" тянет на отдельную книгу. В нем ничего никогда не работает так, как вы ожидаете. Тут мы захотели выделять часть строки отдельным шрифтом (имя файла жирным, а кодировку курсивом), и к тому же прокручивать окно по мере поступления строк. В принципе мы добились, чего хотели, но к самой первой строке курсивный текст добавляется почему-то отдельной строкой — если хотите, можете с этим побороться самостоятельно. Скорее всего, нужно использовать процедуру замены выделенной части текста через сообщение `EM_REPLACESEL`, но я оставил вам это в качестве домашнего задания — вывод в `RichEdit` мы все равно в дальнейшем заменим на другой способ отображения.

А вот, наконец, основная функция `ReadFileFormat`, ради которой все и затевалось:

```
function ReadFileFormat: boolean;
begin
result:=True;
assignfile(fd,fname);
try
  reset(fd,2); {пробуем открыть - вдруг он занят}
except
  result:=False;
  exit;  {тогда на выход}
end;
if filesize(fd)>250*1024 {если размер больше 250К 2-байтных слов}
then begin result:=False; closefile(fd); exit; end; {на выход}
  altn:=0; {в этих переменных будем накапливать статистику}
  winn:=0;
  koi8:=0;
  while not eof(fd) do
  begin
   blockread(fd,xw,1);
   xofs:=hi(xw);
   xs:=lo(xw);
   if (xs=0) or (xofs=0) then {если хоть один символ=0}
   begin result:=False; closefile(fd); exit; end; {на выход}
   if (xofs>127) and (xs>127) then
       {только русские двухбуквенные сочетания}
   begin
    {проверка ALT}
    ns1:=alt[xs];
      {номер символа по алфавиту, соотв. кодировке ALT}
    ns2:=alt[xofs];
      {номер символа по алфавиту, соотв. кодировке ALT}
    if (ns1<>0) and (ns2<>0) then
    if chardouble[ns1,ns2]<>0 then altn:=altn+1;
    {проверка WIN}
    ns1:=win[xs];
      {номер символа по алфавиту, соотв. кодировке WIN}
    ns2:=win[xofs];
      {номер символа по алфавиту, соотв. кодировке WIN}
    if (ns1<>0) and (ns2<>0) then
    if chardouble[ns1,ns2]<>0 then winn:=winn+1;
    {проверка KOI}
    ns1:=koi[xs];
      {номер символа по алфавиту, соотв. кодировке KOI}
    ns2:=koi[xofs];
      {номер символа по алфавиту, соотв. кодировке KOI}
```

```
    if (ns1<>0) and (ns2<>0) then
    if chardouble[ns1,ns2]<>0 then koi8:=koi8+1;
   end;
  end;
  nx:=MaxIntValue([altn,winn,koi8]);
              {максимальное из полученного}
  if (nx<(filesize(fd) div 10)) or (nx=0) then st:=' ASCII'
  {если осмысленных сочетаний не больше 5% от объема,
                    то это английский}
  else if nx=koi8 then st:=' KOI-8' {иначе или KOI8}
  else if nx=winn then st:=' Win1251' {или Win}
  else if nx=altn then st:=' cp866' {или DOS}
  else st:=' unknown';
  {что-то другое - теоретическая ситуация,
                  которой быть не должно}
closefile(fd);
end;
```

Отметим, что, вообще говоря, это типичный пример того, как делать *не надо*. Причем не просто не надо, а *очень не надо*. Для простоты и наглядности мы сделали программу в стиле Turbo Pascal — в DOS с ее 640 Кбайт памяти приходилось все, по возможности, держать на диске и подгружать в память только тогда, когда данные нужны непосредственно в текущей операции. Нечего и думать было о том, например, чтобы загрузить файл в память целиком, а потом что-то с ним делать — "out of memory" вам было обеспечено. В Windows с ее 4 Гбайт виртуальной памяти это, разумеется, непростительная глупость — у меня на компьютере стоит 512 Мбайт ОЗУ, которые даже прожорливой Windows XP обычно используются едва на треть, и туда уместится практически любой файл. И дело тут не в самом по себе быстродействии дисковой подсистемы. Можете себе представить, что происходит, когда мы задаем чтение очередных двух байт: процедура обращается к драйверу, драйвер — к контроллеру, контроллер шарит по таблице расположения файлов в начальной области диска, находит базовый кластер, откуда начинается наш файл, потом находит по цепочке связанных ссылок конкретный кластер, где хранятся нужные данные, потом переносит головки туда, считывает этот кластер (или даже несколько их целым блоком — в современных дисках с 7200 об/мин никто отдельными кластерами читать не будет, слишком неэкономично) в кэш диска, потом драйвер переносит эти данные в оперативную память, выбирает из них нужные нам два байта — и так *каждый* раз! Можно, конечно, было бы считать файл в строку через ту же `blockread` одним блоком (величина файла при поиске известна), или по-

ручить все это системе. К данному важному вопросу мы еще не раз будем возвращаться *(см. главы 14 и 21)*. Когда мы ранее читали DOC-файл, мы так же отмечали этот момент, но там это было не так критично, потому что файл был всего один.

Результаты работы программы над папкой с некоторыми документами автора приведены на рис. 8.5. Всего в папке было 280 файлов, большую часть которых составляли, естественно, документы Word, так что в результирующем списке оказалось всего 59 наименований. На машине с процессором Athlon 1700 МГц и диском 7200 об/мин процесс занял 30 секунд — кошмар!

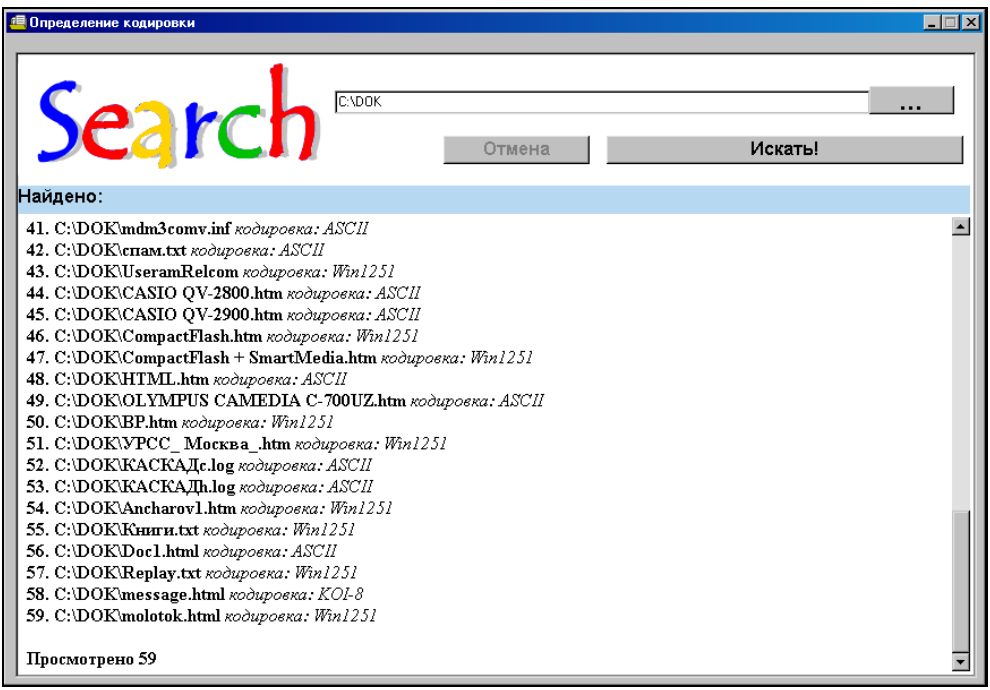


Рис. 8.5. Результаты работы программы Kodirovka

В дальнейшем мы будем улучшать эту программу: введем в нее линейку-индикатор процесса поиска, добавим просмотр вложенных папок, возможность обращения к файлам прямо из программы, избавимся от опостылевшего `RichEdit`, ускорим чтение и, в конце концов, преобразуем ее в локальный поисковик, который, разумеется, до уровня Google Desctop Search дотянуть не удастся, но, тем не менее, он будет вполне работоспособен.

Форматы в буфере обмена (попытка доработки перекодировщика)

В принципе, мы теперь знаем путь, на котором можно довести до работоспособности в Windows XP перекодировщик Layout из *главы 5* — можно просто включить в него переключатель раскладки для того приложения, в котором происходит перекодировка. Тогда русский текст должен вставляться нормально. Однако это "лобовое" решение довольно некрасиво — я, например, терпеть не могу никаких автоматических переключателей раскладки. Очевидное решение заключается в том, чтобы как-то разместить в буфере обмена вместе с текстом указание на то, что он русский. Только как это сделать? Вот очень изящный метод специально "для чайников": разместить на форме невидимый `RichEdit` и "пропустить" текст через него, т. е. вставить туда текст, объявить ему нужный `Charset` и затем снова забрать в буфер методом `RichEdit.CopyToClipboard` — он сам за вас все сделает. К сожалению, это нам не подходит, т. к. у нас нет формы. Наиболее капитальный метод — разместить в буфере, кроме простого текстового формата, еще и тот же текст в формате RTF — приложение само выберет то, что нужно. Оба этих способа приведены в самой большой, наверное, из существующих, подборке FAQ по Delphi на русском языке [24].

Но совсем зря, что ли, в Windows используется Unicode? Оказывается нет, но размещать в буфере обмена текст только в Unicode следует исключительно для семейства NT — в 9*x* либо не получите ничего вообще, либо получите полную чушь вплоть до обрушения системы. Поэтому надо либо опять же "городить" свой буфер обмена с множеством форматов, либо усложнять программу и определять, к какому семейству принадлежит установленная Windows. Последнее может дать поле `dwPlatformId` структуры `OSVERSIONINFO`, которую можно получить через функцию `GetVersionEx`. В пособии (вы уже, наверное, привыкли, что "пособием" без уточнений я называю официальный сайт [14]), написано, что перед вызовом этой функции надо установить размер структуры (т. к. он, видимо, от версии к версии может различаться). То есть в нашем перекодировщике надо соорудить вот такое условие [19]:

```
. . . . .
var
Version: TOSVersionInfo;
. . . . .
<перекодируем строку>
Version.dwOSVersionInfoSize := SizeOf(TOSVersionInfo);
GetVersionEx(Version);
if Version.dwPlatformId = VER_PLATFORM_WIN32_NT then
```

```
begin
<тут оперируем с Unicode>
end else
<тут действуем, как раньше>
. . . . .
```

Скопируем нашу недоделанную программу Layout из папки Glava5\4 в новую папку (Glava8\4) и приступим к внесению изменений. Установки языка через LoadKeyboardLayout нам, понятно, здесь больше не понадобятся. Для того чтобы загрузить в буфер обмена строку в формате Unicode (или вообще в каком-нибудь определенном формате), в модуле Clipbrd определен метод SetAsHandle (соответствующие методы буфера обмена в Delphi основаны на API-функциях SetClipboardData и GetClipboardData). То есть требуется получить дескриптор строки, для чего мы пойдем стандартным путем: расположим Unicode-строку в куче (heap).

Однако предварительно надо и получить строку из буфера обмена, если она там в Unicode. Это проще — по крайней мере не надо определять версию Windows, следует просто узнать, есть ли среди прочих форматов в буфере формат CF_UNICODETEXT.

Вот как это все можно реализовать. Объявим в программе Layout дополнительные переменные:

```
. . . . .
var
. . . . .
pstw:PWideChar;
stw:WideString;
stSize,i,n:integer; {размер строки и счетчики}
Version: TOSVersionInfo;
stHdl: THandle;
stPtr: Pointer;
. . . . .
```

Соответствующий фрагмент текста процедуры перекодировки по нажатию горячей клавиши будет выглядеть следующим образом:

```
. . . . .
Sleep(100);
Clipboard.Open;
if Clipboard.HasFormat(CF_UNICODETEXT) then
begin
 stHdl:=Clipboard.GetAsHandle(CF_UNICODETEXT);
 stPtr:=GlobalLock(stHdl);
```

```
 pstw:=stPtr;
 stw:=pstw;
 GlobalUnlock(stHdl);
 stClb:=stw;
end
else stClb:=Clipboard.AsText; {забрали в строку чистый текст}
. . . . .
<перекодируем строку>
. . . . .
Version.dwOSVersionInfoSize := SizeOf(TOSVersionInfo);
GetVersionEx(Version);
if Version.dwPlatformId = VER_PLATFORM_WIN32_NT then
begin {это NT}
with Clipboard do
  begin
    stSize:=length(stClb)*sizeof(WideChar)+2;
       {длина строки в Unicode}
    stHdl:=GlobalAlloc(GMEM_MOVEABLE,stSize);
       {отводим память в куче}
    stPtr:=GlobalLock(stHdl); {указатель на эту память}
    MultiByteToWideChar(CP_ACP,0,PChar(stClb),Length(stClb),
    stPtr,stSize); {преобразуем в Unicode}
    Clear; {очищаем буфер}
    Clipboard.SetAsHandle(CF_UNICODETEXT,stHdl);
        {забираем в него}
    GlobalUnlock(stHdl);
    GlobalFree(stHdl);  {очищаем память}
  end;
end else {это 9x}
Clipboard.AsText:=stClb;
Clipboard.Close;
<эмулируем нажатие клавиш>
. . . . .
```

Здесь задержка `Sleep(100)` необходима, чтобы в Windows XP приложение, из которого производится копирование, успело закрыть свой буфер обмена — ведь программы выполняются независимо и за время мгновенного перехода от имитации команд копирования к открытию буфера приложение почти гарантированно не успеет этого сделать (если только мы не попадем на самый конец кванта времени, который отводится нашей программе).

Если мы проверим работу этой программы, то увидим, что, по крайней мере, с Блокнотом все теперь работает правильно, но наблюдаются как конфликты с пресловутым множественным буфером обмена в Word XP, так и несурази-

цы с русским языком в некоторых не-Unicode редакторах, о чем шла речь ранее. Чтобы решить оставшиеся проблемы, надо поэкспериментировать с форматами и научиться как-то справляться с Word, но углубляться в эту тему далее мы уже не будем. Для детального изучения есть много ресурсов с более подробным описанием приемов работы с буфером обмена, а сами по себе эти приемы в прикладных программах требуются нечасто — если вы поэкспериментируете, то увидите, что копировать по указанной схеме (без обратной вставки) из Word можно без проблем, а это значительно более распространенная задача.

ГЛАВА 10

Графика и Windows

Приемы отображения и преобразования растровых изображений

Юзер Вася Чайников познакомился в чате с 16-летней девушкой Машей и попросил прислать ее фотку. Так как Маша не знает других форматов, кроме BMP, ее фотка занимает 15 Мбайт. У Васи коннект с провайдером на 2400. При этом связь рвется каждые 20 минут, а дозвон после этого занимает 30 минут. Льготный тариф действует с 3:00 до 5:00. Сколько лет будет девушке Маше, когда Вася докачает ее фотку?

forum.winall.ru

Небольшое отступление в форме memories (*воспоминания* — англ. лит.). В далеком 1996 году, когда до нашей тогда еще очень провинциальной в компьютерном отношении страны докатилась волна всеобщего увлечения новомодной Windows 95, у автора этих строк была отличная "трешка" на 40-мегагерцовом процессоре от AMD и он (т. е. я) искренне недоумевал — *зачем все это*? Ну зачем какие-то 95-е "форточки", когда DOS на этой машине просто летала, ну а для вовсе не ежедневно нужного WYSIWYG есть Windows 3.1 и Word. Последний пришлось освоить еще в версии 2.0, в значительной степени потому, что мой отец тогда написал книгу, и вознамерился ее сам сверстать. Так как он по этому случаю звонил мне каждые два часа на предмет консультаций, то приходилось соответствовать. В ОКБ, где я служил, была одна или две 486/66, на которой эти самые 95-е "окна" кое-как тащились. Ни о каком Plug&Play и речи, разумеется, не было — ради эксперимента мы подсунули "мастдаю" купленный мной первый в жизни привод CDROM Phillips 4x, после чего он ("мастдай", а не привод) минут пятнадцать чирикал винчестером, и, в конце концов, радостно сообщил, что нашел новое оборудование — трехкнопочную мышь! Нужные драйверы под Windows 95 было найти не проще, чем в советские времена достать туалетную бумагу, а записанная на моих глазах продавцами привода дискета рассыпалась в прах,

не прожив и нескольких часов, причем так, что от нее отказались все утилиты восстановления данных — тогда было в моде подсовывать дешевые дискеты 720 Кбайт, отформатированные на 1,44 Мбайт.

Если вы еще учтете, что единственной моей любимой игрой на ПК, кроме быстро прошедшего увлечения всякими разновидностями Тетриса, был и остался "Марьяж", а программированием я занимался по необходимости (программами, написанными мной в те времена на Borland Pascal, пользуются и до сих пор), то легко поймете мое скептическое отношение в продукту, который усилиями дядюшки Билла воспринимался скорее как очередной хит Мадонны, чем нечто для серьезной работы. И переломным моментом в моем отношении к Windows стала ее возможность отображать полноцветные картинки. Экраны по-прежнему регулярно синели, вместо русского языка в новом, 97-м, Word через раз появлялась "арабская вязь", от какого-нибудь из клонов Norton Commander я и до сих пор не могу отказаться в пользу метафоры "рабочего стола", требующей (в Windows) на порядки больше действий и времени на их выполнение, но одновременно я открыл для себя Photoshop. Это было захватывающее дух открытие — фактически домашняя фотолаборатория, с которой я был знаком с детства, вдруг переехала в компьютер с неизмеримо выросшими возможностями! Поэтому я пробил существенную дыру в семейном бюджете, купив передовой по тем временам Pentium-166 с 15-дюймовым монитором от Samsung, и именно графика стала для меня "иглой", подсев на которую, от Windows уже было невозможно отказаться.

Мисс компьютерная индустрия

В ноябрьском номере Playboy за 1972 год "девушкой месяца" была некая Ленна Шьоблом (Lenna Sjööblom), модель из Швеции (на самом деле ее звали просто Лена, в Ленну ее переименовали в Playboy). Фотографом был некто Дуайт Хукер (Dwight Hooker). Примерно через полгода некоему Александру Савчуку (потомственный американец, несмотря на фамилию) из Университета Южной Калифорнии для иллюстрации статьи на тему обработки изображений понадобился фотопортрет с хорошим динамическим диапазоном. В те времена коллекций качественных изображений в цифровой форме еще не существовало, и Савчук, недолго думая, сканировал фрагмент постера из упомянутого Playboy (рис. 10.1). В его распоряжении был сканер с разрешением 100 линий на дюйм, и результирующее изображение получилось 512×512 точек. Вскоре эта картинка превратилась для индустрии в стандарт де-факто: на нем проверялись и оттачивались новые алгоритмы обработки изображений. Отметим, что это именно фрагмент настоящего изображения — полностью фотография никогда не использовалась не только вследствие ее фривольного содержания, но из соображений удобства и воспроизводимости результатов. Одно время портрет Ленны

входил в качестве тестового изображения в комплект поставки некоторых графических пакетов. Слагались даже посвященные ей поэмы.

Сотрудники Playboy ни о чем не подозревали, пока в 1991 году Ленна не появилась на обложке журнала Optical Engineering. Сначала разразился скандал (Playboy всегда очень ревниво относился к своим авторским правам), но потом все было улажено. Еще дольше оставалась в полном неведении сама Лена — она к тому времени давно забросила модельный бизнес, вышла замуж, вырастила пару детей и только в мае 1997 года ее разыскал кто-то из энтузиастов и пригласил на юбилейную конференцию в честь 50-летия Общества по обработке изображений (**imaging.org**). Хотя ей было уже далеко за 40, но все же всем было приятно: Playboy опубликовал репортаж, особо отметив сыгранную им роль в развитии IT, а Лена начала свою речь словами: "Вы, должно быть, так устали от меня... Глазеть на этот снимок столько лет!".

Рис. 10.1. Фрагмент фотографии Ленны Шьоблом, использовавшийся при разработке алгоритмов в качестве тестового изображения

Растровые изображения в Windows

Microsoft всегда заимствовала чужие идеи, но никогда не воспроизводила их один к одному. Иногда у нее получалось лучше, чаще — хуже, но всегда оригинально. Оригинальным стал и родной для Windows формат растровых изображений BMP (что очевидно расшифровывается, как просто BitMaP, т. е. "карта битов"). Этот формат является основным при хранении изображений в памяти (в том числе и при переносе их через буфер обмена), и, естественно, существует в виде формата дискового файла. Другое наименование формата — DIB (Device-Independent Bitmap "аппаратно-независимый битмэп"; собственно, аббревиатуру BMP правильно относить только к дисковым файлам).

Напомним, что вообще-то любая печатная картинка, начиная с появления цинкографического способа воспроизведения изображений, состоит из от-

дельных точек, т. е. представляет собой растр. Качество печатных изображений напрямую зависит от подробности растра; считается, что разрешение в 300 точек на дюйм (dpi) достаточно для самой высококачественной полиграфической печати. Отсюда легко подсчитать, что картинка, соответствующая любительскому фото 10×15, должна иметь размеры примерно в 1200×1600 точек (или 2 мегапиксела в терминах цифровой фотографии). На самом деле это требование критично только для действительно качественной печати (типа "глянцевых" журналов или художественных альбомов), обычная цветная печать обходится разрешениями 100—150 dpi (уточнение этих понятий см. далее во врезке). В домашних условиях, или даже для изданий типа цветных еженедельников, 2-мегапиксельную картинку вполне можно растянуть на формат A4, правда, при условии, если исходный формат файла — не чересчур сжатый JPEG. В различия между форматами сжатия мы здесь вдаваться не будем — об этом написано достаточно много, наш предмет — собственно растровые изображения, т. е. чистая "карта битов". Для полноты картины добавим, что все упомянутые ранее рассуждения о разрешениях справедливы лишь именно для печати, а для воспроизведения на экране действуют совершенно другие законы — прежде всего потому, что максимальное разрешение современных мониторов застыло на достигнутом еще лет 10 назад уровне максимум в 100 ppi (пикселов на дюйм) и больше этого предела в демонстрируемых картинках повышать его нет никакого смысла (картинка "во весь экран" в масштабе 1:1 на обычном мониторе — это примерно 1,2 мегапиксела). Поэтому для размещения в Интернете (если вы не выкладываете картинки специально для скачивания с целью последующей печати, или если это не штриховые чертежи) максимально необходимыми для качественного воспроизведения будут размеры около 640×480 — при большем размере вы будете только получать массу проклятий на свою голову за слишком долгое время загрузки.

Отображение растра при печати

Читатель вправе задать вопрос — а зачем нужны принтеры с разрешением, которое может на сегодняшний день составлять до 2400 точек (физических!) на дюйм и даже более, если даже в качественной полиграфии ограничиваются значением 300? Между понятием физического разрешения устройства печати (измеряемом в dpi) и результирующим разрешением картинки (линеатурой растра в линиях на дюйм, lpi) есть большая разница, которую мы рассмотрим на примере монохромной печати оттенками серого. При типографской печати мы располагаем только одной краской, прозрачность которой никак регулировать нельзя. Как в таком случае обеспечить полутона? Единственным очевидным способом будет там, где изображение светлее, часть точек (лучше всего в случайном порядке) просто не закраши-

вать. Те, кто видел советские газеты 40—50-летней давности и старше, отлично понимает, о чем я — там растр был столь груб, что описанную мной технику воспроизведения полутонов можно было разглядеть невооруженным взглядом. С повышением разрешения печатающих устройств качество стало более приемлемым, но принцип сохранился (и в цвете он точно такой же, только там не один, а четыре разноцветных растра, наложенные друг на друга под разными углами, по числу красок в модели CMYK, на которой мы здесь останавливаться не будем). Этот метод передачи полутонов называется частотной модуляцией.

В результате разрешение печатающего устройства (dpi) и заданное разрешение полученного изображения (lpi) оказываются связаны между собой нелинейной зависимостью. Линеатура есть величина обратная минимальному размеру элементарной ячейки изображения, в которой может быть воспроизведено заданное количество полутонов. Во многих профессиональных принтерах, между прочим, линеатуру можно устанавливать специально. Если взять за элементарную ячейку фрагмент изображения, равный разрешающей способности принтера (или типографского станка), то он может иметь только два цвета — черный или белый. Чем больший фрагмент изображения мы будем рассматривать как элементарную ячейку, тем больше оттенков можно различить. При максимальном качестве оттенков серого должно быть 256 (позже мы узнаем, почему принята именно эта величина). Какой величины ячейка обеспечит столько оттенков при заданном разрешении принтера? Оказывается, есть простая формула, которая позволяет это подсчитать точно: число полутонов равно единице плюс корень квадратный из частного от деления разрешения на линеатуру. Легко подсчитать, что при разрешении принтера 600 dpi и 256 оттенках, линеатура будет всего-навсего 40 lpi, т. е. фактическое разрешение полутоновой картинки будет всего 40 линий на дюйм, или 15 линий на сантиметр! Очень грубая картинка, и это знает каждый, кто пытался печатать полутоновые фото на старых лазерных принтерах. (Кстати, для цвета расчет будет тем же самым, т. к. печать выполняется отдельно для каждого из составляющих цветов.)

Как же выйти из положения? Во-первых, для увеличения качества повышают физическое разрешение при печати — отсюда те самые умопомрачительные цифры. Для современных фотонаборных автоматов при подготовке картинок для качественной печати характерно физическое разрешение в 2540 dpi (100 точек на мм), что при всех воспроизводимых 256 градациях серого при расчете по формуле, приведенной ранее, соответствует линеатуре в 160 lpi или размеру полутоновой точки растра в 0,16 мм — вполне приличная величина, но фактически линеатура может быть установлена и значительно большей, потому что на бумаге с ее динамическим диапазоном максимум в 2D (т. е. отличием между черным и белым максимум в 100 раз) 256 градаций серого никому не нужны, они просто неразличимы. И тем не менее, нельзя же физическое разрешение увеличивать беспредельно?!

Поэтому в струйных принтерах научились регулировать величину точки растра. При этом методе — называемом еще амплитудной модуляцией — приемлемая передача полутонов возможна даже при относительно грубом растре, упомянутая ранее формула уже не работает и величина линеатуры вообще не применима, растрирование при печати не производится, элементарная ячейка соответствует физическому разрешению. И для качества принтера большее значение имеет не разрешение — конечно, сверх некоего предела, который уже достигнут во всех современных моделях — а минимальный размер капли красителя, который он может выдать. Приличной величиной сейчас считается объем капли в 2 пиколитра, что соответствует ее диаметру в 8 микрон. На самом деле капля еще должна растечься, поэтому, чтобы соответствовать декларированному разрешению в 2400 dpi (примерно 12 микрон), поверх этого основного способа в струйных принтерах придумывают еще всякие ухищрения, вроде того, что капля специально делится на множество мелких, что позволяет более тонко регулировать полутона. И все же до качества настоящих фотоизображений, созданным традиционным серебряным способом, не могут дотянуть ни типографские, ни струйные устройства.

На практике ориентировочные цифры такие: при увеличении разрешения выше примерно 1200—1440 dpi цветная полутоновая картинка на струйном принтере зрительно лучше не становится, только чернила больше расходуются, а для нормального качества в быту вполне достаточно и 720 dpi. Заметим еще, что ведь все преимущества высокого разрешения принтера появляются в полной мере, если разрешение исходного изображения достаточно велико (отсюда и требование в 300 dpi) — нет никакого смысла повышать разрешение при печати, если соседние точки все равно будут иметь одинаковый цвет. Но, конечно, оно необязательно должно достигать приведенных огромных величин, принтер при высоком разрешении просто лучше прорабатывает переходы между точками, и по этой причине картинки небольшого размера для печати следует готовить, растягивая их заранее, а не полагаясь на принтер. При грамотных алгоритмах растяжки исходная точка не делится на идентичные маленькие точки, фактически просто увеличиваясь в размерах, а цвета получающихся точек подбираются по специальным алгоритмам.

Имейте также в виду, что картинки при загрузке их в канву компонентов Delphi и в ресурсы приложения сами по себе не ужимаются в размерах — если вы загрузите изображение 2000×3000 точек, то оно уменьшится только при отображении, а в файле будут храниться все положенные мегабайты (то же самое относится и к картинкам, размещаемым в документах Word, закачиваемым с HTML-страницы, или хранимым в файлах PDF). Поэтому при под-

готовке проекта картинки следует заранее подгонять примерно под тот размер, который они будут иметь при отображении на экране.

Поняв теперь, что именно и когда нам требуется в смысле размеров и разрешения, обратимся на уровень ниже — а что такое пиксел при воспроизведении на экране? Смысл этого понятия кроется в устройстве любого экрана, на котором элементарная точка состоит из трех: красной, зеленой и синей составляющей. Балансируя яркостями этих точек, можно получить все оттенки цветов — от черного (все три составляющих погашены) до белого (одинаково максимальная яркость всех трех составляющих). К этой модели цвета, называемой еще "RGB" по именам трех основных цветов (Red-Green-Blue), при отображении на экране приводятся все остальные многочисленные модели цвета, на которых мы задерживаться здесь не будем[1].

Компьютерные системы на основе DOS, о которых я с такой ностальгией вспоминал в начале главы, начиная с 90-х годов стандартно поддерживали так называемый режим VGA (Video Graphic Array, стандарт появился с внедрением компьютеров IBM PS2 в конце 80-х годов), который сейчас поддерживают все компьютеры на уровне BIOS (в этом режиме идет, например, установка Windows). VGA предполагает разрешение экрана 640×480 и 16 воспроизводимых цветов. Нас интересует последняя величина — как это достигается? В этом режиме каждый RGB-пиксел кодируется одним полубайтом (что равносильно одному шестнадцатеричному разряду), который, как известно, принимает как раз 16 значений. Раскладывается этот полубайт так: младшие три бита есть состояние RGB-составляющих (именно в таким порядке, т. е. самый младший бит представляет синий цвет), а старший управляет общей яркостью (фактически упрощенный вариант управления палитрой, которых в данном случае может быть две — более подробно об этом см. далее). Например, чтобы воспроизвести черный цвет, нужно все биты установить в 0, белый — в 1, а серых получится целых два: когда младшие три бита в 1, а старший — в 0 (светлый серый), или наоборот, когда младшие все в 0, а старший — в 1 (темный серый). Соответственно, будут два красных, два синих и тому подобных цвета — теперь вы легко можете понять, откуда взялось понятие "основных" (или "стандартных") 16 цветов и почему их порядок именно таков (см., например, выпадающий список в компоненте `ColorBox` в проекте SlideShow из предыдущей главы). Сами эти цвета перечислены в таблице ниже вместе с их 24-битным представлением. Картинки, в которых представлено всего 16 оттенков, физически занимают на диске (без дополни-

[1] Тем не менее их следует изучить при работе с графикой, даже если вам никогда не придется заниматься допечатной подготовкой иллюстраций (современные принтеры сами за вас все сделают). Знание цветовых моделей сильно поможет при качественной растяжке снимка, оптимальном преобразовании цветной картинки в оттенки серого и т. п.

тельного сжатия, разумеется) ровно в два раза меньше байт, чем арифметическое произведение их высоты и ширины в пикселах (т. е. один байт представляет сразу два пиксела).

На самом деле адаптер VGA имеет возможность отображать много больше оттенков, чем 16 стандартных, поскольку внутреннее представление цвета в нем 18-битное (по 6 бит или 64 градации на каждую из RGB-составляющих). Это достигается с помощью задания так называемой палитры — т. е. установки непосредственного значения в соответствующих регистрах преобразования цвета в выходном цифроаналоговом преобразователе (Digital-Analog Converter, DAC) адаптера. Одновременно может воспроизводиться все равно только 16 оттенков, но они выбираются из 262 144 возможных. Текущая палитра может задаваться в файле с изображением (см. далее устройство формата BMP), или просто устанавливаться программно (процедура `SetRGBcolor` в модуле Graph из TurboPascal и TurboC). Подменой палитры можно, не меняя самого изображения, раскрасить его в другие цвета. Заданием палитры широко пользуются при редукции цветов (т. е. при сведении их к меньшему количеству оттенков) — так, широко известный формат GIF имеет однобайтное представление цвета (256 одновременно воспроизводимых оттенков), но при преобразовании из полноцветного (TrueColor) файла он в палитре сохраняет именно те оттенки, которые наиболее близки к оригиналу.

Набор сочетаний разрешения экрана и количества отображаемых при этом цветов устанавливает поддерживаемый всеми видеокартами стандарт VESA (Video Electronics Standards Association — ассоциация производителей компьютеров, организованная в 1989 году 29 компаниями с целью стандартизации в области компьютерной графики). Редакция 1.2 этого стандарта устанавливает набор графических режимов от 640×400 (256 цветов) до 1280×1024 (16 млн цветов), который поддерживается на уровне BIOS видеоадаптера. Режимы ниже первой величины (в том числе и "любимый" DOS 640×480, 16 цветов) поддерживаются на уровне BIOS материнской платы. Разрешения экрана могут быть и выше указанной в стандарте версии 1.2 величины (есть и версия VESA 2.0.), но они нас сейчас не волнуют, мы займемся количеством отображаемых цветов. Перескочив через редко используемый в наше время режим SVGA в 256 цветов (один целый байт на пиксел), а также через режим HighColor (16 бит на пиксел или 65 536 цветов) и другие, в том числе никогда не использовавшиеся на практике режимы типа 15 бит на пиксел, обратимся к наиболее распространенному в наше время режиму TrueColor, в котором на каждый пиксел приходится по три байта (или 24 бита). Естественно, никакой специальной "палитры" в таком режиме уже нет, разрядность DAC строго совпадает с представлением цвета в памяти. Каждый из отдельных байтов дает 256 вариантов для каждой из RGB-составляющих, итого пресловутые 16 миллионов, а точнее 16 777 216 (2^{24}) цветов. На самом деле и это количе-

ство отнюдь не исчерпывает всех оттенков, возможных в природе — именно поэтому практически все современные сканеры поддерживают "на всякий случай" 48-битный цвет. Такой запас нужен не для отображения, а для более точного представления цвета при преобразованиях — чтобы избежать ошибок округления, например, при сильной растяжке изображения. В конечных изображениях глубина цвета все равно обычно не превышает 24 бита, на практике для отображения на экране и тем более для печати на бумаге (где диапазон яркостей еще на пару порядков меньше) трех байтов более чем достаточно. Если подумать, то легко сообразить, что в таком представлении оттенков серого будет всего 256 (все три байта должны иметь одинаковую величину — и на практике этого также более чем достаточно), поэтому их можно хранить и в одном байте. Так как получается, что серый в цветном изображении и серый в формате именно оттенков серого (Grayscale) — для компьютера несколько разные вещи, то мы можем наблюдать печальные последствия такого разделения при печати на некоторых не самых продвинутых фотопринтерах — при воспроизведении оттенков серого в окружении других цветов они упорно пытаются имитировать серый смешением цветных красок, и результаты бывают довольно плачевными.

Кстати, преобразовать цветной BitMap в оттенки серого можно без специального преобразования палитры, если придать всем трем байтам для каждого пиксела одинаковую величину. Но вот по какому алгоритму? Это сложный и неоднозначный вопрос, но для автоматического преобразования обычно принято использовать формулу, которую применяют в телевидении: Gray = = 0.30R + 0.59G + 0.11B. Формула выведена эмпирически, и хорошо иллюстрирует тот факт, что яркость человек в основном воспринимает в зеленой части спектра. Нужно совершить это преобразование для каждого пиксела и затем приравнять все три его компонента полученной величине[2]. Для цвета, заданного типом `TColor` в вашей программе (именно этот тип имеет массив `Canvas.Pixels`), это можно делать через функции `Get`*X*`Value (Color:Tcolor)` (где *X* равен `R`, `G` или `B`), которые возвращают интенсивность той или иной составляющей, и функцию `RGB (R,G,B)`, которая производит обратное преобразование. В примере с оттенками серого вместо `R`, `G` и `B` в последней функции нужно подставить одну и ту же полученную расчетом величину `Gray`. Хотя далее мы будем делать похожую операцию непосредственным манипулиро-

[2] Для того чтобы увидеть разницу между простым и взвешенным (по приведенной формуле) суммированием цветовых составляющих при преобразовании в оттенки серого, откройте в Paint Shop Pro (но не в Photoshop!) любую контрастную цветную картинку, сделайте ее копию и преобразуйте в оттенки серого штатным методом через меню **Color | GrayScale**. А затем вставьте через буфер обмена фрагмент цветного оригинала поверх серой копии (при этом пересчета по формуле не делается) — разница будет такая, что вы глазам своим не поверите.

ванием битами, использование этих функций может быть проще для понимания.

Из принципов представления цвета в формате RGB понятно, что означают используемые в программировании (а также, например, в HTML) числа типа $FF0080, с помощью которых задается цвет. Старший байт этого числа есть интенсивность синей, средний — зеленой, и младший — красной составляющих (т. е. в обратном порядке, чем это было в 16-цветном режиме). Белый цвет будет $FFFFFF, черный — $000000, базовый красный — $000080, базовый яркий красный — $0000FF и т. д. А наш пример $FF0080 представляет яркий синий + темный красный — красивый оттенок ярко-лилового. Любой из оттенков серого будет характеризоваться одинаковым набором байтов.

В табл. 10.1 приводится соответствие 16 базовых цветов и их 24-битных представлений. В скобках приведено название и значение соответствующей DOS-константы так, как она определена в модуле Graph из Turbo Pascal и Turbo C. Значение ее совпадает со значением полубайта, определяющего цвет в 16-цветном пикселе. Нельзя не отметить, что названия констант в Windows более отвечают бытовому представлению о цветах, хотя и менее логичны с физической точки зрения.

Таблица 10.1. *16 базовых (стандартных) цветов*

Цвет	Представление
clBlack (Black=0)	$000000
clNavy (Blue=1)	$000080
clGreen (Green=2)	$008000
clTeal (Cyan=3)	$008080
clMaroon (Red=4)	$800000
clPurple (Margenta=5)	$800080
clOlive (Brown=6)	$808000
clSilver (LightCray=7)	$C0C0C0
clGray (DarkGray=8)	$808080
clBlue (LightBlue=9)	$0000FF
clLime (LightGreen=10)	$00FF00
clAqua (LightCyan=11)	$00FFFF
clRed (LightRed=12)	$FF0000
clFuchsia (LightMargenta=13)	$FF00FF
clYellow (Yellow=14)	$FFFF00
clWhite (White=15)	$FFFFFF

BMP

Теперь перейдем к тому, как хранятся эти самые пикселы в формате BMP (остальные форматы устроены аналогично, отличаются они лишь способами хранения собственно изображения, а также количеством и расположением информации в заголовках). В начале файла идут две структуры, `BITMAPFILEHEADER` и `BITMAPINFOHEADER`. Самый первый элемент первой структуры представляет собой два ASCII-символа "BM" (сигнатуру), с которых и начинается любой BMP-файл (рассмотрите его в каком-нибудь HEX-редакторе или даже просто откройте в Блокноте). Это представляет удобный и быстрый, хотя и не стопроцентно надежный, способ опознавания формата (кстати, для иконок такого способа уже нет). Для надежности можно использовать дополнительно следующие четыре байта, которые представляют собой число типа `DWord` (записанное, естественно, начиная с младшего байта), равное размеру самого BMP-файла (а не картинки!).

Описание заголовочных структур легко при надобности найти на том же сайте MSDN и, т. к. нам BitMap вручную создавать, слава богу, не придется[3], то здесь заметим лишь, что в заголовках приводится исчерпывающая информация о собственно формате изображения (цветное или оттенки серого, количество цветов в битах на пиксел, ширина и высота картинки, разрешение в точках на метр, наличие и тип сжатия, где в файле расположено начало массива пикселов и другая полезная и не очень информация). Если число оттенков представляется числом бит на пиксел, отличающимся от TrueColor (т. е. меньше, чем 24), то после заголовков может идти палитра — массив `aColors`, представляющий структуры из четырех байтов, содержащих нужные интенсивности красного, зеленого и синего, четвертый байт декларируется как резервный. А собственно массив точек изображения тогда состоит из индексов к этому массиву, т. е. из номеров цветов. Некоторые подробности об устройстве файлов BMP-формата вы можете узнать из *главы 19*, где они будут использованы для иллюстрации приемов стеганографии — скрытой пересылки тайных сообщений.

Если вы попытаетесь сохранить TrueColor-картинку в формате BMP с 16 или 256 цветами, то результат будет таким же, как и в случае GIF — в палитре будут записаны наиболее близкие оттенки (а не базовые цвета!). Для TrueColor массива значений палитры вообще нет, после заголовков сразу начинается массив пикселов изображения (`aBitmapBits`), сами значения байтов в этом массиве дают исчерпывающую информацию о цвете. Так как каждый пиксел в этом случае кодируется тремя байтами, то место, занимаемое на

[3] Для тех, кто увлечется этим вопросом: на странице **http://wall.riscom.net/books/delphi/del_faqs/820.html** приведен подробный и хорошо комментированный (по-английски) алгоритм создания Bitmap, адаптированный к Delphi.

диске BMP-файлом, практически равно утроенному произведению ширины на высоту картинки в пикселах (чуть больше его, если учитывать заголовок), и кстати, эта величина указывается в заголовке для проверки целостности файла (варианты со сжатием мы не учитываем, т. к. на практике в BMP оно используется редко). Собственно расположение байтов в массиве `aBitmapBits` почему-то начинается с нижней строки изображения (т. е. первые три бита кодируют самый левый пиксел самой нижней строки). Какая в этом сермяжная правда — мне узнать не удалось, возможно при составлении формата хотели заставить всех разработчиков экранную координату Y отсчитывать в привычном для нас виде, т. е. по возрастанию снизу вверх. Но эта координата всегда отсчитывается от верхнего обреза экрана (потому что именно отсюда начинается построчная развертка электронного луча, а массивы битов, модулирующих его, располагать в памяти удобно именно по ходу его работы), и такой способ представления информации ведет лишь к дополнительным сложностям.

Все остальные многочисленные форматы для картинок, как уже говорилось, при работе с ними в Windows все равно переводят в BMP (точнее, в DIB) — так, в компонент `Image` мы в *главе 2* загружали картинку JPEG (что можно сделать при наличии соответствующего драйвера из модуля JPEG), но в исполняемом файле она все равно хранится в BMP.

Иконки

Остановимся теперь еще на одном важнейшем для Windows формате — для хранения иконок. Понять, как они устроены изнутри, даже важнее, чем в случае BMP, иначе будут непонятны многие действия, которые мы предпримем далее.

> Не все знают, что у типовых пиктограмм Windows есть конкретный автор — это дизайнер Сьюзен Кэр. Еще интереснее то, что она разрабатывали иконки, логотипы и элементы интерфейса не только для Windows. Еще раньше она сделал то же самое для MacOS, а позднее — для OS/2, и это объясняет тот факт, что, например, "песочные часы" и "корзина" в этих операционных системах очень похожи. Но и участием в этих крупных проектах она не ограничилась. В разное время Сьюзен сотрудничала с Intel, Logitech, Motorola, Netscape, Palm, Xerox и другими компаниями, так что не будет преувеличением утверждать, что интерфейсом почти любого компьютерного "девайса" в мире мы в конечном итоге обязаны именно ей.

Идея, которая лежит в основе воспроизведения иконок в Windows, представлена на рис. 10.2. Собственно иконка содержит не одно изображение, а два:

AND-маску и XOR-маску. Первая всегда представляет собой двухбитное изображение, т. е. только черный или белый цвет для каждого пиксела (на рисунке черный цвет означает биты с нулевым значением). Сначала к фону "прикладывается" именно она. Эта маска потому и носит название AND-маски, что над пикселами совершается побитовая операция AND (каждый бит в AND-маске в любом случае соответствует целому пикселу, т. е. если фон имеет глубину цвета TrueColor, то не одному, а всем 24 битам фона). Согласно определению операции AND (равно 1 только тогда, когда обе исходные величины равны 1), там, где в маске были нулевые биты, в фоне образуется "дыра" (на самом деле пикселы просто окрашиваются в черный цвет). Затем к тому же месту прикладывается XOR-маска, которая в принципе может содержать изображение с глубиной цвета от 2 до 16 оттенков (а теоретически — и 256, и даже TrueColor), но на практике чаще используется именно 16, т. е. цвет хранится в ней в формате полубайта (на рисунке для наглядности XOR-маска показана также с двухбитным цветом). Delphi умеет работать только с 16-цветными иконками, из-за чего у нас возникнут в дальнейшем сложности. С XOR-маской над пикселами фона (с уже имеющейся "дырой") производится операция "Исключающее ИЛИ" (равно 0 тогда, когда оба входа одинаковы, см. также *главу 19*). Там, где в XOR-маске были поля с нулевым значением битов (т. е. черного цвета), фон остается неизменным (если одна исходная величина равна нулю, то операция XOR будет повторять вторую исходную величину), в противном случае на фон с дырой накладывается то изображение, которое записано в XOR.

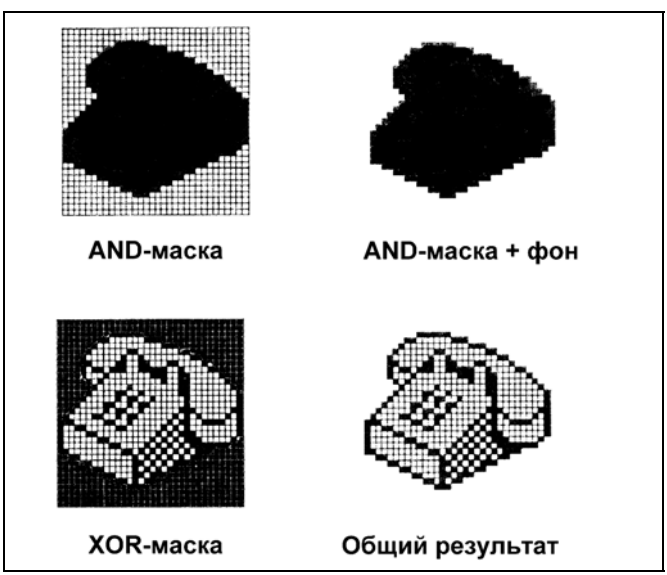


Рис. 10.2. Отображение иконок в Windows (черным показаны пикселы с нулевым значением)

Заметки на полях

Что будет, если мы "поиздеваемся" над изображением с помощью различных логических операций? Если применить к массиву пикселов операцию NOT, то мы получим изображение в инвертированных цветах — именно так проще всего производить выделение (так и делается, скажем, в Word, когда белый нужно заменить на черный и наоборот). Но могут быть и другие варианты — так, в элементах управления, в браузерах выделение традиционно производится синим, а не черным фоном, но делается это по-разному. На диске в папке Glava10\Ctrl-A лежит файл ctrl-A.htm, который содержит потрясающие по выразительности графические произведения неких умельцев, очень хорошо иллюстрирующие возможности операций над пикселами. Попробуйте загрузить этот файл в Internet Explorer и нажать комбинацию клавиш <Ctrl>+<A>. Не пытайтесь открыть картинки отдельно в обычном редакторе, загружать страничку в Opera, FireFox или, к примеру, в Word — того же эффекта вы не получите, там выделение производится иной логической операцией. Кстати, преобразование BMP в JPEG также сильно портит эффект.

И операции копирования канвы, которые мы рассмотрим далее, и перо Pen допускают различные режимы взаимодействия с фоном. Если скопировать изображение фона в отдельный BitMap, а затем скомбинировать его с тем же самым фоном операцией XOR, то оно уничтожится — будет черная дыра. Если применить операцию NOTXOR, то дыра станет белой. Если скомбинировать фон через XOR с белым квадратом — он инвертируется, если с черным — не изменится, а применение XOR с отдельными чистыми цветами будет инвертировать соответствующую составляющую. Так, если исходный квадрат синий, то белый цвет станет желтым, желтый — белым, черный — синим и т. п. Инвертированию подвергнутся только чистые цвета, а "грязные" будут изменяться непредсказуемо. Определения логических операций нелишне выучить наизусть, но даже в этом случае для точного предсказания результата чаще всего необходимы эксперименты.

А для иконок, естественно, подгадывают так, чтобы изображение на XOR-маске совпадало с дырой (т. е. с тем местом, где в AND-маске были нули), иначе полученные цвета не будут соответствовать заданным. Так получается эффект прозрачности иконки — естественно, что для иконки без прозрачных областей AND-маска должна содержать все 0 (белое допускается там, где и на XOR-маске белый цвет), а XOR-маска полное изображение.

Описанный механизм получения "прозрачного" цвета есть не что иное, как физическая реализация альфа-канала — понятия, широко распространенного в компьютерной графике. Он может быть реализован для любого изображения, не только иконки — так, Windows 2000/XP поддерживают подобный механизм для формы в целом, но в Windows 98 у вас ничего этого работать не будет *(см. также главу 12)*. Любой BitMap можно отобразить с "прозрачным" цветом, но только отобразить — сохранить альфа-канал в стандартном файле BMP не удастся. Позже мы вернемся к вопросу, что на практике означает термин Transparency для класса `TBitMap`, а сейчас только заметим, что файлы в формате BMP потому и не используются в профес-

сиональной графике, что не поддерживают, в частности, альфа-канал. Даже устаревший, но по-прежнему популярный GIF поддерживает однобитный (как в иконках) "прозрачный" цвет, а такие форматы, как PNG или профессиональный формат TIFF, поддерживают восьмибитную градацию прозрачности (256 уровней). Также в них нет никаких ограничений на глубину цвета — она может составлять от 2 до 48 бит (больше, чем 281 474 976 710 656 оттенков, видимо, на практике уже и не потребуется).

Из этих соображений легко подсчитать размер файла стандартной иконки 32×32 пиксела с 16 цветами. AND-маска должна содержать 1024 бита или 128 байт, а XOR-маска с 4-битным цветом 512 байт, итого собственно изображение займет в файле 640 байт, плюс оно сопровождается таким же заголовком `BITMAPINFOHEADER`, как и в BMP, размером ровно 40 байт. Кроме этого, в файле имеется 22-байтный заголовок `ICONHEADER` и еще 64 байта занимает палитра в четырехбайтном представлении каждого из 16 возможных цветов. Итого 766 байт (проверьте!). В принципе один файл иконки может содержать и несколько разных иконок, но на практике этим никто не пользуется (достаточно того, что в ресурсах можно задать несколько иконок). Что же касается размеров, то они могут быть 16×16, 32×32 и даже 64×64. Интересно, что традиции представления иконок существуют уже, наверное, лет двадцать без каких-то изменений — как и в Windows первых версий, в последней версии XP наиболее популярны иконки 32×32 с 16 цветами.

Преобразование BitMap в Icon

Как мы видим, иконка (Icon) — это совсем не просто картинка 32×32 пиксела. По этой причине преобразование BitMap в Icon и обратно простым копированием невозможно. А как же быть, если мы страстно хотим создать иконку динамически, прямо при выполнении программы? Придется изворачиваться. Сейчас мы соорудим демонстрационную программу (доводить до полноценного приложения не будем), которая загружает произвольную картинку в формате JPEG или BMP, преобразует ее в иконку и сохраняет полученный ICO-файл. Заодно по ходу дела будут проиллюстрированы многие приемы работы с картинками в Delphi. В качестве пробного камня используем полноцветный женский портрет размером 800×600 пикселов на белом поле (я его заимствовал из сборника экранных обоев), логотип Apple в базовых цветах и фото бабочки, палитра которой изначально также была близка к базовой. Формат всех исходных файлов JPEG (файлы oboi359.jpg, apple.jpg и butterfly1.jpg находятся на диске в папке Glava10\1).

Создадим новый проект (папка Glava10\1), назовем его Iconka, разместим на форме компонент Panel, оставив справа свободное поле, покрасим его (свойство Color) в черный цвет и на него поставим компонент Image — в точности так, как мы это делали в SlideShow в *главе 2*. У компонента Image установим в True свойства Stretch и Proportional. На форму добавим компонент OpenDialog, у которого установим фильтр по файлам с изображениями JPEG и BMP (в правое поле в окне **OpenDialog1.Filter** надо вписать *.jpg; *.bmp). Собственно, формат BMP нам в этом примере в качестве исходного не пригодится, но в принципе не помешает, а реализовать его загрузку очень просто, т. к. JPEG все равно придется переводить в BMP для последующих манипуляций. Объявляем следующие переменные:

```
var
Form1: TForm1;
JpgIm: TJpegImage; {объявляем переменную типа JPEG}
BmpIm,IcoBmp,XORmask,ANDmask: TBitMap; {переменные типа BitMap}
IcoIco:TIcon; {переменная типа иконка}
RectS,RectT:TRect;
IcoInfo : TIconInfo;
st,filename:string;
. . . . .
```

Сразу, не особенно разбираясь, соорудим процедуру уничтожения всех экземпляров классов при закрытии программы (кроме JpgIm, которую мы будем уничтожать сразу после использования):

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin {все уничтожаем}
try
ANDMask.Destroy;
XORMask.Destroy;
IcoBmp.Destroy;
IcoIco.Destroy;
BmpIm.Destroy;
except
end;
end;
```

Далее создадим процедуру инициализации диалога открытия файла при запуске программы, с тем, чтобы он открывался на текущем каталоге:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
chDir(ExtractFileDir(Application.ExeName));
```

```
OpenDialog1.InitialDir:=ExtractFileDir(Application.ExeName);
end;
```

Поставим на форму кнопку Button1 с заголовком **Load** и напишем следующий обработчик для щелчка на ней:

```
procedure TForm1.Button1Click(Sender: TObject);
begin  {Load - загрузка}
 BmpIm := TBitMap.Create; {создаем экземпляр объекта типа BitMap}
 If OpenDialog1.Execute then filename:=OpenDialog1.filename
 else exit;
 try
  If ExtractFileExt(filename)='.jpg' then
  begin
   JpgIm := TJpegImage.Create;
      {создаем экземпляр объекта типа JPEG}
   JpgIm.LoadFromFile(filename);
     {в JPEG загружаем изображение с диска}
   BmpIm.Assign(JpgIm); {в BitMap загружаем изображение из JPEG}
   JpgIm.Destroy; {уничтожаем объект JPEG}
  end;
  If ExtractFileExt(filename)='.bmp' then
  BmpIm.LoadFromFile(filename);
     {в BMP загружаем изображение с диска}
 except
  exit;
 end;
 Image1.Picture.Assign(BmpIm); {выводим BitMap на экран}
end;
```

Пробовать начнем с самого сложного образца — женского портрета. Вот результат загрузки его в окно **Image1** (рис. 10.3).

Заметим, что компонент Image можно было бы и не устанавливать, а рисовать прямо на канве формы или любого другого компонента, просто с ним в данном случае работать удобнее (но, как мы увидим позже, иногда дело идет лучше, если использовать другие компоненты).

Для начала нам надо решить проблему, как преобразовать прямоугольную картинку в квадратную без ее искажения, т. е. осуществить операцию cropping (букв. "подстригание"), известную по любому графическому редактору. Разумеется, если изначальная картинка квадратная и близка к тому, то этого делать не нужно (и, например, для логотипа Apple эту операцию следует пропустить). Несложно реализовать ручное указание области, которую нужно сохранить при данной операции, но я предоставляю читателю самому разобраться с этим делом, а здесь реализую автоматическое выделение квад-

рата по центру картинки, в зависимости от того, по какому измерению картинка больше — по высоте или по ширине. Поместим на форму кнопку Button2 (**Crop**) и напишем для нее следующий обработчик:

```
procedure TForm1.Button2Click(Sender: TObject);
var x,y: integer; {Crop - делаем картинку квадратной}
begin
if BmpIm.Width>BmpIm.Height then {если больше по ширине}
begin
 x:=(BmpIm.Width-BmpIm.Height) div 2; {расст. от левого края}
 RectS:=Rect(x,0,x+BmpIm.Height,BmpIm.Height);
       {исходное поле по центру картинки}
 RectT:=Rect(0,0,BmpIm.Height,BmpIm.Height);
     {поле, куда копировать}
end else {если больше по высоте}
begin
 x:=(BmpIm.Height-BmpIm.Width) div 2; {расст. от верхнего края}
 RectS:=Rect(0,x,BmpIm.Width,x+BmpIm.Width);
       {исходное поле по центру картинки}
 RectT:=Rect(0,0,x+BmpIm.Width,x+BmpIm.Width);
        {поле, куда копировать}
end;
 BmpIm.Canvas.CopyMode:=cmSrcCopy;
       {режим копирования - на всякий случай}
 BmpIm.Canvas.CopyRect(RectT,BmpIm.Canvas,RectS); {копируем}
 BmpIm.Width:=BmpIm.Height; {усекаем до квадрата}
 Image1.Picture.Assign(BmpIm); {выводим BitMap на экран}
end;
```

Рис. 10.3. Образец для преобразования BitMap в иконку

Далее нам нужен BitMap размером 32×32, стандартный размер иконки. Можно все сделать в исходном BitMap, но мы его сохраним на всякий случай в неприкосновенности, и скопируем изображение в новый BitMap `IcoBmp`. Поставим на форму еще одну кнопку `Button3` (**Resize**) и напишем для нее такой обработчик:

```
procedure TForm1.Button3Click(Sender: TObject);
begin  {Resise уменьшаем до размера 32x32 }
 IcoBmp := TBitMap.Create;
     {создаем экземпляр объекта типа BitMap}
 IcoBmp.Height:=32; {новая картинка 32x32}
 IcoBmp.Width:=32;
 RectT:=Rect(0,0,32,32); {поле, куда копировать}
 IcoBmp.Canvas.StretchDraw(RectT,BmpIm);
       {копируем с усечением размеров}
 Image1.Proportional:=False; {демонстрировать будем}
 Image1.Stretch:=False; {в натуральном виде}
 Image1.Picture.Assign(IcoBmp); {выводим BitMap на экран}
 st:=ExtractFileName(filename); {меняем имя файла}
 filename:='0'+st;
  {чтобы не путать с исходным, папку можно не указывать}
 IcoBmp.SaveToFile(ChangeFileExt(filename,'.bmp'));
end;
```

Заметки на полях

Обратите внимание на один нюанс в обращении с областями типа TRect и параметрами Width и Height BitMap. В качестве конечной точки по X и по Y здесь указывается величина, отстоящая от начальной на величину *общего количества* точек (а не номер конечной точки, если считать начальную за нулевую). То есть верхняя граница будет на единицу больше, чем это было бы при указании границы массива. Далее, когда мы будем иметь дело с массивом Pixels канвы, там те же границы будут указываться нормально: (0..31), т. е. всего 32 элемента. Эту разницу легко понять из определений массива и прямоугольника из точек: массив massiv [0..1,0..1] имеет четыре элемента, а прямоугольник rect(0,1,0,1) — один-единственный!

Для контроля мы выводим полученный BitMap в файл с именем исходного, но начинающимся с символа "0". На экран мы выводим картинку в натуральном, а не растянутом виде, чтобы убедиться, что изображение действительно уменьшилось. Далее мы хотим посмотреть, как это будет выглядеть в растянутом виде — ведь именно IcoBmp у нас будет представлять потом маску XORMask. Для этого поставим на форму еще одну кнопку Button4 (**Stretch**) и напишем для нее такой обработчик:

```
procedure TForm1.Button4Click(Sender: TObject);
begin  {Stretch - растягиваем для наглядности}
 Image1.Proportional:=True;
 Image1.Stretch:=True;
 Image1.Repaint; {перерисовать заново}
end;
```

Результат демонстрации будущей иконки в увеличенном виде показан на рис. 10.4.

Рис. 10.4. Будущая иконка 32×32 в увеличенном виде

Теперь займемся конструированием ANDMask. Для этого придется создать монохромный битмэп и скопировать в него наше изображение. Добавим еще одну кнопку Button5 (**ANDMask**) и вот такой обработчик для нее:

```
procedure TForm1.Button5Click(Sender: TObject);
begin {Создаем маску And}
ANDMask := TBitMap.Create;
ANDMask.Assign(IcoBmp);
ANDMask.Monochrome := true; {монохромную}
Image1.Picture.Assign(AndMask); {выводим AND на экран}
end;
```

Вот что мы увидим на экране — рис. 10.5. В принципе нам сейчас не очень важно, какая именно маска получится — рисунок в ANDMask важен лишь при формировании "прозрачного" цвета, в остальных случаях, как мы говорили, можно сделать просто черный прямоугольник, а прозрачностью мы займемся позднее.

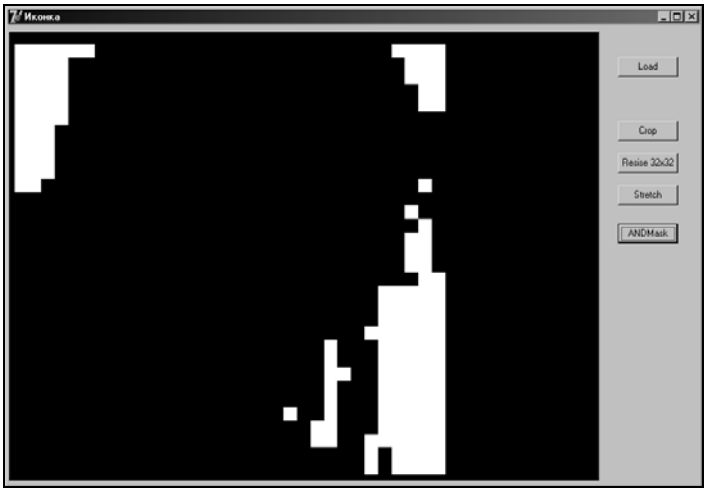


Рис. 10.5. Маска ANDMask в увеличенном виде

Теперь можно заняться, наконец, собственно иконкой. Создавать мы ее будем с помощью функции API `CreateIconIndirect`, которая требует заполнения структуры `ICONINFO` (устроенной много проще, чем структура `BITMAPINFO`). Поставим на форму еще одну кнопку `Button6` (далеко не последнюю, как вы увидите) и создадим обработчик щелчка для нее. Здесь я для наглядности скопировал картинку 32×32 из `IcoBmp` в новый BitMap `XORMask`, разумеется, можно обойтись и без этих промежуточных преобразований (и даже вообще использовать изначальный `BmpIm`, если его не жалко):

```
procedure TForm1.Button6Click(Sender: TObject);
begin {Icon - превращаем в иконку}
XORMask:= TBitMap.Create;
XORMask.Assign(IcoBmp); {в качестве XOR-маски используем наш BMP}
IcoIco := TIcon.Create;
IcoInfo.fIcon := True;
IcoInfo.xHotspot := 0;
IcoInfo.yHotspot := 0;
IcoInfo.hbmMask := ANDMask.Handle;
IcoInfo.hbmColor := XORMask.Handle;
IcoIco.Handle := CreateIconIndirect(IcoInfo);
IcoIco.SaveToFile(ChangeFileExt(OpenDialog1.filename,'.ico'));
Image1.Picture.LoadFromFile
     (ChangeFileExt(OpenDialog1.filename,'.ico'));
               {выводим иконку на экран}
end;
```

В качестве комментария к параметру IcoInfo.fIcon я привел строку из официального описания этой структуры: *"величина True специфицирует иконку"*. Два следующих параметра имеют значение, если мы создаем курсор, а не иконку. Имейте это в виду, если захотите создавать свои курсоры, они создаются в точности так же, как и иконка. Лично мне никогда этого делать не приходилось, потому что, на мой взгляд, всякая самодеятельность в этом отношении может только раздражать пользователей (и, например, в фирменных графических редакторах раздражает безумно). В 99% случаев достаточно обычного курсора в виде стрелки, а для оставшегося одного процента, когда это действительно необходимо, например в браузерах при указании на ссылку, Windows предлагает вполне приличный набор готовых.

Но результат, появившийся в Image1 после выполнения этой процедуры, нас обескуражит: иконка получится черно-белой (поэтому мы кнопку Button6 назвали **Icon B&W**). Проверьте полученный файл с именем oboi350.ico — в нем также окажется черно-белая иконка. Для того чтобы рассмотреть подробнее, что именно получилось, нам следует, во-первых, перезагрузить иконку из полученного файла, во-вторых, превратить ее в BitMap — при простом отображении иконки на канве она не растягивается, а демонстрируется только в оригинальном размере 32×32, изображение иконки на канве не зависит от свойства Stretch. Поставим на форму кнопку Button7 (**Icon Stretch**) и для нее напишем такой обработчик:

```
procedure TForm1.Button7Click(Sender: TObject);
begin {Icon Stretch - растягиваем готовую иконку}
IcoBmp.Destroy; {все заново}
IcoBmp:=TBitMap.Create;
IcoIco.Destroy;
IcoIco := TIcon.Create;
IcoIco.LoadFromFile(ChangeFileExt(OpenDialog1.filename,'.ico'));
IcoBmp.Width:=IcoIco.Width;
IcoBmp.Height:=IcoIco.Height;
IcoBmp.Canvas.Draw(0,0,IcoIco); {преобразуем в BitMap}
Image1.Picture.Assign(IcoBmp);{выводим ICO на экран}
end;
```

При рассмотрении в увеличенном виде окажется, что отдельные цветные пятна на женском портрете все же имеются. Это связано с тем, что исходное изображение было TrueColor, а в случае иконок Delphi разбирает только 16 базовых цветов, которых у нас в данном случае почти нет. Если же мы поэкспериментируем с другими примерами, то увидим, что они отображаются относительно нормально — логотип apple.jpg, например, вообще перенесется в иконку практически один к одному, даже с намеками на тени, из-за того,

что он практически из одних базовых цветов и состоит. Есть два пути решения этой проблемы — один заключается в том, чтобы самостоятельно, полностью с нуля, создать и записать файл иконки, использовав там то количество цветов, какое нам нужно. Неплохое описание подобного процесса имеется в базе FAQ по Delphi [24]. Этот путь вполне нормален для DOS (и автор в свое время создавал такие файлы, написав вполне работоспособный, хотя и не очень удобный DOS-редактор иконок), но мы же живем в эпоху Windows!

Попробуем просто сделать так, чтобы в нашей иконке содержались одни базовые цвета, т. е. реализуем операцию редукции цветов. Причем наша задача — именно свести палитру к базовой, а не просто сократить количество цветов до 16, как это делают графические редакторы по умолчанию. Преобразование исходной картинки в 16-цветную в каком-либо редакторе нам не поможет, потому что к базовому набору цветов палитру если и удастся свести, то результат будет весьма плачевный (см. в папке с проектом эксперименты с файлом oboi16col.bmp, полученным из нашего женского портрета редукцией цветов к базовым через Photoshop). Простое приравнивание палитры иконки палитре исходной XOR-маски также не помогает, даже если исходный файл 16-цветный — скорее всего, на исходную палитру загружаемого изображения переменная типа BitMap в Delphi просто не обращает внимания, всегда формируя 24-битное изображение. Можно попробовать создать отдельно только XOR-маску вручную (через функции API или непосредственно), но этот способ все равно не был бы универсальным, т. к. в формате JPEG вообще может присутствовать только 24-битное представление цвета, и огромное количество подготовительной работы по сведению их к 16 цветам и пересохранению в формате BMP обессмыслило бы всю затею автоматического формирования иконки.

Среди методов класса `TBitMap` есть процедура присвоения глубины цвета, например, можно попробовать записать `XORMask.PixelFormat:=pf4bit`. Но это совсем не процедура редукции — если попробовать присвоить это значение после того, как выполнена операция `Assign`, то вам укажут на ошибку, а до того — естественно, это никак не повлияет на результат. Спасибо, как говорится, и на том, что свойство `Monochrome` нормально работает (хотя создать маску AND ничего не стоит и вручную, все равно для "прозрачного" цвета ее придется править). Нам ничего не остается, кроме как придумать алгоритм редукции TrueColor к 16 базовым цветам самим.

Простой способ редуцировать трехбайтное представление цвета к четырехбитовому мне не известен и вряд ли он существует — когда выбирались базовые цвета для VGA-режима, думали, естественно, не об удобствах преобразования, а о том, чтобы они как можно шире охватывали реальный спектр. Однозначного алгоритма редуцирования разрядности цвета нет вообще —

в продвинутых графических редакторах вам всегда предлагают выбор. Поэтому в приложениях, которые могут выполняться на различных платформах (например, для HTML-страниц), даже рекомендовано использовать конкретные 216 "безопасных" оттенков, которые однозначно отображаются при преобразованиях цветов по цепочке 24-18-8 бит (см. **http://html.manual.ru/book/info/basecolors.php**). Если кто-то из читателей додумается, как это преобразование выполнить наиболее просто и адекватно, пусть напишет. Грамотным решением было бы использование приемов аналитической геометрии для разбиения цветового куба (трехмерной пространственной области, по осям которой отложены значения трех спектральных составляющих от 0 до 255) на 16 областей, близких к базовым цветам. Однако реализация этого алгоритма слишком сложна, и к тому же предполагает использование эвристических данных о восприятии цветов, чтобы имело смысл разбирать его в рамках данной книги.

Здесь мы сделаем это "лобовым методом", без учета зрительного восприятия и точного соответствия оттенков. Конечных вариантов всего 16, все они перечислены в табл. 10.1 и содержат, как мы видим, в основном три варианта значений байтов, равные $FF, $80 и $00, причем варианты совместного присутствия значений $FF и $80 не встречаются, зато имеется отличный от указанных вариант со светло-серым цветом. Последним мы для простоты пожертвуем (он у нас будет воспроизводиться как белый, и это несколько подпортит всю картинку, как мы увидим далее). Алгоритм реализуем в два этапа. На первом поделим все варианты цветов на "светлые" и "темные" по условию: если хотя бы одна из составляющих больше 128, то этот цвет — "светлый", в противном случае — "темный". На втором этапе проанализируем отдельно каждую составляющую. Если общий цвет был классифицирован как "темный", то она может принимать значения в диапазоне 0—128. Разобьем этот диапазон ровно пополам, и будем составляющей присваивать значение $80, если она больше половины, и 0, если меньше. Для "светлых" тонов отдельные составляющие могут лежать в диапазоне 0—255. Также разобьем этот диапазон пополам, и будем присваивать составляющей значение $FF, если она больше половины, и 0, если меньше. Так мы исключим сочетания $80 и $FF, в противном случае 12 вариантов (всего 27 вариантов, нас устраивает 15) пришлось бы отсеивать по отдельности.

Простой доступ к цвету каждого пиксела (т. е. к его значению в 24-битной палитре), по счастью, у нас имеется — это массив `Pixels` канвы (манипуляциями с этим массивом, кстати, можно заменить большинство штатных методов канвы, только выполняться эти операции будут намного дольше оригинальных, основанных на прямом обращении к видеопамяти). Поставим на форму кнопку `Button8` (**XOR Color**) и напишем такую процедуру:

```
procedure TForm1.Button8Click(Sender: TObject);
var xcol,x,y:integer; {XOR Color меняем цвета}
var colR,colG,ColB:byte;
begin
for x:=0 to 31 do
for y:=0 to 31 do {по всему массиву точек}
begin
  xcol:=XORMask.Canvas.Pixels[x,y];
  colR:=Lo(xcol);
  colG:=Lo(xcol shr 8);
  colB:=Lo(xcol shr 16); {RGB-составляющие}
  if (colR>$80) or (colG>$80) or (colB>$80) then
  begin {светлый цвет}
   if colR>$80 then colR:=$FF else colR:=0;
   if colG>$80 then colG:=$FF else colG:=0;
   if colB>$80 then colB:=$FF else colB:=0;
  end else {темный цвет}
  begin
   if colR>$40 then colR:=$80 else colR:=0;
   if colG>$40 then colG:=$80 else colG:=0;
   if colB>$40 then colB:=$80 else colB:=0;
  end;
  xcol:=colR+256*colG+65536*colB; {собираем цвет обратно}
  XORMask.Canvas.Pixels[x,y]:=xcol;
end;
Image1.Picture.Assign(XORMask); {выводим XOR на экран}
end;
```

Результат (рис. 10.6) в данном случае будет, прямо скажем, "не очень" — хотя и несколько лучше, чем при экспериментах с Photoshop (сравните иконки в файлах oboi350.ico и oboi16col.ico). Читатели самостоятельно могут поэкспериментировать с указанным алгоритмом, с целью улучшить качество изображения (например, хотя бы ввести светло-серый). А вот с другими образцами все куда лучше, хотя с ними и изначально было неплохо (рис. 10.7) — что есть следствие наличия нужных или близких цветов в оригинальном изображении. Для логотипа Apple, например, всего лишь уберется тень, что даже лучше при последующем преобразовании в "прозрачную" иконку. Для большего понимания, что происходит с масками при их взаимодействии, сравните все увеличенные результаты для масок AND, XOR и самой иконки для образца с логотипом Apple.

Но никто и не утверждал, что *любой* JPEG можно автоматически превратить в иконку и при этом сразу получить идеальный результат. Для достижения

Рис. 10.6. Результат редукции цветов для женского портрета

а

б

Рис. 10.7. Иконка бабочки в натуральном виде (*а*) и после редукции цветов (*б*)

абсолютно адекватного результата недостаточно даже ручного редактирования, получить его можно только, если использовать иконки TrueColor. Но и тогда для получения более-менее приемлемого результата ручное вмешательство потребуется, и для этого созданную нами программу стоит превратить в нормальный редактор иконок. Это теперь несложно, причем с реализацией встречающейся достаточно редко функции превращения в иконку произвольного изображения. Читатель уже вполне может самостоятельно с этим справиться.

Для того чтобы переделать иконку окончательно, поставим еще одну кнопку `Button9` (**Icon**) и напишем такой обработчик:

```
procedure TForm1.Button9Click(Sender: TObject);
begin {Icon создаем иконку заново}
IcoIco.Destroy; {будем создавать заново}
IcoIco := TIcon.Create;
IcoInfo.fIcon := True;
IcoInfo.xHotspot := 0;
IcoInfo.yHotspot := 0;
IcoInfo.hbmMask := ANDMask.Handle;
IcoInfo.hbmColor := XORMask.Handle;
IcoIco.Handle := CreateIconIndirect(IcoInfo);
IcoIco.SaveToFile(ChangeFileExt(OpenDialog1.filename,'.ico'));
Image1.Picture.LoadFromFile
    (ChangeFileExt(OpenDialog1.filename,'.ico'));
                {выводим иконку на экран}
end;
```

Для того чтобы завершить тему создания иконок, нужно ответить на вопрос — как реализовать "прозрачный" цвет? Так как мы имеем все маски, то проще всего сделать это вручную, преобразованием пикселов. Мы покажем процедуру опять же на примере автоматического преобразования — попробуем осуществить замену заданного цвета на "прозрачный".

Чтобы не возиться с `ColorBox` или другими вариантами "цветовых" компонентов в этой пародии на настоящее приложение, которое мы создаем, для задания цвета поставим на форму компонент `ColorDialog` и только заменим у него свойство `Color` на `clWhite`. Решение очень неаккуратное, но для нас здесь, как временное решение, сойдет — мы хотим лишь показать, как и что делать в соответствующей ситуации. Поставим на форму очередную кнопку `Button10` (**Icon**) и вот как будет выглядеть обработчик для нее:

```
procedure TForm1.Button10Click(Sender: TObject);
var x,y:integer;
```

```
begin {прозрачный цвет}
if ColorDialog1.Execute then
begin
 for x:=0 to 31 do
 for y:=0 to 31 do {по всему массиву точек}
 if XORMask.Canvas.Pixels[x,y]= ColorDialog1.Color then
 begin
  XORMask.Canvas.Pixels[x,y]:=0;
  ANDMask.Canvas.Pixels[x,y]:=clWhite;
 end;
 IcoIco.Destroy; {будем создавать заново}
 IcoIco := TIcon.Create;
 IcoInfo.fIcon := True;
 IcoInfo.xHotspot := 0;
 IcoInfo.yHotspot := 0;
 IcoInfo.hbmMask := ANDMask.Handle;
 IcoInfo.hbmColor := XORMask.Handle;
 IcoIco.Handle := CreateIconIndirect(IcoInfo);
 IcoIco.SaveToFile(ChangeFileExt(OpenDialog1.filename,'.ico'));
 Image1.Picture.LoadFromFile(ChangeFileExt
       (OpenDialog1.filename,'.ico'));
                {выводим иконку на экран}
end;
end;
```

Чтобы понять, что именно тут делается, обратитесь к описанию принципа совместной работы масок AND и XOR ранее. Осталось только показать, как лихо можно подменить иконку самого приложения прямо на ходу. Поставим на форму последнюю кнопку `Button11` (**USE**). Обработчик будет состоять всего из одной строки:

```
procedure TForm1.Button11Click(Sender: TObject);
begin {USE - применить иконку к приложению}
 Application.Icon:=IcoIco;
end;
```

После нажатия на кнопку **USE** текущая иконка, которая содержится в `IcoIco`, сразу появится в заголовке формы.

Заметки на полях

Читатель вправе задать вопрос — а почему для задания "прозрачного" цвета нельзя воспользоваться удобным механизмом, который предлагают свойства `TransparentColor` и `TransparentMode` класса `TBitMap`? Дело в том, что они работают только при отображении BitMap на экране и никак не сказываются на представлении карты битов в памяти, т. е. при сохранении в файл, копировании

в другой BitMap или переносе в иконку эти свойства "пропадают". Попробуйте внести в самую первую процедуру по нажатию Button1 следующие строки (их надо вставить непосредственно перед загрузкой картинки в Image1 операцией Assign):

```
BmpIm.TransparentColor:=clWhite;
Image1.Transparent:=True;
```

При загрузке женского портрета белый фон станет "прозрачным" (выглядеть это будет довольно неаккуратно, т. к. отдельные области фона, на первый взгляд тоже белые, на самом деле таковыми не являются). Но уже на третьем шаге (при переводе в размер 32×32) прозрачность "куда-то" исчезнет. Заметим, что отсутствие функции указания диапазона цветов сильно ограничивает возможности механизма, фактически он применим только к картинкам с однородными по цвету полями (даже логотип Apple из-за наличия теней не будет отображаться как надо). Если вы хотите отображать с эффектом прозрачности произвольные картинки с широкой гаммой цветов, то вам либо придется кропотливо заменять нужные цвета (например, все светлые) на какой-то один, который потом объявляется "прозрачным", либо не миновать "ручной" процедуры создания масок по типу, как мы это делали для иконки. Создавать саму иконку, конечно, не нужно, маски AND и XOR (произвольного размера) комбинируются с фоном также вручную с использованием функции API BitBit или даже проще — путем задания свойства CopyMode при использовании процедуры CopyRect.

Дадим здесь и пару советов, как правильно пользоваться механизмом отображения "прозрачных" BitMap. Вопреки распространенному мнению, если вы задаете свой собственный цвет, который должен стать прозрачным, параметр TransparentMode специально устанавливать в значение tmFixed не надо — это происходит автоматически при установке свойства TransparentColor. Единственное, когда требуется установка TransparentMode — когда вы отказываетесь от собственного цвета, вы задаете свойству значение tmAuto, и цвет будет определяться левым нижним пикселом изображения (именно нижним, хотя в фирменной справке все запутали, указав на два возможных варианта — в Windows, как мы знаем, формат BMP начинается с нижней строки). В этом случае, наоборот, устанавливать свойство TransparentColor уже нельзя. Кроме этого, чтобы отображение "прозрачного" цвета прошло успешно, надо не забывать устанавливать свойство Transparent соответствующего компонента в True (у нас это Image1).

Если стоит задача изменить готовую иконку, то ее надо загрузить из файла, как мы делали раньше. Получить доступ к маске XOR можно также описанным ранее способом через процедуру Draw канвы любого BitMap, а вот с AND-маской придется повозиться. Сначала нужно получить структуру ICONINFO:

```
IcoInfo : TIconInfo;
IcoIco : TIcon;
. . . . .
GetIconinfo(IcoIco.Handle,IcoInfo);
```

В результате в IcoInfo окажется структура ICONINFO нашей иконки, а в ней, как мы уже знаем, содержатся дескрипторы масок под названиями hbmMask (маска AND) и hbmColor (маска XOR). Далее необходимо извлечь информацию по нужному дескриптору, причем просто приравнять BmpIm.Handle:= IcoInfo.hbmMask не выйдет, надо действовать через глобальную кучу. Так что в самом простом случае при изменениях готовой иконки проще будет сформировать AND-маску заново — кажется, именно так и поступал Image Editor старых версий Delphi.

В целом наша программа, как готовое приложение, разумеется, никуда не годится — если сразу после запуска случайно нажать на любую из кнопок, кроме **Load**, получите кучу претензий за обращение к несуществующим объектам. Но мы ее здесь создавали только в качестве иллюстрации того, как вообще надо обращаться с иконками и BitMap.

Приложение-термометр с иконкой в Tray

В программе, которую мы сейчас будем разбирать, использованы многие наши предыдущие достижения. Идея программы такая: вывести в Tray индикатор, показывающий температуру, значение которой мы получаем с некоего датчика.

Термометр

Сначала о датчике. Мы постараемся построить программу так, чтобы в ней можно было использовать данные с любого датчика (необязательно температуры). Условие одно — данные должны укладываться в четыре символьных знака, не считая разделительной запятой. Для температуры это будет представление ее в виде "36,6°". Точность (точнее — разрешающая способность) в 0,1° может показаться излишней, но в книге *"Занимательная электроника"* я уже писал, что представление температуры без десятых градуса — все равно, что наручные часы без секундной стрелки, вроде бы "по жизни" и не очень надо, но как-то... некрасиво.

В данном случае я предлагаю использовать возможность, которая имеется почти в любом компьютере — а именно игровой порт. В минимальной стандартной конфигурации игровой порт содержит целых четыре готовых аналого-цифровых преобразователя, не слишком высокого качества, но для того чтобы демонстрировать температуру в пределах от 0 до 99 градусов с приемлемой погрешностью в 1—2 градуса, они вполне подойдут. Так что тем, кто захочет повторить эту программу в точности, придется браться за паяльник — но предварительно нужно приобрести разъем типа DB-15M — ответную часть ("папу") 15-контактного разъема для стандартного игрового порта.

Кроме этого, в качестве собственно датчика потребуется терморезистор (термистор) с номинальным сопротивлением примерно 50—100 кОм при 20° (его можно приобрести там же) и один постоянный резистор любого номинала от 1 до 100 кОм (в принципе без последнего можно и обойтись).

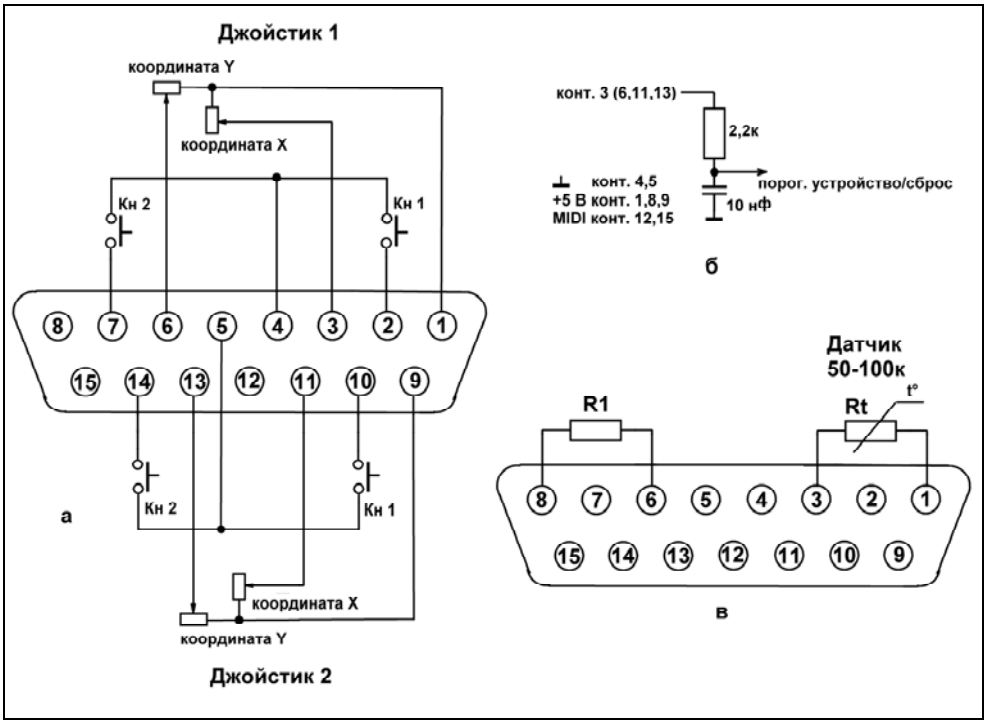


Рис. 10.8. Подключение джойстика к разъему игрового порта (*а*), внутреннее устройство входа АЦП (*б*) и подключение термистора к разъему (*в*). Нумерация соответствует виду спереди на гнездовую (компьютерную) часть разъема DB-15F

Штатное подключение джойстика к разъему игрового порта показано на рис. 10.8, *а*. На рис. 10.8, *б* изображен фрагмент внутренней схемы порта, относящийся к измерению сопротивления. Принцип измерения основан на отсчете времени, необходимого для зарядки конденсатора емкостью 10 нФ от источника питания +5 В через последовательно включенные внутренний резистор 2,2 кОм и измеряемый резистор (т. е. либо штатный потенциометр джойстика, обычно имеющий сопротивление около 100 кОм, либо наш термистор). В начале цикла измерения конденсатор перемыкается накоротко, чтобы сбросить его до нуля, затем начинается зарядка. Когда конденсатор заряжается от 0 В до некоего порога (примерно до середины напряжения питания +5 В — в качестве порогового устройства выступает обычный вход ло-

гического элемента), то выходная схема выдает сигнал об окончании периода зарядки. И далее цикл измерения повторяется заново (такой принцип преобразования аналоговой величины в интервал времени называется *прямое однократное интегрирование*). Если внешнее сопротивление равно 0, то время зарядки будет примерно равно 22 мкс (*RC*), при увеличении сопротивления время пропорционально увеличивается. Согласно описанию порта, измеренная величина времени должна быть на 10% больше величины *RC* — официальная формула почему-то дает сверхточную величину 24,2 мкс, что, конечно же, чистый выпендреж — на практике эта величина может меняться процентов на 30—50 в обе стороны. В Windows есть, разумеется, штатная процедура API, которая дает уже пересчитанную экранную координату джойстика.

Схема подключения термистора Rt к разъему показана на рис. 10.8, *в*. Мы использовали в качестве рабочей X-координату джойстика 1, но можно, конечно, и любой другой вход. Датчик подключается вместо переменного резистора джойстика, между входом схемы (контакт 3) и питанием +5 В (контакт 1). Постоянный резистор (R1 на рисунке) нужен, чтобы вторая координата того же джойстика также была подключена к питанию — в принципе можно и просто соединить указанные контакты перемычкой. Если вы этого не сделаете, то система может решить, что "джойстик" не подключен вообще.

Заметки на полях

Считанный интервал времени можно измерить самостоятельно в условных единицах (или, более грамотно, — с использованием системных таймеров), если запускать преобразование и ожидать его окончания, отсчитывая время. Делается это непосредственно через порт, записью/чтением его регистра по адресу 201h. Простейший вариант функции с использованием встроенного ассемблера выглядит примерно так (вначале мы запускаем преобразование, потом ждем, пока система выдаст сигнал, что оно окончено. Если, например, через 10 000 циклов — эта величина зависит от быстродействия компьютера — мы так и не дождались окончания, выходим из процедуры):

```
function jctk: word;
begin
asm
  mov @result,0
  mov dx,201h
  mov cx,0
  out dx,al
@mc: in al,dx
   inc cx
   cmp cx,10000
   ja @mend
   test al,1
   jne @mc
   mov @result,cx
```

```
@mend: nop
end;
end;
```

Эта процедура вполне будет работать и в Windows 9*x*. Аналоговые входы игровых портов можно использовать и для измерения напряжения, подавая его на нужный контакт через последовательный резистор (иначе разрешающая способность будет очень мала). Только следует учесть, что диапазон измерений здесь будет ограничен снизу напряжением примерно 2,8—3,2 В, иначе конденсатор никогда не зарядится до порогового значения. Зато выше этого порога можно подавать в принципе любое напряжение — такое, чтобы мощность на внутреннем резисторе 2,2 кОм (произведение падения напряжения на нем на ток через входные резисторы) не превысила 0,1 Вт. Разумеется, особенной линейности ждать от такого измерителя не приходится — но в компьютере это и не принципиально, ведь пересчитывать полученное значение можно по формулам любой степени сложности, достаточно измерительный канал один раз откалибровать.

Зависимость сопротивления от температуры для термисторов (полупроводниковых терморезисторов) имеет отрицательный наклон и описывается довольно сложной формулой:

$$Rt = A \cdot e^{\left(\frac{B}{T-273}\right)},$$

где A и B — константы, а T — абсолютная температура.

Так как сопротивление пропорционально величине, которую мы получаем из опроса джойстика, то формула для пересчета будет точно такой же, только величины коэффициентов изменятся. Обозначив величину, получаемую в виде координаты джойстика, через *xt*, мы в результате преобразования приведенной формулы выведем такую зависимость значения температуры (в градусах Цельсия) от "координаты":

$$t = \frac{B}{\ln(xt) - \ln(A)} - 273.$$

Коэффициенты A и B мы сначала зададим теоретически, а точные их значения получим путем калибровки с помощью самой программы.

Приложение

Теперь отвлечемся от электроники и вернемся "к нашим баранам": поговорим о том, как формировать иконку со значениями градусов. Можно превратить в BitMap текст, набранный в любом компоненте, а потом этот BitMap — в иконку, но смысла никакого в такой сложной операции нет — на канве BitMap выводить символы куда проще, и результат будет более предсказуем. Создадим новый проект, назовем его Temperatura (модуль Tempr, папка

Glava10\2) и первым делом напишем процедуру, которая бы создавала иконку с надписью из заданной строки. Иконка пусть будет 16×16 (для Tray больше не надо). Для простоты надпись будет черным на прозрачном фоне (тогда можно обойтись одной монохромной AND-маской — черная надпись на белом фоне, а XOR-маску формировать, как просто черный квадрат). Объявим следующие переменные:

```
var
Form1: TForm1;
IcoBmp,XORmask,ANDmask: TBitMap; {переменные типа BitMap}
gIcon,dIcon: TIcon; {объект типа иконка}
IcoInfo : TIconInfo;
aRect:TRect;
```

Процедура будет выглядеть так:

```
procedure CreateIconDegree(symb:string; var Icon:TIcon);
{создание значка с надписью из symb}
begin
 try
  Icon.Destroy; {если она существовала - удаляем}
 except
 end;
 with IcoBmp do
 begin
 Height:=16; {новая картинка 32x32}
 Width:=16;
 Canvas.Brush.Color:=clWhite; {будущая маска AND}
 aRect:=Rect(0,0,17,17);
 Canvas.FillRect(aRect);
 Canvas.Font.Name:='MS Sans Serif';
 Canvas.Font.Size:=9;
 Canvas.Font.Color:=clBlack;
 Canvas.TextOut(1,1,symb); {выводим текст значка}
 end; {создали белую картинку с текстом}

 ANDMask.Assign(IcoBmp);
 ANDMask.Monochrome := true; {монохромный}
 XORMask.Width:=16;
 XORMask.Height:=16;
 XORMask.Canvas.Brush.Color:=clBlack;
 XORMask.Canvas.FillRect(aRect);
 {обеспечиваем прозрачность полностью черной маской XOR}
```

```
 Icon:=TIcon.Create; {создаем иконку}
 IcoInfo.fIcon := True;
 IcoInfo.xHotspot := 0;
 IcoInfo.yHotspot := 0;
 IcoInfo.hbmMask := ANDMask.Handle;
 IcoInfo.hbmColor := XORMask.Handle;
 Icon.Handle := CreateIconIndirect(IcoInfo); {создали}
end;
```

В квадрате 16×16 пикселов указанным шрифтом MS Sans Serif 9 пунктов (для приблизительных расчетов можно считать, что размер шрифта в пунктах примерно равен высоте его символов в пикселах) уместится 2 обычных знака плюс разделительная запятая (если от нее отказаться, то можно уместить и шрифт в 10 пунктов). Шрифт, его цвет, размер и т. п. нужно всегда указывать явно, непосредственно перед использованием, не полагаясь на установки по умолчанию, которые запросто могут быть изменены "кем-то где-то" — то же самое относится и к установкам кисти Brush или пера Pen, автор не раз "накалывался" в таких случаях, стараясь сократить объем кода программы.

Для показа температуры в том формате, который мы задумали, нам придется создавать две иконки и располагать их рядом. Теперь надо оформить процедуру демонстрации иконок. Сделаем так: при нажатии кнопки минимизации приложение будет сворачиваться в Tray, а в обычном режиме иконки пусть демонстрируются в окне. Добавим обычные наши переменные, которые скопируем из проекта в *главе 3* (дескриптор иконки нам не понадобится, он уже у нас есть):

```
. . . . .
 Ico_Message: integer=wm_User; {сообщение от иконки окну}
 NoIconData: TNotifyIconData; {дескриптор структуры}
 FHandle: HWnd; {дескриптор окна}
. . . . .
```

А в список процедур в секции объявлений добавим две необходимые процедуры:

```
. . . . .
type
TForm1 = class(TForm);
. . . . .
procedure WndProc(var Message: TMessage);
procedure OnMinimizeProc(Sender:TObject);
. . . . .
```

Заметки на полях

У читателя, вероятно, возникает вопрос — а почему я иногда располагаю объявления процедур или даже переменных в секции методов и свойств класса (т. е. между ключевыми словами **class...end**), а иногда — просто так, где обычно? Например, процедура OnMinimizeProc из примера в *главе 4* есть метод формы Form1. А процедура CreateMyicon в том же примере — обычная процедура, как структурная единица программы. И если вы используете эти процедуры в другом модуле, то первую надо вызвать, как Form1.OnMinimizeProc, а вторую — как просто CreateMyicon. Есть ли какая-нибудь разница? Если вы создаете свой собственный компонент или библиотеку, то с этим делом вам придется знакомиться подробнее [3, 30], а в обычном приложении (тем более, если форма всего одна) это почти безразлично, за некоторыми важными исключениями, например, обработчики событий всегда нужно размещать в описании класса формы. Но нет смысла вдаваться в эти нюансы подробнее: в начале книги я обещал, что изучением ООП мы заниматься не будем. Если вы совершите ошибку, вам на нее укажут, не сомневайтесь.

Далее создадим обработчик события onCreate формы, в котором будем заодно создавать BitMap-объекты (иконки мы ранее создаем в процедуре):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 ANDMask := TBitMap.Create;
 XORMask:= TBitMap.Create;
 IcoBmp := TBitMap.Create; {создали IcoBmp }
 Application.onMinimize := OnMinimizeProc;
 CreateIconDegree('00',dIcon);{создание иконки 1}
 CreateIconDegree('00',gIcon);{создание иконки 2}
end;
```

Иконки (какие-нибудь) создаем сразу, т. к. в дальнейшем мы их будем создавать по таймеру, и попытка минимизации приложения в самый первый момент может привести к его краху. Теперь распишем процедуры сворачивания и восстановления окна с расположением двух иконок в Tray:

```
procedure TForm1.OnMinimizeProc(Sender: TObject);
begin
  Form1.Hide; {скрыли окно}
  FHandle := AllocateHWnd(WndProc); {получаем дескриптор окна}
  with NoIconData do begin
    cbSize:=Sizeof(TNotifyIconData); {размер структуры}
    Wnd:=FHandle; {дескриптор окна}
    uID:=0; {первая иконка}
    UFlags:=NIF_MESSAGE or NIF_ICON or NIF_TIP;
          {взводим все флаги}
```

```
    SzTip:='Температура'; {всплывающая подсказка}
    HIcon:=dIcon.Handle; {дескриптор иконки целых градусов}
    uCallBackMessage:=Ico_Message;
       {определяемое пользователем сообщение}
  end;
  Shell_NotifyIcon(NIM_ADD,@NoIconData); {создали иконку}
  with NoIconData do begin
    cbSize:=Sizeof(TNotifyIconData); {размер структуры}
    Wnd:=FHandle; {дескриптор окна}
    uID:=1; {вторая иконка}
    UFlags:=NIF_MESSAGE or NIF_ICON or NIF_TIP;
          {взводим все флаги}
    SzTip:='Температура'; {всплывающая подсказка}
    HIcon:=gIcon.Handle; {дескриптор иконки десятых градуса}
    uCallBackMessage:=Ico_Message;
       {определяемое пользователем сообщение}
  end;
  Shell_NotifyIcon(NIM_ADD,@NoIconData); {создали иконку}
end;

procedure TForm1.WndProc(var Message: TMessage);
     {обработка пользовательских сообщений}
begin
  if Message.Msg = Ico_Message then
   if (Message.lParam=WM_LBUTTONUP) or
      (Message.lParam=WM_RBUTTONUP) then
   {была отпущена какая-нибудь кнопка мыши}
   begin
    Form1.Show; {восстанавливаем окно}
    Application.Restore; {восстанавливаем все параметры}
    Application.BringToFront; {помещаем поверх всех окон}
    DeallocateHWnd(FHandle);
     {убираем из памяти дескриптор окна}
    NoIconData.uID:=1;
    Shell_NotifyIcon(NIM_Delete,@NoIconData);
          {удаляем первую иконку}
    NoIconData.uID:=0;
    Shell_NotifyIcon(NIM_Delete,@NoIconData);
          {удаляем вторую иконку}
    Application.ProcessMessages;
    {на всякий случай обрабатываем системные сообщения}
   end;
end;
```

Уничтожать все созданные объекты тоже надо, это будем делать по событию onDestroy:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
try
 ANDMask.Destroy; {подобрали мусор}
 XORMask.Destroy;
 IcoBmp.Destroy;
 dIcon.Destroy;
 gIcon.Destroy;
except
end;
end;
```

Осталось расписать процедуру собственно формирования иконок с нужными цифрами. Для этого, во-первых, поместим на форму компонент Timer, во-вторых, обычный Label, на канве которого мы будем эти иконки демонстрировать. Компонент Image здесь использовать неудобно — в нем вечно возникают какие-нибудь проблемы при попытке вывести одновременно два разных изображения. Label1 покрасим в белый цвет (установим его для красоты сразу на панель Panel1, на которой потом будут еще элементы). Процедура по событию таймера будет выглядеть так:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
 t:=t+0.1; xt:=round(t); {проверка работы}
 if t>99 then t:=99;
 if t<0 then t:=0;
 st:=FloatToStrF(t,ffFixed,3,1);
 if pos(',',st)<>0 then {русская запятая}
 begin
  st1:=copy(st,1,pos(',',st));
  st2:=copy(st,pos(',',st)+1,1)+'°';
 end
 else {английская точка}
 begin
  st1:=copy(st,1,pos('.',st));
  st2:=copy(st,pos('.',st)+1,1)+'°';
 end;
 if length(st1)<3 then st1:='0'+st1;
 CreateIconDegree(st1,dIcon);{создание иконки целых градуса}
 CreateIconDegree(st2,gIcon);{создание иконки десятых градуса}
```

```
 if Application.Active then {если окно активно}
 begin
  Label1.Canvas.Brush.Color := ClWhite; {очистить канву}
  Label1.Canvas.FillRect(Label1.Canvas.ClipRect);
  Label1.Canvas.Draw(20,2,dIcon); {отображаем в Label}
  Label1.Canvas.Draw(36,2,gIcon);
 end
 else {иначе, если окно скрыто}
 begin
  NoIconData.uID:=0;
  NoIconData.HIcon:=dIcon.Handle;
     {дескриптор иконки целых градуса}
  Shell_NotifyIcon(NIM_Modify,@NoIconData);
       {меняем первую иконку}
  NoIconData.uID:=1;
  NoIconData.HIcon:=gIcon.Handle;
     {дескриптор иконки десятых градуса}
  Shell_NotifyIcon(NIM_Modify,@NoIconData);
       {меняем вторую иконку}
 end;
end;
```

Разумеется, в список переменных надо добавить температуру `t`, значение "координат" джойстика `xt` и вспомогательные строки:

```
. . . . .
xt:integer;
t:extended;
st,st1,st2:string;
. . . . .
```

Температура должна быть типа именно `extended` (а не `double`, и тем более не `real`, `currency` или `comp`), т. к. для всех величин с плавающей точкой в этой программе (их будет еще несколько) потребуется максимально возможная разрядность с целью минимизации возможных ошибок при расчете по сложной формуле, приведенной ранее. Если вы прикинете порядки коэффициентов A и B, подставив в формулу реально измеренные величины, то увидите, что коэффициент A имеет значащие цифры в третьем-четвертом порядке после запятой, а B — наоборот, составляет сотни и тысячи. На самом деле представление в виде `extended` ничуть не удлиняет время на расчеты, т. к. блок выполнения операций с плавающей точкой в любом случае все приводит к форме `extended`, а остальные форматы существуют только для удобства.

Пока мы тут просто имитируем поступление данных (первая строка процедуры по таймеру) с целью отработки алгоритма. В дальнейшем производится

перевод значения температуры в строку с фиксированным количеством знаков, и эта строка уже разбивается на две — целой части с запятой, и дробной части с добавлением значка градусов. Остальное понятно из комментариев в листинге.

Заметки на полях

А зачем в программе разбирается так много вариантов представления числа с плавающей точкой в виде строки? В своей ориентированности на "чайников" разработчики Windows (и вслед за ними — Delphi) совершенно забыли, что если уж вводить разные региональные стандарты представления чисел, то неплохо бы обеспечить их взаимную совместимость при выполнении стандартных процедур. Если вы введете в русской версии Windows число в привычном для любого компьютерщика виде с разделительной точкой, то будете очень удивлены результатом: нет такого формата для числа! Но при чтении процедурами Delphi, например, из файла, все останется нормально — т. е. действует всегда только точка. Разумеется, это можно изменить в **Региональных установках** через **Панель управления**, но кто же, во-первых, этим будет заниматься, во-вторых — тогда наоборот, запятая не будет восприниматься, как легальный знак. Правильным подходом к решению этой проблемы было бы чтение из реестра установленного в системе формата (возможно, именно это и имели в виду упомянутые разработчики) — но в данном случае варианта всего два, так что короче будет их просто перебрать. И неужели было бы не проще вообще не городить эту проблему, а приводить к единому виду системными способами — в каждой локализации своим, вдруг австралийским аборигенам или индейцам племени Юта окажется привычнее отделять дробную часть апострофом или кавычкой?

В данном случае нам всего лишь пришлось загромождать программу лишними условиями, но еще намного хуже ситуация со стандартами даты и времени, которых может существовать несчетное количество. Так, в английской версии Windows XP формат даты устанавливался по умолчанию по такому образцу: "12 okt 04". Ни одна из дельфийских функций перевода строки в дату-время такой формат не принимает — зато обратные функции (дату-время в строку) в таком формате дату отлично выдают. Там есть выход из ситуации — использование функции `FormatDayTime` для единообразного представления — однако мне по-прежнему не ясно, почему проблемы разработчиков Windows перекладываются на плечи прикладников, ведь они могут просто не иметь понятия о том, что данная величина может меняться в локализованных версиях. Интересно, а в арабских версиях сама запись чисел ведется европейскими знаками, или тоже локализуется?

Теперь программу можно запускать в пробном варианте — она будет нам демонстрировать либо в компоненте `Label1`, либо — после минимизации — в Tray посекундно возрастающие числа от 0,1 до 99,9 со значком градуса (см. рис. 10.9, *б*). Осталось организовать только процедуру обращения к джойстику с расчетом значения температуры по формуле выше, и процедуру калибровки. Для этого сначала внесем в список такие переменные:

```
. . . . .
t1,t2,xt1,xt2:extended;
```

```
A:extended=0.0281; {ориентировочные значения коэффициентов}
B:extended=3050;
FlagC:Byte=0;
. . . . .
```

Ориентировочные значения коэффициентов A и B получены вот из каких соображений. Входной у нас является величина экранной координаты, которая может принимать значения от 0 до некоего верхнего предела, превышающего реально установленный его размер в пикселах. Так как сопротивление зависит от температуры с отрицательным наклоном, то меньшим значениям xt будет соответствовать более высокая температура. Нулю сопротивление датчика не может быть равно, примем "от фонаря", что температуре в 100 градусов (t1) соответствует значение xt в 100 единиц (xt1), а температуре 0 градусов (t2) — значение xt в 2000 единиц (xt2). Тогда, исходя из формулы зависимости сопротивления термистора от температуры, приведенной ранее, можно рассчитать значения коэффициентов по следующему алгоритму:

```
t1:=t1+273; t2:=t2+273; {перевели в абсолютную температуру}
B:=t1*t2*ln(xt1/xt2)/(t2-t1); {рассчитали B}
A:=xt1/exp(B/t1); {рассчитали A}
```

Полученные в результате значения и подставлены в формулу выше (в программе мы потом используем этот алгоритм в процедуре калибровки). А сама процедура обращения к джойстику выглядит проще простого:

```
procedure jctk; {получение данных с игрового порта}
var
joy: TJoyInfo;
begin
if JoyGetPos(JoystickId1,@Joy)=JOYERR_NOERROR then
begin
 xt:=joy.wXpos; {в xt - величина с джойстика}
 t:=(B/(ln(xt)-ln(A)))-273; {пересчитанное значение температуры}
end
else begin xt:=0; t:=0; end; {если джойстик недоступен}
end;
```

В предложение **uses** нужно добавить модуль mmsystem. Второй джойстик, если вы захотите его использовать, будет иметь идентификатор JoystickId2. Если функция не заработает сразу, то дело, скорее всего, не в вашей программе, а в Windows — проверьте, имеется ли джойстик в списке оборудования. Попробуйте соединить с питанием также обе координаты второго джойстика (см. рис. 10.8). Имейте в виду, что коды ошибок, которые возвращает функция JoyGetPos, кроме JOYERR_NOERROR, здесь не совпадают с определен-

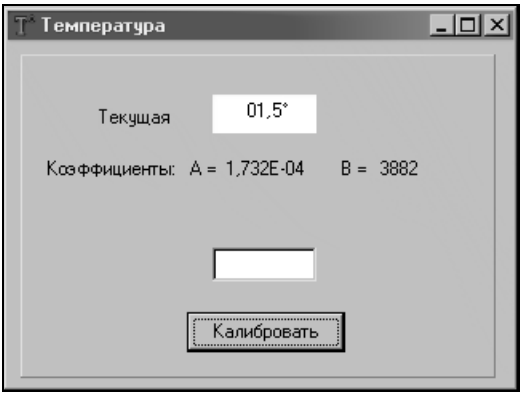
ными для нее Microsoft, их можно посмотреть в исходнике модуля mmsystem.pas (..Delphi7\Source\Rtl\Win).

Если у вас все готово (спаянный разъем вставлен куда надо), можете закомментировать первую строку в процедуре таймера и поставить вместо нее вызов процедуры `jctk` (на прилагаемом диске сделано наоборот — с целью обеспечить беспроблемный пробный запуск). Как видите, можно сюда подставить любую процедуру, получающую данные с какого-то датчика, лишь бы у нее выходная величина оказывалась в переменной `t` и укладывалась в диапазон от 0 до 9999 (или от –999 до +999). Запятую можно, естественно, убрать или перенести в другое место, значок градуса заменить еще на одну значащую цифру, а первый символ заменить на минус. Предлагаю читателю самому поразмыслить над вопросом, что и как придется усложнить в программе, чтобы демонстрировать отрицательные температуры (это не такой уж и очевидный вопрос, как представляется на первый взгляд). В дальнейшем мы еще остановимся на способах ввода данных в компьютер (через COM-порт), а сейчас организуем процедуру калибровки — без нее ничего не выйдет, т. к. при запуске программа наверняка будет показывать "погоду на Марсе".

Сама процедура калибровки заключается в том, чтобы зафиксировать величины `xt1` и `xt2`, получаемые с датчика при двух известных температурах `t1` и `t2`, и рассчитать действительные значения коэффициентов `A` и `B` (коэффициентов у нас всего два, потому двух калибровочных точек достаточно). Несколько более универсальным способом было бы вообще исключить из этого дела условную величину `xt` (которая является вспомогательной и пересчет ее в температуру есть внутреннее дело процедуры получения данных) и рассчитывать коэффициенты по фактическому значению переменной `t` (какая она есть) и известным температурам (какая она должна быть). Это несложно, но требует известных интеллектуальных усилий, которые в нашем случае будут лишними, т. к. условная величина `xt` вполне доступна (бывает, что она спрятана внутри датчика), а сама процедура калибровки так и так будет зависеть от конкретного датчика — т. е. от вида самого уравнения. Поэтому мы пойдем наиболее простым и очевидным путем. Поместим на форму кнопку `Button1` (с заголовком **Калибровка**), редактор `Edit1` (изначально деактивированный), и несколько компонентов `Label` для демонстрации значений коэффициентов. В процедуру `Form1.Create` внесем такие строки:

```
Label5.Caption:=FloatToStrF(a,ffExponent,4,2);
Label7.Caption:=FloatToStrF(b,ffGeneral,5,1);
```

После запуска программы окно ее должно иметь вид, показанный на рис. 10.9, *а*. Напротив редактора стоит `Label3`, при запуске первоначально пустой. Не смотрите на то, что окно выглядит не очень эстетично — оно само по себе нам не требуется, вся работа происходит в Tray (рис. 10.9, *б*).



а

б

Рис. 10.9. Окно программы "Температура" (*а*) и ее значки в Tray (*б*)

Теперь напишем саму процедуру калибровки по нажатию кнопки Button1. Флаг FlagC будет сигнализировать о том, на каком этапе калибровки мы находимся.

```
procedure TForm1.Button1Click(Sender: TObject);
var err:integer;
begin  {Калибровка}
if Button1.Caption='Калибровать' then
begin
 Button1.Caption:='Ok';
 Edit1.Enabled:=True;
 Label3.Caption:='Введите значение температуры 1 :';
 FlagC:=1;
 Edit1.Text:=FloatToStrF(t,ffFixed,3,2);
 Edit1.SetFocus;
end else
if FlagC=1 then {первая калибровочная точка}
begin
  try
   if pos(',',Edit1.Text)<>0 then
    {исправляем ошибку с разделительным знаком}
   t1:=StrToFloat(Edit1.Text)
   else val(Edit1.Text,t1,err);
       {реальное значение температуры 1}
   if err<>0 then raise Exception.Create('');;
  except
   ShowMessage('Неправильный формат действительного числа');
   Edit1.SetFocus;
   exit;
  end;
```

```
 xt1:=xt; {данные с датчика для температуры 1}
 Label3.Caption:='Введите значение температуры 2 :';
 FlagC:=2;
 Edit1.Text:=FloatToStrF(t,ffFixed,3,2);
 Edit1.SetFocus;
end else
if FlagC=2 then {вторая калибровочная точка}
begin
  try
   if pos(',',Edit1.Text)<>0 then
    {исправляем ошибку с разделительным знаком}
   t2:=StrToFloat(Edit1.Text)
   else val(Edit1.Text,t2,err);
     {реальное значение температуры 2}
   if err<>0 then raise Exception.Create('');;
  except
   ShowMessage('Неправильный формат действительного числа');
   Edit1.SetFocus;
   exit;
  end;
 xt2:=xt; {данные с датчика для температуры 2}
 Label3.Caption:='';
 FlagC:=0;
 Button1.Caption:='Калибровать';
 Edit1.Enabled:=False;
 {расчет коэффициентов:}
 if t1<>t2 then
 begin
  t1:=t1+273.15; t2:=t2+273.15;
  B:=t1*t2*ln(xt1/xt2)/(t2-t1);
  A:=xt1/exp(B/t1);
  Label5.Caption:=FloatToStrF(a,ffExponent,4,2);
  Label7.Caption:=FloatToStrF(b,ffGeneral,5,1);
  st:='Калибровка прошла удачно';
  ShowMessage(st);
 end else
 begin
  st:='Температуры 1 и 2 не могут быть равны между собой.'
  +#13+'Повторите калибровку.';
  ShowMessage(st);
 end;
end;
end;
```

Здесь мы применили хитрый метод избавления от возможной ошибки при вводе действительного числа с "неправильным" разделительным знаком: процедура `val` есть наследство Turbo Pascal и потому понимает только точку, а процедура `StrToFloat` использует системные настройки. Если обе процедуры не выполнятся (строка вообще некорректна), возникнет то или иное исключение (которое в случае `val` мы вызываем искусственно). Поэтому вводить число можно в любой форме и при любых региональных настройках (кроме как для упомянутых австралийских аборигенов, если они что-то особенное придумали).

Методика выполнения калибровки такая: устанавливаете датчик при определенной температуре рядом с термометром и жмете на кнопку **Калибровка**. При этом она поменяет заголовок на **Ok**, а рядом с оживившимся редактором появится приглашение ввести действительную температуру. Вводите ее значение, и после нажатия на кнопку повторяете процесс при другом значении температуры. Все — коэффициенты рассчитаны. Чем дальше будут отстоять друг от друга значения `t1` и `t2`, тем точнее будет калибровка. Датчик и термометр лучше опускать в перемешиваемую воду (изолировав, конечно, его выводы — можно засунуть в полиэтиленовый пакет), на воздухе разница температур между ними может быть слишком велика. Несомненно, читатель сам доведет программу до ума, организовав запоминание полученных коэффициентов в INI-файле, как мы это делали в предыдущей главе — пусть это будет ему домашнее задание.

В этой главе мы сосредоточились на иконках потому, что данный момент обычно вызывает больше всего вопросов. Ну, а примеры обращения с обычными изображениями мы уже встречали раньше, при создании SlideShow, а также еще вернемся к этим вопросам не раз на протяжении всей книги, поэтому не будем терять времени и перейдем к следующему вопросу — к ресурсным файлам.

ГЛАВА 11

Возобновляемые ресурсы

Как работать с ресурсами исполняемого файла

В 1996 году на базе части Министерства охраны окружающей среды и природных ресурсов Российской Федерации, Роскомгеологии и Роскомвода было создано Министерство природных ресурсов Российской Федерации, которое в 2001 году реорганизовалось еще раз.

www.mnr.gov.ru

Мало кто из начинающих программистов пытается лезть в файл ресурсов приложения (res), который автоматически создается Delphi. И действительно, опыт показывает, что попытка в него вмешаться из самой Delphi в лучшем случае оканчивается ничем. Это ставит "ламера" в тупик, после чего он надолго забывает о самой возможности управлять ресурсами приложения. И, хотя мы уже успешно редактировали этот файл в главах, посвященных приложениям без формы, все же гораздо удобнее и безопаснее свои ресурсы располагать в отдельном файле, тем более что в данной главе речь пойдет не только о тех ресурсах, которые можно создать через Image Editor.

Ресурсы в Windows универсальны, в них можно даже использовать фрагменты выполняемого кода. Общая методика работы с ресурсами в Windows такая: через редактор ресурсов или вручную по определенным правилам создается текстовый файл описания ресурсов (обычно он имеет расширение rc). Затем с помощью одной из утилит компиляции *(см. далее)* этот текстовый файл превращают в бинарный файл ресурсов (res). Этот файл, как вы уже знаете, подключают ("прилинковывают") к исполняемому файлу. Все эти действия одинаковы для любой среды программирования, в том числе и ассемблер. Для главного ресурсного файла программы компилятор Delphi делает это все автоматически, и именно поэтому так трепетно относится к вмешательству в него.

Чтобы было понятно, с чем мы тут имеем дело, я здесь изменю своему правилу не повторять то, что всегда можно посмотреть в справке, и приведу полный список всех видов ресурсов, которые поддерживаются Windows:

```
RT_ACCELERATOR Accelerator table
RT_ANICURSOR Animated cursor
RT_ANIICON Animated icon
RT_BITMAP Bitmap resource
RT_CURSOR Hardware-dependent cursor resource
RT_DIALOG Dialog box
RT_FONT Font resource
RT_FONTDIR Font directory resource
RT_GROUP_CURSOR Hardware-independent cursor resource
RT_GROUP_ICON Hardware-independent icon resource
RT_ICON Hardware-dependent icon resource
RT_MENU Menu resource
RT_MESSAGETABLE Message-table entry
RT_RCDATA Application-defined resource (raw data)
RT_STRING String-table entry
RT_VERSION Version resource
```

У читателей, склонных к тому, чтобы делать все как можно проще, наверняка возник вопрос: а зачем вообще нужен столь сложный механизм с какой-то промежуточной компиляцией, если можно ресурсы просто расположить в отдельных файлах соответствующего формата? Например, строки явно удобнее редактировать, если они расположены в обычном тестовом файле, разве INI-файлы, например, нельзя использовать для той же цели? Конечно, можно. Для того чтобы проиллюстрировать основное назначение ресурсов исполняемого файла, я приведу такой пример. Известно, что HTML-файл не является контейнером (на эту тему см. *главу 16*), поэтому все, что выходит за рамки обычного текста, должно храниться где-то отдельно. Это, с одной стороны, дает большую гибкость в манипуляции элементами оформления, а с другой — попробуйте скачать какую-нибудь красиво оформленную интернет-страницу к себе на диск, и внимательно исследуйте папку, которая при этом образуется (Opera, например, еще и не делает отдельной папки, а все сваливает в кучу в текущей). Иногда там может быть несколько десятков мелких изображений в разных форматах, представляющих элементы оформления страницы, причем эти элементы обычно с небольшими вариациями повторяются для однотипных страниц. Это и есть то, что в EXE-файлах Windows хранится в ресурсах — представляете, что было бы, если бы мы были вынуждены всю эту мелочевку — иконки, меню, курсоры, диалоги — таскать за собой даже пусть не в готовой программе, а на стадии проекта? С одними именами файлов вышла бы такая путаница, что... в общем, понятно.

То есть основное назначение ресурсов — хранение стандартных компонентов. Но никто не мешает нам вводить туда и свои элементы. В пользу этого могут быть несколько соображений. Введение, например, картинок в исполняемый файл уменьшает количество нужных файлов и тем повышает надежность работы программы (соображение, для меня лично, очень существенное). Внедрение номера версии и другой информации о программе позволяет прочесть ее в среде Windows через контекстное меню **Свойства**, не запуская саму программу. Локализацию также удобно производить через ресурсы. Отметим, что далеко не все современные приложения применяют именно ресурсы Windows для таких целей, как локализация. Например, FireFox использует сценарии (скрипты), написанные на JavaScript.

В комплект Delphi входит специальный редактор Resource Workshop, который одинаков для всех сред программирования от Borland, его вполне можно использовать, но мы здесь пойдем другим путем. Мы исключим из круга решаемых задач вопросы редактирования ресурсов в имеющихся DLL- или EXE-файлах — на этот предмет есть по меньшей мере полсотни специализированных программ самой разной степени "фирменности": от наколеночных произведений безымянных умельцев до упомянутого Workshop. Мы ведь создаем свои программы, а не правим чужие, не так ли? Однако иметь одну из таких программ совсем не лишнее — часто хочется сделать иконку или картинку на кнопке по образцу имеющихся, а с помощью таких утилит можно извлечь их из файла чужой программы. Скажу по секрету, что часть иконок в этой книге так и сделана (правда, я никогда не использую чужие иконки "as is", всегда стараюсь придать им оригинальные черты — в результате они, за редкими исключениями, меняются до неузнаваемости). Замечу назло ревнителям копирайта, что другая (большая) часть заимствована из коллекции иконок, приобретенной более десяти лет назад совершенно официально в комплекте с очень примитивным их редактором под Windows 3.0 — пусть никто меня не сможет обвинить в плагиате!

Второе, что мы по мере возможности исключим из рассмотрения — использование ресурсов для локализации приложений. Данный вопрос мы опустим по той причине, что на этот предмет есть множество различного уровня пособий, и сама Delphi включает в себя довольно удобный механизм создания локализованных версий (механизм плагинов, типа используемых в Firefox скриптовых расширений, в принципе много удобнее, но зато приходится все эти скрипты-плагины таскать отдельно). Переписывать все, что уже изложено по этому поводу другими, не имеет смысла — данная книга посвящена вопросам, на которые трудно найти ответы. К их числу относится, например, вопрос — а как включить информацию о версии, разработчике, фирме и т. п. в исполняемый файл, если он не имеет формы, и потому соответствующие

пункты меню **Project** недоступны? Но начнем мы с другого — на простом примере покажем, как работать с ресурсами вообще.

Наглядная агитация

Как можно, например, записать строку в ресурсы простейшим способом и использовать потом ее в качестве константы? Для этого в Delphi есть специальное зарезервированное слово: **resourcestring**. Создадим новый проект, назовем его Resproba (папка Glava11\1), поставим на форму компоненты Label (сразу очистим его заголовок и зададим полужирный шрифт 14 кегля) и Timer, добавим к проекту еще один модуль без формы (**File | New | Unit**) под названием Unitres. Содержание этого модуля будет очень коротким — листинг 11.1.

Листинг 11.1. Модуль Unitres

```
unit Unitres;

interface

resourcestring

  String1 = 'Пролетарии';
  String2 = 'всех стран';
  String3 = 'соединяйтесь!';

implementation

end.
```

Теперь добавим в основном модуле переменную под названием ttime типа byte, которая будет отсчитывать время, добавим после **implementation** строку **uses** Unitres; и напишем следующий обработчик события onTimer:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Label1.Caption:='';
  Label1.Font.Color:=clBlack;
  case (ttime and 3) of
  1: Label1.Caption:=String1;
  2: Label1.Caption:=String2;
  3: begin Label1.Font.Color:=clRed;
  Label1.Caption:=String3; end;
  end;
  ttime:=ttime+1;
end;
```

Запустив приложение, мы увидим посекундно возникающие строки знаменитого коммунистического лозунга, причем последняя строка "соединяйтесь!" будет выделяться красным. Между последовательными демонстрациями будет пауза. Попутно заметим, что здесь проиллюстрирован один из простейших способов организации программного двоичного счетчика — в данном случае до 4. Операция `ttime` **`and`** `3` обнуляет все биты переменной `ttime`, кроме двух младших, а они могут принимать как раз четыре состояния. Так можно легко соорудить счетчик на любое число, кратное степени двойки — а вот для промежуточных чисел придется потрудиться. Я предлагаю читателю самостоятельно поразмыслить над вопросом, как можно организовать счетчик до 3, если мы хотим в данном примере исключить паузу.

Как видим, использование ресурсов в данном варианте довольно бессмысленное занятие: с таким же успехом можно просто задать строковые константы в разделе объявлений основного модуля. Хотя и этот способ может для чего-нибудь пригодиться, но уже другие виды ресурсов так присоединять будет затруднительно. Подумаем, что наглядная агитация гораздо лучше действует, если сопровождается изображениями, и попробуем добавить к нашему лозунгу картинки. В папке с проектом размещено несколько старых агитационных плакатов с сайта plakaty.ru, адаптированных под наши нужды и сохраненных в формате BMP под названиями Image1, Image2 и т. п. Как загрузить эти картинки в ресурсы исполняемого файла?

Для этого сначала создадим файл описания ресурсов — обычный текстовый файл, который будет содержать такие строки:

```
Image1 BITMAP "Image1.bmp"
Image2 BITMAP "Image2.bmp"
Image3 BITMAP "Image3.bmp"
Image4 BITMAP "Image4.bmp"
STRINGTABLE
{
1, "Наглядная агитация"
}
```

Файл назовем Image.rc. В последнем пункте мы ввели в ресурс еще одну строку дополнительно к имеющимся — для иллюстрации, как это делается в обход довольно неудобного механизма **`resourcestring`** в Delphi. Использовать мы ее будем для заставки при запуске. Обратите внимание, что все строковые переменные должны стоять в парных кавычках — так, как это принято в языке C. Признаем, что механизм и здесь не очень совершенен — в полях структуры `STRINGTABLE` нельзя использовать идентификаторы строк, а только их номера (несколько усложнив пример, можно присвоить идентификаторы но-

мерам, но не самим строкам!). Заметим еще две существенные вещи: во-первых, в качестве операторных скобок в структуре STRINGTABLE можно использовать и фигурные скобки в стиле C (как в этом примере), и конструкцию **begin**...**end** в стиле Pascal, что обусловлено универсальным характером утилиты компиляции, о которой сейчас пойдет речь (она едина для всех сред программирования Borland). Во-вторых, в строку можно вводить произвольные символы, предваряя их номер знаком "обратный слеш". Поэтому если нужно включить сам "обратный слеш" (дисковые имена файлов), то он должен повторяться дважды.

Теперь этот файл надо скомпилировать в ресурс с помощью утилиты brcc32 (Borland Resource Command line Compiler). Для удобства создадим раз и навсегда BAT-файл для компиляции ресурсов под названием rescomp.bat со следующим содержимым:

```
"C:\Program Files\Borland\Delphi7\Bin\brcc32.exe" %1
```

Кавычки нужны потому, что мы используем длинные имена папок. Оба файла поместим в папку с проектом (туда же, где находятся файлы с изображениями) и выполним из командной строки следующую команду:

```
rescomp.bat Image.rc
```

В результате компиляции получим файл Image.res, который будет включать в себя все четыре картинки и строку. Осталось присоединить его к нашему проекту, загрузить картинки и строку из ресурсов и выводить их в нужное время. Для этого объявим переменную BmpImg:TbitMap, добавим после **implementation** строку {$R Image.res} и напишем два обработчика для создания BitMap при открытии и закрытии приложения, а также для загрузки заставки:

```
procedure TForm1.FormCreate(Sender: TObject);
  var Buffer: array[0..255] of Char;
begin
  BmpImg:=TBitMap.Create;
  BmpImg.LoadFromResourceName(hInstance,'Image1');
{загрузка картинки заставки}
  LoadString(hInstance,1,Buffer,255); {загрузка строки номер 1}
  Label1.Caption:=StrPas(Buffer); {вывод заголовка заставки}
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  BmpImg.Destroy;
end;
```

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Draw(80,100,BmpImg); {вывод картинки заставки}
end;
```

Как видите, если мы не используем механизм Delphi, а формируем строку в ресурсах сами, процедура ее загрузки усложняется, причем следует использовать в качестве промежуточного буфера именно байтовый массив, а не тип Pchar — иначе могут возникать различные сложности *(см. главы 14 и 21)*.

Обратите также внимание на то, что загружаем начальную картинку мы по событию onCreate, а прорисовываем — по событию onPaint. Если мы будем пытаться прорисовать BitMap при создании формы, то он не покажется. Можно и все делать по событию onPaint, но тогда у нас каждый раз при фокусировке формы будет выполняться загрузка изображения и заставочной строки. Теперь осталось переписать нашу процедуру по таймеру (чтобы картинки можно было бы лучше рассмотреть, увеличьте время срабатывания таймера до 3 сек):

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Label1.Caption:='';
  Label1.Font.Color:=clBlack;
  case (ttime and 3) of
  1: begin
      Label1.Caption:=String1;
      Canvas.FillRect(Canvas.ClipRect); {очистка формы}
      BmpImg.LoadFromResourceName(hInstance,'Image2');
      Form1.Canvas.Draw(80,100,BmpImg);
     end;
  2: begin
      Label1.Caption:=String2;
      Canvas.FillRect(Canvas.ClipRect); {очистка формы}
      BmpImg.LoadFromResourceName(hInstance,'Image3');
      Form1.Canvas.Draw(80,100,BmpImg);
     end;
  3: begin
       Label1.Font.Color:=clRed;
       Label1.Caption:=String3;
       Canvas.FillRect(Canvas.ClipRect); {очистка формы}
       BmpImg.LoadFromResourceName(hInstance,'Image4');
       Form1.Canvas.Draw(80,100,BmpImg);
     end;
  end;
```

```
  ttime:=ttime+1;
end;
```

Если убрать у формы заголовок (`BorderStyle = bsNone`), компонент `Label1` расположить по центру, и установить его свойство `Alignment` в `taCenter` (чтобы оно сработало, надо также свойство `AutoSize` установить в `False`), у нас получится хотя и не совсем доведенная до ума, но вполне приличная художественная инсталляция в стиле соц-арт. В [3] вы можете прочесть, как в Windows XP сделать саму форму при этом прозрачной, что значительно увеличит эффект (см. об этом также *главу 12*). Причем неопытный пользователь даже и не сообразит сразу, как избавиться от окна, которое не имеет ни одной кнопки.

Обратим, кстати, внимание на объем исполняемого файла программы — за счет того, что мы теперь не обязаны таскать все картинки отдельно, файл получился почти 2 Мбайта. Но если вы оставите картинки в отдельных файлах, или загрузите их в DLL, как чаще поступают, то в общем объеме вы ничего не выиграете, а вероятность что-то потерять увеличится. Другое дело, что в полной мере все преимущества пользования ресурсами, как легко заменяемыми модулями приложения, проявляются, если загружать их не в исполняемый файл, а именно в отдельную DLL, тогда можно легко заменять картинки и строки, не перекомпилируя саму программу. Именно так и поступают для многоязычных приложений — использование ресурсных файлов практически единственный удобный способ обеспечить беспроблемное размножение одной программы на разных языках, хотя и в самом исполняемом файле, как мы увидим позже, этих языков может быть задано много. Но мы не будем на всем этом специально останавливаться — читатель, без сомнения, теперь и сам может справиться с подобными задачами.

Заставка и номер версии в SlideShow

Используем полученные сведения, чтобы избавиться от таскаемой с собой картинки для заставки в нашем SlideShow, и заодно доделаем диалог пользовательских установок, как мы обещали в *главе 9*, а кроме того, покажем, как из ресурсов извлечь номер версии и другую информацию. Перенесем проект из *главы 9* (Glava9\3) в новую папку (Glava11\2) и присвоим ему номер версии 1.31.

Сначала создадим текстовый файл zastavka.rc с единственной строкой:

```
Zastavka BITMAP "zastavka.bmp"
```

Скомпилируем его с помощью нашего BAT-файла и получим ресурсный файл zastavka.res. Теперь в текст модуля zastavka добавим строку `{$R`

zastavka.res}, и изменим в единственной имеющейся у нас там процедуре прорисовки функцию загрузки картинки из файла на такую:

```
Form2.BmpImage.LoadFromResourceName(hInstance,'Zastavka');
```

Обратим внимание, что после компиляции размер исполняемого файла, как и ожидалось, стал в два раза больше, зато теперь BMP-файл не нужен. Как видите, все просто, осталось только реализовать обещанные установки пользователя — демонстрировать заставку или нет, а также сворачивать ли приложение в иконку при минимизации и при закрытии. Для этого сделаем следующее — введем в панель установок три компонента CheckBox с соответствующими заголовками и установленным в True свойством Checked (рис. 11.1).

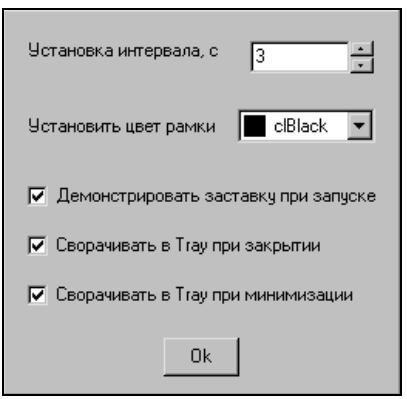


Рис. 11.1. Панель пользовательских установок SlideShow

В тексте основной программы (SlideShow.dpr) добавим в предложении **uses** модули SysUtils и IniFiles, и в секцию **var** переменную IniFile:TIniFile;, а конец его (после определения повторного запуска) изменим так:

```
IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini'));
{если не было - создаем, иначе открываем}
with IniFile do
begin
     if SectionExists('Sets') then
                    {если файл и секция в нем уже есть}
     begin
        if ReadBool('Sets','Zastavka',True) then
        begin
          Form2:=TForm2.Create(Application);
  {создаем форму-заставку}
          Form2.Show; {показываем заставку}
          while Form2.Timer1.Enabled do Application.ProcessMessages;
```

```
  {пустой цикл, пока таймер активен}
        Form2.Free; {уничтожаем заставку}
      end;
    end else {если секция еще не создана}
    begin
      WriteBool('Sets','Zastavka',True);
      WriteBool('Sets','CloseWithTray',True);
      WriteBool('Sets','MinimizeWithTray',True);
      Form2:=TForm2.Create(Application); {создаем форму-заставку}
      Form2.Show; {показываем заставку}
      while Form2.Timer1.Enabled do Application.ProcessMessages;
  {пустой цикл, пока таймер активен}
      Form2.Free; {уничтожаем заставку}
    end;
    Destroy;
end;
  {создаем главную форму}
Application.CreateForm(TForm1, Form1);
Application.CreateForm(TForm3, Form3);
Application.Run; {запускаем приложение}
```

Секцию установок в INI-файле мы создали, теперь осталось реализовать обращение к ней в основном модуле. Допишем в процедуру по событию onCreate формы Form1 внутри директивы **with** IniFile **do**... перед процедурой destroy такие строки:

```
if SectionExists('Sets') then {если файл и секция в нем уже есть}
begin
   CheckBox1.Checked:=ReadBool('Sets','Zastavka',True);
   CheckBox2.Checked:=ReadBool('Sets','CloseWithTray',True);
   mayClose:= not CheckBox2.Checked; {в Tray при закрытии}
   CheckBox3.Checked:=ReadBool('Sets','MinimizeWithTray',True);
   {в Tray при минимизации}
end;
```

А в процедуру Button2Click добавляем обратные действия (также перед строкой IniFile.Destroy):

```
. . . . .
IniFile.WriteBool('Sets','Zastavka',CheckBox1.Checked); {демонстрация
заставки}
IniFile.WriteBool('Sets','CloseWithTray',CheckBox2.Checked);
mayClose:= not CheckBox2.Checked; {в Tray при закрытии}
IniFile.WriteBool('Sets','MinimizeWithTray',CheckBox3.Checked);
{в Tray при минимизации}
. . . . .
```

Для того чтобы реализовать отмену и разрешение сворачивания в иконку при минимизации независимо от закрытия (сейчас у нас при минимизации вызывается метод `Close` формы), нам придется переписать процедуру минимизации, с тем, чтобы в ней также создавать иконку и скрывать форму:

```
procedure TForm1.OnMinimizeProc(Sender: TObject);
begin
  If CheckBox3.Checked then
  begin
     FHandle := AllocateHWnd(WndProc); {получаем дескриптор окна}
     HIcon1:=CopyIcon(Application.Icon.Handle);
               {получаем дескриптор иконки}
     with noIconData do begin
          cbSize:=Sizeof(TNotifyIconData); {размер структуры}
          Wnd:=FHandle; {дескриптор окна}
          uID:=0; {единственная иконка}
          UFlags:=NIF_MESSAGE or NIF_ICON or NIF_TIP;

             {взводим все флаги}
          SzTip:='SlideShow'; {всплывающая подсказка}
          HIcon:=HIcon1; {дескриптор иконки}
          uCallBackMessage:=Ico_Message;
       {определяемое пользователем сообщение}
     end;
     Shell_NotifyIcon(NIM_ADD,@noIconData); {создали иконку}
     Form1.Hide; {скрыли окно}
  end;
       end;
```

Кроме этого, придется изменить установки флага `mayClose`. В процедурах `WndProc` и `PopupRestore1Click` вместо `mayClose:=False;` следует записать ту же самую строку:

```
mayClose:= not CheckBox2.Checked;
```

А в процедуру `PopupClose1Click` надо добавить вначале строку `mayClose:=True;`. Теперь у нас процедуры закрытия и минимизации будут работать независимо друг от друга. Отметим, что флажок **Демонстрировать заставку** сейчас модно располагать на самой заставке, и я думаю, что вынести его туда читатель сможет при желании самостоятельно.

Осталось показать, как можно прочесть из ресурсов номер версии и вывести его в заголовок формы автоматически. Надо сказать, что если в Windows API и есть более запутанные вещи, то их немного. Чтобы прочитать все поля записи типа `RT_VERSION`, следует обратиться к структуре `VS_VERSION_INFO`. Но

так просто это сделать не получится, надо сначала выявить список языков, а уж затем выцарапывать номер версии в текстовом виде для самого первого в списке языка. Дело, видимо, в том, что в соответствующем поле структуры содержится полный номер версии (включая номера реализации (Release) и сборки (Build), которые мы не заполняем, см. закладку **Version Info** в **Project | Options**), и справедливо предполагается, что они могут быть разными для различных языков. Отсюда и вся "навороченность" — несомненно те, кто все это придумал, были типичными представителями высшей касты "настоящих программистов".

Так что комментировать эту совершенно неоправданно усложненную процедуру я не буду — если кому-то надо разобраться подробнее, милости просим в Интернет. Дополним начало процедуры `TForm1.FormCreate` следующим текстом:

```
procedure TForm1.FormCreate(Sender: TObject);
var x:integer;
  versize: integer;
  verh:dword;
  verbuffer,versionNum,P: PChar;
  FLangCharSet,stver:string;
begin
{===определение номера версии и загрузка в заголовок===}
   versize:=GetFileVersionInfoSize(PChar(Application.ExeName),verh);
   verbuffer := StrAlloc(versize+1);
   try
     GetFileVersionInfo(PChar(Application.ExeName),
                                  0, versize, verbuffer);
     VerQueryValue(verbuffer,'\VarFileInfo\Translation',
                                      pointer(P),verh);
      x:=0;
      StrLCopy(@x, P, 2);
      FLangCharSet:=IntToHex(x, 4);
      StrLCopy(@x,P+2,2);
      FLangCharSet:=FLangCharSet+IntToHex(x, 4);
{строка '041904E3'=10491251 - язык + кодовая страница}
    VerQueryValue(verbuffer, pchar('\StringFileInfo\'+FLangCharSet+
    '\'+'FileVersion'), pointer(versionNum), verh);
    {получили полный номер версии}
    stver:=copy(versionNum,1,4); {получаем сокращенный номер версии}
    Form1.Caption:='SlideShow '+stver; {выводим в заголовок}
    finally
     StrDispose(verbuffer);
   end;
```

```
{=====}
. . . . .
end;
```

Номер версии в приложении без формы

Самостоятельно задать в ресурсном файле номер версии и другую информацию несколько проще, чем ее оттуда извлечь. Вот шаблон заготовки ресурсного файла для нашего перекодировщика текстов Layout из *главы 8*:

```
1 VERSIONINFO
FILEVERSION 1,0,0,0
PRODUCTVERSION 1,0,0,0
FILEFLAGSMASK VS_FFI_FILEFLAGSMASK
FILEOS VOS__WINDOWS32
FILETYPE VFT_APP

{
  BLOCK "StringFileInfo"
  {
    BLOCK "041904E3"
    {
      VALUE "CompanyName", ""
      VALUE "FileDescription", "Layout switcher Mainfile\000"
      VALUE "FileVersion", "1.0.0.0"
      VALUE "InternalName", ""
      VALUE "LegalCopyright", "@Revich Y.V.\000"
      VALUE "LegalTrademarks", ""
      VALUE "OriginalFilename", "Layout.exe\000"
      VALUE "ProductName", "Layout\000"
      VALUE "ProductVersion", "1.0.0.0"
      VALUE "Comments", "No Comments"
    }
  }

  BLOCK "VarFileInfo"
  {
    VALUE "Translation", 1049, 1251
  }
}
```

Строки требуется заканчивать символами нуля, иначе они будут отображаться неадекватно — я же говорил, что придумали все это "настоящие програм-

мисты". Скомпилировав полученный ресурс под названием info.res, мы можем внести строку {$R info.res} в текст программы Layout из папки (Glava8\4) и заново скомпилировать ее (папка Glava11\3).

Произвольные ресурсы

Если вы поковыряетесь в файлах, созданных с помощью Delphi, с помощью одной из упомянутых ранее программ для доступа к ресурсам, то увидите, что все интерфейсные элементы задаются именно через ресурсы. С помощью определяемого пользователем ресурса типа RT_RCDATA можно загрузить в ресурсы приложения практически любой объект. Вот какова может быть последовательность действий при размещении в ресурсах, например, нашей таблицы перекодировок из проекта Layout *главы 5*. Сначала мы создаем вспомогательную программу из одного модуля для записи таблицы в файл:

```
program MakeFile;
. . . . .
const CharEngRus: array [32..126] of byte =
($20,$21,$DD,$B9,$3B,$25,$3F,$FD,$28,$29,$2A,{*}
 $2B,$E1,$2D,$FE,$2E,$30,$31,$32,$33,$34,$35, {5}
 $36,$37,$38,$39,$C6,$E6,$C1,$3D,$DE,$2C,$22,{@}
 $D4,$C8,$D1,$C2,$D3,$C0,$CF,$D0,$D8,$CE,$CB,{K}
 $C4,$DC,$D2,$D9,$C7,$C9,$CA,$DB,$C5,$C3,$CC, {V}
 $D6,$D7,$CD,$DF,$F5,$5C,$FA,$3A,$5F,$B8,$F4,{a}
 $E8,$F1,$E2,$F3,$E0,$EF,$F0,$F8,$EE,$EB,$E4,{l}
 $FC,$F2,$F9,$E7,$E9,$EA,$FB,$E5,$E3,$EC,$F6,{w}
 $F7,$ED,$FF,$D5,$2F,$DA,$A8 );
var
. . . . .
  fCode: file of array [32..126] of byte;
begin
  assign(fCode, 'CharEngRus.dat');
  rewrite(fCode);
  write(fCode, CharEngRus);
  close(fCode);
end.
```

Обратите внимание на то, что для файла мы используем паскалевские процедуры assign и close — т. к. здесь нет никаких форм и других объектов, то нет и боязни, что мы случайно вызовем соответствующий метод какого-нибудь объекта. Однако "во избежание" делать этого в Delphi-программах не рекомендуется — так, и в программе-шпионе из *главы 6*, и в переключателе

клавиатуры из *главы 7* мы применяли привычные `assignfile` и `closefile`, хотя там это тоже было безразлично.

Получив таким образом дисковый файл с таблицей, мы загружаем его в ресурсы. В нашем приложении создаем файл CharEngRus.rc со строкой

```
CharEngRus RCDATA "CharEngRus.dat"
```

Затем компилируем его с помощью rescomp.bat, подключаем полученный RES-файл к проекту и загружаем ресурс в секции `initialization`:

```
type
  TCharEngRus=array [32..126] of byte;
var
  PCharEngRus: ^TCharEngRus;
  CharEngRus: TCharEngRus;
implementation
{$R CharEngRus.res}
. . . . .
initialization
  PCharEngRus:=LockResource(LoadResource(hInstance,
   (FindResource(hInstance, 'CharEngRus', RT_RCDATA))));
  if PCharEngRus = nil then <неправильный ресурс>
  else CharEngRus:=PCharEngRus^;
end.
```

Метод загрузки для произвольного ресурса, как видите, несколько сложнее, чем в случае BitMap или строки. Переменная `PCharEngRus` будет указывать на начало ресурса в исполняемом файле. Чтобы не мучиться потом с указателями, мы соорудили в программе переменную подходящего типа `CharEngRus`, которая сразу при запуске инициализируется значением массива из ресурса. Переменную можно заменить и на типизируемую константу, т. к. я уже говорил, что в Delphi различие между типизируемыми константами и переменными совершенно гомеопатическое.

ГЛАВА 12

Бабушка в окошке

Нестандартные окна

> Сложный ремонт оконных переплетов может производить только тот, кто хорошо овладел основами столярного дела, умеет выстругать бруски заданного размера и профиля, хорошо подгонять их и склеивать.
>
> *С. Иванчиков, "Учись делать сам"*

Прежде чем приступать к увлекательному занятию по созданию окон нетрадиционной формы, как следует подумайте над вопросом: а оно вам надо? Одной из причин (но, конечно, не единственной), по которой я никогда не пользуюсь программами из пакета Norton Utilities for Windows, было совершенно отталкивающее впечатление, произведенное на меня главным окном этой программы. На кого рассчитана эта картинка в стиле иллюстраций к Толкиену? Напомним, что речь идет не о Media Player, а о чисто вспомогательных программах, которые просто обязаны быть как можно незаметнее. Не рискуя сильно ошибиться, могу предположить, что разработчики из Symantec серьезно больны весьма распространенной болезнью — показать всему миру, какие они крутые программисты. Не дай бог вам, читатель, заболеть этим недугом — вылечиться бывает весьма трудно. Конечно, на вкус и цвет, как говорится... но на всякий случай я обхожу как можно дальше продукцию этой фирмы, представив себе, что они могут наворотить, например, в Norton Antivirus, когда программа скрыта с глаз пользователя.

Но в других случаях украшательства в программе вполне уместны — и мы с вами именно для этого и использовали заставку при запуске SlideShow. На ее примере мы далее покажем, как можно делать окна нестандартной формы и с нестандартным поведением, а пока остановимся на том, как это *вообще* делается.

Для того чтобы менять форму окна, в Windows есть специально предусмотренный механизм создания регионов (regions). Регионы могут быть квадрат-

ные (CreateRectRgn), с закругленными краями (CreateRoundRectRgn), эллиптические (CreateEllipticRgn), произвольной формы (CreatePolygonRgn) и даже составленными из набора произвольных полигонов либо стандартных форм (CreatePolyPolygonRgn, CombineRgn). Как вы догадались, в скобках для каждого случая приведены функции, с помощью которых такой регион создается. Позже вы увидите практические примеры использования некоторых из них, а пока заметим, что параметры функций, естественно, для разных случаев разные, но общим для них остается указание дескриптора объекта, форму которого предполагается изменить. Вот общий порядок использования функций на примере создания круглого окна диаметром 400 пикселов:

```
var fReg:hRGN;
. . . . .
fReg:=CreateEllipticRgn(0,0,400,400);
SetWindowRgn(Form1.handle,fReg,True);
```

Если вставить эту процедуру в обработчик события, например, Form1.Create, то при создании окно главной формы сразу станет круглым. Причем в качестве первого параметра необязательно использовать дескриптор именно формы, это может быть любой компонент, и даже канва формы (Form1.Canvas.Handle) или другого компонента. Если вы зададите для канвы заставки в проекте SlideShow такую форму, то получите круглую картинку в прямоугольном окне. Только имейте в виду, что если объект, для которого задается конфигурация, изначально меньше по размерам, чем заданная область, то или ничего не произойдет вообще, или отрежется только часть компонента. Тут мы пришли к крупнейшему недостатку данного механизма: реально все эти функции управляют только отображением заданной области, деформировать регион нельзя. Нельзя, например, вставить картинку в непрямоугольный компонент так, чтобы она должным образом изменила свою форму. Нельзя создать круглую кнопку только с использованием этих функций — ее сначала придется отдельно нарисовать. Заголовок полукруглого окна также придется рисовать самостоятельно. Мало того, нельзя даже автоматически задать вывод текста в фигурный компонент так, чтобы текст не обрезался, придется придумывать собственные сложные процедуры. Если поразмыслить, то все это отчасти оправданно тем, что деформация любого изображения требует использования специальных алгоритмов и довольно значительного количества вычислительных ресурсов, но все же это ограничение намного снижает потенциал метода.

И все же указанные функции позволяют получить многие полезные эффекты. Для ориентировки читателя перечислим некоторые вспомогательные функции: FrameRgn обводит регион рамкой указанной кистью заданной ширины, FillRgn заполняет его указанной кистью в стиле, который может быть задан

функцией SetPolyFillMode, и т. п. Приводить примеры мы не будем, они достаточно просты, и их при необходимости легко найти в Сети. Единственное, на что стоит обратить внимание: у начинающих обычно ничего не выходит с созданием полигонального региона (CreatePolygonRgn), т. к. в нем надо задавать многоугольник через параметр типа **array of** TPoint. Правильно координаты вершин будет задать, например, по такому образцу:

```
const
Points : array[0..3] of TPoint = ((X:80;Y:0), (X:0;Y:80), (X:80;Y:160),
(X:160;Y:80));
```

И сразу ответим на известный вопрос, как перемещать по экрану форму, которая не имеет заголовка, или у которой заголовок сформирован искусственно. Для этого надо перехватывать сообщение WM_NCHITTEST. В простейшем виде реализация этого способа выглядит так:

```
. . . . .
private
    procedure Wdrop (var msg: TMessage); message WM_NCHITTEST;
. . . . .
procedure TForm1.Wdrop(var msg: TMessage);
begin
 Inherited;
 msg.Result := HTCAPTION;
end;
```

Данная процедура подменяет результат сообщения так, что форму можно "подцепить" за любое место, и она будет считать, что это заголовок. Несколько более усложненный вариант позволяет выделить на форме область, которая будет считаться заголовком:

```
. . . . .
Inherited;
with msg do
{переводим экранные координаты мыши в оконные}
with ScreenToClient(Point(XPos,YPos)) do
if PtInRegion(rTitle, X, Y) then
Result := HTCAPTION;
. . . . .
```

Здесь rTitle — регион, определяющий область заголовка, а функция PtInRegion определяет, принадлежит ли точка (X, Y) этому региону. Можно тем же путем даже обеспечить имитацию системных кнопок, если при нажатии на нужные области возвращать через Result значения HTMAXBUTTON, HTMINBUTTON, HTCLOSE или HTSYSMENU. Далее мы покажем, как на практике пе-

рехватывать сообщения типа WM_NC*xxxx*, т. е. сообщения, которые относятся, согласно терминологии разработчиков Windows, к событиям мыши *"while the cursor is within the nonclient area of a window"* (когда курсор находится в неклиентской области окна) — это цитата из официальной справки.

А сейчас мы перестанем заниматься, наконец, теорией и немножко разукрасим нашу заставку в проекте SlideShow.

Красивая заставка в SlideShow

Перенесем проект SlideShow из папки Glava11\2 в новую папку (Glava12\1 — естественно, лишние теперь файлы ресурсов, картинку заставки и т. п. переносить не требуется) и изменим номер версии на 1.32. Добавим следующие переменные в секции **var** модуля zastavka.pas:

```
fReg:hRGN;
tsek:integer = 0;
fWidth: integer=500;
fHeight: integer=400;
```

Далее создадим обработчик onCreate формы, и допишем в него следующие процедуры:

```
procedure TForm2.FormCreate(Sender: TObject);
begin
{задаем с начальными размерами}
    fReg:=CreateRoundRectRgn(0,0,fWidth,fHeight,100,100);
{регион с закругленными углами}
    SetWindowRgn(Handle,fReg,True);
    SetWindowPos(Handle,HWND_TOP,(Screen.Width div 2)-250,(Screen.Height
div 2)-200,500,400,SWP_SHOWWINDOW);
end;
```

Последний оператор нужен вот зачем — свойство формы Position на будущее лучше изменить на poDesigned, т. к. иначе будет некрасиво, а чтобы изначально форма возникала в центре, пришлось установить ее там принудительно.

Теперь мы изменим интервал компонента Timer1 на 1000 (он у нас был установлен в 5000), а затем обработчик по событию onTimer перепишем следующим образом:

```
procedure TForm2.Timer1Timer(Sender: TObject);
var fTop,fLeft: integer;
```

```
begin
   inc(tsek);
   if (tsek>1) and (tsek<5) then
   begin
     fWidth:=fWidth+50*(tsek-1)*3; fHeight:=fHeight+40*(tsek-1)*3;
     fReg:=CreateRoundRectRgn(0,0,fWidth,fHeight,100,100);
{регион с закругленными углами}
     SetWindowRgn(handle,fReg,True);
     fLeft:=(Screen.Width div 2)-(fWidth div 2);
     fTop:=(Screen.Height div 2)-(fHeight div 2);
     SetWindowPos(Handle,HWND_TOP,fLeft,fTop,fWidth,fHeight,
                  SWP_SHOWWINDOW);
       aRect:=Rect(0,0,fWidth,fHeight); {размеры картинки}
       Form2.Canvas.StretchDraw(aRect,Form2.BmpImage);
{загружаем картинку в канву формы}
   end;
   if tsek=5 then Timer1.Enabled:=false;
end;
```

Если вы запустите программу (значение параметра Zastavka в INI-файле, естественно, должно быть установлено в `True` — должен быть установлен флажок на панели установок), то у вас возникнет заставка с полукруглыми краями, которая через две секунды начнет расширяться на весь экран, пока не исчезнет. Самое увлекательное в этом деле — поиграться с формулами для `fWidth` и `fHeight`, а также для `fLeft` и `fTop`, одновременно меняя интервал таймера и предельное число "тиков", которое у нас тут равно 5. Можно заставить заставку уменьшаться вместо увеличения, дрейфовать ее к краю или к углу экрана и т. п. Можно ввести полигональную фигуру и заставить заставку деформироваться. Но когда вы наиграетесь окончательно, то поймете, что самое главное, что тут выглядело бы по-настоящему уместно, сделать таким образом не получится: а именно заставить окно менять размеры *плавно*. Не выйдет даже на тех системах, которые все еще не у всякого рядового пользователя имеются, типа Pentium Extreme Edition — даже у них не хватит быстродействия, чтобы заставить окно заставки по-настоящему естественно исчезать с экрана. Но все же, если все тщательно проработать, то можно получить красивые эффекты типа осыпания заставки или ее улетания кувырком в угол. Я оставляю это на усмотрение читателей, а здесь мы на будущее оставим только эффект скругления углов окна, а всю эту анимацию отменим, оставив процедуру `Timer1.Timer` в том состоянии, в котором она была раньше. На диске в папке Glava12\1 я пока оставил все, как здесь написано, но в следующий раз анимированный вариант будет отменен.

Прозрачная форма и окно flystyle

Функция CombineRgn очень удобна, ее механизм применения напоминает способы копирования канвы методом CopyRect. И там и там задаются параметры комбинации накладываемого изображения (региона) с фоном, только CombineRgn обладает более широкими возможностями, т. к. позволяет складывать различными способами участки произвольной формы и, самое главное, позволяет вычитать накладываемый регион из фона, получая прозрачные области. В Windows XP, как мы уже не раз упоминали, есть механизм обеспечения прозрачности участков формы. Причем даже два механизма: один есть чистая (как в Photoshop) реализация альфа-канала с переменной прозрачностью (свойства AlphaBlend и AlphaBlendValue), а другой аналогичен заданию "прозрачного" цвета для BitMap (TransparentColor и TransparentColorValue). Но они, к сожалению, не работают в Windows 9*x*. А CombineRgn предоставляет универсальный механизм для такого рода операций, причем не только для равномерно закрашенных областей. Однако задавать прозрачный регион вам придется вручную, что при сложной его форме может стать достаточно громоздким занятием. Пример такой операции можно найти в [5] — идея автора состоит в том, что мы просматриваем последовательно строка за строкой (или столбец за столбцом) весь массив пикселов, и последовательно присоединяем к первоначальному региону одномерные области — фрагменты строки, цвет которых отличается от заданного "прозрачным". Мы же продемонстрируем некоторые возможности этого механизма на примере создания окна flystyle — т. е. окна, которое при запуске имеет только заголовок, а полностью раскрывается при щелчке на нем, подобно пунктам меню. Похожим образом реализованы, например, управляющие панели инструментов в графическом редакторе Paint Shop Pro версии 7.0 и выше.

Создадим пробный проект под названием Region (папка Glava12\2) и установим для формы следующие свойства: BorderStyle в bsDialog, Color в clBlack, Position в poDesktopCenter, HorzScrollBar.Visible и VertScrollBar.Visible в False. Чтобы не возиться с загрузкой в канву, установим на форму компонент Image и сразу загрузим в него через свойство Picture картинку, в качестве которой я использовал портрет своего любимого шотландского вислоухого кота по кличке Бакс. В заголовке формы я так и написал на чистом русском языке Kot, чтобы ни у кого не возникало сомнений, что перед вами не кошка. Картинка имеет ширину 433 пиксела и под нее подогнаны все остальные размеры (рис. 12.1).

Добавим следующие переменные в тексте модуля формы и напишем обработчик события onCreate:

```
var
  Form1: TForm1;
```

```
  fReg,cReg:hRGN;
  ClientX,ClientY,i: Integer;
  fView:boolean=True;
. . . . .
procedure TForm1.FormCreate(Sender: TObject);
begin
  ClientX:=0;
  ClientY:=Height-ClientHeight;
  fReg:=CreateRectRgn(0,0,Width,Height); {вся форма}
  cReg:=CreateRectRgn(ClientX,ClientY,ClientX+Width,
ClientY+ClientHeight);
  {клиентская часть формы}
  CombineRgn(fReg,fReg,cReg,rgn_Diff); {вычитаем}
  SetWindowRgn(handle,fReg,True);{демонстрируем}
end;
```

Рис. 12.1. Форма с картинкой на этапе конструирования

Если мы запустим приложение, то получим голый заголовок формы — вся клиентская часть станет прозрачной, т. е. для пользователя будет отсутствовать. Я специально задал здесь координаты клиентской части окна в общем виде, для того чтобы этот пример можно было бы использовать как прототип для других вариантов. Например, можно задать ее координаты так, чтобы прозрачной стала только внутренняя часть окна (собственно клиентская область), а рамка осталась видимой:

```
ClientX:= (Width - ClientWidth) div 2;
ClientY:= Height-ClientHeight-ClientX;
```

Теперь нужно создать обработчик события щелчка по заголовку, в котором мы будем показывать и убирать обратно картинку — т. е. реализуем режим flystyle. Для того чтобы определять текущее состояние — видна картинка или не видна — я ввел переменную-флаг fView, по умолчанию имеющую значение True. Объявим в секции **private** процедуру-ловушку под названием CaptionClick, на этот раз для сообщения WM_NCLBUTTONDOWN. Вопрос о том, какое именно сообщение использовать, на самом деле тут довольно принципиален — так, если мы будем раскрывать/закрывать окно по простому щелчку, как я предлагаю, то при попытке просто подвигать заголовок, форма все равно раскроется, что может раздражать. Если мы хотим, чтобы форма раскрывалась не по щелчку, а, например, просто при наведении курсора, то можно попробовать использовать сообщение WM_NCMOUSEMOVE — правда, при этом придется крепко подумать над логикой работы процедуры. При использовании двойного щелчка — пользователь может просто не понять, что от него ждут именно этого. Наилучшим вариантом является логика, аналогичная той, которую используют в Windows при работе с иконками на рабочем столе — если просто щелкнули, то при отпускании происходит раскрытие, а если хоть чуть-чуть подвигали, то больше ничего не будет. Но здесь мы не будем усложнять — просто продемонстрируем, как это делается в принципе. Итак, дописываем объявление процедуры и ее обработчик:

```
. . . . .
private
    procedure CaptionClick(var msg: TMessage);
message WM_NCLBUTTONDOWN;
. . . . .
procedure TForm1.CaptionClick(var msg: TMessage);
begin
 Inherited;
 if msg.WParam=HTCAPTION then
 begin
   if fView = True then
   begin
      fView:=False;
      fReg:=CreateRectRgn(0,0,Width,Height); {вся форма}
      cReg:=CreateRectRgn(ClientX,ClientY,ClientX+Width,
                          ClientY+ClientHeight);
      {клиентская часть формы}
      CombineRgn(fReg,fReg,cReg,rgn_Diff); {вычитаем}
      for i:= 0 to ControlCount-1 do {собираем "контролы"}
```

```
      with Controls[I] do begin
      cReg:= CreateRectRgn(ClientX+Left, ClientY+Top,
              ClientX+Left+Width,ClientY+Top+Height);
      CombineRgn(fReg,fReg,cReg,rgn_Or);
      end;
    end else
    begin
      fView:=True;
      fReg:=CreateRectRgn(0,0,Width,Height); {вся форма}
      cReg:=CreateRectRgn(ClientX,ClientY,ClientX+Width,
                          ClientY+ClientHeight);
      {клиентская часть формы}
      CombineRgn(fReg,fReg,cReg,rgn_Diff); {вычитаем}
    end;
  end;
  SetWindowRgn(handle,fReg,True);{демонстрируем}
end;
```

Рис. 12.2. Раскрытое окно с картинкой после щелчка на заголовке

Что мы тут делаем? При первом щелчке мы сначала заново создаем тот же самый регион, состоящий из одного видимого заголовка, но добавляем к не-

му все визуальные объекты-"контролы", расположенные на форме, — в этом случае фон окна останется прозрачным, а все имеющиеся компоненты станут видимыми. На самом деле у нас здесь всего один такой объект — `Image1`, но я опять же написал в общем виде для удобства усовершенствования программы. В результате мы при щелчке на заголовке получим вот такую картинку (рис. 12.2) — я ее специально воспроизвел на фоне текста этого примера. А при втором щелчке окно опять свернется в заголовок.

Все эти механизмы открывают необычайные перспективы для создания нестандартных программ: от полноправных Windows-приложений, остающихся тем не менее невидимыми, или псевдоиконок на рабочем столе, до серьезных приложений с полупрозрачными дочерними окнами и управляющими элементами, экономящими экранную площадь. Главное — не позволить себе слишком увлечься.

ГЛАВА 13

Приставание с намеком

Прокрутка колесиком, режим Drag&Drop, работа с ProgressBar и другие мелочи

Drag — тянуть, волочить
Drop — капать, опускать

Англо-русский словарь

Тянуть — приставать, надоедать
Капать — намекать, доносить

В. С. Елистратов,
"Словарь московского арго"

Сказать, что поддержка колесика мыши в Delphi реализована недостаточно корректно — значит выразиться слишком мягко. Далеко не все потомки `TWinControl`, для которых определены события типа `onMouseWheel`, реально обрабатывают соответствующие сообщения Windows. Возможно, это зависит от типа мыши (т. е. от драйвера производителя), но факт, что в некоторых случаях прокрутка колесиком не работает или работа ее зависит от версии ОС — у меня, например, для `RichEdit` прокрутка работает в Windows XP, но не работает в Windows 98, а для формы в целом, как и для самого универсального компонента — `ScrollBox` — не работает ни там ни там.

Кстати, и в самой Windows функции колесика реализованы достаточно бездумно. Одна из самых ненавидимых мной функций — привязка к колесику прокрутки списка в компонентах типа `ListBox`, из-за чего не раз случалось наворотить делов, пока удавалось разобраться, в чем дело. Особенно поидиотски это выглядит в браузерах, когда, случайно провернув колесико, ты оказываешься вдруг перед фактом, что-то вдруг начинает загружаться... загружаться... но ты же вообще никуда не указывал! Но гораздо чаще прокрутка действительно нужна, линейки прокрутки при отсутствии колесика — вещь более чем неудобная, и мы об этом уже говорили в *главе 1*. Так как же обеспечить бесперебойную работу этого механизма?

Прокрутка в компоненте *ScrollBox*

Для отслеживания поворота колесика проще всего перехватить сообщение WM_MOUSEWHEEL в оконной процедуре всего приложения (свойство OnMessage объекта Application) — в конкретную форму оно может и не поступить. У этого сообщения параметр wParam несет сразу тройную нагрузку: в знаке старшего слова указывается направление прокрутки (вниз — минус, вверх — плюс), в самом значении этого слова — количество условных "тиков", на которое колесико прокрутили, а в младшем слове — состояние одновременно нажатых клавиш-модификаторов. Нас интересует только знак и клавиши, с "тиками" нам разбираться незачем.

Создадим новый проект ScrollDrop (папка Glava13\1) с красивой иконкой (pciii.ico), на форму поставим компонент ScrollBox и растянем его на всю форму, оставив внизу небольшое пустое пространство. В это пространство поместим компонент Label, немного левее центра — в него мы будем выводить подпись. У компонента ScrollBox1 свойство AutoScroll должно быть равно True, а AutoSize — False. В левый верхний угол ScrollBox1 поставим компонент Image, у которого AutoSize, наоборот, должно быть установлено в True.

Добавим в секцию **private** процедуру перехвата сообщений:

```
private
procedure WndProc(var Msg: TMsg; var Handled:boolean);
. . . . .
```

и напишем следующий обработчик события onCreate формы:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
if paramcount<>0 then {если есть параметр загрузки}
try
 Image1.Picture.LoadFromFile(paramstr(1));
{пробуем загрузить картинку}
 Form1.Label1.Caption:=paramstr(1);
except
 Image1.Picture.LoadFromFile('pole.jpg');
{не получается - по умолчанию}
 Form1.Label1.Caption:='pole.jpg';
end else
begin
 Image1.Picture.LoadFromFile('pole.jpg');
{загружаем по умолчанию}
 Form1.Label1.Caption:='pole.jpg';
end;
```

```
Application.OnMessage:=WndProc; {перехват оконной процедуры}
DragAcceptFiles(Form1.Handle,True);
{сообщаем, что готовы к приему файлов}
end;
```

Здесь мы сразу предусмотрели вот что: если вы запускаете программу из командной строки с параметром в виде имени файла с картинкой, то она загрузится в `Image1`, в противном случае (или если параметр — не картинка) загрузится картинка по умолчанию pole.jpg, которую я расположил в папке с проектом. Картинку по умолчанию я специально выбрал достаточно большой (фотография в стиле "родные просторы" 1136×852), чтобы при загрузке появились обе линейки прокрутки в компоненте `ScrollBox1`. В компоненте `Label1` будет отображаться имя файла с картинкой (рис. 13.1). Добавим, что запуск программы с параметром в Windows осуществляется не только при ручном запуске из командной строки, но и при "бросании" файла с картинкой на файл программы в Проводнике — т. е. мы сразу осуществили режим Drag&Drop для не запущенной программы. Но это бы не сработало, если бы мы не уведомили Windows, что готовы к такому повороту событий. Это делается вызовом функции `DragAcceptFiles` в последней строке процедуры. В предложение **uses** следует добавить два модуля: ShellApi и JPEG.

Рис. 13.1. Окно примера ScrollDrop с загруженной по умолчанию картинкой

Осталось оформить собственно процедуру WndProc. Она будет выглядеть так:

```
procedure TForm1.WndProc(var Msg: TMsg; var Handled:boolean);
begin
if Msg.message=WM_MOUSEWHEEL then
begin
 if (Msg.wParam and MK_CONTROL) = 0 then
 begin
   if Msg.wParam>0 then {крутим вниз}
   ScrollBox1.VertScrollBar.Position:=
        ScrollBox1.VertScrollBar.Position+8
   else {крутим вверх}
   ScrollBox1.VertScrollBar.Position:=
        ScrollBox1.VertScrollBar.Position-8;
 end
 else {нажата Ctrl}
 begin
   if Msg.wParam>0 then {крутим вправо}
   ScrollBox1.HorzScrollBar.Position:=
        ScrollBox1.HorzScrollBar.Position+8
   else {крутим влево}
   ScrollBox1.HorzScrollBar.Position:=
        ScrollBox1.HorzScrollBar.Position-8;
 end;
end;
end;
```

То есть при просто прокрутке колесика у нас будет изображение передвигаться в вертикальном направлении, а при прокрутке с нажатой клавишей <Ctrl> — в горизонтальном.

Полный Drag&Drop

Отлично, но как бы еще принять файл, "брошенный" на уже открытую программу? Надо сказать, что в Delphi этот режим не реализован вообще никак — тот механизм, что имеется, обеспечивает довольно удобный обмен через перетаскивание объектов между компонентами одной формы, но мне лично такое еще не понадобилось ни разу. Разумеется, это совершенно не лишняя функциональность, но перетаскивание извне куда интереснее. Для того чтобы осуществить прием "брошенного" файла, нам придется создать еще один перехватчик сообщения, на этот раз WM_DROPFILES. Его можно перехватить в той же процедуре WndProc, но удобнее соорудить отдельный специализированный перехватчик (для формы, а не для приложения целиком). Добавим в ту же секцию **private** следующую строку:

```
procedure DropPicture(var Message: TWMDROPFILES);
     message WM_DROPFILES;
```

В секцию **var** нужно добавить переменную `buffer : array[0..255] of char`, а сама процедура будет выглядеть так:

```
procedure TForm1.DropPicture(var Message: TWMDROPFILES);
begin {прием файлов, брошенных на форму}
DragQueryFile(Message.Drop,0,@buffer,sizeof(buffer));
try
 Image1.Picture.LoadFromFile(buffer); {пробуем загрузить}
 Form1.Label1.Caption:=buffer;
except
end;
end;
```

Здесь процедура `DragQueryFile` возвращает нам в переменной `buffer` строку с именем файла. Если файл — картинка, он загрузится в `Image1`, а его имя в `Label1`.

Программа для поиска файлов

Программу для поиска файлов (пока по названию, а в дальнейшем и по содержанию) мы сделаем, как и обещали, на основе программы Kodirovka из *главы 8* (папка Glava8\3). Для этого ее придется сильно модернизировать, и мы не будем, как обычно, просто копировать проект, а переименуем и его, и единственный его модуль штатным способом — через пункты **File | Save Project as** (для проекта) и **File | Save as** (для модуля). Назовем проект Trace (что по смыслу не совсем точно, но звучит красиво и позволяет отличить программу от многочисленных Search), а модуль — poisk (папка Glava13\2). Отметим, что мы на первых порах будем действовать "по старинке" — читать все из дискового файла, хотя это и категорически неправильно (см. замечание по этому поводу в *главе 8*). Я это делаю сознательно: мы доделаем программу по максимуму, чтобы вы убедились, насколько это "тормозная" штука — а потом исправим положение, использовав для чтения один из механизмов быстрого чтения файлов (применявшийся в *главе 7* "маппинг"). Задумка моя состоит в том, чтобы на этом примере подвигнуть вас на изучение и использование механизмов виртуальной памяти — опыт показывает, что изучавшие Turbo Pascal сопротивляются этому до последнего. Результат сравнения алгоритмов будет достаточно впечатляющ, увидите.

Прежде всего, заменим картинку в `Image1` на другую аналогичную с новым названием программы Trace (файл trace.bmp в папке на диске) и заголовок формы на **Поиск файлов**. Первое капитальное изменение — заменим

RichEdit на компонент WebBrowser (расположен последним пунктом в палитре **Internet**).

После замены придется внести изменения в текст модуля poisk. Прежде всего, удалим все, относящееся в RichEdit (если вы что-то забудете, то не сомневайтесь, на ошибку вам укажут). Затем добавим переменные: ftempname:**string**, nall:integer и ftemphtm:TextFile. В начало процедуры по нажатию кнопки Button2 (**Искать**) вставим следующие строки:

```
. . . . .
ftempname:=ExtractFilePath(Application.ExeName)+'trace000.htm';
assignfile(ftemphtm,ftempname);
rewrite(ftemphtm); {создаем заново временный файл}
closefile(ftemphtm);
. . . . .
```

Для удаления файла в конце программы создадим обработчик onDestroy формы:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
if FileExists(ftempname) then
erase(ftemphtm); {уничтожаем временный файл}
end;
```

Далее создаем такую процедуру, расположив ее текст выше обработчика по кнопке **Искать**:

```
procedure createHTMLtext(st1,st:string);
begin
if FileExists(ftempname) then
append(ftemphtm) else exit; {вдруг его удалили?}
st1:='<B>'+st1+'</B><I>'+st+'</I><BR>';
write(ftemphtm,st1);
closefile(ftemphtm);
end;
```

То есть мы просто-напросто создаем текстовый файл с HTML-тегами, которые форматируют нам текст, как надо[1]. Осталось его загрузить в WebBrowser. Для этого вместо всех процедур красивого вывода в RichEdit нужно добавить только две строки:

[1] Предполагается, что читатель имеет представление о простейших приемах форматирования текста в HTML. Если это не так, то см. *главу 16*, где эти сведения излагаются в минимально необходимом объеме.

```
createHTMLtext(st1,st);
WebBrowser1.Navigate(Pchar(ftempname));
```

Самодеятельную процедуру удаления возможных пробелов в начале строки stpath заменим на официальную функцию Trim, которая сразу удаляет их с обеих концов. Для того чтобы не расписывать и дальше все мелкие исправления, я приведу полный текст новой процедуры по нажатию кнопки Button2 (**Искать**):

```
procedure TForm1.Button2Click(Sender: TObject);
{ищем файлы и обрабатываем их}
begin
 ftempname:=ExtractFilePath(Application.ExeName)+'trace000.htm';
 assignfile(ftemphtm,ftempname);
 rewrite(ftemphtm); {создаем заново временный файл}
 closefile(ftemphtm);
FlagCancel:=0; {флаг отмены}
Button3.Enabled:=True; {активируем Отмену}
Button2.Enabled:=False; {дезактивируем Поиск}
stpath:=Edit1.Text; {название папки}
if stpath='' then exit; {если оно пустое - сразу на выход}
stpath:=Trim(stpath); {удаляем возможные пробелы по концам}
if stpath='' then exit; {если пусто сразу на выход}
if stpath[length(stpath)]='\' then
delete(stpath,length(stpath),1);
{удаляем концевой знак '\', если он есть}
try
 ChDir(stpath); {проверяем, есть ли такая папка}
except
  Application.MessageBox('Несуществующая директория',
            'Error',MB_OK);
  exit; {если нету - на выход}
end;
Form1.Label1.Caption:=''; {очистили заголовок Label}
Application.ProcessMessages; {чтобы сразу сработало}
stsearch:='\*';
stsearch:=stpath+stsearch; {строка для поиска файлов}
nfile:=0;
if FindFirst(stsearch,$23,sf)=0 then
{ $23 = не просматриваем системные файлы, тома и каталоги}
repeat
  if FlagCancel<>0 then break;
      {если флаг отмены не 0, то прерываем}
```

```
  fname:=stpath+'\'+sf.Name; {полное имя файла}
  if ANSIUpperCase(fname)=ANSIUpperCase(ftempname) then continue;
                {временный пропускаем}
  nall:=nall+1; {новый объект}
  {все к одному регистру:}
  if pos(ExtractFileExt(ANSIUpperCase(fname)),
         ANSIUpperCase(stExt))<>0 then continue;
  {если расширение совпадает с запрещенным, то на выход}
  if not ReadFileFormat then continue;{читаем файл}
  nfile:=nfile+1; {что-то получили}
  Form1.Label1.Caption:='Просмотрено: '+IntToStr(nall)
            +' Найдено: '+IntToStr(nfile);
  st:=' кодировка:'+st;
  st1:=IntToStr(nfile)+'. '+fname; {номер найденного файла}
  createHTMLtext(st1,st);
  WebBrowser1.Navigate(Pchar(ftempname));
  Application.ProcessMessages; {чтобы все прокрутилось}
until FindNext(sf)<>0; {пока файлы не закончатся}
Form1.Label1.Caption:='Просмотрено: '+IntToStr(nall)+
            ' Найдено: '+IntToStr(nfile);
st1:=Form1.Label1.Caption;
st:='<BR><A NAME="1">Поиск закончен</A>';
createHTMLtext(st1,st);
st:=ftempname+'#1';
WebBrowser1.Navigate(Pchar(st));
  Form1.Edit1.SetFocus; {возвращаем фокус в Edit1}
  Edit1.SelStart:=length(Edit1.Text);
  Edit1.SelText:=''; {курсор в конец текста Edit1}
FindClose(sf); {конец поиска}
Button2.Enabled:=True; {активируем Поиск}
Button3.Enabled:=False; {дезактивируем Отмену}
end; {Button2}
```

Здесь переменная `nall` нам считает все просмотренные объекты. Обратите внимание, что в процедуре поиска мы вставили строку **`if`** `fname=ftempname` **`then`** `continue`, для того чтобы в процессе поиска избежать чтения, нашего временного файла. Прокрутку колесиком `WebBrowser`, по счастью, надежно поддерживает, и это есть великое благо, потому что добраться до его "скролл-баров" невозможно — в перечне свойств объекта они попросту отсутствуют, и позже вы поймете, почему. Но нам хочется остановить вывод по окончании поиска в конце текста, а не в начале — для наглядности. Для этого мы не стали ковыряться в недрах объекта `WebBrowser`, а попросту пометили последнюю строку обычной HTML-меткой *(см. главу 16)* и заставили браузер

перейти на нее в конце вывода. Если несколько усложнить процедуру, вставив метку с самого начала, а затем перенося ее каждый раз в последнюю строку, можно даже добиться того, что `WebBrowser` будет прокручивать текст по мере вывода, но это будет достаточно сильно тормозить вывод.

Заметки на полях

Как вы поняли, компонент `WebBrowser` есть не что иное, как движок Internet Explorer — тот самый, интегрированный в Windows, из-за которого Microsoft чуть не засудили. Представляет собой он трансляцию соответствующих ActiveX-компонентов *(см. главу 18)*. Причем он в Delphi довольно удобно реализован и позволяет при необходимости осуществлять далеко не только отображение HTML-файлов, для которого мы тут его используем. Главные его недостатки — медленная скорость работы и некоторые "глюки". Вообще в этой книге вопросы программирования для Интернета я стараюсь тщательно обходить. Причина, думаю, понятна — это особая тема, которая требует иного подхода и заслуживает отдельной книги. Но отображение удобного и компактного формата HTML требуется очень часто, и для `WebBrowser` мы сделали исключение. Можно ли найти для `WebBrowser` замену попроще и, главное, пошустрее? В принципе просмотрщиков HTML имеется сколько угодно, однако подавляющее большинство из них использует тот же самый Internet Explorer и польза от них сомнительная. Мне пришлось пользоваться одним из редких независимых от Internet Explorer компонентов под названием `THtmlViewer` (**http://www.torry.net/vcl/internet/html/htm345d.zip**). Впечатления вполне благожелательные, но он, во-первых, достаточно громоздкий (дистрибутив почти 3 Мбайта), во-вторых, платный, в-третьих, использует коммерческую библиотеку просмотра графики ilda32.dll, которую лучше таскать с собой (ее может не оказаться в системе, особенно в ранних версиях Windows). По всем этим причинам пришлось от него отказаться.

Попробуем теперь ввести стандартный в таких случаях ползунок, показывающий ход процесса "перелопачивания" материала, и заодно обсудим вопрос — как рассчитать время до конца операции? О том, насколько это непросто — корректно отображать процесс, длительность которого априорно неизвестна, — могут свидетельствовать многочисленные "ляпы", допускаемые в таких случаях даже самыми продвинутыми программами. Нередко дело доходит до анекдота — один из подобных случаев представлен на рис. 13.2.

Рис. 13.2. Пример ошибки расчета времени при копировании файла

Анекдотических ситуаций мы постараемся избежать, но предупреждаю, что удовлетворительного результата все равно добиться очень сложно.

Продлим форму немного вниз и слева поставим на нее стандартный для таких случаев ProgressBar, а чуть правее — Label2, в который будем выводить время до конца операции (если у вас нет проекта перед глазами, то загляните на рис. 13.3, чтобы понять, что к чему). Для того чтобы правильно настроить ползунок, надо знать заранее объем предстоящей работы, и ничего не остается, кроме как перед началом поиска посчитать все файлы в заданной папке. Свойство Step ползунка установим в 1, и включим в проект такую процедуру:

```
procedure Calculatefiles;
begin
Form1.ProgressBar1.Max:=0;
if FindFirst(stsearch,$23,sf)=0 then
{ $23 = не просматриваем системные файлы, тома и каталоги}
repeat
 Form1.ProgressBar1.Max:=Form1.ProgressBar1.Max+1;
until FindNext(sf)<>0; {пока файлы не закончатся}
FindClose(sf); {конец поиска}
end;
```

Не очень умно устраивать такую долгую процедуру только для того, чтобы сделать красивый вывод, но выхода нет — кстати, в дальнейшем мы ее изменим, и она уже будет служить не только для красоты.

Раз уж мы имеем заранее полное число файлов, то нелишне его и продемонстрировать. Изменим в процедуре поиска вывод в Label1 на следующий оператор:

```
Form1.Label1.Caption:=' Всего файлов '
+IntToStr(Form1.ProgressBar1.Max)+
' Просмотрено: '+IntToStr(nall)+' Найдено: '+IntToStr(nfile);
```

Теперь вставим вызов Calculatefiles перед самым началом цикла поиска, а процедуру инкремента ProgressBar — в середину:

```
. . . . .
Calculatefiles; {подсчитали файлы}
ProgressBar1.Position:=0; {установили в 0}
if FindFirst(stsearch,$23,sf)=0 then
. . . . .
nall:=nall+1; {новый объект}
Form1.ProgressBar1.StepIt; {инкремент ProgressBar}
. . . . .
```

Обратите внимание на строку:

```
ProgressBar1.Position:=0;
```

Наиболее часто случающиеся "проколы" с "синей колбасой"[2] появляются от того, что этой установкой пренебрегают — ползунок запоминает предыдущее состояние, в результате чего доходит до "упора" слишком рано и повторно начинает с начала, и потом наоборот, не доходит до конца. Ведь в конце предыдущего обращения к нему он находится вовсе не в нулевом состоянии — близком, но необязательно равном максимальному, и ошибка эта накапливается в дальнейшем. Впрочем, заставить ползунок работать строго, как положено — задача "не для белых людей". В этой программе, если вы обратили внимание, несмотря на все усилия, он также доходит до конца вовсе не всегда. Второй крупный недостаток этого компонента — то, что наличие `ProgressBar` довольно сильно тормозит процесс, позже вы в этом сами убедитесь.

Теперь попробуем справиться со временем. Введем две переменные типа `TDateTime`: `tt` и `ttold`. Перед вызовом `Calculatefiles` добавим такой оператор: `ttold:=Time;`. Далее мы будем каждый раз определять, сколько секунд прошло с этого момента, сопоставлять это время с количеством обработанных файлов и общим их количеством, и рассчитывать ориентировочное время окончания. Эта процедура будет выглядеть так:

```
procedure CalculateTime;
var ts:integer;
begin
 tt:=Time; {текущее время}
 tsek:=SecondsBetween(tt,ttold);
 with Form1.ProgressBar1 do
 begin
  ts:=round((tsek*Max)/(Position));{расчет макс. времени}
  tsek:=ts-round((tsek*Position/Max)); {сколько сек осталось}
 end;
//  tt:=tt-IncSecond(tt,tsek);
end;
```

Разумеется, необходимо объявить переменную `tsek` и добавить в **uses** модуль DateUtils. Переменная `tsek` у нас будет иметь тип `integer`, хотя на самом деле ей положен тип `int64`, имеющий умопомрачительный диапазон почти до 10^{19} степени по абсолютной величине. Трудно представить себе такое количество секунд, но диапазону типа `integer` в секундах соответствует всего 68 лет и в общем случае это, разумеется, оправданно. Имейте в виду, что все функции

[2] **http://zhurnal.lib.ru/b/brigadir_j_a/test.shtml**.

типа *xxx*Between из модуля DateUtils возвращают корректное положительное значение независимо от порядка параметров при вызове процедуры. О закомментированной строке — чуть позже. Вставляем вызов этой процедуры и вывод результатов в середину поиска:

```
. . . . .
nfile:=nfile+1; {что-то получили}
CalculateTime; {расчет времени}
if tsek>0 then
Form1.Label2.Caption:='Осталось '+IntToStr(tsek)+' сек';
. . . . .
```

В начале процедуры, одновременно с Label1, заголовок Label2 надо очищать:

```
. . . . .
Form1.Label1.Caption:=''; {очистили заголовок Label1}
Form1.Label2.Caption:=''; {очистили заголовок Label2}
. . . . .
```

Если мы запустим теперь программу, и зададим поиск, то получим нечто, подобное рис. 13.3, где поиск ведется в той же папке, что в *главе 8*, только похудевшей с тех пор на один файл.

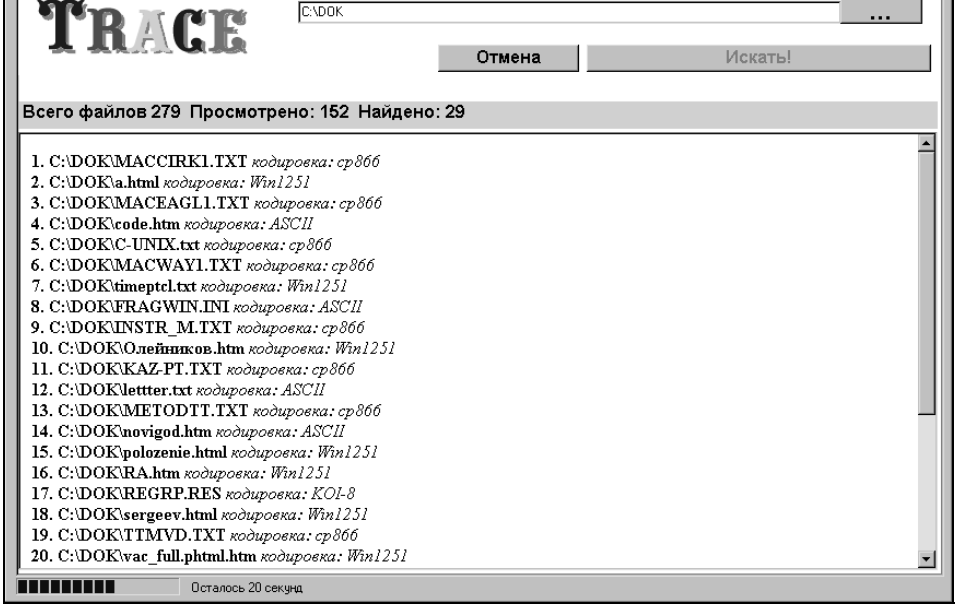


Рис. 13.3. Работа программы Trace

Если вы исследуете работу программы для разных по объему папок, то увидите, что результаты расчета времени начинают приближаться к похожим на правду значениям только к концу процесса. Это, безусловно, следствие того, что файлы все разного размера и наш расчет очень приблизителен. Я привел здесь эту процедуру только для примера, и в дальнейшем мы ее не будем использовать ввиду ее сомнительной информативности — слишком легко тут попасть в ситуацию, отраженную на рис. 13.2 Автору этих строк еще ни разу не приходилось встречать программы, которые абсолютно верно отображали бы время до конца процесса априорно неизвестной длительности. Единственный, пожалуй, относительно корректный пример дают сетевые загрузчики файлов, но весь фокус в том, что как раз там длительность априорно известна — т. к. более-менее точно известна скорость перекачки. Если же она меняется непредсказуемо, то и эти программы попадают впросак. Обратите, например, внимание на то, что любой такой загрузчик при демонстрации скорости перекачки (обычно в кбайт/сек) сначала всегда показывает невероятно высокую скорость — загрузка начинается на всякий случай заранее, пока пользователь еще мучительно размышляет на вечную тему "а надо ли?", и отсюда подобные "ляпы". Показательно, что никто даже и не пытается исправить эту ситуацию, видимо, зная по опыту, что будет только хуже.

Альтернативную возможность представления результатов расчета времени дает закомментированная строка: если вы ее раскомментируете и замените в операторе вывода в `Label2` выражение `IntToStr(tsek)` на `TimeToStr(tt)`, то получите вывод в виде ЧЧ:ММ:СС. Следует отметить, что работа с форматами времени и даты требует опыта, терпения и представляет собой довольно сложный процесс: так, вывод через `TimeToStr` у меня дает результаты без ведущего нуля, но почему-то только в значении часов — кому вообще мог прийти в голову формат времени без ведущего нуля в разрядах? Для получения заведомо определенного представления даты/времени во всех таких случаях следует использовать функцию `FormatDateTime` (или `DateTimeToString`) — не нужно пытаться, например, просто дополнить строку нулем. Это следует делать по той причине, что работа функции `TimeToStr` зависит от системных установок и вы можете попасть впросак, если запустите вашу программу, к примеру, под английской версией. Другой нюанс, связанный со временем — здесь мы выдаем только разницу во времени в часах-минутах-секундах, а что будет, если поиск придется на полночь? В общем, есть над чем поломать голову.

На данном этапе это еще не законченная программа, конечно, потому она пока ничего не ищет, просто перебирает все текстовые файлы, по ходу дела определяя кодировку. Разумеется, такая операция не имеет большого практического смысла, и в следующей главе мы займемся усовершенствованием программы и превращением ее в программу поиска по тексту.

О работе с индикаторами длительности процесса

Несколько слов о работе с ползунками, которые отображают время длительного процесса. Вместо стандартного `ProgressBar` можно взять более продвинутый `Gauge` (закладка **Samples**), но в этой книге мы этого делать не будем, т. к. он тормозит процесс уже до совершенно неприемлемых величин. Для того чтобы проиллюстрировать обращение с разными типами ползунков и протестировать их возможности, я расположил на диске в папке Glava13\Polzunok демонстрационную программу Polzunok. В ней примерно 100-килобайтный файл (наша картинка из SlideShow 0klen2.jpg) читается побайтно в строку (о том, как это делается, см. *главу 21*). До этого и после этого системным таймером фиксируется время и выводится длительность процесса в секундах. Рассматриваем четыре случая: вообще без графического ползунка (с выводом процентов выполнения в текстовой форме в компонент `Label`), со стандартными `ProgressBar` и `Gauge`, а также с самостоятельно сконструированным ползунком на основе компонентов `Image` и `Panel`. Ручаюсь, что вы будете несколько ошеломлены результатами: на моем относительно медленном компьютере длительность процедуры без графического сопровождения составила 2 секунды, с компонентом `ProgressBar` — 55 секунд, а с компонентом `Gauge` — целых 130 секунд, в 65 раз медленнее! При этом наш самодельный ползунок дает фору "официальным" компонентам — с ним длительность составила 46 секунд. Однако не сомневаюсь, что если на его основе сделать собственный компонент, то "классовая" обертка замедлит процесс, и никакого выигрыша мы по сравнению с `ProgressBar` уже не получим, поэтому возиться, видимо, не стоит. Конечно, на компьютерах с другими видеосистемами, более быстрым процессором и доступом к памяти результаты будут другими, но соотношение вряд ли изменится принципиально. В DOS я бы не задумался над тем, чтобы написать свой собственный ползунок с прямым выводом в видеопамять (см. на эту тему замечание в *главе 8*), но в Windows это настолько громоздкое занятие, притом с непредсказуемым результатом в отношении выигрыша во времени, что в критичных случаях проще обойтись вообще без ползунка.

Чтобы сократить впустую потраченное время, можно увеличить шаг `Step` и вызывать процедуру `StepIt` не каждый раз, а, к примеру, каждый десятый. Но это может привести ко всяким дерганиям, если файлов мало, и к усугублению неприятного эффекта, когда по окончании процесса полоска не доходит до конца. Так что решайте сами — для того чтобы вызывать `StepIt`, например, каждый десятый раз, в нашей программе Trace достаточно поставить условие `if (nfile>0) and (nfile mod 10 = 0)`.

ГЛАВА 14

Читать умеете?
Доработка программы Trace

> Дешифровка эламской части надписи хотя и была связана с преодолением известных трудностей, все же, казалось, обещала более быстрый успех, нежели дешифровка более запутанной вавилонской.
>
> *Эрнст Добльхофер, "Знаки и чудеса"*

Для того чтобы превратить нашу программу Trace из прошлой главы действительно в поисковую, т. е. выполняющую задачу просмотра текстовых файлов и выделения тех из них, в которых встречаются заданные слова и выражения, надо решить как минимум три основные задачи.

- Выбрать собственно алгоритм поиска и решить проблему чтения различных форматов.
- Обеспечить поиск во вложенных папках. Тот, кто пытался найти что-то с помощью встроенного поиска Word (пункт **Файл | Открыть**), понимает, что поиск *только* в текущей папке — в общем, никому не нужная вещь. Для решения этой задачи нам придется научиться составлять дерево каталогов.
- Обеспечить чтение файлов не с диска, а из памяти, пользуясь одним из способов, предлагаемых операционной системой. (О том, что `ProgressBar` тормозит процесс, постараемся забыть — в процессе собственно чтения файла он нам не мешается.) Ее решение, по соображениям наглядности (см. комментарий на эту тему в *главе 13*), мы оставим почти на самый конец.

Сначала поподробнее о первой задаче. В общем случае нам очень хотелось бы читать все возможные форматы, или, по крайней мере, главные — кроме просто текстовых (туда же относится и HTML), которые мы уже умеем читать в трех основных кодировках, это форматы DOC и RTF. Причем если читать их напрямую доморощенными способами (примерно так, как мы это делали в *главе 8* для Unicode-текста), то дело осложняется тем, что и DOC и

RTF могут, вообще говоря, содержать и Unicode, и обычный plain text (в принципе и текст HTML в версии 4 тоже может содержать формат Unicode, но, слава богу, это встречается достаточно редко). В принципе изобретать ничего не требуется — достаточно воспользоваться услугами технологии COM (она же OLE) и просто переводить такие тексты в plain text методами Office, не вникая в их суть. То же относится и к XML, PDF или базам почтовых программ (напомним, что такие форматы вообще умеют читать только продвинутые поисковики типа Google Desktop Search). Кое-какие способы это сделать — со всеми их недостатками — мы укажем в *главе 18*, а пока ограничимся только текстовыми форматами.

Насчет третьей задачи напомню, что с самого начала мы пошли в части чтения не самым лучшим способом — читаем из дискового файла байт за байтом (ну ладно, слово за словом), как будто живем в эпоху DOS с ее 640 Кбайтами памяти *(см. главу 8)*. Но сначала мы продолжим эту линию, и сделаем паллиативный, но зато простой и наглядный вариант специально "для чайников": после того как мы определили кодировку, мы прочтем (пока побайтно) содержимое файла в строку, а потом будем определять, имеется ли в ней вхождение искомой строки.

Для того чтобы оправдать свои труды и не повторять чужие достижения, а придать программе оригинальную функциональность, мы введем в нее возможность поиска любого из нескольких слов — функцию, которой очень не хватает в большинстве подобных программ. Когда в *главе 18* мы узнаем про методы преобразования Word-форматов в текст, то доработанная с учетом этого программа может вполне претендовать на свое место среди подобных, как простая альтернатива навороченным поисковикам от того же Google. Отметим и крупнейший общий недостаток такой программы в сравнении с "фирменными" — поиск у нас будет осуществляться каждый раз заново. Оправданием тут служит то, что и встроенный поиск Windows, и утилиты сторонних производителей (например, встроенные в клоны Norton Commander) также не имеют средств для ускорения поиска. Центральным узлом "настоящей" поисковой системы, типа Яндекса или того же Google, является механизм *предварительной* индексации файлов и составления соответствующей базы данных, что намного ускоряет процедуру. Такой механизм всегда есть "ноу-хау" разработчиков, и можно только представить, как пришлось поломать голову создателям Google Desktop Search, чтобы размер этой базы оказался приемлемым. А изюминкой поисковой машины Яндекс, например, является поиск по словоформам — чего нет в многоязычной Google. Но в основном разработчики локальных систем действуют по нашему принципу — так, дабы максимально ускорить поиск в Windows XP, в ней вообще решили "не замечать" файлы с "немикрософтовскими" расширениями (скажем, текстовый поиск по слову "uses" не покажет вам ни одного PAS-файла), что

обессмысливает использование этой функции до полной ее непригодности, даже в сравнении с убогим аналогичным сервисом в Windows 9*x*.

Итак, приступим к решению сначала второй задачи — как составить список всех вложенных папок?

Составление списка вложенных папок

Перенесем, как всегда, проект из папки Glava 13\2 в новую папку (Glava 14\1). Список вложенных папок мы будем хранить в переменной типа `TStringList` — назовем ее `CatDir` и добавим в список глобальных переменных. Так как мы все равно будем перебирать все папки, то нам ничего не стоит заодно и пересчитывать файлы в них[1]. Расчет оставшегося времени мы, как и говорили, отменим, но количество файлов нам все равно понадобится. Для удобства мы введем специальную переменную `ncount` типа `integer` — обращаться каждый раз к `Form1.ProgressBar1.Max` — больно громоздкое занятие. Затем создадим такую процедуру:

```
procedure GetTreeDirs(Root: string; OutCat: TStringList);
{составление списка всех вложенных папок}
var
  i: integer;
  stx: string;
  procedure InsDirs(stc: string; ind: Integer; Path: string; OutList:
                    TStringList);
  var {Создает список вложенных папок}
    sr: TSearchRec;
  begin
    if DirectoryExists(Path) then
      if FindFirst(Path+'\*',faAnyFile,sr)=0 then
      begin
        repeat
          if ((sr.Attr and faDirectory)<>0)
                    and (sr.Name[Length(sr.Name)]<>'.')
```

[1] Кстати, слово file в своем изначальном значении переводится как "досье", "дело", "подшивка", а folder — как "папка", "скоросшиватель", т. е. значения у них весьма близки. Слово file среди компьютерщиков сначала не имело столь откровенно канцелярского оттенка и означало почти в точности то, что и имеется в виду в общеупотребительном смысле — контейнер для неких данных. А folders в приложении к компьютеру есть вообще придумка маркетологов — общепризнанные в среде программистов "директории" или "каталоги" намного лучше подходят по смыслу, это отражено и в названиях функций. Я здесь употребляю слово "папка" только для единообразия — иначе несчастный пользователь Windows вообще рискует запутаться.

```
          then {если папка, но не вышележащая}
          begin
           if ((sr.Attr and faHidden)=0)
           then OutList.Insert(ind,stc+sr.Name)
           {и не скрытая, то включаем}
          end else if (sr.Name[Length(sr.Name)]<>'.') then
          ncount:=ncount+1; {иначе дополняем количество файлов}
        until (FindNext(sr)<>0);
        FindClose(sr);
      end
  end;  {конец InsDirs}
begin
  ncount:=0;
  if not DirectoryExists(Root) then exit;
  Form1.ProgressBar1.Max:=0;
  {Создаем список каталогов первой вложенности}
  InsDirs(Root+'\', OutCat.Count, Root, OutCat);
  i:=0;
  if OutCat.Count<>0 then {теперь для каждой вложенной папки}
  repeat
    stx := OutCat[i]; {в stx получаем путь к уже внесенной в список}
    InsDirs(stx+'\',i+1,OutCat[i],OutCat);
          {вставляем найденную сразу за данной директорией в списке}
    inc(i);
  until (i=OutCat.Count);
  OutCat.Insert(0,Root); {вносим корневую папку}
  Form1.ProgressBar1.Max:=ncount;
end; {конец GetTreeDirs}
```

Остановимся на условиях, при которых мы здесь относим найденные имена к папкам и файлам. Требование, чтобы найденное имя файла или папки (`sr.Name`) не содержало в конце символа точки, означает, что мы исключаем из рассмотрения вышележащие папки (в MS-совместимых файловых системах это "." и ".."). Сложнее понять, откуда взялось условие `(sr.Attr and faDirectory)<>0` — почему бы не поставить просто `sr.Attr=faDirectory`? Если мы сделаем по второму варианту, то мы упустим, например, папку Program Files — она имеет еще и атрибут "только для чтения", т. е. ее атрибут равен числу $11 (`faDirectory + faReadOnly`), а не $10 (просто `faDirectory`). А вот скрытые папки ("корзину" и т. п.) нам желательно опустить, поэтому условие усложняется дальнейшими "наворотами". Зато скрытые файлы в общее количество войдут — хотя здесь это и не принципиально, если очень захочется, мы можем отсеять их уже при просмотре.

Чтобы обеспечить просмотр файлов во всех папках, которые содержатся в списке, придется заключить наш цикл просмотра в текущей папке внутрь еще одного цикла, для чего в процедуре Button2Click создадим локальную переменную i:integer. Не забудем также до начала цикла создавать экземпляр списка (CatDir.Create), а после сразу его уничтожать (CatDir.Free). При следующем поиске список будет создаваться заново. Наконец, неплохо бы для большей информированности пользователя выводить конкретную папку, в которой в данный момент ведется просмотр, как это обычно делается в таких программах. Для этого у нас есть неиспользуемый теперь Label2 — будем надеяться, что длины оставшейся части формы хватит для большинства реально встречающихся имен папок. Все, что относится к выводу времени в цикле, мы удалим, а переменную ttold используем для того, чтобы отображать прошедшее время в конце поиска.

Теперь надо еще обеспечить выбор — искать только в заданной папке, как и раньше, или по всему списку вложенных папок. Для этого установим два компонента RadioButton ниже полоски редактора Edit1. Для того чтобы они были связаны только друг с другом, их следует установить на отдельную маленькую панель (Panel2), на панель Panel1 в дальнейшем нам придется устанавливать еще такие же кнопки. Чтобы сделать панель Panel2 невидимой на фоне Panel1, следует у нее установить свойство ParentColor в True, а свойство BevelOuter в bvNone (оно вместе с BevelInner определяет наличие и форму окантовки). В заголовке RadioButton1 запишем Только в каталоге и сделаем ее отмеченной по умолчанию, а в заголовке RadioButton2 — Включая подкаталоги. Шрифт для обеих кнопок сделаем "полужирный" Arial 8-го кегля. Если кому не терпится взглянуть на то, как это должно выглядеть, то окончательный внешний вид заголовка формы после всех изменений показан на рис. 14.1.

Так как процедура составления списка папок и пересчета файлов может затянуться, то пользователю нужно как-то сообщить, что мы делаем. Поэтому мы будем не просто очищать заголовок Label1, а выведем туда сообщение. Процедуру расчета времени CalculateTime мы, как и договаривались, удалим, а Calculatefiles перепишем следующим образом:

```
procedure Calculatefiles;
begin
ncount:=0;
if FindFirst(stpath+'\*',$23,sf)=0 then
{ $23 = не просматриваем системные файлы, тома и каталоги}
repeat
 ncount:=ncount+1;
```

```
until FindNext(sf)<>0; {пока файлы не закончатся}
FindClose(sf); {конец поиска}
Form1.ProgressBar1.Max:=ncount;
CatDir.Insert(0,stpath); {вносим единственный элемент в список}
end;.
```

После всех этих изменений процедура поиска (Button2Click) будет выглядеть так (неизмененные места я пропускаю для сокращения записи):

```
. . . . . {все, как было}
Form1.Label1.Caption:='Составление списка вложенных папок...'; {заголовок
Label1}
Form1.Label2.Caption:=''; {очистили заголовок Label2}
Application.ProcessMessages; {чтобы сразу сработало}
CatDir:=TStringList.Create; {создаем список папок}
if RadioButton2.Checked then {поиск во всех папках}
 GetTreeDirs(stpath,CatDir) {пересчитываем папки и файлы}
else Calculatefiles; {иначе только файлы, как раньше}
nfile:=0; nall:=0;
ttold:=Time;
ProgressBar1.Position:=0; {установили в 0}
for i:=0 to CatDir.Count-1 do
begin
 stpath:=CatDir[i];
 stsearch:='\*';
 stsearch:=stpath+stsearch; {строка для поиска файлов}
 if FindFirst(stsearch,$23,sf)=0 then
 { $23 = не просматриваем системные файлы, тома и каталоги}
 repeat
   Form1.Label1.Caption:=' Всего файлов '
   +IntToStr(ncount)+' Просмотрено: '+IntToStr(nall)+
   ' Найдено: '+IntToStr(nfile);
   Form1.Label2.Caption:=stpath;
   . . . . . {все, как было}
   st:=' '+IntToStr(sf.Size)+' bytes '+
      DateToStr(FileDateToDateTime(sf.Time))+
                ' кодировка:'+st;
   . . . . . {все, как было}
 until FindNext(sf)<>0; {пока файлы не закончатся}
end;
FindClose(sf); {конец поиска}
CatDir.Free;
Form1.Label1.Caption:= ' Всего файлов '+IntToStr(ncount)
           +'Просмотрено: '+IntToStr(nall)
           +' Найдено: '+IntToStr(nfile);
```

```
Form1.Label2.Caption:='Время поиска '
+FormatDateTime('hh:nn:ss',Time-ttold);
. . . . . {все, как было}
```

Для большей информативности я здесь заодно добавил вывод размера и даты создания найденного файла (через строку st). Заметьте также, что для вывода времени мы использовали функцию FormatDateTime.

Если мы зададим теперь поиск в какой-то достаточно большой по объему папке, то увидим, что происходит он очень некрасиво — т. к. WebBrowser каждый заново перезагружает страницу, то он все время "моргает" линейками прокрутки и все это к тому же сильно тормозит процедуру — попробуйте исключить вызов процедуры Navigate из цикла вообще, и вы увидите, что она стала выполняться в несколько раз быстрее. Но не сидеть же перед пустым экраном в ожидании, пока поиск не закончится? Давайте ограничим вывод во времени. Для этого поместим на форму компонент Timer, зададим ему интервал, например, 3000, а свойство Enable установим в False. Строку WebBrowser1.Navigate(Pchar(st)) перенесем из цикла в обработчик события onTimer, добавим к ней для надежности Application.ProcessMessages. Непосредственно перед началом цикла вставим строку Timer1.Enabled:=True, а сразу же по его окончании — Timer1.Enabled:=False. В самом цикле Application.ProcessMessages перенесем повыше, сразу после строки Form1.ProgressBar1.StepIt (иначе крутящиеся в процедуре ReadFileFormat циклы будут тормозить событие таймера). Теперь у нас обновление содержимого браузера будет происходить раз в три секунды (примерно, потому что циклы будут тормозить так или иначе), а если вдруг все выполнится быстрее, то таймер не сработает ни разу, и содержимое браузера обновится единственный раз при вызове процедуры Navigate уже после окончания цикла (см. полный листинг в *главе 13*).

Поиск заданной строки

Теперь приступим к поиску. Установим на Panel1 еще один Edit (Edit2) выше первого (соответствующим образом сдвинув все компоненты). Перед ними обоими установим два компонента Label для заголовков (шрифт полужирный" Arial 10-го кегля), а ниже Edit2 два компонента RadioButton с заголовками Фразу целиком (рис. 14.1) и Любое из слов, которые назовем RadioButtonAND и RadioButtonOR (т. к. они установлены на основной панели, то связи с RadioButton1 и RadioButton2 не будет). На самом деле это не совсем точные названия, продвинутый поиск требует трех функций: "Фразу целиком", "Все слова" (собственно функция AND) и "Любое из слов" (функция OR), а по большому счету и этим не ограничивается, см. далее.) Но удовлетворимся тем, что функция "Любое из слов" перекроет также и варианты, ко-

гда встречаются все слова, но в разных местах (как видите, я все время оставляю большой простор будущим поколениям для самоусовершенствования!).

В Label3 (напротив Edit2) напишем Искать что, а в Label4 (напротив Edit1) — Искать где. RadioButtonAND должна быть отмечена по умолчанию. У обоих компонентов Edit установим AutoSize и AutoSelect в False. Окончательно верхняя часть окна программы должна выглядеть так, как показано на рис. 14.1.

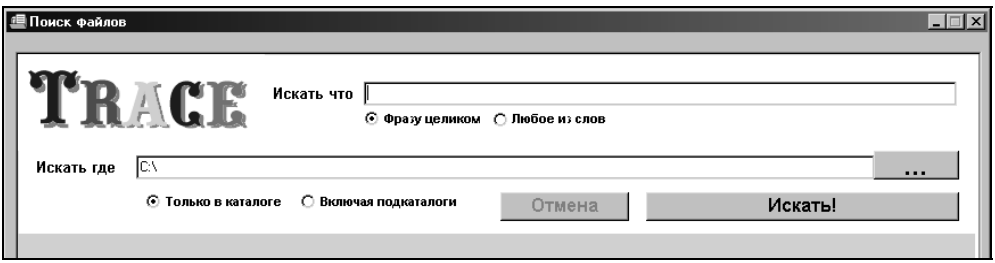


Рис. 14.1. Заголовок программы Trace для поиска заданной строки

В обработчике события onShow формы (а также в конце процедуры поиска) заменим три строки установок для редактора Edit1 на следующие:

```
procedure TForm1.FormShow(Sender: TObject); {при запуске}
begin
  Edit1.SelStart:=length(Edit1.Text);
  Edit1.SelText:=''; {курсор в конец текста Edit1}
  Edit2.SetFocus; {фокус в Edit2}
  Edit2.SelStart:=length(Edit2.Text);
  Edit2.SelText:=''; {курсор в конец текста Edit2}
end;
```

Для того чтобы обеспечить удобный запуск поиска, надо обязательно связать событие нажатия клавиши <Enter> в поле редактора Edit2 (**Искать что**) с процедурой щелчка по кнопке Button2. Выделим все содержимое обработчика щелчка на Button2 (т. е. нашу процедуру поиска) в отдельную процедуру, которую назовем SearchFile. Чтобы не редактировать в ней все строки, где встречаются наименования компонентов, в начале процедуры придется добавить инструкцию **with** Form1 **do begin**, а перед ее концом поставить дополнительный **end**. А процедура обработчика onClick для кнопки теперь будет выглядеть так:

```
procedure TForm1.Button2Click(Sender: TObject);
{Искать!}
begin
SearchFile;
end; {Button2}
```

И создадим обработчик по нажатию клавиши (onKeyDown) для редактора Edit2:

```
procedure TForm1.Edit2KeyDown(Sender: TObject; var Key: Word;
Shift: TShiftState);
begin {при нажатии Enter в поле редактора}
 if Key=vk_Return then SearchFile;
end;
```

Чтобы сделать совсем комфортно, свяжем процедуру по нажатию клавиши в поле редактора Edit1 (**Искать где**) также с этим обработчиком (напоминаю, что это делается выбором из выпадающего списка на закладке **Event**).

Теперь приступим к реализации нашей функции поиска. Сначала надо проверять, что именно записано в Edit2 — а вдруг там ничего нет, или одна буква, или одни пробелы? Если поразмыслить, то и двухбуквенные слова также не имеет смысла искать — значимых двухбуквенных слов очень мало, мы сделаем так, что в случае чего их можно будет задать в компании с каким-то другим словом при поиске по "Любому из слов". Кроме того, если в строке поиска задано всего одно слово, то функция "Любое из слов" теряет смысл, и мы тогда будем принудительно переводить программу в режим поиска "Фразу целиком". С учетом всего сказанного, в начале процедуры поиска (теперь это SearchFile) запишем:

```
. . . . .
{проверяем строку Edit2:}
stFind:=Edit2.Text;
if stFind='' then begin Edit2.SetFocus; exit end;
stFind:=Trim(stFind); {удаляем возможные пробелы по концам}
if length(stFind)<3 then begin Edit2.SetFocus; exit end;
          {одно- и двухбуквенные слова не ищем}
if pos(' ',stFind)=0 then RadioButtonAND.Checked:=True;
 {если слово только одно, то поиск только по AND}
. . . . .
```

Гениально! Но до собственно поиска мы так и не добрались, так что двинемся дальше. В список глобальных переменных добавим переменные stfile, stfind и sttext типа **string**, а в самый конец процедуры ReadFileFormat добавим строку:

```
result:=FindString; {поиск строки}
```

Теперь напишем саму функцию FindString, она окажется довольно большой:

```
function FindString:boolean; {поиск строки в файле}
var i,j,n:integer;
```

```
var ast: array [0..99] of string; {до 100 слов по OR}
var sttemp:string;
begin
result:=False; {и не надейтесь, что найдем}
assignfile(fd,fname);
reset(fd,1); {проверки не надо - уже открывали}
stFile:='';
while not eof(fd) do
begin
  blockread(fd,xs,1); {по одному байту}
  if xs>31 then {только для символов}
  begin
   if (st=' KOI-8') or (st=' cp866') then {перекодировка}
   if xs>127 then
   begin
    if nx=koi8 then ns1:=koi[xs]; {номер символа в KOI}
    if nx=altn then ns1:=alt[xs]; {номер символа в Alt}
    for i:=128 to 255 do {обратная кодировка, ищем символ}
    if win[i]=ns1 then break; {нашли его код по Win1251}
    xs:=i;
   end;
   stFile:=stFile+chr(xs); {имеем строку с содержимым файла}
  end;
end;
closefile(fd);
if Form1.RadioButtonAND.Checked then {ищем по AND}
begin
  n:=pos(AnsiUpperCase(stFind),AnsiUpperCase(stFile));
{от регистра не должно зависеть}
  if n=0 then exit; {ничего не нашли}
  result:=True;
{нашли и формируем фрагмент текста с найденной строкой:}
  stText:=copy(stFile,n,100);
  while pos('<',stText)<>0 do
  begin
   if pos('>',stText)<>0 then {удаляем возможные HTML-теги}
   delete(stText,pos('<',stText),abs(pos('>',stText)-
                  pos('<',stText))+1)
   else delete(stText,pos('<',stText),1);
  end;
  {выделяем его жирным:}
  insert('</B>',stText,length(stFind)+1);
  stText:='<B>'+stText+'<BR>';
end; {AND}
```

```
if Form1.RadioButtonOR.Checked then {ищем по OR}
begin
  i:=0;
  sttemp:=stFind; {чтобы строку не испортить к следующему разу}
  while pos(' ',sttemp)<>0 do {разбираем sttemp на слова}
  begin
   ast[i]:=copy(sttemp,1,pos(' ',sttemp)-1); {копируем слово}
   if (length(ast[i])<2) then continue;
         {однобуквенные слова игнорируем}
   i:=i+1; if i=98 then break;
   delete(sttemp,1,pos(' ',sttemp)); {удаляем до пробела}
   sttemp:=TrimLeft(sttemp); {пробелов может быть много}
  end;
  ast[i]:=sttemp; i:=i+1;
  stText:='';
  for j:=0 to i do {ищем по очереди}
  begin
   n:=pos(AnsiUpperCase(ast[j]),AnsiUpperCase(stFile));
   if n=0 then continue; {не нашли - ищем следующее}
   result:=True; {нашли по крайней мере одно
   и формируем фрагмент текста с найденным словом:}
   sttemp:=copy(stFile,n,100);
   while pos('<',sttemp)<>0 do
   begin
    if pos('>',sttemp)<>0 then {удаляем возможные HTML-теги}
    delete(sttemp,pos('<',sttemp),abs(pos('>',sttemp)-
                  pos('<',sttemp))+1)
    else delete(sttemp,pos('<',sttemp),1);
   end;
   {выделяем его жирным:}
   insert('</B>',sttemp,length(ast[j])+1);
   sttemp:='<B>'+sttemp+'<BR>';
   stText:=stText+sttemp; {нашли и добавили к тексту}
  end;
end; {OR}
stFile:=''; {освобождаем память}
end;
```

В этой функции мы сначала читаем файл целиком побайтно (о ужас! как будто нам не хватало чтения с диска при определении кодировки) и загружаем в строку символы, перекодируя их в соответствии с ранее определенной кодировкой. Есть сильное искушение ускорить этот процесс, и читать большими кусками, благо процедура `blockread` это позволяет. Но во избежание всяких неприятностей, мы не должны допустить, чтобы в строке случайно оказался

символ с нулевым значением, а также некоторые другие управляющие символы с кодами, меньше 32, иначе программа может просто рухнуть. И вместо того, чтобы потом анализировать строку, "выковыривая" из нее ненужные символы, я их таким образом сразу удаляю. Конечно, неплохо бы коды 10+13 (перевод строки) и 12 (перевод страницы) заменять на пробел (мы ниже в процедуре выводим фрагменты текста для сведения), но мы оставим это на потом, когда будем оптимизировать чтение с диска и там это можно будет сделать проще и быстрее. Отметим, что определять текущую кодировку нужно именно по строке с ее названием, а не по значению nx, т. к. определение по условию `(nx=koi8) or (nx=altn)` будет некорректным в случае, если ни одного символа со значением больше 127 не встретилось, кодировка определилась, как ASCII, и все три величины (`nx`, `koi8` и `altn`) одинаково равны нулю.

После того как содержимое файла (наконец-то!) окажется в строке (в кодировке Win1251), мы начинаем собственно поиск. Для варианта поиска строки в режиме "Фразу целиком" все просто — мы ищем первое вхождение заданной строки, и, если его нет, выходим из обеих процедур последовательно с возвращаемым значением `False`. Если же строка находится, то мы формируем из текста файла фрагмент в 100 знаков (строка `stText`), начинающийся с искомой строки, при этом выделяем ее тегами жирного шрифта. На всякий случай мы заранее удаляем из этого фрагмента возможные HTML-теги, которые нам могут навредить в дальнейшем при отображении. Отметим про себя, что тут есть неоднозначность, заключенная в том, что мы можем задать в поиске именно HTML-тег, и тогда мы получим фрагмент без него — все подобные мелочи надо записывать, чтобы при дальнейшем усовершенствовании программы сделать действительно более продвинутый вариант, а не новую версию "для галочки".

Гораздо сложнее устроена процедура поиска отдельных слов (по "OR"). Сначала мы разбираем строку поиска на отдельные слова, причем не очень хорошо, что мы делаем это каждый раз заново — в принципе можно и делать это один раз в самом начале. Но я учел, что процедура разбора строки даже в 100 слов (максимальная емкость нашего массива) все равно протекает много быстрее, чем наше многократное чтение файла и последующий поиск вхождения, и решил не усложнять программу — чем меньше запутанных условий, тем меньше вероятность ошибки. Потом мы ищем вхождение каждого слова в отдельности и формируем общий текстовый фрагмент для вывода. В конце всей процедуры мы на всякий случай обнуляем строку `stFile`, в которой содержится текст из файла — лучше, если по окончании процедуры или при ее прерывании по нажатию кнопки **Отмена** в памяти не останется большого массива символов.

Для того чтобы показать сформированный в `stFile` фрагмент, мы модернизируем процедуру `createHTMLtext`, добавив туда строку `write(ftemphtm,stText)`.

Чтобы она не выводилась в конце поиска, по окончании цикла в процедуре `SearchFile` (перед последним вызовом `createHTMLtext`) мы включаем строку `stText:='';`. Для законченности придадим проекту номер версии 1.0 (и внесем это в заголовок формы: `Поиск файлов 1.0`), и установим название проекта "Trace".

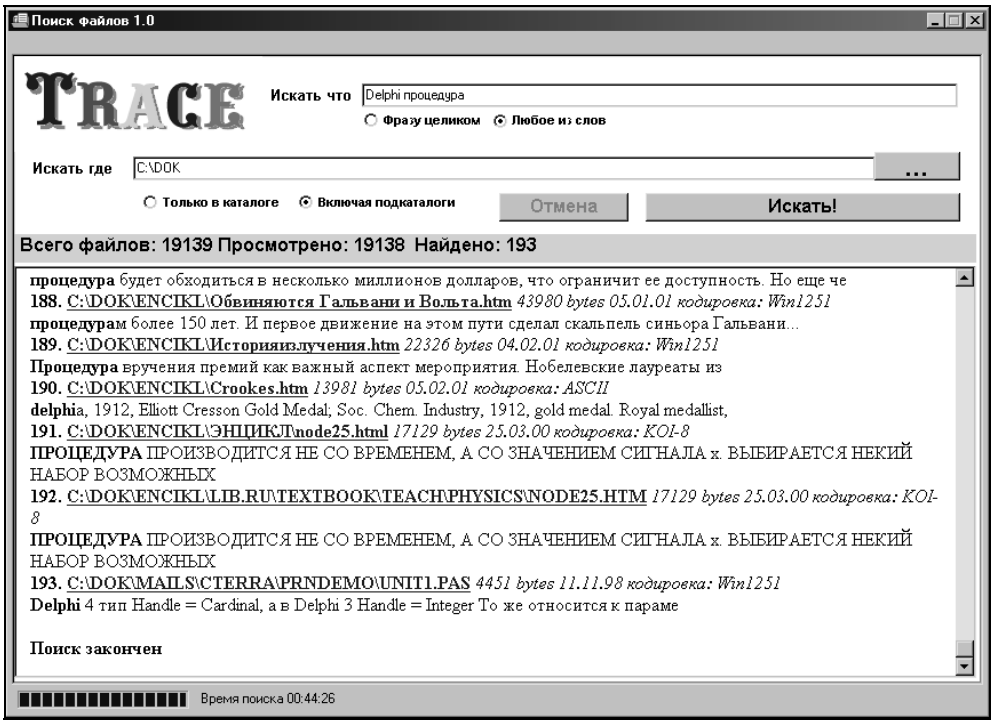


Рис. 14.2. Результаты работы программы Trace при чтении файлов с диска

Результаты поиска слов "Delphi" и "процедура" в большой папке с многоуровневыми вложенными каталогами и общим количеством файлов более 19 тыс. показаны на рис. 14.2. Как видите, процесс протекал довольно долго, почти 45 минут (Athlon XP 1,7 ГГц, 512 Мбайт памяти, диск 7200 об/мин), и в связи с этим просто напрашивается установка еще одной кнопки, которая бы приостанавливала процесс без его отмены, чтобы можно было спокойно рассмотреть промежуточные результаты. Конечно, такое возможно, но реализуется очень непросто — нужно по этой кнопке запомнить все промежуточное состояние поиска (номер папки в списке, имя файла, на котором остановились, и т. п.), а потом еще как-то умудриться запустить это дело заново с нужного места. Указав таким образом возможный путь решения проблемы, я опять же оставлю это на откуп читателям — вы увидите, что в дальнейшем,

когда мы, наконец, оптимизируем процесс чтения, эта проблема не будет столь острой.

Если вы немного погоняете программу, то обнаружите, что она заметно тормозит систему — загрузка памяти, правда, всего от 10 до 25% (причем с программой, запущенной в отладочном режиме из Delphi), зато загрузка процессора на все 100%! Должен вас разочаровать — на самом деле процессор так загружен отнюдь не анализом строк, а всякой чепухой, вроде взаимодействия с драйверами жесткого диска. Но, кроме того, сама программа очень медленно реагирует на всякие операции типа сворачивания и восстановления окна. Несколько улучшить положение можно, если расставить вызовы `Application.ProcessMessages` внутри всех критичных циклов — чтения с диска и поиска вхождения строк. Избежать "тормоза" вообще, в принципе, здесь не выйдет — посмотрим, что получится у нас потом, после обещанной модернизации.

Но перед тем как приступить, наконец, к ней, мы сделаем еще четыре вещи.

Полируем почти до блеска

Во-первых, т. к. процесс долгий (и в общем случае останется таковым даже после модернизации), никто не будет смотреть на экран все время, пока идет поиск, и неплохо бы выводить в заголовок свернутой программы проценты выполнения "плана" — фактически то, что показывает ползунок.

Если бы мы использовали компонент `Gauge` *(см. главу 13)*, то проценты у нас уже были бы готовыми. Но и тут вычислить их очень просто: объявим в процедуре `SearchFile` переменную `px` типа `integer` (дополнительно к уже имеющейся переменной `i`) и вставим в середину цикла, где у нас стоит вызов процедуры `Form1.ProgressBar1.StepIt`, следующие строки:

```
. . . . .
px:=round(nall*100/ncount);
Application.Title:= 'Trace '+IntToStr(px)+'%';
Form1.Caption:='Поиск файлов '+IntToStr(px)+'% выполнено';
Application.ProcessMessages; {чтобы все прокрутилось}
. . . . .
```

По окончании цикла добавим следующую строку:

```
Application.Title:= 'Trace: 100% выполнено';
```

При изменении заголовка формы мы номер версии в заголовке "1.10" потеряем, но это не страшно: так даже красивее. Вот вопрос: когда именно следует восстанавливать "испорченный" заголовок приложения, который у нас в начале есть просто "Trace"? Неплохо бы сделать так, чтобы пока программа

свернута, в заголовке не убиралась надпись "100% выполнено" после окончания поиска. А когда же надо ее убирать? Проще всего осуществить это по событию onPaint, которое обязательно возникнет при разворачивании окна — если программа закончила поиск, то заголовок вернется к нормальному виду, если нет — при сворачивании в первом же цикле нарисуется с процентами заново:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Application.Title := 'Trace';
end;
```

Вторая проблема — с повторным запуском. Отметим, что для предотвращения повторного запуска старый способ простого сравнения строк заголовков в данном случае уже не годится — придется анализировать строку на наличие в ней записи "Trace". Я специально отложил решение вопроса до этого момента: предотвращение повторного запуска тут важная вещь, т. к. мы работаем с временным файлом определенного названия, могут быть всякие коллизии (чтобы их не было, достаточно присваивать ftemphtm случайное имя, но лучше все же предотвратить повторный запуск). Вот как может выглядеть эта процедура (при попытке повторного запуска будем пробовать распахнуть запущенное приложение):

```
. . . . .
dWin:= GetWindow(Application.Handle, GW_HWNDFIRST);
while dWin <> 0 do
begin
 if (dWin <> Application.Handle) and
            {собственное окно игнорируем}
  (GetWindow(dWin, GW_OWNER) = 0) and {дочерние окна игнорируем}
  (GetWindowText(dWin, pbuff, sizeof(pbuff)) <> 0)
               {без названия игнорируем}
 then
 begin
   GetWindowText(dWin, pbuff, sizeof(pbuff));
          {получаем текст названия приложения}
   st:=string(pbuff); {переводим его в строку}
   if pos('Trace',st)<>0 then
   begin
      ShowWindow(dWin,SW_SHOWNORMAL);
      BringWindowToTop(dWin);
        {выводим первое приложение вперед}
    exit; {прерываем программу}
   end;
 end;
```

```
  dWin:= GetWindow(dWin, GW_HWNDNEXT);
      {ищем следующее приложение из списка}
end;
. . . . .
```

Разумеется, нужно объявить соответствующие переменные (см. проект SlideShow в *главе 4*) и внести модуль Windows в перечень модулей. Кстати, тут именно перебор файлов (а не просто вызов `FindWindow` с нужным заголовком) полностью оправдан — мы не знаем заранее, как будет выглядеть заголовок.

Третья задача вот какая: особенностью нашей программы является то, что здесь мы не задаем маску имени файла, ограничившись общим поиском, что, вообще говоря, неправильно. Мы отфильтровываем ненужные файлы, а нельзя ли, наоборот, еще и задавать тип файла, как это положено во всех таких поисковых программах? На самом деле нам ничего не стоит сделать это, вводя маску файла прямо в строке задания начальной папки. Чтобы сделать это удобным для пользователя, надо предусмотреть возможность задания как просто пути к папке, так и пути с именем файла, а также постараться сохранить однажды введенную маску файла в неприкосновенности, чтобы не приходилось при смене папки каждый раз ее вводить заново. Дальше нам остается только проводить анализ строки — если имя файла пусто, то мы ищем все файлы, как и раньше. Введем специальную переменную `stMask:string` и добавим такой текст в начало поиска после строки `if stpath='' then exit`:

```
. . . . .
stMask:=ExtractFileName(stpath);
if (stMask<>'') and (pos('*',stMask)<>0)
then {имя файла может быть только со звездочкой}
begin
 stMask:='\'+stMask;
 stpath:=ExtractFilePath(stpath);
end
else stMask:='\*';
. . . . .
```

То есть мы разделяем имя файла и путь к папке, причем знак "обратный слеш" приписываем к имени (в следующем операторе он у нас удалится, см. код в предыдущей главе). Здесь имя файла может быть только со звездочкой, т. е. представлять собой маску — более подробное пояснение этого ограничения см. далее. Теперь осталось во всех вызовах `FindFirst` ввести именно это имя. Удаляем в цикле строку `stsearch:='\*';`, а следующую строку записываем так:

```
stsearch:=stpath+stMask; {строка для поиска файлов}
```

Теперь перейдем к процедуре `GetTreeDirs` и вместо выражения

`FindFirst(Path+'\*',faAnyFile,sr)`

запишем `FindFirst(Path+stMask,faAnyFile,sr)`.

Наконец, в процедуре подсчета файлов `Calculatefiles` также заменим `'\*'` на `stMask`, а процедуру вызова диалога установки папки перепишем так:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
stMask:=ExtractFileName(Edit1.Text);
stpath:=Edit1.Text;
if (stMask<>'') and (pos('*',stMask)<>0)
then stMask:='\'+stMask else stMask:='';
{имя файла здесь может быть только со звездочкой}
ChDir(ExtractFilePath(Edit1.Text)); {устанавливаем текущую
    директорию и вызываем диалог установки директории:}
if SelectDirectory(stpath,[],0) then
begin
 if stpath[length(stpath)]='\' then
     delete(stpath,length(stpath),1);
 Edit1.Text:=stpath+stMask;
end;
Edit2.SetFocus; {фокус в Edit2}
end;
```

Заметки на полях

Имя файла здесь должно обязательно содержать звездочку (т. е. представлять собой маску) потому, что мы имеем дело не с именами настоящих (существующих на диске) файлов, а всего лишь со строками. А в строке отличить, например, имя файла "vasya.txt" от имени папки "vasya.txt" невозможно — папки могут иметь также имена с расширением. И поэтому функция `ExtractFileName` просто тупо читает все, что после последнего "обратного слеша", будучи примененной к строке "C:\DOK", она вернет имя несуществующего файла "DOK". Если бы мы имели дело с существующим дисковым файлом или папкой, то отличить их можно было бы по атрибутам — как при чтении, так и при записи — то, что содержится в параметре **`Name`** структуры `TSearchFile`, никаких разночтений не допускает. Но есть и еще одно обстоятельство: если функция `ExtractFilePath` всегда возвращает путь со знаком "обратный слеш" на конце (ее можно даже использовать вместо действия "+'\'"), то диалог открытия файла почему-то делает это только для имени диска ("C:\" — видимо, чтобы выглядело красивее), а папки всегда возвращаются без концевого "слеша". Из-за всей этой путаницы и пришлось городить сложные условия.

После всех этих установок у нас имя папки и в `Label2` всегда (кроме первого запуска с установками по умолчанию) будет демонстрироваться без "обрат-

ного слеша" на конце, а, например, для "голого" диска C: это действительно некрасиво, потому мы добавим его туда искусственно:

```
Form1.Label2.Caption:=stpath+'\'.
```

Кстати, теперь в программе есть одна недоработка, которую я оставляю вам в качестве домашнего задания. После того как мы введем в строке "Искать где" маску файлов, легко сообразить, что произойдет с ProgressBar. Предположим, в папке 1000 файлов, но только 10 из них с заданным нами расширением. Программа просмотрит только эти 10 файлов и поиск будет окончен, однако ползунок при этом вообще с места не успеет сдвинуться. Подумайте, как можно устранить этот недостаток?

Наконец, четвертая проблема, которую нам нужно решить — как помочь пользователю перейти к найденному файлу? Если ему придется запоминать результаты в уме, то вся наша работа обессмысливается. Для решения этой проблемы мы сделаем две вещи: во-первых, мы позволим сохранять полученный HTML-файл отдельно. Это просто: поставим на форму диалог сохранения SaveDialog1, установим у него свойство Filter в Файлы HTML|*.htm; *.html и напишем отдельную процедуру:

```
procedure Savehtmlfile;
var htmlname:string;
 fhtm:TextFile;
begin {запоминаем результаты поиска}
with Form1 do
begin
 SaveDialog1.InitialDir:=ExtractFileDir(Application.ExeName);
 htmlname:=Edit2.Text+'.htm'; {имя = равно строке поиска}
 SaveDialog1.FileName:=htmlname;
 if SaveDialog1.Execute then
 begin
  assignfile(fhtm,ftempname); {имя временного файла}
  Rename(fhtm,SaveDialog1.FileName); {создаем копию}
 end;
end;
end;
```

Вставим ее вызов в конец основной процедуры SearchFile:

```
. . . . .
if nfile>0 then {если нашли больше одного}
if Application.MessageBox('Сохранить результаты поиска?',
                 '',mb_OKCANCEL) = idOK
then Savehtmlfile;
. . . . .
```

Заметки на полях

В данном случае мы переименовываем наш временный файл ftempname, потому что в начале поиска мы все равно его создадим заново. Если в подобных случаях возникает задача скопировать файл под новым именем, то удобнее всего воспользоваться API-функцией копирования по такому шаблону:

CopyFile(pchar(fname),pchar(fnewname),True).

Если последний параметр равен True, как в этом примере, то функция возвратит ошибку при встрече уже существующего файла, иначе она его перепишет заново.

Немедленно возникает вопрос: ну и что делать с этими результатами? Просто демонстрируемые в окне браузера результаты нам бесполезны, потому что единственный способ их использовать, который я могу придумать, — распечатать сохраненный файл на принтере. Поэтому неплохо бы изобрести способ переходить к найденному файлу, как по обычной ссылке. Вспомним, что почти с каждым файлом в системе связано то или иное действие, и наша задача — попробовать реализовать эту возможность.

Запуск файлов из приложения

Для этого сначала преобразуем имя файла в как бы HTML-ссылку. Процедуру createHTMLtext мы перепишем так:

```
procedure createHTMLtext(stnum,st1,st:string);
begin
if FileExists(ftempname) then
append(ftemphtm) else exit; {вдруг его удалили?}
st1:='<B>'+stnum+'<A HREF="'+st1+'">'+st1+
'</A></B><I>'+st+'</I><BR>';
write(ftemphtm,st1); {строка с именем файла}
write(ftemphtm,stText); {выводим строку с текстом}
closefile(ftemphtm);
end;
```

По этой процедуре у нас будет из имени формироваться "ссылка", а номер файла и последующий текст останутся просто текстом с соответствующим форматированием. Теперь внесем изменения в процедуре поиска SearchFile. Там, где у нас формировались строки для последующей записи в ftempname (перед самым концом цикла), изменим текст на такой:

```
. . . . .
st:=' кодировка:'+st;
st1:=IntToStr(nfile)+'. '; {номер найденного файла}
createHTMLtext(st1,fname,st);
. . . . .
```

Отсюда, кстати, понятно, зачем я в данном случае изменил своему правилу и передавал в createHTMLtext строки через параметры, а не через глобальные переменные, как обычно — в случае чего строки легко поменять местами. А по окончании цикла мы вовсе откажемся от вызова createHTMLtext, нам ссылки выводить уже не надо, и по большому счету достаточно всего одной строки "Поиск закончен" с меткой:

```
. . . . .
st:='<BR><B><A NAME="1">Поиск закончен</A></B>';
st1:='';
stText:='';
append(ftemphtm); {это вместо createHTMLtext}
write(ftemphtm,st); {выводим строку с текстом}
closefile(ftemphtm);
. . . . .
```

"Ссылки" мы имеем, а что с ними делать? Для того чтобы запустить навигацию по ссылке из компонента WebBrowser, надо использовать событие onBeforeNavigate2 (существовали ли в природе события BeforeNavigate0 и BeforeNavigate1 — не спрашивайте, не в курсе). Найдите это событие на закладке **Events** для WebBrowser1 (оно самое первое по списку) и создайте обработчик. Само событие возникает при любых попытках изменить содержимое окна браузера — например, оно возникнет и при вызове метода Navigate, что мы делаем регулярно, и эти вызовы для нас явно лишние, так что придется придумывать, как их обойти. Из всего чудовищного количества параметров, которые передаются в процедуре WebBrowser1BeforeNavigate2, нас интересует только два: URL и Cancel. Если второму параметру придать внутри процедуры значение True, то никаких изменений в окне браузера не произойдет. Но перед этим можно сделать что-то полезное, чем мы и воспользуемся — ведь по-настоящему у нас нет никаких ссылок в смысле сетевых адресов, а есть только имя дискового файла, которое через параметр URL и передается. Вместо того чтобы заставлять браузер пытаться загрузить адрес файла на диске (чего он сделать не сможет, как будто между файлом в Сети и файлом на вашем диске есть принципиальная разница — а ведь помнится, кто-то самый богатый в мире твердил о полной интеграции Web-функций в систему, не так ли?), мы будем вызывать функцию ShellExecute, выполняющую то действие, которое в системе с данным файлом ассоциировано. И даже если не ассоциировано, то мы найдем выход — запустим стандартный диалог Windows "Открыть с помощью".

Чтобы процедура не выполнялась, когда мы сами вызываем метод Navigate для обновления окна браузера, можно воспользоваться тем, что мы всегда передаем ему имя нашего временного файла (один раз с добавкой метки #1),

и отфильтровать соответствующие события. Но мы поступим более грамотно. В начале поиска, перед оператором `Timer1.Enabled:=True`, мы сделаем вид, что процедуры `WebBrowser1BeforeNavigate2` вообще не существует:

```
WebBrowser1.OnBeforeNavigate2:=nil;
```

А по окончании цикла поиска в самом конце процедуры `Searchfile` запишем:

```
. . . . .
Application.ProcessMessages;
WebBrowser1.OnBeforeNavigate2:=WebBrowser1BeforeNavigate2;
. . . . .
```

Обработку событий в системе (`Application.ProcessMessages`) мы здесь вставили еще раз вот почему: может случиться так, что браузеру будет отдана команда `Navigate` (см. выше по тексту процедуры), а при смене отображаемых страничек он будет ее выполнять с еще большей неторопливостью, чем обычно, и обратится к поиску процедуры `BeforeNavigate2` уже тогда, когда она будет реально для него существовать (сама процедура `SearchFile` закончится), в результате чего она выполнится, когда нам этого еще вовсе не надо.

Теперь заполним собственно обработчик `OnBeforeNavigate2`:

```
procedure TForm1.WebBrowser1BeforeNavigate2(Sender: TObject;
const pDisp: IDispatch; var URL, Flags, TargetFrameName, PostData,
Headers: OleVariant; var Cancel: WordBool);
begin  {когда браузер хочет Navigate}
if ShellExecute(Self.Handle, 'open',Pchar(AnsiUppercase(URL)),
                nil, nil, SW_SHOWMAXIMIZED)<=32
then
 begin {если <32 - ошибка, вызываем диалог "Открыть с помощью"}
  st:='shell32.dll,OpenAs_RunDLL '+st;
  ShellExecute(Self.Handle, 'open','rundll32.exe',
         Pchar(st), nil, SW_SHOWNORMAL);
 end;
 Cancel:=True; {и дальше ничего не делаем}
end;
```

Функция `ShellExecute` содержится в модуле ShellAPI, поэтому вам придется включить его в предложение **uses**. Тогда при щелчке мышью на "ссылке" с именем файла у нас запустится нужное приложение и загрузит его. Если это файлы без расширений или с незарегистрированными расширениями, то программа предложит с таким-то файлом что-нибудь сделать. Во всем этом деле есть один нюанс: исполняемые файлы мы исключили из рассмотрения, но такие, например, типы файлов, как BAT или JS (в сущности, тоже просто тек-

стовые файлы) попытаются выполниться. Можно попробовать их также отфильтровать, но всего все равно не предусмотришь, так что надо будет оговорить это в Справке. Кстати, кроме всего прочего, правая кнопка мыши при щелчке на "ссылке" будет нормально работать, вызывая обычное в таких случаях меню — т. е. вы можете, например, скопировать файл в другое место через пункт **Сохранить объект как** или даже что-то напечатать (а вот пункт **Скопировать ярлык** работать не будет).

Но это не все. Что нам делать с сохраненными файлами-результатами поиска? Ведь при загрузке в обычный браузер ссылки останутся недействующими. Поэтому нам придется немного потесниться, и установить справа внизу кнопочку **Загрузить результаты** (`Button4`). Создавать сейчас меню только для одного пункта мы не будем, а позже все равно его создадим для настроек и справки, и тогда перенесем функцию кнопки в него. У компонента `Label2` с установленным в `True` свойством `AutoSize` придется ограничить длину (в пункте **Constraints | MaxWidth** ввести значение 530 — как раз до кнопки). Кроме этого, на форму придется установить компонент `OpenDialog` с тем же фильтром, что и у диалога сохранения, и написать следующий обработчик события нажатия кнопки:

```
procedure TForm1.Button4Click(Sender: TObject);
begin  {загрузить результаты}
WebBrowser1.OnBeforeNavigate2:=nil;
If OpenDialog1.Execute then
WebBrowser1.Navigate(Pchar(OpenDialog1.FileName));
Application.ProcessMessages;
WebBrowser1.OnBeforeNavigate2:=WebBrowser1BeforeNavigate2;
end;
```

А чтобы кнопка не создавала нам проблем во время поиска, мы в начале процедуры поиска ее дезактивируем:

```
. . . . .
Button3.Enabled:=True; {активируем Отмену}
Button2.Enabled:=False; {дезактивируем Поиск}
Button4.Enabled:=False; {дезактивируем Результаты}
. . . . .
```

А в конце снова активируем:

```
. . . . .
Button2.Enabled:=True; {активируем Поиск}
Button4.Enabled:=True; {активируем Результаты}
Button3.Enabled:=False; {дезактивируем Отмену}
. . . . .
```

Излишне добавлять, что указанный метод вызова ShellExecute годится, естественно, в любом случае, когда нужно вызвать внешнюю программу — не обязательно это делать столь заковыристым путем через браузер. Но вот работоспособность программы во всех версиях Windows я гарантировать не могу: все, что касается WebBrowser, явно должно быть как-то связано с конкретной версией Internet Explorer (у меня 6.0). Причем без больших хлопот я не могу даже проверить работоспособность всего этого, например, в Internet Explorer 4 или Internet Explorer 5 — ведь вернуться на предыдущую версию Internet Explorer нельзя без полной переустановки Windows. Остается только надеяться, что все будет "в абажуре", а если нет — при необходимости можно запретить запуск программы в чуждой ей среде.

Оптимизация чтения через memory mapped files

Ах да, вы же, наверное, с нетерпением ждете, когда я приступлю к оптимизации доступа к файлу — нельзя же дальше терпеть это любительство с многократным чтением с диска, притом побайтно! На самом деле это очень просто, нам даже не придется вносить капитальных изменений в программу. Мы, кстати, уже умеем создавать отображения файлов в память — см. *главу 7*. Но здесь все еще проще — никаких структур ведь не надо, требуется только перевести содержимое файла в строку. Нам нужно осуществить такую последовательность операций: получить дескриптор дискового файла (CreateFile), создать файл в памяти (знакомая функция CreateFileMapping) и получить указатель на этот файл (также знакомая нам MapViewOfFile). Потом мы считаем значения по этому указателю в строку и в обратном порядке все уничтожим (UnmapViewOfFile и CloseHandle).

Заметки на полях

Для того чтобы ускорить процесс чтения файлов, есть, вообще говоря, несколько механизмов. Это потоковое чтение с использованием TFileStream (см. [3]), простое применение API-функций CreateFile и ReadFile с созданием вручную (через механизм Allocate) буфера в куче, чтение нетипизированных файлов в массивы по указателю, в динамические массивы или просто сразу в строку (подробности см. в *главе 21*). Механизм отображения файлов в память (file mapping) хорош тем, что прост, и в то же время все заботы о кэшировании система берет на себя. Формально говоря, с точки зрения системы в данном случае, когда мы файл читаем последовательно байт за байтом, лучшим способом было бы использование потокового чтения — при нем оперативная память меньше загружена и больше ресурсов остается другим приложениям. Но наше ограничение читаемого файла размером 500 Кбайт фактически сводит эту разницу к минимуму — если она вообще будет заметной. Отметим, что при отображении файла в память, естественно, его когда-то в любом случае придется

прочесть с диска. Разница та, что теперь это делает за нас система — обращение с файлами через mapping в принципе ничем не отличается от обращения с любыми другими данными — например, массивом или строкой — в памяти, для которой система сама решает, когда и какой объем подгрузить в оперативную память, а какой — оставить на диске, причем сделает это максимально быстро и "прозрачно" для пользователя. В таких системах, как у меня, где оперативной памяти достаточно много (512 Мбайт), это должно сильно ускорять работу — хотя, что означают слова "достаточно много памяти" применительно к современным ОС, точно сказать никто не может.

Перенесем проект Trace в новую папку (Glava14\2), придадим ему новый номер версии (1.10, не забудем исправить начальный заголовок формы) и объявим две новых глобальных переменных типа integer: fmax и fsize. Переменные xw и fd можно удалить — они нам больше не понадобятся. Замечу в скобках, что удалять ненужные переменные при переделке программы следует всегда — это дает дополнительную гарантию, что ошибки "вылезут" сразу при компиляции и вам не придется долго давить на <F8>, чтобы обнаружить присвоение значения переменной, которая уже вовсе не используется. Кстати, еще одно замечание — иллюстрация к положению, что по мере возможности программу нужно писать сразу по плану. Так, отладка этой программы после переделки (а вы увидите, что изменения не так уж и велики) заняла у меня ровно столько же времени, сколько написание предыдущего варианта. В данном случае я шел на это сознательно (чтобы продемонстрировать, насколько могут различаться по времени работы программы, выполненные "по-старому" и "по-новому"), а вообще-то поступать так — пустая трата времени.

В основном цикле, после оператора nall:=nall+1;, добавим следующие две строки:

```
. . . . .
if sf.Size>fmax then continue; {если больше fMax - не рассматриваем}
fsize:=sf.Size;
. . . . .
```

Они нам сохранят размер текущего файла, который понадобится позже, и заменят проверку размера файла, которая стоит в процедуре ReadFileFormat — нам придется от нее избавиться. Переменная fMax также требует инициализации, и мы создадим обработчик события onCreate формы, в который временно поместим такую строку (потом мы fMax будем инициализировать через настройки):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 fmax:=500*1024; {500 Кбайт}
end;
```

Теперь создадим нашу главную новую функцию для получения строки в памяти — ReadMapFile.

```
function ReadMapFile:boolean;
var
hfile,hMap:THandle;
pFile:pointer;
pb:^byte;
i:integer;
begin  {читаем файл в строку stFile}
result:=True;
try
 hFile:=CreateFile(Pchar(fname),GENERIC_READ,0,nil,
          OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0);
 hMap:=CreateFileMapping(hFile,nil,PAGE_READONLY,0,0,nil);
 pFile:=MapViewOfFile(hMap,FILE_MAP_READ,0,0,0);
 stFile:='';
 for i:=0 to fsize-1 do
 begin
  pb:=pointer(integer(pFile)+i);
  {Избавляемся от символов <32, заменяя их на пробелы:}
  if pb^>31 then
  stFile:=stFile+chr(pb^) else stFile:=stFile+' ';
 end;
 UnmapViewOfFile(pFile);
 CloseHandle(hMap);
 CloseHandle(hFile); {удалили все объекты}
except
  result:=False;
end;
end;
```

Заметьте, что мы не стали разбираться с возможными ошибками по отдельности — нас интересует только общий результат: прочли — отлично, не прочли (вдруг файл занят) — и бог с ним. Процедура формирования строки stFile здесь также далека от совершенства (почему — вы узнаете в *главе 21*), но нам так в данном случае удобнее — заодно мы избавляемся от лишних символов.

Заметки на полях

При работе с указателями, которые ссылаются на массив символов (например, при размещении строк в глобальной куче) всегда есть искушение использовать тип Pchar. В самом деле, вроде бы эта штука устроена специально так, что одновременно представляет собой и строку и указатель на эту строку, почему бы не написать просто:

```
var pst: Pchar;
stFile: string;
pFile: pointer;
. . . . .
pst:=pFile; {Pchar, как указатель}
stFile=pst; {Pchar, как строка}
```

И компилятор, заметьте, не сделает вам ни одного замечания! Мало того, если вы такую вещь проделаете, то даже можете получить при трассировке правильный результат: обе строки будут содержать в себе весь текст файла. Но если вы попробуете поработать всерьез, то скоро программа начнет рушиться в самые непредсказуемые моменты — и это относится не только к данному случаю, например, в предотвращении повторного запуска мы тоже используем не Pchar, а массив символов, хотя они формально вроде бы совместимы. Анализ показал, что в нашем случае ошибки обусловлены следующим обстоятельством: файловый массив в памяти, на который будет нацелен указатель pFile, совершенно необязательно заканчивается нулем — это будет так, например, если память в этом месте была до запуска программы заполнена нулями. Но стоит разок-другой попользоваться памятью, как она замусорится, и хотя при присвоении значения указателя на массив в памяти Pchar-строке длина массива формально будет присвоена длине строки (т. е. под нее будет зарезервирована память), но это не будет настоящая строка типа Pchar, т. е. заканчивающаяся нулем. Дело еще и в том, что здесь никто не гарантирует отсутствия нулевого символа внутри файла, и тогда мы просто потеряем всю оставшуюся его часть (см. на эту тему также *главу 21*). Резюме такое — с Pchar надо обращаться с осторожностью, использовать тип Pchar безоговорочно можно только, если он изначально создается именно как Pchar, а не через присвоение значения, как указателю. Эту тему мы еще будем более подробно разбирать в *главе 21*.

Теперь приступим к внесению изменений в остальной части программы и начнем с функции ReadFileFormat. Внесем в ее заголовок переменную i:integer для перебора элементов строки. После того как мы удалим все, относящееся к чтению из файла, и заменим его на анализ строки (в том числе и фильтрацию по условию (xs=0) **or** (xofs=0), она нам также не потребуется, все нули отсеяли заранее), эта функция будет выглядеть следующим образом:

```
function ReadFileFormat: boolean;
var i:integer;
begin
result:=ReadMapFile; {в stFile - содержимое файла}
if result=False then exit;
 altn:=0; {в эти переменные будем накапливать статистику}
 winn:=0;
 koi8:=0;
 i:=1;
 while i<length(stFile)-1 do {на 2 меньше длины строки}
```

```
 begin
   xofs:=ord(stFile[i]);
   i:=i+1;
   xs:=ord(stFile[i]);
   i:=i+1; {номер для следующего раза}
   if (xofs>127) and (xs>127) then
{только русские двухбуквенные сочетания}
   . . . . . {все, как было}
 end;
 nx:=MaxIntValue([altn,winn,koi8]); {максимальное из полученного}
 . . . . . {все, как было}
 result:=FindString; {поиск строки}
end;
```

Оператор `closefile(fd)` мы в конце также удалили. Теперь внесем аналогичные изменения в функцию `FindString`, причем необходимые локальные переменные уже имеются:

```
function FindString:boolean; {поиск строки в файле}
var i,j,n:integer;
var ast: array [0..99] of string; {до 100 слов по OR}
var sttemp:string;
begin
result:=False; {и не надейтесь, что найдем}
j:=1;
if (st=' KOI-8') or (st=' cp866') then {перекодировка}
begin
 while j<length(stFile) do {на 1 меньше длины строки}
 begin
  xs:=ord(stFile[j]);
  if xs>127 then
  begin
    if nx=koi8 then ns1:=koi[xs]; {номер символа в KOI}
    if nx=altn then ns1:=alt[xs]; {номер символа в Alt}
    for i:=128 to 255 do {обратная кодировка, ищем символ}
    if win[i]=ns1 then break; {нашли его код по Win1251}
    xs:=i;
  end;
  stFile[j]:=chr(xs);
  j:=j+1; {номер для следующего раза}
 end;
end; {конец перекодировки}
if Form1.RadioButtonAND.Checked then {ищем по AND}
. . . . . {все, как было}
end;
```

Кстати, удалим из строки отображения времени поиска разряды часов, сейчас вы увидите, что они нам больше не понадобятся:

```
. . . . .
st:=FormatDateTime('hh:nn:ss',Time-ttold);
delete(st,1,3); {удаляем часы - они нам больше не понадобятся}
Form1.Label2.Caption:='Время поиска '+st;
. . . . .
```

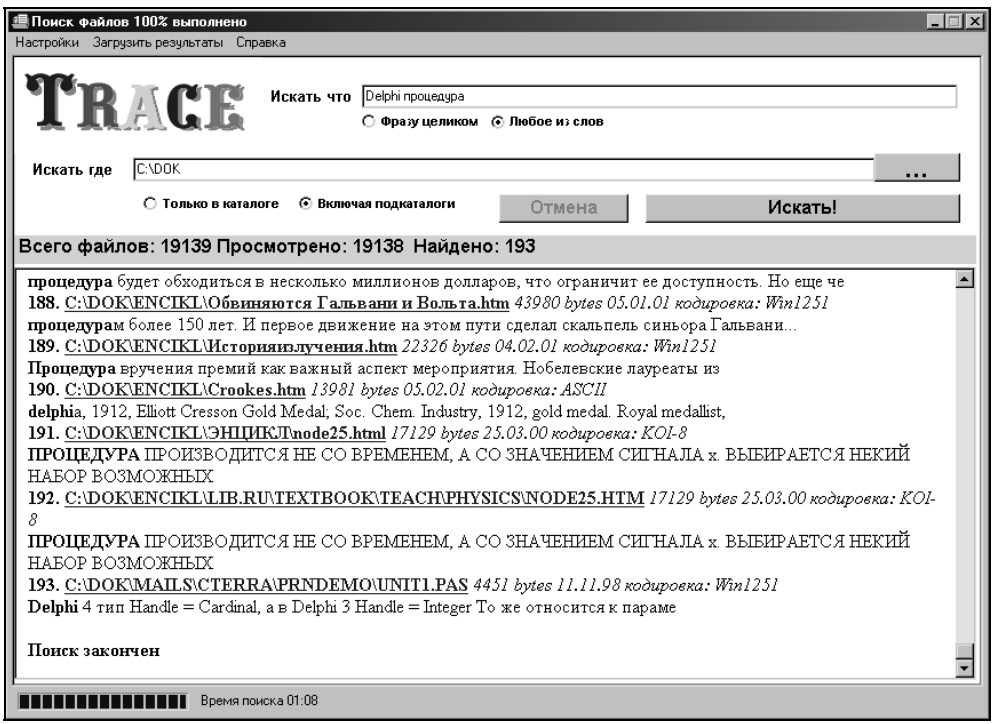


Рис. 14.3. Результаты работы программы Trace при чтении файлов из памяти

Вот и все доделки: если вы запустите программу теперь, то изумитесь результатам (рис. 14.3): поиск, аналогичный тому, что мы делали в старом варианте, занял всего 1 минуту 8 секунд — в сорок раз быстрее! Признаюсь, для пущего эффекта я закрыл все посторонние программы, запустив, правда, Trace в отладочном режиме из среды Delphi. Если вы поэкспериментируете, то увидите, что время теперь очень зависит от количества свободной памяти, даже с моими 512 Мбайтами достаточно запустить еще пару-другую "монстров" вроде Word или Photoshop, как время поиска увеличится в разы. Но, конечно, таких умопомрачительных "тормозов", как при прямом чтении с диска, мы уже не получим. Кстати, на фоне общего увеличения скорости поиска

теперь будет очень заметно, как тормозит браузер при загрузке странички — особенно в конце, когда нужно перейти на метку. Но тут я ничего поделать не могу — к тому же эффект этот заметен только в Windows 98, в XP все перезагружается значительно быстрее.

Заметки на полях

Казалось бы, при перекодировке нам логично, как мы делаем это в той же процедуре далее, воспользоваться парой процедур:

```
delete(stFile,j,1);
insert(chr(xs),stFile,j);
```

Предостерегаю — если вы так поступите, то вместо того, чтобы ускорить процесс, вы его резко замедлите. Представьте себе, как работают процедуры `delete` и `insert`: для того чтобы удалить единственный символ, надо "перелопатить" всю строку, сдвинув все, что было после удаленного символа, а затем повторить это в обратную сторону для `insert`, и так каждый символ — а если их полмиллиона? Отметим заодно, что операция конкатенации (складывания строк), в отличие от `delete` и `insert`, выполняется более быстро, хотя там тоже есть свои "тараканы" (см. на эту тему *главу 21*). Но вы спросите — а как же дальше, там ведь мы используем `delete` и `insert` для удаления HTML-тегов? Во-первых, там строки однозначно короче, во-вторых, мы делаем это всего один-два раза, а не с каждым байтом строки, так что тут на замедление можно не обращать внимания. А вот повторение разбора строки поиска каждый раз при поиске по "OR" — не здорово, конечно, и тут кроется резерв для небольшого увеличения быстродействия, который в данном случае, в отличие от варианта с чтением с диска, стоит принять во внимание. Увеличение это, однако, будет настолько гомеопатическим (можете проверить), что можно и не усложнять программу, особенно учитывая то, что время поиска значительно в большей степени зависит от наличия свободной памяти. Да и про `ProgressBar` не забудем.

На рис. 14.3 вы уже можете видеть меню с пунктами **Настройки**, **Справка** и **Загрузить результаты**. Справкой мы займемся в *главе 16*, а сейчас давайте решим вопрос с настройками и результатами поиска.

Настройки

Составим полный список того, что именно мы хотим сохранить в настройках.

1. Состояние кнопок `RadioButtonAND` и `RadioButton1` (достаточно двух ведущих).
2. Максимальный размер файла для поиска (`fMax`).
3. Последнюю папку (возможно, с маской имени файла), в которой осуществлялся поиск.
4. Строку с заданием запрещенных расширений файлов.

Исключительно ради последнего пункта нам следует здесь ввести кнопочку **Вернуть установки по умолчанию** — строка длинная и специфическая, и пользователь может ее так "заредактировать", что сам забудет, что к чему. Сохранять ли строку поиска — вопрос дискуссионный, и мы его еще обсудим чуть ниже.

Итак, добавим модуль IniFiles в **uses**, создадим переменную `IniFile:TIniFile` и сделаем следующий фокус: создадим строковую переменную `stExt`, а существующую константу `stExt` назовем `stDefExt`, в тексте программы при этом ничего менять не придется. Перепишем уже существующий у нас обработчик `onCreate` формы таким образом:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini')); {если не
было - создаем, иначе открываем}
 with IniFile do
 begin
  if SectionExists('Main') then
        {если файл и секция в нем уже есть}
  begin
   if ReadBool('Main','AND',True) then
   RadioButtonAND.Checked:=True else
   RadioButtonOR.Checked:=True;
   if ReadBool('Main','ReadDirOnly',True) then
   RadioButton1.Checked:=True else
   RadioButton2.Checked:=True;
   Edit1.Text:=ReadString('Main','Path','C:\');
   fMax:=ReadInteger('Main','MaxSizeOfFile',500*1024);
   stExt:=ReadString('Main','Taboo extension',stDefExt);
  end else {если секция еще не создана}
  begin
   WriteBool('Main','AND',True);
   WriteBool('Main','ReadDirOnly',True);
   WriteString('Main','Path','C:\');
   WriteInteger('Main','MaxSizeOfFile',500*1024);
   WriteString('Main','Taboo extension',stDefExt);
   stExt:=stDefExt;
   fMax:=500*1024;
  end;
  Destroy;
 end;
end;
```

Отлично, теперь реализуем запоминание текущих установок. В процедуре `Form1.Destroy` запишем:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini')); {открываем INI}
with IniFile do
begin
 WriteBool('Main','AND',RadioButtonAND.Checked);
 WriteBool('Main','ReadDirOnly',RadioButton1.Checked);
 WriteString('Main','Path',Edit1.Text);
 WriteInteger('Main','MaxSizeOfFile',fMax);
 WriteString('Main','Taboo extension',stExt);
 Destroy;
end;
if FileExists(ftempname) then
erase(ftemphtm); {уничтожаем временный файл}
end;
```

Вернемся тут к вопросу — сохранять ли строку поиска? Я не стал этого делать, ориентируясь на то, что в большинстве случаев пользователь не настолько рассеян, чтобы закрывать программу с нужной ему строкой, и при начальной загрузке она будет только мешаться. Но если читатель решит, что сохранять надо — тогда имеет смысл не уничтожать временный файл `ftempname`, содержащий результаты поиска по этой строке, а также загружать его при начальном запуске программы — это будет и логичное и красивое решение. Подобным образом поступают многие программы. Самым грамотным решением было бы даже ввести отдельную установку — сохранять строку/не сохранять.

Теперь реализуем установки для `fMax` и `stExt` по ходу работы с программой. Я еще раньше, как вы видели на рис. 14.3, создал меню с пунктом **Настройки**. Данный пункт я назвал `Set1` и присвоил ему горячую клавишу <F2> (хотя в главном меню это и не отображается, и даже всплывающую подсказку (свойство `Hint`), которая тут как раз позарез нужна, для `MainMenu` создать просто так не удастся). Поместим на форму, как и в *главе 9*, компонент `Panel` (он получит имя `Panel3`), и сразу установим для него свойство `Visible` в `False`. После того как мы расположим на этой панели нужные компоненты, она приобретет вид, показанный на рис. 14.4. Тут для разнообразия мы использовали для ввода чисел упоминавшийся ранее компонент `SpinEdit` (закладка **Samples**; у него надо установить свойство `MinValue` в 1, а `MaxValue` в 100 000 — больше, чем 100-мегабайтный файл, вряд ли кто-то захочет просматривать).

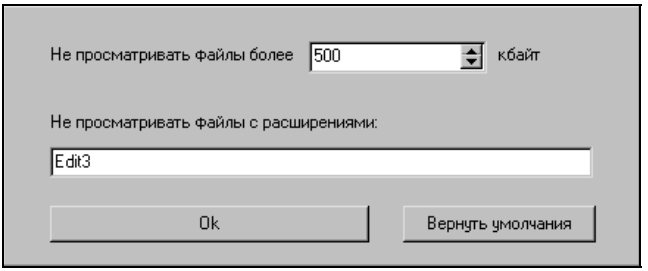


Рис. 14.4. Панель настроек программы Trace

Далее напишем четыре обработчика событий:

```
procedure TForm1.Set1Click(Sender: TObject);
begin {пункт меню Настройки}
 Edit3.Text:=stExt; {установки на панели настроек}
 SpinEdit1.Value:=fMax div 1024;
 Panel3.Visible:=True; {панель с настройками}
 SpinEdit1.SetFocus;
end;

procedure TForm1.SpinEdit1KeyDown(Sender: TObject; var Key: Word;
                Shift: TShiftState);
begin
{нажатие на Enter в SpinEdit и Edit3 после очередного ввода}
if key=vk_Return then
begin
 FindNextControl(Sender as TWinControl, true, true,
         false).SetFocus;
 Edit3.SelStart:=length(Edit3.Text);
 Edit3.SelText:=''; {в Edit3 курсор - в конец текста}
end;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin {Ok на панели настроек}
 stExt:=Edit3.Text;
 fMax:=SpinEdit1.Value*1024;
 Panel3.Visible:=False; {прячем панель с настройками}
end;

procedure TForm1.Button5Click(Sender: TObject);
begin {вернуть умолчания}
 Edit3.Text:=stDefExt;
 SpinEdit1.Value:=500;
end;
```

По первому из них (щелчок на пункте меню Set1 **Настройки**) панель показывается, и фокус устанавливается в SpinEdit1. Второй обработчик — общий для обоих редакторов при событии onKeyDown, если нажата клавиша <Enter> (для Edit3 надо объявить эту процедуру для того же события через закладку **Events**). В нем не только передается фокус ввода, но и курсор в Edit3 устанавливается в конец строки — так меньше вероятность что-то испортить и удобнее строку разглядывать. Третий — щелчок на кнопке Button4 **Ok**, закрывает панель и устанавливает величины. В результате, если мы откроем панель (<F2>), при троекратном нажатии <Enter> она закроется без каких-то изменений в величинах. Наконец, четвертая процедура — обработчик события нажатия на кнопку Button5 **Вернуть умолчания** — устанавливает наши "умолчательные" значения. Программа эта не такая "попсовая", как SlideShow, поэтому возиться с закрытием панели по щелчку вне ее я не стал — если читатель желает, он может доделать и это самостоятельно (образец см. в *главе 9*).

Кроме этого, в меню уже имеется и пункт **Загрузить результаты** (Res1), который должен делать то, что в предыдущем варианте делала кнопка Button4. Кнопку мы удалим, для Label2 в свойстве Constraints.MaxWidth опять запишем 0, чтобы она могла растягиваться до конца формы. Заменим в двух местах Button4 на Res1. Выводить диалог "Сохранить результаты" каждый раз неудобно — лишнее действие для пользователя, поэтому мы изменим концовку процедуры поиска, удалив из нее вызов MessageBox, и запишем вместо него такой оператор:

```
if nfile>0 then {если нашли больше одного}
  Res1.Caption:='Сохранить результаты'
else Res1.Caption:='Загрузить результаты';
```

Теперь создадим обработчик обращения к пункту меню **Загрузить результаты** Res1Click, перенесем в него содержимое обработчика нажатия на кнопку **Button4** и дополним его вызовом диалога по условию, что заголовок в меню предлагает сохранение:

```
procedure TForm1.Res1Click(Sender: TObject);
begin {результаты}
if Res1.Caption='Сохранить результаты' then
            {если поиск закончился}
begin
 Res1.Caption:='Загрузить результаты';
 if Application.MessageBox('Сохранить результаты
        поиска?','',mb_OKCANCEL) = idOK
 then Savehtmlfile;
 exit; {сохраняем и выходим}
end;
```

```
{загрузить результаты:}
WebBrowser1.OnBeforeNavigate2:=nil;
If OpenDialog1.Execute then
WebBrowser1.Navigate(Pchar(OpenDialog1.FileName));
Application.ProcessMessages;
WebBrowser1.OnBeforeNavigate2:=WebBrowser1BeforeNavigate2;
end;
```

И чтобы уж совсем довести программу до ума, создадим еще и такой обработчик события onClose формы, чтобы пользователь не забыл сохранить результаты:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin  {при закрытии формы}
if Res1.Caption='Сохранить результаты' then
            {если поиск закончился}
if Application.MessageBox('Сохранить результаты
        поиска?','',mb_OKCANCEL) = idOK
then Savehtmlfile;
end;
```

Справку мы, как я уже говорил, сделаем в *главе 16* (вместе с другими справками), а здесь я только отмечу, что программа у нас получилась ничуть не хуже фирменных, особенно если довести до ума некоторые детали, до которых здесь просто руки не дошли. На некоторые я обращал внимание по ходу дела, но вот еще, например: при поиске программа найдет не только файлы с отдельными словами, но и все те, в которых заданное слово содержится в составе более длинных слов. Так что если вы зададите, скажем, поиск фразы "про Delphi" с условием "любое из слов", то получите фантастическое количество мусора (сочетание "про" статистически самое распространенное из трехбуквенных сочетаний). И однозначно решить эту проблему нельзя: в разных случаях может потребоваться как поиск отдельного слова (режим "только слово целиком"), так и поиск фрагмента в составе более сложных слов. В плане интеллектуальных способностей программу можно совершенствовать до бесконечности, главное только — не переборщить.

ГЛАВА 15

Вася, посмотри, какая женщина!

Доделываем SlideShow

> Шуруп, забитый молотком, держится лучше, чем гвоздь, закрученный отверткой.
>
> *Народная примета*

Настало время нам довести до ума наш самый первый проект — SlideShow. Точнее, доделать функциональную часть — потому что в дальнейшем мы еще обязаны "прикрутить" к нему как минимум справку. Так как, несмотря на довольно навороченный код (а то ли еще будет!), у нас эта программа все равно останется в одном-единственном файле (кроме справки, которую все равно лучше располагать всегда отдельно) и не требует ровным счетом никаких системных установок[1], то возиться с инсталлятором здесь просто ни к чему. И доведя в этой главе до некоторого логического конца функциональность SlideShow, а в следующей — расправившись со справкой, мы на этом с ней закончим. Я решил посвятить доработке SlideShow специальную главу, т. к., чтобы программой было удобно пользоваться, доделывать придется довольно много, и при этом все равно мы ее по большому счету не доделаем. Хотя некоторые приемы вы здесь встретите впервые, большинство операций вам уже знакомы по предыдущим главам — если бы мы вводили их здесь по ходу доработки, то эта глава растянулась бы на полкниги и освоить ее было бы значительно труднее.

Итак, приступим: перенесем наш последний вариант из *главы 12* (папка Glava12/1) в новую папку (Glava15 — я специально не делаю подпапок, т. к. вариант будет всего один) и дадим ему номер версии 2.00 — переделки будут достаточно капитальные.

---

[1] Единственное, что мы могли бы всерьез предложить — ассоциировать файлы-картинки с нашей программой. Но такая опция никому не нужна, даже на выбор — картинки обычно ассоциированы с каким-нибудь редактором.

Составим список этих переделок, дополнив тот, что мы приводили в *главе 2*:

- поиск файлов не только с расширением jpg, но и jpeg, а также, по возможности, и других форматов;
- сортировка имен файлов по алфавиту;
- отображение имени файла и размера изображения;
- возможность просмотра картинок в виде набора "превьюшек" (preview);
- обеспечение операций Drag&Drop и загрузки файлов из командной строки;
- возможность распахнуть окно во весь экран без рамки;
- возможность запустить музыкальное сопровождение.

Что касается "других форматов", то штатно Delphi позволит нам иметь дело еще только с BMP. Для чтения TIFF, GIF и PNG (а по большому счету этим вполне можно ограничиться) придется устанавливать специальные библиотеки (см., например, [26]) или пользоваться возможностями технологий ActiveX./COM/OLE и т. п. Мы здесь этим заниматься не будем.

Для того чтобы решить первые четыре задачи, мы пойдем таким путем: будем сразу составлять список всех файлов с картинками, которые содержатся в текущей папке. Это облегчит нам процедуры их перелистывания и отображения в виде "превьюшек". Кроме того, мы сможем отображать их общее количество и номер текущей картинки. Заодно в таком списке легко обеспечить сортировку по алфавиту. Список мы будем составлять неоптимальным на первый взгляд способом — сначала оценим общее количество файлов JPG, JPEG и BMP в папке (чтобы можно было установить ползунок), а потом уточним его, пытаясь загрузить каждый файл в соответствующий объект (`TBitmap` или `TJpegImage`). Данная процедура относительно длительная — ведь каждый файл нужно прочесть с диска, хотя это и происходит много быстрее, чем в нашем побайтном чтении текстовых файлов в *главе 13* и *14*. Но она оправдывается — так мы точно определим, является ли файл корректной картинкой и будем уверены, что список наш правильный.

Учитывая то, что в дальнейшем нам придется опять читать файлы при их отображении (а, возможно, и не один раз — если еще и показывать их в виде "превьюшек"), возникают вопросы: нельзя ли либо определять "валидность" файла иным, более быстрым способом, либо, раз уж мы один раз прочитали, загрузить заодно при этом картинку в какой-либо контейнер в памяти и потом только отображать ее? Ответ на первый вопрос в принципе положительный — можно читать форматы напрямую, согласно их спецификации, и это будет, возможно, несколько быстрее, чем делать то же самое через посредников в виде объектов. Но гипотетическое увеличение скорости здесь не оправдывает многократно возрастающую сложность программного кода — чем

сложнее, тем больше можно наделать ошибок. Что же касается второго предложения — например, было бы заманчиво загружать картинки в какой-нибудь DrawGrid, и тем самым сразу формировать "превьюшки", не так ли? Но предварительное чтение и создание некоей базы картинок может привести к ускорению разве только при десятке-другом файлов (а такое количество будет читаться достаточно быстро и без предварительной загрузки, в любом случае), потом оперативная память переполнится, и в результате вместо ускорения мы получим только "тормоза" для всей системы в целом.

Обосновав таким образом наш подход, приступим к практической его реализации. Отметим, что составлять список нам нужно в четырех случаях: при загрузке конкретного файла (пункт **Файл | Открыть**), при смене папки (**Файл | Перейти к папке**), а также при операциях Drag&Drop и загрузке из командной строки (что то же самое, при "бросании" файла с картинкой на иконку программы в Проводнике, как мы знаем из *главы 13*). Во втором случае у нас конкретного файла нет, и мы будем загружать первый по списку, а в остальных будем загружать конкретный файл, а потом определять его номер.

Процедура составления списка файлов с картинками

Удалим кнопку Button1 и расставим на форме следующие компоненты: вверху, пониже меню, растянем во всю длину панели компонент StaticText (StaticText1) — он находится на вкладке **Addtional** и всем напоминает Label, но дополнительно может быть с рельефной рамкой. У него установим свойство BorderStyle в sbsSunken, свойство AutoSize в False. Он будет служить для отображения текущей папки. Теперь скопируем в буфер обмена этот компонент и размножим его внизу формы четыре раза (разумеется, каждый раз подгоняя размеры), получив компоненты, которые назовем StaticTextName (имя файла), StaticTextSize (размеры картинки), StaticTextN (номер картинки в списке) и StaticTextNFile (всего картинок в списке). У этих компонентов установим свойство Alignment в taCenter (выравнивание текста по центру). Левее последнего поставим обычный Label с надписью **Всего**, а левее предпоследнего — также Label с надписью **Рис. №**.

Теперь решим проблему с красивыми кнопками. Поместим на форму две кнопки типа BtnBtn (BtnBtnL и BtnBtnR) — они будут служить для перелистывания картинок по списку вправо и влево. Им хочется задать картинки в виде стрелочек, но по какой-то причине в Delphi 7 нет типовой библиотеки BitMap для кнопок (возможно, она поставляется отдельно — просто не знаю). Я заимствовал стрелочки из библиотеки Delphi 3, и расположил их в папке с про-

ектом под названиями arrL.bmp и arrR.bmp. Кнопки расположим справа внизу, очистим в них заголовки и через свойство Glyph загрузим указанные BitMap в виде стрелочек. Сразу перенесем текст из обработчика несуществующей теперь кнопки Button1 в обработчик щелчка на кнопке BtnBtnR. Наконец, самой последней поставим обычную кнопку под названием ButtonP с надписью **Предпросмотр** (вообще-то можно было и переименовать Button1, но так надежнее) — она у нас будет служить для запуска окна "превьюшек".

Эти компоненты надо "заякорить", чтобы они не разбежались при распахивании окна. У всех нижних StaticText и Label в свойстве Anchors в True должны быть установлены пункты akLeft и akBottom, а у кнопок — пункты akRight и akBottom. Поверх же верхнего StaticText1 мы установим компонент ProgressBar (ProgressBar1), причем растянем его так, чтобы он полностью вписался в StaticText1. Имейте в виду, что StaticText, в отличие от Label, имеет собственное окно и может служить "родителем", что значительно облегчает дело. У ProgressBar1 установим свойство Step в 1, а свойство Visible в False. После всей этой возни главная форма на этапе конструирования должна приобрести вид, показанный на рис. 15.1.

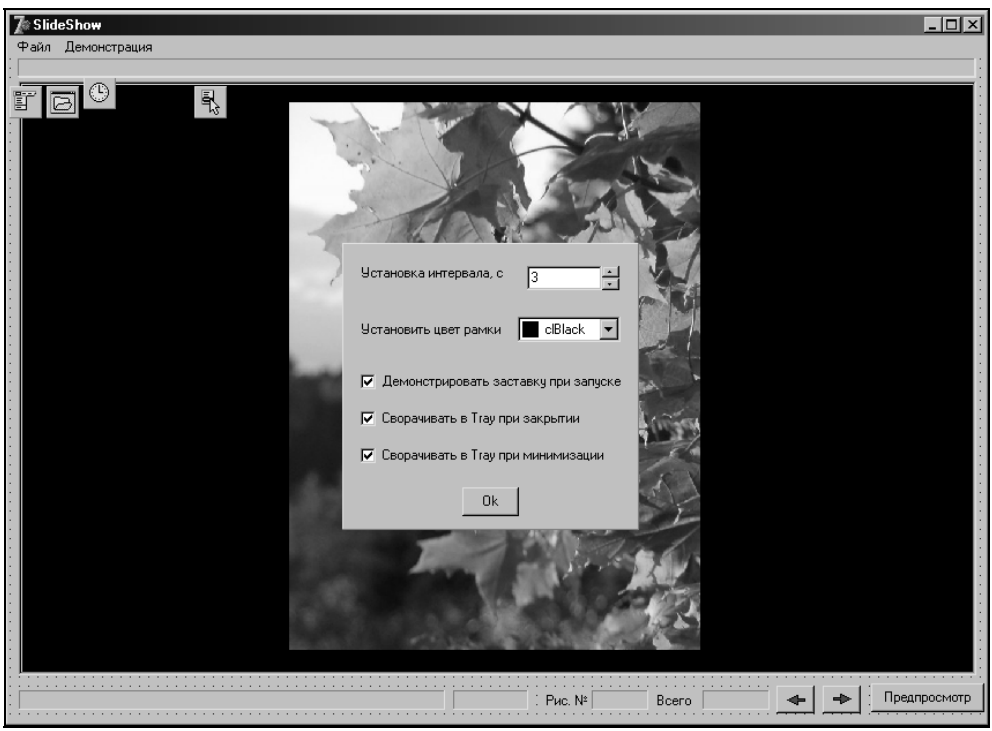


Рис. 15.1. Главная форма SlideShow

Теперь, наконец, возьмемся за алгоритм. Для начала отредактируем в OpenDialog1 фильтр (свойство Filter), теперь он должен выглядеть, как Картинки JPEG и BMP|*.jpg; *.jpeg; *.bmp. Внесем в тексте модуля Slide следующие переменные в секцию **public**:

```
. . . . .
Filelist,SizeList :TStringList; {объекты типа StringList
 для хранения имен и размера файлов}
JpgIm: TJpegImage; {объект типа JPEG}
BmpIm: TBitMap; {объект типа BitMap}
. . . . .
```

А в секции **var** добавим такие:

```
. . . . .
nfiles: integer;
fname:string;
buffer : array[0..255] of char,
. . . . .
```

Будем иметь в виду, что переменная n, как признак загрузки *(см. главу 2)*, нам больше не понадобится, но мы ее потом используем, как счетчик картинок.

В процедуру FormCreate добавим в самом конце такие строки:

```
. . . . .
Filelist:=TStringList.Create; {создаем экземпляр списка имен}
Filelist.Sorted:=False; {пока сортировать не будем}
SizeList:=TStringList.Create; {создаем экземпляр списка размеров}
Sizelist.Sorted:=False; {здесь сортировка не нужна}
DragAcceptFiles(Form1.Handle,True); {сообщаем, что готовы
                  к приему файлов}
. . . . .
```

Самая последняя строка — заготовка для последующей реализации Drag&Drop.

Создадим обработчик события onDestroy формы, в котором будем уничтожать наши списки:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
FileList.Destroy; {уничтожаем списки}
Sizelist.Destroy;
end;
```

Наконец, напишем функцию для предварительного пересчета файлов, расположив ее в самом верху текста модуля Slide, сразу под процедурой CreateMyicon:

```
function NumFiles:boolean;
begin  {пересчет файлов с картинками в заданном каталоге}
result:=True;
nfiles:=0;
if FindFirst('*.*',faAnyFile,srf)=0 then
begin {просматриваем все файлы и пересчитываем те, что jpg и bmp}
 repeat
  if (ExtractFileExt(AnsiLowerCase(srf.Name))='.bmp') or
    (ExtractFileExt(AnsiLowerCase(srf.Name))='.jpg')
  or (ExtractFileExt(AnsiLowerCase(srf.Name))='.jpeg') then
  nfiles:=nfiles+1;
 until (FindNext(srf)<>0);
end;
FindClose(srf);
if nfiles=0 then result:=False else
Form1.ProgressBar1.Max:=nfiles;
        {если нашли, устанавливаем ползунок}
end;
```

А ниже ее расположим основную процедуру, которую назовем PictureList, для выявления файлов с картинками и размещения их имен и размеров в списке. В этой процедуре мы сначала располагаем имя и размер в одной и той же строке в списке Filelist, который сортируется по мере создания, а потом запрещаем сортировку и делим полученные строки на два списка: FileList с именами и Sizelist с размерами — способ не очень красивый, но зато позволяет надежно привязать имя к размеру, иначе после сортировки установить соответствие было бы весьма сложно[2]. По ходу дела мы будем также определять и выводить размер уже загруженного файла (если имеется его имя), а в самом начале процедуры устанавливать папку по умолчанию для всех диалогов, выводить ее в INI-файл, а также очищать окна для последующего вывода информации. В результате процедура получится довольно навороченной:

```
procedure PictureList; {составление списка файлов с картинками}
var st:string;
begin
if not NumFiles then exit; {если ни одного, то выход}
Form3.DirectoryListBox1.Directory:=stpath;
            {на следующее открытие папки}
```

[2] Очевидный способ — вставлять в список SizeList строку с размерами методом Insert — не поможет, так как список FileList все время сортируется. На самом деле из этой ситуации в принципе можно вывернуться, но процедура будет достаточно сложной, а мы будем делать, как проще.

```
with Form1 do
begin
  OpenDialog1.InitialDir:=stPath; {на следующее открытие файла}
  ChDir(stpath); {устанавливаем текущую папку}
  IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini'));
                          {открываем ini}
  IniFile.WriteString('Main','Path',stpath);
  IniFile.Destroy;
  StaticText1.Caption:=stPath;  {выводим путь и имя}
  if fname<>'' then
  StaticTextName.Caption:=ExtractFileName(fname);
  StaticTextSize.Caption:=''; {очистили размер}
  StaticTextN.Caption:=''; {очистили номер}
  StaticTextNFile.Caption:=''; {очистили количество}
  Application.ProcessMessages; {показали, что загрузили}
 ProgressBar1.Visible:=True; {показали ползунок}
 ProgressBar1.Position:=0; {установили в 0}
 FileList.Clear; {очистили список}
 Filelist.Sorted:=True;
         {это чтобы файлы сортировались по алфавиту}
 JpgIm := TJpegImage.Create; {создаем JPEG}
 BmpIm := TBitMap.Create; {создаем BitMap}
 if FindFirst('*.*',faAnyFile,srf)=0 then
 begin {просматриваем все файлы, и проверяем те, что jpg и bmp}
 repeat
  if (ExtractFileExt(AnsiLowerCase(srf.Name))='.bmp') then
  begin
     try
      BmpIm.LoadFromFile(srf.Name);
             {пробуем загрузить картинку с диска}
     except {если ничего не вышло - это вовсе не BitMap}
      continue; {продолжаем цикл}
     end;
     Filelist.Add(srf.Name+'*'+IntToStr(BmpIm.Width)+
           'x'+IntToStr(BmpIm.Height));
     {добавляем имя найденного файла и его размеры к списку}
     if fname<>'' then
     if srf.Name = ExtractFileName(fname) then
     begin
      StaticTextSize.Caption:=IntToStr(BmpIm.Width)+
              'x'+IntToStr(BmpIm.Height);
      Application.ProcessMessages;
        {показываем размеры открытого файла}
     end;
```

```
    ProgressBar1.StepIt;
 end;
 if (ExtractFileExt(AnsiLowerCase(srf.Name))='.jpg') or
   (ExtractFileExt(AnsiLowerCase(srf.Name))='.jpeg') then
 begin
    try
     JpgIm.LoadFromFile(srf.Name);
          {пробуем загрузить картинку с диска}
    except {если ничего не вышло - это вовсе не JPEG}
     continue; {продолжаем цикл}
    end;
    Filelist.Add(srf.Name+'*'+IntToStr(JpgIm.Width)+
          'x'+IntToStr(JpgIm.Height));
    {добавляем имя найденного файла и его размеры к списку}
    if fname<>'' then
    if srf.Name = ExtractFileName(fname) then
    begin
     StaticTextSize.Caption:=IntToStr(JpgIm.Width)+
              'x'+IntToStr(JpgIm.Height);
     Application.ProcessMessages;
       {показываем размеры открытого файла}
    end;
    ProgressBar1.StepIt;
 end;
until (FindNext(srf)<>0);
end;
BmpIm.Destroy; {уничтожаем BitMap}
JpgIm.Destroy; {уничтожаем JPEG}

Filelist.Sorted:=False; {больше сортировать не будем}

for nfiles:=0 to Filelist.Count-1 do {делим имя и размер}
begin
 st:=Filelist[nfiles];
 Filelist[nfiles]:=copy(st,1,pos('*',st)-1);
 SizeList.Add(copy(st,pos('*',st)+1,length(st)));
end; {теперь в Filelist - имена, а в SizeList - размеры}

n:=0;
for nfiles:=0 to Filelist.Count-1 do
{показываем номер открытого файла - до этого он не известен}
if Filelist[nfiles] = ExtractFileName(fname) then
begin StaticTextN.Caption:=IntToStr(nfiles+1);
   n:=nfiles; break end;
{если файл открыт, то в n - его номер в списке, иначе n=0}
```

```
 nfiles:=Filelist.Count;
 StaticTextNFile.Caption:=IntToStr(nfiles);
          {показываем сколько файлов}
 ProgressBar1.Visible:=False; {убрали ползунок}
end;
end {PictureList};
```

Теперь нам нужно внести эту процедуру в четыре места, по числу перечисленных ранее возможных действий: открыть файл, открыть папку, загрузить файл через Drag&Drop и загрузить файл через командную строку. Первые два действия у нас уже реализованы. Внесем в конец процедуры `OpenClick` изменения: удалим оператор `n:=0;` и добавим формирование имени файла и папки, а также вызов нашей процедуры:

```
procedure TForm1.OpenClick(Sender: TObject); {открытие файла}
begin
Panel2.Visible:=False; {закрываем панель}
If OpenDialog1.Execute then
       {если диалог открытия файла завершился удачно}
Image1.Picture.LoadFromFile(OpenDialog1.FileName)
              {загружаем картинку в Image1}
else exit; {иначе, если диалог завершился неудачно,
               то выходим из процедуры}
 fname:=OpenDialog1.FileName; {запоминаем имя загруженного файла}
 stpath:=ExtractFileDir(fname);
             {запоминаем папку без последнего "\"}
PictureList; {составление списка файлов с картинками}
end;
```

Теперь внесем изменения в вызов диалога установки папки. В тексте модуля Papka удалим из процедуры `Button3Click` все, что относится к INI-файлу и к установке директории по умолчанию, в результате процедура упростится:

```
procedure TForm3.Button3Click(Sender: TObject);
begin {Ok для выбора папки}
 stpath:=DirectoryListBox1.GetItemPath
          (DirectoryListBox1.ItemIndex);
 ChDir(stpath); {устанавливаем текущую папку}
 Form3.Close;
end;
```

А процедура `OpenFolderClick` в модуле Slide, в которой стоял единственный вызов `Form3.ShowModal`, наоборот, усложнится:

```
procedure TForm1.OpenFolderClick(Sender: TObject);
begin  {открыть папку}
```

```
 Form3.ShowModal;
 fname:=''; {пустое имя}
 PictureList; {составление списка файлов с картинками}
 if Filelist.Count=0 then exit;
 StaticText1.Caption:=stPath;
 {загружаем первую картинку по списку:}
 StaticTextN.Caption:='1';
 fname:=stpath+'\'+Filelist[0];
 StaticTextName.Caption:=ExtractFileName(fname);
 Image1.Picture.LoadFromFile(fname);
 StaticTextSize.Caption:=Sizelist[0];
end;
```

Теперь остальные действия. Процедуру по загрузке файла в командной строке (она же при "бросании" файла с картинкой на иконку программы в Проводнике) мы реализуем через обработчик события onActivate формы:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
if paramcount<>0 then {если есть параметр загрузки}
begin
 try
  Image1.Picture.LoadFromFile(paramstr(1));
  {пробуем загрузить картинку}
 except
  exit;
 end;
 fname:=paramstr(1);
 stpath:=ExtractFileDir(fname);
        {запоминаем папку без последнего "\"}
 PictureList; {составление списка файлов с картинками}
end;
end;
```

Наконец, реализуем Drag&Drop, для чего внесем в секцию **private** нашу процедуру из *главы 13*:

```
procedure DropPicture(var Message: TWMDROPFILES); message WM_DROPFILES;
```

Сама процедура будет выглядеть так:

```
procedure TForm1.DropPicture(var Message: TWMDROPFILES);
begin {прием файлов, брошенных на форму}
DragQueryFile(Message.Drop,0,@buffer,sizeof(buffer));
try
 Image1.Picture.LoadFromFile(buffer); {пробуем загрузить}
```

```
except
  exit;
end;
fname:=buffer;
stpath:=ExtractFileDir(fname);
        {запоминаем папку без последнего "\"}
PictureList; {составление списка файлов с картинками}
end;
```

Вот, теперь можно запускать и проверять в разных режимах. Осталось реализовать две вещи: собственно демонстрацию, которая у нас по-прежнему основана на старой процедуре `LoadFile` (которая теперь к тому же не будет работать из-за того, что мы не устанавливаем, как надо, переменную `n`), и, конечно, режим загрузки "превьюшек".

Демонстрация картинок по списку

Процедура демонстрации у нас теперь упростится. Мы используем переменную `n` в новом качестве — как счетчик картинок, и вот как будет тогда выглядеть модернизированная `Loadfile`:

```
procedure Loadfile; {процедура поиска и загрузки файлов JPEG}
begin
with Form1 do
begin
 StaticTextN.Caption:=IntToStr(n+1);
 StaticTextName.Caption:=Filelist[n]; {имя из списка}
 fname:=stpath+'\'+Filelist[n];
 Image1.Picture.LoadFromFile(fname); {загружаем}
 StaticTextSize.Caption:=Sizelist[n]; {размер из списка}
end;
end;
```

Посмотрев на текст процедуры `OpenFolderClick` ранее, мы поймем, что там, в сущности, делается то же самое. Поэтому логично заменить в ней все, что после комментария "`загружаем первую картинку по списку`" на две строки:

```
. . . . .
n:=0;
Loadfile;
. . . . .
```

Зато несколько усложнятся процедуры по ручной и автоматической демонстрации картинок. Создадим обработчик щелчка на кнопке `BitBtnL`, который

у нас до сих пор отсутствовал, и тогда три обработчика (вправо, влево и по таймеру) будут выглядеть так:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
if Filelist.Count=0 then exit; {если нет картинок - выходим}
n:=n+1;
if n>Filelist.Count-1 then n:=0; {вкруговую}
 {загружаем следующую картинку по списку:}
Loadfile;
end;

procedure TForm1.BitBtnRClick(Sender: TObject);
begin {вправо - к концу списка}
Panel2.Visible:=False; {закрываем панель}
if Filelist.Count=0 then exit; {если нет картинок - выходим}
n:=n+1;
if n>Filelist.Count-1 then n:=0; {вкруговую}
 {загружаем следующую картинку по списку:}
Loadfile;
end;

procedure TForm1.BitBtnLClick(Sender: TObject);
begin {влево - к началу списка}
Panel2.Visible:=False; {закрываем панель}
if Filelist.Count=0 then exit; {если нет картинок - выходим}
if n=0 then n:=Filelist.Count-1 else n:=n-1;{вкруговую}
  {загружаем предыдущую картинку по списку:}
Loadfile;
end;
```

Правильно ли мы тут все так упростили, просто пуская список вкруговую? Не надо ли при достижении конца списка (и начала — при обратном перелистывании) выводить предупреждение? Не думаю, на мой вкус, это излишне, ведь номер картинки у нас выводится. А вот предусмотреть возможность запуска автоматической демонстрации по таймеру не только вперед, но и назад по списку — вполне можно, но я оставляю это на откуп читателям (надо в пункте меню **Демонстрация** создать еще два подпункта **Вперед** и **Назад**, и переключать между ними, используя их свойство Checked, а также запоминать их состояние в INI-файле — как видите, довольно громоздко получается).

Сделаем еще одну вещь. Добавим пункт главного меню под названием Screen1 и с заголовком **Во весь экран**. Горячую клавишу ему объявлять не будем — позже мы ее сделаем искусственно. По нему мы будем распахивать окно программы во весь экран и убирать рамку и заголовок формы. Сделаем

так, чтобы при первом нажатии окно распахивалось, а при втором — рамка с заголовком появлялись обратно. Для события щелчка на этом пункте меню напишем такую процедуру:

```
procedure TForm1.Screen1Click(Sender: TObject);
begin  {во весь экран}
 if Form1.Filelist.Count=0 then exit;
      {если в списке ничего нет - выход}
 Panel2.Visible:=False; {закрываем панель настроек}
 if BorderStyle=bsSizeable then
 begin
  BorderStyle := bsNone;
  FormStyle  := fsStayOnTop;
  Left    := 0;
  Top     := 0;
  Height:= Screen.Height;
  Width := Screen.Width;
 end else
 begin
  BorderStyle := bsSizeable;
  FormStyle  := fsNormal;
  Height:= 620;
  Width := 850;
  Position:=poDesktopCenter;
 end;
end;.
```

Сделать невидимым также и главное меню сложнее, но если хотите, то можете по очереди сделать невидимыми все его пункты.

Теперь сделаем все то же самое, но с клавиатуры. Для этого мы используем событие, которое еще ни разу не использовали, только упоминали о нем в *главе 9* — `onShortCut` формы. Оно, в отличие от стандартных `onKeyPress` и `onKeyDown`, позволяет распознать код любой нажатой клавиши, включая клавиши управления, и действует независимо от установки параметра `KeyPreview`. Чтобы перехватить нажатие, надо для нужного виртуального кода установить внутри процедуры параметр `Handled` в `True`, иначе будут выполняться всякие лишние действия. Мы сделаем так, чтобы по клавише "Стрелка влево" список перелистывался к началу, по "Стрелке вправо" и "Пробелу" — к концу, а по нажатию <Esc> происходило бы то же самое, что в пункте меню **Во весь экран** (<Esc> и будет для него горячей клавишей):

```
procedure TForm1.FormShortCut(var Msg: TWMKey; var Handled: Boolean);
{перелистывание с клавиатуры}
```

```
begin
 if Form1.Filelist.Count=0 then exit; {если в списке ничего нет}
 if (Msg.CharCode=vk_Right) or (Msg.CharCode=vk_Space) then
 {нажата клавиша "Стрелка вправо" или пробел}
 begin
  Handled:=True;
   {перехват управления - иначе событие нажатия дублируется}
  Panel2.Visible:=False; {закрываем панель}
  BitBtnR.SetFocus;
  if Filelist.Count=0 then exit; {если нет картинок - выходим}
  n:=n+1;
  if n>Filelist.Count-1 then n:=0; {вкруговую}
 {загружаем следующую картинку по списку:}
  Loadfile;
 end;
 if Msg.CharCode=vk_Left then {нажата клавиша "Стрелка влево"}
 begin
  Handled:=True;
   {перехват управления - иначе событие нажатия дублируется}
  Panel2.Visible:=False; {закрываем панель}
  BitBtnL.SetFocus;
  if Filelist.Count=0 then exit; {если нет картинок - выходим}
  if n=0 then n:=Filelist.Count-1 else n:=n-1;{вкруговую}
  {загружаем предыдущую картинку по списку:}
  Loadfile;
 end;
 if Msg.CharCode=vk_Escape then {нажата клавиша Esc}
 begin  {во весь экран}
  Handled:=True;
   {перехват управления - иначе событие нажатия дублируется}
  Panel2.Visible:=False; {закрываем панель}
  if BorderStyle=bsSizeable then
  begin
   BorderStyle := bsNone;
   FormStyle  := fsStayOnTop;
   Left    := 0;
   Top     := 0;
   Height:= Screen.Height;
   Width := Screen.Width;
  end else
  begin
   BorderStyle := bsSizeable;
   FormStyle  := fsNormal;
```

```
   Height:= 620;
   Width := 850;
   Position:=poDesktopCenter;
  end;
 end;
end;
```

Для того чтобы пользователь, не читая справки, узнал о существовании горячих клавиш, установим свойства `ShowHint` для обеих кнопок `BitBtnR` и `BitBtnL` в `True`, и запишем в свойстве `Hint` первой из них `Стрелка вправо`, `Пробел`, а второй — просто `Стрелка влево`.

И наконец, "до кучи" сделаем все то же по щелчку мышью на компоненте `Image1` — подобно тому, как это делается в фотогалереях в Интернете (причем автоматически то же самое будет происходить и при щелчке на панели или на самой форме, ведь мы делали для них общий обработчик, см. *главу 9*). Однако не будем повторять грубую концептуальную ошибку разработчиков Windows, в которой при простой смене фокуса событие нажатия клавиши мыши не теряется, как должно было бы быть, а производит действия, как будто окно было активировано. То есть одновременно как бы происходят два события вместо одного и попасть мышью так, чтобы при этом что-нибудь не открылось, очень сложно (приятным исключением является окно **Найти и Заменить** в Word). У нас по щелчку мышью на `Image1` уже одно действие происходит — закрытие панели настроек, и если при всех других действиях оно совмещено с иной функциональностью специально, то перелистывать картинки одновременно с закрытием панели настроек совершенно ни к чему (в следующей главе мы заменим панель настроек на другой способ их отображения и эта проблема отпадет сама собой). В результате обработчик получится такой:

```
procedure TForm1.Image1Click(Sender: TObject);
begin {перелистывание мышью}
if Panel2.Visible=True then Panel2.Visible:=False
{закрываем панель установки интервала}
else
begin
 if Filelist.Count=0 then exit; {если нет картинок - выходим}
 n:=n+1;
 if n>Filelist.Count-1 then n:=0; {вкруговую}
  {загружаем следующую картинку по списку:}
 Loadfile;
end;
end;
```

Музыка без медиаплеера

Неплохо бы сопровождать показ слайдов тихой музыкой, да? Но т. к. мы не делаем тут программу-проигрыватель (вот их, безусловно, не меньше миллиона итак имеется на все вкусы), то нельзя ли обойтись минимальными средствами, без всяких медиаплееров и ActiveX? Запросто, и для этого есть API-функция `PlaySound`, которая делает все замечательно, кроме одного — она умеет проигрывать только WAV-файлы. Но мы ведь хотели обойтись минимальными средствами... Так что переживем — превратить любой файл в WAV при надобности очень легко, для этого есть множество программ. А так — можно использовать, например, файлы рингтонов для мобильников. Вопрос не в файлах и не в способе их воспроизведения, а в том, чтобы сделать все это удобным для пользования. И как мы сейчас увидим, осуществить это с "полтычка" не получится.

Два музыкальных клипа с WAV-файлами для отработки программы я расположил в отдельной папке внутри папки с проектом (Glava15\Music). Ограничимся тем, что будем воспроизводить музыку лишь во время автоматической демонстрации. В предложение **uses** модуля slide.pas добавим модуль mmSystem и введем две новых переменных:

```
. . . . .
fmusic:string='';
FlagMusic:byte=0;
. . . . .
```

В строке `fmusic` мы будем хранить имя файла с клипом, а флаг `FlagMusic` нам понадобится для того, чтобы запускать функцию `PlaySound` только один раз, иначе плавного воспроизведения нам не добиться. Теперь добавим в меню **Демонстрация** после пункта **Запуск** пустой разделительный пункт (знак "-" в `Caption`), а после него два пункта: **Музыкальное сопровождение** (`Music1`, <Ctrl>+<M>) и **Выбрать клип** (`BrowseMusic`, <Ctrl>+<K>). На форму поставим еще один диалог открытия файла (`OpenDialog2`), и в свойстве `Filter` у него запишем `Файлы WAV|*.wav`.

В процедуру по таймеру (`Timer1Timer`) допишем такой фрагмент:

```
. . . . .
if fmusic<>'' then {если какое-то имя файла есть}
if Music1.Checked=True then
begin
 if FlagMusic=0 then {если флаг =0}
 begin
  if not PlaySound(PChar(fmusic),0,SND_NOSTOP+SND_ASYNC)
```

```
  then Application.MessageBox
  (' Устройство занято или неправильный формат','Ошибка',MB_OK);
  FlagMusic:=1; {в следующий раз не воспроизводим}
 end;
end;
. . . . .
```

Здесь, если у нас имя файла с клипом не пустое, и при этом пункт **Музыкальное сопровождение** отмечен (на самом деле это перекрывающиеся требования — мы в дальнейшем сделаем так, чтобы при пустом имени файла нельзя было бы отметить `Music1`, но хуже не будет), мы при первом событии таймера вызываем функцию `PlaySound`. С первым ее параметром все ясно, равенство второго нулю означает, что звук берется не из ресурса, а из внешнего файла, а набор констант в третьем (`SND_NOSTOP+SND_ASYNC`) означает, что звук будет воспроизводиться только, если ничего не воспроизводится другого (не будет прерывать уже запущенное воспроизведение) и при этом функция сразу передаст управление системе — т. е. звучать будет до тех пор (при необходимости вкруговую), пока мы все это дело не остановим. Если звук включен, то при следующем событии таймера обращения к `PlaySound` не будет (`FlagMusic` устанавливается в 1), иначе будут заметные щелчки. Сообщение "Устройство занято..." на самом деле возникнет только, если функция не может проиграть файл по причине занятости устройства, а при неправильном имени файла почему-то значение `False` не возвращается, но это не очень важно — мы и не будем вводить имя вручную.

Теперь в первую очередь наладим выключение. В процедуру `RunClick` добавим при остановке демонстрации следующее:

```
. . . . .
end else {иначе, если демонстрация уже идет}
begin
 Run.Caption:='Запуск';
    {меняем название пункта меню обратно на Запуск}
 Timer1.Enabled:=False; {таймер остановлен}
 PlaySound(Pchar(fmusic),0, SND_PURGE); {выключили музыку}
 FlagMusic:=0;{в следующий раз опять запустим}
end;
. . . . .
```

Константа `SND_PURGE` и означает остановку звучания. Теперь надо наладить обращение к пунктам меню, в первую очередь загрузку имени файла:

```
procedure TForm1.BrowseMusicClick(Sender: TObject);
var st:string;
```

```
begin {пункт меню Выбрать клип}
If OpenDialog2.Execute then
begin
 fmusic:=OpenDialog2.FileName;
 FlagMusic:=0; {будем играть заново}
 PlaySound(Pchar(fmusic),0, SND_PURGE);
end
else {ничего не нашли}
begin
 fmusic:=''; {если не нашли файл, то очистили имя}
 Music1.Checked:=False; {остановили музыку}
 PlaySound(Pchar(fmusic),0, SND_PURGE);
end;
st:=StaticText1.Caption;
if (st='') or ((st[1]='К') and (st<>'')) then
{т. е. или пусто, или надпись "Клип" по-русски}
StaticText1.Caption:='Клип: '+fmusic;
end;
```

Здесь (начнем рассмотрение снизу) у нас имя загруженного файла для контроля выведется в строку StaticText1, где обычно располагается имя папки с картинками — но только в том случае, если она пустая или там было отображено именно имя клипа, т. е. если мы загружаем его в самом начале после запуска программы. Если там уже есть имя папки, то клипом придется пожертвовать. Если мы в диалоге нажали кнопку **Отмена**, то имя очистится, и снимается отметка с пункта меню Music1. При этом также останавливается воспроизведение, если оно было запущено. И, наконец, при успешном завершении диалога мы останавливаем воспроизведение, и в следующий раз оно начнется заново только при условии, что пункт меню Music1 отмечен — это надо сделать отдельно. К чему мы сейчас и перейдем, вот какой обработчик обращения к пункту Music1 мы сделаем:

```
procedure TForm1.Music1Click(Sender: TObject);
begin {пункт меню Музыка}
if fmusic='' then exit;
if Music1.Checked=False
then
begin {подготавливаем клип к запуску}
 Music1.Checked:=True;
 FlagMusic:=0;
end
else
begin {иначе выключаем}
 Music1.Checked:=False;
```

```
 PlaySound(Pchar(fmusic),0, SND_PURGE);
end;
end;
```

Здесь все, кажется, понятно — по первому обращению пункт становится отмеченным и взводится флаг `FlagMusic`, по второму — отметка снимается, а звучание останавливается (с флагом при этом ничего делать не надо). Резюмируем логику работы: музыка играет тогда, когда мы установили флажок в пункте меню **Музыкальное сопровождение** (нажали <Ctrl>+<M>), если предварительно было загружено имя музыкального клипа. Если нажать повторно, музыка остановится. Если в процессе загрузить другой клип, то первый прервется и начнется второй. Можно ничего не загружать, если нажать в диалоге открытия файла кнопку **Отмена**, в этом случае имя файла окажется пустым.

Манипулируя с диалогом `OpenDialog2`, мы не должны забывать про `OpenDialog1` — ведь мы меняем текущую папку (именно на этот случай я в списке файлов с изображениями сохранял полное имя файла с путем к нему). Поэтому позаботимся о том, чтобы устанавливать для `OpenDialog1` начальную папку принудительно (для окна открытия папки этого делать не надо). В процедуре `OpenClick` добавим в самом начале оператор:

```
OpenDialog1.InitialDir:=stpath;
```

Все, что мы сейчас проделали, есть типичный пример функциональности, внести которую можно только вручную, никакие автоматизированные системы проектирования вам тут помочь не смогут. Надо держать в голове всю логику наших действий и реакцию программы, иначе легко наворотить такого, что программой вообще пользоваться будет нельзя. Вы заметили, например, что я здесь не выводил почти никаких сообщений (кроме редко возникающего "Устройство занято...") — не надо "грузить" пользователя лишней информацией. Если он не удосужится прочесть справку, то ничего страшного не произойдет, когда музыка вдруг не заиграет, это совершенно второстепенная функция. Но вот в справке все надо объяснить подробно.

Но это еще не все! Нам надо, конечно, все это запоминать — если заставлять пользователя загружать клип при каждом запуске, то можно было бы ничего и не затевать с музыкой вообще. Обратимся к нашей главной по этой части процедуре `Form1Create` и по условию **if** `SectionExists('Main')` добавим такие строки:

```
. . . . .
{музыка:}
   fmusic:=ReadString('Main','Musicfile','');
                    {по умолчанию пусто}
```

```
   Music1.Checked:=ReadBool('Main','Music',False);
   if fmusic<>'' then
   begin  {установки диалога для поиска клипа}
    StaticText1.Caption:='Клип: '+fmusic;
                {отображаем имя клипа}
    OpenDialog2.InitialDir:=ExtractFilePath(fmusic);
    OpenDialog2.FileName:=ExtractFileName(fmusic);
   end else {если имя клипа пусто}
    OpenDialog2.InitialDir:=ExtractFilePath(Application.ExeName);
. . . . .
```

То есть при пустом имени клипа мы будем искать, начиная с папки с программой, вряд ли в папке с картинками найдется клип. Далее мы создаем эти два пункта в первый раз (ниже по условию "если секция еще не создана"):

```
. . . . .
WriteString('Main','Musicfile',''); {по умолчанию пусто}
WriteBool('Main','Music',False);
. . . . .
```

А собственно запоминание мы осуществим при выходе из программы в имеющейся у нас уже процедуре FormDestroy:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
FileList.Destroy; {уничтожаем списки}
Sizelist.Destroy;
{запоминаем музыку:}
IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini'));
                    {открываем ini}
IniFile.WriteString('Main','Musicfile',fmusic); {имя файла клипа}
IniFile.WriteBool('Main','Music',Music1.Checked);
IniFile.Destroy;
end;
```

Заметки на полях

У читателя уже, несомненно, возник недоуменный вопрос — а зачем было возиться с запоминанием установок в разных местах для других параметров, нельзя ли их также было собрать на закрытие программы? Отвечаю: те установки были более важными с точки зрения основной функциональности. Если у вас при запущенной программе вдруг выключится компьютер, то все установки при следующем запуске восстановятся — все, кроме музыкальных, которые не успели сохраниться. И мы тут этим сознательно жертвуем, чтобы не усложнять программу, но когда дело доходит до вещей существенных, то следует всегда их сохранять в дисковом файле своевременно. Мне даже не очень понятно, для чего в Word, например, придумана такая сложная и довольно "глючная" систе-

ма восстановления файлов после сбоев, если можно просто своевременно закрывать дисковые файлы после каждого сохранения. Возможно, что я действительно чего-то недопонимаю, или, скорее всего, все было задумано "как надо", а получилось "как всегда". Даже если у вас текущее содержимое находится во временном файле (что разумно), его также можно держать закрытым, т. е. с заведомо сохранной информацией, и лишь периодически сбрасывать туда последние изменения из оперативной памяти. Тем более что есть прекрасный механизм безопасной двухступенчатой записи, который легко воспроизвести и самостоятельно. Заключается он в следующем: на первом шаге запись производится в некий временный файл (LOG), после чего файл, который действительно необходимо перезаписать, помечается, как находящийся в переходном состоянии. Затем производится запись в этот файл. И только после этого с файла снимается пометка о переходном состоянии. Если сбой произойдет в процессе первой перезаписи (LOG-файла), то все останется, как было. Если же в процессе второй, то после восстановления программа просто продолжит перезапись. Так устроена, например, запись в реестр Windows XP.

Вот теперь, кажется, все в порядке — если и есть какие-то несообразности, то мне они незаметны, и я предлагаю читателям самим протестировать программу окончательно. Не зря же корпорации даже нанимают тестировщиков за деньги, а наша программа уже вполне вышла на тот уровень, когда один человек отловить сразу все ошибки не в состоянии. Учитывая, особенно, что мы все еще ее не доделали, теперь займемся, наконец, "превьюшками".

Демонстрация "превьюшек"

Для демонстрации "превьюшек" мы создадим новую форму (`Form4`), и назовем модуль preview.pas. Вопрос состоит в том, куда именно выводить "превьюшки"? Если вы ожидаете, что я воспользуюсь чем-нибудь вроде `DrawGrid` (что было бы самым логичным), то ошибаетесь. У таблицы `DrawGrid`, как и у `StringGrid` (которую также можно использовать) слишком много, на мой вкус, "фичей", которые придется мужественно преодолевать без какой-либо в них надобности. Все, что нам требуется — расположить уменьшенные картинки рядами, вписавшись в ширину формы. И наиболее просто это сделать, используя канву и уже знакомый нам по *главе 10* метод `StretchDraw`. Единственная сложность, которая подстерегает нас на этом пути — вычислить номер картинки при щелчке на изображении ее "превьюшки", чтобы отобразить ее в основном окне. Но это, как увидите, совсем нетрудно — в сравнении с тем, какое количество параметров нам пришлось бы настраивать при использовании компонентов-таблиц и сколько при этом было бы возни с формированием уменьшенных изображений.

Итак, поставим на форму `Form4` компонент `ScrollBox` (`ScrollBox1`), растянем его и установим внутрь наш любимый `Image` (`Image1`). Вообще-то называть одинаково компоненты на двух формах одного приложения — плохой стиль,

слишком легко что-то перепутать, но мы постараемся избежать ошибок — Image1 выглядит привычно, а использовать мы его будем в основном в своих процедурах, где Delphi без уточнений, что чему принадлежит, все равно не обойдется. Ниже установим кнопку Button1 и запишем у нее в заголовке **Остановить**.

У ScrollBox1 установим для свойства HorScrollBox пункт Visible в False, а свойство Color в clSilver — пусть фон для "превьюшек" всегда будет серый. Чтобы Image1 имел тот же цвет, сразу создадим обработчик onCreate для формы Form4:

```
procedure TForm4.FormCreate(Sender: TObject);
begin
 Image1.Canvas.Brush.Color:=Form4.ScrollBox1.Color;
              {цвет фона картинок clSilver}
end;
```

А для самой формы установим Position в poDefault, FormStyle в fsStayOnTop, BorderStyle в bsDialog, а в Caption запишем просто SlideShow. После всех этих действий мы получим заготовку формы, показанную на рис. 15.2.

Рис. 15.2. Форма Preview программы SlideShow

В основном модуле (slide.pas) мы объявим дополнительные переменные: счетчик `ni:integer` и `FlagE:byte=0` (последний будет отвечать за режим остановки загрузки "превьюшек" по ходу процесса). Кроме этого туда же добавим еще один флаг — `MinFlag:boolean=True`. Чтобы из модуля preview.pas также были доступны переменные главного модуля, мы вставим ссылку на последний в главное предложение **uses** (а не после **implementation**, как это делает Delphi автоматически). Туда же добавим ссылку на модуль JPEG. А в slide.pas еще вынесем в интерфейсную секцию строку **procedure** `Loadfile` (рядом с `CreateMyicon`), она нам понадобится в preview.pas.

Теперь научимся запускать окно предпросмотра. Чтобы не думать о запутанной логике действий при минимизации в Tray, мы попросту будем запрещать все нестандартные разновидности минимизации, пока окно предпросмотра виднеется на экране. Именно для этого нам понадобился флаг `MinFlag`. Внесем такие изменения в тексте основного модуля: в процедуре `Form1Create` внутри операторных скобок по условию **if** `SectionExists('Sets')` добавим строку:

```
MinFlag:=CheckBox3.Checked; {запоминаем состояние минимизации}
```

То же самое допишем в процедуре окончания установок `Button2Click` в самом конце ее. А теперь заменим в процедурах `OnMinimizeProc` и `FormDeactivate` условие **if** `CheckBox3.Checked` **then** на **if** `MinFlag` **then**. И, наконец, создадим обработчик щелчка на кнопке `ButtonP`:

```
procedure TForm1.ButtonPClick(Sender: TObject);
begin {предпросмотр}
Panel2.Visible:=False; {закрываем панель настроек}
if Filelist.Count=0 then exit; {если нет картинок - выходим}
if Timer1.Enabled=True then
    {если демонстрация идет - останавливаем}
begin
 Run.Caption:='Запуск';
    {меняем название пункта меню обратно на Запуск}
 Timer1.Enabled:=False; {таймер остановлен}
 PlaySound(Pchar(fmusic),0, SND_PURGE); {выключили музыку}
 FlagMusic:=0; {в следующий раз опять запустим}
end;
ButtonP.Enabled:=False; {чтобы в другой раз не нажать}
MinFlag:=False; {запрещаем минимизацию в Tray}
mayClose:=True; {запрещаем закрытие в Tray}
Form4.Top:=Form1.Top+64;
Form4.Left:=Form1.Left+14;
Form4.Show;
end;
```

Согласно последним операторам, у нас новая форма откроется как бы внутри главного окна. Если кому это не понравится, то изменить ее положение легко. Причем главное окно и окно предпросмотра никак не будут связаны между собой — их легко можно передвигать по экрану, за исключением того, что окно предпросмотра будет стремиться "вылезти" вперед — например, если вы распахнете главное окно клавишей <Esc> (см. выше), то окно предпросмотра скроется, но тут же вновь появится, если отменить полнооконный режим. А вот создавать его в MDI-стиле (как отдельные окна документов в Word) нецелесообразно — это приведет только к потере полезной площади. Заметим попутно, что MDI-окна вообще сейчас не в моде — гораздо удобнее делать многооконные приложения в виде одного окна с закладками, просто тут у нас всего одно "лишнее" окно и закладки нам ни к чему.

Теперь возьмемся вплотную за модуль preview.pas и в первую очередь создадим основную процедуру вывода "превьюшек" в компонент `Image1` формы `Form4`. Перебор списка происходит согласно счетчику `ni`, который мы установим позднее. В модуле preview.pas объявим следующие переменные:

```
var
Form4: TForm4;
Arect: TRect;
nframe,nlast:integer; {счетчик и признак выхода из загрузки}
ratio:double; {соотношение сторон картинки}
iWidth,iHeight,iTop,iLeft,iLShift,iTShift:integer;
. . . . .
```

Тогда основная процедура будет выглядеть так:

```
procedure imagepreview;
{основная процедура загрузки "превьюшек" на страницу}
begin
 Form4.Button1.Caption:='Остановить';
 with Form1 do
 begin
  BmpIm:= TBitMap.Create; {создаем экземпляр BitMap}
  JpgIm:= TJpegImage.Create; {создаем экземпляр JPEG}
 end;
 ni:=0;
 repeat {основной цикл загрузки}
   if FlagE<>0 then
   begin
    Form4.Button1.Caption:='Продолжить';
    exit; {если признак выхода не 0 - выходим}
   end;
```

```
with Form1 do
begin
 try
  BmpIm.LoadFromFile(Filelist[ni]);
        {загружаем BMP-картинку с диска}
 except {это не BMP}
  JpgIm.LoadFromFile(Filelist[ni]);
        {загружаем JPEG-картинку с диска}
  BmpIm.Assign(Form1.JpgIm); {ассоциируем BitMap с JPEG}
 end;
  ratio:=BmpIm.Height/BmpIm.Width;
        {определяем соотношение сторон}
end;
iLShift:=0; {сдвиг картинки слева =0}
iTShift:=0; {сдвиг картинки сверху =0}
iWidth:=66; {ширина поля для картинки = 66 пикселов}
iHeight:=round(66*ratio);
  {высота поля для картинки с сохранением соотношения сторон}
if iHeight>66 then
       {если высота вылезла за пределы квадрата 66x66}
begin
 iHeight:=66; {тогда наоборот, высота=66}
 iWidth:=round(66/ratio);
       {а ширину поля определяем по соотношению сторон}
 iLShift:=(66-iWidth) div 2;
        {сдвиг поля слева от стороны квадрата 66x66}
 iLeft:=iLeft+iLShift; {координата левого края картинки}
end else {если высота не выходит за рамки}
begin
 iTShift:=(66-iHeight) div 2;
       {сдвиг поля сверху от стороны квадрата 66x66}
 iTop:=iTop+iTShift; {координата верхнего края картинки}
end;
if iLeft+70>=Form4.Image1.Width then
  {если правый край поля картинки
       вылезает за правый край компонента}
begin
 iTop:=iTop+70; {то организуем следующий ряд картинок -
                 сверху прибавляем 70 пк}
 iLeft:=iLShift; {слева первая картинка в новом ряду
               только на величину сдвига}
end;
```

```
  with Form4 do
  begin
   if (iTop-iTShift+70-490*nframe)>490 then
              {если достигнут нижний край окна}
   begin
    Form4.Image1.Height:=Image1.Height+490;
                 {увеличиваем высоту Image}
    nframe:=nframe+1;
    Image1.Picture.Bitmap.Height:=Image1.Picture.Height+490;
                {увеличиваем высоту картинки}
    ScrollBox1.VertScrollBox.Position:=
           ScrollBox1.VertScrollBox.Position+490;
            {сдвигаем ползунок прокрутки вниз}
    application.ProcessMessages;
              {заставляем Windows отреагировать}
   end;
  end;
  Arect:=Rect(iLeft,iTop,iLeft+iWidth,iTop+iHeight);
      {задаем размеры прямоугольника для картинки}
  iLeft:=iLeft-iLShift;
      {задаем начальные координаты для следующего раза}
  iTop:=iTop-iTShift;
  iLeft:=iLeft+70;
  Form4.Image1.Canvas.StretchDraw(aRect,Form1.BmpIm);
    {загружаем картинку в заданный прямоугольник из BitMap}
  application.ProcessMessages;{заставляем Windows отреагировать}
  ni:=ni+1;
 until ni=Form1.Filelist.Count; {и так до конца списка}
 Form1.BmpIm.Destroy; {уничтожаем объект BitMap}
 Form1.JpgIm.Destroy; {уничтожаем объект JPEG}
 Form4.Button1.Caption:='Закрыть';
end;
```

Здесь мы отвели рисунку поле (так сказать, одно "рисункоместо") 70×70 пикселов, но само изображение вписываем в квадрат 66×66 — по два пиксела со всех сторон остается для разделения соседних рисунков. Отметим, что корректно работать этот алгоритм будет именно при той высоте `Image1` (490 пикселов) и `ScrollBox1` (>510 пикселов), которые установлены на этапе конструирования в данном конкретном случае, что, конечно, концептуально неправильно. Но, с другой стороны, такие процедуры требуются нечасто, а в данном случае форму мы сделали с неизменяемыми размерами, так что все будет работать как надо. Более универсальный алгоритм, который бы годился для любых форм и любого размера "превьюшек", потребовал бы от нас на порядок больше времени на отладку.

Саму процедуру мы будем выполнять по событию onPaint формы, чтобы все происходило на глазах пользователя. А для того чтобы процесс выполнялся не каждый раз при прорисовке формы, а только один раз — когда в Image1 еще ничего нет, мы установим условие, что прорисовка выполняется, если счетчик ni равен 0:

```
procedure TForm4.FormPaint(Sender: TObject);
begin
if ni<>0 then exit; {уже нарисовали}
Form4.ScrollBox1.VertScrollBar.Position:=0;
{начальная позиция ползунка линейки прокрутки - самый верх}
iTop:=0; {расстояние первой картинки от верхнего края
     компонента Image1 =0}
iLeft:=0; {и от левого края тоже =0}
nframe:=0; {нулевой кадр}
imagepreview; {загружаем картинки}
end;
```

Так когда же мы будем счетчик ni устанавливать в начальное нулевое значение? Естественно, каждый раз, когда меняется содержание списка Filelist. Именно поэтому нам и потребовалась глобальная переменная ni, которую мы включили в главный модуль (могли бы, впрочем, и оставить в preview.pas, если перенести ссылку на него в модуле slide.pas также в общее предложение **uses**). Включим в процедуру PictureList в модуле slide.pas такие строки (сразу после первой строки с проверкой наличия картинок по вызову NumFiles):

```
. . . . .
if Form1.Timer1.Enabled=True then {если демонстрация идет -
                  останавливаем}
begin
 Form1.Run.Caption:='Запуск'; {меняем название пункта меню
                обратно на Запуск}
 Form1.Timer1.Enabled:=False; {таймер остановлен}
 PlaySound(Pchar(fmusic),0, SND_PURGE); {выключили музыку}
 FlagMusic:=0;{в следующий раз опять запустим}
end;
Form4.Close; {закрываем форму Preview}
ni:=0; {обнуляем счетчик}
FlagE:=0; {загружать Preview с начала}
Form4.Image1.Canvas.FillRect(Form4.Image1.Canvas.ClipRect);
              {чистим Preview от старых картинок}
. . . . .
```

Заодно мы здесь также выключаем автоматическую демонстрацию, если она была запущена, и удаляем с глаз долой окно предпросмотра со старым со-

держимым. А как быть с кнопкой Button1 (с заголовком **Остановить**) на форме предпросмотра? Для ее обработчика нам и понадобился флаг FlagE:

```
procedure TForm4.Button1Click(Sender: TObject);
begin
 if Button1.Caption='Остановить' then
 begin
  FlagE:=1; {если нажали на кнопку при загрузке - остановим}
  exit;
 end;
 if Button1.Caption='Продолжить' then
 begin
  FlagE:=0; {продолжим}
  imagepreview; {загружаем картинки}
  exit;
 end;
 if Button1.Caption='Закрыть' then
 Form4.Close; {закрываем форму Preview}
end;
```

Но это еще не все — минимизацию мы запретили, а кто будет восстанавливать? Для этого создадим такой обработчик события onClose формы:

```
procedure TForm4.FormClose(Sender: TObject; var Action: TCloseAction);
begin
with Form1 do
begin
 MinFlag:=CheckBox3.Checked;
        {восстанавливаем состояние минимизации}
 mayClose:= not CheckBox2.Checked;
        {восстанавливаем состояние при закрытии}
 ButtonP.Enabled:=True; {восстанавливаем кнопку Предпросмотр}
end;
end;
```

Результат загрузки в окно предпросмотра папки с фотографиями из домашнего альбома автора показан на рис. 15.3.

При выбранной нами логике окно предпросмотра будет минимально "напрягать" пользователя — его можно закрыть и вручную, а при смене папки оно закроется автоматически. В то же время загрузку можно остановить в любой момент (а при большом количестве файлов, особенно при чтении с CD, загрузка может продолжаться несколько минут). Если же ничего не менялось, то повторной загрузки не будет — при нажатии на кнопку **Предпросмотр** окно с "превьюшками" возникнет на экране сразу в том состоянии, в котором оно было закрыто.

Рис. 15.3. Результат вызова окна предпросмотра программы SlideShow

Осталось два заключительных штриха — во-первых, обещанный переход от "превьюшки" к изображению. Для этого создадим обработчик события onMouseDown для компонента Image1 формы Form4:

```
procedure TForm4.Image1MouseDown(Sender: TObject;
                Button: TMouseButton;
                Shift: TShiftState; X, Y: Integer);
begin   {вычисляем номер "кликнутой" картинки}
 if not FlagDblClick then exit;
 {если это простой щелчок - на выход}
 FlagDblClick:=False;
 If Button<>mbLeft then exit; {если кнопка не левая - на выход}
 n:=(x Div 70)+ (Image1.Width div 70)*(y div 70);
 {так вычисляется номер "кликнутой" картинки
            по координатам в рамках Image1}
 if n>Form1.Filelist.Count-1 then exit;
        {если щелкнули за последней картинкой}
 if Button1.Caption='Остановить' then
 FlagE:=1; {если нажали на кнопку при загрузке — остановим ее}
```

```
 Loadfile;
 Form4.Close; {закрываем форму Preview}
end;
```

Так как одинарным щелчком пользоваться для перехода неудобно, то мы ввели здесь флаг FlagDblClick, который объявлен в секции **var** этой же формы следующим образом:

```
FlagDblClick:boolean=False;
```

А устанавливать его будем в событии двойного щелчка, которое происходит раньше, чем onMouseDown, но в нем, к сожалению, нельзя определять координаты и кнопку:

```
procedure TForm4.Image1DblClick(Sender: TObject);
begin  {удостоверяем, что DblClick произошло}
 FlagDblClick:=True;
end;
```

При загрузке "кликнутой" картинки в главное окно мы закрываем форму предпросмотра — ведь в случае надобности ее легко восстановить одним щелчком на кнопке **Предпросмотр**. Остановка же загрузки, если она не была закончена, при этом необходима, иначе порядок отображения картинок будет нарушен.

А второй "последний" штрих — обеспечение поддержки колесика мыши при прокрутке окна с "превьюшками", как мы говорили в *главе 13*, компонент ScrollBox поддерживает колесико через пень-колоду (если вообще поддерживает). Тут, по счастью, у нас все готово: добавим в секции **private** модуля preview.pas процедуру:

```
procedure WndProc(var Msg: TMsg; var Handled:boolean);
```

В обработчике события onCreate формы Form4 добавим строку:

```
Application.OnMessage:=WndProc; {перехват оконной процедуры}
```

И, наконец, скопируем один к одному саму процедуру из *главы 13*, заменив только Form1 на Form4 и удалив то, что относится к горизонтальной прокрутке:

```
procedure TForm4.WndProc(var Msg: TMsg; var Handled:boolean);
begin
if Msg.message=WM_MOUSEWHEEL then
begin
 if (Msg.wParam and MK_CONTROL) = 0 then
 begin
   if Msg.wParam>0 then {крутим вниз}
   ScrollBox1.VertScrollBar.Position:=
           ScrollBox1.VertScrollBar.Position+8
```

```
    else {крутим вверх}
    ScrollBox1.VertScrollBar.Position:=
           ScrollBox1.VertScrollBar.Position-8;
  end;
end;
end;
```

В следующей главе мы сделаем справку, и на этом, как и обещали, с данной программой закончим. Хотя в принципе ее еще можно совершенствовать и совершенствовать — чего бы, например, не сделать по щелчку правой кнопкой мыши в окне предпросмотра всплывающее меню с информацией о картинке? Или не ввести возможность масштабировать картинки в основном окне, чтобы разглядеть детали? Или не создавать файлы предпросмотра для ускорения загрузки — как это делают все продвинутые программы подобного рода[3]? Или не обеспечить различные эффекты при переходе от одного слайда к другому? Или, наконец, читатель наверняка недоумевает, почему я тут не запрограммировал характерную для подобных приложений панель инструментов в виде кнопок? Из всего сказанного вы можете сделать вывод, почему программы обычно не делают сразу "как надо", а выпускают все новые и новые версии — кушать-то хочется все время, а дорабатывать программу можно до бесконечности. Главное тут — вовремя остановиться и не навязывать пользователю ненужную ему функциональность. И, конечно, обязательно в новой версии исправлять старые ошибки и недоработки так, чтобы количество новых ошибок не превысило разумного предела.

[3] Хочу только предостеречь читателей от одной ошибки, которой подвержены все без исключения разработчики таких программ. Ни в коем случае не располагайте файл с базой данных для предпросмотра в папке с изображениями. Нет совершенно никаких препятствий к тому, чтобы создать для хранения "превьюшек" отдельную скрытую от пользователя папку и таким образом не замусоривать и без того до крайности замусоренную среду Windows. Ведь чисто не там, где метут, а там, где не сорят.

ГЛАВА 16

About help

Справка и окно *О программе*

Help... help... can anybody help me now?
Help... help... my mind is lying on the ground
Кинг Даймонд

Если вы подумали, что я поклонник музыки в стиле "сатанинский хэви-мэтл", то крупно ошиблись — просто, на мой взгляд, изучение текстов англоязычной рок-музыки (а иногда и поп-музыки), для человека, который не воспринимает на слух беглую английскую речь, очень интересное и полезное занятие. Но об этом как-нибудь в другой раз — здесь мы поговорим о том, как же "поднять с земли" разум утомленного пользователя компьютерных программ?

А начнем мы, немного повторив то, что уже говорилось в *главе 1* по поводу справки. Во-первых, напомним эмпирическое правило, касающееся всех "ширпотребовских" программ: если пользователь не может освоиться с основными функциями программы без обращения к справке максимум за полчаса, то он ее бросает. Это касается также и *программных продуктов* (тут уместнее именно такой термин), которые предназначены для решения узкоспециальной задачи ограниченным кругом лиц — например, сервисных программ для обслуживания научных или медицинских приборов, программ для проведения диагностики (скажем, автомобильных систем) и т. п. Причем к этой категории замечание о предельной простоте и удобстве использования относятся даже в большей степени, чем к "обычным" программам — каким-нибудь текстовым редактором можно пользоваться, а можно и не пользоваться, а в случае узкоспециальных программ у пользователя просто нет выбора. А каких программ не касается такое замечание? В качестве примера можно привести программу Photoshop — и не нужно пытаться приступать к ней без подробного изучения руководства. И данном случае это правильно. Подумайте, почему?

И тем не менее справка должна быть в любом случае. И я уже писал о том, что для программиста составление справки — один из важнейших этапов

создания программы, это нужно ему самому не в меньшей степени, чем пользователю. Когда вы пишете справку, вы проходите всю логику работы с программой заново, причем с точки зрения постороннего, и тут могут выявляться такие "ляпы" и недоработки, которые в противном случае могут возникнуть только после долговременной эксплуатации. Я уверен, что если бы программисты Microsoft, прежде чем вводить понятие "языка", о чем мы долго говорили в *главе 8*, попытались письменно бы изложить свое видение этой проблемы в расчете на пользователя системы, то такой жуткой путаницы бы не возникло.

К сожалению, большинство программ, в том числе фирменных, являют собой примеры того, как справку делать *не следует*. MS-справка, как по Windows в целом, так и по отдельным продуктам, в большинстве случаев — совершенно бесполезная вещь. Лично мне еще ни разу не удалось найти там ни одного ответа ни на один реальный вопрос — включая обращение к так называемому "мастеру подсказок". Справедливости ради надо отметить, что в Windows 2000 и XP справка значительно улучшена (за счет организации по типу поисковых систем в Интернете). Но все же: вы в курсе, например, что автозапуск CD-ROM в Windows XP можно отменить как минимум тремя различными способами (а для версии XP Pro есть еще и четвертый), причем в каждом случае результат будет различный (можно отменить автозапуск на один сеанс, можно — навсегда, а можно — лишь для определенного типа содержимого)? В справке вы можете проковыряться хоть целый час, но внятного описания всех этих возможностей сразу не найдете. В Интернете ответ чаще всего найти можно гораздо быстрее. А в Windows 98 единственный, пожалуй, полезный "help" — "справка по Visual Basic", но не потому, что она правильно составлена (составлена она кошмарно), а потому, что она представляет собой относительно полный справочник по языку, и если запастись терпением, то ответ на почти любой вопрос там, в принципе, можно найти. В точности то же самое относится и к справке Delphi. Правда, чтобы действительно найти ответ в этих справках, надо еще и *правильно* сформулировать вопрос — но это не так страшно, как в общей справке Windows, где ответов чаще всего вовсе не содержится. В общем и целом я совершенно уверен в том, что общеизвестное "пользователь справку не читает" обусловлено тем, что он и не надеется найти там что-нибудь дельное — его много лет тщательно приучали к тому, что справка может содержать что угодно (например, начинаться с никому в подавляющем большинстве случаев не нужного пункта "история версий", наличие которого в самом начале справки можно расценивать только, как проявление крайнего нарциссизма авторов), но не описание программы.

Как и справка по Windows в целом, так и "Мастер подсказок" в особенности, основаны на порочном подходе, который мы условно назовем "парадигма FAQ" — когда составители пытаются перебрать максимум возможных во-

просов, которые, в их представлении, должны возникать у пользователя, и отвечают на них. И не пытайтесь идти по этому пути — даже для самых простых программ вы никогда не сможете вообразить все многообразие жизненных ситуаций. Тут уместно провести параллель с уголовным кодексом — есть законы, а есть их толкования (FAQ), и толкования составляются в большинстве случаев апостериори, т. е. представляют собой сборник ответов на конкретные вопросы, которые возникают в конкретных жизненных ситуациях, и во много раз по объему превышают сам кодекс. Если бы законодатели при составлении кодексов попытались бы заранее предусмотреть все ситуации — у них все равно ничего бы не вышло. А ведь именно так поступают составители справки Windows. Если вы хотите помочь пользователям — составляйте, конечно, сборники "ЧАВО", но только в дополнение к основной справке, а не как замену ее.

А как надо тогда составлять справку? Во-первых (и это касается не только справки, но вообще любых руководств, а по большому счету — вообще любых текстов), вы должны поставить себя на место того, кто вашей справкой будет пользоваться. Если у вас это решительно не получается, то попросите написать справку кого-нибудь еще — это может оказаться даже предпочтительнее. Не стесняйтесь разместить вместо настоящей справки простую статью о возможностях программы, написанную в свободной форме, без всяких "наворотов" типа оглавления и указателя, хорошим образцом могут служить статьи из "софтовых" рубрик компьютерных журналов. Но не пытайтесь делать это формально, чтобы только отвязаться — если ничего толкового не выходит, то поставьте тогда вместо справки адрес вашей электронной почты и попросите при возникновении вопросов обращаться без стеснения, это будет честным поступком.

Это то, что касается подхода, а как должна выглядеть грамотно составленная справка на практике? Пример хорошо написанной справки — Help в упоминавшемся мной графическом редакторе Paint Shop Pro, очень неплохая справка в WinRar. Хорошая справка — это не что иное, как *описание* (по возможности полное) вашей программы. Описание это должно представлять собой древовидную структуру и охватывать все (желательно без исключения) объекты и привязанные к ним действия, которые в вашей программе имеются. Описывайте, *что делает* тот или иной пункт вашей программы, не пытайтесь командовать, *что нужно делать* пользователю — кроме, разве что, самых очевидных случаев, когда для работы программы необходимо именно действие пользователя. ***Нельзя*** создавать пункты справки по такому образцу:

1. Как искать файлы?

Чтобы найти файл, щелкните кнопку "Поиск", после чего в открывшемся окне "Поиск файлов" укажите папку...

Что именно хочет пользователь — для вас тайна за семью печатями, может у него в жизни такого вопроса не возникало — "как искать файлы?". ***Правильно*** будет написать так:

1. Кнопка "Поиск".

При щелчке левой кнопкой мыши на кнопке "Поиск" откроется окно с заголовком "Поиск файлов", в котором вы увидите перечень папок. При указании одной из папок...

Если все же без явного указания задачи пользователя не обойтись, то желательно хотя бы начинать его с предположения: *"Если вы хотите найти файл, то..."*, тут неявно имеется в виду, что задач может быть много разных, но вот если вы хотите именно это, то тогда будьте любезны, уж не поленитесь щелкнуть такую-то кнопочку. Не смейтесь — я совершенно серьезен, потому что написание программ есть одна из разновидностей оказания сервисных услуг, и не может не подчиняться общим законам в этой области, первый из которых известен: "покупатель всегда прав" (жаль только что так называемые "маркетологи" обычно плохо эти законы усваивают).

Заметки на полях

Не могу не остановиться тут на одном вопросе, который вообще-то к теме книги прямого отношения не имеет. Крупнейшей и, к сожалению, стандартной для современного разработчика софта ошибкой является позиция, когда производитель начинает активно диктовать свои требования пользователю. Особенно неприемлемой следует считать подмену понятия "покупки" понятием "лицензирования": почти все без исключения фирменные программы на самом деле не продаются (т. е. не передаются в собственность) — они как бы предоставляются клиенту во временное (хотя, возможно, и с неограниченным сроком) пользование. Такая подмена понятий позволяет производителю оставить права на программу у себя и ввести множество ограничений, которые в случае "покупки" были бы просто незаконными: программу нельзя "ломать" (дизассемблировать), нельзя передавать другим лицам, нельзя модифицировать и перепродавать и т. п. Причем при всех этих ограничениях, тем не менее, производитель не берет на себя абсолютно никаких обязательств и не несет ответственности, скажем, за ошибки и просто плохую работу программы. То есть ситуация, в которой производитель любой материальной продукции был бы по закону обязан осуществлять замену, ремонт и даже осуществлять возврат денег, производителей софта, укрытых за юридически непробиваемой стеной лицензий, как будто не касается вовсе. Безусловно, это положение нельзя называть приемлемым — мало того, сейчас существует тенденция распространить точно такие же положения и вообще на любую так называемую "интеллектуальную собственность", не только на программы — показательны в этом смысле попытки внедрения DRM (Digital Rights Management, управление цифровыми правами) в отношении музыкальных файлов или цифрового видео. Безусловно, в недалеком будущем положение придется исправлять и запрещениями тут не отделаешься. Обсуждение подобных вопросов, конечно, выходит за рамки этой книги, я только хочу призвать читателей, которые захотят своими программами торговать, не становиться в позицию упомянутых производителей. Дело еще и

в том, что независимо от вашей позиции относительно "авторских прав" нельзя отрицать, что все эти лицензии, установочные ключи и пароли, активации через Сеть и прочие ухищрения в конечном итоге ударяют исключительно по интересам конечного пользователя, а вовсе не пиратов (имеются в виду настоящие пираты — те, кто зарабатывает деньги на продажах, а не те "пираты", которые, например, обмениваются музыкой через пиринговые файлообменники). У пиратов времени достаточно, и "на всякую хитрую гайку найдется болт с левой резьбой", как гласит народная мудрость, а вот для простого пользователя одна только необходимость вводить пароль при переустановке может заставить обратиться к пиратам. Я лично знаком с человеком, который, имея в распоряжении 3 (три!) лицензионных копии Windows XP, все же устанавливает на компьютере пиратскую — просто чтобы не связываться с активацией, потому что ему по роду деятельности — он тестирует разное "железо" — часто приходится систему переустанавливать.

Но довольно политинформаций, вернемся к справке. В указанном ранее стиле следует давать и дополнительные пояснения, что произойдет, если пользователь предпримет то-то и то-то. Например: *"если вы нажмете клавишу <F2> при открытом окне "Поиск файлов", то это вызовет перезапуск компьютера с полным разрушением файловой структуры жесткого диска"*. Обязательно давать подобные пояснения тогда, когда имеются какие-то ограничения, особенно неочевидные — например, как для настроек нашего переключателя раскладки следует упомянуть, что буквенных клавиш с атрибутом "расширенная" просто не существует в природе.

Крупнейшей ошибкой также почти всех без исключения составителей справки является то, что они не формулируют задачу в целом: что вообще делает программа, для чего она предназначена? Иногда это очень трудно выяснить даже после установки программы, не говоря уж о том, чтобы вывести краткую справку на эту тему *перед* установкой. О пояснении используемых терминов я уж и не говорю: в любимой мной программе The Bat! есть пункт под названием (в русском варианте) **Сжать все папки**. Попробуйте с ходу сообразить, что имеется в виду (на самом деле — очистка от мусора), — и в справке, которая является самым слабым местом этой замечательной программы, выяснить что-то практически невозможно.

Есть и более анекдотичные примеры: скажем, слово "контакты" применительно к телефонной книге. В русском языке, кроме "контактов" в электротехническом смысле, напрашивается значение этого слова в терминологии врачей-эпидемиологов, что вызывает совершенно неприличные ассоциации. А какие такие "контакты" могут быть в телефонной книге? Я сомневаюсь, что и в английском термин contacts применим в данном случае[1], и это совсем не только проблема недостаточной языковой грамотности разработчиков — это проблема непонимания ими случаев, когда употребляемое слово есть *специ-*

[1] Но не в американском, как меня уверили, — видимо, отсюда и все эти несообразности.

альный термин, а не то, что мы обычно подразумеваем[2]. Никто не запрещает называть записи в телефонной книге "контактами", или то, что вы сейчас читаете — "документом", просто нужно не забыть пояснить, что "документ" — это в данном случае все, что создано с помощью Word, а не паспорт, военный билет или справка по уровню доходов в семье[3].

Но оставим этот тяжелый вопрос — по поводу правильного употребления слов следует написать отдельную и совершенно некомпьютерную книгу, и этот момент еще не самое страшное: можно написать справку на корявом жаргоне и с орфографическими ошибками, но так, что будет все понятно и доходчиво изложено. А можно быть блестящим стилистом, но написать мутный текст, который придется перечитывать несколько раз, чтобы только попытаться понять, "о чем они тут?".

Вернемся к собственно справке. Только после того, как вы составите такое описание, вы можете озаботиться всеми остальными "прибамбасами" — горизонтальными гиперссылками (обязательно!), указателем и поиском. По поводу последних надо сказать отдельно. Само понятие "указателя" пришло из печатных книг, где это единственный, помимо оглавления, способ обеспечить более-менее удобную навигацию по тексту. В электронном варианте при наличии приличного поиска указатель — совершенно лишняя функция, но все же иногда может быть удобен, как средство отсеивания "мусора", если поиск организован недостаточно корректно. Но помните, что информативность указателя полностью на вашей совести. И в любом случае оглавление справки — та самая древовидная структура, о которой я говорил ранее — должно быть всегда основным способом ориентирования по ней, если реально пользоваться справкой можно только через указатель и поиск — такая справка составлена категорически неправильно. Даже странно, что в печатных технических книгах оглавления составлены обычно достаточно удобно, как будто в электронной справке действуют не те же самые законы. Между прочим, на составление технической документации есть ГОСТ, и по большому счету не

---

[2] Формально упомянутое употребление этого слова является корректным — "Словарь современного русского языка" дает в том числе толкование слова "контакт", как "связь", "взаимодействие" (латинское contactus означает "соприкосновение"). Но перед этим толкованием стоит пояснение "*перен.*", то есть любое употребление этого слова, кроме специально-электротехнического, может быть только в переносном смысле. Например, выражение "неконтактный человек" является в русском языке таким же незаконным, как и "контакт" применительно к телефонной книге, и требует, вообще говоря, специальных пояснений, это также *термин*, а не общеупотребительное значение.

[3] "Словарь современного русского языка": ***Документ.*** *1. Деловая бумага, служащая письменным доказательством чьего-либо права или обязательства. 2. Письменное удостоверение, подтверждающее личность предъявителя, его общественное, служебное положение. 3. Письменный акт, грамота…*

мешает с ними ознакомиться — хотя и необязательно, конечно, действовать строго по букве этих, иногда надуманных, правил.

Вопрос о том, создавать ли справку в стандартном стиле Windows Help, я оставляю на усмотрение читателей. Формат этот, на мой вкус, сделан совершенно безобразно, а гиперссылки, долгое время бывшие серьезным аргументом в его пользу, теперь можно вставлять даже в документы Word. Единственным серьезным доводом в пользу этого устаревшего формата является его тесная интеграция с программой, когда любому компоненту можно придать свою собственную справку, не выходя за рамки единой структуры. Но использование контекстной справки, на мой взгляд, вообще вещь порочная, типичное порождение упомянутой "парадигмы FAQ", предусмотреть именно тот вопрос, который у пользователя в данный момент возникнет, практически невозможно. Если вы отметили, я даже никогда не пользуюсь всплывающими подсказками (hint) — в большинстве случаев они только загромождают поле зрения и отвлекают пользователя. Единственный, на мой вкус, правильный способ использования всплывающих подсказок — демонстрировать доступные горячие клавиши, и надо же так случиться, что когда такая подсказка действительно потребовалась по ходу нашего повествования (в *главе 14* для пункта меню) — то ее как раз создать оказалось нельзя (не вообще нельзя, а нельзя в удобной для пользователя форме — для пунктов меню подсказки можно только выводить в какой-нибудь компонент через обработку события `onHint`).

Куда более удобной и для разработчика и для пользователя является современная справка в формате HTML (и еще более продвинутых форматах, таких как CHM). Есть много специальных инструментов, которые позволяют создавать такие справки, и мы здесь не будем их рассматривать — для этого есть специальная литература. Мы, как люди сермяжные, рассмотрим коротко только два вопроса — как можно создать и отобразить без излишних хлопот короткую справку без использования специальных средств.

Основы основ HTML

Небольшую справку в формате HTML можно сделать, конечно, в любом HTML-редакторе. Способ этот не является оптимальным по одной только причине: практически любой редактор создает HTML-код, намного превышающий по объему необходимый и достаточный. Рекордсменом в этом смысле является, конечно, MS Word и, к счастью почти забытый, MS FrontPage, но и остальные немногим лучше. Я покажу сейчас, как можно превратить вручную с помощью всего нескольких тегов HTML любой текст в красивую и удобную справку. При создании HTML-странички таким образом удобно использовать текстовые редакторы, которые имеют подсветку син-

таксиса HTML. Некоторые из них (Edit Plus) позволяют при этом и одновременно просматривать вид готовой странички в параллельном окне (обычно используя точно такой же WebBrowser, как и у нас в программе Trace), но это необязательно — проверять результат можно и через стандартный браузер.

Идеологически HTML-теги есть не что иное, как упомянутые в *главе 8* Shift-коды. Теги пишутся в угловых скобках. Встречая такой тег, браузер интерпретирует все, что встречается после него, определенным образом, пока не встретит либо отменяющий (закрывающий тег), либо — в отдельных случаях — другой такой же. Теги могут иметь параметры (атрибуты) и неограниченно вкладываться друг в друга. Закрывающие теги такие же, как открывающие, но с добавлением знака "прямой слеш" впереди, и всегда без атрибутов.

Вот основное, что нам нужно уметь для того, чтобы создавать красивые тексты:

1. Выделять элементы текста полужирным и курсивом.
2. Устанавливать абзац, перевод строки и абзацный отступ.
3. Устанавливать атрибуты шрифта (цвет, размер и начертание).
4. Создавать гиперссылки как локальные (в пределах страницы), так и глобальные (между страницами).
5. Формировать некоторые свойства страницы: заголовок и цвет фона.
6. Загружать на страницу рисунки.

По большому счету — это все. Стандартные теги `<HTML></HTML>`, которыми обычно обрамляется HTML-страничка, здесь не требуются (но если вы делаете ее для выкладки в Сеть, то нужно их поставить). Заголовочной части (`<HEAD></HEAD>`, в ней обычно располагается служебная информация) у нас также не будет (но опять же для расположения в Сети в ней нужно, по крайней мере, явно установить кодировку страницы). Могут понадобиться таблицы (а с их помощью можно сделать верстку почти любой сложности), но создавать их вручную уже, на мой взгляд, нецелесообразно — тут проще воспользоваться редактором типа Dreamweaver.

Начнем с самого начала по списку. **Выделение элементов текста** — самое простое, что можно предпринять. Полужирным отметится все, что заключено между тегами `<B></B>`, а курсивом — между тегами `<I></I>` (теги можно писать и маленькими буквами, это неважно). Упомянутая ранее возможность вкладывать теги друг в друга на практике означает, что они действуют независимо: так, конструкция `<B>жирный<I> жирный курсив</B> курсив</I>` в браузере отразится так: **жирный *жирный курсив* *курсив***.

Элементы абзаца. Обычных знаков перевода строки (символы 10+13 для Windows) для браузера не существует — точнее, он заменяет их пробелами.

Поэтому о разделении текста на абзацы нужно специально позаботиться. Отметим, что в отличие от чистого текста, где существуют только переводы строк, в HTML перевод строки и начало нового абзаца — разные вещи. На самом деле это разные вещи и в MS Word, просто не все об этом знают, думая, что там, наоборот, существуют одни абзацы, а перевода строк нет. Перевод строки в Word без образования нового абзаца осуществляется нажатием клавиш <Shift>+<Enter>, но там эта операция необязательная — абзацы могут иметь нулевой промежуток между собой, и тогда по внешнему виду результаты могут быть неотличимы (особенно если используется выравнивание влево или вправо, а не по ширине). А в браузерах абзац всегда отделяется от предыдущего промежутком, поэтому обойтись без отдельного знака перевода строки нельзя. Перевод строки в HTML обозначается тегом `<BR>` и закрывающего тега не имеет. А новый абзац начинается с тега `<P>`, который имеет закрывающий тег `</P>`, хотя его использование в случае открывающего тега `<P>` без атрибутов необязательно.

Выравнивание текста производится с помощью атрибута `ALIGN`, например, так: `<P ALIGN=center>`. В современных браузерах нормально работает и привычное для русского глаза выравнивание по ширине (`ALIGN=justify`), однако на практике его используют редко и только для колонок текста достаточно большой ширины, потому что иначе текст (без расстановки переносов) будет очень некрасивым. Кроме `center` и `justify`, атрибут `ALIGN` может, естественно, принимать значения `left` (по умолчанию) и `right`. Он также употребляется и с другими тегами (см. далее). Наконец, отступ абзаца (всего целиком, а не первой строки) задается тегом `<BLOCKQUOTE> </BLOCKQUOTE>`. Есть и отдельный тег `<CENTER> </CENTER>`, которым можно задать выравнивание по центру для целого блока, включая все его элементы.

Иногда перечисленного уже достаточно, чтобы сделать вполне читаемый текст. Пойдем дальше — **атрибуты шрифта**. Они устанавливаются с помощью тегов `<FONT> </FONT>` с нужными атрибутами. В число атрибутов может входить и название шрифта (программы от MS его всегда устанавливают) — а нам это надо? Если вы хотите сделать действительно компактный код, то нет, не надо. По умолчанию браузер (по крайней мере, в Windows) использует Times New Roman 12 кегля, и нет никакой нужды его специально объявлять. Специально следует устанавливать шрифт только, если вы хотите отобразить что-то именно конкретным шрифтом (например, Arial или Symbol), причем выбирать, разумеется, следует только из заведомо установленных в системе (и при оформлении странички для Сети не забывайте, что в отличных от Windows системах шрифт может определяться неадекватно). А для установки иных свойств шрифта достаточно атрибутов `SIZE` и `COLOR`. Причем `SIZE` (размер шрифта) можно устанавливать в абсолютных единицах (`SIZE=5`), а можно в относительных (`SIZE=+2`). В последних удобнее, т. к. абсо-

лютная шкала в HTML не есть привычный нам кегль (шрифт 12 кегля соответствует SIZE=3), и в относительных единицах мы точнее можем угадать, что получится. Цвет задается в точности так же, как для компонентов Delphi, в 24-битной модели RGB, причем можно использовать как константы-названия, так и прямое задание величины с предваряющим знаком #, в обоих случаях константа пишется в парных кавычках: COLOR="#0000FF" или COLOR="blue". Так, следующая конструкция:

```
<FONT COLOR="#FF0000" SIZE=+1>Увеличенный красный </FONT><FONT
COLOR="#FF0000">нормальный красный</FONT><FONT SIZE= -1> уменьшенный
обычный</FONT>
```

воспроизведется, как (красный, я, конечно, отобразить здесь могу только серым):

Увеличенный красный нормальный красный уменьшенный обычный

После этого опять пойдет нормальный текст.

Теги с атрибутами (и не только <FONT>) можно дробить: запись <FONT COLOR="#FF0000"><FONT SIZE=+1> приведет к тому же результату, что и просто <FONT COLOR="#FF0000" SIZE=+1>, но в первом случае закрывающих тега должно быть также два!

Для задания равномерного шрифта не требуется специально объявлять, например, Courier, а достаточно использовать теги <TT> </TT>. То есть фактически мы можем иметь дело только с двумя разновидностями шрифтов: пропорциональным и равномерным. Такой подход предпочтительнее не только с точки зрения компактности кода, но просто более грамотный, чем явное объявление шрифтов, которых в системе может и не оказаться. Для того чтобы вставить в текст специальные знаки (мы об этом упоминали в *главе 8*), следует использовать их представление в виде особых констант, начинающихся со знака &, например, неразрывный пробел обозначается, как , знак градуса — &deg и т. п. (полный их список имеется, например, на **http://fgener.narod.ru/web/html/sintax_h.htm**). Даже угловые скобки в тегах можно заменить на &lt (то же, что <), &gt (то же, что >). Обратите внимание, что ввести можно любой символ, если набрать его номер с предваряющими знаками &# (этот номер в общем случае таблице ASCII не соответствует, см указанный ресурс). Введение специальных знаков в таком представлении делает их шрифтонезависимыми — в обычный кириллический текст, если это не Unicode, очень непросто ввести испанские или французские буквы с диакритическими знаками, а в HTML — запросто.

Дальше у нас по списку идут **гиперссылки**. Они делаются по следующему образцу: <A HREF="http://bhv.ru">Издательство BHV</A>. В парных кавычках должен стоять URL (или адрес обычного файла, например, для загрузки), по-

сле окончания открывающего тега и до закрывающего — текст (или иной объект), который будет выделен в качестве ссылки. В качестве атрибута тега <A> можно указать target=blank, например, так: <A HREF= "http://bhv.ru" target =_blank>Издательство BHV</A>. При щелчке на этой ссылке откроется сайт издательства в новом окне (или новой вкладке в Opera и FireFox).

Два важных частных случая употребления тега <A> относятся к локальной ссылке (переходе на метку) и ссылке на электронную почту. В первом случае ссылка выглядит таким образом: <A HREF="#p1">Комментарии</A>, где "p1" обязательно существующая на той же странице локальная метка. Она задается вот такой записью: <A NAME="p1"> </A>, причем во избежание неприятностей, если метка стоит в конце страницы (как у нас в программе Trace), то какой-нибудь текст перед закрывающим тегом </A> должен быть (хотя бы пробел, как в данном случае). Сама метка в браузере никак не отображается. На метку, определенную таким способом, можно перейти с другой страницы, если указать ее в конце адреса URL без пробела (см. опять же текст программы Trace в *главе 13*). А ссылка на электронную почту записывается так (на примере моего собственного адреса):

<<A HREF="mailto:revich@homepc.ru">revich@homepc.ru</A>>.

Вторые угловые скобки просто отобразятся в браузере, т. е. результат будет выглядеть, как это обычно принято: <**revich@homepc.ru**>. Далее мы покажем, как можно формировать подобные ссылки в любом компоненте Delphi, не только в браузере.

Чтобы украсить страничку, можно и нужно придать ей какой-то **цвет фона** — человеку гораздо комфортнее воспринимать черный текст на желтоватом или зеленоватом фоне, но не слишком, конечно, насыщенном, чем на белом. Неплохо также выглядят странички с цветным текстом на черном фоне, только во всех таких случаях надо обязательно соблюдать правило: если текст светлых тонов, то фон должен быть из темных, и наоборот. Ни в коем случае нельзя, например, шрифт цвета Margenta ($800080) располагать на черном фоне, в таком случае можно выбрать только цвет LightMargenta ($FF00FF) или близкий к нему, причем желательно выбирать из 216 так называемых "безопасных" цветов *(см. главу 10)*. А сам цвет фона странички задается через атрибуты тега <BODY>: <BODY BGCOLOR="#FFFFCC"> (светло-желтый). Закрывающий тег </BODY> следует задать в самом конце текста. В атрибутах тега <BODY> можно установить еще много чего (цвет ссылок и текста по умолчанию, размер текста по умолчанию, фоновую картинку и т. п.), но нам здесь это не требуется.

И, наконец, **о картинках**. HTML-документ не является контейнером, как документы Word или PDF, и потому картинки должны всегда храниться отдельно. Во избежание сложностей они должны иметь один из четырех фор-

матов: JPEG, GIF, PNG или BMP (последний является не очень законным для Web, но исправно отображается всеми современными браузерами). Отображение их на страничке производится включением тега по следующему образцу: `<IMG SRC="1.JPG" HSPACE=10 VSPACE=10 HEIGHT=467 WIDTH=359 ALIGN=LEFT>`. Адрес файла рисунка `"1.JPG"` при необходимости может задаваться с путем к нему. Длину и ширину (`WIDTH` и `HEIGHT`) можно и не задавать специально, но удобнее это делать явно, чтобы точнее подогнать рисунок (а его истинные размеры при этом должны примерно соответствовать установленным, во избежание загрузки лишних мегабайт или искажений при растяжке). Если не указать атрибут `ALIGN`, то картинка займет отдельное место на страничке, а текст начнется с нижнего правого края от нее, поэтому его лучше включать (разумеется, значение `ALIGN=justify` здесь не имеет смысла, зато, кроме перечисленных ранее, есть значения `top` и `bottom`, а значение `middle` предоставляет экзотическую возможность выровнять картинку посередине документа). Наконец, обычная ошибка всех начинающих (и даже не очень начинающих) состоит в игнорировании атрибутов `HSPACE` и `VSPACE` — если их не указать, то текст будет по сторонам картинки и сверху-снизу (в зависимости от значения `ALIGN`) примыкать к ней вплотную, что очень некрасиво. В данном случае мы указываем отступы в 10 пикселов. В заключение укажем для примера, как можно оформить картинку в виде ссылки: `<A HREF="http://bhv.ru"><IMG SRC="bhvlogo.jpg"></A>`. Здесь `"bhvlogo.jpg"` — файл с логотипом издательства BHV, при щелчке на котором загрузится сайт **www.bhv.ru**. Атрибутов я здесь не указал, но они, конечно, также могут присутствовать.

Добавим еще для полноты картины, что в начале текста неплохо поставить такую конструкцию: `<TITLE>Название программы</TITLE>`. Все, что заключено между данными тегами, будет отражено в заголовке браузера и наименовании окна, конечно, это может быть не только название программы. Разумеется, подобное оформление будет сказываться только при просмотре странички в настоящем браузере. Если этих тегов не ставить, то там будет демонстрироваться название файла. И при этом не следует забывать, что мы предотвращаем повторный запуск именно через наименование главного окна программы, потому строка заголовка здесь не должна быть идентична нашему `Application.Title` — что обеспечить не так уж сложно, учитывая зависимость процедуры сравнения заголовков от регистра букв.

Если вы ранее не имели никакого представления об HTML, то вам вся эта наука покажется на первых порах безумно сложной, но поверьте, что это много проще, чем программирование. Разумеется, "настоящий" HTML с таблицами стилей, фреймами, базами данных, включением скриптов требует серьезного изучения, но того, что я изложил (если добавить сюда еще таблицы и некоторые метатеги), вполне достаточно для создания даже очень "навороченных" сайтов и страничек, которые будут в выгодную сторону отли-

чаться от многих других аналогичных практически мгновенной загрузкой (если только не очень увлекаться графикой) и заведомо адекватным отображением в любом, даже самом экзотическом браузере. Самый крупный их недостаток — весьма муторный и требующий предельного внимания процесс внесения изменений. А для наших целей составления справки это просто идеальный вариант. Хочу также заметить, что заучивать все сказанное ранее наизусть совершенно не требуется — в сети достаточно справочников по тегам HTML. А если вам все же лень самому делать все с нуля, то наиболее быстрым способом является создание заготовки странички в каком-нибудь редакторе попроще, а затем ее ручная доработка прямо в процессе написания текста — это обычно быстрее и эффективнее, чем "вылизывать" все через редактор.

Примеры текстов справки в HTML-формате вы можете найти в упомянутых далее проектах на прилагаемом диске, а теперь займемся вопросом, как можно отображать такую справку различными способами.

Справка и пункт *О программе* для Trace

Самый простой способ предоставляет нам программа Trace *(см. главу 14)*, в которой все для отображения HTML-справки уже есть — в том смысле, что компонент `WebBrowser` установлен и готов к использованию. Я перенес проект из папки Glava14\2 в новую папку (Glava16\1) и расположил там файл с текстом справки tracehlp.htm. Обратите внимание, что в заголовке страницы (в тегах `<TITLE> </TITLE>`) название программы приведено в верхнем регистре (TRACE) для того, чтобы при просмотре через браузер заголовок окна последнего отличался от заголовка окна самой программы, где это название записано, как Trace. Если этого не сделать, то вы попросту не сможете запустить программу одновременно с просмотром Help в браузере — сработает наш механизм предотвращения повторного запуска.

Чтобы отображать и убирать справку одной и той же клавишей, для удобства воспользуемся опять флагом: объявим глобальную переменную с нулевым начальным значением `FlagHelp:byte=0`. Пункт меню **Справка** у нас уже имеется, мы создадим в нем подпункт, который также назовем **Справка**, присвоим ему для краткости имя `F1` и придадим горячую клавишу также <F1>. Создадим следующий обработчик щелчка на этом пункте:

```
procedure TForm1.F1Click(Sender: TObject);
begin  {справка F1}
 if FlagHelp=0 then
 begin
  WebBrowser1.OnBeforeNavigate2:=nil;
```

```
  if FileExists(Pchar(ExtractFilePath(Application.ExeName)
                    +'tracehlp.htm'))
 then
  WebBrowser1.Navigate(Pchar(ExtractFilePath
          (Application.ExeName)+'tracehlp.htm'));
  FlagHelp:=1;
 end else
 begin
  if FileExists(ftempname) then
  WebBrowser1.Navigate(Pchar(ftempname));
  FlagHelp:=0;
  Application.ProcessMessages;
  WebBrowser1.OnBeforeNavigate2:=WebBrowser1BeforeNavigate2;
 end;
end;
```

При первом нажатии у нас в окне браузера показывается текст, при втором — исчезает. Чтобы при повторном нажатии клавиши даже с самого начала, когда еще поиск не производился, окно браузера у нас очищалось, добавим в процедуру `onCreate` формы создание пустого файла `ftemphtm` сразу после запуска программы:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 ftempname:=ExtractFilePath(Application.ExeName)+'trace000.htm';
 assignfile(ftemphtm,ftempname);
 rewrite(ftemphtm); {создаем временный файл}
 closefile(ftemphtm);
 IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini')); {если не
было - создаем, иначе открываем}
. . . . .
end;
```

Но надо и предусмотреть очистку браузера автоматически, когда пользователь жмет на **Найти**, чтобы не заставлять его производить лишние неочевидные действия. Для этого достаточно включить в процедуру `SearchFile` самой первой строкой оператор `FlagHelp:=0;`. Кстати, в таком алгоритме работы есть одна неприятная деталь: располагать внешние ссылки в файле помощи здесь бессмысленно, ведь мы запретили браузеру "выходить наружу" (`WebBrowser1.OnBeforeNavigate2:=nil`). Если же мы не поставим этого запрещения, то он будет каждый раз вызывать внешний браузер при переходе на метки внутри страницы. Но это только потому, что мы здесь `WebBrowser` используем в своих целях, при нужде данную ситуацию можно обойти — просто тут это необязательно, ссылку на домашнюю страничку мы поставим в окне **О программе**, которой сейчас и займемся.

Не будем использовать отдельную форму, а поступим так же, как и в случае настроек — поставим невидимую изначально панель, на которой и расположим все, что требуется. В данном случае сделаем это окно в минималистском стиле: просто продублируем в нем то, что написано в последних строках файла помощи, где приведено имя, e-mail автора и также добавим URL его домашней странички, который, как мы писали ранее, в самом файле Help расположить так просто не получается. Но тут встает интересная задача: а можно ли сделать так, чтобы ссылка в компоненте, например, `Label`, отображалась именно как ссылка и имела всю нужную функциональность?

Самый простой путь для этого — просто имитировать ссылку с помощью вызова знакомой нам функции `ShellExecute`. Но чтобы все было "по-взрослому", надо также придать строке соответствующий внешний вид и сделать так, чтобы курсор мыши менялся на изображение руки (в кои-то веки нам реально потребовалось изменить курсор!). Для того чтобы это сделать, удобнее всего просто поставить отдельный компонент, которому и придать все нужные свойства прямо целиком. Использовать мы будем для этой цели, однако, не `Label`, а `Edit`, по той причине, что пользователь может захотеть скопировать текст, содержащий существенную информацию — не перейти по ссылке, а просто где-то ее зафиксировать. А из компонента `Label`, или канвы самой формы, как часто размещают информацию **О программе**, скопировать ничего нельзя — и пользователь вынужден перепечатывать с экрана фамилию разработчика, название фирмы, какой-нибудь почтовый адрес или телефон, а если есть и интернет-ссылка, которая не работает, как ссылка, то и ее — более идиотскую ситуацию трудно себе представить.

Итак, поставим на форму панель (`Panel4`) , очистим у нее заголовок, и придадим ее свойству `Font` значения "полужирный" 10-го кегля (чтобы не возиться со всеми компонентами в отдельности). Свойству `Visible` придадим значение `False`. Чтобы панель не мешала нам разглядывать остальные компоненты на форме, мы ее расположим пониже, а при запуске будем устанавливать на нужном уровне динамически. На панель поставим компонент `Memo`, установим у него свойства `ReadOnly`, `ParentColor` и `ParentFont` в `True`, а также уберем рамку, чтобы он слился с фоном (`BorderStyle` в `bsNone`). Ниже поставим два компонента `Label` и два — `Edit`, у первых надо установить свойство `ParentFont` в `True`, а у вторых (`Edit4` и `Edit5`) — свойство `Font` в "полужирный" "подчеркнутый" 10-го кегля и цвета `clBlue`. Также следует у компонентов `Edit` установить в `True` свойство `ParentColor` и `ReadOnly`, свойство `Cursor` в `crHandPoint` и лишить их рамки, как и `Memo` (`BorderStyle` в `bsNone`). В самом низу установим кнопку **Закрыть** (`Button6`). После того как компоненты будут расставлены, и нужный текст будет в них введен, панель будет иметь вид, показанный на рис. 16.1.

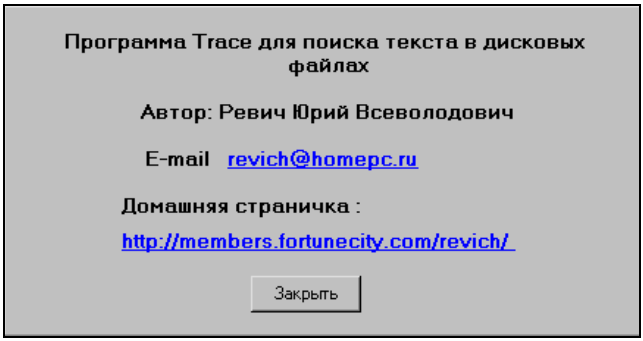


Рис. 16.1. Панель **О программе** программы Trace

Теперь добавим в меню **Справка** еще один пункт **О программе** (назовем About1, придадим горячую клавишу <Ctrl>+<F1>) и напишем для него такой обработчик:

```
procedure TForm1.About1Click(Sender: TObject);
begin {О программе}
 Panel4.Top:=Panel3.Top;
 Panel4.Visible:=True;
 Button6.SetFocus;
end;
```

Чтобы долго не вычислять положение панели, мы просто выводим ее на тот же уровень, что и **Настройки**. А для кнопки **Закрыть** сделаем такой обработчик:

```
procedure TForm1.Button6Click(Sender: TObject);
begin {закрыть О программе}
 Panel4.Visible:=False;
end;
```

Теперь осталось только сымитировать щелчок по ссылке, как мы писали ранее. Для этого напишем идентичные обработчики для события onClick компонентов Edit4 и Edit5 (проще написать отдельные обработчики, т. к. процедуры разные):

```
procedure TForm1.Edit4Click(Sender: TObject);
var mst:string;
begin {ссылка на e-mail}
 mst:='mailto:'+Edit4.Caption;
 ShellExecute(Self.Handle,'open',pChar(mst),
       NIL,NIL,SW_SHOWNORMAL);
end;
```

```
procedure TForm1.Edit5Click(Sender: TObject);
begin {ссылка на Home Page}
 ShellExecute(Self.Handle,'open',pChar(Edit5.Caption),
       NIL,NIL,SW_SHOWNORMAL);
end;
```

Теперь возьмемся за остальные наши программы.

Справка для переключателя клавиатуры

Напомню, что переключатель клавиатуры LangSwitch *(см. главу 7)* мы еще не доделали — эта программа, в отличие от остальных наших примеров, обязательно требует автоматической инсталляции — чтобы не заставлять пользователя вносить изменения в системные настройки вручную. Поэтому мы сейчас напишем справку и покажем, как ее выводить, а в дальнейшем доделаем программу, только нужно запомнить, что в текст справки придется внести изменения. Перенесем проект из папки Glava7\3 в новую папку Glava16\2, причем все, что относится к ловушке (Langhook), мы переносить не будем, за исключением готового файла Langhook.dll — туда вмешиваться нам уже не потребуется. Справку я расположил в файле Lshelp.htm и поместил его в ту же папку.

Начнем с окна **О программе**. Здесь целесообразно добавить к проекту новую форму — порядка ради пойдем официальным путем, и добавим форму в стиле AboutBox (**File | New | Other | Forms | AboutBox**). Вообще-то никаких принципиальных удобств в ее использовании нет — ничего такого сверхъестественного она не предлагает, чего нельзя было бы так же просто сделать своими руками. К тому же нам все равно придется исправлять упомянутую ранее концептуальную ошибку — все заготовки для надписей на этой форме реализованы через компоненты `Label`, и скопировать текст через буфер обмена не получится (рис. 16.2).

Рис. 16.2. Стандартная форма в стиле AboutBox

Компоненты типа Label под названием Product Name, Version и Copyright мы используем, заполнив их, правда, не совсем соответствующими названиям надписями, справа от компонента Label1 (где по умолчанию написано Comments) поставим Edit1 и ниже его еще один Edit — Edit2. В них внесем адреса e-mail и домашней странички, а также придадим им свойства (Font, ParentColor и т. п.), как мы делали ранее для аналогичных компонентов в Trace. Изменим порядок фокусировки, чтобы кнопка ButtonOk при отображении формы фокусировалась первой, для этого ее свойство TabOrder установим в значение 0 (у панели тогда оно автоматически станет равным 1). В заголовке формы напишем О программе LangSwitch. Назовем образовавшийся модуль About, внесем в его предложение **uses** модуль ShellApi, и напишем следующие обработчики событий:

```
procedure TAboutBox.FormCreate(Sender: TObject);
begin
  Ico:=TIcon.Create; {иконку для отображения}
  instance :=GetModuleHandle(nil); {получаем дескриптор модуля}
  Ico.Handle :=LoadIcon(Instance,'MAINICON');
                {дескриптор главной иконки}
  ProgramIcon.Picture.Graphic:=Ico; {загружаем в Image}
end;

procedure TAboutBox.OKButtonClick(Sender: TObject);
begin
 Ico.Destroy;
 AboutBox.Hide; {скрываем форму}
end;

procedure TAboutBox.Edit1Click(Sender: TObject);
var mst:string;
begin {ссылка на e-mail}
 mst:='mailto:'+Edit1.Text;
 ShellExecute(Self.Handle,'open',pChar(mst),
          NIL,NIL,SW_SHOWNORMAL);
end;

procedure TAboutBox.Edit2Click(Sender: TObject);
begin {ссылка на Home Page}
 ShellExecute(Self.Handle,'open',pChar(Edit2.Text),
              NIL,NIL,SW_SHOWNORMAL);
end;
```

В компоненте Image, который тут называется ProgramIcon, мы и отображаем MAINICON программы. Нет никаких проблем и в том, чтобы нарисовать специ-

альную картинку и вставить ее прямо на этапе конструирования или загрузить через ресурсы, как мы делали в *главе 11*. В остальном тут ничего нового нет.

В основной программе мы сразу еще сделаем два пункта всплывающего меню. В секции объявлений добавим константы `idmAbout=3` и `idmHelp=4` и вставим создание пунктов **Справка** и **О программе** через вызов `AppendMenu`. Затем добавим ниже процедуру создания формы `AboutBox` (напомню, что в нашей программе ничего автоматически не создается) — туда, где уже есть процедура создания формы `Form1` **Установки**. С пунктом **Справка** разберемся позднее. Итого весь этот фрагмент кода будет выглядеть так:

```
. . . . .
popMenu:=CreatePopupMenu;
AppendMenu(PopMenu, MF_STRING,idmHelp, 'Справка');
AppendMenu(PopMenu,MF_STRING,idmAbout,'О программе');
AppendMenu(PopMenu, MF_STRING,idmSets, 'Установки...');
AppendMenu(PopMenu, MF_SEPARATOR, 0,'' );
AppendMenu(PopMenu,MF_STRING,idmEXIT,'Закрыть');
Form1:=TForm1.Create(Application); {создаем экземпляр формы}
AboutBox:=TAboutBox.Create(Application); {создаем форму О программе}
. . . . .
```

Теперь создадим обработчик щелчка на пункте **О программе**, добавив в оконной процедуре по условию `if Msg = wm_COMMAND` такую строку:

```
if wpr =idmAbout then AboutBox.Show; {показываем форму О программе}
```

Если после запуска мы щелкнем правой кнопкой мыши на иконке программы, а затем по пункту **О программе**, то получим окно, которое изображено на рис. 16.3. В нем, как и в программе Trace, можно по щелчку запустить почту или загрузить домашнюю страничку через браузер.

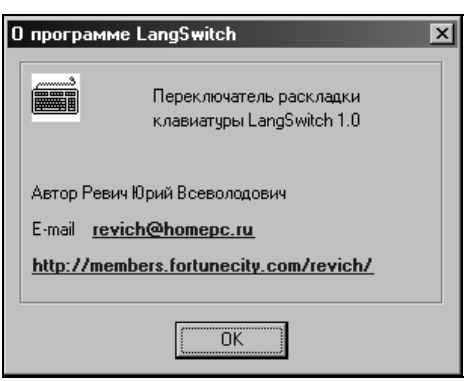


Рис. 16.3. Окно **О программе LangSwitch**

Осталось соорудить отображение справки. Раз уж у нас все равно основное приложение существует только в виде иконки, то зачем нам грузить всякими окнами его и дальше? Давайте больше не будем "пихать" туда формы, а просто вызовем браузер, который установлен в системе по умолчанию:

```
if wpr =idmHelp then {отображаем Help}
ShellExecute(FHandle,'open',Pchar(ExtractFilePath
    (Application.ExeName)+'LShelp.htm'),NIL,NIL,SW_SHOWNORMAL);
```

Ну, а в SlideShow мы пойдем для отображения справки и окна **О программе** еще одним путем.

Справка в SlideShow

Перенесем проект SlideShow из папки Glava15 в новую папку (Glava16\3) и придадим ему номер версии 2.1. Файл со справкой я назвал slide.htm и расположил его в той же папке. Добавим к проекту новую форму типа Tabbed Pages (**File | New | Other | Forms | Tabbed Pages**), которая представляет собой форму-диалог с установленным компонентом `PageControl`. В нем как раз будет три закладки (компонента типа `TtabSheet`), первую из которых мы назовем **Справка**, вторую — **О программе**, а третья зачем? А третью мы назовем **Установки** и потом перенесем на нее нашу панель с настройками, чтобы все было единообразно. Этот перенос и будет самым хлопотным в переработке программы.

Снизу новой формы, которую мы назовем `FormAbout` (а модулю дадим имя Abouthelp.pas), имеются три стандартные кнопки, они нам не нужны и мы их удалим вместе с панелью, на которой они установлены. Всю форму немного уменьшим в размерах. На первую закладку поместим наш любимый `WebBrowser`, а на вторую перенесем прямо через буфер обмена панель со всеми компонентами из предыдущего проекта, с формы `AboutBox` (для удобства переноса можно просто временно запустить еще один экземпляр Delphi). Внесем в эти компоненты некоторые изменения: во-первых, заменим текст, во-вторых, сделаем шрифт в надписях полужирным и покрупнее, в-третьих, соорудим из заглавной картинки с кленами (она у нас находится в папке Glava2 под названием 0klen2.jpg) маленькую "превьюшку" (IcoKlen2.jpg) размером 66 пикселов по ширине, и вставим ее в компонент `ProgramIcon` (остальные придется немного при этом сдвинуть). Третью закладку пока не трогаем. Переходим к модулю slide.pas и добавляем к главному меню пункт **Справка**, а в нем подпункты **Справка** (под названием `F1` и с горячей клавишей также <F1>) и **О программе** (под названием `About` и с горячей клавишей <Ctrl>+<F1>). Осталось только для них написать обработчики:

```
procedure TForm1.F1Click(Sender: TObject);
begin {Демонстрация справки}
 if FileExists(ExtractFilePath(Application.ExeName)+'slide.htm')
 then
 with FormAbout do
 begin
  TabSheet1.Enabled:=True;
  WebBrowser1.Navigate(ExtractFilePath(Application.ExeName)+
                       'slide.htm');
  Caption:='Справка';
  PageControl1.ActivePage:=TabSheet1; {выводим вперед Справку}
  ShowModal;
 end;
end;

procedure TForm1.AboutClick(Sender: TObject);
begin {О программе}
 with FormAbout do
 begin
 if not FileExists(ExtractFilePath
           (Application.ExeName)+'slide.htm') then
 TabSheet1.Enabled:=False;
 Caption:='О программе';
 PageControl1.ActivePage:=TabSheet2; {выводим вперед О программе}
 ShowModal;
 end;
end;
```

Здесь мы с помощью свойства ActivePage выводим вперед каждый раз нужную закладку. В случае, когда файла помощи не существует, первый пункт меню просто не сработает, а панель **О программе** все равно будет выводиться, при этом закладка **Справка** деактивируется, но если файл вдруг каким-то чудом появится, она при очередном обращении к первому пункту будет активирована заново. Закрывать форму FormAbout специально не требуется, она закрывается кнопкой-крестиком на заголовке. Перед компиляцией Delphi, разумеется, предложит добавить форму FormAbout в **uses** модуля slide.pas, и мы с удовольствием согласимся. Результаты демонстрации справки и **О программе** для SlideShow представлены на рис. 16.4 и 16.5.

Теперь возьмемся за нелегкую задачу переноса панели настроек. То есть перенести-то как раз ее легче всего — надо просто выделить ее целиком, забрать в буфер обмена и вставить в выделенный компонент TabSheet3. Сложности начнутся дальше — во-первых, при попытке компиляции Delphi выдаст нам с полсотни ошибок, но и это тоже не такая уж страшная проблема. Доба-

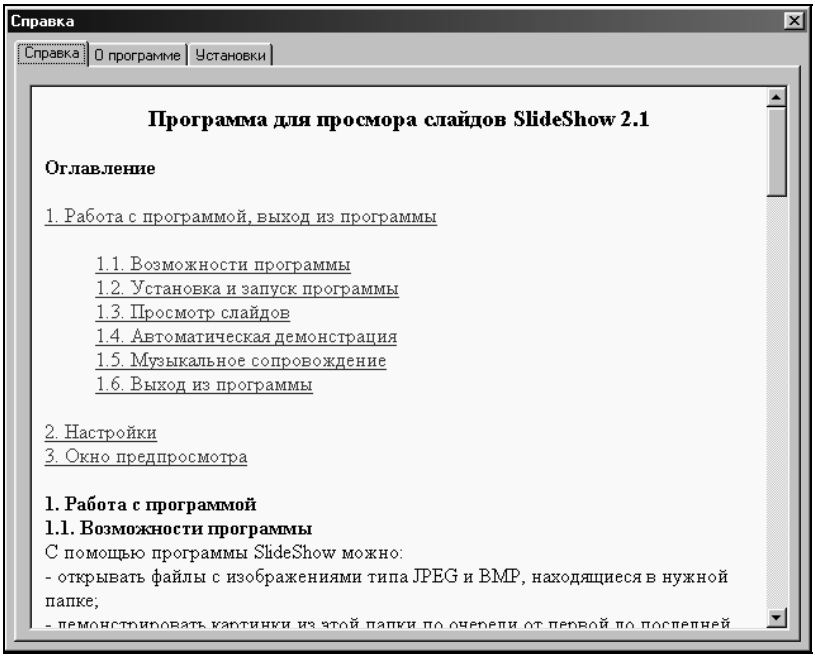


Рис. 16.4. Вкладка **Справка** окна **Справка** SlideShow

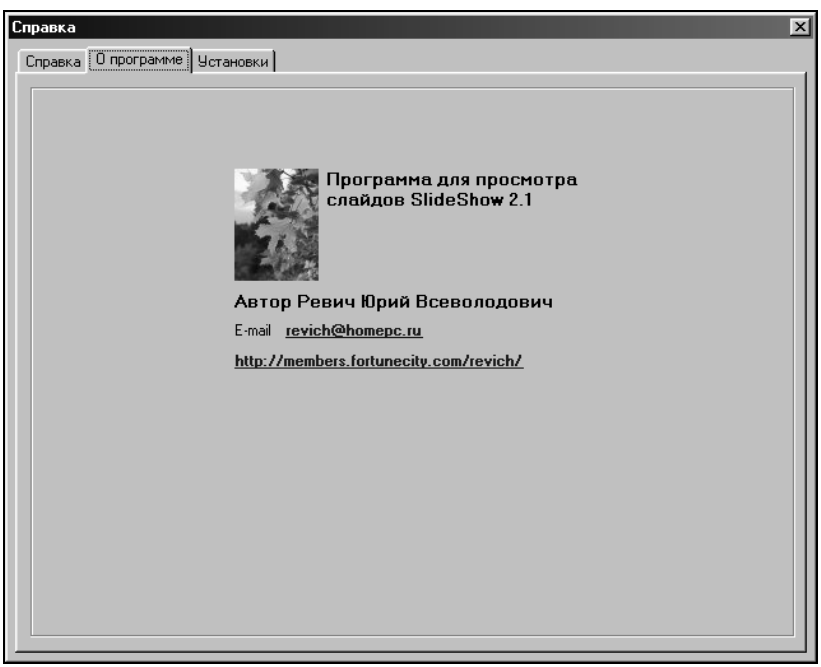


Рис. 16.5. Вкладка **О программе** окна **Справка** SlideShow

вим для начала в модуль Abouthelp ссылку на модуль slide, а потом в модуль preview ссылку на модуль Abouthelp. Затем будем исправлять по очереди все ошибки, на которые нам указывает Delphi. Прежде всего, надо будет везде удалить строку `Panel2.Visible:=False`, остальные исправления, на которые Delphi укажет нам сама, я перечислять не буду — слишком они многочисленны, но в то же время сложностей там никаких нет: сами компоненты (кроме `Edit1`, который теперь называется `Edit3`, и собственно панели, называющейся теперь `Panel3`) названия сохранили, к ним надо только добавить принадлежность к `FormAbout`.

А вот те исправления, где подсказок Delphi не будет, я перечислю. Из модуля slide.pas в модуль Abouthelp.pas надо перенести обработчик нажатия кнопки `Button2` (которая также сохранила свое название, хотя и `Button1` у нас нет), добавив к нему в конце вместо `Panel2.Visible:=False` строку с оператором `FormAbout.Close`. Все наименования `Edit1` в этой процедуре надо заменить на `Edit3`. У самой панели (`Panel3`) надо установить свойство `Visible` в `True`. Содержание обработчика события `Edit1.KeyDown` (он никуда не делся, хотя самого `Edit1` на главной форме уже нет) также надо перенести в модуль Abouthelp.pas, создав обработчик события `Edit3KeyDown`, а потом приписать компоненту `ColorBox1`, как и было у нас сделано ранее, для аналогичного события `ColorBox1KeyDown` тот же самый обработчик `Edit3KeyDown`. Наконец, в конце модуля следует добавить процедуру загрузки справки в браузер в фоновом режиме (иначе при загрузке модуля Abouthelp, например, через пункт **Установки**, и последующем обращении к закладке **Справка**, окно браузера окажется пустым). Делается это по событию `onEnter` компонента `PageControl1`, а по сути это будет просто укороченная процедура демонстрации справки (см. ранее процедуру `TForm1.F1Click`):

```
procedure TPagesDlg.PageControl1Enter(Sender: TObject);
begin {Демонстрация справки}
 if FileExists(ExtractFilePath(Application.ExeName)+'slide.htm')
 then
 with FormAbout do
 begin
  TabSheet1.Enabled:=True;
  WebBrowser1.Navigate(ExtractFilePath(Application.ExeName)+
                       'slide.htm');
  Caption:='Справка';
 end;
end;
```

Наконец, в модуле slide.pas надо сделать одну важную операцию. Все, что в процедуре `Form1.Create` относится к манипуляциям с настройками, надо

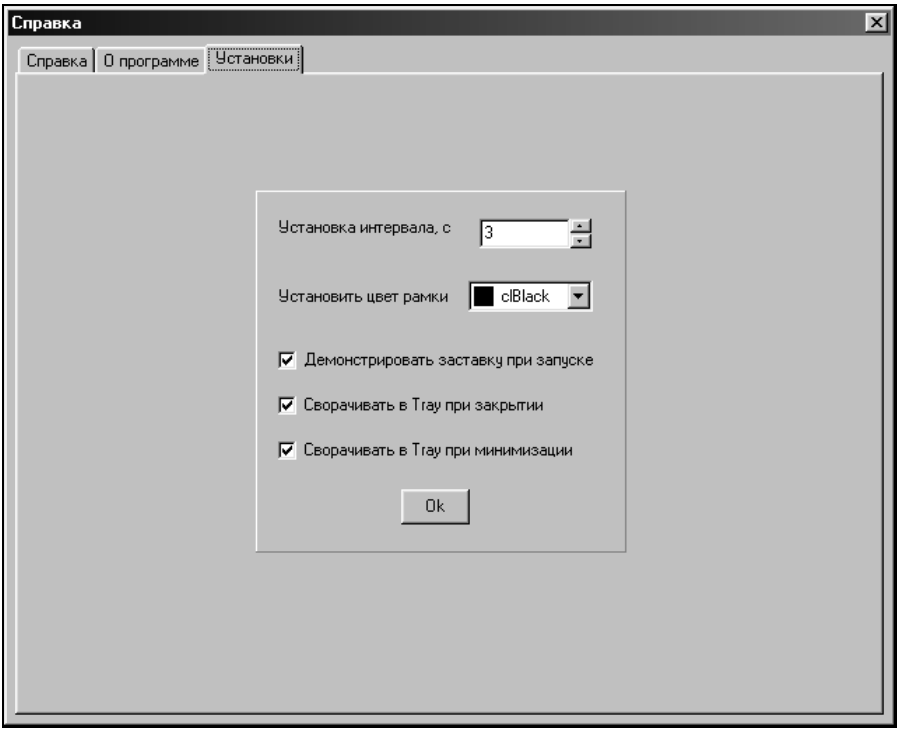


Рис. 16.6. Вкладка **Установки** окна **Справка** SlideShow

перенести в обработчик события `onActivate` той же формы `Form1` — дело в том, что форма `FormAbout` создается позже `Form1`, и когда вы обращаетесь к свойствам ее компонентов из обработчика `onCreate` главной формы, их еще просто не существует. Первую строку из этой процедуры (`FormAbout.Label1.Caption:=...`) следует удалить. Ну, и последнее, что нужно сделать, — переписать обработчик обращения к пункту меню `Set1` (**Установки**):

```
procedure TForm1.Set1Click(Sender: TObject);
begin
{обращение к пункту меню Установки}
 with FormAbout do
 begin
 if not FileExists(ExtractFilePath(Application.ExeName)
                  +'slide.htm') then
 TabSheet1.Enabled:=False;
 Caption:='Установки';
 PageControl1.ActivePage:=TabSheet3; {выводим вперед Установки}
```

```
  ShowModal;
  end;
end;
```

То что у нас должно получиться, представлено на рис. 16.6.

Ну, вот, кажется, и все с проектом SlideShow. Мы его много раз по ходу дела "перелопачивали", и я надеюсь, что внимательному читателю это пойдет на пользу — при этом мы рассмотрели множество самых разных вариантов реализации тех или иных функций, которые в случае чего можно использовать в своих программах. Наверняка, вы обнаружите ошибки и просто недоработки — не стесняйтесь, пишите.

ГЛАВА 17

Регистрируем и инсталлируем

Как создать инсталлятор и деинсталлятор самостоятельно

Свидетельство № 990365 об официальной регистрации программы в реестре.

Случайно подсмотренное

Тот, кто будет утверждать, что самостоятельно создавать инсталлятор нет никакого смысла — для этого есть множество разных программ, начиная с монструозной Install Shield самой Delphi (правда, с красивым интерфейсом результата) и заканчивая творениями безвестных умельцев — будет прав и неправ одновременно. Прав — потому что действительно, все эти программы наличествуют, и пользоваться ими, по крайней мере, не сложнее, чем делать самому. А неправ — потому что самое трудное в создании инсталлятора заключается не в том, чтобы его написать, а в том, чтобы сделать это *правильно*. Ну и вообще — данная книга не о том, как пользоваться чужими программами, а о том, как писать свои. Поэтому мы будем делать все ручками. Кстати, а что именно делать? Мы в этой главе создадим инсталлятор для переключателя раскладки клавиатуры LangSwitch, и тем самым покончим и с ней. Для SlideShow и остальных наших программ мы инсталляторы договорились не делать, потому что они располагаются максимум в двух файлах (считая справку) и в систему не вмешиваются — так зачем для них инсталлятор? Побольше бы таких программ делалось — меньше было бы мусора в реестре и на диске, а манипуляторы типа "мышь" реже выходили бы из строя.

Но "переключалку" инсталлировать надо — там необходимо вносить изменения в системные настройки, хоть и минимальные, но заставлять делать это самого пользователя очень уж по-любительски. А раз инсталлировать — значит, и деинсталлировать. И так, чтобы ни следа не осталось! Я на своем веку встретил буквально пару-тройку программ, которые выполняли бы процедуру деинсталляции до конца корректно. Даже любимый браузер Firefox все почти делает правильно, а вот папку Mozilla оставляет почти пустой, но неубранной!

Я не буду вдаваться в тонкости устройства реестра, потому что это бессмысленно — даже раритетная книга [36], специально этому делу посвященная, охватывает едва ли треть необходимых сведений. Но почитать ее перед сном очень рекомендую — общие принципы знать никогда не мешает. Здесь же нам не придется даже понимать, какие бывают разновидности ключей и их параметров — мы просто воспользуемся чужим (моим) опытом и сделаем все по шаблону. Некоторые дополнительные пояснения я дам по ходу дела.

Заметки на полях

А как вообще стоит использовать реестр? Например, нужно ли переносить туда пользовательские установки? Отвечаю — нет, не нужно, хотя никто вам этого запретить не может. Но, например, регистрация списка последних открытых файлов может понадобиться вам для того, чтобы эти файлы отображались в системном меню **Документы**. Только для этого нужно не просто вести этот список в реестре, а принять еще некоторые меры, иначе любая программа-чистильщик реестра отнесет ваш список к неиспользуемым параметрам и будет совершенно права. Если же вам не надо ничего такого — мое мнение, что лучше вообще в реестр не соваться, а сохранять установки в старых добрых INI-файлах, которые всегда под рукой. Пользователи вам скажут только спасибо, если для удаления вашей программы достаточно стереть рабочую директорию вашего приложения со всеми файлами и забыть о нем. Можно еще использовать реестр для хранения секретных ключей — например, для предотвращения повторной установки shareware-программ с ограниченным сроком действия, но при этом следует помнить, что взломать такую защиту куда проще, чем если вы, например, создадите безымянный секретный файлик где-нибудь в недрах системных папок. В остальном — мое глубоко выстраданное мнение заключается в том, что если реестр вообще для чего-то нужен (обходятся же без него в Unix и ничего себе живут), то для того, чтобы хранить системные установки (в широком смысле — автозапуск, например, тоже относится к системным установкам). И вмешиваться туда следует только с целью их изменения. О том, что это нужно делать с осторожностью, вы и так знаете, но мы тут ничего обрушающего систему до полной неработоспособности делать не намереваемся.

А теперь займемся составлением детального плана инсталляции и на его основе — деинсталляции.

Итак, при ***инсталляции*** нам требуется:

1. Показать пользователю текст с описанием основной функциональности программы. Мы с вами не Microsoft, и лицензию демонстрировать не бу-

дем, но упомянуть о правах пользователя считается хорошим тоном. Неплохо, например, подпустить немного иронии в таком духе: *"Все мысли, которые могут прийти в голову при чтении данной книги, являются объектом авторского права. Их нелицензированное обдумывание запрещено"* (Пелевин). Тут же предложить выбор: устанавливать — не устанавливать. Выход из диалога: **Начать установку**, **Прервать установку**.

2. Указать папку, куда будут копироваться файлы программы, и предложить подтвердить или выбрать другую. Скопировать в выбранную папку файлы программы. Об организации процесса по пунктам 1 и 2 см. далее.
3. Предложить на выбор: убрать системный индикатор раскладки или не убирать. Запомнить для деинсталляции, демонстрировался ли системный индикатор или нет.
4. Если это Windows XP, то сообщить, что будет отключен пункт **Переключать раскладки клавиатуры** *(см. главу 7)*. Запомнить для деинсталляции состояние этого пункта.
5. Предложить на выбор: регистрировать программу в автозапуске или нет.
6. Зарегистрировать программу (точнее, ее деинсталлятор) в меню **Установка и удаление программ**.
7. Предложить перезапустить компьютер. Выход из диалога: **Перезапустить сейчас**, **Закончить установку без перезапуска**.

При ***деинсталляции***:

1. Продемонстрировать диалог: удалять — не удалять программу.
2. Вернуть системный переключатель и пункт **Переключать раскладки клавиатуры** в прежнее состояние.
3. Удалить программу из меню **Установка и удаление программ** и из автозапуска, если она там есть.
4. Удалить файлы, в том числе сам деинсталлятор, и папку программы. Если в ней имеются посторонние файлы, запросить пользователя разрешения на удаление. Примечание: то же самое нужно сделать при инсталляции программы, если пользователь прервет установку, только набор удаляемых файлов будет другой.
5. Продемонстрировать сообщение **Удаление завершено** и предложить перезапустить компьютер.

Как видите, никаких таких специальных Shield не требуется. Все небольшие изменения в реестре, которые нам нужны, легко внести с использованием объекта `TRegistry`. Пункт в главном меню программ у нас создавать не требуется, но если потребуется, то его легко включить по методике с использова-

нием DDE, как указывается в заметке Олега Зайцева, продублированной на множестве ресурсов в Сети (например, здесь: **http://www.delphimaster.ru/cgi-bin/faq.pl?look=1&id=15-988622372**).

Теперь решим вопрос с формой представления программы. Хотя файла всего два, все равно удобно использовать самораспаковывающийся (SFX) архив. WinRar, который я так обругал за излишне навороченный интерфейс в начале книги, умеет создавать отличные SFX-архивы с демонстрацией, что немаловажно, текста и диалога выбора перед распаковкой и всеми остальными "прибамбасами", включая выбор папки, демонстрацию линейки хода распаковки и т. п., так что для выполнения первых двух пунктов ничего выдумывать не придется. Задача нашей программы setup будет состоять в том, чтобы выполнить необходимые установки, начиная с пункта 3. Правда, если пользователь решит, что установку надо прервать на одном из последующих этапов, то, как указано ранее, подчищать за собой (удалять распакованные файлы и т. п.) придется уже нашей программе.

Заметки на полях

Если программа использует DLL, как в нашем случае, то их часто включают в перечень библиотек, запускаемых системно — аналог автозапуска для обычных EXE-файлов. Так как в данном случае никто, кроме нас, использовать нашу ловушку не будет, то замусоривать память нам абсолютно ни к чему, особенно это касается Windows XP, которая имеет привычку "на всякий случай" хранить все единожды запущенные DLL в оперативной памяти. Но на другой "всякий случай" укажу ссылку на статью с хорошим примером, как это можно сделать: **http://msk.nestor.minsk.by/kg/2003/26/kg32602.html**.

Разберем некоторые остальные пункты. Системный индикатор раскладки есть программа, которая носит странное имя internat.exe (находится в папке ..\WINDOWS\system32). Она просто присутствует в автозапуске и ее удаление оттуда не вызывает никаких проблем. Если же вы, несчастный, имели смелость (или глупость — как кому больше нравится) установить Office XP (в любой из версий Windows), то эта программа заменяется на другую с не менее странным названием ctfmon.exe (на самом деле это так называемые "расширенные службы текстового ввода", включающие ввод текста голосом, распознавание рукописного ввода и т. п.). Причем в Windows XP эта программа вместо internat.exe имеется изначально. Ее также можно спокойно удалять из автозапуска, хотя теоретически с этим могут быть проблемы (см. **http://www.computery.ru/upgrade/faq/soft/2003/sfaq_112.htm**). Перечень программ автозапуска находится во всех версиях Windows в реестре по адресу: `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run`. Причем в 98-й и в XP почему-то регистр названия "Software" разный (в XP — заглавными буквами), но, к счастью, компонент `TRegistry` их не различает.

Включить нашу программу в автозапуск можно, добавив ее по тому же ключу. Разумеется, можно и просто добавить ее в папку Автозапуск, но в Windows XP этих папок может быть так много, что проще один раз обратиться к реестру.

Отключение пункта **Переключать раскладки клавиатуры** (оставив отмеченным пункт **Переключать языки ввода**, см. *главу 7*) в Windows XP (и других системах, если вдруг этот пункт там появится после установки Office XP — я это проверять не стал) делается заменой значения ключа `Layout Hotkey` по адресу `HKEY_CURRENT_USER\Keyboard Layout\Toggle` на 3 (изначально он имеет значение 1 или 2, и это значение надо запомнить для деинсталляции). Несмотря на цифровое значение, тип этого параметра — `REG_SZ` (строковый).

Зарегистрировать программу в меню **Установка и удаление программ** можно, если добавить ключ с произвольным названием в ветви реестра `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall`.
В этом ключе должно быть два параметра: `DisplayName` (то имя, которое будет отображаться в меню) и `UninstallString` (путь к файлу с деинсталлятором). Оба параметра должны также иметь тип `REG_SZ`.

Ну, а теперь приступим — сначала к собственно программам. Создадим новый проект под названием, как и полагается, просто Setup (модуль назовем inst.pas — папка Glava17\Setup) и скопируем в ту же папку с проектом три необходимых файла (из папки Glava16\2): Langswitch.exe, LangHook.dll и LShelp.htm. Отметим, что отладка инсталляционной программы — весьма занудное занятие, при изменениях в реестре приходится всякий раз перезапускать компьютер, чтобы убедиться, что все правильно работает.

Форму уменьшим и придадим ее свойству `BorderStyle` значение `bsDialog`. В заголовке формы напишем **Установка LangSwitch**. Установим на форму два компонента `CheckBox` (для фиксации состояний **Удалить системный индикатор раскладки** и **Внести LangSwitch в автозапуск**) и две кнопки `Button` (**Продолжить** и **Прервать**). У верхнего `CheckBox1` свойство `Enable` следует установить в `False`, у нижнего `CheckBox2` — в `True`. Оба компонента `CheckBox` отметим по умолчанию (`Checked=True`). После заполнения всех заголовков форма должна выглядеть так, как показано на рис. 17.1.

Список глобальных переменных будет таким:

```
var
  Form1: TForm1;
  IniFile:TIniFile;
  Reg:TRegistry;
```

```
  st:string;
  sf:TsearchRec;
  FlagLay:boolean=False;
  FlagRStrt:boolean=False;
  FlagDel:boolean=False;
. . . . .
```

Рис. 17.1. Окно Setup для программы LangSwitch

В предложение **uses** внесем модули IniFiles, Registry и ShellAPI и присвоим проекту иконку самой программы LangSwitch (она находится в папке Glava7\2 в файле под названием Keyboard.ico). Далее создадим такой обработчик onCreate формы:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  st:='';
  Reg := TRegistry.Create;
  with Reg do
  begin
   Rootkey:=HKEY_LOCAL_MACHINE;
   if Openkey('SOFTWARE\Microsoft\Windows\CurrentVersion\Run',False) then
   try    {системная переключалка клавиатуры}
     st:=ReadString('internat.exe');
   except
     try
       st:=ReadString('ctfmon.exe');
     except
       CloseKey; {нет ни того, ни другого}
       Free;
       exit;
     end;
   end;
   if st<>'' then CheckBox1.Enabled:=True;
   st:='';
   Rootkey:=HKEY_CURRENT_USER;
```

```
    if Openkey('Keyboard Layout\Toggle',False) then
    try  {пункт Переключать раскладки клавиатуры}
      st:=ReadString('Layout Hotkey');
    except
        CloseKey; {нет такого}
        Free;
        exit;
    end;
    if st<>'' then FlagLay:=True; {есть такой}
    CloseKey;  {подчистили}
    Free;
  end;
end;
```

Здесь мы определяем, есть ли в автозапуске системный переключатель и существует ли параметр Layout Hotkey. Если системный переключатель имеется, то CheckBox1 будет активирован.

Теперь сначала создадим процедуру, которую надо предпринять при нажатии кнопки **Прервать**. Напоминаю, что в этом случае мы должны не просто выйти из программы, но и удалить все ранее скопированные файлы и созданную папку. Если вдруг между началом установки и отменой в папке появились лишние файлы, то мы извинимся:

```
procedure TForm1.Button2Click(Sender: TObject);
begin    {прервать установку}
  Form1.Hide;
  ChDir(ExtractFilePath(ParamStr(0)));
  if FindFirst('*.*',faAnyFile,sf)=0 then
  begin
     repeat
      if not FlagDel then
      if
      (ANSIUpperCase(sf.Name)<>ANSIUpperCase('Langswitch.exe')) and
      (ANSIUpperCase(sf.Name)<>ANSIUpperCase('LangHook.dll')) and
      (ANSIUpperCase(sf.Name)<>ANSIUpperCase('LShelp.htm')) and
      (ANSIUpperCase(sf.Name)<>ANSIUpperCase('Setup.exe')) then
      begin
{в uninstall убрать setup.exe, добавить еще Langswitch.ini, SETUP.sav,
uninstall.exe}
       st:='В папке '+ExtractFilePath(ParamStr(0))+'  '+#10+
       'найден неизвестный файл '+sf.Name+
       ' Удалить все такие файлы?';
       if Application.MessageBox(Pchar(st),'Ошибка
                          деинсталляции',mb_OKCANCEL)<>idOK
```

```
      then break else FlagDel:=True;
     end;
     DeleteFile(sf.Name);
    until (FindNext(sf)<>0);
    FindClose(sf);
    try
     RmDir(ExtractFilePath(ParamStr(0))); {удаляем весь каталог}
    except
      st:='Невозможно удалить папку'
                        +ExtractFilePath(ParamStr(0))+'  ';
      Application.MessageBox(Pchar(st),'Извините',mb_OK);
    end;
  end;
  Halt;
end;
```

После этого создадим отдельную процедуру, которая будет демонстрировать нам диалог при окончании установки:

```
procedure EndInst;
begin
  Form1.Hide;
  if FlagRStrt then  {нужен перезапуск}
  begin
    st:='Для окончания установки необходим перезапуск компьютера.'
    +#10+' Перезапустить его сейчас?';
    if Application.MessageBox(Pchar(st),'Установка Langswitch
                                  завершена',mb_OKCANCEL)=idOK
    then Win32Check(ExitWindowsEx(ewx_REBOOT,0));
  end else
  if Application.MessageBox('Установка завершена. Показать
                       справку?','Langswitch',mb_OKCANCEL)=idOK
  then ShellExecute(Form1.Handle,'open','LShelp.htm',
                                      nil,nil,SW_SHOWNORMAL);
  {запускаем саму программу:}
  ShellExecute(Form1.Handle, 'open', 'Langswitch.exe',
                                    nil, nil, SW_SHOWNORMAL);
  DeleteFile('setup.exe');
  Form1.Close;
end;
```

Флаг `FlagRStrt` мы будем устанавливать в зависимости от того, внесли мы изменения в реестр или нет. Точнее, мы не будем его устанавливать в том случае, если в реестр внесены *только* изменения, касающиеся добавления

нашей программы в автозапуск. И наконец, вот так будет выглядеть процедура самой инсталляции:

```
procedure TForm1.Button1Click(Sender: TObject);
begin {продолжить инсталляцию}
  Form1.Hide;
  if (CheckBox1.Enabled=True) and (CheckBox1.Checked) then
  begin {будем убирать системный переключатель}
    FlagRStrt:=True; {придется перезапускать}
    Reg := TRegistry.Create;
    with Reg do
    begin
     Rootkey:=HKEY_LOCAL_MACHINE;
     if Openkey(
     'Software\Microsoft\Windows\CurrentVersion\Run',False) then
     try    {системная переключалка клавиатуры}
       st:=ReadString('internat.exe');
     except
       try
         st:=ReadString('ctfmon.exe');
       except
       end;
     end;
     if DeleteValue(st) then
     {если успешно удалили, то запоминаем параметр}
     begin
      IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.sav'));
      IniFile.WriteString('Run','System',st);
      IniFile.Destroy;
     end;
     CloseKey; {подчистили}
     Free;
   end;
  end; {убрали системный переключатель}

  if FlagLay=True then
  begin {убираем пункт Переключать раскладки клавиатуры}
    FlagRStrt:=True; {придется перезапускать}
    Reg := TRegistry.Create;
    with Reg do
    begin
      Rootkey:=HKEY_CURRENT_USER;
```

```
    if Openkey('Keyboard Layout\Toggle',False) then
    try  {пункт Переключать раскладки клавиатуры}
     st:=ReadString('Layout Hotkey'); {запоминаем параметр}
     IniFile:=TIniFile.Create(ChangeFileExt(ParamStr(0),'.sav'));
     IniFile.WriteString('Layout','Hotkey',st);
     IniFile.Destroy;
     WriteString('Layout Hotkey','3'); {успешно изменили}
    except
    end;
    CloseKey;  {подчистили}
    Free;
     st:='Будет отключен пункт "Переключать раскладки
         клавиатуры".'+#10+'Подробности см. в справке Langswitch';
     Application.MessageBox(Pchar(st),'Внимание!',mb_OK);
  end;
end; {убрали пункт Переключать раскладки клавиатуры}

if CheckBox2.Checked then
begin {вносим нашу программу в автозапуск}
  Reg := TRegistry.Create;
  with Reg do
  begin
   Rootkey:=HKEY_LOCAL_MACHINE;
   if Openkey(
   'Software\Microsoft\Windows\CurrentVersion\Run',False)
   then
   try
     st:=ExtractFilePath(Application.ExeName)+'Langswitch.exe';
     WriteString('Langswitch.exe',st);
   except
   end;
    CloseKey;  {подчистили}
    Free;
  end;
end;  {внесли программу в автозапуск}

{вносим нашу программу в меню Установка-Удаление:}
Reg := TRegistry.Create;
with Reg do
begin
   Rootkey:=HKEY_LOCAL_MACHINE;
   if   Openkey(
   'Software\Microsoft\Windows\CurrentVersion\Uninstall',
   False) then
```

```
      begin
        if CreateKey('Langswitch') then
        begin
          WriteString('DisplayName','LangSwitch');
          st:=ExtractFilePath(Application.ExeName)+'uninstall.exe';
          WriteString('UninstallString',st);
        end;
      end;
      CloseKey;  {подчистили}
      Free;
    end;
  EndInst; {диалог окончания}
end;
```

Как видите, писать инсталляционные программы — дело довольно муторное. Но это только первый из трех необходимых этапов. Второй этап — создание пока отсутствующей у нас uninstall.exe. Ее проект под тем же названием я поместил в отдельную папку (Glava17\Uninstall). Создадим там точно такую же форму и поместим на нее две кнопки и компонент Label. По умолчанию на кнопках будут заголовки **Удалить** и **Не удалять**, а Label1 мы будем заполнять динамически, т. к. нам понадобится многострочный вывод (рис. 17.2).

Для этой программы все будет несколько проще, вот так будет выглядеть процедура onCreate:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Label1.Caption:='Вы действительно хотите удалить'+#10+'программу
                    LangSwitch?';
end;
```

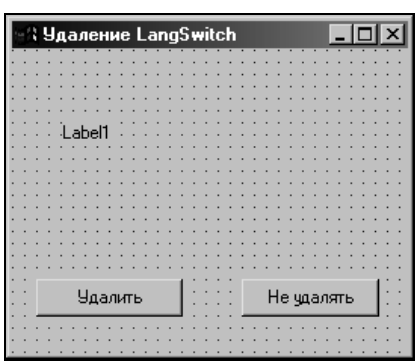


Рис. 17.2. Форма Uninstall для программы LangSwitch

В этой программе у нас будут предусмотрены два диалога для выбора по желанию пользователя — удалять ли программу вообще и перезагружать ли компьютер после удаления. Но процедура отмены в обоих случаях будет выглядеть одинаково просто:

```
procedure TForm1.Button2Click(Sender: TObject);
begin {не удалять и не перезагружать}
  Form1.Close;
end;
```

Тогда основная процедура по нажатию кнопки Button1 будет такой (в ней выбор между действиями осуществляется в зависимости от ее заголовка, после внесения всех изменений он поменяется на **Перезагрузить**):

```
procedure TForm1.Button1Click(Sender: TObject);
begin {удалить или перезагрузить}
  if Button1.Caption='Удалить' then
  begin
    IniFile:=TIniFile.Create('SETUP.sav');
    Reg := TRegistry.Create;
    Form1.Hide;
  {удаление из меню Установка-Удаление:}
    with Reg do
    begin
       Rootkey:=HKEY_LOCAL_MACHINE;
       if Openkey(
      'Software\Microsoft\Windows\CurrentVersion\Uninstall',False)
       then
       DeleteKey('Langswitch');
       CloseKey;  {подчистили}
    end;
  {возврат пункта Переключать раскладки клавиатуры:}
    st:=IniFile.ReadString('Layout','Hotkey','');
    if st<>'' then
    begin
      with Reg do
      begin
        Rootkey:=HKEY_CURRENT_USER;
        if Openkey('Keyboard Layout\Toggle',False) then
        try
           WriteString('Layout Hotkey',st); {восстановили}
        except
        end;
```

```
      CloseKey;  {подчистили}
    end;
  end;
{возврат системного переключателя:}
  st:=IniFile.ReadString('Run','System','');
  IniFile.Destroy; {больше не понадобится}
  if (st='internat.exe') or (st='ctfmon.exe') then
  begin
    Reg := TRegistry.Create;
    with Reg do
    begin
      Rootkey:=HKEY_LOCAL_MACHINE;
      if Openkey(
     'Software\Microsoft\Windows\CurrentVersion\Run',False)
      then
      try
         st:=ExtractFilePath(Application.ExeName)+st;
         WriteString(ExtractFileName(st),st);
      except
      end;
      CloseKey;  {подчистили}
    end;
  end;
{удаление из автозапуска:}
  with Reg do
  begin
    Rootkey:=HKEY_LOCAL_MACHINE;
    if Openkey(
   'Software\Microsoft\Windows\CurrentVersion\Run',False) then
    DeleteValue('Langswitch.exe');
    CloseKey;  {подчистили}
    Free;
  end;
 {удаление папки:}
  ChDir(ExtractFilePath(ParamStr(0)));
  if FindFirst('*.*',faAnyFile,sf)=0 then
  begin
   repeat
    if not FlagDel then
    if
    (ANSIUpperCase(sf.Name)<>ANSIUpperCase('Langswitch.exe')) and
    (ANSIUpperCase(sf.Name)<>ANSIUpperCase('LangHook.dll')) and
    (ANSIUpperCase(sf.Name)<>ANSIUpperCase('LShelp.htm')) and
```

```
      (ANSIUpperCase(sf.Name)<>ANSIUpperCase('Langswitch.ini')) and
      (ANSIUpperCase(sf.Name)<>ANSIUpperCase('SETUP.sav')) and
      (ANSIUpperCase(sf.Name)<>ANSIUpperCase('uninstall.exe')) then
      begin
        st:='В папке '+ExtractFilePath(ParamStr(0))+'  '+#10+
        'найден неизвестный файл '+sf.Name+
        ' Удалить все такие файлы?';
        if Application.MessageBox(Pchar(st),
          'Ошибка деинсталляции',mb_OKCANCEL)<>idOK
        then break else FlagDel:=True;
      end;
      DeleteFile(sf.Name);
    until (FindNext(sf)<>0);
    FindClose(sf);
    try
     RmDir(ExtractFilePath(ParamStr(0))); {удаляем весь каталог}
    except
      st:='Невозможно удалить папку '
      +ExtractFilePath(ParamStr(0))+'  ';
      Application.MessageBox(Pchar(st),'Извините',mb_OK);
    end;
  end; {удалили папку}
  Form1.Show;
  Label1.Caption:='Для полного удаления программы'+#10+
                  'LangSwitch необходимо'+#10+
                  'перезагрузить компьютер.'+#10+
                  'Сделать это сейчас?';
  Button1.Caption:='Перезагрузить';
  Button2.Caption:='Не перезагружать';
  exit;
 end; {Удалить}
 if Button1.Caption='Перезагрузить' then
    Win32Check(ExitWindowsEx(ewx_REBOOT,0));
end;
```

Остался третий этап — создание самораспаковывающегося архива. Я не буду в подробностях объяснять, как это делается — все отлично и доходчиво объяснено в справке программы WinRar. В архив нам надо собрать следующие файлы:

- Langswitch.exe
- LangHook.dll
- LShelp.htm

- Setup.exe
- Uninstall.exe

Поля, которые нужно заполнить в процессе создания этого архива, такие:

```
Title=Программа LangSwitch
Text
{
Если у вас уже установлен какой-либо переключатель языковой раскладки,
кроме системного переключателя Windows, удалите его перед установкой
LangSwitch.

Программа LangSwitch предназначена для переключения языковой раскладки
клавиатуры с русского языка на английский и обратно. Программа дополняет
работу системного переключателя, использующего сочетания клавиш Alt-Shift
или Ctrl-Shift, тем, что позволяет задавать в качестве переключателя
любую одиночную клавишу на клавиатуре. Работа программы основана на
имитации нажатия установленной системной комбинации клавиш. Программа
тестировалась в операционных системах Windows 98 и Windows XP.

Дополнительно программа LangSwitch также позволяет отключать функцию
клавиши Alt, как входа в меню Windows.

В процессе установки потребуется внесение изменений в системный реестр и
последующий перезапуск Windows. Для удаления установленной программы
LangSwitch обратитесь к пункту "Установка и удаление программ" в "Панели
управления".

Пользователь вправе дизассемблировать, модифицировать и распространять
программу LangSwitch на любых условиях при указании имени автора этой
программы в качестве первоисточника. За возможный вред, который может
быть нанесен пользователю в процессе инсталляции, работы и деинсталляции
программы LangSwitch, ответственность несет корпорация Microsoft, которая
не удосужилась сделать нормальный программный интерфейс для функции
переключения языковой раскладки. За потери времени на инсталляцию и
деинсталляцию программы LangSwitch в случае признания ее бесполезной, а
также за вред, который может быть нанесен пользователем самому себе при
дизассемблировании, модификации и распространении указанной программы,
автор ответственности не несет.

Ревич Ю.В. <revich@homepc.ru>
}
Path=LangSwitch
Overwrite=1
Setup=setup.exe
```

Сам архив я намеренно не разместил на диске — если читатель собирается его сделать, то сначала необходимо более тщательно оттестировать обе программы Setup и Uninstall. Я тестировал их только для двух имеющихся у меня ОС (русские версии Windows XP Home Edition и Windows 98 SP2 — другой возможности у меня не было), но не могу исключить, что, несмотря на вы-

полнение обеих программ в соответствии с рекомендациями MSDN, в какой-нибудь, например, Windows 2000, или английской версии Windows XP, что-то может пойти наперекосяк. И если для обычных программ это не так страшно, то для программы Setup это очень критичный момент, что является еще одним лишним поводом для того, чтобы по мере возможности избегать создания инсталляторов вообще. Идеальный вариант — разместить программу в виде двух архивов — самораспаковывающегося с инсталлятором и простого ZIP, именно в расчете на два варианта и составлен модифицированный текст справки (файл LShelp.htm в папке Glava17\Setup).

Как видите, все отлично можно сделать самостоятельно, не прибегая к специальным программам. Эти специальные программы могут вам помочь только в одном — не придется писать так много кода. Зато придется изучать конкретную программу, и в самом главном — построении логики работы инсталляции вам никто помочь все равно не сможет.

ГЛАВА 18

Читаем документы Word

Технология OLE Automation

> Идентификатором COM-класса примем CLSID, известный также как ProgID. ProgID — это понятный человеку синоним CLSID. Теперь понятно, почему ActiveX-технологии, которые по сути являются OLE-технологиями, поскольку термин OLE в настоящее время неприменим в данном контексте, считаются очень сложной для понимания темой. Во многих, если не в большинстве случаев, лучшего и желать нельзя!
>
> *Д. Чеппел,*
> *"Технологии ActiveX и OLE" — Microsoft Press, 1997*

> Когда не знаешь, что именно ты делаешь — делай это тщательно.
>
> *Правило для лаборантов*

OLE означает Object Linking and Embedding (связывание и внедрение объектов). OLE Automation — это часть технологии OLE, которая отвечает за интеграцию приложений (см. мечты Джефа Раскина в *главе 1*). COM — это не то, что COM-порт, а вовсе даже Component Object Model. Разница между OLE Automation и COM в том, что вторая появилась позднее и является более продвинутой версией, позволяющей, в том числе, взаимодействующим приложениям находиться на разных компьютерах. Для простоты будем считать, что для наших задач между ними никакой разницы нет. Модуль, отвечающий за объекты OLE Automation, в Delphi носит говорящее название ComObj. Мы будем говорить OLE, подразумевая OLE Automation, а термин "COM" мы с этого момента вообще вслух постараемся не произносить, чтобы не путаться.

Кстати, ActiveX из той же серии, только компоненты ActiveX обычно представляют собой специально сделанные программки (альтернативу Java). Из компонентов ActiveX фактически "слеплен" Internet Explorer, наш любимый `WebBrowser` есть не что иное, как вызов этих компонентов. Компоненты ActiveX мы в этой книге не рассматриваем, во-первых, потому что с ними

куда проще работать через Visual Basic, во-вторых, потому что по мере возможности лучше вообще с ними не работать из-за "глючности", неповоротливости и крайней негибкости в использовании. Впрочем, OLE Automation, как мы увидим, не сильно отличается в этом смысле от компонентов ActiveX.

Заметки на полях

Хотелось бы подчеркнуть разницу между OLE/COM/ActiveX и обычным API-интерфейсом. OLE-объект — не функция или процедура, а то приложение, к которому вы обращаетесь, целиком, со всеми его достоинствами и недостатками. Мы хорошо это видели на примере неповоротливого и плохо управляемого, но в остальном вполне работоспособного `WebBrowser`. Поэтому правильно поставленная задача, которую может решить технология OLE, есть встраивание некоего приложения (MS Word, MS Excel, Acrobat Reader, Internet Explorer и т. п.) в вашу программу, или просто вызов другой программы (не функции!) для совершения неких действий. Причем, в отличие от вызовов API, это можно сделать и с удаленного компьютера через Сеть. Описаний задач подобного рода вы можете найти в Сети сколько угодно (см., например, [24]). Позже мы попробуем сделать немного иначе и выяснить, насколько можно продвинуться в использовании OLE-объектов — подобно функциям API — для выполнения процедур, которые мы сами делать не умеем, с помощью вызова приложения, которое таким умением обладает, но стараясь сам факт вызова скрыть. Как мы увидим, это удастся нам только отчасти.

Через OLE можно в вашем приложении сделать все, что умеет приложение, которое имеет Automation Server. Для этого нужно сделать так, чтобы ваше приложение стало Automation Container. Эти устрашающие термины не должны вас пугать — вы просто создаете у себя объект с определенным названием с помощью вызова функции `CreateOleObject` и им манипулируете. И, главное, не забываете его потом уничтожить, причем в этом деле есть нюансы: вы, конечно, можете просто уничтожить данный объект у себя в приложении, как обычно, но он в системе от этого не исчезнет. Поэтому уничтожать его нужно дважды — сначала подходящим методом (типа `Close` или `Quit`) самого объекта его закрывают, а потом уже уничтожают ссылку на него в вашей программе через вызов функции `Unassigned`, причем последнее даже необязательно (подобно тому, как необязательно уничтожать ассоциацию файловой переменной с конкретным именем файла, но обязательно его закрыть). Если же вы забудете его уничтожить, то последствия могут быть довольно печальными, причем учтите, что если при отладке программы из среды Delphi ее придется прервать из-за ошибок, то запущенный объект останется "висеть", и его придется удалять за списка запущенных приложений системными средствами (через <Ctrl>+<Alt>+<Del>). Разумеется, при всех этих манипуляциях и само приложение, и классы его Automation Server должны быть установлены в системе, иначе ничего не выйдет.

После того как вы создали OLE-объект, про Delphi можете почти забыть. Методы и свойства объекта теперь нужно вызывать из набора его свойств, и

Delphi к этому касательства никакого иметь не будет. Причем вы не только не сможете получить привычную подсказку относительно свойства или метода, поставив точку после названия объекта — компилятор вообще не будет осуществлять проверку, что вы там такое понаписали. И даже обработку ошибок через механизм исключений `try...except` вам осуществить удастся не всегда. Это, конечно, плохо, потому что те объекты, с которыми нам придется иметь дело (объекты MS Office), имеют отвратительный механизм обработки ошибок. Однако это положение действует не всегда, а только лишь в случае "позднего связывания" объектов, на чем мы остановимся позже.

Далее мы будем говорить исключительно об объектах MS Office. Из всех задач, которые могут быть решены через OLE, вызов функций Office есть одна из самых распространенных, например, The Bat! именно таким способом осуществляет проверку орфографии. Нам проверку орфографии осуществлять не надо, мы и так грамотные, а вот прочесть текст документа в формате DOC или RTF нам бы очень хотелось. И вообще-то это может быть осуществлено, как минимум, тремя разными путями.

"Официальный" путь — воспользоваться компонентами Delphi с закладки **Servers**. Есть множество задач, где эти компоненты использовать удобнее — хотя бы из-за наличия проверки синтаксиса программы. Управление ими сложнее, чем просто OLE-объектом, например, методы VBA и WordBasic могут вызываться с произвольным числом параметров, а Delphi-процедуры, в которые они превращаются, — нет. Есть и другие вещи, которые к нашему случаю касательства не имеют (вроде метода `Connect`). На сайте Borland выложен Help по этим компонентам для пятой версии (**ftp://ftp.borland.com/pub/delphi/techpubs/delphi5/d5ms97.zip**), но, как обычно, бестолковый и малоинформативный.

Практически тот же самый случай — когда создать компоненты-серверы предлагается самостоятельно через импорт TLB-библиотек MS Word. Тогда также на этапе компиляции осуществляется контроль параметров и вызовов методов. Еще одно преимущество обоих этих способов, носящих еще название "раннее связывание" — создание OLE-серверов выполняется несколько быстрее, чем при динамическом обращении к ним по ходу выполнения программы ("позднее связывание"). Кроме того, при динамическом связывании будут действовать не все методы объектов. Но ведь действовать "официально" мы не любим и, раз так можно, будем делать как проще — самостоятельно.

Для позднего связывания предлагается два варианта объектов. Сначала — еще в начале 90-х годов — Microsoft создала специальный диалект своего любимого языка Basic под названием Word Basic, специально "заточенный" под программирование макросов для Word и доступа к OLE-серверам. Работа с ним достаточно проста, функций там на порядок меньше, чем в VBA, и все

напоминает наш любимый Pascal — в такой же мере, в какой его напоминает сам Basic. Единственное, но очень большое "но" в этом деле заключается в том, что справку по Word Basic в наши дни получить непросто: она входила в комплект Word 6.0 и Office 95, а потом из официальных источников исчезла, осталась только упомянутая далее официальная таблица соответствия. Находится эта справка (для Word 6.0) в файле wrdbasic.hlp, и у меня он есть, но положить на диск к этой книге я его не могу, т. к. хотя и Word 6.0 давно уже не поддерживается, права на распространение я не имею. Поэтому мы сделаем так: я сейчас вам покажу, как можно действовать через Word Basic для узкой задачи чтения и преобразования файлов в формате DOC и RTF, а далее мы будем действовать "по правилам" и манипулировать с объектами через VBA.

Работа с Word через объект *Word Basic*

Создадим новый проект (Glava18\1) под названием WordTxt, поместим на форму компонент `Memo`, две кнопки `Button` (**Открыть документ Word** и **Сохранить текст**), а также диалог `OpenDialog`, у которого в свойстве `Filter` установим маску `Документы DOC и RTF|*.doc; *.rtf`. В предложение **uses** надо вставить упомянутый ранее модуль ComObj, а в секции **var** объявить переменную `WordApp: OLEVariant`. Для большей широты охвата темы я также покажу, как проверять наличие OLE-объекта `Word` в системе через реестр, для чего добавим в **uses** еще и ссылку на модуль Registry.

Функция проверки тогда будет выглядеть так:

```
function WordInstalled: Boolean;
var {установлен ли Word - через реестр}
Reg: TRegistry;
begin
Reg:=TRegistry.Create;
with Reg do
begin
 RootKey := HKEY_CLASSES_ROOT;
 Result := KeyExists('Word.Application');
 Free;
end;
end;
```

А процедуру создания формы при запуске приложения тогда сделаем такую:

```
procedure TForm1.FormCreate (Sender: TObject);
begin
if not WordInstalled then {если не установлен}
```

```
begin
 Application.MessageBox('Текстовые файлы DOC и RTF не могут быть
прочитаны','MS Word не установлен',mb_OK);
 exit; {на выход}
end;
end;
```

Наконец, главная процедура создания объекта Word, открытие выбранного файла и чтение из него текста:

```
procedure TForm1.Button1Click(Sender: TObject);
var  st:string;
begin {открываем документ, как объект Word Basic,
читаем текст в строку и выводим в Memo}
Memo1.Lines.Clear;
if OpenDialog1.Execute then
begin
 Form1.Caption := ExtractFileName(OpenDialog1.FileName);
 WordApp := CreateOleObject('Word.Basic');
 if not VarIsEmpty(WordApp) then
 begin
  WordApp.AppHide;
  WordApp.FileOpen(OpenDialog1.FileName,ConfirmConversions:=0);
  WordApp.EditSelectAll;
  st:=WordApp.Selection;
  Memo1.Text:=st;
 end
 else ShowMessage('MS Word не найден');
end;
end;
```

Обратите внимание, что при выполнении этой процедуры сам объект мы не показываем (WordApp.AppHide). Параметр ConfirmConversions определяет, подтверждать ли преобразование при открытии, нам этого не надо, потому мы задаем его равным 0. Правда, если при открытии возникнет ошибка, то мы получим обычное сообщение Word, но бороться с этим мы пока не будем — здесь это, как вы увидите, еще не самое страшное. Обратите внимание, что, как и говорилось ранее, функции объекта Word допускают вызов с переменным числом параметров — обязательно одно только имя файла, а на самом деле там могут быть еще много параметров, кроме ConfirmConversions. То же самое относится и к вызову функций VBA (см. далее).

По выполнении этой процедуры в Memo возникнет неразрывный текст в виде одной строки с вкраплениями черных прямоугольничков — Word, как уже говорилось, использует в качестве конца абзаца только один знак <CR> (#13),

а свойство Memo.Text переводы строки не признает и просто выводит его, как "неопознанный символ".

Далее мы хотим при нажатии кнопки Button2 сохранить открытый файл в другом формате. Я приведу прямо в процедуре всю таблицу доступных форматов в виде комментария, и для примера сохраню файл в виде чистого текста с концами строк:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
{Format: Value:
0 Normal (Word format)
1 Document Template
2 Text Only (extended characters saved in ANSI character set)
3 Text+Breaks (plain text with line breaks; extended characters
 saved in ANSI character set)
4 Text Only (DOS) (extended characters saved in IBM PC character
 set)
5 Text+Breaks (DOS) (text with line breaks; extended characters saved
 in IBM PC character set)
6 Rich-text format (RTF)}

{сохраняем, как текст с концами строк:}
 if not VarIsEmpty(WordApp) then
 begin
  WordApp.FileSaveAs(ChangeFileExt(OpenDialog1.FileName,'.txt')
                  ,Format:=3);
  WordApp.AppClose;
  WordApp:=Unassigned;:
  Form1.Caption := 'NS Word';
 end
 else ShowMessage('Текст MS Word не найден');
end;
```

Наконец, если мы захотим выйти без сохранения, то напишем процедуру закрытия Word при выходе из программы:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin {при закрытии уничтожаем, если есть}
 if not VarIsEmpty(WordApp) then
 begin
  WordApp.AppClose;
  WordApp:=Unassigned;
 end;
end;
```

Пример чисто демонстрационный, потому что в нем есть баг — нельзя без сохранения файла или без перезапуска программы еще раз нажать на кнопку **Загрузить**, потому при этом создастся еще один `Word` под тем же именем объекта. Далее в более "солидном" примере с использованием VBA мы от этого избавимся. Важнее, что при вызове метода `AppClose` нас подстерегают серьезные подводные камни: если файл, который вы открывали, менялся по ходу дела, а вы его не сохранили, то получите по полной программе: возникнет диалог "Сохранить изменения в документе...", как будто вы с настоящим Word работаете (что на самом деле и происходит). Но здесь мы ничего такого не делаем, поэтому гораздо хуже, что при открытии файла, например, в формате Word 6.0, вы получите при закрытии неотключаемое предложение сохранить его в нормальном формате. И пути, как бороться с этим недостатком, в рамках Word Basic я не нашел, что, если вдуматься, естественно — у Word 6.0 такого диалога вообще не было. Поэтому, как мы видим, сохранить DOC-файл в ином формате таким способом вполне возможно, а для простого чтения попробуем пойти по более "правильному" пути.

Работа с Word через объект *VBA*

К концу 90-х годов был разработан диалект Visual Basic под названием VBA (Visual Basic for Appilications), который перекрыл в том числе и все функции Word Basic. Именно перекрыл, а не включил в себя, чтобы установить соответствие, следует воспользоваться справкой по Visual Basic, которая устанавливается вместе с MS Office (по умолчанию она отсутствует, если ее нет, то следует ее специально установить). Справка эта для Office 97 находится в папке ..\Microsoft Office\Office и называется vbawrd8.hlp, а для последующих версий она уже делалась в формате CHM и раскидана по разным файлам (большинство из которых для Windows XP находится в папке ..\Program Files\Common Files\Microsoft Shared\VBA), так что удобнее ее вызывать изнутри самого Word. Любопытно, что для Office 97 справка была частично переведена на русский, а в последующих версиях это сделать поленились, поэтому удобнее раздобыть справку из Office 97 и пользоваться ею — в большинстве необходимых функций она ничем не отличается от более поздних версий. Можно и воспользоваться услугами сайта MSDN, где (на английском, естественно) воспроизведен тот же текст, что прилагается к Word XP. В этой справке, в частности, есть раздел "Visual Basic Equivalents for WordBasic Commands", в котором и устанавливается соответствие между функциями и свойствами старого Word Basic и нового VBA.

Разберемся в "правильном" методе обращения с MS Office-объектами через VBA. Так как справка по VBA имеется, да еще и частично доступна на русском, все вроде бы просто — за исключением, правда, того, что вам придется

вникать в непростую логику создателей VBA, которые не могли, конечно, создать простых функций для самых очевидных вещей. Например, как вы бы посчитали с точки зрения здравого смысла, должно ли быть у объекта под названием Document свойство Text? Да это самое основное его свойство! — подумаете вы. И ошибетесь — такого свойства у объекта Document нет вообще. На самом деле текст извлечь все же можно. Покажем общий принцип создания объекта типа Word в своей программе методом позднего связывания. Для этого в предложение **uses** надо вставить тот же самый модуль ComObj, и так же объявить переменную WordApp: OLEVariant. Основная процедура создания и уничтожения объекта будет выглядеть так:

```
try
 WordApp := CreateOLEObject('Word.Application');
except
 ShowMessage := 'Word не найден.';
 exit;
end;
WordApp.Visible := False; {не показываем Word}
WordApp.Documents.Open (<filename>); {открыли файл}
. . . . .
{чего-то с ним делаем}
. . . . .
WordApp.Quit; {закрыли объект}
WordApp:=Unassigned; {и уничтожили саму память о нем}
. . . . .
```

Попробуем на этой основе переделать нашу демонстрационную программу WordTxt. Процедуру проверки через реестр оставим без изменений, хотя она в принципе и не нужна, а остальные наши процедуры будут выглядеть так (в комментарии я внес, как и ранее, описание соответствующих констант):

```
procedure TForm1.Button1Click(Sender: TObject);
var  st:string;
  wst:WideString;
begin {открываем документ, как объект VBA,
читаем текст в строку и выводим в Memo}
if not VarIsEmpty(WordApp) then
begin {если компонент уже был - закрываем без сохранения}
  WordApp.Quit(SaveChanges:=0);
  WordApp:=Unassigned;:
end;
Memo1.Lines.Clear;
if OpenDialog1.Execute then
```

```
begin
 Form1.Caption := ExtractFileName(OpenDialog1.FileName);
 WordApp := CreateOleObject('Word.Application');
 if not VarIsEmpty(WordApp) then
 begin
  WordApp.Visible:=False;
  WordApp.Documents.Open(OpenDialog1.FileName,
              ConfirmConversions:=False);
  wst:=WordApp.ActiveDocument.Content;
  st:=wst;
  Memo1.Text:=st;
 end
 else ShowMessage('MS Word не найден');
end;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
{FileFormat. Can be one of the following constants:
wdFormatDocument = 0
wdFormatDOSText = 4
wdFormatDOSTextLineBreaks = 5,
wdFormatRTF = 6
wdFormatTemplate = 1,
wdFormatText = 2,
wdFormatTextLineBreaks = 3
or wdFormatUnicodeText.}

{сохраняем как текст с концами строк:}
 if not VarIsEmpty(WordApp) then
 begin
  WordApp.ActiveDocument.SaveAS
  (ChangeFileExt(OpenDialog1.FileName,'.txt'),FileFormat:=3);
{и просто закрываем:}
  WordApp.Quit;
  WordApp:=Unassigned;:
  Form1.Caption := 'NS Word';
 end
 else ShowMessage('Текст MS Word не найден');
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin {при закрытии уничтожаем, если есть}
{SaveChanges. Can be one of the following constants:
wdDoNotSaveChanges = 0
```

```
wdPromptToSaveChanges = $FFFFFFFE
or wdSaveChanges = $FFFFFFFF}
 if not VarIsEmpty(WordApp) then
 begin
{и закрываем без сохранения}
  WordApp.Quit(SaveChanges:=0);
  WordApp:=Unassigned;
 end;
end;
```

Попробовав поработать с той и другой программой, особенно с Office XP, вы поймете, почему Microsoft рекомендует обращаться в VBA, а не использовать устаревшие функции Word Basic. Как видите, удалось даже легко обойти отсутствие свойства `Text` у `Document`, его вполне заменяет свойство `ActiveDocument.Content`, и выделять ничего не потребовалось. Только на всякий случай пришлось воспользоваться промежуточной "широкой" Unicode-строкой — опыт показал, что так надежнее[1]. Давать объяснения, почему иногда (например, при запуске в отладочном режиме) все отлично работает и без Unicode-строки, а в других случаях конверсия не работает, я не берусь — но это, кстати, точно так же происходит в других подобных случаях, необязательно связанных с серверами Office. Во всех случаях, когда есть подозрение, что читаемая строка содержит двухбайтные символы, следует использовать промежуточное преобразование ее через тип `WideString`.

Но главное, что удается закрыть программу без всяких предупреждений, используя параметр `SaveChanges`. Кстати, приятно еще и то, что процедура открытия файла здесь сама по себе — в отличие от Word Basic — изящно обходит все возможные ошибки: я долго заставлял ее открывать различные файлы, включая даже бинарные, переименованные в DOC или RTF, и все их она исправно открывала, останавливаясь на первом встреченном символе с нулевым кодом, и извлекая текст из всего остального. Так что в принципе мы могли бы ей целенаправленно подсунуть и чисто текстовые файлы — если бы только она еще умела определять кодировку! Не сбило ее даже предложение открыть уже открытый в "настоящем" Word файл — он был обработан, как ни в чем не бывало. Сбой может случиться только в редчайшем случае, когда программе попадается испорченный DOC-файл (у меня такой имеется, и не один, и я их специально подсунул, чтобы посмотреть, что получится). Позже мы постараемся этот момент обойти, используя службу обработки исключений Delphi.

---

[1] Между прочим, "официальные" компоненты вследствие контроля типов не позволят вам приравнять строке свойство `ActiveDocument.Content`, хотя оно, как видите, прекрасно в строку конвертируется (и этому есть примеры в самой справке по VB). Для `TWordDocument` придется воспользоваться свойством `Range.Text` или `Selection.Text`.

Для ускорения процесса иногда рекомендуют отключать проверку орфографии вот так:

```
WordApp.Options.CheckSpellingAsYouType:=False;
WordApp.Options.CheckGrammarAsYouType:=False;
```

Однако в нашем случае это занятие бессмысленное: мы ничего с текстом не делаем, и проверка орфографии все равно не должна работать. Если все же вам придется изобретать что-то в этом роде, то не забудьте, что обратно включать проверку тоже необходимо, иначе придется при первом же входе в Word делать это вручную.

Доработка программы Trace

И тут мы вспомним про нашу программу Trace, которую я не без умысла довел до некоторого логического конца, и только теперь обратился к теме чтения DOC- и RTF-файлов. Можно ли включить функцию чтения этих форматов в нее и что из этого выйдет? Для этого нам придется сначала еще раз обратиться к демонстрационной программе WordTxt и погонять ее под различными версиями Office. Тогда мы увидим, что на Office XP наше указание `WordApp.Visible:=False` действует очень своеобразно: при создании объекта Word XP действительно не виден, но вот в процессе выполнения процедуры загрузки документа он появляется на экране, причем еще и норовит вылезти "on top". Word 97 ведет себя куда скромнее и действительно работает в фоновом режиме, как мы указали. По этой причине довести таким образом Trace до ума во всех вариантах не удастся — все время открытый (и тем более то появляющийся, то пропадающий) в процессе поиска Word нам не нужен совершенно. Так что версия программы Trace, которую мы сделаем, будет работать на поиск DOC- и RTF-файлов только в присутствии Word 97. Вполне вероятно, что нормальная работа возможна и в других отличных от Office XP версиях, но я этого проверить не смог — читатель, несомненно, при желании сделает это без труда.

Для того чтобы сделать все корректно, нам надо как-то узнать, установлен ли Word 97 в системе. Это можно сделать через реестр, причем в двух местах:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\MS Office 97
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\8.0\Word
```

Если эти ключи имеются, то можно предполагать, что в системе установлен именно Word 97. Неясно, правда, что будет, если установить сразу два Office, но это все же ситуация редчайшая — они явно будут мешать друг другу. Если вы все же желаете и такую ситуацию предусмотреть, то можно попробовать вместо объекта `Word.Application` вызывать объект `Word.Application.8` (если

Word только один, то это одно и то же), но поможет ли это — не знаю, не проверял. Мы будем действовать старым способом, причем для проверки воспользуемся вторым ключом — он внушает больше доверия.

Перенесем последний вариант проекта Trace (папка Glava16\1) в новую папку (Glava18\3) и придадим программе для начала новый номер версии 1.20 (исправить надо в двух местах, в проекте и в заголовке формы, автоматически здесь в заголовок ничего не выводится). INI-файл переносить не будем, пусть он создастся заново. Удаляем в модуле poisk.pas из константы `stDefExt` расширения doc и rtf. В дальнейшем, если при первом обращении в этим форматам мы не найдем в системе Word 97, выведем предупреждение и тогда добавим `".doc .rtf"` к строке `stExt`, которая у нас несет текущие значения расширений для исключаемых файлов. В предложение **uses** добавляем модули Registry и ComObj, в **var** — переменную `WordApp: OLEVariant`, и пишем следующую отдельную функцию `DocRtfRead`, которую располагаем выше имеющейся функции `ReadFileFormat`

```
function DocRtfRead(fname:string; var st:string):boolean; stdcall;
var Reg: TRegistry;
  wst:WideString;
begin
 if VarIsEmpty(WordApp) then {если Word еще не создан}
 begin
   Reg:=TRegistry.Create;
   with Reg do
   begin
   RootKey := HKEY_CURRENT_USER;
   if not KeyExists('SOFTWARE\Microsoft\Office\8.0\Word')
   then begin Result:=False; Free; exit; end;
   Free;
   end;
   WordApp := CreateOleObject('Word.Application'); :
         {создаем объект Word}
 end;
 result:=True;
 if not VarIsEmpty(WordApp) then {если он создался}
 begin
  {открываем документ и читаем в строку:}
  WordApp.Visible:=False;
  try
   WordApp.Documents.Open(fname,ConfirmConversions:=False);
  except
   st:='';
```

```
   exit;
  end;
  wst:=WordApp.ActiveDocument.Content;
  st:wst;
  WordApp.Documents.Close(SaveChanges:=0);
            {закрываем без сохранения}
 end
 else result:=False; {если нет, возвращаем ошибку}
end;
```

А ниже в функции ReadFileFormat строку с оператором result:=ReadMapFile заменяем на следующий текст:

```
if (ExtractFileExt(ANSIUpperCase(fname))='.DOC') or
(ExtractFileExt(ANSIUpperCase(fname))='.RTF')
then {если это документы Word}
begin
 if not DocRtfRead(fname,stFile) then
 begin
  st:='В системе не найден MS Word.'+
  #10+'Исключить файлы .doc .rtf из поиска?';
  if Application.MessageBox(Pchar(st),'Ошибка',mb_OKCANCEL)=idOK
  then stExt:='.doc .rtf '+stExt;
  result:=False;
  exit;
 end;
 if stFile='' then begin result:=False; exit; end;
 {кодировку определять не надо:}
  if (ExtractFileExt(ANSIUpperCase(fname))='.DOC')
  then st:=' MS Word DOC'
  else st:=' RTF';
  result:=FindString; {поиск строки}
  exit;
end else result:=ReadMapFile; {в stFile - содержимое файла}
```

При отсутствии Word 97, а также — предположительно — в случае ошибок открытия файла функция DocRtfRead вернет значение False, и программа предложит исключить расширения doc и rtf из поиска на дальнейшее (при закрытии программы это зафиксируется в INI-файле). Отдельно создаем процедуру закрытия Word:

```
procedure CloseWord; {закрытие Word}
begin
 if not VarIsEmpty(WordApp) then {если он вообще есть}
 begin
  {закрываем }
```

```
  try
   WordApp.Quit(SaveChanges:=0);
   WordApp:=Unassigned;
  except
   Application.MessageBox
   ('Не могу закрыть MS Word.','Ошибка',mb_OK);
  end;
 end;
end;
```

Вставляем ее вызов в процедуру поиска SearchFile по окончании основного цикла:

```
. . . . .
FindClose(sf); {конец поиска}
CatDir.Free;
CloseWord; {закрытие Word}
. . . . .
```

В самом цикле мы немного изменим вывод в Label2, теперь мы в него будем выводить полное имя файла, и его придется вставить уже после отсеивания запрещенных файлов:

```
. . . . .
if pos(ExtractFileExt(ANSIUpperCase(fname)),ANSIUpperCase(stExt))<>0 then
continue;
   {если расширение совпадает с запрещенным, то на выход}
Form1.Label2.Caption:=fname;
Application.ProcessMessages; {чтобы все прокрутилось}
. . . . .
```

Запустите программу и проверьте — все отлично ищется, но скорость... Конечно, файлов мы просматриваем намного больше, но назвать ее высокой язык не поворачивается, хотя она все же втрое выше, чем когда мы файлы читали допотопными DOS-методами. Поиск в моей "испытательной" папке (C:\DOK с почти 20 000 файлов) занял примерно 15 минут, сравните с результатами в *главе 14*. Загрузка всех 512 Мбайт физической памяти при этом — на 100% (напомню, что мы читаем файлы не более 500 Кбайт).

Медленная скорость и то, что вся эта система не работает, как надо, с Word XP — еще не единственные ее недостатки. Несмотря на обработку исключений, все сбои Word отследить не удается. Это касается в первую очередь некоторых испорченных DOC-файлов, но, кроме того, также и случаев самопроизвольного "падения" Word в процессе работы. Во втором случае программу можно просто закрыть, а в первом случае Trace "виснет", и ее

приходится принудительно прерывать (через <Ctrl>+<Alt>+<Del>), при этом также следует удалить тем же способом запущенный в качестве сервера экземпляр Winword. Лучше всего найти и поудалять файлы, которые могут вызвать сбои, и именно для облегчения этой задачи мы стали выводить полное имя файла, а не только папку, как ранее. Параллельно (но заранее!) запущенный "обычный" Word чаще всего на работу никакого влияния не оказывает, только загрузить в него файл во время запущенного поиска обычным "ассоциативным" путем (через Проводник) не удастся. Все это я честно отразил в справке (см. папку Glava18\3 на диске). Добавлю, что именно эту версию программы я использовал при составлении указателя-FAQ к данной книге и такой опыт ее использования позволил исправить некоторые отмеченные ранее недостатки.

Заметки на полях

Не думаю, что нам следует сильно упрекать разработчиков этих новомодных технологий за рассмотренные "глюки". Они все делали, как им заказывали, и никто, конечно, не виноват, что мы используем OLE не совсем по назначению — нам вызов приложения и не нужен, нам нужны его функции, а вот этого нам отдельно не предоставляют. Безусловно, все эти технологии очень полезны, когда возникает задача удаленного управления неким приложением, типичный случай вы все видите, когда, например, Acrobat Reader загружается в окне Internet Explorer или другого браузера. А на естественный вопрос: почему бы заодно и не написать нормальный набор API в стандартной форме — функции-то эти уже итак есть? — ответ вы знаете сами: когда компаниями руководят профессиональные маркетологи, то цель деятельности в виде создания некоего продукта подменяется целью заработать на нем как можно больше денег. *Они* считают, что так всем будет лучше — можете попробовать с ними поспорить.

Единственное, что я могу посоветовать для ускорения программы и избавления от всех этих "глюков" — забыть про OLE и читать эти форматы самостоятельно, примерно как мы читали Unicode в *главе 8*. Форматов достаточно много — есть эта самая Unicode-версия, которую мы уже читали в *главе 8*, есть Word 6 "родной" (однобайтная версия), есть Word 6.0/RTF, есть, наконец, просто RTF. Для сведения укажу, как можно надежно отличать форматы DOC от других: "настоящий" DOC-файл всегда начинается с сигнатуры `chr($D0) + chr($CF) + chr($11) + chr($E0) +chr($A1)`, у RTF в тех же пяти первых символах стоит `{\rtf'`. Далее в файле должна обязательно встретиться где-то строка `Word.Document.8` (для Unicode-версии) или `Word.Document.6` (для однобайтной версии Word 6.0).

RTF-формат общедоступен, его описание имеется на сайте MSDN, можно найти в Сети и пример реализации "читалки" на C (она входит в MS Software Development Kit). Описание достаточно запутанное, поэтому, чтобы сделать корректную процедуру чтения самому, придется очень постараться — прав-

да, для приблизительной "читалки" в стиле той, что мы употребляли для Unicode, отсеивать весь мусор и не требуется. Почти ничем от стандартного RTF не отличается и формат Word 6.0/RTF, в котором сохраняются файлы при обращении из более старших версий к пункту **Сохранить как | Word 6.0/95**. Для RTF можно попробовать использовать упоминавшийся прием чтения через невидимый `RichEdit` — вряд ли это будет более "криво", чем вызывать Word для такой цели. Конечно, всегда можно поискать специально предназначенные компоненты или библиотеки, только они обычно платные (см., например, **http://www.xkee.com/business/cz-doc2txt-com/**), и/или требуют дополнительных самодельных модулей, из-за чего слишком сложны в использовании (**http://www.torry.net/vcl/vcltools/text/msconv.zip**). В общем, что-то придумать можно.

ГЛАВА 19

Любительская криптография

Приемы простейшего шифрования и стеганографии

> И я сразу решил, что передо мной примитивный шифр, но притом такой, который незатейливой фантазии моряка должен был показаться совершенно непостижимым.
>
> *Эдгар По, "Золотой жук"*

В заголовок главы вынесено слово "любительская" сразу по двум причинам. Во-первых, потому что профессиональная криптография — специальная дисциплина с использованием самых абстрактных разделов высшей математики и одновременно искусство, которое требует серьезной практической подготовки. Все эти детские забавы с решеткой Кардано[1] в настоящее время преодолеваются одним щелчком мыши. Обычно, когда я слышу призывы "доверять только специалистам", я отношусь к этому с изрядной долей скепсиса — в подавляющем большинстве жизненных ситуаций "специалиста" вполне заменяет здравый смысл, небольшая эрудиция и руки, растущие из нужного места. Но серьезная криптография — одно из немногих исключений из этого правила. Если вам действительно нужно что-то засекретить — не жалейте денег и приглашайте лучших специалистов, не пользуйтесь услугами малоизвестных "контор" и патентованными "ширпотребовскими" решениями. Пример того, какие "ляпы" могут таиться в широко разрекламированных общедоступных криптографических технологиях, приведен далее.

При слове "шифровка" у каждого возникает в голове образ бесстрашной разведчицы Кати за рацией в тылу врага. Штирлиц в известном культовом сериале использовал относительно эффективный ручной (т. е. без применения компьютеров или специальных шифровальных машин) шифр, который

[1] В квадрате вырезаются отверстия таким образом, чтобы при нескольких поворотах они покрывали всю его площадь. В эти отверстия и вписывается текст.

дожил почти до наших дней с незапамятных времен — это шифровка сообщения по тексту книги. В этом шифре сообщение состоит из чисел. В качестве ключа получателю сообщения указывается страница, строка и буква в определенном издании, о котором уславливаются заранее. Начиная с этой буквы, он должен отсчитать количество знаков, равное очередному числу в сообщении, и подставить полученную букву в текст. В шпионских романах разведчиков обычно "брали", находя в книгах наколки булавкой, появлявшиеся при отсчете знаков. Отметим, что подобный шифр в том простейшем виде, как я его изложил, элементарно вскрывается "методом Эдгара По", т. е. частотным анализом текста (см. цитированный в эпиграфе рассказ), поэтому его применяли в комбинации с другими методами.

Ручные шифры, изобретенные еще в средние века, несколько, правда, усовершенствованные, применяли вплоть до 20-х годов XX века. Сейчас это все происходит не так. Ключевым изобретением, положившим начало современной цифровой криптографии, стали "одноразовые блокноты" американского инженера Гилберта Вернама, сотрудника секции безопасности телеграфной связи корпорации AT&T. Вернам ввел в обиход основную операцию цифрового шифрования — поразрядное "исключающее ИЛИ" — и обнаружил, что при длине ключа, большей длины текста, и истинно случайной последовательности символов, составляющих ключ, шифр становится принципиально невскрываемым. При одном, правда, условии — если каждый ключ применять только один раз, потому ключевые последовательности Вернама и были названы "одноразовыми блокнотами". Этот факт объясняется просто: при шифровании ключ складывается с текстом через операцию XOR (см. далее). Если взять два таких текста, преобразованных одинаковым ключом, и сложить их друг с другом той же операцией, то ключ просто-напросто вычтется, и останется сумма битов исходных текстов. Восстановить исходные тексты из такой суммы тоже может быть непросто, но все же принципиально проще, чем из зашифрованного текста. Забавно, что в MS Office средства шифрования допускают как раз именно эту ошибку: если вы что-то зашифруете, сохраните на диске, потом внесете исправления и сохраните в другом месте, то формально разные файлы окажутся зашифрованными одним и тем же ключом.

С математической точки зрения стойкость шифров Вернама основана на том, что в природе, видимо, не существует алгоритма, позволяющего разложить произвольное число на простые множители, хотя математики бьются над этим вопросом уже не одно столетие (никто не доказал, что такого алгоритма не существует). А задача подбора множителей (угадывания ключа методом перебора) является типичной задачей с экспоненциально возрастающей сложностью: уже при длине ключа в 512 бит и выше для такого взлома не хватит вычислительной мощности всех компьютеров мира.

Так что первая причина, по которой я назвал криптографию, которой мы займемся, любительской — то, что никаких таких "стойких алгоритмов" мы применять не будем и самостоятельно что-то тут изобретать нет никакого смысла, т. к. соответствующие средства и без того имеются и вполне доступны любому. В том числе существуют и специальные CryptoAPI для разработки приложений, но это специальная тема, к которой собственно программирование и тем более Delphi отношение имеют лишь косвенное. И на примере MS Word (см. врезку) хорошо видно, почему получается однозначно лучше, когда все эти методы используют специалисты.

Ну, а вторая причина наличия слова "любительское" — то, что любые средства шифрования всегда — за исключением профессиональных надобностей — шутка сама по себе очень на любителя. Прежде чем применять их, задайте себе вопрос — зачем? И когда ответите на него, то примите во внимание еще и следующее соображение. Любое, даже самое стойкое шифрование, не сможет устоять перед двумя вещами: горячим утюгом и безалаберностью пользователя. Это научно — я не смеюсь — зафиксированный факт, в "официальной" формулировке он звучит так: любые средства шифрования хороши, если они применяются в комплексе прочих мер защиты. Если вы приклеили бумажку с паролем на монитор, или используете один и тот же пароль для доступа к порносайту и к своему счету в системе WebMoney, то никакое шифрование вам не поможет. Кроме того, любой детективной конторе сейчас доступны средства электронной разведки — чтобы перехватить нажатия клавиш, программный "клавиатурный шпион", типа, как мы сооружали в *главе 6*, не обязателен, их вполне можно записать дистанционно[2]. Точно так же можно записать и излучение кабеля, соединяющего компьютер и монитор. Для защиты от этого нужно ставить компьютер в экранированное помещение, что само по себе обойдется много дороже любой криптографии. Отметим также, что одной из важнейших задач в этом комплексе является обеспечение обмена ключами так, чтобы их не перехватили по дороге. Ну, и т. д.

С другой стороны, может оказаться, что "стойкое" шифрование не так уж часто и нужно на практике. Конечно, я имею в виду не банковские системы, а лишь нужды обычного пользователя, где определяющим может стать эффект "неуловимого Джо", которого потому никто и не поймал, что он никому не нужен. Если вы применяете средства шифрования в открытую, то сам этот факт может заинтересовать вашу жену, сослуживцев, начальство или даже компетентные органы, пусть вы, к примеру, обсуждаете по служебной электронной почте всего лишь достоинства "девушки месяца" из последнего

[2] Чтобы убедиться, насколько это просто, возьмите телефонную DECT-трубку и попробуйте понабирать что-то на клавиатуре в то время, когда в ней нет никаких маскирующих звуков (пауза в разговоре) — вы услышите отчетливые щелчки.

Playboy. Такова психология человека, как стадного животного. Мораль: скрывать следует, прежде всего, сам факт наличия секретного текста. И это одна из самых эффективных мер в том комплексе защиты, о котором говорилось ранее.

Но прежде, чем мы перейдем к необходимому минимуму общих принципов, которые мы будем использовать для шифрования текстов, несколько слов об одной широко обсуждаемой теме, имеющей прямое касательство к предмету нашего разговора, а именно — к защите программ. Автор этих строк является принципиальным противником каких-либо программных защит, но подробное обсуждение этого вопроса мы оставим за рамками книги. Мнения своего я никому не навязываю и хочу лишь заметить по этому поводу следующее. Если вы уж собрались защищать свою программу, делайте это, по крайней мере, квалифицированно. Нет ничего глупее защиты, которая взламывается, как говорится, "с полпинка" любым юным читателем журнала "Хакер", такая защита на практике никаких последствий, кроме приступа раздражения у пользователей, иметь не будет. Берите пример с компании ABBYY, которая снабдила свои продукты такой защитой, что вряд ли вы сможете купить у пиратов нормально взломанный Fine Reader за пресловутые три доллара. Защита программ — это вопрос хитрости и умело расставленных ловушек. Самый хитрый криптографический код ничего не даст, если он представляет собой единственное препятствие — "крякер" почтет за честь его "распечатать" просто из принципа. Полсотни относительно простых, но разнообразных ловушек, расставленных по всей программе, будут куда эффективнее. С другой стороны, если кому-то это очень понадобится, любой код все равно будет взломан (тот же Fine Reader все же на пиратском рынке имеется, правда, не за три доллара). И об этом никогда забывать не следует. Для иллюстрации приведу историю из жизни.

Однажды у меня в квартире заклинило замок в металлической двери. Все мы оказались снаружи, так что даже сломать его окончательно было невозможно, не ломая дверь. Для любителя без инструментов это нереальная задача — даже пожарные ломают хорошую металлическую дверь не в пять минут (сам был свидетелем такого случая). Пришлось вызывать службу спасения. Удостоверившись в наличии у меня паспорта с пропиской, они споро принялись за дело и не более чем через три с половиной минуты замок был открыт, при этом на двери не осталось ни единой царапины. Единственными инструментами, которые применили спасатели, были три маленьких отвертки: один человек орудовал двумя, а второй помогал ему третьей, держа в другой руке фонарик. Когда после их ухода мне удалось поднять с пола и пристроить на место отвалившуюся нижнюю челюсть, я задал риторический вопрос в пространство: если замок (причем заклинивший, не забудьте) можно открыть за три минуты отвертками из детского

конструктора, какой смысл ставить металлические двери? Ответ простой: от спланированного профессионалами ограбления квартиры никакие замки с металлическими дверями не спасут. Возможно, тут может помочь только бронированная комната-сейф, и то не во всех случаях. Зато от мелких жуликов и простых хулиганов — спасут обязательно, те кто не умеет работать отвертками на уровне упомянутых ранее спасателей поковыряются — и пойдут в другое место. Тут действует простой закон, когда-то сформулированный относительно автомобилей: "если вы не хотите, чтобы вашу машину взломали, поставьте ее в лужу". То есть любая защита должна основываться не на математике, программировании или точной механике, а на *психологии*. Когда вы тщательно обдумаете в этой связи цели, которые вы преследуете установкой защиты на свои программы, саму целесообразность этого мероприятия, и после этого придете к выводу, что все же это требуется, для общего образования советую прочесть статью [37].

Операция XOR и простейшее шифрование файлов

Согласно определению, двухместная однобитовая операция "исключающее ИЛИ" выполняет действия, показанные в табл. 19.1.

Таблица 19.1. *Исключающее ИЛИ(XOR)*

Вход 1	Вход 2	Выход
0	0	0
0	1	1
1	0	1
1	1	0

Ее можно еще назвать "операция несовпадения" — на выходе логическая единица тогда, когда значения на входах не совпадают. Из этого определения легко вывести основное свойство этой операции: будучи применена дважды к одному и тому же операнду, она ничего не изменит, независимо от значения второго операнда, лишь бы он не менялся. Этим, в частности, широко пользуются в компьютерной графике: если приложить через операцию XOR к фону маску, состоящую из всех единиц (чисто белую), то изображение на этом месте инвертируется по цветам, при повторении того же действия все восстанавливается в неизменном виде (так, в частности, удобно производить выделение, см. *главу 10*).

Данное свойство и положено в основу практически всех алгоритмов шифрования, которые иногда могут быть очень навороченными, но мы не будем углубляться в этот вопрос, а просто попробуем применить этот метод в его изначальном виде. Для проверки можно использовать такую простейшую процедуру. Пусть мы хотим зашифровать некий текстовый файл. Так как мы дипломатические секреты не прячем, то нам хватит обычной в таких случаях длины ключа в 8 символов (64 бита) — этого достаточно, чтобы сделать взлом шифра методом перебора на обычном персональном компьютере задачей "с налета" нерешаемой. Для примера ключом будет служить слово "yvrevich". Создавать такие ключи из своей фамилии на практике ни в коем случае нельзя, нельзя также употреблять любые словарные слова, даты, номера телефонов — еще Вернам показал, что для эффективного шифрования ключ должен быть строго случайным, здесь это делается только в качестве примера (о том, как можно задавать случайный ключ, мы поговорим позже).

Создадим новый проект (в папке Glava19\1) под названием ProbaCrypt, и разместим в той же папке некий файл на пробу. Я взял текст песни "Однажды мир прогнется под нас" из репертуара А. Макаревича (в текстовом же формате, файл mashinavremeni.txt) с аккордами, т. к. достаточно сложное форматирование этого текста сделает пример нагляднее[3].

Поместим на форму компонент `Memo`, диалог `OpenDialog` и две кнопки `Button`. В заголовке `Button1` напишем **Зашифровать**, в заголовке `Button1` — **Расшифровать**. Объявим такие переменные:

```
var
Form1: TForm1;
fname,key:string;
fi,fo:file of byte;
i:integer;
xb:byte;
. . . . .
```

При создании формы инициализируем ключ и диалог:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 key:='yvrevich';
 OpenDialog1.InitialDir:=ExtractFileDir(Application.ExeName);
end;
```

[3] Так как текст взят с официального сайта "Машины времени", надеюсь, я ничей копирайт не нарушил? В наше время ни в чем таком нельзя быть уверенным. Кстати, вся орфография ("прогнется-погнется") также полностью на совести авторов сайта.

В обработчике нажатия кнопки Button1 напишем следующий довольно длинный код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin {Зашифровать}
 OpenDialog1.FileName:=''; {очистим}
 OpenDialog1.Filter:='';
 if OpenDialog1.Execute then
 fname:=OpenDialog1.FileName else exit;
 assignfile(fi,fname); {задали исходный файл}
 try
  reset(fi); {открыли исходный}
 except
  exit; {если не открывается - на выход}
 end;
 assignfile(fo,ChangeFileExt(fname,'.sec'));
 rewrite(fo);
 read(fi,xb); {прочли первый байт}
 Form1.Caption:='ProbaCrypt: '+ExtractFileName(fname);
            {название файла - в заголовок}
 Memo1.Lines.Clear;
 Memo1.Lines.Add('Подождите...');
 Application.ProcessMessages; {чтобы увидеть предупреждение}
 Memo1.Lines.Clear;
 while not eof(fi) do
 begin
  for i:=1 to length(key) do
  begin
   xb:=xb xor ord(key[i]); {шифруем}
   Memo1.Text:=Memo1.Text+chr(xb); {выводим в Memo}
   write(fo,xb); {записываем в файл}
   try
    read(fi,xb); {если конец файла - выходим}
   except break; end;
  end;
 end;
 closefile(fi);
 erase(fi); {уничтожаем исходник}
 closefile(fo);
 exit;
{зашифровали}
end;
```

Тут мы складываем через операцию XOR каждый байт исходного файла с байтами ключа по очереди; когда ключ заканчивается, мы опять начинаем с его первого символа. Результаты пишем в файл с расширением sec (от "security") и выводим в Memo1. В точности ту же операцию мы производим при расшифровке, только уже с зашифрованным файлом, в результате чего исходный файл восстанавливается полностью:

```
procedure TForm1.Button2Click(Sender: TObject);
begin {Расшифровать}
 OpenDialog1.FileName:=''; {очистим}
 OpenDialog1.Filter:='Шифрованные файлы|*.sec';
 if OpenDialog1.Execute then
 fname:=OpenDialog1.FileName else exit;
 assignfile(fi,fname); {открыли шифрованный}
 try
  reset(fi);
 except
  exit;
 end;
 assignfile(fo,ChangeFileExt(fname,'.txt'));
 rewrite(fo); {перезаписываем старый}
 read(fi,xb);
 Form1.Caption:='ProbaCrypt: '+ExtractFileName(fname);
 {название файла - в заголовок}
 Memo1.Lines.Clear;
 Memo1.Lines.Add('Подождите...');
 Application.ProcessMessages; {чтобы увидеть предупреждение}
 Memo1.Lines.Clear;
 while not eof(fi) do
 begin  {все без изменений, как выше}
  for i:=1 to length(key) do
  begin
   xb:=xb xor ord(key[i]);
   Memo1.Text:=Memo1.Text+chr(xb);
   write(fo,xb);
   try
    read(fi,xb);
   except break; end;
  end;
 end;
 closefile(fi);
 closefile(fo);
{расшифровали}
end;
```

Обратите внимание, что я зашифрованный файл не уничтожаю — в принципе операцию шифрования можно производить сколько угодно раз над уже зашифрованным текстом, для его расшифровки придется также повторить ее столько же раз, и этот прием часто применяют в "официальных" алгоритмах шифрования (только в данном случае придется зашифрованный файл переименовать вручную или несколько модифицировать программу, в имеющемся виде она вам не позволит открыть один и тот же файл и для расшифровывания и для шифрования). Естественно, процедуру шифрования можно применять к абсолютно любым файлам, не только текстовым.

Есть тонкий момент, связанный с уничтожением оригинала — как известно, при уничтожении дискового файла он не стирается, подобно музыкальной записи на магнитной ленте, а просто в заголовочных структурах FAT место, которое он занимает, помечается, как свободное (примерно также происходит это и в NTFS). Именно с этой особенностью была связана работа DOS-программы Unerase (если кто помнит, что это такое). Поэтому, если вы даже удалите файл из Корзины (удаленный из нашей программы файл в Корзину, правда, итак не попадает), на диске останется его содержание до тех пор, пока туда не будет записано что-то еще. Поэтому для особо параноидальных личностей продвинутые программы шифрования предлагают опцию, при которой файлы после стирания уничтожаются гарантированно — на их место записываются нули. В нашем случае для этого в принципе достаточно не закрывая файл, заполнить его байтами с нулевым значением (или любым другим, но не увеличивая и не уменьшая размер файла), записать это на диск (закрыв файл), а потом уже его уничтожать. Правда, в Windows стопроцентной гарантии, что он запишется в точности на то же место, я дать не могу, поэтому лучше в таких случаях применять все же "официальные" программы, которые, кстати, уничтожают следы исходника не только в той области диска, где он хранился, но и SWAP-файле Windows, если они там остались.

Что касается генерации случайных ключей, то вот один из способов. Показывать реализацию полностью я не буду, так как она проста. Генератор псевдослучайных чисел в Delphi (и не только в Delphi) устроен следующим образом: через переменную `RandSeed` задается начальное число генератора (по умолчанию оно равно 0). Тогда функция `Random` при последовательном обращении к ней *всегда* будет возвращать один и тот же набор чисел, независимо от того, в какой программе и когда мы ее используем. Отсюда и способ — вы устанавливаете в программе такой генератор и на его основе генерируете ключ, например, вот так можно сгенерировать случайный 16-байтный (128-битный) набор символов, который будет зависеть только от величины начального смещения генератора `init`:

```
var
xb:byte;
```

```
init:integer;
st:string;
. . . . .
st:='';
RandSeed:=init;{начальное смещение}
while length(st)<16 do
begin
 xb:=Random(255);
 if xb>31 then st:=st+chr(xb);
end;
```

При этом вы передаете вашему корреспонденту не сам ключ, а величину init. У него будет сгенерирован по идентичной процедуре в точности тот же ключ. Можно придумать, разумеется, и более хитрые механизмы реализации этого метода.

Хочу заметить, чтобы не затемнять суть дела, ранее в программе я использовал традиционное побайтное чтение из дискового файла. Резко ускорить процедуру можно при использовании одного из механизмов предварительного чтения файла в память — file mapping, как в *главе 14*, потокового чтения или любого другого способа организации динамических массивов в памяти *(см. главу 21)*. В примере с использованием стеганографии, к которому мы сейчас приступим, частично положение будет исправлено.

Стеганография на коленке

Вообще говоря, к стеганографии (*греч.* steganos — секрет) относится любой прием, который позволяет скрыть сам факт наличия тайного сообщения: невидимые чернила, микрофотоснимки, условные знаки, тайники, кодовые сообщения в открытых радиопередачах и прочая шпионская мишура — все это есть предмет стеганографии. Легко догадаться, что она гораздо старше криптографии — если сообщение достаточно хорошо спрятано, то его можно и не шифровать. Мы рассмотрим прием стеганографии, который позволяет скрыть текст сообщения в цифровой картинке. Кстати, похожим, только более хитрым способом в картинки вносят цифровую подпись — копирайт автора, который в принципе можно распознать, даже если сканировать репродукцию с такого цифрового оригинала, выполненную полиграфическим методом. Совершенно аналогичным способом можно хранить и скрытый текст (и вообще любую информацию) в аудиофайлах (для беззаголовочных файлов типа WAV это даже проще).

Для этого мы подробнее рассмотрим, как хранится изображение в файле типа BMP *(см. также главу 10)*. Собственно картинка там хранится в виде трехбайтных последовательностей, каждый байт в которой представляет собой

интенсивность одной из трех составляющих модели RGB: красной, зеленой и синей. Если мы изменим всего один младший бит в каждом из этих байтов, визуально картинка не потеряет ровным счетом ничего — изменение каждой из составляющих на 1/256 долю глазом не заметно, даже если эта картинка — "Черный квадрат" Малевича. На практике можно менять и больше битов (до 4-х), но мы ограничимся одним[4]. Разумеется, файл не должен использовать сжатие (и это придется проверять), иначе мы все порушим. В нашем "прямолинейном" способе нам придется только позаботиться о том, чтобы изображение по дороге к адресату не подвергалось никаким преобразованиям типа сжатия или изменения размеров. Размеры изображения в нашем случае должны быть такими, чтобы произведение ширины и длины картинки в пикселах, умноженное на число бит на пиксел, превышало размер текста сообщения, умноженный на 8. Для того чтобы понять, имеется ли в данном файле зашифрованное нами сообщение, будем добавлять к нему в начале свою сигнатуру — например, слово "steganographia". Придется также хранить в сообщении и его длину.

Для того чтобы составить корректный алгоритм, нам придется чуть-чуть углубиться в недра BMP-файла. Согласно описанию формата на сайте MSDN, заголовок собственно файла BMP (`BITMAPFILEHEADER`) занимает всегда ровно 14 байт. Напомним *(см. главу 10)*, что идентифицируется формат по первым двум байтам в этом заголовке (самым первым байтам файла), которые должны составлять комбинацию символов "BM". Следующее за этой сигнатурой поле `bfSise` — четырехбайтное число, которое должно быть равно размеру файла. Кстати, в процессе внесения изменений мы не будем менять размер файла, так что ничего в заголовке не нарушим и его без изменений можно перенести в новый файл. Вот если бы там была контрольная сумма — другое дело, но, к счастью, разработчики формата не стали морочить голову такими сложностями. Все эти поля нам проверять не придется, потому что мы сразу попробуем загрузить исходную картинку на экран, и если ничего не выйдет, то увидим, что это не BMP-файл.

Последние четыре байта заголовка (11—14-й байты, т. е. с номерами 10, 11, 12 и 13, если присвоить первому байту файла номер 0) образуют поле `bfOffBits` — число типа `DWord`, которое содержит важную для нас информацию — начало в файле собственно байтового массива с изображением, в который мы и будем вносить изменения. Но прежде чем перейти к этому, нам надо выяснить еще две вещи: во-первых, сравнить размер данного байтового

[4] Кстати, на этом принципе основан один из "самодеятельных" алгоритмов сжатия — вы просто отбрасываете половину (младшую) битов, и картинка уменьшается по объему файла в два раза, ничего почти не теряя. На самом деле это, конечно, не так — все равно, что уменьшить число воспроизводимых оттенков с 16 миллионов до чуть более 4 тысяч.

массива с размером текста сообщения, чтобы понять, уместится ли оно, во-вторых, проверить на всякий случай, не используется ли в файле сжатие, иначе, как уже говорилось, мы своим вмешательством можем все испортить. Неплохо также проверять, сколько цветов в изображении — если оно в оттенках серого, или просто имеет палитру не меньше 256 цветов (по одному байту, или 8 бит на пиксел), то все наши приемы точно так же будут работать, как и в случае TrueColor. Если же оно содержит меньше цветов, чем 256, то мы признаем этот файл непригодным.

После заголовка собственно файла `BITMAPFILEHEADER` идет структура `BITMAPINFOHEADER`, которая и содержит нужные нам сведения. Первые четыре байта этой структуры мы пропускаем, а в следующих восьми (начиная с байта номер 18, напомню, что мы договорились считать от начала файла с нуля) идет ширина (`biWidth`) и длина (`biHeight`) изображения в пикселах — по четыре байта (одному числу `DWord`) на каждое измерение. Если пропустить еще два байта, то в следующем двухбайтном поле `biBitCount` будет число бит на пиксел (для TrueColor там будет стоять число 24=$18). А после него, начиная с байта номер 30, будет нужное нам двухбайтное поле `biCompressed`: если оно содержит нули, то файл не использует сжатия, в противном случае он для нашей цели непригоден.

После структуры `BITMAPINFOHEADER` и начинается массив пикселов, но фиксированный размер (40 байт) структура эта имеет только для модели TrueColor, в случае меньшего количества цветов в нее входит еще палитра, так что правильно определять начало массива с картинкой нужно по указанному выше полю заголовка `bfOffBits`. В случае True Color в этом поле должно содержаться число 54 ($36, т. е. 14+40) — проверьте!

Я специально приводил отсчет байтов от начала файла, потому что мы не будем заморачивать себе голову всякими структурами, а прочтем байты напрямую из файла по одному. Единственная сложность при этом, которую мы себе позволим, — преобразование формата чисел через указатели, ведь читаем мы побайтно, а с числами нам придется иметь дело, как типа `Word` (двухбайтными), так и типа `DWord` (четырехбайтными). Еще об этом методе читайте также в *главе 21*, а здесь я его просто использую без особых пояснений — все и так будет понятно из текста программы. Так как размеры, сообщения и тем более картинки могут быть весьма значительны, придется ввести еще и нормальное (не побайтное) чтение файла — по причинам, изложенным в той же *главе 21*, автор предпочитает использовать для этой цели нетипизированные указатели. Впрочем, основные операции чтения заголовка BMP и кодирования мы будем делать "по-старинке". До законченного образца мы программу здесь доводить не будем — увидите, что она получится и так весьма "навороченная", и сейчас нам важно показать суть дела на демонстрационном примере.

Итак, создадим новый проект (Glava19\2), назовем его Steganos и поместим в папку с проектом картинку в BMP-формате (okno.bmp — это вид из окна моего домика в деревне). В качестве шифруемого сообщения используем фрагмент работы пламенного большевика Л. Троцкого "Преданная революция"[5] (файл trotzki.txt).

Разместим на форме следующие компоненты: `Image1` на панели `Panel1` (у нее установим свойство `BorderStyle` в `bsSingle` и очистим заголовок), `Memo1` (у которого подключим вертикальную линейку прокрутки: `ScrolBars = ssVertical`) и пять кнопок на двух панелях, которым придадим соответствующие заголовки. У `Image1` установим в `True` свойства `Proportional` и `Stretch`. Кроме этого, поставим на форму диалог открытия файла `OpenDialog1`. В конечном итоге все это должно выглядеть так, как показано на рис. 19.1.

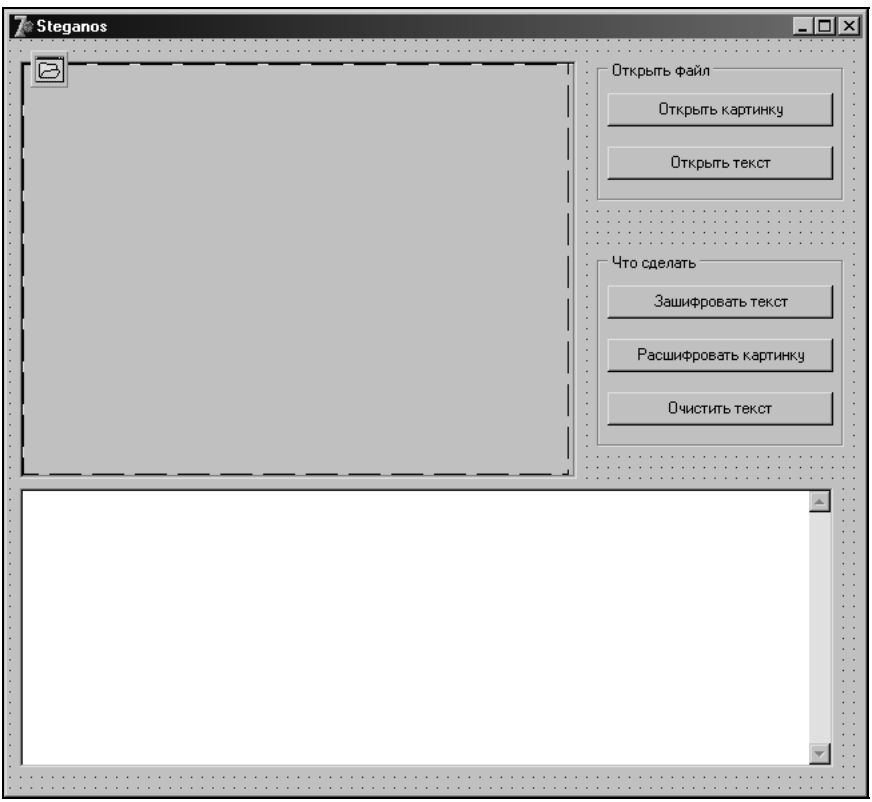


Рис. 19.1. Форма проекта Steganos

[5] Эта намек будущим подпольщикам — работа знаменитого диссидента действительно распространялась в нашей стране во все годы существования Советского государства исключительно "стеганографическим" путем.

У кнопок Button3 (**Зашифровать текст**) и Button4 (**Расшифровать картинку**) свойство Enabled установим в False — пока нет ни текста, ни картинок, расшифровывать и зашифровывать нечего. Текст программы я приведу без особых пояснений, основной алгоритм мы описали ранее, а остальное будет ясно из комментариев в тексте. Объявляем следующие переменные и типы:

```
type
ab=array [0..3] of byte;
wordp=^word;
longp=^dword;
bytep=^byte;
abp=^ab;

var
Form1: TForm1;
fnamep,fnamet,st,sttext:string;
fipic,fopic,ftext:file;
i,j,picsize,textsize,picoffs:integer;
xb,tb,ib:byte;
xw,yw:word;
plong:longp;
pword:wordp;
pab:abp;
pp:pointer;
pb:bytep;
. . . . .
```

При запуске программы инициализируем диалог и переменные:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 OpenDialog1.InitialDir:=ExtractFileDir(Application.ExeName);
 picsize:=0;
 textsize:=0;
end;
```

При нажатии на кнопку **Открыть картинку**:

```
procedure TForm1.Button1Click(Sender: TObject);
begin {открыть картинку}
 OpenDialog1.FileName:=''; {очистим}
 if OpenDialog1.Execute then
 fnamep:=OpenDialog1.FileName else exit;
 try
  Image1.Picture.LoadFromFile(fnamep);
```

```
except
 exit; {если не открывается - на выход}
end;
assignfile(fipic,fnamep); {задали исходный файл с картинкой}
reset(fipic,1);
{проверка:}
New(pab);
seek(fipic,18); {начиная с 18-го байта - размеры}
for i:=0 to 3 do blockread(fipic,pab^[i],1);
plong:=longp(integer(pab));
picsize:=plong^; {ширина}
for i:=0 to 3 do blockread(fipic,pab^[i],1);
plong:=longp(integer(pab));
picsize:=picsize*plong^; {ширина*длину в пикселах}
seek(fipic,28);
{начиная с 28-го байта - бит на пиксел,
          потом компрессия по два байта}
for i:=0 to 3 do blockread(fipic,pab^[i],1);
pword:=wordp(integer(pab));
xw:=pword^; {бит на пиксел}
pword:=wordp(integer(pab)+2);
yw:=pword^; {компрессия}
if (xw<8) or (yw<>0) then
begin
 st:='В файле '+ExtractFileName(fnamep)+' используется сжатие'+
 #10+'изображения или в нем слишком мало цветов.'+#10+
 'Подберите другой BMP-файл.';
 Application.MessageBox(Pchar(st),'Ошибка',mb_OK);
 picsize:=0;
 closefile(fipic);
 exit;
end;
{проверяем размер, если текст уже открыт:}
if textsize<>0 then
begin
if (picsize*xw)<(textsize*8) then
begin
 st:='Файл '+ExtractFileName(fnamep)+
 ' имеет недостаточный размер'+
 #10+'Подберите другой BMP-файл.';
 Application.MessageBox(Pchar(st),'Ошибка',mb_OK);
 picsize:=0;
 Button3.Enabled:=False; {кнопка шифрации недоступна}
```

```
  closefile(fipic);
  exit;
 end else Button3.Enabled:=True; {кнопка шифрации доступна}
 end;
 Button4.Enabled:=True;
    {кнопка дешифрации доступна, когда открыта картинка}
 seek(fipic,10); {начиная с 10-го байта - смещение}
 for i:=0 to 3 do blockread(fipic,pab^[i],1);
 plong:=longp(integer(pab));
 picoffs:=plong^;
   {в picoffs - смещение массива пикселов от начала файла}
 Dispose(pab);
 closefile(fipic);
end;
```

При нажатии на кнопку **Открыть текст**:

```
procedure TForm1.Button2Click(Sender: TObject);
begin  {открыть текст}
 OpenDialog1.FileName:=''; {очистим}
 if OpenDialog1.Execute then
 fnamet:=OpenDialog1.FileName else exit;
 assignfile(ftext,fnamet); {задали исходный файл с текстом}
 reset(ftext,1);
 textsize:=filesize(ftext);
 {проверяем размер, если картинка уже открыта:}
 if picsize<>0 then
 begin
 if (picsize*xw)<(textsize*8) then
 begin
  st:='Файл '+ExtractFileName(fnamet)+
  ' слишком велик для выбранного изображения.'+
  #10+'Подберите другой BMP-файл.';
  Application.MessageBox(Pchar(st),'Ошибка',mb_OK);
  textsize:=0;
  Button3.Enabled:=False; {кнопка шифрации недоступна}
  closefile(ftext);
  exit;
 end else Button3.Enabled:=True; {кнопка шифрации доступна}
 end;
 getmem(pp,textsize);
 blockread(ftext,pp^,textsize,j);{прочтем текст за один прием}
 for i:=0 to textsize-1 do {и переведем в строку}
 begin
  pb:=bytep(integer(pp)+i);
```

```
  sttext:=sttext+chr(pb^);
 end;
 freemem(pp,textsize);
 Memo1.Text:=sttext; {выведем в Memo}
 closefile(ftext);
end;
```

Обратите внимание, что и при открытии картинки и при открытии текста используются процедуры проверки достаточности размера картинки — в зависимости от того, что было открыто раньше.

При нажатии на кнопку **Зашифровать текст**:

```
procedure TForm1.Button3Click(Sender: TObject);
begin  {зашифровать текст}
 assignfile(fipic,fnamep); {исходный файл с картинкой}
 reset(fipic,1);
 ChDir(ExtractFileDir(fnamep));
        {на всякий случай устанавливаем папку}
 assignfile(fopic,'0'+ExtractFileName(fnamep));
                {имя выходного файла}
 rewrite(fopic,1);
 seek(fipic,picoffs); {все до picoffs игнорируем}
 st:=IntToStr(length(sttext));
 while length(st)<10 do st:='0'+st; {числовое поле 10 знаков}
 sttext:='steganographia'+st+sttext; {добавляем сигнатуру
    и размер записи, всего 24 байта заголовок}
 for i:=1 to length(sttext) do {основная процедура}
 begin
  tb:=ord(sttext[i]); {очередной байт текста}
  for j:=0 to 7 do
  begin
   blockread(fipic,ib,1); {очередной байт изображения}
   ib:=ib and $FE; {обнуляем младший бит изображения}
   xb:=tb shr j; {сдвигаем до нужного бита}
   xb:=xb and $01; {обнуляем все, кроме младшего бита}
   ib:=ib or xb; {записываем младший бит}
   blockwrite(fopic,ib,1);
  end;
 end;
 {записываем остаток сразу куском:}
 j:=filesize(fipic)-filepos(fipic);
 getmem(pp,j+1);
  blockread(fipic,pp^,j,i);
  blockwrite(fopic,pp^,i,i);
```

```
 freemem(pp,j+1);
 closefile(fipic);
 closefile(fopic);
 st:='Текст зашифрован в файле '+'0'+ExtractFileName(fnamep);
 Application.MessageBox(Pchar(st),'Все отлично',mb_OK);
end;
```

Расшифровать картинку несколько проще, только проверок больше:

```
procedure TForm1.Button4Click(Sender: TObject);
begin {расшифровать картинку}
 Memo1.Lines.Clear; {очищаем Memo}
 textsize:=0; {как будто текстового файла не было}
 Button3.Enabled:=False; {кнопка шифрации недоступна}
 assignfile(fipic,fnamep); {исходный файл с картинкой}
 reset(fipic,1);
 seek(fipic,picoffs); {все до picoffs игнорируем}
 st:='';
 for i:=1 to 24 do {чтение заголовка}
 begin
  tb:=0;
  for j:=0 to 7 do
  begin
   blockread(fipic,ib,1); {очередной байт изображения}
   ib:=ib and $01;{обнуляем все, кроме младшего бита}
   ib:=ib shl j; {сдвигаем до нужного бита}
   tb:=tb or ib; {записываем младший бит}
  end;
  st:=st+chr(tb); {очередной байт заголовка}
 end;
 if pos('steganographia',st)=0 then {если там нет информации}
 begin
  st:='В файле '+ExtractFileName(fnamep)+' отсутствует текст.';
  Application.MessageBox(Pchar(st),'Ошибка',mb_OK);
  exit;
 end;
 delete(st,1,14);
 j:=StrToIntDef(st,0); {извлекаем длину}
 if j=0 then
 begin
  st:='В файле '+ExtractFileName(fnamep)+
  ' длина сообщения равна 0.';
  Application.MessageBox(Pchar(st),'Ошибка',mb_OK);
  exit;
 end;
```

```
 sttext:='';
 for i:=1 to j do {чтение сообщения}
 begin
  tb:=0;
  for j:=0 to 7 do
  begin
   blockread(fipic,ib,1); {очередной байт изображения}
   ib:=ib and $01;{обнуляем все, кроме младшего бита}
   ib:=ib shl j; {сдвигаем до нужного бита}
   tb:=tb or ib; {записываем младший бит}
  end;
  sttext:=sttext+chr(tb); {очередной байт сообщения}
 end;
 Memo1.Text:=sttext; {выведем в Memo}
 closefile(fipic);
end;
```

Очистить текст может потребоваться, если мы загружаем новую картинку просто для расшифровки (тогда проверка размера не будет выполняться):

```
procedure TForm1.Button5Click(Sender: TObject);
begin {очистить текст}
Memo1.Lines.Clear;
textsize:=0;
Button3.Enabled:=False; {кнопка шифрации недоступна}
end;
```

Если вы рассмотрите новую картинку (в файле 0okno.bmp), то вы не найдете в ней никаких отличий от старой даже под большим увеличением. Тем не менее расшифровка будет исправно выдавать текст Льва Давидовича, как будто он там всегда был.

Никто, естественно, не мешает использовать стеганографию в совокупности с шифрованием, как мы это делали ранее, а также хранить любую другую информацию, кроме текста — скажем картинку в картинке (представляете себе газетную сенсацию: найдены чертежи секретного объекта в фотографии Анны Курниковой!). Для того чтобы довести программу до законченного варианта, которым можно пользоваться повседневно, нужно еще доделать много чего: как минимум, выводить куда-то имена открытых файлов (желательно и их размеры), ввести в диалог открытия нужные фильтры, ввести процедуру сохранения расшифрованного текста в файле, сделать меню, справку и т. п. Возможно, при больших объемах сообщений придется ввести ползунок и/или доработать программу в целях ускорения шифрования. Само по себе такое приложение можно делать весьма разнообразными способами, в зависимости от цели, так что я оставляю его доработку на откуп читателям.

ГЛАВА 20

Последовательные интерфейсы COM и USB

И немного о программах реального времени под Windows

Помимо двух настоящих извращений — хоккея на траве и балета на льду, существует и третье — диалап через IP-телефонию.

www.connect.ru

Когда Intel в 2004 году объявила о том, что всем нам в недалеком будущем предстоит стройными рядами переходить на последовательный интерфейс PCI Express, я, как инженер-электронщик, задумался — а чем, собственно, последовательные интерфейсы лучше параллельных? Ведь изученные в свое время до мельчайших деталей стандартные COM и LPT демонстрировали в сравнении совершенно обратную картину — при примерно равном быстродействии логики, COM совершенно законно имел скорость примерно в 10 раз более низкую, чем LPT. Если считать в одних и тех же единицах, то максимальная скорость COM достигает 115 Кбит/с, а LPT — до 1,3 Мбит/с, что логично: параллельный порт передает 8 бит за 1 такт, а последовательный — за 10 (считая стартовые/стоповые биты). Оказалось же вот что: даже учитывая этот коэффициент, более быструю последовательную логику делать дешевле. И еще намного дешевле становится раскладка печатных плат — не только из-за площади, а еще и из-за требования к длине проводников, разница в которой для параллельной передачи при современных скоростях оказывается узким местом. Но нам здесь не придется разбирать эти скоростные интерфейсы — существует очень много технических задач, которые не требуют обмена с подобными скоростями. Как пример "ширпотребовских" устройств такого типа можно привести широко распространившиеся в последние годы модули GPS, обычно имеющие COM-интерфейс и работающие на скоростях порядка 9600 бод. И большинство устройств научного, медицинского или инженерного назначения нормально обмениваются данными именно со скоростями того же порядка.

COM-порт отчасти ошибочно еще называют портом RS-232 (Recommended Standard, рекомендованный стандарт). Правильно сказать так: COM-порт передает данные, основываясь на стандарте последовательного интерфейса RS-232. Последний, кроме собственно протокола передачи, стандартизирует также и электрические параметры, и даже всем знакомые разъемы DB-9 и DB-25. Есть и еще один термин, который имеет непосредственное отношение к этой теме: UART (Universal Asynchronous Receiver-Transmitter, универсальный асинхронный приемопередатчик). UART есть составная часть COM-порта, его основа, то устройство, которое собственно передает и принимает данные, и к которому адресуются пользовательские программы. Кроме UART, в COM-порт входит преобразователь уровня UART/RS-232. Некоторые технические подробности, включая приемы программирования UART со стороны микроконтроллеров и схемы сопрягающих устройств, те, кто этим интересуется, могут узнать из *приложения 4*. Для того чтобы писать работающие программы, сведений, излагаемых в этой главе (и в указанном приложении), достаточно, а тех, кто желает изучить вопрос более подробно, я бы отослал к книге [33], которая, несмотря на некоторое количество досадных неточностей и излишних "наворотов", является отличным справочником по программированию последовательного порта (включая даже такую экзотику, как реализация режима Plag&Play для RS-232).

Что же касается USB, то, вопреки распространенному мнению (и многочисленным рекламным заявлениям), это не замена COM. USB — это не столько последовательный интерфейс для обмена данными, сколько шина для подключения устройств различного типа в режиме Plag&Play. Можно сказать так: для системы USB — это скорее PCI-устройство, но ни в коем случае не UART. С помощью USB можно эмулировать передачу данных в режиме UART, но для этого требуются специальные драйверы и переходные устройства. Далее мы рассмотрим некоторые возможности по организации такого режима для обмена данными. В целом же рассмотрение USB выходит за рамки книги — это, во-первых, слишком обширная, а во-вторых, специфическая тема, которую стоит специально изучать, если вы собираетесь заняться проектированием Plag&Play-устройств со скоростным обменом данными. Для введения в тему USB я рекомендую книгу [29], которая на момент написания этих строк являлась единственным достаточно полным и относительно последовательным изложением построения интерфейса USB на русском языке. А теперь займемся собственно программированием COM-порта.

Передача данных через COM-порт

Мы себе задачу ограничим — не будем рассматривать синхронные протоколы (организация которых возможна при использовании дополнительных выводов COM-портового разъема), т. к. большинство устройств используют

трехпроводной асинхронный протокол передачи в "чистом" виде. Задача установки в определенное состояние некоторых из этих дополнительных выводов — она требуется, например, если внешнее устройство питается от одной из таких линий, подобно COM-портовой мыши — разбирается в *приложении 4*.

Начнем с постановки задачи — как вообще организовать обмен данными через COM? Отметим, что таким образом поставить эту задачу так же невозможно, как в общем виде ответить на вопрос "как послать письмо по электронной почте?" Если вы хотите иметь какой-то определенный способ обмена, то придется проектировать и устройство, и программу во взаимосвязи. Но, как правило, мы уже имеем некое устройство. Зададим вопрос поконкретнее — как оно может работать?

Самым надежным способом является обмен данными по типу запрос/ответ[1]. Внешнее устройство, как правило, является источником данных, поэтому вы посылаете соответствующие команды-запросы, а устройство в ответ вам посылает данные. Объем посылки бывает строго оговорен (самый простой вариант), но иногда бывает, что информация о количестве данных содержится в заголовке ответного сообщения (заголовок должен при этом иметь строго оговоренный формат), или кадр с данными заканчивается чем-то конкретным — нулевым байтом, переводом строки/возвратом каретки ($13+$10) или еще как-то. Типичным представителем протоколов обмена по типу запрос/ответ является протокол Garmin для GPS-приемников одноименной фирмы, там кадр с данными начинается всегда с байта $10, оканчивается всегда сочетанием байтов $10+$03, но, кроме того, имеется и информация о длине посылки (см. [27]). Если вам кажется, что односторонний протокол NMEA (когда GPS-приемник автоматически выдает сообщения в компьютер без всяких запросов), который мы далее будем использовать на практике, проще в программной реализации, то вы и правы и не правы одновременно. Подробнее об обмене с собственно GPS-модулями мы поговорим позже, а пока попытаемся понять, почему односторонний протокол передачи теоретически сложнее в реализации.

О программах реального времени

Любой асинхронный обмен в одну сторону — когда с устройства поступает некий непрерывный поток данных — требует от принимающей стороны реа-

[1] В терминологии Windows API "синхронным" называют именно этот способ. Это не совсем так — синхронными называют протоколы, которые на физическом уровне (через сигналы, например, по линиям RTS/CTS или другим) автоматически синхронизируют свою работу. Если мы не используем подобные "рукопожатия" (handshakes, см. *приложение 4*), то любой обмен данными по RS-232 с помощью только двух проводов будет асинхронным.

лизации режима реального времени. В этом режиме основная задача — "не потерять ни капли жира с этого драгоценного гуся" ("Три мушкетера"), т. е. ни одного байта данных. Данные передаются обычно кадрами, отдельные байты в которых часто связаны между собой (например, они образуют многобайтовые числа), и ошибка в приеме такого кадра может дорого обойтись. Windows принципиально не может обеспечить режим реального времени, если это касается длительностей порядка миллисекунд и даже десятков миллисекунд — в этом случае, например, если вы параллельно со своей программой запустили что-то ресурсоемкое (хотя бы просто инициализировали запуск другой достаточно громоздкой программы), то ваша программа будет заторможена, и вы можете что-то потерять. Конечно, на практике все эти соображения могут относиться в основном к случаю сравнительно медленных компьютеров, но тут дело не в самой по себе скорости работы. Максимальное время переключения между потоками в семействе NT составляет порядка миллисекунд даже на быстрых компьютерах и линейно растет с увеличением числа потоков (сравните — за 1 мс можно передать/принять целый байт со скоростью всего 9600 бод). Среднее время может быть намного меньше, но в общем случае гарантий не может дать никто. Разумеется, мы можем из-за этого потерять байт-другой и при передаче запрос/ответ, но вероятность этого существенно меньше, т. к. нам не надо следить за системой непрерывно. Обычное значение кванта времени, выделяемое конкретному потоку, составляет не менее нескольких десятков миллисекунд, и за это время буфер обычного размера (см. далее) мы гарантированно успеваем очистить, а вот в случае непрерывного поступления данных это может оказаться не так.

Способ обеспечения режима для конкретной программы, более-менее напоминающего режим реального времени, можно в принципе обеспечить с помощью регулирования приоритета конкретного потока (thread). У потоков может быть семь уровней относительных приоритетов (от Idle до Time critical), а у процесса — шесть классов (от Idle class до Real time class). При назначении общего приоритета они комбинируются. Вот так можно обеспечить процессу и его потоку максимальный приоритет [40]:

```
var
PriorityClass, Priority: Integer;
. . . . .
{запоминаем для восстановления:}
PriorityClass := GetPriorityClass(GetCurrentProcess);
Priority := GetThreadPriority(GetCurrentThread);
{устанавливаем наивысший:}
SetPriorityClass(GetCurrentProcess, REALTIME_PRIORITY_CLASS);
SetThreadPriority(GetCurrentThread, THREAD_PRIORITY_TIME_CRITICAL);
. . . . .
```

```
{восстанавливаем обратно:}
SetThreadPriority(GetCurrentThread, Priority);
SetPriorityClass(GetCurrentProcess, PriorityClass);
```

В реальности это следует делать только на достаточно короткое время, потому что иначе, если даже Windows рано или поздно и не рухнет, то работать в такой системе с чем-то еще будет очень затруднительно. Лучше все это делать в консольном приложении, запускаемом в режиме командной строки, когда графическая оболочка с ее чудовищным количеством сообщений не загружена. Вопрос, однако: разве это жизнь? Лично я бы в таком случае лучше сделал однозадачную DOS-программу, которая гарантированно владеет всеми ресурсами, а самое правильное решение — использовать ОС QNX, которая специально для таких целей имеет гарантированное максимальное время переключения между задачами.

Но вот, например, для периодического точного измерения времени коротких процессов этот механизм в совокупности с введенной, начиная с процессоров Pentium MMX (а также и с AMD K5), командой `rdtsc` (Read Time Stamp Counter), вполне можно использовать. Команда `rdtsc` возвращает в регистрах `edx:eax` восьмибайтное слово, равное количеству "тиков" CPU, прошедшего с момента его последнего сброса. Так как когда-то никто не был уверен, что команда поддерживается данным конкретным ассемблером (например, включенным в состав Object Pascal), ее традиционно вызывают просто в виде опкода 0F 31:

```
var TickLo, TickHi:dword;
. . . . .
asm
    dw 310Fh; {rdtsc}
    mov TickLo, eax
    mov TickHi, edx
end;
```

Эту процедуру вызывают в начале и по окончании события, время которого измеряется, и вычитают из второго значения первое. Причем вычитание можно произвести прямо в ассемблерной процедуре:

```
asm
    dw 310Fh
    sub eax, TickLo
    sbb edx, TickHi
    mov TickLo, eax
    mov TickHi, edx
end;
```

На время этого измерения потоку назначают максимальный приоритет по методике, указанной ранее. В результате вы получаете в TickHi:TickLo точное значение "тиков" процессора за интересующий вас период, что потенциально может дать чудовищную точность. Подводный камень тут один: превратить это в единицы времени можно только, зная достаточно точно частоту процессора, а это обычно проблема, т. к. измерить ее можно либо через системный таймер, либо через RTC (Real Time Clock, часы реального времени), которые находятся в CMOS-чипе с резервной батарейкой. Чтобы убедиться, насколько может врать системный таймер, установите на форму компонент Memo и запустите по какому-нибудь событию простую программку, которая выполняет четырьмя разными способами секундную задержку, и измеряет ее длительность:

```
uses DateUtils;
. . . . .
var
tt,told:TDateTime;
i:integer;
. . . . .
  Memo1.Lines.Clear;
  told:=Time;
  for i:=1 to 1000 do
  sleep(1);
  tt:=Time;  {измеряем время}
  st:=IntToStr(MilliSecondsBetween(tt,told));
  Memo1.Lines.Add(st);
  told:=Time;
  for i:=1 to 100 do
  sleep(10);
  tt:=Time;  {измеряем время}
  st:=IntToStr(MilliSecondsBetween(tt,told));
  Memo1.Lines.Add(st);
  told:=Time;
  for i:=1 to 10 do
  sleep(100);
  tt:=Time;  {измеряем время}
  st:=IntToStr(MilliSecondsBetween(tt,told));
  Memo1.Lines.Add(st);
  told:=Time;
  sleep(1000);
  tt:=Time;  {измеряем время}
  st:=IntToStr(MilliSecondsBetween(tt,told));
  Memo1.Lines.Add(st);
. . . . .
```

Ручаюсь, что по выполнении этой программы у вас глаза на лоб полезут, особенно если вы ее повторите несколько раз. Вот что выводится в Memo в одном из моих экспериментов (в среде Windows 98):

```
5770
1480
1049
979
```

Первое значение может "гулять" в пределах нескольких секунд, но даже относительно длинные промежутки отмеряются с совершенно неприемлемой для измерения времени погрешностью порядка 2—5%. Отметим, что в Windows XP, согласно моим наблюдениям, ситуация одновременно и хуже и лучше: малые промежутки времени вообще не отмеряются, можно сказать, никак (первое число составило около 15 секунд), а большие (секунда и выше) измеряются точнее. Так что обычно частоту процессора измеряют несколько раз, применяя ту же rdtsc, и затем усредняют полученные значения. Заметим также, что в ноутбуках с механизмом подстройки тактовой частоты все это просто может не заработать.

Углубляться в эту тему далее мы не будем, заметим еще только, что на практике Windows-программы общего назначения, которые требуют соблюдения интервалов времени с достаточно высокими точностями, никто и не делает — возросшая мощность микроконтроллеров позволяет подобные критичные функции передать самому устройству, а в компьютер передавать уже результаты расчетов, как, к примеру, и поступают приемники GPS (что при взгляде со стороны, согласитесь, выглядит несколько странновато — процессор в PC заведомо мощнее любого микроконтроллера, а занимается всякой ерундой вроде обработки движений мыши — но тут уж ничего не поделаешь). "Прилично" себя ведет Windows только на временных отрезках порядка 1 сек или несколько менее.

Для того чтобы гарантированно избежать ошибок при обмене, организуют различные проверки и синхронизации. Самой распространенной из проверок на то, что байты не были искажены при передаче по линии, является сравнение контрольных сумм. Контрольная сумма оговоренного количества байтов рассчитывается на отправляющей стороне и передается в конце основной посылки. Чтобы не посылать большое число, в которое может вылиться обычная сумма, ее упаковывают. Самый простой метод формирования более-менее уникальной для данной последовательности контрольной суммы — нахождение для нее так называемого "дополнения до 2" в виде контрольного байта.

Заметки на полях

Контрольный байт рассчитывается следующим образом: обычную сумму всех байтов строки вычитают из любой степени числа 2, большей, чем полученная сумма (но не меньшей 256). Младший байт полученной разности и будет искомым значением контрольного байта. На практике этот алгоритм идентичен следующему: мы изначально приравниваем значение контрольной суммы 0, а потом последовательно вычитаем из нее все входящие в сумму значения байтов по очереди. На самом деле такой метод проверки целостности сообщения не идеален — эта сумма, например, не изменится, если байты просто переставить местами. Однако, когда возможные ошибки имеют вероятностный характер, этого вполне достаточно — и сама вероятность сбоя мала, а вероятность возникновения двойной ошибки в битах именно так, чтобы два байта поменяли свое значение друг на друга, можно не принимать во внимание.

Но искажение при передаче — не столь уж часто встречающаяся вещь. Другой момент при приеме в реальном времени гораздо важнее — программа может быть запущена тогда, когда передающий прибор уже включен и поток данных вовсю поступает в компьютер. Вероятность попасть при этом в середину кадра ни в коем случае нельзя сбрасывать со счетов. (Заставлять при этом пользователя соблюдать строго определенный порядок включения — самое крайнее любительство, какое только можно придумать.) Конечно, можно заставить приемную программу проверять паузу между посылками, оговорив, что она не меньше, например, чем 4 мс, и не больше 8 мс — мне приходилось выполнять подобную задачу, и более неудобного способа синхронизации потока придумать трудно. Поэтому всегда, когда посылка состоит из двух и более связанных между собой байтов, их следует сопровождать так называемой "сигнатурой" — неким сочетанием байтов с определенным значением. Обычно достаточно, чтобы первые два байта имели всегда определенную величину — даже если такое же сочетание может встретиться внутри посылки, вероятность попасть именно на него в момент включения тем меньше, чем больше временной промежуток между кадрами. Но для большей надежности можно, например, дублировать принятые в виде сигнатуры байты внутри потока дважды — такое сочетание в начале посылки исключено. А как узнать конец посылки? Самое удобное — также заканчивать ее определенной сигнатурой (если, разумеется, длина посылки не строго оговорена). Все перечисленные способы вместе (включая подсчет контрольной суммы) и использует, например, упомянутый протокол Garmin.

Прием и передача одного или нескольких байтов

В самом идеальном случае организации обмена, как мы уже говорили, программа посылает запрос, в ответ на который устройство отвечает фиксированным и заранее известным количеством байтов. Рассмотрим простейший

вариант такой процедуры. Как вы увидите, если делать все, как надо, то он окажется не таким уж и "простейшим" (а кто сказал, что в DOS было все так уж и просто? сами убедитесь, что в некоторых отношениях здесь даже удобнее), но зато в дальнейшем он послужит основой для более сложных протоколов. К большому сожалению, вы, скорее всего, не сможете сразу проверить работу этой программы у себя. Хотя я и размещаю на диске, как обычно, полный проект, но изучать вам его придется теоретически — "часы", которые служат в качестве подключенного устройства для этого примера, я приложить не могу, и описание устройства, которое я использовал здесь в качестве "часов", выходит далеко за рамки этой книги. Впрочем, один выход у вас имеется (кроме, конечно, как соорудить подобное устройство самому) — "часы" можно эмулировать вторым компьютером, соединив у них COM-порты через нуль-модемный кабель *(см. приложение 4)* и установив на него программу, аналогичную описанному далее "эмулятору терминала", измененную так, чтобы она воспринимала команды и посылала нужный набор байтов. Еще показательней было бы использовать один и тот же компьютер с двумя портами, которые и соединить между собой. Если вы внимательно изучите эту главу, то сделаете все это без труда и заодно приобретете бесценный опыт.

Итак, предположим, имеется некое устройство (часы), которое может передать нам через COM-порт текущее время. Для его получения мы обязаны отправить запрос — байт со значением $A2. После получения такого запроса часы немедленно выдадут нам время в виде последовательности 6 байтов (Sec:Min:Hour:Date:Month:Year). Все эти значения выдаются в упакованном BCD-формате (обычный формат для электронных часов, см. *приложение I*). Обмен идет со скоростью 9600 бод, в соответствии с формулой "8n1" *(см. приложение 4)*.

Создадим пробный проект под названием COMproba (расположен в папке Glava21\1). На форме расставим компоненты `Label` и `StaticText` в количестве 6 штук каждый. Чтобы форма выглядела более эстетично, для всех `StaticText` установим свойство `Color` в `clNavy` (темно-синий), а свойство `Font` в полужирный 10 кегля с цветом `clAqua` (голубой). Заголовки `StaticText` очистим, а в `Label` напишем `Число`, `Месяц`, `Год`, `Час`, `Мин` и `Сек`. Всем `StaticText` для удобства придадим соответствующие названия `StaticTextMin`, `StaticTextSek` и т. п. (причем в случае "sek" не сочтите меня неграмотным — для значения секунд в идентификаторах я часто использую обозначение Sek, а не Sec, т. к. последнее, например, есть идентификатор функции секанса). Ниже поставим еще один `Label` для отображения реально установленного порта и компонент `ComboBox` для выбора порта. В `ComboBox` в свойство `Items` запишем построчно наименования портов (без пробелов!): `COM1`, `COM2`, `COM3` и `COM4`. Еще ниже поставим кнопку `Button1` **Запрос** (см. рис. 20.1). Разместим также в папке с проектом модуль Ariphm.

Собственно передача и прием данных через COM-порт неоднократно описаны во множестве публикаций и теоретически в них ничего сложного нет. Однако не все так просто. А что вы будете делать, если:

- устройство отключено;
- это не то устройство;
- выбран несуществующий COM;
- на выбранном порту находится модем.

В общем, в этой процедуре главное — обработка исключительных ситуаций, причем тут это жизненно необходимая вещь, если вы сделаете что-то не так, то Delphi за вас ничего не поправит. И, как вы увидите, подводных камней тут достаточно много.

Один из них — определение имеющихся в системе последовательных портов и их характера. Сама Windows, конечно, точно знает, сколько у нее портов и каких — это отображается в Диспетчере устройств. Однако, как ни странно, адекватной процедуры для получения их перечня не имеется. Последовательные порты должны быть перечислены в ключах реестра Windows 98:

```
HKEY_LOCAL_MACHINE\Hardware\DeviceMap\SerialComm
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Class\Ports
HKEY_LOCAL_MACHINE\Enum
```

Для XP аналогичные ключи выглядят так:

```
HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\SERIALCOMM
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\serenum\Enum
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\serial\Enum
```

Соединив сведения из [33, 36] и MSDN, я попробовал поковыряться в реестре своего компьютера (два обычных COM1 и COM2, а также модем на COM3) для обеих установленных у меня систем (Windows 98 и XP), но понял, что разобраться в логике устройства этих веток с налета не удастся. Например, в первой из перечисленных веток для Windows 98 перечислено почему-то целых пять идентичных портов от COM1 до COM5, в то время как для Windows XP в том же ключе записано три порта, правда, без намеков на то, какой из них модем. Сопоставляя сведения из всех этих веток можно, наверно, разобраться, но ведь есть еще функция `EnumPorts`, которая, согласно описанию, должна возвращать описания всех портов в системе? Процедура ее использования с легкой руки автора [33] растиражирована по Интернету. Однако, как показывает простой эксперимент, корректно она работает только в Windows 98. А вот в Windows XP она возвращает полную чушь:

```
COM1: Локальный порт
COM2: Локальный порт
COM3: Локальный порт
COM4: Локальный порт
```

Откуда она берет все эти данные, ума не приложу, но факт, что использовать эту функцию в Windows XP нельзя даже для того, чтобы определить количество реально действующих LPT-портов (как прямо рекомендует MSDN), ибо она мне вернула их аж целых три штуки, что в природе до сих пор не случалось (я только один раз в жизни встретил компьютер, в котором было больше одного LPT-порта). У меня есть один LPT, никакого другого, даже виртуального типа PDF или PostScript, в системе совершенно точно нет.

Мне кажется, что наиболее надежным способом определять количество последовательных портов в системе будет все же простой перебор их с попыткой инициализации каждого. Как при этом отличить модем от простого последовательного порта, я покажу далее.

Итак, приступим. Для начала установим следующие переменные:

```
var
Form1: TForm1;
hCOM:hFile=0;
pDCB:TDCB;
comtime:TCOMMTIMEOUTS;
xb:byte;
xn:dword;
ab: array[1..6] of byte;
st,stcom:string;
ttime,told:TDateTime;
. . . . .
```

Начнем мы с самого главного — с отдельной процедуры инициализации порта, которую назовем IniCOM. Вот ее текст:

```
procedure IniCOM;
var i:integer;
begin
{инициализация COM - номер в строке stcom}
hCOM:=CreateFile(Pchar(stcom),
 GENERIC_READ+GENERIC_WRITE,0,nil,OPEN_EXISTING,0,0);
if (hCom = INVALID_HANDLE_VALUE) then
begin
 st:=stcom+' не найден';
```

```
 Application.MessageBox(Pchar(st),'Error',MB_OK);
 exit;
end;
if GetCommState(hCOM,pDCB)
then st:=stcom+': baud=9600 parity=N data=8 stop=1';
if BuildCommDCB(Pchar(st),pDCB) then SetCommState(hCOM,pDCB)
else
begin
 st:=stcom+' занят или заданы неверные параметры';
 Application.MessageBox(Pchar(st),'Error',MB_OK);
 exit;
end;
GetCommTimeouts(hCom,comtime); {устанавливаем задержки:}
comtime.WriteTotalTimeoutMultiplier:=1;
comtime.WriteTotalTimeoutConstant:=10;
comtime.ReadIntervalTimeout:=10;
comtime.ReadTotalTimeoutMultiplier:=1;
comtime.ReadTotalTimeoutConstant:=2000; {ждем чтения 2 сек}
SetCommTimeouts(hCom,comtime);
ab[1]:=ord('A'); {будем посылать инициализацию модема}
ab[2]:=ord('T');
ab[3]:=13;{CR}
ab[4]:=10;{LF}
WriteFile(hCOM,ab,4,xn,nil);
if ReadFile(hCOM,ab,10,xn,nil) then {ответ модема 10 знаков}
begin
  st:='';
  for i:=1 to 10 do st:=st+chr(ab[i]);
  if pos('OK',st)<>0 then
  begin
   st:=stcom+' занят модемом';
   Application.MessageBox(Pchar(st),'Error',MB_OK);
   CloseHandle(hCOM);
   hCOM:=0;
   Form1.Label7.Caption:='COM?';
   exit;
  end;
end;
Form1.Label7.Caption:=stcom+' 9600';
end;
```

К началу выполнения этой процедуры у нас в строке stcom должно содержаться название порта, например, COM1. Сначала текст тут мало отличается от того, что описано во всех стандартных рекомендациях по программированию

порта. Единственный момент, который несколько выходит за рамки стандарта — мы не устанавливаем поля структуры `DCB` напрямую, а используем функцию `BuildCommDCB`. Я это делаю отчасти потому, что структура `DCB` в Delphi транслируется из API не полностью (сравните ее описание в Windows.pas и в Win32.hlp), и хотя для данного случая, разумеется, все нужные поля имеются, но использование `BuildCommDCB` все равно удобнее.

Заметки на полях

Любой преподаватель программирования, глядя на этот вопиющий случай использования глобальных переменных для передачи параметров (имеется в виду строка `stcom`), немедленно сделал бы мне замечание. И по правилам структурного программирования, как говорилось в *главе 1*, в данном случае, конечно, следовало бы завести локальную переменную — неизвестно, где и когда мы еще будем использовать описанную ранее процедуру. Но я уже предупреждал в *главе 1*, и повторю — традиционно я стараюсь локальных переменных избегать. Это, конечно, развращающее влияние ассемблера, где для организации локальных переменных требуется много команд `push...pop`, так загромождающих программу, что проще использовать ячейку памяти в качестве еще одной глобальной переменной. Все мое существо восстает против, когда я начинаю представлять себе, сколько *лишних* команд придется выполнить компьютеру только потому, что кому-то боязно забыть о своевременной установке некой переменной. Что, разумеется, есть рассуждение совершенно "ламерское": современные программы делают так много лишнего, что еще один уровень стека никакой роли сыграть не может, зато гарантий от ошибок больше.

После стандартных установок мы сразу делаем две вещи, о которых упоминают далеко не всегда. Во-первых, мы устанавливаем все возможные Timeout для разных вариантов приема и передачи. В параметрах, которые заканчиваются на `Multiplier`, можно для простоты всегда ставить 1 (если больше, то процедуры чтения/записи будут отслеживать еще и скорость поступления байтов, что нам не надо). А остальные делают следующее: если задержка посылки через порт больше, чем `WriteTotalTimeoutConstant` (в миллисекундах), то будет прервана передача, а при задержке между поступающими байтами больше, чем `ReadIntervalTimeout`, и при задержке всей процедуры чтения (в данном случае — самый главный параметр) больше, чем `ReadTotalTimeoutConstant`, будет прерван прием. Последний параметр мы установили равным 2 с. При выборе этих параметров следует иметь в виду, что один байт при скорости 9600 передается/принимается примерно за 1 мс. Если эти параметры вообще не устанавливать (оставить их в значении 0, как по умолчанию), то при отсутствии принимаемых байтов процедура чтения через `ReadFile` просто зациклится и "повесит" всю программу.

Заметки на полях

Среди коммуникационных функций [31] есть некоторые, которые работают с состоянием "обрыва" линии — могут как устанавливать линию TxD в это состояние *(см. приложение 4)*, так и определять "обрыв" на линии RxD. Если честно,

то я не вижу никакой необходимости в манипуляции этим состоянием. Оно могло бы пригодиться для определения того, что устройство не подключено к компьютеру (или внезапно было отключено), но какая нам разница — подключено оно или нет, если мы так или иначе не получаем с него ни одного байта? Намного надежнее устанавливать факт подключения по получению от устройства посылки либо автоматически, либо в ответ на посланную (возможно даже — специальную для этой цели) команду.

Наконец, в самую последнюю очередь мы определяем, не является ли установленный нами порт модемом. Так как мы рассматриваем "чистый" RS-232, то для нас модемный порт все равно как бы занят. Если же вы соберетесь программировать сам модем, то, во-первых, в Windows это лучше делать не напрямую через UART, а через TAPI ("телефонные" API), а во-вторых, и в этом случае процедура его определения вам пригодится. Определяем модем мы очень просто: посылаем в выбранный порт символьный код инициализации, который одинаков для всех модемов: AT<CR><LF> (65 84 13 10). В ответ мы от модема должны получить строку AT<CR><LF><CR><LF>OK<CR><LF> (65 84 13 10 13 10 79 75 13 10), но все такие подробности нам не требуются, подозреваю, что строка для разных модемов может немного отличаться, но в любом случае в ней должны содержаться символы OK (если модем, конечно, свободен — с занятым модемом я предоставляю читателю разобраться самостоятельно). В последнем операторе, если порт инициализировался нормально, выводим в Label номер порта и скорость.

Эту процедуру мы выполним сразу при запуске (для COM1). Заодно напишем процедуру закрытия порта (ведь порт все время занят, пока программа запущена):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
{инициализация COM1 при запуске}
stcom:='COM1';
IniCOM;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin {уничтожаем COM}
 CloseHandle(hCOM);
end;
```

Сразу же напишем и процедуру установки нового порта при обращении к ComboBox:

```
procedure TForm1.ComboBox1Select(Sender: TObject);
begin
CloseHandle(hCOM); {закрываем старый COM}
```

```
stcom:=ComboBox1.Text; {устанавливаем порт COM1,2,3,4}
IniCOM;
end;
```

Заметки на полях

Если у читателя возникнет такая необходимость, то он может при запуске программы заполнить список в ComboBox именами реально действующих COM-портов, выполнив примерно вот такой цикл их опроса (здесь мы перебираем порты от COM1 до COM8):

```
. . . . .
for i:=1 to 8 do
begin
   stcom:='COM'+IntToStr(i);
   hCOM:=CreateFile(Pchar(stcom),
   GENERIC_READ+GENERIC_WRITE,0,nil,OPEN_EXISTING,0,0);
   if (hCom=INVALID_HANDLE_VALUE) then continue;
   ComboBox.Items.Add(stcom);
   CloseHandle(hCOM);
end;
. . . . .
```

Ну и, наконец, главная процедура запрос/ответ по нажатию кнопки Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin  {запрос}
if (hCOM=0) or (hCOM=INVALID_HANDLE_VALUE) then exit;
          {если порт еще не инициализирован - выход}
PurgeComm(hCOM,PURGE_RXCLEAR); {очищаем буфер}
xb:=$A2;
WriteFile(hCOM,xb,1,xn,nil);
told:=Time;
if ReadFile(hCOM,ab,6,xn,nil) then {читаем 6 байтов в массив ab}
begin
 ttime:=Time;
 if SecondsBetween(told,ttime)>0 then
 begin
  Application.MessageBox('Устройство не
              обнаружено','Error',MB_OK);
  exit;
 end;
 if xn<>6 then
 begin
  Application.MessageBox('Неправильный формат
              данных','Error',MB_OK);
```

```
  exit;
 end;
 StaticTextYear.Caption:=hexb(ab[6]);
 StaticTextMonth.Caption:=hexb(ab[5]);
 StaticTextDate.Caption:=hexb(ab[4]);
 StaticTextHour.Caption:=hexb(ab[3]);
 StaticTextMin.Caption:=hexb(ab[2]);
 StaticTextSek.Caption:=hexb(ab[1]);
end else {не сработало}
begin
 Application.MessageBox('COM сломался','Error',MB_OK);
 exit;
end;
end;
```

Несколько моментов в этой процедуре требуют пояснений. Во-первых, нужна ли процедура очистки приемного буфера PurgeComm и вообще что это за буфер? Напомню, что в DOS никакого приемного буфера у UART не было (по крайней мере, если мы не открывали COM-порт, как файл), его приходилось организовывать самостоятельно. Но Windows (семейств и 9*x* и NT) создают по умолчанию системный буфер, равный 128 байтам. Если мы его не очистим перед чтением, то можем попасть в дурацкую ситуацию, если, например, устройство не только выполняет посылку данных по команде, но и все время выдает что-то самостоятельно. То же самое будет и в случае, если посылка длиннее 6 байт — лишние байты будут считаны в следующем сеансе, а "хвост" в буфере еще больше увеличится. Поэтому команду очистки лучше на всякий случай добавлять пред запросом, тогда мы уверены, что получим именно то, что запрашивали.

И еще такой вопрос возникает — а много это или мало, 128 байт приемного буфера? Скажем так — если мы не ведем прием сплошного потока данных (как, например, голосового потока через voice-модем) или просто не собираемся принимать больших массивов, то этого достаточно, главное вовремя данный буфер очищать, а последнее требуется при любом объеме буфера. Если мы при чтении произвольного потока очищаем буфер прямо сразу при поступлении очередного байта или их цепочки (как мы это будем делать позже), то величина его в принципе вообще значения не имеет — но на практике он все же требуется для страховки на случай "торможения" от самой Windows. Заведомо достаточной будет величина буфера примерно такая: сколько байт в секунду поступает при заданной скорости, т. е. для скорости 9600 это 1024 байта. Установить величину буфера для данного порта по умолчанию можно ручным редактированием секции [386Enh] файла system.ini, в которой нужно добавить строку (на примере COM1):

```
Com1: Com1Buffer=8192
```

Динамически буфер устанавливать также очень просто: через функцию `SetupComm`, параметрами которой и являются величины выходного и входного буфера. Например, вот так можно установить указанное ранее значение 8192 (сохранив при этом для выходного буфера значение 128):

```
SetupComm(hCOM,8192,128);
```

Последний вопрос о буфере — а какое значение имеет выходной буфер? Отвечаем: если мы не посылаем в устройство сразу большой массив данных, то никакого, пусть так и равным 128 байтам и остается.

Второй момент, который нужно прокомментировать: обнаружение устройства. Мы здесь, как видите, все делаем очень просто: читаем системное время до и после вызова функции `ReadFile` и выясняем — если прошло более 1 секунды (а Timeout у нас задан в 2 секунды), то "устройство не обнаружено". Опыт показывает, что это самый надежный метод. Если же связь каким-то образом прервется посередине посылки, то мы получим меньше байтов, чем заказывали, и программа выдаст сообщение "`Неправильный формат данных`". Не забудьте, что надо добавить ссылку на модуль DateUtils.

Третий момент — предвкушаю недоумение внимательного читателя, который, без сомнения, обратил внимание на утверждение о том, что "значения выдаются в упакованном BCD-формате". И что, их не надо распаковывать? Нет, не надо — ведь мы с ними не производим никаких операций, а легко сообразить, что, будучи представлен в HEX-форме, упакованный BCD-формат даст нам отображение времени в привычном виде. Легко, кстати, на основе строковых преобразований перевести BCD-данные в обычные числа без всякой распаковки, достаточно применить к нему обратное преобразование в число:

```
var st:string;
xb:byte;
. . . . .
{xb=59h, в BCD-формате это будет просто 59, в десятичном виде 89}
st:=hexb(xb); {st ='59'}
xb:=StrToInt(st); {xb=59 в десятичной форме}
. . . . .
```

Теперь уже можно производить с данными и всякие операции. Напоминаю, что вместо моих процедур из модуля Ariphm *(см. приложение 1)* в последних версиях Delphi можно использовать функцию `IntToHex`.

И, наконец, четвертый момент: что это за бредовое сообщение "`COM сломался`"? Все дело в том, что вернуть значение `False` процедура `ReadFile` при установленных нами в данном случае параметрах в функции `CreateFile` просто не может. Ошибка могла бы возникнуть, например, если бы мы осуществляли

проверку на четность. Еще более важный момент связан с возможным возвратом значения False, если бы CreateFile вызывалась с флагом **FILE**_FLAG_OVERLAPPED и соответствующими дополнительными установками (определение события и т. д.). Механизм, связанный со структурой OVERLAPPED, направлен на то, чтобы освободить вызывающий поток и сам "файл" (т. е. порт в данном случае) на время ожидания операций чтения/записи, которые в принципе могут быть весьма длительны. При обычных операциях с COM-портом, когда данные передаются туда-сюда в час по чайной ложке, возиться с механизмом "оверлапинга" (overlapping) абсолютно ни к чему, т. к. большинство коммуникационных приложений основное время проводят впустую в ожидании поступления очередного байта и синхронное общение по принципу запрос/ответ тут ничуть не замедляет процесс. А вот когда мы точно не знаем, когда именно должен поступить очередной байт данных — другое дело, и позже мы как раз об этом и будем разговаривать. Здесь же именно поэтому я и охарактеризовал единственный случай, когда теоретически может быть возвращено значение False, как неисправность COM. А точнее не COM, а UART — если в процессе чтения порт просто "сгорит", то это обычно происходит с внешним преобразователем уровней, который UART не затрагивает, и мы получим в этом случае, скорее всего, сообщение "Устройство не обнаружено".

Результат работы программы COMproba при взаимодействии с моими "часами" см. на рис. 20.1.

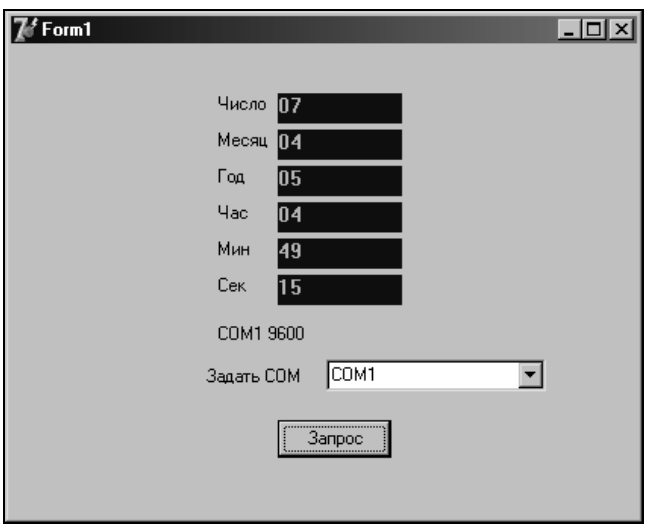


Рис. 20.1. Результат работы программы COMproba

Прием и передача в реальном времени

В DOS для организации режима реального времени было достаточно просто зациклить процедуру, которая все время определяет, поступил ли байт в приемный регистр UART, и при поступлении немедленно его обрабатывает. Второй вариант — организовать работу по аппаратным прерываниям от COM-порта. По прерыванию принятый байт складывается в некий буфер в памяти, сама же программа при этом мало чем отличается от первого варианта — она так или иначе в цикле должна проверять наличие байтов в буфере.

В Windows же в некотором отношении все даже проще — любая программа и так представляет собой цикл. Причем к прерываниям COM-порта вас непосредственно там не пустят[2], и остается только работать через указанные ранее функции Windows. На практике для организации подобной процедуры лучше всего организовать в программе параллельный процесс, в котором только опрашивать порт и складывать в буфер полученные байты, а в основной программе обрабатывать их, например, по таймеру. Но крупнейший недостаток этого напрашивающегося способа, который, к сожалению, иногда рекомендуют — нам надо все устроить так, чтобы обработка происходила своевременно. Подогнать событие таймера точно под период следования данных нам не удастся даже теоретически, единственный вариант — чтобы оно происходило заведомо чаще, иначе данные попросту будут пропадать. Поэтому такая процедура не является универсальной — она годится, если данные поступают максимум раз в секунду (ну хорошо, несколько раз в секунду). А в общем случае ее удовлетворительной признать нельзя. Теоретически можно обрабатывать данные в самом потоке при возникновении события, но это еще более неправильно: мы для того и организовывали параллельный поток, чтобы максимально разгрузить систему на момент приема данных. И много ли смысла в том, что вся наша программа постепенно "переедет" в этот поток?

Самым правильным и наиболее универсальным решением будет организовать в приемном потоке некое событие, которое посылается главному окну, как только из буфера прочитан очередной символ, и в его обработчике уже производить все нужные манипуляции с данными. Тогда мы можем быть уверены, что данные не потеряются, и что мы неоправданно не перегружаем систему. Хочу заметить, что в зависимости от характера входной последовательности событие это можно средствами Windows организовать самым различным способом — например, если мы знаем, что некую последователь-

[2] Точнее, в Windows 9*x* к ним еще как-то можно подступиться, но все равно это крайне неграмотный подход — если, конечно, не делать свой собственный драйвер.

ность замыкает символ <CR>, то можно посылать событие при приеме этого символа, остальные байты при этом будут к тому времени уже накоплены в буфере. Но в целях универсальности процедуры мы сделаем все же асинхронный прием каждого байта по мере поступления. В сущности, мы тогда получим прототип "эмулятора терминала" — программы, которая принимает произвольный поток входных данных, отображая их в том или ином виде на экране и/или складывая их в дисковый файл, и одновременно может по командам посылать данные. Модернизация, которой мы должны подвергнуть нашу программу, не так уж сложна: авторы учебников по Delphi сделали, кажется все, чтобы организация параллельных потоков приложения выглядела в глазах пользователей мероприятием, по сложности сравнимым с постройкой Эйфелевой башни. На самом деле никаких специальных модулей типа Thread Object не требуется, надо только помнить, что в организованном потоке нельзя обращаться к свойствам визуальных компонентов формы. Куда сложнее с самими коммуникационными функциями. Но обо всем по порядку.

Прием и передача данных с помощью параллельного потока

Мы рассмотрим прототип "эмулятора терминала" на том же примере с "часами". Я усложнил задачу, выставив свои "часы" в такой режим: они самостоятельно каждую секунду посылают значение секунд, а в начале каждой минуты выдают полную строку времени и даты (формат строки см. ранее). Старая возможность — получить в любой момент полное время по запросу посылкой команды $A2 — также имеется. Для того чтобы сформировать такой асинхронный протокол, мы и организуем параллельный поток, который будет посылать нам сообщения, если в буфере порта что-то есть. Для этого надо сначала установить событие, которое будет отслеживаться в потоке (не пользовательское сообщение, которое мы будем посылать форме — это само собой, — а именно внутреннее событие порта). Это делается с помощью функции `SetCommMask`, в которой устанавливается некий флаг. Из всего разнообразия вариантов этого флага мы выберем значение `EV_RXCHAR`, тогда порт нам будет сигнализировать о том, что в буфере что-то есть (что именно — рассмотрим далее). О том, что установленное событие произошло, можно определить через функцию `WaitCommEvent`.

Последняя представляет собой, как надо понимать, замкнутый цикл ожидания установки бита 0 регистра статуса UART в единичное состояние, что и означает "байт данных получен". Такой вывод я делаю на основе официального описания работы этой функции [14,31]: если не использовать "оверлапинг", то она просто-напросто должна затормозить всю систему (на самом

деле только, конечно, поток, в котором она выполняется), пока принятый байт не обнаружится. Насколько это описание соответствует действительности, посмотрим позже.

Заметки на полях

"Затормозимся" и мы на этом моменте — скажите сами, дорогой читатель, такой подход — это *правильно*? Если вам интересно, то в *приложении 4* вы можете посмотреть, как устроен аналогичный механизм в абсолютно однозадачной операционной среде микроконтроллера. Так вот, там тоже предлагается аналогичная "тормозящая" процедура, которая, однако, может при правильном построении программы прерываться, извините за тавтологию, прерываниями и в таком случае особенно ничего не тормозить. Но предлагать процедуру (по смыслу это — именно процедура, а никакая не функция), которая потенциально может намертво заморозить программу — тут я, вероятно, чего-то недопонимаю. Такие процедуры с замкнутым циклом, условие выхода из которого может не выполниться никогда — вообще нонсенс для высокоуровневого программирования. Почему было не сделать просто-напросто обычное Windows-событие по аппаратному прерыванию порта? Объяснение натянуть на эту ситуацию можно — все эти функции являются универсальными, применимыми к любой записи/чтению в файл, и все же это разработчиков, на мой взгляд, не извиняет.

Разумеется, выход из этой ситуации также предлагается — сильно напоминающий попытки достать одно известное место через другое не менее известное, но все же выход. Но прежде чем мы его рассмотрим, остановимся еще на особенностях работы `WaitCommEvent`. Предположим, мы все сделали, как написано. Что и когда вернет нам эта функция? Если верить "пособию", то она вернет нам то самое значение `EV_RXCHAR` и тогда, когда в приемном буфере окажется ровно один байт. А если они поступают быстро и программа не успевает их обрабатывать? Тогда мы обнаружим первый и второй байты (см. [32]), а об остальных просто не узнаем. И читать нужно не один байт (хотя мы и настраивались именно на это), а определять, сколько байтов в буфере, и столько же читать. Это делается через функцию с неподходящим к случаю названием `ClearCommError`, которая возвращает структуру состояния порта (`COMSTAT`), а в ней — размер буфера.

Я сейчас опишу особенности работы всех основных вариантов асинхронного чтения, как они предлагаются в различных источниках, по мере их усложнения. До собственно организации потока мы дойдем позднее, когда будет описан окончательный вариант программы, а пока предположим, что у нас уже имеется поток, в котором выполняется некий цикл асинхронного чтения, и при наступлении события `EV_RXCHAR` он посылает главной форме сообщение `wmCOMPORT`. Осталась также аналогичная описанному ранее процедура посылки команды, т. е. записи в порт через `WriteFile`. При запуске программы на экране должно отображаться все, что часы посылают, а при посылке коман-

ды — еще и принятая строка. Все это также должно работать и в первом варианте работы часов, когда они сами ничего не посылают. Из описанного получается, что в параллельном потоке можно попробовать сделать примерно вот такой замкнутый цикл (без всяких отдельных событий типа EV_RXCHAR):

```
var
 StatCOM : TComStat;
 bb:boolean;
 TMask:DWord;
. . . . .
bb:=True;
while bb do  {бесконечный цикл}
begin
 WaitCommEvent(hCOM,TMask,nil); {ожидаем приема}
 ClearCommError(hCOM,xn,@StatCOM);
          {получаем состояние COM в StatCOM}
 xn:=StatCOM.cbInQue;
   {в StatCOM.cbInQue - реальное количество байтов в буфере}
 if ReadFile(hCOM,ab,xn,xn,nil) then {читаем байты в ab}
 SendMessage(Form1.Handle,wmCOMPORT,1,0);
             {посылаем сообщение главному окну}
end;
```

Будет ли это дело работать? При запуске из-под Windows 98 — просто отлично. Мало того, хотя, судя по описанию ранее, функция WaitCommEvent должна тормозить если не всю программу (она все же в параллельном потоке), то, по крайней мере, намертво блокировать доступ к порту, пока туда ничего не поступило — этого не происходит. Если вы попробуете из главного окна послать в порт команду через WriteFile, она отлично пройдет даже и в первом варианте работы часов (во втором варианте с автоматической посылкой WriteFile должна в любом случае срабатывать, когда при приходе байта WaitCommEvent "отпускает" систему до окончания его обработки). Увидев такую картину, я даже засомневался — зря, что ли ругался, все работает на самом деле гораздо лучше, чем описано? Но ситуация разъяснилась при переходе к Windows XP — там-то все пошло, как положено и даже хуже. В частности, WriteFile, несмотря ни на какие наши Timeout, намертво "висла" и "вешала" при этом всю программу до прихода ближайшего байта, а если его так и не поступало, то Windows XP еще и услужливо сообщала в заголовке, что "программа не отвечает". Получается, что и там и там — "недокументированные особенности", но если в 98-й они нам на руку, то в XP — строго наоборот. Что делать? Придется прибегнуть к "оверлапингу".

Идея "оверлапинга" (overlapping) следующая: `WaitCommEvent` возвращает управление сразу, но мы должны теперь уже отслеживать даже не событие, заданное через установку флага `EV_RXCHAR`, а некое другое, которое устанавливается в структуре `OVERLAPPED` через функцию `CreateEvent`. Красиво получается — сообщение, о том, что было сообщение, о том, что было сообщение...

В секции **var** надо объявить структуру

```
StrOvr : TOverlapped;
```

и в `CreateFile` добавить предпоследним параметром флаг `FILE_FLAG_OVERLAPPED`. Остальные изменения в программе в связи с введением "оверлапинга" будут приведены далее. Если мы запустим программу на этой стадии внесения изменений, она не будет работать вообще (точнее, будет непрерывно выполнять пустую операцию чтения и посылки сообщения), т. к. `WaitCommEvent` теперь не ожидает байта, а сразу прерывается. Для того чтобы все, что после нее выполнялось только в нужном случае, мы сначала попробуем применить вот какой прием: не будем проверять никаких лишних сообщений, а просто дождемся, пока `WaitCommEvent` возвратит нам через `TMask` нужную величину `EV_RXCHAR`:

```
while bb do  {бесконечный цикл}
begin
 TMask:=0;
 WaitCommEvent(hCOM,TMask,@StrOvr); {ожидаем приема}
 if TMask=EV_RXCHAR then
 begin
  ClearCommError(hCOM,xn,@StatCOM);
            {получаем состояние COM в StatCOM}
  xn:=StatCOM.cbInQue;
   {в StatCOM.cbInQue - реальное количество байтов в буфере}
  if ReadFile(hCOM,ab,xn,xn,@StrOvr) then {читаем байты в ab}
  SendMessage(Form1.Handle,wmCOMPORT,1,0); {посылаем сообщение}
 end;
end;
```

Как ни удивительно, но этот абсолютно "незаконный" механизм будет работать! Байты принимаются, посылка команды работает, но... С одной оговоркой — более-менее "как надо" все это будет работать только в Windows XP. В Windows 98 же нужное событие, когда `TMask` принимает значение `EV_RXCHAR` (одно из миллионов в процессе непрерывного цикла) вполне может потеряться. На практике это не приведет к потере принятых байтов — в следующий раз считается два байта, ну и что? Но все это крайне некрасиво — а если следующего раза не будет? И вообще, наша цель — чтобы байты читались стро-

го в момент прихода, а не через час после него. Так что придется делать согласно "пособию", а чтобы у вас на отладку нормально работающей процедуры ушло не больше недели (у меня по первому разу ушло три дня), я сейчас покажу самый простой и надежно работающий вариант. В этом варианте убраны все лишние проверки, по которым все равно непонятно, что делать, и опыт многолетней эксплуатации показывает, что эти проверки не нужны (но это не значит, что нужно о них забыть, если вы, например, делаете свой компонент Delphi для распространения, см. *главу 1*):

```
while bb do  {бесконечный цикл}
begin
 if not WaitCommEvent(hCOM,TMask,@StrOvr) then
 {запускаем прием}
 if GetLastError = ERROR_IO_PENDING
 then WaitForSingleObject(StrOvr.hEvent, INFINITE);
          {ожидаем приема до бесконечности}
 ClearCommError(hCOM,xn,@StatCOM);
            {получаем состояние COM в StatCOM}
 xn:=StatCOM.cbInQue;
   {в StatCOM.cbInQue - реальное количество байтов в буфере}
 if xn > 0 then
 if ReadFile(hCOM,ab,xn,xn,@StrOvr) then {читаем байты в ab}
  SendMessage(Form1.Handle,wmCOMPORT,1,0); {посылаем сообщение}
end;
```

Теперь, зная как и что нужно делать в потоке, возьмемся за переделку программы. Новый проект COMproba я разместил в папке Glava21\2 (перенеся его из папки Glava21\1). Объявим в секции **var** новые переменные:

```
var
. . . . .
ThreadID:dword;
COMThread : THandle;
StatCOM : TComStat;
FlagCOM:boolean=False;
StrOvr : TOverlapped;
bb:boolean=True;
. . . . .
```

Я увеличил емкость массива ab до 1024 символов — хотя это в данном случае необязательно, в принципе максимум может быть равен емкости буфера, т. е. 128 байтам. В функции CreateFile второй от конца параметр (под названием dwFlagsAndAttributes — пока там стоял ноль) заменим на константу FILE_FLAG_OVERLAPPED, и во всех вызовах функций ReadFile, WriteFile и WaitCommEvent последний параметр (**nil**) заменим на указатель на объявлен-

ную структуру `@StrOvr`. В процедуру инициализации порта `IniCOM` добавим оператор:

```
StrOvr.hEvent:=CreateEvent(nil,True,False,nil);
```

Уберем с формы все компоненты для вывода (оставив только `Label7` для отображения текущего порта и `ComboBox1`), и вместо них поставим компонент `Memo`, у которого установим свойство `Color` в `clNavy`, свойство `Font` в "полужирный" 10 кегля цвета `clAqua`, в свойстве `ScrollBars` установим `ssVertical` (как ни странно, но `Memo` прокручивается по мере поступления данных автоматически и специально об этом думать не надо), а `ReadOnly` в `True`. Свойство `Lines` очистим.

Теперь напишем полностью процедуру, которая будет выполняться в потоке (она из одного приведенного ранее цикла и состоит). Назовем ее `ReadComPort`:

```
procedure ReadComPort; {это поток и есть}
var TMask:DWord;
begin
while bb do  {бесконечный цикл}
begin
 if not WaitCommEvent(hCOM,TMask,@StrOvr) then
 {запускаем прием}
 if GetLastError = ERROR_IO_PENDING
 then WaitForSingleObject(StrOvr.hEvent, INFINITE);
          {ожидаем приема до бесконечности}
 ClearCommError(hCOM,xn,@StatCOM);
            {получаем состояние COM в StatCOM}
 xn:=StatCOM.cbInQue;
   {в StatCOM.cbInQue - реальное количество байтов в буфере}
 if xn > 0 then
 if ReadFile(hCOM,ab,xn,xn,@StrOvr) then {читаем байты в ab}
  SendMessage(Form1.Handle,wmCOMPORT,1,0); {посылаем сообщение}
end;
end;
```

Ее нужно разместить в самом начале программы. Теперь нужен обработчик нашего сообщения `wmCOMPORT`. Для этого в самом начале модуля, после **uses**, вставим такое объявление:

```
const
wmCOMPORT=wm_User+11; {сообщение от порта}
```

Ниже, в секции **private** добавим:

```
{ Private declarations }
```

```
procedure ReceiveCOM(var MSG:TMessage); message wmCOMPORT;
```

Сама процедура ReceiveCOM будет выглядеть так:

```
procedure TForm1.ReceiveCOM(var MSG:TMessage);
{чтение очередного байта по сообщению wmCOMPORT}
var i:integer;
begin
ttime:=Time; {фиксируем время прихода байта}
if MilliSecondsBetween(told,ttime)>500 then
       {если больше полсекунды, очищаем строку}
st:='';
 for i:=1 to xn do st:=st+hexb(ab[i])+' ';
 if Form1.Memo1.Lines.Count>65000 then Form1.Memo1.Lines.Clear;
 {больше 64К строк нельзя}
 Form1.Memo1.Lines.Add(st); {выводим в Memo}
 told:=ttime; {для сравнения в следующий раз}
 if FlagCOM then {если это была посылка}
 begin
  FlagCOM:=False;
  st:=''; {очищаем строку}
  Timer1.Enabled:=False; {выключаем таймер}
 end;
end;
```

Здесь два момента пока непонятны — во-первых, зачем вначале очищать строку каждые полсекунды, и откуда берется для этого значение told? Очищать строку мы будем для того, чтобы принятые байты, которые разделены временным промежутком, у нас выводились построчно. Так как у нас промежуток равен 1 секунде, то полсекунды будет в самый раз. Если же они поступают подряд, то выведутся в одной строке. Значение told мы пока устанавливаем только автоматически при очередном выводе строки в Memo1. Второй момент, который связан с таймером, сейчас прояснится. Установим на форму таймер, и его свойство Timer1.Enable в False. Для события onTimer напишем такой обработчик:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin  {таймер на ошибку}
ttime:=Time;
if SecondsBetween(told,ttime)>1 then
{если через две секунды ничего нет}
begin
 Timer1.Enabled:=False;
 FlagCOM:=False;
 Application.MessageBox('Устройство не обнаружено','Error',MB_OK);
```

```
end;
end;
```

А когда мы его будем взводить? Естественно, по нажатию кнопки **Запрос**:

```
procedure TForm1.Button1Click(Sender: TObject);
begin  {запрос}
if (hCOM=0) or (hCOM=INVALID_HANDLE_VALUE) then exit;
{если порт еще не инициализирован - выход}
PurgeComm(hCOM,PURGE_RXCLEAR); {очищаем буфер на всякий случай}
xb:=$A2;
WriteFile(hCOM,xb,1,xn,@StrOvr);
st:='Послано: '+hexb(xb);
Form1.Memo1.Lines.Add(st); {записали в Memo}
st:=''; {очистили строку}
told:=Time; {зафиксировали момент посылки}
FlagCOM:=True; {обозначаем посылку}
Timer1.Enabled:=True; {запускаем таймер}
end;
```

Осталось расправиться с событием EV_RXCHAR и с потоком. Последняя операция, которая обычно так пугает новичков, проще простого. Добавим в самый конец процедуры IniCOM такие строки:

```
. . . . .
SetCommMask(hCOM,EV_RXCHAR); {отслеживаем событие в буфере}
COMThread := CreateThread(nil,0,@ReadComPort,nil,0,ThreadID);
                    {создаем поток}
told:=Time; {фиксируем время запуска}
st:=''; {очищаем строку для вывода}
. . . . .
```

Перед тем как закрывать форму или менять порт, поток надо уничтожать:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin {уничтожаем COM}
 TerminateThread(COMThread,0); {уничтожаем поток}
 CloseHandle(hCOM); {закрываем COM}
end;

procedure TForm1.ComboBox1Select(Sender: TObject);
begin
TerminateThread(COMThread,0); {уничтожаем поток}
CloseHandle(hCOM); {закрываем старый COM}
stcom:=ComboBox1.Text; {устанавливаем порт COM1,2,3,4}
IniCOM;
```

```
end;
```

Все! На основе этой программы вы спокойно можете делать любой приемопередатчик через последовательный порт. Естественно, выводить совершенно необязательно в HEX-форме, если функцию `hexb` заменить на `chr`, то выведутся символы (тут нужно быть аккуратным, т. к. вывод символа с нулевым кодом в некоторые компоненты может все порушить), а если на `IntToStr`, то выведутся десятичные числа.

Результат работы программы COMproba, как прототипа "эмулятора терминала", показан на рис. 20.2. Здесь вы видите, как часы в конце минуты выдали полную строку времени, а потом мы то же самое запросили отдельно.

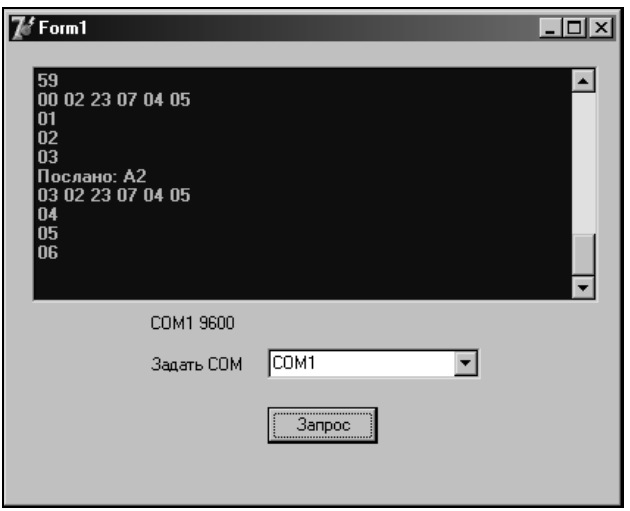


Рис. 20.2. Результат работы модернизированной программы COMproba

Но прежде чем перейти, наконец, от теоретических примеров к практическим, я хочу сделать одно замечание. Если честно, то в большинстве случаев писать самим такие программы большого смысла нет. Есть множество различных готовых процедур, в том числе — оформленных в виде компонентов Delphi. Есть и универсальные для всех сред программирования компоненты ActiveX (правда, на поставляемый с Visual Basic компонент ActiveX под названием `MSComm` требуется отдельная платная лицензия). Технологию ActiveX, которая позволяет работать и с COM и с USB, и даже прямо с GPS-навигаторами, которыми мы займемся далее, мы в этой книге обходим — это отдельная обширная тема, к тому же подход этот по причинам, отчасти изложенным в *главе 18*, не является оптимальным. Теоретически мы при использовании любых сторонних компонентов теряем в гибкости — если вы изучи-

те возможности использованных нами ранее функций подробнее [31], то увидите, что они позволяют осуществить очень много разнообразных вариантов. Другой вопрос — что гибкость эта не особенно и требуется, универсальные процедуры побайтного приема/передачи вполне успешно работают в большинстве случаев. И сейчас вы сами сравните, что проще — прямое программирование через API или использование таких компонентов.

Прием и передача данных с помощью компонента *AsyncFree*

Один из самых удачных и профессионально сделанных компонентов для COM-порта (а я их перепробовал достаточно много), на мой взгляд, — свободно распространяемый `AsyncFree` некоего Петра Вониса (Petr Vones), судя по e-mail, из Чехии. Компонент доступен бесплатно, с исходными кодами и скачанный архив включает в себя, в том числе, и файлы DPK, что упрощает процедуру установки до предела — нужно просто щелкнуть мышью на том из этих файлов, который соответствует имеющейся у вас версии Delphi, и компонент установится самостоятельно, без утомительных процедур ручной инсталляции. Хотя к самому компоненту приложена ссылка на (отличный, кстати) сайт Delphree Open Source Initiative (**http://delphree.clexpert.com**), однако на нем вы найдете только старую версию `AsyncFree` под Delphi 5, а скачивать последние версии лучше отсюда: **http://sourceforge.net/project/showfiles.php?group_id=20226**. Все эти адреса я привожу на всякий случай, потому что архив с компонентом есть на диске (в папке Glava21\AsyncFree). Все его возможности я вам не покажу — это нереально, да и большинство из них вряд ли пригодится, на его основе мы сейчас просто сделаем вариант той же самой программы COMproba.

Перенесем проект из папки Glava3\2 в папку Glava3\3. Считаем, что компонент установлен и находится в палитре компонентов на вкладке `AsyncFree`. На самом деле там образуется много компонентов, и нам требуется самый первый из них под названием `AfComPort`. Установим его на форму. Из перечня переменных удалим все, что касается файлов, потоков и т. д., добавив лишь еще один флаг и переменную `tall` целого типа:

```
var
Form1: TForm1;
xb:byte;
ab: array[1..1024] of byte;
st,stcom:string;
ttime,told:TDateTime;
```

```
FlagCOM:boolean=False;
FlagSend:boolean=False;
tall:integer;
. . . . .
```

Начнем с того, что перепишем процедуру IniCOM:

```
procedure IniCOM;
var i, err :integer;
begin
FlagCOM:=False;
Form1.Label7.Caption:='COM?';
{инициализация COM - номер в строке stcom}
Form1.AfComPort1.Close; {закрываем старый COM, если был}
val(stcom[length(stcom)],i,err);
if err=0 then Form1.AfComPort1.ComNumber:=i else exit;
                        {номер порта}
Form1.AfComPort1.BaudRate:=br9600; {скорость 9600}
try
 Form1.AfComPort1.Open;
except
 if not Form1.AfComPort1.Active then {если не открылся}
 begin
  st:=stcom+' does not be present or occupied.';
  Application.MessageBox(Pchar(st),'Error',MB_OK);
  exit {выход из процедуры - неудача}
 end;
end;
ab[1]:=ord('A'); {будем посылать инициализацию модема}
ab[2]:=ord('T');
ab[3]:=13;{CR}
ab[4]:=10;{LF}
for i:=1 to 4 do Form1.AfComPort1.WriteData(ab[i],1);
{ответ не сразу:}
Form1.Timer1.Enabled:=True;
tall:=0;
while tall<1 do application.ProcessMessages; {пауза в 1 с}
Form1.Timer1.Enabled:=False;
st:=Form1.AfComPort1.ReadString; {ответ модема 10 знаков}
if pos('OK',st)<>0 then {модем}
begin
   st:=stcom+' занят модемом';
   Form1.Timer1.Enabled:=False;
   Application.MessageBox(Pchar(st),'Error',MB_OK);
```

```
   exit;
end else {все нормально, открываем COM}
begin
   Form1.Label7.Caption:=stcom+' 9600';
   st:=''; {очищаем строку для вывода}
   FlagCOM:=True;
end;
end;
```

Как видим, процедура создания порта много понятнее, чем в том случае — все через привычную установку свойств компонента. FlagCOM теперь играет у нас роль индикатора — доступен порт или нет. При определении модема здесь применен хитрый способ задания паузы — вместо обычного оператора Sleep, который тормозит программу, мы использовали таймер. Чтобы это сработало, в процедуре onTimer при этом достаточно добавить строку inc(tall) (далее мы покажем эту процедуру полностью).

Как только мы обратились к процедуре AfComPort1.Open, у нас немедленно будет создан параллельный поток и весь прием пойдет через него. Поэтому, чтобы при определении модема принятые байты не обрабатывались, надо не забыть добавить в процедуру приема выход по условию FlagCOM=False.

Для создания этой процедуры удаляем процедуры ReadComPort и ReceiveCOM. Вместо них обычным способом — через инспектор объектов — создадим обработчик события AfComPort1DataRecived[3].

```
procedure TForm1.AfComPort1DataRecived(Sender: TObject;
                  Count: Integer);
{чтение очередного байта по сообщению wmCOMPORT}
var i:integer;
begin
if FlagCOM=False then exit; {если модем еще не опрошен}
if count<>0 then {если что-то принято}
begin
{если порт существует}
 ttime:=Time; {фиксируем время прихода байта}
 if MilliSecondsBetween(told,ttime)>500 then
        {если больше полсекунды, очищаем строку}
 st:='';
 AfComPort1.ReadData(ab,count); {читаем буфер}
```

[3] Написание слова "receive" есть непреодолимое препятствие для любого, кто не является носителем английского языка. Каких только вариантов не встретишь на просторах Сети! Как видим, наш чех Петя Вонис также не избежал общей участи.

```
 for i:=1 to count do
 st:=st+hexb(ab[i])+' ';
 if Form1.Memo1.Lines.Count>65000 then Form1.Memo1.Lines.Clear;
 {больше 64К строк нельзя}
 Form1.Memo1.Lines.Add(st); {выводим в Memo}
 told:=ttime; {для сравнения в следующий раз}
 if (SecondsBetween(told,ttime)<1) and FlagSend then
                  {если это была посылка}
 begin
  FlagSend:=False;
  st:=''; {очищаем строку}
  Timer1.Enabled:=False; {выключаем таймер}
 end;
end;
end;
```

Как видим, она почти не отличается от ReceiveCOM. На самом деле условие count<>0 не требуется, оно введено просто ради порядка. Осталось только переписать остальные процедуры, они изменятся в сторону упрощения:

```
procedure TForm1.Button1Click(Sender: TObject);
begin  {запрос}
if FlagCOM=False then exit;
{если порт еще не инициализирован - выход}
AfComPort1.PurgeRX; {очищаем буфер на всякий случай}
xb:=$A2;
AfComPort1.WriteData(xb,1); {посылаем команду}
st:='Послано: '+hexb(xb);
Form1.Memo1.Lines.Add(st); {записали в Memo}
st:=''; {очистили строку}
told:=Time; {зафиксировали момент посылки}
FlagSend:=True; {обозначаем посылку}
Timer1.Enabled:=True; {запускаем таймер}
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
{инициализация COM1 при запуске}
stcom:='COM1';
IniCOM;
end;

procedure TForm1.ComboBox1Select(Sender: TObject);
```

```
begin
stcom:=ComboBox1.Text; {устанавливаем порт COM1,2,3,4}
IniCOM;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin  {таймер на ошибку}
ttime:=Time;
inc(tall);
if FlagCOM=False then exit;{запрос модема}
if SecondsBetween(told,ttime)>1 then
       {если через две секунды ничего нет}
begin
 Timer1.Enabled:=False;
 FlagSend:=False;
 Application.MessageBox('Устройство не обнаружено','Error',MB_OK);
end;
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
AfComPort1.Close; {закрытие порта}
end;
```

Из этого текста мы видим, что программа не стала существенно проще с точки зрения объема кода, но зато особенности реализации различных механизмов вроде "оверлапинга" тут знать не требуется. Так что выбирайте сами.

Программа для чтения данных с GPS-навигатора

Теперь создадим для примера вполне практическую программу для общения компьютера с GPS-навигатором по протоколу NMEA. В качестве прототипа используем процедуру с использованием компонента AsyncFree, но читатель, при нежелании устанавливать этот компонент, без сомнения, легко перепишет ее с использованием функций API. Тем более что общение с устройством тут полностью одностороннее.

Сначала о протоколе NMEA. Хотя эта программа испытывалась с навигатором Garmin eTrex, никакой разницы нет — любой навигатор, от самых сложных профессиональных приборов до дешевых ручных устройств поддерживает этот протокол. Для этого его нужно установить в меню установок прибора — по умолчанию там, как правило, установлен фирменный протокол (для фирмы Garmin это и есть упоминавшийся ранее протокол Garmin). Хотя

фирменный протокол имеет более широкие возможности (см. [27]), но NMEA обычно достаточно, чтобы извлечь из навигатора практически всю информацию, которую он умеет выдавать. Протокол этот был разработан Национальной ассоциацией морской электроники (National Marine Electronics Association, NMEA) еще до появления GPS и под названием NMEA0183 (IEC1162) стал в настоящее время мировым стандартом. Крупнейшим недостатком протокола обычно называют невозможность присоединения более чем одного устройства к одному порту.

Для общения с навигаторами есть, как уже упоминалось, специальный компонент ActiveX, но мы обойдемся без него. Технические характеристики NMEA по умолчанию всегда такие: скорость 4800, формула передачи 8n1, передача производится ежесекундно. Частичное описание протокола на русском есть, например, здесь [28]. Полностью его можно найти на сайте самой NMEA за большие деньги, но и частичного описания для подавляющего большинства нужд более чем достаточно. Вот что каждую секунду выдает у меня навигатор Garmin eTrex:

```
$GPRMC,154140,A,5552.6225,N,03733.3341,E,0.0,0.0,080405,9.5,E,A*1E
$GPRMB,A,,,,,,,,,,,A,A*0B
$GPGGA,154138,5552.6224,N,03733.3334,E,1,04,3.5,268.5,M,15.4,M,,*45
$GPGSA,A,2,,,04,13,,,23,24,,,,,3.7,3.5,1.0*31
$GPGSV,3,2,10,16,13,111,00,20,69,147,00,23,69,247,43,24,49,246,47*72
$GPGLL,5552.6224,N,03733.3334,E,154138,A,A*4B
$GPBOD,,T,,M,,*47
$GPVTG,0.0,T,350.5,M,0.0,N,0.0,K*4D
$PGRME,42.2,M,49.1,M,64.8,M*1C
$PGRMZ,489,f*01
$PGRMM,WGS 84*06
$HCHDG,147.3,,,9 5,E*24
$GPRTE,1,1,c,*37
```

Каждое сообщение начинается с символа `$` и заканчивается переводом строки (<CR>+<LF>, т. е. символами с номерами 13 и 10), отдельные поля разделяются запятыми. Последний байт перед этими символами (`1E` в первой строке) — контрольная сумма. Мы ее отслеживать не будем. Если навигатор какую-то величину "не знает", то соответствующее поле между запятыми будет пустым. Забавно, что между одинаковыми полями сообщений имеются некоторые разночтения, которые, очевидно, вызваны тем, что сообщения формируются в разное время, хотя и выдаются единовременно.

Все это море информации нам не требуется, давайте ограничимся простой задачей: вывести в окошки координаты места (в нормальных единицах широ-

ты/долготы), высоту места над уровнем моря и точное время по Гринвичу. Для этого достаточно проанализировать, например, сообщение `$GPGGA`.

Первое поле в этом сообщении — время по Гринвичу (Greenwich Mean Time, GMT), второе и третье — широта (`N` — значит "северная", North), четвертое и пятое — долгота (`E` — значит "восточная", East). Высота места над уровнем моря — поле перед первым по ходу строки символом `M`, который обозначает единицу ее измерения (метр, Meter). Надо так понимать, что могут быть и футы, но это явно должно устанавливаться в самом навигаторе, так что будем рассчитывать на метры, в крайнем случае программа просто не выведет ничего. Разберемся с представлением координат — что это за числа: 5552.6225, 03733.3341? Оказывается, это десятичные градусы, точнее — минуты. Первые два (или три для долготы) знака есть целые градусы, еще два до точки — минуты, остальное есть секунды, выраженные в десятичных долях вместо шестидесятеричных. Для того чтобы перевести их в привычные секунды, достаточно дробную часть умножить на 60 — получим секунды с сотыми долями.

Заметки на полях

Кстати, любопытно, а какова получается разрешающая способность такого представления координат (четыре разряда минут после запятой), если ее выразить в метрах? По широте на этот вопрос можно ответить более-менее однозначно: 1 минута широты (угловая минута дуги меридиана) есть одна морская миля, т. е. 1852 метра (отсюда 1 градус — примерно 111 км). Тогда одна десятитысячная часть составит примерно 20 см (а одна секунда широты — 31 м, а ее сотая часть — 31 см, как видите, при переводе долей десятичных минут в угловые секунды мы теряем в разрешении). На самом деле это цифры приблизительные, потому что при таких точностях надо учитывать форму геоида, навигаторы делать это умеют, и в сообщениях NMEA информация для этого содержится, но углубляться в этот вопрос здесь мы не будем. Заметим только, что для долготы уже такой однозначности нет — к полюсам меридианы сходятся, на широте 60 градусов длина широтной дуги становится в два раза меньше меридиональной. Для прикидок в уме можно принимать, что для средних широт 40—50° длина 1 минуты по долготе примерно в 1,5 раза меньше, чем по широте. А методика более точных расчетов по данным GPS-навигатора имеется на множестве ресурсов в Сети — дисциплины геодезия и картография, ранее бывшие прерогативой отдельных специалистов с военным и геологическим уклоном, теперь пришли в широкие массы, которые раньше даже и не подозревали о существовании какого-нибудь эллипсоида Красовского.

Разобрались, теперь можно писать программу. Для этого создадим новый проект под названием GPSNavigate (в папке Glava21\4), установим на форму то же самое `Memo` с теми же свойствами, как и в проекте COMProba ранее (только уберем у него линейку прокрутки: `ScrollBars = ssNone`), и тот же самый `AfComPort`. Из визуальных компонентов нам потребуется еще только `ComboBox` для выбора порта, который мы тоже скопируем из проекта-пробы. При инициализации порта стоит учесть, что в каждой посылке навигатора

может быть порядка 500 символов, и на всякий случай установить буфер побольше (пусть это будет 1024 байта).

В секции объявления переменных мы запишем следующее:

```
var
Form1: TForm1;
st,stcom:string;
FlagCOM:boolean=False;
FlagGPS:boolean=False;
tall:integer;
```

А процедура инициализации порта будет выглядеть так:

```
procedure IniCOM;
var i, err :integer;
begin
FlagCOM:=False;
st:='COM?';
Form1.Memo1.Lines.Clear;
Form1.Memo1.Lines.Add(st);
{инициализация COM - номер в строке stcom}
Form1.AfComPort1.Close; {закрываем старый COM, если был}
val(stcom[length(stcom)],i,err);
if err=0 then Form1.AfComPort1.ComNumber:=i else exit;
                    {номер порта}
Form1.AfComPort1.BaudRate:=br4800; {скорость 4800}
Form1.AfComPort1.InBufSize:=1024; {емкость буфера}
try
 Form1.AfComPort1.Open;
except
 if not Form1.AfComPort1.Active then {если не открылся}
 begin
  st:=stcom+' does not be present or occupied.';
  Application.MessageBox(Pchar(st),'Error',MB_OK);
  exit {выход из процедуры - неудача}
 end;
end;
st:='AT'+#13+#10; {будем посылать инициализацию модема}
Form1.AfComPort1.WriteString(st); {ответ не сразу}
Form1.Timer1.Enabled:=True;
tall:=0;
while tall<1 do application.ProcessMessages; {пауза в 1 с}
Form1.Timer1.Enabled:=False;
st:=Form1.AfComPort1.ReadString; {ответ модема 10 знаков}
FlagGPS:=False; {это еще не проверка GPS}
```

```
if pos('OK',st)<>0 then {модем}
begin
   st:=stcom+' занят модемом';
   Form1.Timer1.Enabled:=False;
   Form1.Memo1.Lines.Clear;
   Form1.Memo1.Lines.Add(st);
   exit;
end else {все нормально, открываем COM}
begin
   Form1.Memo1.Lines.Clear;
   st:=''; {очищаем строку для вывода}
   FlagCOM:=True;
   tall:=0;
   Form1.Timer1.Enabled:=True; {запускаем опять таймер}
   FlagGPS:=True; {проверка GPS}
end;
end;
```

Видите, насколько удобнее пользоваться готовым компонентом — захотели послать целую строку — послали без дополнительных раздумий (см. в тексте процедуры определение модема), захотели отдельный байт — также послали (на самом деле можно посылать и отдельный символ, как символ). То же самое касается и приема — байтовый массив нам тут держать уже нужды нет.

Процедура по событию таймера будет выглядеть так:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
inc(tall);
if (tall>2) and (FlagGPS=True) then {устройство не обнаружено}
begin
 st:='Устройство не обнаружено';
 Form1.Memo1.Lines.Clear;
 Form1.Memo1.Lines.Add(st);
end;
end;
```

То есть таймер работает все время (отсчитывая секунды в переменной `tall`, т. к. величину свойства `Interval` мы не меняли, она по умолчанию 1 с и равна). Для того чтобы теперь отслеживать контакт с устройством, достаточно каждый раз обнулять переменную `tall` в процедуре приема — как только промежуток между поступлениями данных превысит 2 с, программа "завопит", что "устройство не обнаружено" и автоматически возобновит прием, как только данные пойдут опять.

Прежде чем перейти к процедуре приема, создадим все остальные необходимые процедуры, они тут будут очень просты:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
stcom:='COM1';
IniCOM;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
AfComPort1.Close; {закрытие порта}
end;

procedure TForm1.ComboBox1Select(Sender: TObject);
begin
Form1.Timer1.Enabled:=False; {останавливаем таймер}
stcom:=ComboBox1.Text; {устанавливаем порт COM1,2,3,4}
IniCOM;
end;
procedure TForm1.Memo1Change(Sender: TObject);
begin
HideCaret(Form1.Memo1.Handle); {скрываем курсор}
end;
```

Как видите, для красоты мы добавили процедуру скрытия текстового курсора. И наконец, самая главная процедура приема данных, анализа строк сообщения и вывода результатов на экран:

```
procedure TForm1.AfComPort1DataRecived(Sender: TObject; Count: Integer);
var st1:string;
 sek:real;
 i,err:integer;
begin
 if FlagCOM=False then exit; {если модем еще не опрошен}
 tall:=0; {устройство обнаружено}
 st:=st+Form1.AfComPort1.ReadString; {посылка GPS }
 if pos('$',st)=0 then begin st:=''; exit end;
 {если знак $ имеется}
 delete(st,1,pos('$',st)-1); {до знака $ все удаляем}
 if length(st)<6 then exit; {накапливаем }
 if pos('GPGGA',st)=0 then begin st:=''; exit end;
                 {не то сообщение}
 if pos(chr(13)+chr(10),st)<>0 then {конец сообщения GPGGA}
```

```
 begin
  {анализ сообщения}
  Memo1.Lines.Clear;
{st:='$GPGGA,154138,5552.6224,N,03733.3334,E,1,04,
3.5,268.5,M,15.4,M,,*45';}
  st1:=copy(st,pos(',',st)+1,6);{время}
  if pos(',',st1)<>0 then {нет данных}
  begin
   st1:='Нет данных';
   Memo1.Lines.Add(st1);
   st:='';
   exit;
  end;
  insert(':',st1,5);
  insert(':',st1,3);
  Memo1.Lines.Add('Время GMT: '+st1);
  delete(st,1,pos(',',st));
  delete(st,1,pos(',',st)); {широта}
  st1:=copy(st,pos('.',st),4); {дес. секунды}
  val(st1,sek,err);
  sek:=sek*60; {обычные секунды}
  str(sek:5:2,st1);
  st1:=st1+'"';
  st1:=copy(st,1,pos('.',st)-1)+st1; {широта градусы + минуты}
  insert('`',st1,5);
  insert('°',st1,3);
  delete(st,1,pos(',',st)); {какая?}
  st1:=st1+' '+copy(st,1,1); {N}
  Memo1.Lines.Add('Широта: '+st1);

  delete(st,1,pos(',',st)); {долгота}
  st1:=copy(st,pos('.',st),4); {дес. секунды}
  val(st1,sek,err);
  sek:=sek*60; {обычные секунды}
  str(sek:5:2,st1);
  st1:=st1+'"';
  st1:=copy(st,1,pos('.',st)-1)+st1; {долгота градусы + минуты}
  insert('`',st1,6);
  insert('°',st1,4);
  delete(st,1,pos(',',st)); {какая?}
  st1:=st1+' '+copy(st,1,1); {E}
  Memo1.Lines.Add('Долгота: '+st1);
  {теперь найдем высоту над уровнем моря:}
  st1:='';
```

```
  i:=pos('M',st)-2;
  while st[i]<>',' do begin st1:=st[i]+st1;i:=i-1; end;
  Memo1.Lines.Add('Высота над уровнем моря: '+st1+' м');
  st:='';
 end;
end;
```

Закомментированная строка с образцом сообщения `$GPGGA` предназначена для того, чтобы было удобнее отлаживать программу. В остальном я подробно описывать процедуру не буду — там все понятно из комментариев в тексте.

Результаты работы программы с "живым" GPS-навигатором показаны на рис. 20.3. Кстати, если вы захотите направить по этим координатам ракету класса "земля-земля", то имейте в виду, что ошибка определения координат моей форточки составляла в этом сеансе не меньше 140 метров, а из стратегически важных объектов в этом радиусе имеется только частная "Автомойка" и небольшой ларек с чипсами и пивом.

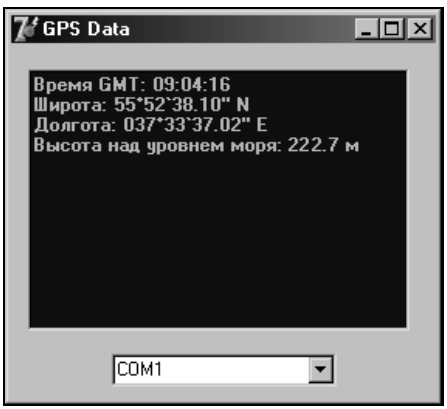


Рис. 20.3. Результат работы программы GPSNavigate

Эмуляция COM-порта через шину USB

Еще не так давно считалось, что протокол обмена данными по USB настолько сложен, что его реализовать под силу только далеко не рядовым специалистам. Но спрос рождает предложение. Вероятно, самое удобное на сегодняшний день решение для эмуляции последовательного порта через USB предлагает английская (точнее шотландская) фирма Future Technology Devices International Ltd — FTDI. К ее устройствам (см. также *приложение 4*) прилагаются бесплатные и свободно распространяемые драйверы под все основные

ОС, в том числе и под Windows семейств NT и 9*x*. Разновидностей таких драйверов две — это VCP- и D2XX-драйверы.

VCP на самом деле означает Virtual Communication Port — этот драйвер просто-напросто транслирует все стандартные функции Win32 API, которые мы использовали ранее, в необходимые команды для USB и через микросхему FT232BM (или аналогичные ей, находящиеся в устройстве) опять преобразует их в битовые последовательности UART. При подключении устройства в системе возникает новый COM-порт, и все описанные ранее программы без переделок будут работать через него. Это касается и случая, когда используется фирменный "шнурок" USB/RS-232, и когда вы сами нечто подобное изобрели *(см. приложение 4)*. Мало того, в Windows XP подобный драйвер уже есть, и там ничего вообще устанавливать не надо — единственная разница в том, что виртуальный COM, в отличие от реального, не будет виден в системе, пока вы свое устройство не подключите к USB.

Единственный серьезный вопрос, который может возникнуть в случае использования VCP-драйверов, касается скорости передачи — если уж USB, то хочется использовать хоть малую долю возможностей этой шины, между тем стандартно COM-порт через структуру `DCB` может устанавливать скорость в 256 000 бод. Есть и приемы установки для COM-порта нестандартной скорости обмена, но вопрос о том, насколько это корректно для данного случая, лично для меня, ввиду новизны ситуации, пока открыт (официальная документация не слишком уверенно утверждает, что можно использовать только стандартные скорости). Второй — менее серьезный — недостаток VCP заключается в том, что вы не можете через него работать с встроенной в FT-устройство EEPROM, в которой записан идентификатор устройства и прочие необходимые для USB "прибамбасы".

В этих случаях следует использовать D2XX-драйвер, работа с которым не сложнее, чем с разобранным ранее компонентом `TAfComPort`, но программы, конечно, придется переписывать. С установкой D2XX-драйвера связана одна совершенно глупая трудность (так и хочется написать — "характерная для продуктов MS") — как указывалось ранее, в Windows XP VCP-драйвер уже есть, и для установки D2XX-драйвера нужно устраивать некоторые "пляски с бубном", цитирую из русского переложения фирменных рекомендаций по этому поводу (**http://www.efo.ru/doc/Ftdi/Ftdi.pl?1046**):

"Немного сложнее обстоит дело в случае использования операционной системы Windows XP, которая уже имеет в своем составе сертифицированные VCP-драйверы FTDI. При попытках присоединить к компьютеру новое USB-устройство со стандартными идентификаторами FTDI (например, любой DLP-модуль) система по умолчанию, не спрашивая пользователя, самостоятельно установит VCP-драйверы. Пользователю, желающему работать с

D2XX-драйверами, необходимо в этот момент вспомнить, что очень полезно воспитывать в себе терпение и воспользоваться утилитой ftxprcvr.exe, входящей в состав дистрибутива D2XX-драйверов для Windows XP. Утилита ftxprcvr.exe, используя установившиеся по умолчанию VCP-драйверы, перепрограммирует внешнюю EEPROM, используемую в присоединенном устройстве, и задаст новые значения идентификаторов (VID=0403 и PID=6006). После этого необходимо повторить процедуру установки D2XX-драйверов сначала, т. е. отключить и снова присоединить устройство. Теперь система даст возможность пользователю указать директорию для установки D2XX-драйверов."

Перепрограммировать EEPROM в устройстве USB на основе микросхем FT232BM через свою программу не обязательно, для этого есть фирменная утилита EditEEPROM (**http://www.efo.ru/doc/Ftdi/Ftdi.pl?798**). Приводить примеры программ для работы с D2XX-драйвером я не вижу никакого смысла — работающего устройства под рукой и у меня, и у вас нет, а просто переписывать фирменную документацию — только бумагу переводить, она уже не однажды переписана и переведена на русский, см., например, цитированную ранее статью или книгу [29]. На сайте екатеринбургской фирмы "Институт радиотехники" есть, в том числе, работающий пример проекта на Delphi (**http://www.institute-rt.ru/ftdi/d2xxappl.zip**) — а вот сами драйверы с данного сайта скачивать не следует, на момент написания этих строк там лежит устаревший вариант, лучше обратиться к первоисточнику (**http://www.ftdichip.com**) или на упоминавшийся сайт компании "Эфо" (**http://www.efo.ru/doc/Ftdi/Ftdi.pl?784**).

О некоторых особенностях обработки поступающих через последовательный порт данных мы поговорим в следующей главе.

ГЛАВА 21

Массивы и память

Работа с большими массивами информации

Закладывая что-то в память ЭВМ, помните, куда вы это положили.

Первая компьютерная аксиома Лео Бейзера

Это единственная глава в книге, кроме первой, которая посвящена в основном болтовне, а не созданию практических примеров. Задача ее — дать читателю представление о том, как можно обращаться с потоками данных, поступающих в компьютер в реальном времени. Собственно задача будет состоять в том, что мы хотим не просто сохранить эти данные, но еще и по ходу дела их обрабатывать с произвольным доступом к каждому элементу образующегося массива. Мы не будем рассматривать специальные случаи мегабитных скоростей поступления данных, типа обработки видео- или аудиопотока, т. к. в подобных случаях все равно придется изучать специальную литературу. Примем, что типичная задача читателя этой книги состоит в том, чтобы принимать, записывать на диск и представлять в графической форме побайтно поступающие данные со скоростями, не превышающими возможности стандартного последовательного порта (256 Кбит или 32 Кбайт в секунду, см. *главу 20*). И основная проблема при этом состоит в организации динамических массивов — ведь, кроме "официального" типа данных под таким названием, есть и другие способы сохранения в памяти наперед неизвестного количества данных с произвольным доступом к ним.

Различные способы организации динамических массивов

Я терпеть не могу язык C за его абсолютно бесчеловечный синтаксис (попробуйте с ходу расшифровать, например, нечто вроде `**p++^=q++=*r-s`) и некоторые семантические особенности (вольное обращение с понятием функции и отмене в связи с этим процедур), но когда речь заходит о массивах, мне

иногда хочется перейти на C. Дело в одной только особенности: в C ты никогда не забываешь, что указатель — это просто число. И как число, его можно складывать и вычитать с любым другим числом. То есть в принципе любой указатель в C есть готовый массив — нужно только, разумеется, позаботиться о выделении соответствующего количества памяти.

Строка типа *PChar*

Если вы усвоите сказанное в предыдущем абзаце, то у вас не будет проблем со строками `PChar` в Delphi (см. также примечание на этот счет в *главе 14*), которые есть ни что иное, как C-указатель на символ (а именно, на самый первый в строке символ с номером 0), его определение и выглядит так: `PChar = ^char`. Идея, легшая в основу ASCIIZ-строк, была проста: видимо, когда памяти у компьютеров было мало, объем программ исчислялся байтами, и программист мог удержать в голове, что и где у него располагается (см. эпиграф), возникла мысль хранить строковые данные так, чтобы вообще не надо было знать, сколько они места занимают. Просто считываешь до первого встреченного нуля, и все. Не знаю, так ли это на самом деле, только в современных условиях за памятью все равно приходится следить. Но язык C весь построен на идее указателей[1], так что в любых языках, основанных на его синтаксисе, все строки являются нуль-терминированными.

Мы уже неоднократно отмечали различные сложности и ограничения при работе с типом `PChar`. Безоговорочно его использовать можно только, если он изначально создается именно как `PChar`, иначе замыкающего нулевого символа может и не оказаться. Или вот еще пример кода, который выглядит корректным, но выдаст знакомую ошибку "Access Violation":

```
function IntToPChar (n :Integer) :pChar;
var st:String;
begin
st:=IntToStr(n);
Result:=PChar(st);
end;
. . . . .
ShowMessage(IntToPChar(100)); {"Access Violation . . ."}
. . . . .
```

Здесь строка `st` в момент выхода из функции уничтожается, и ссылка на нее становится указывающей в никуда. В данном случае достаточно `st` объявить глобальной переменной, но все равно могут быть и другие ошибки, по этому

[1] В ассемблере это называется "косвенная адресация".

поводу см., например, [39]. И — реплика в сторону — оцените, насколько удобнее работать с паскалевскими строками: хотя нуль-терминированные строки вроде бы гибче, но правильное их использование требует от программиста гораздо более высокой квалификации. Низкий наш поклон Никлаусу Вирту!

PChar — единственный тип указателей в языке Pascal, с которым, как и в C, можно выполнять арифметические операции. Например, в результате следующего фрагмента кода:

```
pst,pch:PChar;
. . . . .
pst:='vasya';
pch:=pst+1;
```

вы получите pch = 'asya'. При простом переборе в цикле значений pst от 0 до 4 вы получите указатели поочередно на все символы строки. Поэтому в принципе использовать строку типа PChar, как массив с произвольным доступом, было бы очень удобно, но все портит особое значение нулевого символа, как конца строки. В результате присваивания pst:='vas'+#00+'ya' вы получите, к сожалению, только строку 'vas', остальное пропадет. Так как мы считаем, что в потоке данных могут содержаться любые коды символов, то этот способ нам в общем случае не подойдет.

На каждую хитрую гайку... или нетипизированные указатели, как способ организации массивов

Иметь в Pascal произвольный доступ к массиву байтов по указателю любого вида путем простых арифметических операций с ним, как и в языке C, все же можно, правда, несколько "кривым" путем. Мы этот способ использовали в *главе 14*, когда перебирали символы в массиве, считанном из дискового файла. Для этого нужно просто приводить указатель к численном виду integer, longint (в 32-разрядном Pascal это совершенно идентичные типы) или dword, а потом обратно, путем непосредственного преобразования типов:

```
pb:=pointer(integer(pFile)+1);
```

После выполнения такого оператора в нетипизированном указателе pb окажется ссылка на второй байт, считая от pFile. Разумеется, ответственность за то, чтобы не залезть при таких операциях на чужую делянку в памяти, целиком ложится на программиста, как, впрочем, и во всех подобных случаях организации индексированного доступа к элементам любого массива (строки, обычного массива, динамического массива, объекта-потомка TStringList и т. п.). Поэтому, для того чтобы ваша программа не демонстрировала поль-

зователю синие экраны в стиле "Гибель Помпеи", размеры выделенной памяти нужно своевременно корректировать. Менеджер памяти Delphi (он тут свой, в обход менеджера памяти Windows) предполагает, что программа часто выделяет и освобождает блоки разного размера, но постепенное увеличение одного и того же блока может вызвать разные неприятные последствия, типа утечки памяти неизвестно куда (а на самом деле — вследствие фрагментации кучи). Но самым главным недостатком механизма постепенного увеличения размера однажды выделенного блока памяти является снижение производительности программы по мере увеличения размеров блока. Происхождение этого явления понятно: при выделении блока процедурами типа `GetMem` память выделяется линейно, т. е. весь выделенный кусок занимает непрерывную последовательность адресов (иначе бы описанная ранее манипуляция с указателями-числами была бы невозможна). Потому если вы увеличиваете размер текущего блока памяти, и при этом места для расширения не хватает, то выделяется новый блок памяти, куда переносится содержимое старого (см. также описание особенностей работы процедур `delete` и `insert` в *главе 14*). Естественно, что данный процесс сильно замедляет работу. Особенно глупо увеличивать память мелкими блоками — многоступенчатый механизм выделения памяти устроен так, что выделяется (но не всегда используется) все равно больше, чем надо.

Ну, и как же физически все это делается в Delphi? Первоначально память выделяется с помощью `GetMem` по нетипизированному указателю типа `pointer`, а последовательное увеличение блока может осуществляться с помощью редко упоминаемой в учебниках процедуры `ReallocMem` (аналога `realloc` в C), которая вызывается точно так же, как и `GetMem`:

```
ReallocMem(var P : Pointer; NewSize: Integer);
```

Здесь `P` — тот же самый указатель, который был уже использован при вызове `GetMem`, а `NewSize` — новый размер блока. Причем если указатель `P` не был инициализирован ранее, то `ReallocMem` просто сработает, как `GetMem`. Освобождать через `FreeMem`, естественно, следует количество памяти, которое соответствует последнему вызову `ReallocMem`.

Однако по причинам, указанным ранее, часто использовать `ReallocMem` нежелательно. Гораздо лучше заранее зарезервировать блок памяти побольше и в Windows в этом ничего страшного в принципе нет (в отличие от DOS, где все время приходилось следить за тем, чтобы не вылезти за пределы свободной памяти) — если вы даже вызовете `GetMem` с просьбой выделить 300 Мбайт при имеющемся ОЗУ в 256 Мбайт, то программа не рухнет и остальные программы не прекратят работу. Но, по мере того, как память эта будет реально заполняться, все операции, разумеется, будут совершаться через SWAP-файл, и система будет становиться все более "задумчивой". Практически такие операции следует организовывать так: следует ограничить массив в памяти раз-

мером в несколько мегабайт, а при его превышении сбрасывать избыток в дисковый файл самостоятельно, не перекладывая это на Windows. Как это лучше делать на практике, причем сохраняя доступ ко всему массиву, который к тому же непрерывно пополняется, мы рассмотрим далее.

Через механизм нетипизированных указателей можно в принципе получить массив любых типов переменных, необязательно байтов. Вот как можно решить задачу расположения в памяти массива вещественных чисел типа double и последующего чтения этого массива. Я, как и обещал ранее, не располагаю соответствующий проект на диске — он очень прост. Поместите на форму в новом проекте кнопку Button и компонент Memo. При создании формы мы будем записывать в массив по нетипизированному указателю четыре вещественных числа, представляющих собой последовательные значения π, 2π, 3π и 4π, а потом по нажатию кнопки считывать их в Memo. Для этого нам потребуются следующие переменные и типы:

```
type
  dubp=^double; {указатель на число типа double}
var
  Form1: TForm1;
  pp:pointer; {нетипизированный указатель}
  pd:dubp; {указатель на число типа double}
  pint:integer; {указатель-число}
  db:double; {число типа double, занимает 8 байт в памяти}
  i:integer; {счетчик}
. . . . .
```

Процедура записи при создании формы будет выглядеть так:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  getmem(pp,32);
  pint:=longint(pp);
  for i:=1 to 4 do
  begin {сохраним в памяти четыре значения
 типа double: пи, 2пи, 3пи и 4пи}
     pd:=dubp(pint+(i-1)*8);
     db:=3.14159*i;
     pd^:=db;
  end;
end;
```

А процедура чтения при нажатии на кнопку Button1 такова:

```
procedure TForm1.Button1Click(Sender: TObject);
var st:string;
```

```
begin
  Memo1.Lines.Clear;
  for i:=1 to 4 do
  begin {извлекаем четыре значения double}
     pd:=dubp(pint+(i-1)*8);
     db:=pd^;
     str(db:10:5,st);
     Memo1.Lines.Add(st);
  end;
end;
```

Не забудем также освободить память:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
     freemem(pp,32);
end;
```

При нажатии на кнопку `Button1` в `Memo1` будет выведено следующее:

```
   3.14159
   6.28318
   9.42477
  12.56636
```

Поиграв с этим механизмом, вы можете убедиться, что доступ к организованному массиву осуществляется побайтно, так что интерпретировать размещенные в нем байты можно по-любому. Это очень удобно, если мы получаем, например, через COM-порт большие числа, которые при этом неизбежно оказываются разбиты на отдельные байты (аналогичная задача возникала у нас в *главе 19* при чтении заголовка BMP-файла). Альтернативой побайтной посылке через последовательный порт является преобразование различных форматов чисел в символьную форму непосредственно во внешнем устройстве (наподобие того, как это делает GPS-навигатор, разбираемый в *главе 20*). Это несложно: достаточно их преобразовать в десятичный неупакованный формат *(см. приложение 1)*, а прибавлять ли еще к каждому разряду число 48 (код символа "0") для перевода в текстовую форму, или нет — дело уже десятое. Но всегда ли целесообразно этим заниматься, нагружая микроконтроллер, и зная, что в компьютере все равно это будет переводиться обратно в число?

Получаемые байты, конечно, можно накапливать в массиве-приемнике, а потом перемножать в соответствии с разрядностью. Например, четырехбайтное число типа `integer` при этом придется вычислять так (пусть байты поступают по "остроконечной" модели, т. е. сначала младший, и заполняют некий массив, начиная с младшего индекса):

```
var
x:integer;
ab: array [1..4] of byte;
. . . . .
x:=ab[1]+256*ab[2]+ 256*256*ab[3]+ 256*256*256*ab[4];
```

Некрасиво? Не то слово, но главное, что для вычисления подобного выражения программе, очевидно, придется привлекать арифметический сопроцессор. На ассемблере я бы такую операцию сделал более быстрым, чем умножение, способом сдвига разрядов (используя тот факт, что множители кратны степени двойки, а сдвиг на один разряд влево эквивалентен умножению на два), но не думаю, что выполнение операций умножения дифференцируется процессором в зависимости от конкретной величины операндов. Потому куда изящнее и быстрее использовать механизм приведения указателей, как мы и делали это в *главе 19*. Заменим в предыдущем примере объявления переменных на следующие:

```
type
  longp=^integer;
var
  Form1: TForm1;
  pp:pointer; {нетипизированный указатель}
  pint:integer; {указатель-число}
  x,i:integer; {4-байтное число и счетчик}
  xb:byte; {число типа byte}
  px:longp; {указатель на integer}
```

В процедуре по созданию формы будем записывать по указателю последовательные степени числа 64:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  getmem(pp,16);
  pint:=longint(pp);
  x:=64;
  for i:=1 to 4 do
  begin
    px:=longp(pint+(i-1)*4);
    px^:=x;
    x:=x*64;
  end;
end;
```

А в процедуре по нажатию кнопки мы их прочитаем:

```
procedure TForm1.Button1Click(Sender: TObject);
var st:string;
```

```
begin
  Memo1.Lines.Clear;
  for i:=1 to 4 do
  begin {извлекаем четыре значения integer}
     px:=longp(pint+(i-1)*4);
     x:=px^;
     str(x,st);
     Memo1.Lines.Add(st);
  end;
end;
```

В Memo1 по выполнению этой программы мы получим:

```
64
4096
262144
16777216
```

Не забудем также потом освободить память через Freemem!

Динамические массивы, строки и *TMemoryStream*

Читатель, несомненно, давно уже недоумевает: а зачем все это, если есть штука, которая так и называется — динамические массивы (они были введены в Delphi, начиная с версии 4)? Ну, во-первых, при побайтном поступлении больших чисел удобнее и быстрее механизма их преобразования, чем только что изложенный, я все равно не знаю (приближается к нему — по удобству — только механизм TStream, о котором далее). Но главное не в этом. Динамические массивы, в том числе строки типа **string** (которые отличаются от первых только тем, что в них не надо специально следить за текущей длиной, это делается автоматически), используют описанный ранее механизм динамического наращивания длины блоков памяти, и потому при больших объемах данных работают медленно. Утверждается, например, что время выполнения операции типа st:=st+chr(byte) пропорционально квадрату длины строки. Лучше уж использовать обычные массивы, но, на мой вкус, работа с указателями куда гибче и удобнее. Подчеркиваю — морочить себе этим всем голову стоит только при больших объемах данных, поступающих с большой скоростью. Иначе можно спокойно использовать любые механизмы.

> Кстати, а как правильно прочесть нетипизированный файл в строку или динамический массив? Если вы просто напишете код типа:
>
> ```
> var
> st :string;
> ```

```
ff :file;
. . . . .
SetLength(st,1024);
BlockRead(ff,st,length(st));
```

то получите ошибку, которую не всегда можно отследить даже через **try...except**. Это происходит потому, что st есть на самом деле указатель на строку, а не сама строка (в отличие от "родных" паскалевских строк типа ShortSring), и в этом смысле она похожа на PChar. И потому нечто запишется по указанному оператору только в то место, где хранятся 4 байта текущей длины. Правильно будет записать, например, так:

```
BlockRead(ff,PChar(st)^,length(st));
```

Пример использования этой процедуры см. в демонстрационной программе Polsunok *(глава 13)*. Чтение и запись строк, а также всякие их преобразования хорошо разобраны в [39].

Более эрудированный читатель, несомненно, от раздражения давно уже готов выбросить эту книжку в ближайший мусорный ящик, т. к. он знает, что в Delphi есть совершенно официальный механизм для чтения/записи потоковых данных под названием TStream[2] и его наследник TMemoryStream, специально "заточенный" под организацию массивов с произвольным доступом в памяти. Формально рассуждая, я должен был бы с него начать этот раздел. Доступ к данным при использовании TStream осуществляется в точности, как к нетипизированным файлам (которые оказываются, таким образом, просто частным случаем потоковой модели), единственное различие в том, что в процедуре Read (Write) для потока, в отличие от BlockRead (BlockWrite) для файла, не нужен четвертый (и там необязательный) параметр, возвращающий действительно прочитанное (записанное) количество блоков, оно возвращается через идентификатор заданного количества. Действительно, механизм довольно удобный. Вот как можно было бы осуществить процедуру побайтной записи и затем чтения чисел типа integer с использованием нашего экспериментального проекта. Пример заполнения массива несколько искусственный для наглядности: в первые четыре байта мы записываем комбинацию 255, 0, 0, 0, во вторые — 255, 255, 0, 0, и т. д., а потом по нажатию кнопки читаем уже четырехбайтные числа:

[2] Не путать используемый здесь термин "поток" (stream) с "потоком" в смысле выполняющегося программного кода (thread — см. сноску 1 в *главе 1* и *главу 20*). Stream переводится именно как "поток". Термин же thread правильнее было бы переводить как "нить" или просто "тред" — что часто и делают в некоторых источниках.

```
var
. . . . .
x,i:integer;
xb,yb:byte;
Stream:TStream;
. . . . .
procedure TForm1.FormCreate(Sender: TObject);
begin
Stream:=TMemoryStream.Create;
xb:=255;yb:=0;
for i:=1 to 16 do
begin
 case i of
 1,5,6,9,10,11,13,14,15,16:
 Stream.Write(xb,1);
 2,3,4,7,8,12:
 Stream.Write(yb,1);
 end;
end;
end;
procedure TForm1.Button1Click(Sender: TObject);
var st:string;
begin
Stream.Position:=0;
Memo1.Lines.Clear;
for i:=1 to 4 do
begin
 Stream.Read(x,4);
 str(x,st);
 Memo1.Lines.Add(st);
end;
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
 Stream.Free;
end;
```

В результате нажатия на кнопку Button1 мы получим в Memo1:

```
255
65535
16777215
-1
```

Если мы изменим тип x на dword, то получим вместо –1 в последней строке 4 294 967 295. Здорово? Здорово, причем TStream имеет все полагающиеся свойства и методы для произвольного доступа к записанному потоку. Некоторые вы можете видеть в примере, кроме них есть еще и Size, Seek (смещение на нужное число байтов от начала или от текущей позиции) и очень удобный метод CopyFrom, который, например, позволяет копировать из потока TMemoryStream в поток TFileStream и обратно, обмениваясь таким образом данными из памяти с дисковым файлом.

Так почему же я не стал разбирать эту возможность с самого начала? Во-первых, потому что выделение памяти в TMemoryStream происходит с помощью функции GlobalReAlloc, совершенно аналогичной по механизму действия ReallocMem, да другого механизма придумать и нельзя, используем ли мы менеджер памяти Windows или Delphi, или даже свой собственный. Можно только оптимизировать выделение памяти под конкретную задачу, но сводиться это будет все равно к выбору, сколько лучше всего байтов зарезервировать *заранее*. Некоторые тонкости в работе различных менеджеров памяти мы тут, естественно, опускаем, потому что они, по большому счету, несущественны: самым быстрым способом останется захватить как можно больше, а там уж разбираться. И использовать ли тут указатели, динамические массивы или TMemoryStream — вопрос только удобства. Мы могли бы применять потоковый механизм (и даже просто блочное чтение из нетипизированного файла а-ля Pascal) вместо "маппинга" при чтении файла в программе Trace *(см. главу 14)*, но в том конкретном случае это не ускорило бы работу — там нет потока данных, как такового, есть готовые файлы заранее известного размера. Отличие, кстати, реализации потокового механизма в C++ от Delphi (за что последнюю часто ругают) и заключается в том, что в первом случае управление потоками гораздо более гибкое и позволяет буферизовать обмен данными, например, при операциях с дисковыми файлами.

Во-вторых, и, пожалуй, в главных — пресловутая негибкость управления потоками проявляется не только в отсутствии механизмов регуляции выделения памяти. Сейчас мы сравним два метода сброса больших массивов памяти на диск и посмотрим, что быстрее, удобнее и "безглючнее".

Произвольный доступ к большим массивам данных

Если мы будем получать данные с COM-порта с экстремальными значениями выставленного нами самими скоростного порога — 32 Кбайта в секунду, то объем данных достигнет величины в несколько десятков мегабайт уже минут за двадцать. А что делать, если нужно что-то отслеживать сутками? Даже

GPS-навигатор, который мы изучали в *главе 20*, при своих никчемных 4800 бод выдает каждую секунду примерно 500 байт информации, которую приходится принимать, а уж потом в ней разбираться, т. е. за сутки он "накидает" нам примерно 40 Мбайт. Такие массивы в памяти — предел для нормальной работы среднего компьютера под управлением Windows, который параллельно еще и выполняет другие задачи, и увеличение объема ОЗУ тут проблемы, естественно, принципиально не решит. Поэтому периодически надо данные сбрасывать на диск, не полагаясь на Windows. Как это лучше сделать?

Для измерения времени работы программы добавим к нашему пробному приложению модуль DateUtils. Модельная задача будет такая: накапливать в памяти данные, добавляя по одному байту, а при превышении некоего порога сбрасывать их на диск. Отметим, что реальные задачи сложнее — сбрасывать обычно надо не весь поток, по ходу дела нужно получать доступ к файлу, например, из другого "треда" и т. п. Мы же сейчас хотим сравнить механизмы накопления в "голом" виде. Сначала попробуем то, что нам предлагает механизм TSrteam.

Дополним список переменных следующим:

```
. . . . .
Stream,FStream:TStream;
tt,told:TDateTime;
. . . . .
```

В процедуре инициализации запишем:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
Stream:=TMemoryStream.Create;
FStream:=TFileStream.Create('temp.tmp',fmCreate); {перезаписываем}
FStream.Free; {начнем потом с нуля}
end;
```

По нажатию кнопки будем делать следующее:

```
procedure TForm1.Button1Click(Sender: TObject);
var st:string;
begin
Stream.Position:=0;
Memo1.Lines.Clear;
i:=0;
told:=Time;
while i<209715200 do {до 200 Мбайт данных}
begin
 Stream.Write(byte(i),1);
```

```
 if Stream.Size>5*1048576 {если превысило 5 Мбайт}
 then
 begin
  FStream:=TFileStream.Create
     ('temp.tmp',fmOpenReadWrite or fmShareDenyNone);
  FStream.Position:=FStream.Size;
  Stream.Position:=0; {с нуля}
  FStream.CopyFrom(Stream,Stream.Size); {копируем в файл}
  Stream.Free; {опять с нуля}
  FStream.Free; {опять с нуля}
  Stream:=TMemoryStream.Create;
 end;
 i:=i+1;
 Application.ProcessMessages;
end;
  FStream:=TFileStream.Create
     ('temp.tmp',fmOpenReadWrite or fmShareDenyNone);
  FStream.Position:=FStream.Size;
  Stream.Position:=0; {с нуля}
  FStream.CopyFrom(Stream,Stream.Size);
           {копируем в файл концовку массива}
  Stream.Free; {все удаляем}
  FStream.Free; {все удаляем}
  tt:=Time; {измеряем время}
  st:=IntToStr(SecondsBetween(tt,told));
  st:=IntToStr(i)+' '+st+' s';
  Memo1.Lines.Add(st);
end;
```

В соответствии с этим кодом программа будет в непрерывном цикле накапливать значения байтов от 0 до 255 сначала в массиве `Stream` в памяти и при превышении порога в 5 Мбайт сбрасывать его на диск через файловый поток `FStream` в файл temp.tmp. Работа закончится, когда общий объем файла составит 200 Мбайт. Порог массива в памяти в 5 Мбайт я выбрал вынужденно — при попытке сделать больше программа исправно выдавала "Out of memory". С чем это связано и почему именно 5 Мбайт — я выяснить не смог, возможно, это просто особенности функционирования класса `TStream`. Отметим данный факт в крупный минус методу и запустим программу. Вот что выводится в `Memo` по окончании процедуры:

```
209715200 846 s
```

То есть программа работала более 14 минут (разумеется, на другом компьютере и в других условиях результаты будут иными). Можно попробовать

ускорить процесс, если не уничтожать и не создавать заново файловый поток каждый раз, но тогда вне зависимости от всяких fmShareDenyNone просматривать его по ходу дела не получается. Запишем это также в минус.

Теперь переделаем то же самое с использованием указателей и обычной записи в нетипизированный файл. Так как сейчас позиция в самостоятельно организованном массиве автоматически передвигаться не будет, то придется быть предельно внимательными, чтобы не "вылететь" за пределы зарезервированного участка памяти. Однако, как мы увидим, эти усилия полностью оправдаются. Итак, объявим следующие переменные:

```
type
bytep=^byte;
var
Form1: TForm1;
pp:pointer; {нетипизированный указатель}
pint:integer; {указатель-число}
i,j,x:integer; {счетчики}
xb:byte;
pb:bytep; {указатель на байт}
ff:file;
tt,told:TDateTime;
```

Процедура инициализации:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
getmem(pp,5*1048576); {резервируем 5 Мбайт}
pint:=longint(pp);
assignfile(ff,'temp.tmp');
rewrite(ff,1); {переписываем с нуля}
x:=5*1048576; {это будет объем записи}
end;
```

Собственно тестовая процедура:

```
procedure TForm1.Button1Click(Sender: TObject);
var st:string;
begin
Memo1.Lines.Clear;
i:=0;j:=0;
told:=Time;
while i<209715200 do {до 200 Мбайт данных}
begin
 if i>=(j+1)*5*1048576 then {если равно 5 Мбайт}
```

```
 begin
   j:=j+1;
   reset(ff,1); {открываем заново}
   Seek(ff,FileSize(ff)); {позицию - в конец файла}
   blockwrite(ff,pp^,x);
   closefile(ff);
 end;
 xb:=byte(i);
 pb:=bytep(pint+i-(j*5*1048576)); {после каждых 5 Мбайт с нуля}
 pb^:=xb;
 i:=i+1;
 Application.ProcessMessages;
end;
   reset(ff,1);
   Seek(ff,FileSize(ff));
   blockwrite(ff,pp^,x);
   closefile(ff);
  tt:=Time; {измеряем время}
  st:=IntToStr(SecondsBetween(tt,told));
  st:=IntToStr(i)+' '+st+' s';
  Memo1.Lines.Add(st);
end;
```

Не забудем также освободить зарезервированную память в процедуре onDestroy. Результат будет несколько ошеломляющим:

```
209715200 237 s
```

То есть скорость работы возросла почти в четыре раза! Если вы поэкспериментируете с этим алгоритмом, то выясните, что скорость практически не зависит от того, открываем ли мы файл каждый раз заново или нет. Это при том, что у нас не возникает никаких проблем с размером резервируемой области. В пределах объема массива в памяти от 1 до 10 Мбайт скорость работы не меняется. Зато в обоих случаях скорость (в Windows 98) падала, когда программу сворачивали в панель задач. Из всего изложенного можно сделать вывод, что процедуры на основе самостоятельно организованных массивов с записью непосредственно в память предпочтительнее "официального" механизма — возможно, что и потоковый механизм можно "вылизать" так, чтобы он работал, "как часы", но зачем?

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ 1

О системах счисления

О том, что мы считаем в десятичной системе, т. к. у нас десять пальцев на двух руках, осведомлены, вероятно, все. Персонажи из мультфильмов студии "Пилот ТВ" — Хрюн Моржов и Степан Капуста — считают, наверное, в восьмеричной системе, так как у них по четыре пальца. У древних ацтеков и майя в ходу была двадцатеричная система (вероятно потому, что закрытая обувь в том климате была не в моде). Вместе с тем, история показывает, что привязка к анатомическим особенностям строения человеческого тела совершенно необязательна. Со времен древних вавилонян у нас в быту сохранились остатки двенадцатеричной и шестидесятеричной систем, что выражается в количестве часов в сутках и минут в часах, или, скажем, в том, что столовые приборы традиционно считают дюжинами или полудюжинами (а не десятками и пятерками). Так что само по себе основание системы счисления не имеет значения — точнее, есть дело привычки и удобства. Однако такое положение справедливо лишь для ручного счета — для компьютеров выбор системы счисления имеет более важное значение. Попробуем ответить на вопрос — почему? Для этого нам придется сначала разобраться — как мы, собственно говоря, считаем, что при этом происходит, что такое вообще система счисления и ее основание.

Из понятия числа, как объективно существующей абстракции, вытекает, что его материальное представление может быть произвольным, лишь бы оно подчинялась тем же правилам, что и сами числа. Проще всего считать палочками (и в детском саду нас учат именно такому счету), в качестве которых могут выступать и пластмассовые стерженьки, и пальцы, и черточки на бумаге. Один — одна палочка, два — две палочки, десять — десять палочек. А сто палочек? Уже посчитать затруднительно, потому придумали сокращение записи: доходим до пяти палочек, ставим галочку, доходим до десяти — ставим крестик:

1	2	5	7	10	11
I	II	V	VII	X	XI

Узнаете? Конечно, это всем знакомая римская система, сохранившаяся до настоящих времен на циферблатах часов или в нумерации столетий. Она представляет собой пример *непозиционной* системы счисления — потому что значение определенного символа, *обозначающего* то или иное число, в ней не зависит от позиции относительно других символов — все значения в записи просто *суммируются*. То есть записи "XVIII" и "IIIXV" в принципе должны означать одно и то же. На самом деле это не совсем так: в современной традиции принято в целях сокращения записи использовать и позицию: скажем, в записи "IV" факт, что палочка стоит перед галочкой, а не после нее, означает придание ей отрицательного значения, т. е. в данном случае единица не прибавляется, а вычитается из пяти (то же самое относится и к записи девятки "IX"). Если вы человек наблюдательный, то могли заметить, что на часах четверку пишут почти всегда, как "IIII", а не как "IV", что, несомненно, более отвечает духу непозиционной системы. Однако при всех возможных отклонениях главным здесь остается факт, что в основе системы лежит операция *суммирования*.

Позиционные системы

Большие числа в римской системе записывать трудно. Поэтому были придуманы позиционные системы, к которым, в частности, принадлежала и упомянутая ранее вавилонская шестидесятеричная (см. рис. П1.1). Позднее в Европе позиционную систему переоткрыл (видимо) Архимед, затем от греков она была воспринята индусами и арабами, и на рубеже I и II тысячелетий опять попала в Европу[1] — с тех пор мы называем цифры арабскими, хотя по справедливости их следовало бы назвать индийскими. Это была уже современная десятичная система в том виде, в котором мы ее используем по сей день, у арабов отличается только написание цифр. С тем фактом, что заимствована она именно у арабов, связано не всеми осознаваемое несоответствие порядка записи цифр в числе с привычным нам порядком следования текста: арабы, как известно, пишут справа налево. Поэтому значение цифры в зависимости от позиции ее в записи числа возрастает именно справа налево, что в нашем случае нелогично — приходится заранее обозревать число целиком и готовить ему место в тексте. Впрочем, в быту к этому все привыкли и неудобств

[1] Перевод соответствующего трактата арабского ученого аль Хорезми на латынь относится к 1120 году (на самом деле его звали Мухаммед аль Хорезми, то есть "Мухаммед из Хорезма"; между прочим, от его прозвища произошло слово "алгоритм").

не испытывают, а вот для программистов это приводит к некоторым сложностям: по мере чтения из памяти компьютера числа располагаются по старшинству адресов, т. е. в естественном для европейца порядке записи текста, слева направо. Между тем, сами числа (при побайтном представлении) являются двухзнаковыми, и при их записи соблюдается обычный арабский порядок — справа налево. Поэтому получаются довольно неудобные для восприятия конструкции: скажем, десятичное число 1234 в такой записи имело бы вид 3412[2].

Строгое определение позиционной системы является следующим: сначала выбирается некоторое число *p*, которое носит название *основания системы счисления*. Тогда любое число в такой системе может быть представлено следующим образом:

$$a_n \cdot p^n + a_{n-1} \cdot p^{n-1} + ... + a_1 p^1 + a_0 \cdot p^0. \quad (1)$$

В самой записи числа степени основания подразумеваются, а не пишутся (и для записи основания даже нет специального значка), т. е. запись будет представлять собой просто последовательность $a_n ... a_0$ (еще раз обратим внимание на то, что запись производится справа налево по старшинству — обычная математическая запись выглядела бы наоборот). Отдельные позиции в записи числа называются *разрядами*. Например, в десятичной системе (т. е. в системе с основанием 10) полное представление четырехразрядного числа 1024 таково:

$$1 \cdot 10^3 + 0 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0.$$

Так как любое число в нулевой степени равно 1, то степень в младшем разряде можно и не писать, но ради строгости мы ее будем воспроизводить, т. к. это позволяет нам лучше вникнуть в одно обстоятельство: степень старшего разряда всегда на единицу меньше, чем количество разрядов, т. к. нумерация степеней ведется с нуля.

Ну, а как можно представить число в системе счисления с другим основанием? Для любой системы с основанием *p* нужно не меньше (и не больше) чем *p* различных *цифр* — т. е. *значков* для изображения чисел. Для десятичной системы их десять — это и есть известные всем символы от 0 до 9. Выбор начертания этих значков совершенно произволен — так, у арабов и по сей день 1 обозначается, как и у нас, палочкой. А вот цифра 2 обозначается знаком, похожим на латинскую строчную "r", причем тройка тоже имеет похожее начертание, и я плохо себе представляю, как их там различают — дело, впрочем, привычки, у нас тоже знаки "5" и "6" в некоторых случаях разли-

[2] Этот формат записи — когда младший разряд записывается первым — иногда называют еще "остроконечным" (little-endian), в отличие от обычного "тупоконечного" (big endian), когда первым записывается старший байт.

чить непросто, не говоря уж о сходстве между нулем "0" и буквой "О". В ручном написании текстов программ, а также в матричных компьютерных шрифтах, которые были в ходу до появления графического интерфейса, для этого ноль даже изображали перечеркнутым, наподобие знака диаметра: "∅"[3]. Чтобы древним вавилонянам, несчастным, не приходилось учить аж 60 разных начертаний знаков, они придумали логичную систему наподобие римской (обратите внимание на рис. П1.1) — действующую, впрочем, только в пределах первых шестидесяти чисел, а далее у них система становилась вполне позиционной.

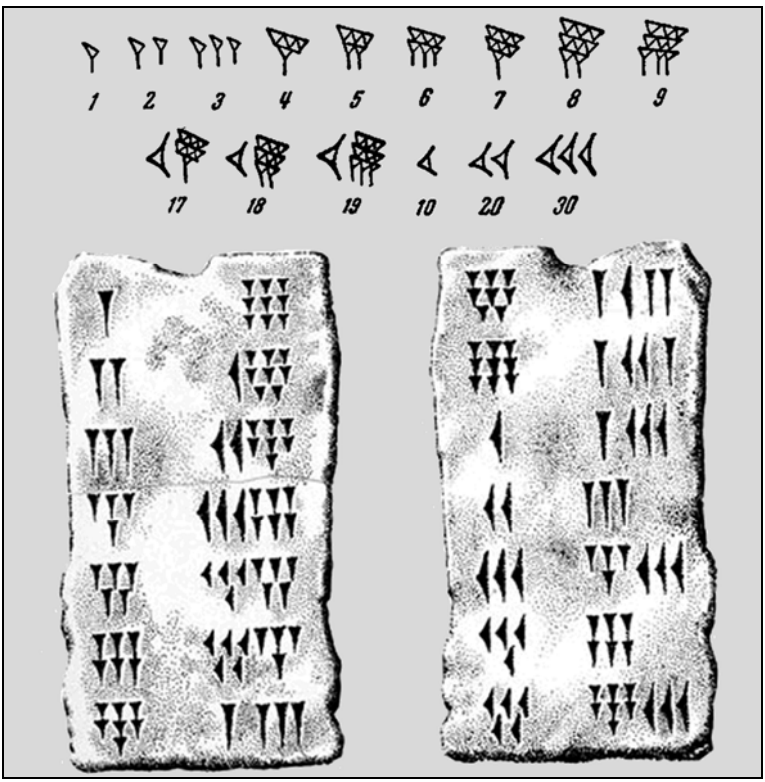


Рис. П1.1. Вавилонские глиняные таблички с записью чисел. Вверху перевод некоторых из них в десятичную систему

[3] Попробуйте различить записи "15Ом" и "150м", если пробел забыли поставить в нужном месте — в случае матричных шрифтов или ручной записи, да и в любом случае, если символы "0" и "О" не стоят рядом, это неразрешимая задача, если только из контекста не ясно, когда идет речь об омах, а когда — о метрах. К одной из программ, которой я пользуюсь, производитель дал такой серийный номер: "RO-1T9O265…". Так как в одном из случаев там ноль, а во втором буква "О", то каждый раз, когда я эту программу переустанавливаю, приходится гадать на кофейной гуще.

Двоичная система

В двоичной системе необходимо всего два различных знака для цифр: 0 и 1. Это и вызвало столь большое ее распространение в электронике: смоделировать два состояния электронной схемы и затем их безошибочно различить неизмеримо проще, чем три, четыре и более, не говоря уж о десяти. Правда, существовали и десятичные компьютеры, а вот с троичным компьютером, который был на практике под названием "Сетунь" построен Н. Брусенцовым в МГУ на рубеже 60-х годов прошлого века, связана отдельная история. При разработке первых компьютеров перед конструкторами встал вопрос об экономичности систем счисления с различными основаниями. Под экономичностью системы понимается тот запас чисел, который можно записать с помощью данного количества знаков. Чтобы записать 1000 чисел (от 0 до 999) в десятичной системе, нужно 30 знаков (по десять в каждом разряде), а в двоичной системе с помощью 30 знаков можно записать $2^{15} = 32\,768$ чисел, что гораздо больше 1000. То есть двоичная система явно экономичнее десятичной. В общем случае, если взять n знаков в системе с основанием p, то количество чисел, которые при этом можно записать, будет равно $p^{n/p}$. Легко найти максимум такой функции, который будет равен иррациональному числу $e = 2.718282...$ Но поскольку система с основанием e может существовать только в воображении математиков, то самой экономичной считается система счисления с основанием 3, ближайшим к числу e. В компьютере, работающем по такой системе, число элементов, необходимых для представления числа определенной разрядности, минимально. Реализацию троичной системы в электронике можно представить себе, как схему с такими, например, состояниями: напряжение отсутствует (0), напряжение положительно (1), напряжение отрицательно (–1). И все же брусенцовская "Сетунь" осталась историческим курьезом — слишком велики сложности схемной реализации[4]. Что еще важнее, двоичная система прекрасно стыкуется как с булевскими логическими переменными, так и с тем фактом, что величина, принимающая два и только два состояния и получившая название *бит*[5], есть естественная единица количества информации. Это было установлено в 1948 году одновременно упоминавшимся уже Клодом Шенноном и Нобертом Винером, "отцом" кибернетики — меньше, чем один бит, информации не бывает. Разряды двоич-

[4] Точнее, сам Николай Петрович Брусенцов как раз сложностей не испытывал, так как использовал для представления троичных цифр — тритов — трансформаторы, в которых наличие тока в обмотке в одном направлении принималось за 1, в другом — за –1, а отсутствие тока обозначало 0. Но реализовать на транзисторах такое представление значительно сложнее, чем двоичное.

[5] Bit — по-английски "кусочек, частица чего-либо". Существует и слово bite, которое также можно перевести как "кусок", но с несколько другим оттенком — "кусок пищи", "укус".

ных чисел (т. е. чисел, представленных в двоичной системе) также стали называть битами.

Итак, запись числа в двоичной системе требует всего две цифры, начертание которых заимствовано из десятичной системы и выглядит, как 0 и 1. Число, например, 1101 тогда будет выглядеть как:

$$1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13.$$

Чтобы отличить запись числа в различных системах, часто внизу пишут основание системы:

$1101_2 = 13_{10}$

Шестнадцатеричная система

Шестнадцатеричная система имеет, как ясно из ее названия, основание шестнадцать. Для того чтобы получить шестнадцать различных знаков, изобретать ничего нового не стали, а просто использовали те же цифры от 0 до 9 для первых десяти, и заглавные латинские буквы от A до F для одиннадцатого-шестнадцатого знаков. Таким образом, известное нам число 13_{10} выразится в шестнадцатеричной системе, как просто D_{16}. Соответствие шестнадцатеричных знаков десятичным числам следует выучить наизусть: A — 10, B — 11, C — 12, D — 13, E — 14, F — 15. Значения больших чисел вычисляются по обычной формуле, например:

$$A2FC_{16} = 10 \cdot 16^3 + 2 \cdot 16^2 + 15 \cdot 16^1 + 12 \cdot 16^0 = 40960 + 512 + 240 + 12 = 41724_{10}.$$

Как следует из сказанного ранее, перевод в десятичную систему любых форматов не представляет сложности и при надлежащей тренировке может осуществляться даже в уме. Для того чтобы быстро переводить в десятичную систему двоичные числа (и, как мы увидим, и шестнадцатеричные тоже), следует выучить наизусть таблицу степеней двойки до 16:

2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8
2	4	8	16	32	64	128	256
2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
512	1024	2048	4096	8192	16384	32768	65536

На первое время достаточно запомнить верхний ряд, остальное выучится позже само.

Сложнее переводить из десятичной системы, и для этого в учебниках рекомендуется устрашающая процедура, основанная на делении столбиком.

Я сейчас попробую вам показать способ, который позволяет переводить числа в двоичную систему несколько более простым методом, причем небольшие числа можно переводить даже в уме. Естественно, что в сущности это то же самое деление, но без лишних сложностей и формальностей. Для этого нужно сначала запомнить следующее правило: число, равное какой-либо степени двойки, имеет 1 в разряде с номером, на единицу большим степени, остальные все нули:

$2^1 = 2_{10} = 10_2$

$2^2 = 4_{10} = 100_2$

$2^3 = 8_{10} = 1000_2$ и т. д.

Способ состоит в следующем: пусть мы имеем, например, десятичное число 59. Подбираем наибольшую степень двойки из таблицы ранее, не превышающую этого числа: 32, что есть 5-я степень. Ставим 1 в шестом разряде: 100000. Вычитаем подобранную степень из исходного числа (59 – 32 = 27) и подбираем для остатка также степень, его не превышающую: 16 (2^4). Ставим единицу в 5-м разряде: 110000. Повторяем процедуру вычитания-подбора: 27 – 16 = 11, степень равна 8 (2^3), ставим единицу в 4-м разряде: 111000. Еще раз: 11 – 8 = 3, степень равна 2 (2^1), ставим единицу во 2-м разряде: 111010. Последнее вычитание дает 1, которую и ставим в младший разряд, окончательно получив $59_{10} = 111011_2$. Если бы исходное число было четным, к примеру, 58, то в последнем вычитании мы бы получили 0, и число в двоичной системе также оканчивалось бы на ноль: $58_{10} = 111010_2$.

Кстати, полезно также обратить внимание, что числа, на единицу меньшие степени двойки, имеют количество разрядов, равное степени, и все эти разряды содержат единицы:

$2^1 - 1 = 1_{10} = 1_2$

$2^2 - 1 = 3_{10} = 11_2$

$2^3 - 1 = 7_{10} = 111_2$ и т. д.

Подобно тому, как наибольшее трехразрядное число в десятичной системе равно 999, и чисел таких всего $10^3 = 1000$ (от 000 до 999), в двоичной системе тех же трехразрядных чисел будет $2^3 = 8$ штук, в диапазоне от 000 до 111, т. е. от 0 до 7. Таким образом, наибольшее двоичное число с данным количеством разрядов будет всегда содержать все единицы во всех разрядах.

А вот из двоичной системы в шестнадцатеричную и обратно перевод очень прост: все дело в том, что 16 есть 2^4 и без всяких вычислений можно утверждать, что одноразрядное шестнадцатеричное число будет иметь ровно 4 двоичных разряда. Поэтому перевод из двоичной системы в шестнадцате-

ричную осуществляется так: двоичное число разбивается на так называемые тетрады, т. е. группы по четыре разряда, а затем каждая тетрада переводится отдельно и результаты выписываются в том же порядке. Так как в тетраде всего 16 вариантов, то их опять же легко выучить наизусть:

0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111
8	9	A (10)	B (11)	C (12)	D (13)	E (14)	F (15)
1000	1001	1010	1011	1100	1101	1110	1111

Например, число 59_{10}, т. е. $0011\ 1011_2$, будет равно 3Bh.

Точно так же осуществляется и обратный перевод — каждое шестнадцатеричное число просто записывается в виде тетрады. Так, число $A2FC_{16}$ выразится, как $1010\ 0010\ 1111\ 1100_2$. Заметьте, что пробелы между тетрадами введены просто для удобства восприятия, подобно пробелам между тройками разрядов (классами) в записи больших десятичных чисел, и никакой иной нагрузки не несут. При записи двоичных чисел в тексте программ, как ассемблерных, так и программ на языках высокого уровня, естественно, эти пробелы ставить запрещается.

Шестнадцатеричный формат записи часто еще обозначают как HEX (HEXadecimal), двоичный — как BIN (BINary), а десятичный — как DEC (DECimal). Так как с помощью матричных шрифтов на компьютерах с текстовыми дисплеями воспроизводить индексы было невозможно, то вместо того, чтобы обозначать основания системы цифрой справа внизу, их стали обозначать буквами: "B" (или "b") — двоичная система, "H" (или "h") — шестнадцатеричная, "D" (или "d") — десятичная. Отсутствие буквы также означает десятичную систему:

13 = 13d = 00001101b = 0Dh.

Такая запись принята, например, в языке ассемблера для процессоров Intel. Популярность языка C внесла в это дело некоторый разнобой: там десятичная система обозначается буквой "d" (или никак), двоичная буквой "b", а вот шестнадцатеричная буквой "x", причем запись во всех случаях предваряется нулем (чтобы не путать запись числа с идентификаторами переменных, которые всегда начинаются с буквы):

13 = 0d13 = 0b00001101 = 0x0D.

В языке Pascal принято, как известно, следующее представление HEX-формата: $0D. Обратите внимание, что запись в HEX-формате обычно ведет-

ся в двухразрядном виде, даже если число имеет всего один значащий разряд, как в нашем случае, то в старшем пишется 0, то же самое относится и к двоичной записи, которая дополняется нулями до восьми разрядов. Кроме перечисленных, в ходу еще так называемый двоично-десятичный формат BCD (Binary-Coded Decimal).

Представление чисел в формате BCD

Электронные устройства "заточены" под использование двоичных и родственных им систем счисления, потому что основой являются два состояния, т. е. двоичная цифра. Так что, соединив несколько устройств вместе с целью оперирования многоразрядными числами, мы всегда будем получать именно двоичное число. При этом четыре двоичных разряда могут представлять шестнадцать различных состояний, и использовать их для представления десятичных чисел было бы попросту не экономично: часть возможного диапазона осталась бы неиспользованной. Подсчитайте сами: для представления числа с шестью десятичными разрядами в десятичном виде нужно $6 \cdot 4 = 24$ двоичных разряда, а для представления того же числа в двоичном виде с избытком хватит 20 разрядов ($2^{20} = 1\,048\,576$). А меньше, чем четыре двоичных разряда, для представления одного десятичного числа не хватит ($2^3 = 8$). К тому же с чисто двоичными числами, как мы увидим в дальнейшем, оперировать значительно проще. И все же применять двоично-десятичный формат приходится всегда, когда речь идет о выводе чисел, например, на цифровой дисплей. Поэтому приходится преобразовывать шестнадцатеричные числа в десятичные и хранить их в таких же байтовых регистрах или ячейках памяти. Это можно делать двумя путями: в виде упакованного и неупакованного BCD. Неупакованный формат попросту означает, что мы тратим на каждую десятичную цифру не тетраду, как необходимо, а целый байт. Зато при этом не возникает разночтений: $05h = 05_{10}$ и никаких проблем. Однако ясно, что это крайне не экономично — байтов требуется в два раза больше, а старший полубайт при этом все равно всегда ноль. Потому BCD-числа всегда упаковывают, занимая и старший разряд второй десятичной цифрой: скажем, число 59 при этом запишется как просто 59. Однако это не 59h! 59 в шестнадцатеричной форме есть 3Bh, как мы установили ранее, а наше 59 процессор прочтет, как $5 \cdot 16 + 9 = 89$, что вообще ни в какие ворота не лезет! Поэтому перед проведением операций с упакованными BCD-числами их обязательно распаковывают, перемещая старший разряд в отдельный байт, и заменяя в обоих байтах старшие полубайты нулями. Иногда для проведения операций с BCD в микропроцессоре или микроконтроллере предусмотрены специальные команды, так что отдельно заниматься упаковкой-распаковкой не требуется. В качестве примера хранения чисел в BCD-формате можно привести значения часов, минут и секунд в энергонезависимых часах компьютера.

Модуль Ariphm

В обычном Pascal и первых вариантах Object Pascal шестнадцатеричное представление чисел можно было осуществлять только в одностороннем порядке: вы могли в тексте программы использовать записи с предваряющим знаком "$", однако вывести числа в той же форме не получалось. Для преодоления этого недостатка автором был создан модуль под названием Ariphm, который переводит числа (типа `byte`, `word` и `longint` — последний совместим с типом `integer`) в строковое представление шестнадцатеричных чисел[6]. В позднейших версиях появилась функция `IntToHex (Value:integer;Digits:înteger)`, но автором к тому времени было наработано столько текстов программ, что он по привычке по-прежнему пользуется этим модулем. У функций из модуля есть также мелкое преимущество — они специализированы под определенный тип аргумента и потому требуют всего один параметр (число знаков `Digits` указывать не требуется). Разумеется, предлагаемые функции в текстах можно без проблем заменить на вызов `IntToHex` с соответствующим значением параметра `Digits` (для байта, например, оно должно быть равно 2). Вы можете без изменений использовать текст этого модуля также в DOS-варианте Borland Pascal. Модуль находится на прилагаемом диске в папке Priloz/Ariphm и в составе многих проектов на том же диске, а здесь я воспроизвожу на всякий случай его текст:

```
UNIT Ariphm;

interface

function hexb( dh:byte):string; {перевод байта в HEX}
function hexw( dh:word):string; {перевод двухбайтного слова в HEX}
Function hexlong(dh:longint):string; {перевод четырехбайтного слова
                                в HEX}

implementation

function hexb;
var d:byte;
var ch,cl:char;
begin
d:=dh;
d:=d shr 4;
```

[6] Название модуля не совсем отвечает его содержанию, но оно сложилось исторически: первоначально в нем еще содержались некоторые математические функции, которые были затем удалены ввиду появления в Delphi модуля Math.

```
if d<10 then ch:=chr(d+48)
else if d<16 then ch:=chr(d+55);
d:=dh;
d:=d shl 4;d:=d shr 4;
if d<10 then cl:=chr(d+48)
else if d<16 then cl:=chr(d+55);
hexb:=ch+cl
end;

function hexw;
var d:word;
var ch1,cl1,ch0,cl0:char;
begin
d:=dh;
d:=d shr 12;
if d<10 then ch1:=chr(d+48)
else if d<16 then ch1:=chr(d+55);
d:=dh;
d:=d shl 4;d:=d shr 12;
if d<10 then cl1:=chr(d+48)
else if d<16 then cl1:=chr(d+55);
d:=dh;
d:=d shl 8;d:=d shr 12;
if d<10 then ch0:=chr(d+48)
else if d<16 then ch0:=chr(d+55);
d:=dh;
d:=d shl 12;d:=d shr 12;
if d<10 then cl0:=chr(d+48)
else if d<16 then cl0:=chr(d+55);
hexw:=ch1+cl1+ch0+cl0;
end;

Function hexlong(dh:longint):string;
var d:word;
var st:string;
begin
d:=word(dh);
st:=hexw(d);
d:=dh div 65536;
hexlong:=hexw(d)+st;
end;

begin
end.
```

ПРИЛОЖЕНИЕ 2

Виртуальные и скан-коды для 101/104-кнопочной клавиатуры

В табл. П2.1 суммированы все сведения о клавишах, которые были получены из официальной документации [14], а также из книг [2,15], файла Windows.pas (..\Source\Rtl\Win\) и справки по Win32. Из логики преимущественного, по мнению разработчиков, использования при работе с цифрами цифровой клавиатуры вытекает и присвоение названий констант виртуальных кодов — так, vk_Add относится именно к клавише <+> цифровой клавиатуры, а не к клавише <+=> в верхнем ряду основной клавиатуры, которая называется vk_OEM_Plus. Причем последняя константа в файле Windows.pas не определена, так что с этой клавишей, как и с ей подобными, в Delphi придется работать по числовому значению. Все такие случаи, когда константа в Delphi отсутствует, помечены ссылкой на сноску 2. С цифро-буквенными клавишами, виртуальный код которых совпадает с номером соответствующего символа, также нужно работать либо через непосредственное значение, равное символьному коду ASCII *(см. приложение 3)*, либо через вызов функции ord. Подробнее об этом, а также о кодах символов дублирующих клавиш см. *главу 6*.

Таблица П2.1. *Скан-коды и виртуальные коды для 101/104-кнопочной клавиатуры*

Скан-код		Виртуальный код			Символ[1] или клавиша	Символ +<Shift>
HEX	DEC	Номер		Константа		
		HEX	DEC			
01	1	1B	27	vk_ESCAPE	<Esc>	
02	2	31	49	vk_1[2]	1	!
03	3	32	50	vk_2[2]	2	@
04	4	33	51	vk_3[2]	3	#

Таблица П2.1 *(продолжение)*

Скан-код		Виртуальный код			Символ[1] или клавиша	Символ +<Shift>
HEX	DEC	Номер		Константа		
		HEX	DEC			
05	5	34	52	vk_4[2]	4	$
06	6	35	53	vk_5[2]	5	%
07	7	36	54	vk_6[2]	6	^
08	8	37	55	vk_7[2]	7	&
09	9	38	56	vk_8[2]	8	*
0A	10	39	57	vk_9[2]	9	(
0B	11	30	48	vk_0[2]	0	)
0C	12	BD	189	vk_OEM_MINUS[2]	-	_
0D	13	BB	187	vk_OEM_PLUS[2]	=	+
0E	14	08	8	vk_BACK	<Backspace>	
0F	15	09	9	vk_TAB	<Tab>	
10	16	51	81	vk_Q[2]	q	Q
11	17	57	87	vk_W[2]	w	W
12	18	45	69	vk_E[2]	e	E
13	19	52	82	vk_R[2]	r	R
14	20	54	84	vk_T[2]	t	T
15	21	59	89	vk_Y[2]	y	Y
16	22	55	85	vk_U[2]	u	U
17	23	49	73	vk_I[2]	i	I
18	24	4F	79	vk_O[2]	o	O
19	25	50	80	vk_P[2]	p	P
1A	26	DB	219	vk_OEM_4[2]	[	{
1B	27	DD	221	vk_OEM_6[2]	]	}
1C	28	0D	13	vk_RETURN[10]	<Enter>	
1D	29	11	17	vk_CONTROL[3]	<Ctrl>	
		A2	162	vk_LCONTROL	<LCtrl>	
		A3	163	vk_RCONTROL	<RCtrl>	

Таблица П2.1 *(продолжение)*

Скан-код		Виртуальный код			Символ[1] или клавиша	Символ +<Shift>
HEX	DEC	Номер		Константа		
		HEX	DEC			
1E	30	41	65	vk_A[2]	a	A
1F	31	53	83	vk_S[2]	s	S
20	32	44	68	vk_D[2]	d	D
21	33	46	70	vk_F[2]	f	F
22	34	47	71	vk_G[2]	g	G
23	35	48	72	vk_H[2]	h	H
24	36	4A	74	vk_J[2]	j	J
25	37	4B	75	vk_K[2]	k	K
26	38	4C	76	vk_L[2]	l	L
27	39	BA	186	vk_OEM_1[2]	;	:
28	40	DE	222	vk_OEM_7[2]	'	"
29	41	C0	192	vk_OEM_3[2]	`	~
2A	42	10	16	vk_SHIFT[5]	<Shift>	
		A0	160	vk_LSHIFT	<LShift>	
2B	43	DC	220	vk_OEM_5[2]	\	|
2C	44	5A	90	vk_Z[2]	z	Z
2D	45	58	88	vk_X[2]	x	X
2E	46	43	67	vk_C[2]	c	C
2F	47	56	86	vk_V[2]	v	V
30	48	42	66	vk_B[2]	b	B
31	49	4E	78	vk_N[2]	n	N
32	50	4D	77	vk_M[2]	m	M
33	51	BC	188	vk_OEM_COMMA[2]	,	<
34	52	BE	190	vk_OEM_PERIOD[2]	.	>
35	53	BF	191	vk_OEM_2[2]	/	?
36	54	A1	161	vk_RSHIFT[4]	<RShift>	
37	55	2C	44	vk_SNAPSHOT	<PrScr>	

Таблица П2.1 *(продолжение)*

Скан-код		Виртуальный код			Символ[1] или клавиша	Символ +<Shift>
HEX	DEC	Номер		Константа		
		HEX	DEC			
38	56	12	18	vk_MENU[6]	<Alt>	
		A4	164	vk_LMENU	<LAlt>	
		A5	165	vk_RMENU	<RAlt>	
39	57	20	32	vk_SPACE	<Space>	
3A	58	14	20	vk_CAPITAL	<Caps Lock>	
3B	59	70	112	vk_F1	<F1>	
3C	60	71	113	vk_F2	<F2>	
3D	61	72	114	vk_F3	<F3>	
3E	62	73	115	vk_F4	<F4>	
3F	63	74	116	vk_F5	<F5>	
40	64	75	117	vk_F6	<F6>	
41	65	76	118	vk_F7	<F7>	
42	66	77	119	vk_F8	<F8>	
43	67	78	120	vk_F9	<F9>	
44	68	79	121	vk_F10	<F10>	
46	70	91	145	vk_SCROLL	<Scroll Lock>	
47	71	67	103	vk_NUMPAD7[7]	<Num> 7	
		24	36	vk_HOME[8]	<Home>	
48	72	68	104	vk_NUMPAD8[7]	<Num> 8	
		26	38	vk_UP[8]	<стрелка вверх>	
49	73	69	105	vk_NUMPAD9[7]	<Num> 9	
		21	33	vk_PRIOR[8]	<Page Up>	
4A	74	6D	109	vk_SUBTRACT	<Num> -	
4B	75	64	100	vk_NUMPAD4[7]	<Num> 4	
		25	37	vk_LEFT[8]	<стрелка влево>	
4C	76	65	101	vk_NUMPAD5[7]	<Num> 5	
		0C	12	vk_CLEAR[9]		

Таблица П2.1 *(окончание)*

Скан-код		Виртуальный код			Символ[1] или клавиша	Символ +<Shift>
HEX	DEC	Номер		Константа		
		HEX	DEC			
4D	77	66 27	102 39	vk_NUMPAD6[7] vk_RIGHT[8]	<Num> 6 <стрелка вправо>	
4E	78	6B	107	vk_ADD	<Num> +	
4F	79	61 23	97 35	vk_NUMPAD1[7] vk_END[8]	<Num> 1 <End>	
50	80	62 28	98 40	vk_NUMPAD2[7] vk_DOWN[8]	<Num> 2 <стрелка вниз>	
51	81	63 22	99 34	vk_NUMPAD3[7] vk_NEXT[8]	<Num> 3 <Page Down>	
52	82	60 2D	96 45	vk_NUMPAD0[7] vk_INSERT[8]	<Num> 0 <Insert>	
53	83	6E 2E	110 46	vk_DECIMAL[7] vk_DELETE[8]	<Num> . <Delete>	
56	86	E2	226	vk_OEM_102[6]	\	|
57	87	7A	122	vk_F11	<F11>	
58	88	7B	123	vk_F12	<F12>	
5B	91	5B	91	vk_LWIN	<Пуск>	
5C	92	5C	92	vk_RWIN	<Пуск>	
5D	93	5D	93	vk_APPS	<pop menu>	

[1] Естественно, без учета национальной раскладки клавиатуры.

[2] В модуле Windows.pas константа не определена, необходимо пользоваться числовым кодом виртуальной клавиши.

[3] Общая виртуальная клавиша и скан-код для левого и правого <Ctrl>. Для правого младший бит в старшем байте `LParam` равен 1.

[4] Определяется только в Windows XP.

[5] Общая виртуальная клавиша для левого и правого <Shift>.

[6] Общая виртуальная клавиша и скан-код для левого и правого <Alt>. Для правого младший бит в старшем байте `LParam` равен 1.

[7] С включенным <NumLock> на цифровой клавиатуре.

[8] С выключенным <NumLock> на цифровой клавиатуре, а также для клавиш на дополнительной клавиатуре (в последнем случае младший бит в старшем байте `LParam` равен 1).

[9] С выключенным <NumLock>. В Windows и DOS никакой операции не соответствует, но может быть использована в прикладных программах.

[10] Для правой (на цифровой клавиатуре) клавиши <Enter> младший бит в старшем байте `LParam` равен 1.

ПРИЛОЖЕНИЕ 3

Коды символов

В табл. П3.1 приведены коды символов ASCII, которые составляют основу любой кодировки (включая Unicode). В их число (коды 32—127) входят прописные и строчные буквы английского алфавита, числа, знаки препинания, а также некоторые специальные знаки. Кроме приведенных в таблице, в число символов ASCII включают также специальные коды (с номерами от 0 до 31), которые на старых компьютерных терминалах воспринимались, как команды. Из них в настоящее время однозначно интерпретируются только знаки <CR> (Carriage Return — возврат каретки, код 13), <LF> (Line Feed — перевод строки, код 10) и иногда некоторые другие (например, символ табуляции — код 09, начало новой страницы — код 12), но в основном коды-команды сейчас интерпретируются иначе, чем это предполагали их создатели, и однозначного стандарта для них нет. В качестве примера можно привести код 27 (Escape), который первоначально означал отмену предыдущего символа или команды (и в таковом качестве и сейчас используется в клавиатурных кодах, см. *приложение 2*), но одно из главных его применений сегодня — предварять команды управления принтерами, чтобы отличить их от данных в общем байтовом потоке. По этой причине коды с номерами 0—31 в таблице не приводятся, за исключением кода "возврат каретки".

В табл. П3.2 приведены реально действующие русскоязычные кодировки. Отметим, что кодировка ISO 8859-5 на практике не используется, но вследствие того, что стандартизирована ISO *(см. главу 8)*, присутствует в браузерах или почтовых программах, основанных на кодах сообщества Mozilla (FireFox, Netscape, Thunderbird и др.) — это объясняется тем, что Unix-сообщество (а Mozilla первоначально создавалось именно для Unix) всегда стремится максимально стандартизировать программы, а не выдумывать что-то свое (как Windows с ее кодировкой Win1251), и не его вина, что реально используемые кодировки официальному стандарту не соответствуют.

Таблица П3.1. *Коды символов ASCII*

Символ	HEX	DEC	Символ	HEX	DEC	Символ	HEX	DEC
LF	0D	13	?	3F	63	_	5F	95
Space	20	32	@	40	64	`	60	96
!	21	33	A	41	65	a	61	97
"	22	34	B	42	66	b	62	98
#	23	35	C	43	67	c	63	99
$	24	36	D	44	68	d	64	100
%	25	37	E	45	69	e	65	101
&	26	38	F	46	70	f	66	102
'	27	39	G	47	71	g	67	103
(	28	40	H	48	72	h	68	104
)	29	41	I	49	73	i	69	105
*	2A	42	J	4A	74	j	6A	106
+	2B	43	K	4B	75	k	6B	107
,	2C	44	L	4C	76	l	6C	108
-	2D	45	M	4D	77	m	6D	109
.	2E	46	N	4E	78	n	6E	110
/	2F	47	O	4F	79	o	6F	111
0	30	48	P	50	80	p	70	112
1	31	49	Q	51	81	q	71	113
2	32	50	R	52	82	r	72	114
3	33	51	S	53	83	s	73	115
4	34	52	T	54	84	t	74	116
5	35	53	U	55	85	u	75	117
6	36	54	V	56	86	v	76	118
7	37	55	W	57	87	w	77	119
8	38	56	X	58	88	x	78	120
9	39	57	Y	59	89	y	79	121
:	3A	58	Z	5A	90	z	7A	122
;	3B	59	[	5B	91	{	7B	123
<	3C	60	\	5C	92	|	7C	124
=	3D	61	]	5D	93	}	7D	125
>	3E	62	^	5E	94	~	7E	126

Таблица П3.2. *Русскоязычные кодировки*

Символ	KOI8-R (Unix)		CP1251 (Win)		CP866 (DOS)		Mac		ISO 8859-5		Unicode[1]	
	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
А	E1	225	C0	192	80	128	80	128	B0	176	0410	16
Б	E2	226	C1	193	81	129	81	129	B1	177	0411	17
В	F7	247	C2	194	82	130	82	130	B2	178	0412	18
Г	E7	231	C3	195	83	131	83	131	B3	179	0413	19
Д	E4	228	C4	196	84	132	84	132	B4	180	0414	20
Е	E5	229	C5	197	85	133	85	133	B5	181	0415	21
Ж	F6	246	C6	198	86	134	86	134	B6	182	0416	22
З	FA	250	C7	199	87	135	87	135	B7	183	0417	23
И	E9	233	C8	200	88	136	88	136	B8	184	0418	24
Й	EA	234	C9	201	89	137	89	137	B9	185	0419	25
К	EB	235	CA	202	8A	138	8A	138	BA	186	041A	26
Л	EC	236	CB	203	8B	139	8B	139	BB	187	041B	27
М	ED	237	CC	204	8C	140	8C	140	BC	188	041C	28
Н	EE	238	CD	205	8D	141	8D	141	BD	189	041D	29
О	EF	239	CE	206	8E	142	8E	142	BE	190	041E	30
П	F0	240	CF	207	8F	143	8F	143	BF	191	041F	31
Р	F2	242	D0	208	90	144	90	144	C0	192	0420	32
С	F3	243	D1	209	91	145	91	145	C1	193	0421	33
Т	F4	244	D2	210	92	146	92	146	C2	194	0422	34
У	F5	245	D3	211	93	147	93	147	C3	195	0423	35
Ф	E6	230	D4	212	94	148	94	148	C4	196	0424	36
Х	E8	232	D5	213	95	149	95	149	C5	197	0425	37
Ц	E3	227	D6	214	96	150	96	150	C6	198	0426	38
Ч	FE	254	D7	215	97	151	97	151	C7	199	0427	39
Ш	FB	251	D8	216	98	152	98	152	C8	200	0428	40
Щ	FD	253	D9	217	99	153	99	153	C9	201	0429	41
Ъ	FF	255	DA	218	9A	154	9A	154	CA	202	042A	42

Таблица П3.2 *(продолжение)*

Символ	KOI8-R (Unix)		CP1251 (Win)		CP866 (DOS)		Mac		ISO 8859-5		Unicode[1]	
	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
Ы	F9	249	DB	219	9B	155	9B	155	CB	203	042B	43
Ь	F8	248	DC	220	9C	156	9C	156	CC	204	042C	44
Э	FC	252	DD	221	9D	157	9D	157	CD	205	042D	45
Ю	E0	224	DE	222	9E	158	9E	158	CE	206	042E	46
Я	F1	225	DF	223	9F	159	9F	159	CF	207	042F	47
а	C1	193	E0	224	A0	160	E0	224	D0	208	0430	48
б	C2	194	E1	225	A1	161	E1	225	D1	209	0431	49
в	D7	215	E2	226	A2	162	E2	226	D2	210	0432	50
г	C7	199	E3	227	A3	163	E3	227	D3	211	0433	51
д	C4	196	E4	228	A4	164	E4	228	D4	212	0434	52
е	C5	197	E5	229	A5	165	E5	229	D5	213	0435	53
ж	D6	214	E6	230	A6	166	E6	230	D6	214	0436	54
з	DA	218	E7	231	A7	167	E7	231	D7	215	0437	55
и	C9	201	E8	232	A8	168	E8	232	D8	216	0438	56
й	CA	202	E9	233	A9	169	E9	233	D9	217	0439	57
к	CB	203	EA	234	AA	170	EA	234	DA	218	043A	58
л	CC	204	EB	235	AB	171	EB	235	DB	219	043B	59
м	CD	205	EC	236	AC	172	EC	236	DC	220	043C	60
н	CE	206	ED	237	AD	173	ED	237	DD	221	043D	61
о	CF	207	EE	238	AE	174	EE	238	DE	222	043E	62
п	D0	208	EF	239	AF	175	EF	239	DF	223	043F	63
р	D2	210	F0	240	E0	224	F0	240	E0	224	0440	64
с	D3	211	F1	241	E1	225	F1	241	E1	225	0441	65
т	D4	212	F2	242	E2	226	F2	242	E2	226	0442	66
у	D5	213	F3	243	E3	227	F3	243	E3	227	0443	67
ф	C6	198	F4	244	E4	228	F4	244	E4	228	0444	68
х	C8	200	F5	245	E5	229	F5	245	E5	229	0445	69

Таблица П3.2 *(окончание)*

Символ	KOI8-R (Unix)		CP1251 (Win)		CP866 (DOS)		Mac		ISO 8859-5		Unicode[1]	
	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
ц	C3	195	F6	246	E6	230	F6	246	E6	230	0446	70
ч	DE	222	F7	247	E7	231	F7	247	E7	231	0447	71
ш	DB	219	F8	248	E8	232	F8	248	E8	232	0448	72
щ	DD	221	F9	249	E9	233	F9	249	E9	233	0449	73
ъ	DF	223	FA	250	EA	234	FA	250	EA	234	044A	74
ы	D9	217	FB	251	EB	235	FB	251	EB	235	044B	75
ь	D8	216	FC	252	EC	236	FC	252	EC	236	044C	76
э	DC	220	FD	253	ED	237	FD	253	ED	237	044D	77
ю	C0	192	FE	254	EE	238	FE	254	EE	238	044E	78
я	D1	209	FF	255	EF	239	DF	223	EF	239	044F	79
Ё	B3	179	A8	168	F0	240	DD	221	A1	161	0401	1
ё	A3	163	B8	184	F1	241	DE	222	F1	241	0451	81
«	—		AB	171	—		C7	199	—		00AB	171
»	—		BB	187	—		C8	200	—		00BB	187
№	BE[2]	190	B9	185	FC	252	DC	220	F0	240	2116	8470
тире –	—		96	150	—		D0	208	—		2013	8211
тире —	—		97	151	—		D1	209	—		2014	8212
Неразрывный пробел	9A, A0[3]	154 160	A0	160	FF	255	CA	202	A0	160	00A0	160

[1] В десятичном представлении номеров буквенных символов приведено значение только младшего байта двухбайтной кодировки (для получения абсолютного значения к этой величине надо прибавить 1024), а для спецсимволов — полное абсолютное значение.

[2] KOI8 (RFC 1489) предписывает для кода BE символ "BOX DRAWINGS DOUBLE VERTICAL AND HORIZONTAL" (крест из двойных горизонтальных и вертикальных линий), но де-факто большинство KOI8-шрифтов размещают в этом месте символ номера "№".

[3] KOI-8R (RFC 1489) предписывает для кода A0 символ "BOX DRAWINGS DOUBLE HORIZONTAL" (двойная горизонтальная черта), а для неразрывного пробела отводится код 9A. Однако в целях большей совместимости согласно ISO-IR-111 в A0 помещают неразрывный пробел.

ПРИЛОЖЕНИЕ 4

Последовательные порты компьютера COM и USB

В этом приложении мы подробнее рассмотрим физическую реализацию последовательного порта с точки зрения внешних устройств, начиная с разъема на корпусе компьютера и далее, вплоть до вопросов программирования UART, входящего в состав современных микроконтроллеров (на примере Atmel AVR). Все эти решения годятся и для организации последовательного канала передачи через USB, при условии использования переходного кабеля-преобразователя COM-USB (см. далее). Для USB, ввиду его много большей в сравнении с COM-портом сложности, мы здесь рассмотрим только один из вариантов схемотехнической реализации устройства USB/RS232, наиболее подходящий для использования в простых технических устройствах.

Принципы передачи информации по интерфейсу RS-232

Существует несколько стандартов RS-232, различающихся буквой в суффиксе: RS-232C, RS-232D, RS-232E и пр. Вдаваться в различия между ними нет никакого смысла — они являются лишь последовательным усовершенствованием и детализацией технических особенностей одного и того же устройства. Все современные порты поддерживают спецификации RS-232D или RS-232E. В состав любого порта с интерфейсом RS-232 (в том числе COM-порта PC) входит универсальный асинхронный приемопередатчик (Universal Asynchronous Receiver-Transmitter, UART), который потому и носит название "универсального", что одинаков для всех подобных интерфейсов (кроме RS-232, это RS-485 и RS-422[1]). Также в RS-232 входит схема преобразования

[1] Некоторые подробности по поводу интерфейсов, отличающихся от RS-232 аппаратной реализацией, что позволяет достичь более высоких скоростей передачи, вы можете найти в [33].

логических уровней UART (это обычные логические уровни 0÷5 или 0÷3,3 В) в уровни RS-232, где биты передаются разнополярными уровнями напряжения, притом инвертированными относительно UART. В UART действует положительная логика, где логическая 1 есть высокий уровень (+3 или +5 В), а у RS-232 наоборот, логическая 1 есть отрицательный уровень от –3 до –12 В, а логический 0 — положительный уровень от +3 до +12 В.

Сама идея передачи по этому интерфейсу заключается в передачи целого байта по одному проводу в виде последовательных импульсов, каждый из которых может быть 0 или 1. Если в определенные моменты времени считывать состояние линии, то можно восстановить то, что было послано. Однако эта простая идея натыкается на определенные трудности. Для приемника и передатчика, связанных между собой тремя проводами ("земля" и два сигнальных провода "туда" и "обратно"), приходится задавать скорость передачи и приема, которая должна быть одинакова для устройств на обоих концах линии. Эти скорости стандартизированы, и выбираются из ряда 1200, 2400, 4800, 9600, 14 400, 19 200, 28 800, 38 400, 56 000, 57 600, 115 200, 128 000, 256 000 (более медленные скорости я опустил)[2]. Число это обозначает количество передаваемых/принимаемых бит в секунду (бод). Проблема состоит в том, что приемник и передатчик — это физически совершенно разные системы, и скорости эти для них не могут быть строго одинаковыми в принципе (из-за разброса параметров тактовых генераторов), и даже если их каким-то фантастическим образом синхронизировать в начале, то они в любом случае быстро "разъедутся". Поэтому такая передача всегда сопровождается начальным (стартовым) битом, который служит для синхронизации. После него идут восемь (или девять — если используется проверка на четность) информационных битов, а затем стоповые биты, которых может быть один, два и более, но это уже не имеет принципиального значения — почему, мы сейчас увидим.

Общая диаграмма передачи таких последовательностей показана на рис. П4.1. Хитрость заключается в том, что состояния линии передачи, называемые стартовый и стоповый биты, имеют разные уровни. В данном случае стартовый бит передается положительным уровнем напряжения (логическим нулем), а стоповый — отрицательным уровнем (логической единицей)[3], по-

[2] Отметим, что стандарт RS-232E устанавливает максимальную скорость передачи 115 200, однако функции Windows позволяют установить более высокую скорость.

[3] Отсюда следует важный для практики вывод — большую часть времени линия пребывает в состоянии с отрицательным напряжением, и мы этим фактом воспользуемся далее для создания преобразователя уровней. Именно такая комбинация уровней имеет большой смысл со схемотехнической точки зрения — со стороны UART стоповый бит должен иметь высокий уровень, что соответствует состоянию вывода "с открытым коллектором" (или "с открытым истоком"), когда большую часть времени выходной транзистор выключен, т. е. вывод не потребляет тока.

тому фронт стартового бита всегда однозначно распознается. В этот-то момент и происходит синхронизация. Приемник отсчитывает время от фронта стартового бита, равное ¾ периода заданной частоты обмена (чтобы попасть примерно в середину следующего бита), и затем восемь (или девять, если это задано заранее) раз подряд с заданным периодом регистрирует состояние линии. После этого линия переходит в состояние стопового бита и может в нем пребывать сколь угодно долго, пока не придет следующий стартовый бит. Задание минимального количества стоповых битов, однако, производится тоже — для того чтобы приемник знал, сколько времени минимально ему нужно ожидать следующего стартового бита (как минимум, это может быть, естественно, один период частоты обмена, т. е. один стоповый бит). Если по истечении этого времени стартовый бит не придет, приемник может регистрировать так называемый Timeout, т. е. перерыв, по-русски, и заняться своими делами. Если же линия "зависнет" в состоянии логического 0 (высокого уровня напряжения), то это может восприниматься устройством, как состояние "обрыва" линии — не очень удобный механизм, и в микроконтроллерах он через UART не поддерживается. Это не мешает нам, естественно, для установки или определения такого состояния просто отключать UART и устанавливать состояние логического нуля на выводе TxD (что и есть имитация физического "обрыва"), или определять уровень логического 0 на выводе RxD, но серьезных причин для использования этой возможности, я, честно говоря, не вижу (см. на эту тему также замечание в *главе 20*).

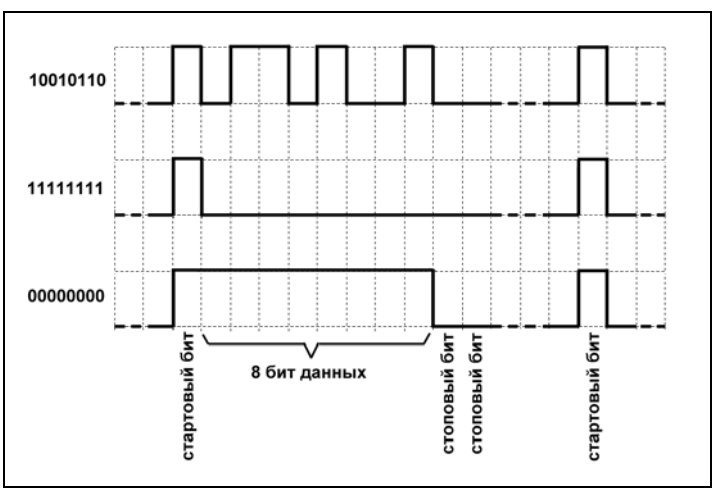


Рис. П4.1. Диаграмма передачи данных по последовательному интерфейсу RS-232 в формате 8N2

Обычный формат данных, по которому работает львиная доля всех устройств, обозначается 8n1, что читается так: 8 информационных бит, no parity,

1 стоповый бит. "No parity" означает, что проверка на четность не производится. Это самая распространенная схема работы такого порта, причем, т. к. никакими тайм-аутами (Timeout) мы также себе голову заморачивать не будем, то нам в принципе все равно, сколько стоповых битов будет, но во избежание излишних сложностей следует их устанавливать всегда одинаково — у передатчика и у приемника. На диаграмме рис. П4.1 показана передача некоего кода, а также, для наглядности, передача байта, состоящего из всех единиц и из всех нулей в формате, опять же для наглядности, 8n2.

Из описанного алгоритма работы понятно, что погрешность несовпадения скоростей обмена может быть такой, чтобы фронты не "разъезжались" за время передачи/приема всех десяти-двенадцати битов более, чем на полпериода, т. е. в принципе фактическая разница скоростей может достигать 4—5%, но на практике их стараются все же сделать как можно ближе к стандартным величинам.

Приемник RS-232 часто дополнительно снабжают схемой, которая фиксирует уровень не единожды за период действия бита, а трижды, при этом за окончательный результат принимается уровень двух одинаковых из трех полученных состояний линии, таким образом удается избежать случайных помех. Длина линии связи по стандарту не должна превышать 15 м, но на практике это могут быть много большие величины. Если скорость передачи не выбирать слишком высокой, то такая линия может надежно работать на десятки метров (автору этих строк удавалось без дополнительных ухищрений наладить обмен с компьютером на скорости 4800 по кабелю, правда, довольно толстому, длиной около полукилометра). В табл. П4.1 приведены ориентировочные эмпирические данные по длине неэкранированной линии связи для различных скоростей передачи.

Таблица П4.1. *Длина кабеля RS-232 для разных скоростей передачи данных*

Скорость, бод	Длина кабеля (неэкранированного), м
9600	75
2400	150
110	900

Эти данные ни в коем случае не могут считаться официальными — слишком много влияющих факторов (уровень помех, толщина проводов, их взаимное расположение в кабеле, фактические уровни напряжения, выходное/входное

сопротивление портов и т. п.). В случае экранированного кабеля[4] эти величины можно увеличить примерно в полтора-два раза. Во всех случаях использования "несанкционированной" длины кабеля связи следует применять меры по дополнительной проверке целостности данных — контроль четности, и/или программные способы (вычисление контрольных сумм и т. п.), описанные в *главе 20*.

Для работы в обе стороны нужно две линии, которые у каждого приемопередатчика обозначаются RxD (приемная) и TxD (передающая). В каждый момент времени может работать только одна из линий, т. е. приемопередатчик либо передает, либо принимает данные, но не одновременно (так называемый "полудуплексный режим" — это сделано потому, что у UART-микросхем чаще всего один регистр и на прием и на передачу). Кроме линий RxD и TxD, в разъемах RS-232 присутствуют также и другие линии. Полный список всех контактов для обоих стандартных разъемов типа DB (9- и 25-контактного) приведен в табл. П4.2. Нумерация контактов DB-разъема обычно написана прямо на нем, она также есть на рис. 10.8 в *главе 10* (на примере гнезда разъема для игрового порта DB-15F).

Таблица П4.2. *Контакты для DB-разъемов*

COM 9(25)	Обозначение	Направление	Сигнал
1 (8)	DCD	Вход	Детектор принимаемого сигнала с линии (Data Carrier Detect)
2 (3)	RxD	Вход	Принимаемые данные (Receive Data)
3 (2)	TxD	Выход	Передаваемые данные (Transmit Data)
4 (20)	DTR	Выход	Готовность выходных данных (Data Terminal Ready)
5 (7)	GND	—	Общий (Ground)
6 (6)	DSR	Вход	Готовность данных (Data Set Ready)
7 (4)	RTS	Выход	Запрос для передачи данных (Request To Send)

[4] В правильно построенной экранированной линии экран не должен быть одним из токоведущих проводов, то есть контакты GND соединяются отдельным проводом в кабеле, а экран либо соединяется с GND только на одной стороне — там, где имеется более качественное настоящее заземление, либо — в случае, если сигнальная "земля" GND является "плавающей" относительно "настоящей" земли — вообще только с заземлением.

Таблица П4.2 *(окончание)*

COM 9(25)	Обозначение	Направление	Сигнал
8 (5)	CTS	Вход	Разрешение для передачи данных (Clear To Send)
9 (22)	RI	Вход	Индикатор вызова (Ring Indicator)

Смысл дополнительных линий в том, что они могут применяться для организации различных синхронных протоколов обмена (протоколов с handshakes — "рукопожатием"). В "чистый" UART они не входят, в контроллере их организуют выводами обычных портов (но они входят в отдельные микросхемы UART для реализации полного протокола RS-232). Большинство устройств их не использует[5]. Однако любое устройство, применяющее "рукопожатия", можно подключить к устройству, их не использующему (потеряв, конечно, возможности синхронизации), если соединить на каждой стороне между собой выводы RTS-CTS, а также выводы DSR, DCD и DTR.

Для нормальной совместной работы приемника и передатчика выводы RxD и TxD, естественно, нужно соединять накрест — TxD одного устройства с RxD второго и наоборот (то же относится и к RTS-CTS и т. д.). Кабели RS-232, которые устроены именно таким образом, называются еще нуль-модемными (в отличие от простых удлинительных). Их стандартная конфигурация показана на рис. П4.2. В варианте "c" (справа на рисунке) дополнительные выводы соединены именно так, как описано ранее.

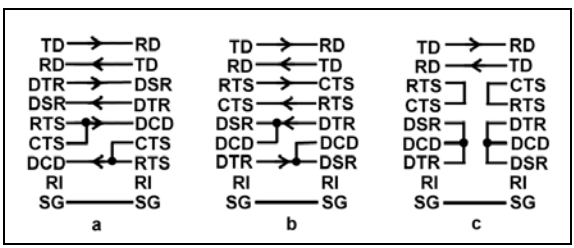


Рис. П4.2. Схемы нуль-модемных кабелей RS-232: *a*, *b* — различные полные варианты, *c* — минимальный вариант

[5] Такие протоколы удобно, например, использовать для автоматической установки скорости обмена (baud rate) — скорости перебираются до тех пор, пока устройство не примет осмысленный байт, тогда оно выставит линию RTS в логическую 1, и обмен будет считаться установленным. Так как это требует лишней "проволоки" (как минимум одной) в кабеле, то решайте сами — надо это вам или нет. Автоматическое установление скорости обмена можно организовать и другими способами.

Выходные линии RTS и DTR иногда могут использовать и для "незаконных" целей — питания устройств, подсоединенных к COM-порту. Именно так устроены, например, компьютерные мыши, работающие через COM. Позже мы покажем пример устройства (преобразователя уровней), которое будет использовать питание от вывода RTS. А как при необходимости можно установить эти линии в нужное состояние?

Установка линии RTS в DOS и Windows

При начальной загрузке компьютера линии RTS и DTR чаще всего оказываются в состоянии с отрицательным уровнем напряжения (от –9 до –12 В), но, вообще говоря, могут оказаться в любом состоянии. В среде DOS и Windows 95/98/ME для установки в положительный уровень можно применить любой имеющийся под рукой DOS-драйвер COM-портовой мыши, который удобно загружать, например, через autoexec.bat прямо при включении компьютера (если только пренебречь опасностью, что при поступлении байтов на этот порт курсор может самопроизвольно начать бегать по экрану и производить всякие нехорошие действия). Однако в Windows NT и других ОС из этого семейства этот способ, естественно, работать не будет. Рассмотрим, как можно установить уровни принудительно на примере линии RTS.

В DOS (этот способ будет также работать и в Windows 9*x*) можно написать простую программку на Turbo Pascal, которая под названием RTSDOS имеется на прилагаемом диске (папка Priloz4). Исходный текст ее расположен в файле RtsDos.pas. Запускается она из командной строки с ключами типа `+COM`*x* или `-COM`*x* (где *x* есть 1, 2, 3 или 4). Если первый символ "+", то линия установится в положительный уровень напряжения, если наоборот — в отрицательный. Если все в порядке, программа вернет (в текстовом режиме) номер порта ввода/вывода для заданного COM ($03F8, например), если его не существует, то не вернется ничего. При запуске без ключа программа выдаст текст (на достаточно корявом английском), рассказывающий примерно то же самое, что я тут описал.

В этой программе мы сначала определяем в служебной области памяти BIOS (сегмент 0040h) номер порта ввода/вывода для заданного COM (они расположены в самых первых адресах этого сегмента, каждый адрес порта занимает двухбайтную ячейку). Если там записаны нули, то порт не существует, если же он есть, то мы используем ассемблерную процедуру для установки линии RTS через прерывание Int14. Подробнее об этом в книге, посвященной Windows-программированию, рассказывать не имеет смысла, интересующихся я отсылаю к литературе по ассемблеру и устройству PC ([13,15]).

Аналогичная Windows-программа называется RTSWIN и расположена в одноименной папке внутри папки Priloz4 на прилагающемся диске. Она работа-

ет также только под Windows 9*x* (относительно семейства NT см. пояснения далее). Написана она в виде консольного приложения, которое запускается по той же методе, что и описанная ранее DOS-программа. Всю информацию программа выдает в текстовое окно.

Отметим две особенности этой программы. Во-первых, в ней используется "незаконный" переход на метку через оператор goto, которым я, признаться, воспользовался с большим удовольствием. А вторая особенность связана с тем, что в Delphi, как указано в *главе 20*, структура DCB транслируется не полностью. В частности, там отсутствует поле fRtsControl, через которое можно управлять режимом линии RTS, зато имеется поле Flags, через биты которого и предлагается в том числе этот режим устанавливать. Последнее не очень удобно (есть опасность что-то порушить по соседству), но опыт показал, что ничего страшного не случается, если сделать все, как надо. Сначала через Flags там устанавливается режим управления дополнительными линиями (константа RTS_CONTROL_HANDSHAKE = $100), при этом само управление осуществляется через функцию EscapeCommFunction, вот так:

```
tpDCB.Flags:=(tpDCB.Flags and $FFFFC0FF) or $00000100;
SetCommState( pCOM, tpDCB);
// Reset RTS
if ch='-' then EscapeCommFunction(pCOM, CLRRTS);
// Set RTS
if ch='+' then EscapeCommFunction(pCOM, SETRTS);
```

Теперь главный вопрос — а почему все это не работает в Windows семейства NT? На самом деле приведенная процедура вполне будет работать в любой Windows (и обязана это делать, т. к. функции Win32 везде одинаковые), но NT тщательно следит за тем, чтобы все ресурсы использовались каждой программой независимо от других. Если запустите указанную программу в XP (удалив, естественно, из нее условие выбора ОС), то изначально установленный, например, в состояние с отрицательным уровнем вывод RTS на долю секунды перейдет в положительное состояние, а потом, когда программа закончит работу, вернется обратно в исходное. То есть установка вывода порта действует только на время работы программы. Отсюда методика управления выводом RTS в семействе NT может быть только такой: если ваше устройство использует вывод RTS для питания, то прилагаемая к нему программа должна устанавливать этот вывод самостоятельно каждый раз при запуске. Для такой установки можно использовать указанную процедуру из утилиты RTSWIN.

Приемы программирования UART в микроконтроллерах на примере AVR

Задача этого раздела — показать, как можно разными способами запрограммировать микроконтроллер (МК) для асинхронного обмена данными с компьютером. Мы ограничимся микроконтроллерами AVR фирмы Atmel, т. к. для остальных их разновидностей (и даже при прямом программировании UART в самом компьютере) методика похожая. Мы рассмотрим программирование для семейства AVR Classic, некоторые особенности семейства AVR Mega будут рассмотрены параллельно. Всех подробностей я, разумеется, изложить не могу, так что для полного понимания придется заглянуть в руководство по AVR-контроллерам.

За портом UART в микроконтроллерах AVR зарезервировано три аппаратных прерывания и несколько регистров портов ввода/вывода, главными из которых являются два — регистр данных UDR и регистр управления USR (UCSRA для Mega). Регистры UBRR (UBRRL и UBRRH для Mega) и UCR (UCSRB и UCSRC) служат для установки скорости и режима обмена.

Начнем с установки скорости (9600) и режима (8n1). Для этого нужно выполнить такую процедуру (temp — любой регистр общего назначения от r16 до r31):

```
ldi     temp,25 ;скорость передачи 9600 при 4 МГц
out     UBRR,temp ;устанавливаем
ldi     temp,(1<<RXEN|1<<TXEN|1<<RXB8|1<<TXB8)
out     UCR,temp ;разрешение приема/передачи 8 битов
```

Здесь мы выбрали частоту генератора МК равной 4 МГц. Смысл первого и второго оператора — задание скорости обмена через задание делителя частоты кварца. Этот делитель выбирается по таблице, имеющейся в каждом руководстве по AVR, или рассчитывается по формуле $BAUD = f_{кв.рез}/16(UBRR + 1)$. Для семейства Mega имеет смысл использовать более высокие частоты, и там процедура несколько усложняется, потому что скорость можно регулировать более точно:

```
ldi temp,103 ;9600 при 16 МГц
out UBRRL, temp
; Enable Receiver and Transmitter
ldi temp, (1<<RXEN)|(1<<TXEN)
out UCSRB,temp
; Set frame format: 8data, 1 stop bit
ldi temp, (1<<URSEL)|(3<<UCSZ0)
out UCSRC,temp
```

При частоте кварца 4 МГц мы с приемлемой точностью можем получить скорости обмена не более 28 800 бод, правда, при выборе специального кварца 3.6864 МГц можно получить с нулевой ошибкой весь набор скоростей вплоть до 115 200. Для получения скоростей передачи выше указанных (стандартно COM-порт позволяет установить скорости, как указано ранее, до 256 Кбод) надо увеличивать частоту и подбирать под нее коэффициент деления. Так, при кварце 8 МГц и коэффициенте деления (UBRR) равном 1, мы получим скорость 250 000, что отличается от стандартных 256 000 на приемлемые 2,4%.

Теперь собственно о приеме/передаче. Если контроллер не перегружен, то хорошо работает упрощенный алгоритм, который вообще не использует прерывания. Он состоит в следующем: в начале программы (например, после перечисления векторов прерываний) вы записываете две коротких процедуры:

```
Out_com: ;посылка байта из temp с ожиданием готовности
       sbis   USR,UDRE ;ждем готовности буфера передатчика
       rjmp   out_com
       out    UDR,temp ;собственно посылка байта
ret ;возврат из процедуры Out_com

In_com: ;прием байта в temp с ожиданием готовности
       sbis   USR,RXC ;ждем готовности буфера приемника
       rjmp   in_com
       in     temp,UDR ;собственно прием байтов
ret ;возврат из процедуры In_com
```

Для семейства Mega надо заменить все USR на UCSRA. Обращение к процедуре In_com при этом вставляется в пустой цикл в конце программы:

```
Cykle:
       rcall  In_com
. . . . .
<анализируем полученный в temp байт, и что-то с ним делаем, например,
посылаем ответ через процедуру Out_com>
. . . . .
rjmp   Cykle ;зацикливаем программу
```

При таком способе контроллер большую часть времени ожидает приема, выполняя проверку регистра UDR (в процедуре In_com), этот процесс прерывается только на время выполнения настоящих прерываний. Прерывания все равно должны выполняться много быстрее, чем байт, в случае, если он пришел,

успевает в UDR смениться следующим, так что мы ничего особо не потеряем. А процедура посылки Out_com сама по себе может выполняться долго (если посылать несколько байтов подряд), но также в основном будет заключаться в том, что контроллер будет ожидать очищения UDR, и т. к. это не прерывание, то процедура в любой момент может быть прервана настоящим прерыванием, и мы опять же ничего не теряем. Но чтобы ничего действительно не потерять, при таком способе нужно быть очень внимательным: так, нужно следить за использованием temp. Правда, если мы будем применять процедуру Out_com внутри процедуры прерывания, куда другое прерывание "влезть" не может, то temp меняться не будет, но тогда контроллер будет терять время на ожидание, потому лучше, кроме разве что расстановки контрольных точек при отладке программы, так не поступать.

С использованием прерываний процедура усложняется, но зато все будет послано и передано гарантированно и без потерь времени (в случае, конечно, если вычислительной мощности контроллера хватает). Мы покажем принцип организации такой процедуры для более-менее общей задачи: контроллер, как и ранее, все время ожидает команды от компьютера, и при ее получении должен послать в ответ некоторое количество байтов.

Сначала вы инициализируете прерывание "прием закончен" (для чего надо установить бит RXCIE в регистре UCR). Возникновение этого прерывания означает, что в регистре данных UDR имеется принятый байт. Процедура обработки этого прерывания тогда выглядит так:

```
UART_RXC:
        in temp,UDR ;принятый байт — в переменной temp
        cbi UCR,RXCIE ;запрещаем прерывание "прием закончен"
. . . . .
<анализируем команду, если это не та команда — опять разрешаем прерывание
"прием закончен" и выходим из процедуры по метке END_R:
        sbi UCR,RXCIE
        rjmp END_R
В противном случае готовим данные, самый первый посылаемый байт должен
быть в переменной temp>
. . . . .
        sbi UCR,UDRIE ;разрешение прерывания "регистр данных пуст"
END_R:
reti
```

Далее у нас почти немедленно возникает прерывание "регистр данных пуст".

Обработчик этого прерывания состоит в том, что мы посылаем байт, содержащийся в переменной temp, и готовим следующие данные:

```
UART_DRE:
        out UDR,temp ;посылаем байт
        cbi UCR,UDRIE ;запрещаем прерывание "регистр данных пуст"
. . . . .
<готовим данные, следующий байт - в temp. Если же был отправлен последний
нужный байт, то опять разрешаем прерывание "прием закончен" и далее rjmp
на метку END_, иначе выполняем следующий оператор:>
        sbi UCR,UDRIE ;разрешаем прерывание "регистр данных пуст"
END_:
reti
```

Обратим еще раз внимание на то, что переменная `temp` не должна в промежутках между прерываниями использоваться еще где-то, в противном случае ее надо сохранять, например, в стеке, но удобнее все же отвести для этого дела специальный регистр.

Сравните данные методы с процедурами для Windows в *главе 20*, и вы, несомненно, найдете определенное идеологическое сходство.

Преобразователи уровня UART/RS-232

Вопрос о том, как преобразовать уровни UART в уровни RS-232, имеет более важное значение, чем кажется. Так как UART и RS-232 оперируют с логикой противоположной полярности, то и для приема и передачи при этом удобно использовать простые транзисторные ключи, которые инвертируют сигнал (ах, как разработчики стандарта были предусмотрительны!). Такая схема преобразователя уровня показана на рис. П4.3. В ней мы использовали отмеченный ранее факт, что линия TxD со стороны компьютера большую часть времени пребывает в состоянии низкого уровня, и мы запасаем это напряжение на конденсаторе через диод, а потом используем его при передаче. Это несколько снижает входное сопротивление линии RxD устройства (и повышает выходное TxD), но в принципе прекрасно работает, даже если байты идут туда-сюда сплошным потоком.

"Официальный" путь состоит в том, чтобы применять специальные микросхемы приемопередатчиков RS-232 (правильнее их было бы называть преобразователями уровня), это, например, MAX202, MAX232, ADM202 и подобные, которые содержат внутри преобразователь-инвертор напряжения. Вариант построения такой схемы показан на рис. П4.4. Выходные уровни вывода TxD здесь при интенсивном обмене не менее ±7 В.

Применение таких приемопередатчиков не решает одной проблемы — гальванической развязки устройства с COM-портом. А такая развязка очень даже может понадобиться — на корпусе компьютера "висит" обычно вполне при-

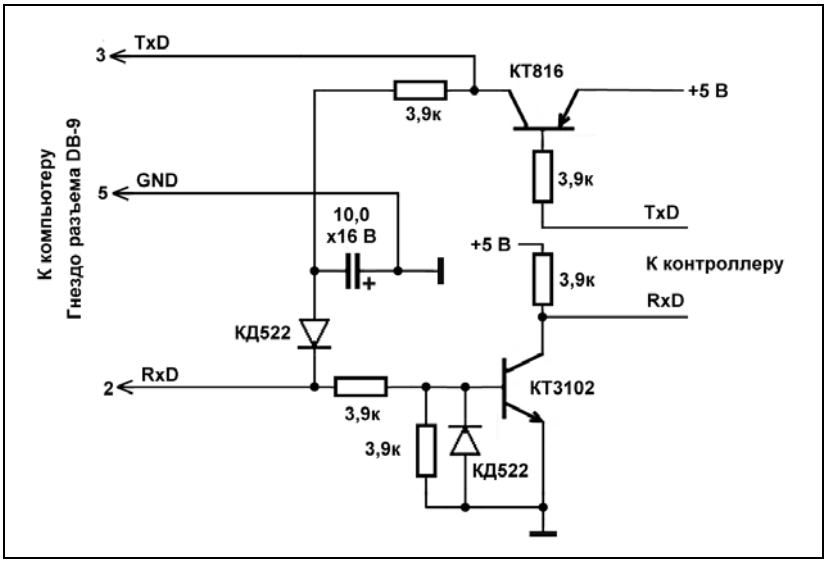


Рис. П4.3. Простейший вариант самодельного преобразователя уровней RS-232 — UART при соединении контроллера с компьютером

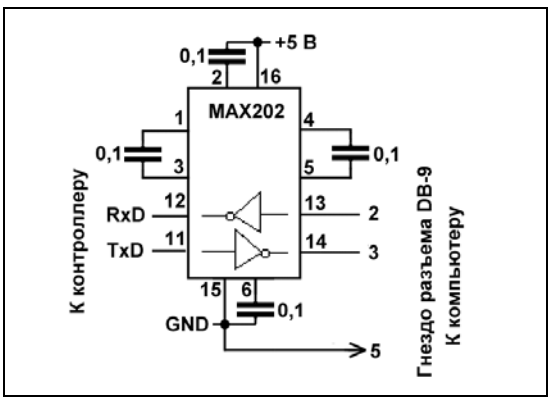


Рис. П4.4. Вариант одноканального преобразователя уровней RS-232 — UART на микросхеме MAX202

личный потенциал[6]. Автор этих строк однажды чуть не убил одно несчастное животное (до сих пор в кошмарах вспоминается), когда проектировал

[6] По этой причине, кстати, нужно внешнее металлическое обрамление разъемов DB-9 соединять с "землей" и со стороны компьютера, и со стороны прибора — оно первое входит в соприкосновение и потенциалы выравниваются до того, как успевают соприкоснуться контакты разъема. Заставлять пользователей подключать устройства исключительно при выключенном компьютере — прошлый век.

прибор для целей измерения внутричерепного давления у обезьян. Главная причина всей этой катавасии — в отсутствии, разумеется, нормального заземления в наших постройках, но даже в редчайшем случае его наличия все равно развязка не помешает. Один из вариантов такой развязки с использованием относительно быстродействующего оптрона типа 6N139 показан на рис. П4.5.

Верхняя часть схемы (оптрон D1) служит для передачи сигналов от контроллера к компьютеру. Сигнал TxD с контроллера должен иметь положительный уровень не ниже 4,5 В под нагрузкой, в противном случае следует увеличить номинал резистора R1.

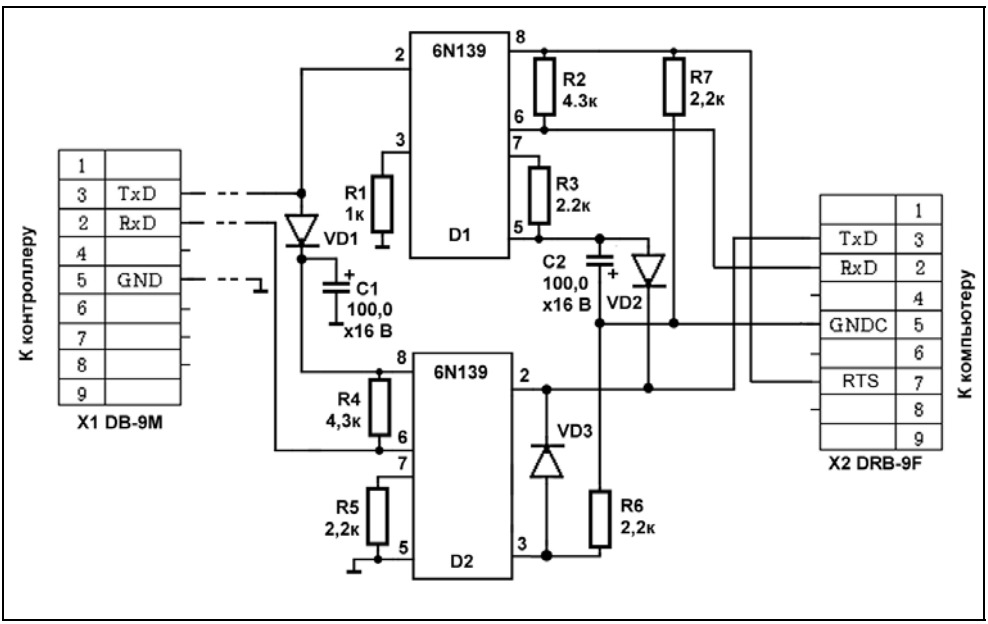


Рис. П4.5. Вариант одноканального преобразователя уровней RS-232 — UART с гальванической развязкой и питанием от линии RTS

Положительный уровень сигнала, поступающего на вход RxD COM-порта, обеспечивается от линии RTS, заранее установленной в положительный уровень напряжения описанным ранее образом. Когда линия TxD COM-порта простаивает, то отрицательное напряжение с нее накапливается через диод VD2 на конденсаторе C2 и тем самым обеспечивается отрицательный уровень этого сигнала.

Приемная часть построена на оптроне D2. Ток через входной светодиод оптрона идет во время положительного уровня напряжения на линии TxD COM-порта, а диод VD3 защищает этот светодиод от обратного напряжения.

Со стороны контроллера питание выходного каскада оптрона D2 обеспечивается способом, аналогичным питанию выхода D1. Так как сигнал TxD контроллера почти все время находится в состоянии +5 В, это напряжение через диод VD1 накапливается на конденсаторе C1 и поступает на питание выходного транзистора оптрона.

Оптоизолятор выполнен в форме удлинительного кабеля для COM-порта. Разъем X2 типа DRB и остальные детали, кроме разъема X1, монтируют на макетной плате размерами 30×60 мм. С противоположной от разъема стороны распаивают трехжильный плоский кабель примерно 0,5 м длиной, закрепляют его на плате и соединяют с разъемом X1. Разъем X1 может быть любого удобного типа, например, IDC. После проверки плату затягивают в отрезок термоусадочного кембрика подходящего диаметра. Разъем X2 вместе с платой втыкается прямо в COM-порт.

Схема для преобразования USB/RS-232

Наилучшим для практики способом построения последовательного порта через USB-интерфейс является использование микросхемы FT232BM[7] английской фирмы Future Technology Devices International Ltd — FTDI. С возможностями ее и других USB-микросхем этой фирмы можно ознакомиться из хорошей подборки статей на сайте компании "ЭФО" [34]. Самым главным преимуществом этой микросхемы является наличие драйверов для Windows (притом бесплатно распространяемых), которые обеспечат в том числе и полную эмуляцию последовательного COM-порта со скоростями до 1 Мбод *(см. главу 20)*. На рис. П4.6 я без лишних объяснений привожу схему кабеля USB/RS-232, которая без изменений честно заимствована из фирменной документации FTDI [35]. В русифицированном варианте эта схема приведена в [29], где она также честно заимствована из размещенной на сайте екатеринбургской фирмы "Институт радиотехники" (**http://www.institute-rt.ru/common/statyi/conv1/index.html**) статьи А. Лысенко, Р. Назмутдинова, И. Малыгина для журнала "Радио" (2002, № 6 и 7).

Согласно уверениям производителя, если вы просто припаяете микросхему FT232BM без дополнительного программирования внешней EEPROM (микросхема 93C46 на схеме), в которой должны храниться идентификаторы устройства и прочая служебная информация, и даже вообще без нее, то устройство все равно будет работать, хотя могут возникнуть сложности с подключением других подобных устройств. Если же есть желание EEPROM

[7] Имейте в виду, что подключение снятой с производства FT232AM (ее полное название FT8U232AM) несколько отличается.

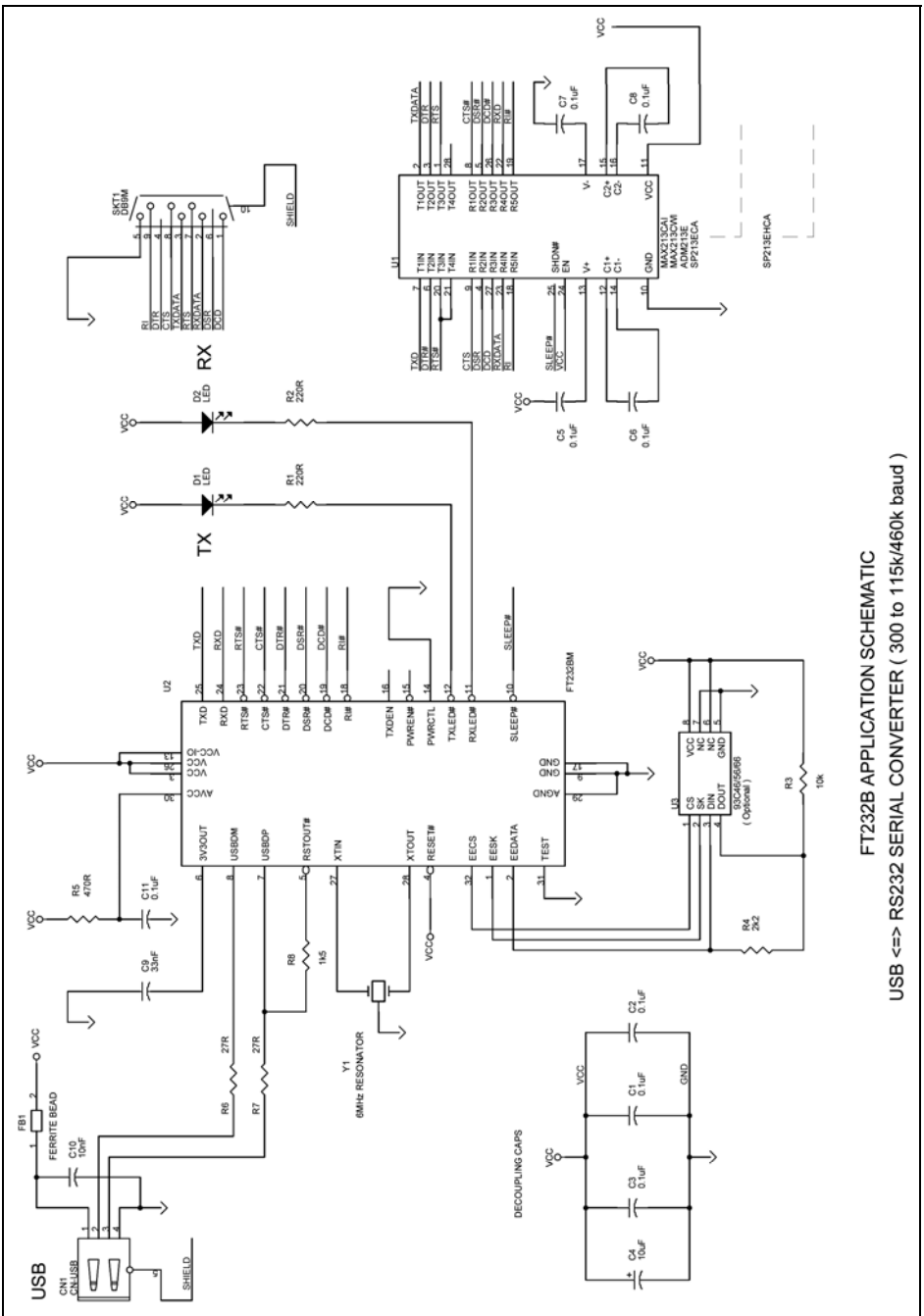


Рис. П4.6. Рекомендуемая производителем схема преобразователя USB/RS-232 с использованием микросхемы FT232BM

запрограммировать, то специально этим заниматься не требуется, при установке драйвера типа D2XX (как указано в *главе 20*), это можно сделать прямо на готовой плате через специальную фирменную утилиту EditEEPROM. Есть, по слухам, некоторые особенности с обеспечением скоростного режима этих микросхем, но вдаваться в подробности в рамках этой книги не имеет смысла. Читатель, несомненно, в курсе того, что подобные готовые "переходники" COM/USB имеются в продаже вместе с уже настроенным драйвером.

Имейте в виду, что обеспечить максимальную скорость обмена — здесь проблема не интерфейса, а применяемых компонентов, так, в схеме по рис. П4.6 преобразователи MAX213 или ADM213 могут обеспечить 115 Кбод, микросхема SP213 обеспечит 500 Кбод, а 1 Мбод вы получите только с использованием MAX3245, правда, при этом встанет необходимость еще и запрограммировать виртуальный COM-порт на такие скорости (о чем см. *главу 20*). На самом деле применять именно такую схему, как приведена на рис. П4.6, есть нужда только при использовании устройств с уже готовым интерфейсом RS-232 (как, например, разбираемые в *главе 20* GPS-навигаторы). Если вы устройство целиком проектируете самостоятельно, то нет никакого смысла преобразовывать уровни UART в уровни RS-232 и обратно, дважды устанавливая приемопередатчик — в этом случае его из схемы на рис. П4.6 надо исключить, а вместо него линии RxD и TxD подсоединить прямо к контроллеру. Остальные линии просто могут не использоваться, вывод CTS# микросхемы FT232BM при этом следует заземлить.

ПРИЛОЖЕНИЕ 5

Описание компакт-диска

На прилагаемом диске (табл. П5.1) расположены все примеры в том виде, в котором они описаны в тексте. Название папки GLAVA*x* соответствует номеру главы *x* в книге. Вложенные папки располагаются в порядке представления их содержимого по тексту главы.

Таблица П5.1. *Содержимое компакт-диска*

Папка	Описание
GLAVA2	Первоначальный проект SlideShow
GLAVA3	
\1	Сворачивание приложения в Tray Bar при потере фокуса
\2	Сворачивание приложения в Tray Bar вместо закрытия
\3	Сворачивание приложения в Tray Bar вместо минимизации
GLAVA4	
\1	Предотвращение повторного запуска приложения
\2	Демонстрация заставки. Сворачивание в Tray Bar при запуске
GLAVA5	
\1	Горячая клавиша с вызовом всплывающего меню
\2	Простая программа в виде иконки — отладочный пример
\3	Заготовка программы для исправления раскладки текста
\4	Резидентная программа для исправления раскладки текста
GLAVA6	
\1	Определение виртуального и скан-кода клавиши
\2	Клавиатурный шпион

Таблица П5.1 *(продолжение)*

Папка	Описание
GLAVA7	
\1	Самый простой переключатель раскладки
\2	Переключатель с заменой системной иконки
\3	Переключатель с установками
GLAVA8	
\1	Программа преобразования Unicode в чистый текст
\2	Проблема автоматического переключения раскладки в `RichEdit`
\2\Lib	Компонент `RichEditInt`
\3	Автоматическое определение кодировки текстовых файлов
\4	Доработка программы для исправления раскладки текста
GLAVA9	
\1	Диалог типа `MessageBox`
\2	Диалог для установки таймера в SlideShow
\3	Диалог с установкой нескольких параметров и сохранение установок
GLAVA10	
\1	Преобразование BitMap в иконку
\2	Приложение-термометр с иконкой в Tray
\CTRL-A	Иллюстрация к операциям над пикселами
GLAVA11	
\1	Наглядная агитация (ресурсы исполняемого файла)
\2	Заставка и номер версии в SlideShow
\3	Номер версии в приложении без формы
GLAVA12	
\1	Красивая заставка в SlideShow
\2	Прозрачная форма и окно flystyle
GLAVA13	
\1	Прокрутка в компоненте `ScrollBox` и режим Drag&Drop
\2	Программа для поиска файлов
\POLZUNOK	Пример к работе с индикаторами длительности процесса

Таблица П5.1 *(окончание)*

Папка	Описание
GLAVA14	
\1	Составление списка вложенных папок и др. (программа Trace)
\2	Оптимизация чтения через mapped files. Сохранение настроек
GLAVA15	Доделываем SlideShow
\MUSIC	Музыкальные клипы в формате WAV
GLAVA16	
\1	Справка и пункт **О программе** для Trace
\2	Справка для переключателя клавиатуры
\3	Справка в SlideShow
GLAVA17	
\SETUP	Инсталлятор для переключателя раскладки клавиатуры
\UNINSTALL	Деинсталлятор для переключателя раскладки клавиатуры
GLAVA18	
\1	Работа с Word через объект `Word Basic`
\2	Работа с Word через объект VBA
\3	Доработка программы Trace
GLAVA19	
\2	Операция XOR и простейшее шифрование файлов
\1	Стеганография на коленке
GLAVA20	
\1	Прием и передача одного или нескольких байтов через COM-порт
\2	Прием и передача в реальном времени
\3	Прием и передача данных с помощью компонента `AsyncFree`
\4	Программа для чтения данных с GPS-навигатора
\ASYNCFREE	Компонент `AsyncFree`
PRILOZ1	
\ARIPHM	Модуль ARIPHM
PRILOZ4	Установка линии RTS в DOS
\RTSWIN	Установка линии RTS в Windows

Литература

1. Руководство программиста по Microsoft Windows 95/Пер. с англ. — М., Издательский отдел "Русская редакция" ТОО "Channel Trading Ltd.", 1997.
2. Фаронов В. В. Паскаль и Windows. — М.: Учебно-инженерный центр "МВТУ-ФЕСТО ДИДАКТИК", 1995.
3. Фаронов В. В. Система программирования Delphi в подлиннике. — СПб.: БХВ-Петербург, 2003.
4. Фленов М. В. "Библия Delphi". — СПб.: БХВ-Петербург, 2004.
5. Фленов М. В. "Программирование в Delphi глазами хакера". — СПб.: БХВ-Петербург, 2003.
6. Архангельский А. Я. Разработка прикладных программ для Windows в Delphi 5. — М.: БИНОМ, 1999.
7. Ханекамп Д., Вилькен П. Программирование под Windows/ Пер. с нем. — М.: ЭКОМ, 1996.
8. Мадрел Тео. Разработка пользовательского интерфейса/ Пер. с англ. — М.: ДМК, 2001.
9. Сайт "Мастера Delphi" (**http://www.delphimaster.ru/**).
10. Сайт "Delphirus" (**http://www.delphirus.com.ru**).
11. Сайт "Королевство Delphi" (**http://www.delphikingdom.info**).
12. Озеров В. Delphi. Советы программистов (2-е издание). — СПб.: Символ-Плюс, 2002.
13. Зубков С. В. Assembler для DOS, Windows и UNIX. — М.: ДМК Пресс, 2000.
14. Сайт "MSDN Library" — официальный сайт Microsoft для разработчиков (**http://msdn.microsoft.com/library/default.asp**).

15. Джоржейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT и AT / Пер. с англ. — М.: Финансы и статистика, 1992.
16. MSDN — Windows API. Персональный сайт Владимира Соковикова (**http://vsokovikov.narod.ru/**).
17. Kyle Marsh Хуки в Win32 / Перевод Олега Быкова (**http://www.rsdn.ru/article/baseserv/winhooks.xml**).
18. Win 32 API по шагам. Владимир Соковиков (**http://www.firststeps.ru/mfc/winapi/**).
19. Тенцер Анатолий. Пишем перехватчик клавиатуры (**http://www.realcoding.net/articles.php?article=182**).
20. Павлов Алексей. Hooks — аспекты реализации (**http://www.realcoding.net/article7.html**).
21. Фаронов В. Профессиональная работа в Delphi 6. — СПб.: Питер, 2002.
22. Фаронов В. В. Искусство создания компонентов Delphi. — СПб.: Питер, 2004.
23. Рощин И. Автоматическое определение кодировки текста — 2: Возвращаясь к напечатанному ("РМ. ВК" N 5-6/2002) // Радиомир. Ваш компьютер. — 2004. — № 6. — С. 35—37 (**http://ivr.webzone.ru/articles/defcod_2/**).
24. Delphi Russian Knowledge Base (**http://www.delphist.com/DRKB.zip**).
25. Ревич Ю. Звездные часы Xerox PARC (**http://www.computer-museum.ru/frgnhist/xeroxpcs.htm**).
26. Torry's Delphi Pages (**http://www.torry.net**).
27. GARMIN GPS Interface Specification. GARMIN Corporation, 12.02.04 (**http://www.garmin.com/support/pdf/iop_spec.pdf**).
28. Описание протокола NMEA-0183 версии 2.1 (**http://www.agp.ru/support/nmea/index.htm**).
29. Агуров П. Интерфейс USB. Практика использования и программирования. — СПб.: БХВ-Петербург, 2004.
30. Бобровский С. И. Delphi 7. Учебный курс. — СПб.: Питер, 2003.
31. Коммуникационные функции. Перевод Куковинец А. В. (**http://www.bcbdev.ru/msdn/hh/commun_6g37.htm**).
32. Allen Denever "Serial Communications in Win32" (**http://msdn.microsoft.com/library/default.asp?url=/library /en-us/dnfiles/html/msdn_serial.asp**).
33. Агуров П. Последовательные интерфейсы ПК. Практика программирования. — СПб.: БХВ-Петербург, 2004.

34. Статьи по применению микросхем FTDI (**http://www.efo.ru/cgi-bin/go?778**).
35. FT232BM Designers Guide Version 2.0 (**ftp://ftp.efo.ru/pub/ftdichip/Documents/dg232v20.pdf**).
36. Кокорева О. Реестр Windows XP. — СПб.: БХВ-Петербург, 2002.
37. Зайцев О. Защита программ от взлома (**http://www.delphirus.com.ru /modules.php?name=News&file=article&sid=36**).
38. Гофман В. Э., Хомоненко А. Д. Delphi 6. — СПб.: БХВ-Петербург, 2001.
39. Сысоев А. П. Строковые типы в Delphi. Особенности реализации и использования (**http://progers.ru/articles/137.htm**).
40. Утилита диагностики компьютера (выпускная работа) (**http://www.5ka.ru/67/15592/1.html**).

Предметный указатель

Б

К

П

С

Ч