Оглавление

Содержание

ЧАСТЬ II

Предисловие

В компании Borland я начал работать летом 1985 года. Мне было поручено принять участие в разработке новых программных инструментов, предназначенных для совершенствования процесса программирования (системы UCSD языка Pascal и обычных инструментов командной строки было уже недостаточно). Предполагалось, что достигнутое повышение эффективности труда позволит ускорить разработку программ и снизить нагрузку на программистов. В конечном счете, это даст возможность им (включая и меня) больше времени проводить с семьей и друзьями и вообще сделает их жизнь богаче и интересней. Появление пакета Turbo Pascal 1.0 навсегда изменило требования, предъявляемые к инструментам программирования. Это произошло в 1983 году.

Выход пакета Delphi еще раз изменил наше представление о процессе программирования. Среда Delphi 1.0 была разработана для поддержки объектно-ориентированного программирования, программирования в Windows и создания приложений для работы с базами данных. В последующих версиях Delphi была упрощена разработка приложений для Internet и добавлена поддержка создания распределенных приложений. Хотя Web-узел с описаниями основных свойств наших продуктов существует уже много лет и содержит тысячи страниц печатной документации и многие мегабайты данных интерактивной справочной системы, имеется много другой полезной информации, рекомендаций и советов, которые были бы полезны разработчикам для успешного завершения их проектов.

На создание Delphi 5 ушло шестнадцать лет. Не на подготовку этой книги, а на разработку самого продукта. "Шестнадцать лет?" — может спросить удивленный читатель. Да, именно так. Приблизительно шестнадцать лет назад, в ноябре 1983 года вышла первая версия пакета Turbo Pascal. При измерении времени в стандартах Internet, прошедшее с этого момента количество единиц времени может легко переполнить 64-разрядное целое число. Именно столько этих единиц потребовалось для появления Delphi 5.

Фактически, это — тринадцатая версия нашего компилятора. Если вы мне не верите, то достаточно просто запустить программу `dcc32.exe` из командной строки (имеется в виду командная строка DOS). Вы собственными глазами увидите текущий номер версии компилятора, а заодно и справочные данные о параметрах его командной строки. Для создания этого продукта потребовалось множество усилий со стороны инженеров, специалистов по тестированию, разработчиков документации, художников, поклонников нашего продукта, друзей и родственников. Кроме того, потребовалось найти и подготовить целую плеяду способных писать книги о Delphi людей.

Как вы думаете, что необходимо, чтобы написать руководство разработчика? Самый простой ответ: "Очень многое". Но как определить это "многое" точнее? Сделать такое определение совсем не просто, пожалуй, даже невозможно. Вместо точного определения я могу предложить только несколько отдельных положений, которые помогут его сформулировать. Своего рода, "рецепт изготовления" — если вы не возражаете.

Рецепт Дэви-Хакера для быстрого изготовления книги " Delphi 5. Руководство разработчика".

Состав:

- **Delphi 5 (издание "Standard", "Professional" или "Enterprise");**
- **два опытных профессиональных автора технической литературы (весом не менее 60 кг каждый);**
- **1000 столовых ложек слов;**

- **1000 чашек исходных текстов программ;**
- **одно-два десятилетия полезного опыта (включая несколько лет работы с Delphi);**
- **несколько горстей мудрости;**
- **многие часы попыток взлома программ;**
- **десятки недель отладки;**
- **неограниченное количество жидкости (я предпочитаю диетическую пепси-колу);**
- **сотни часов сна.**

Способ приготовления.

- **Прогрейте ваш компьютер до 110 вольт (до 220 вольт, если работа ведется за пределами США).**
- **Подогрейте авторов.**
- **С помощью собственной головы смешайте имеющуюся версию Delphi 5, весь приготовленный текст и все подготовленные фрагменты программ.**
- **Добавьте годы опыта, часы хакерских усилий, недели отладки, весь запас мудрости, после чего залейте все это необходимым количеством жидкости.**
- **Используйте весь запас часов сна для подсушки полученного продукта.**
- **Оставшиеся ингредиенты добавьте через некоторое время, когда блюдо остынет до комнатной температуры.**

Выход.

Один экземпляр "Delphi 5, Руководство разработчика", написанный Стивом Тейксейрой и Ксавье Пачеко.

Возможные замены.

Вместо предложенной жидкости можно использовать ваш любимый сорт газировки, сок, кофе и т.д.

А теперь отставим шутливый тон и вернемся к серьезным вещам. Я уже многие годы знаю Стива Тейксейру (некоторые называют его T-Rex — "Тиранозавр Rex") и Ксавье Пачеко (некоторые называют его просто X — "кси") как своих друзей, замечательных работников, непременных ораторов на всех ежегодных конференциях и давних членов сообщества пользователей продуктов фирмы Borland.

Предыдущее издание их "Руководства разработчика" было с энтузиазмом встречено пользователями Delphi всего мира. И вот теперь у вас в руках новейшая версия этого издания, способная доставить вам немало приятных часов увлекательного чтения.

Читайте, учитесь и развлекайтесь вместе с авторами. Любой из продуктов Delphi выпускался для того, чтобы доставить вам радость творчества, ведущего к успеху и достойному вознаграждению.

Девид Интерсимон (David Intersimone) — "Девид I".

Вице-президент по связям с разработчиками. Inprise Corporation

Посвящение

Анне.

Ксавье Пачеко

Хелен и Куперу.

Стив Тейксейра

Об авторах

Стив Тейксейра (Steve Teixeira) — является вице-президентом по разработке программного обеспечения компании DeVries Data Systems, Силикон Вэлли, Калифорния, США. Эта фирма специализируется на предоставлении консультаций по продуктам Borland/Inprise. Ранее Стив работал инженером-исследователем в области разработки программного обеспечения в корпорации Inprise и принимал участие в создании пакетов Borland Delphi и Borland C++ Builder. Он ведет одну из колонок в журнале Delphi Magazine, а также пользуется международным признанием как оратор, консультант и преподаватель. Стив с женой и сыном живет в Саратоге, штат Калифорния.

Ксавье Пачеко (Xavier Pacheco) — президент и главный консультант фирмы Xapware Technologies, Inc., Колорадо Спрингс, США. Эта фирма специализируется на предоставлении консультаций и обучении. Ксавье часто выступает на отраслевых конференциях и регулярно пишет статьи, посвященные Delphi, для периодических изданий. Он пользуется международным признанием как специалист и консультант по Delphi, а также является членом особой группы “TeamB” корпорации Borland, в задачи которой входит поддержка начинающих пользователей. Вместе со своей женой Анной и дочерью Амандой Ксавье живет в Колорадо Спрингс. Их общие любимцы — немецкие овчарки Роки и Шаста.

Благодарности

Мы благодарим всех, без чьего содействия эта книга никогда не была бы написана. Кроме того, просим никого не винить в ошибках, если таковые будут замечены в книге. В любом случае, все обнаруженные ошибки мы относим на свой счет.

Прежде всего, мы хотим поблагодарить наших технических рецензентов и хороших друзей Лансе Буллока (Lance Bullock), Криса Хесика (Chris Hesik) и Элли Петерс (Ellie Peters). Идеальный технический рецензент должен быть внимателен ко всем мелочам, и мы благодарны судьбе, позволившей нам встретиться со специалистами, отвечающими этим требованиям в самом широком смысле. В весьма сжатые сроки они выполнили огромный объем работы, и мы всегда будем им благодарны за потраченные усилия.

Мы выражаем свою исключительную признательность нашим соавторам, использовавшим весь свой опыт разработчиков программного обеспечения и писательское мастерство для создания тех фрагментов, без которых эта книга никогда бы не стала тем, чем она сейчас является. Дэн Мизер (Dan Miser), признанный специалист по MIDAS, внес свой неоценимый вклад, написав для этой книги главу 32, "Разработка приложений MIDAS". Окончательный вариант главы 27, "Разработка приложений CORBA в Delphi", был написан нами под пристальным контролем Лансе Буллока, нашего технического рецензента, — за что мы выносим ему двойную благодарность. Ник Ходжес (Nick Hodges), известный специалист по Web (и изобретатель `TSmily`), принял участие в подготовке и этого издания нашей книги. Он написал главу 31, "Компоненты WebBroker открывают двери в Internet".

Мы благодарны Девиду Интерсимону за то, что несмотря на исключительно напряженный график своей работы, он нашел время и написал для этой книги предисловие.

Во время написания книги мы получили множество советов и рекомендаций от своих друзей и сотрудников. В их числе Алан Лино Тадрос (Alan Lino Tadros), Роланд Бюшер (Roland Bouchereau), Шарль Калверт (Charlie Calvert), Джош Дэлби (Josh Dahlby), Девид Семпсон (David Sampson), Джесон Спренгер (Jason Sprenger), Скотт Фролич (Scott Frolich), Джефф Питерс (Jeff Peters), Грег де Вриес (Greg de Vries), Марк Дункан (Mark Duncan), Андерс Олсон (Anders Ohlsson), Девид Стривер (David Streever), Рич Джонс (Rich Jones) и многие другие — всех мы просто не в состоянии перечислить.

И, наконец, огромное спасибо всей бригаде, работавшей над книгой: Шелли Джонстону (Shelley Johnston), Гасу Миклосу (Gus Miklos), Дену Шерфу (Dan Scherf) и многим другим, которые скромно трудились за сценой, но без которых книга никогда не стала бы реальностью.

Благодарности от Ксавье

Я никогда не смогу в полной мере отблагодарить Господа за его заботу и благословения, высшее из которых Сын Его Иисус Христос — мой Спаситель. Я также благодарен Господу за мою жену Анну, чья любовь, терпение и понимание так нужны мне. Спасибо тебе, Анна, за твою поддержку и ободрение и, в особенности, за твои молитвы к нашему Отцу Всевышнему! Я благодарен моей дочери Аманде за ту радость, которую она приносит. Аманда, ты истинное благословение моей жизни!

Благодарности от Стива

Хочу выразить благодарность моей семье и, в особенности, Элен. Она постоянно напоминала мне, что является самым важным, и этим помогала в самые трудные моменты моей работы. Я благодарен и Куперу, за помощь в правильном понимании перспектив жизни.

Сообщите нам ваше мнение

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать электронное письмо или просто посетить наш Web-узел, оставив свои замечания — одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более подходящими для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш факс или номер телефона. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

E-mail:	info@williamspublishing.com
WWW	http://www.williamspublishing.com

Введение

Можете ли вы поверить, что прошло уже около пяти лет с тех пор, как мы начали работу над первым изданием "Delphi. Руководство разработчика". В то время мы были всего лишь разработчиками из отдела поддержки языков фирмы Borland, искавшими новые области применения созданных продуктов. Тогда же у нас родилась идея создания книги, содержащей широкий обзор методов написания программ и новейших технологий программирования, но без освещения всех тех вопросов, ответы на которые могут быть найдены в документации на продукт. Наш опыт работы по предоставлению технической поддержки разработчикам позволял надеяться, что мы сможем ответить даже на те их вопросы, которые они еще только собираются задать. Мы поделились этой идеей с сотрудниками Sams Publishing, и она показалась им привлекательной. И тогда начались изматывающие месяцы работы над рукописью, программирования и снова работы, и снова программирования долгими бессонными ночами. Я думаю, что несколько контрольных сроков оказались пропущенными именно из-за обилия примеров программирования. Но всему приходит конец, и в какой-то момент книга оказалась написанной.

Против наших скромных ожиданий, продажи книги показали: это именно то, что "доктор (в нашем случае, скорее, разработчик) прописал". Оценкой наших усилий стало признание читателями того факта, что это лучшая книга о Delphi.

Такая оценка не могла не вдохновить нас на дальнейшую работу, и нами были выпущены книги, посвященные Delphi 2, Delphi 3 и Delphi 4. Последняя из них вновь была признана читателями лучшей книгой года по Delphi. И, наконец, мы создали книгу, которую вы держите в руках. Надеемся, что по количеству и качеству излагаемого материала она превосходит все созданное нами ранее.

В настоящее время Стив является вице-президентом по разработке программ компании DeVries Data Systems. Эта компания специализируется на предоставлении консультаций по продуктам фирмы Borland и расположена в Силикон Вэлли, Калифорния. Ксавье основал собственную фирму XAPWARE Technologies, Inc., специализирующуюся на обучении и предоставлении консультаций по Delphi. Надеемся, что уникальная комбинация нашего опыта разработчиков и практиков и глубоких знаний продукта пошла книге на пользу.

Книга предназначена для тех, кто намерен разрабатывать приложения с помощью Delphi. Цель, которую мы ставили перед собой, — не просто рассказать, как создать приложения с помощью Delphi, но и объяснить, как это сделать *правильно*. Delphi — уникальный по своей мощи и возможностям инструмент разработки. Одновременно с резким сокращением времени разработки, он позволяет создавать приложения, производительность которых не уступает, а зачастую и превосходит показатели продуктов, разработанных с помощью существующих компиляторов C++. Книга расскажет вам, как взять лучшее из этих двух миров и как создать по-настоящему хороший, ясный и эффективный код.

Книга разбита на пять частей. Часть I, "Эффективность работы и ее составляющие", посвящена основам программирования в Delphi и Win32. Часть II, "Профессиональное программирование", построенная на фундаменте первой части и на примерах создания маленьких, но полезных приложений, поможет расширить ваши знания и умения. В части III, "Компонентно-ориентированная разработка", обсуждается разработка компонентов VCL и приложений с использованием технологии COM. Часть IV, "Работа с базами данных", посвящена работе с базами данных в Delphi — от локальных таблиц до баз данных SQL и многоуровневых решений. В части V, "Быстрая разработка приложений баз данных", весь изложенный материал собирается воедино для построения масштабного реального приложения. Общий объем этой книги весьма велик, поэтому части I и II были выделены в первый том, а все остальные части и приложения — во второй.

На кого рассчитана эта книга

Как следует из названия книги, она предназначена для разработчиков. Поэтому, если вы — разработчик программного обеспечения и используете Delphi, эта книга — для вас. Мы предполагали, что книга заинтересует три основные группы читателей.

- Разработчиков на Delphi, желающих повысить свой уровень программистов.
- Опытных программистов на Pascal, BASIC или C/C++, намеревающихся приступить к работе с Delphi.
- Программистов, работающих с Delphi, но желающих научиться пользоваться всеми его возможностями, а также возможностями, предоставляемыми Win32 API.

Соглашения, использованные в книге

В этой книге использованы следующие типографские соглашения.

- Код, команды, переменные, операторы, типы, вывод программ, любой текст, который вы можете увидеть на экране, даны моноширинным шрифтом.
- Вводимые вами данные также приведены с использованием моноширинного шрифта.
- Заменители в описаниях синтаксиса выделены *курсивным моноширинным* шрифтом. Заменители в реальных командах замещаются настоящими именами файлов, параметрами или другими элементами, которые они представляют.
- Курсив используется при первом появлении технических терминов в тексте, а также для выделения важных мест.
- Процедуры и функции указываются с помощью скобок () после их имени. Хотя это и не является стандартным синтаксисом Pascal, но помогает отделить функции и процедуры от свойств, переменных и типов.

В каждой главе встречается несколько замечаний, советов и предостережений, которые помогут вам обратить внимание на наиболее важные моменты в изложении материала и избежать ошибок при работе.

Все исходные тексты и файлы проектов могут быть найдены на компакт-диске, прилагаемом ко второму тому этой книги. При работе с компакт-диском взгляните на каталог \THRDPRTY, в котором представлено много полезных инструментов и компонентов сторонних производителей. Кроме того, исходные тексты и файлы проектов, представленных в главах, входящих в первый том, можно найти на Web-узле издательского дома "Вильямс" по адресу: www.williamspublishing.com.

Дополнения к книге

Дополнения, исправления и обновления к книге можно найти в Web, обратившись по адресу: http://www.xapware.com/ddg.

Итак, приступим

Нас иногда спрашивают, что движет нами, заставляя писать все новые и новые книги, посвященные Delphi? Это трудно объяснить. Ну хотя бы то, что, когда встречаешься с другим разработчиком Delphi и видишь у него в руках свою потрепанную книгу, понимаешь — жизнь потрачена не зря.

А теперь время отдохнуть и приступить к делу. И хотя мы не будем слишком спешить, вы и оглянуться не успеете, как станете настоящим гуру в Delphi!

Эффективность работы и ее составляющие

ЧАСТЬ I

Глава

1

Программирование для Windows в Delphi 5

В этой главе представлен беглый обзор Delphi, включая сведения по истории этого пакета, краткое описание его возможностей и способов взаимодействия со средой Windows, а также другую общую информацию, необходимую каждому разработчику. Здесь же обсуждаются основные особенности интегрированной среды разработки (IDE) Delphi 5 и приводятся сведения о некоторых недостаточно освещенных в документации возможностях, которые мало известны даже самым искушенным разработчикам. Эта глава не предназначена для обучения основам разработки программ в среде Delphi. Мы полагаем, что читатель приобрел эту книгу с целью узнать из нее что-то новое и интересное, а не просто освежить в памяти содержимое предоставляемой фирмой Borland документации. Исходя из этого, мы считаем своей задачей продемонстрировать наиболее мощные возможности данного продукта и пояснить, как их можно использовать для разработки программ коммерческого уровня. Мы надеемся, что чтение данной главы (и всей этой книги!) окажется полезным как опытным, так и начинающим разработчикам. В отношении последних необходимо заметить, что не следует начинать освоение Delphi с прочтения этой главы. Начните с изучения документации Borland и создания простейших примеров. Чтение этой главы принесет вам пользу лишь после уяснения основных механизмов работы IDE и общей процедуры разработки приложений.

Семейство продуктов Delphi 5

Delphi 5 поставляется в трех различных вариантах (изданиях): Delphi 5 Standard, Delphi 5 Professional и Delphi 5 Enterprise. Каждый из вариантов продукта рассчитан на разработчика определенного типа.

Delphi 5 Standard представляет собой самую простую версию. В ней имеется все необходимое для того, чтобы начать создание приложений в среде Delphi. Эта версия идеальна для любителей или студентов (кроме прочего, она самая дешевая). Версия включает:

- оптимизирующий 32-разрядный компилятор Object Pascal;
- библиотеку Visual Component Library (VCL), содержащую более 85 стандартных компонентов, размещаемых на панели Component Palette;
- поддержку пакетов, позволяющую создавать малые по размеру выполняемые файлы и библиотеки компонентов;
- объектно-ориентированную среду разработки (IDE), включающую редактор, конструктор форм и набор инструментов повышения производительности. Конструктор форм поддерживает функции наследования видимых форм и связывания;
- Delphi 1, предназначенный для создания 16-разрядных приложений Windows;
- полную поддержку Win32 API, включая COM, GDI, DirectX, многопоточность и различные Software Development Kit (SDK) фирмы Microsoft и сторонних производителей.

Версия **Delphi 5 Professional** рассчитана на профессиональных разработчиков, не использующих технологии клиент/сервер. Если вы — профессиональный разработчик, занимающийся созданием и поставкой приложений или компонентов Delphi, то этот продукт предназначен именно для вас. Помимо всего того, что входит в состав версии Standard, в версию Delphi 5 Professional включены:

- более 150 дополнительных компонентов в наборе Component Palette;
- средства поддержки работы с базами данных, включая соответствующие VCL-компоненты, низкоуровневый интерфейс доступа к базам данных Borland Database Engine (BDE) 5.0, BDE-драйверы для локальных таблиц, средства поддержки виртуаль-

ных наборов данных, позволяющие встраивать в компоненты VCL интерфейсы доступа к базам данных других типов, инструмент Database Explorer, компоненты InterBase Express среды InterBase, средства поддержки хранилищ данных и технологии ODBC;

- набор мастеров для создания COM-компонентов.. таких, например, как управляющие элементы ActiveX, ActiveForms, серверы автоматизации и страницы свойств;
- инструментарий QuickReports, предназначенный для интеграции в создаваемые приложения пользовательских отчетов;
- графические компоненты TeeChart для представления данных в виде графиков и диаграмм;
- однопользовательский сервер Local InterBase Server (LIBS), позволяющий разрабатывать SQL-приложения архитектуры клиент/сервер без подключения к сети;
- инструмент Web Deployment, позволяющий распространять содержимое компонентов ActiveX в среде Web;
- InstallSHIELD Express — инструментарий для распространения приложений;
- OpenTools API для разработки компонентов, интегрированных со средой Delphi 5, а также с интерфейсом для системы контроля версий PVCS;
- утилита WebBroker, набор мастеров FastNet Wizards и компоненты, предназначенные для разработки Internet-приложений;
- исходные тексты VCL, библиотеки времени выполнения (Runtime library, RTL) и редакторов свойств;
- WinSight32 — инструмент для просмотра информации об окнах и сообщениях.

Версия **Delphi 5 Enterprise** создана для использования большими фирмами и разработчиками корпоративных приложений клиент/сервер. Если вы разрабатываете приложения, взаимодействующие с SQL-серверами баз данных, в данной версии вы найдете все инструменты, необходимые для построения распределенных приложений. В этой версии содержатся все те компоненты, которые имеются в двух предыдущих, а также следующие:

- более 200 VCL-компонентов в наборе Component Palette;
- поддержка MIDAS (Multitier Distributed Application Service) и лицензия на использование в своих разработках этого продукта, обеспечивающего беспрецедентное упрощение процедуры создания многоуровневых приложений;
- поддержка CORBA, включая сервер VisiBroker ORB версии 3.32;
- компоненты InternetExpress XML;
- пакет TeamSource — программное обеспечение для управления групповой разработкой программ, допускающее одновременную работу с несколькими версиями продуктов (включает поддержку ZIP и PVCS);
- улучшенная поддержка Microsoft SQL Server 7;
- расширенная поддержка СУБД Oracle 8, включающая обработку полей абстрактного типа данных;
- прямая поддержка объектов ActiveX Data Objects (ADO);
- компоненты DecisionCube, позволяющие выполнять визуальный многомерный анализ данных (включая их источники);

- драйверы SQL Links BDE для доступа к базам данных СУБД InterBase, Oracle, Microsoft SQL Server, Sybase, Informix и DB2, а также лицензии на право неограниченного перераспределения этих драйверов;
- приложение SQL Database Explorer, позволяющее просматривать и редактировать специфичные для серверов метаданные;
- утилита SQL Builder — графический инструмент создания запросов;
- пакет SQL Monitor, обеспечивающий просмотр SQL-сообщений обмена информацией с сервером и облегчающий отладку и настройку производительности приложения;
- пакет Data Pump Expert, предназначенный для ускорения масштабирования систем;
- лицензия InterBase для Windows NT на пять пользователей.

Что такое Delphi

Часто нам задают вопросы, подобные следующим: "В чем главные достоинства Delphi?" или "Чем Delphi превосходит пакет X?" За прошедшие годы мы выработали два варианта ответов на эти вопросы — длинный и короткий. Короткий ответ заключается в одном слове — "продуктивность". Просто на сегодняшний день работа в Delphi является самым продуктивным методом создания приложений для Windows. Безусловно, имеются категории лиц (начальство и возможные клиенты), для которых этого ответа будет недостаточно. В этом случае мы даем длинный ответ, в котором подробно описывается то сочетание особенностей среды Delphi, которое делает ее столь продуктивной. Мы полагаем, что общая продуктивность любых инструментов создания программного обеспечения определяется следующими пятью важнейшими аспектами:

- качеством визуальной среды разработки;
- скоростью работы компилятора и быстродействием откомпилированных программ;
- мощностью языка программирования и его сложностью;
- гибкостью и масштабируемостью используемой архитектуры баз данных;
- наличием поддерживаемых средой разработки шаблонов проектирования и использования.

Безусловно, существует еще немало важных факторов — например, вопросы развертывания, документация, поддержка третьих фирм и т.д. Тем не менее, мы пришли к выводу, что этой упрощенной модели вполне достаточно для объяснения, почему имеет смысл остановить свой выбор на Delphi. Некоторые из упомянутых категорий связаны с определенной субъективностью оценки. Как же можно использовать их для оценки продуктивности определенного инструмента разработки? Предлагаемая схема проста. Оцените каждый из пяти показателей анализируемых пакетов по пятибалльной шкале и нанесите соответствующие точки на оси графика, представленного на рис. 1.1. Соедините точки для каждого из пакетов линиями — получится несколько пятиугольников. Чем больше площадь получившегося пятиугольника, тем выше продуктивность данного инструмента разработки.

Мы не имеем целью с помощью предложенной формулы навязать вам свое заранее подготовленное решение — выбор остается за вами! Мы просто предлагаем подробнее остановиться на каждом из аспектов вышеприведенной схемы и оценить соответствующие показатели Delphi в сравнении с другими инструментами разработки приложений для Windows.

Рис. 1.1. Схема построения диаграмм для оценки продуктивности инструментов разработки приложений

Визуальная среда разработки

Визуальная среда разработки, в общем случае, состоит из трех взаимосвязанных компонентов: редактора, отладчика и конструктора форм. В любом из современных инструментов ускоренной разработки приложений (Rapid Application Development — RAD) этих три компонента должны гармонично взаимодействовать друг с другом в процессе создания приложения. При работе в конструкторе форм Delphi неявно генерирует программный код всех тех компонентов, которые помещаются и обрабатываются в формах. В окне редактора в текст автоматически созданной программы можно внести необходимые дополнения, определяющие специфическое поведение данного приложения. Здесь же, в окне редактора, программы могут отлаживаться с помощью внесения точек останова, определения контролируемых переменных и т.д.

Редактор Delphi обычно используется параллельно с другими инструментами. Пожалуй, наиболее мощным из них можно считать утилиту CodeInsight, позволяющую существенно уменьшить объем вводимых с клавиатуры программных текстов. Этот инструмент построен на использовании информации компилятора, а не библиотеки типов (как в Visual Basic), поэтому область его применения значительно шире. Хотя редактор Delphi поддерживает достаточный набор параметров настройки, следует отметить, что возможности настройки редактора пакета Visual Studio несколько шире.

В Delphi 5 отладчик поддерживает уже весь набор функциональных возможностей, присущих отладчику пакета Visual Studio. К вновь добавленным функциям относятся средства удаленной отладки, подключения процессов, отладки пакетов и библиотек DLL, средства контроля значений автоматических локальных переменных и поддержки окна ЦП. Кроме того, Delphi предоставляет удобные средства управления графической средой отладки. Они позволяют в ходе отладки размещать и пристыковывать окна в любом удобном месте, а также запоминать сведения о полученной конфигурации в виде поименованной группы параметров настройки рабочего стола. Одна из очень удобных функций отладчиков, которая широко распространена в среде интерпретаторов (таких как Visual Basic или Java), заключается в воз-

можности изменять программный текст и, следовательно, поведение приложения непосредственно в процессе его отладки. К сожалению, в среде компиляторов реализация подобных функций связана с очень большими трудностями, и поэтому она отсутствует в Delphi.

Конструктор форм является обязательной принадлежностью всех RAD-инструментов, включая Delphi, Visual Basic, C++ Builder и PowerBuilder. Классический вариант среды разработки (например, Visual C++ и Borland C++) обычно включает редакторы диалогов, однако эти инструменты менее удобны для интеграции в рабочий поток создания приложения, чем конструкторы форм. Рассмотрев представленную на рис. 1.1 диаграмму, можно сделать вывод, что отсутствие конструктора форм оказывает заметное негативное влияние на общие показатели продуктивности конкретного инструмента разработки приложений. В последние годы Delphi и Visual Basic оказались втянутыми в постоянную борьбу за первенство в разнообразии возможностей их конструкторов форм. Каждая выпущенная версия этих продуктов просто изумляет своими новыми функциональными возможностями. Например, в конструкторе форм Delphi используется полностью объектно-ориентированная схема построения. В результате внесенные в базовые классы изменения немедленно распространяются и на любые производные от них классы. Этот механизм использован для реализации функции наследования визуальных форм (Visual Form Inheritance — VFI). VFI позволяет динамически порождать новые формы из любых других форм проекта или галереи форм. Более того, внесенные в базовую форму изменения будут немедленно распространены и унаследованы всеми ее формами-потомками. Более подробная информация об этой важной функции приводится в главе 4, “Строение приложения и концепции конструирования”.

Скорость работы компилятора и быстродействие откомпилированных программ

Быстрый компилятор позволяет разрабатывать программное обеспечение поэтапно, поскольку допускает многократное внесение в исходную программу небольших изменений, с последующей перекомпиляцией и тестированием. В результате возникает весьма эффективный цикл разработки. Более медленный компилятор вынуждает разработчика одновременно вносить больший объем изменений, комбинируя несколько отдельных доработок в одном цикле компиляции и отладки. Это, безусловно, снижает эффективность отдельных циклов разработки. Преимущества, достигаемые за счет повышенной эффективности работы откомпилированных программ, очевидны. В любом случае, чем быстрее работает программа и чем меньшее ее объектный код, тем лучше.

Вероятно, самое известное отличие используемого в Delphi компилятора языка Pascal состоит в его быстродействии. Фактически, это — самый быстрый компилятор языка высокого уровня из всех, существующих в среде Windows. В последние годы отмечаются заметные улучшения в работе компиляторов языка C++, который традиционно считался самым медленным в смысле скорости компилирования. Успехи были достигнуты за счет пошагового связывания и различных стратегий кэширования, используемых, в частности, в пакетах Visual C++ и C++ Builder. Тем не менее, даже эти улучшенные компиляторы языка C++ работают в несколько раз медленнее, чем компилятор Delphi.

Означает ли столь высокая скорость компилирования обязательное отставание в эффективности создаваемых программ? Безусловно, ответом на этот вопрос будет “Нет”. Создаваемый в Delphi объектный код имеет те же показатели эффективности, что и объектный код, созданный транслятором C++ Builder. Отсюда можно сделать вывод, что качество создаваемых программ соответствует уровню, обеспечиваемому очень хорошим компилятором языка

C++. По последним достоверным оценкам производительности, программы, созданные компилятором Visual C++, действительно имеют самые высокие показатели скорости выполнения и размеров текста. В основном это достигается за счет очень хорошей оптимизации. Хотя эти небольшие преимущества незаметны при разработке обычных приложений, они могут иметь очень большое значение при создании программ, выполняющих значительный объем вычислений.

Язык Visual Basic является особым в отношении используемой технологии компилирования. В ходе разработки приложения Visual Basic (VB) используется в интерпретирующем режиме и имеет достаточную скорость работы. При необходимости распространения созданного приложения можно воспользоваться VB-компилятором, результатом работы которого является выполняемый файл типа `.EXE`. Этот компилятор весьма медлителен и его показатели сильно отстают от возможностей инструментов C++ и Delphi.

Еще один интересный вариант представляет собой язык Java. Лучшие инструменты этой языковой среды (например, JBuilder и Visual J++) демонстрируют время компиляции, сравнимое с Delphi. Однако эффективность создаваемых программ чаще всего оставляет желать лучшего по той простой причине, что язык Java является интерпретируемым языком. Хотя развитие возможностей инструментов Java идет быстрыми темпами, скорость выполнения создаваемых на нем программ в большинстве случаев далеко уступает Delphi и C++.

Мощность языка программирования и его сложность

Мощность и сложность языка в значительной степени определяются точкой зрения собеседника, поэтому эти категории часто служат поводом для проведения многочисленных перепалок и ожесточенных дискуссий в группах новостей и списках рассылки. То, что совсем просто для одного человека, может оказаться весьма сложным для другого. В свою очередь то, что воспринимается одним человеком как ограничение, может расцениваться другим как самое изящное решение. Поэтому приведенные ниже рассуждения являются изложением точки зрения авторов и отражают их личный опыт и предпочтения.

Наиболее мощным из всех языков является ассемблер. Едва ли существует что-то такое, чего нельзя выполнить с его помощью. Однако написание даже самого простого приложения Windows на ассемблере является весьма сложным заданием, а полученный результат почти наверняка будет содержать ошибки. Мало того, чаще всего практически невозможно обеспечить сопровождение программ на ассемблере группой разработчиков на сколько-нибудь продолжительный период времени. По мере того как текст программ передается от одного исполнителя к другому, выбранные проектные решения и методы становятся все более и более туманными, и так до тех пор, пока текст программы не приобретает совершенно невразумительный вид, больше напоминающий священные тексты на санскрите, чем обычную компьютерную программу. Следовательно, в рассматриваемой категории ассемблеру следует присвоить очень низкую оценку, несмотря на всю его мощь. Главная причина — чрезмерная сложность использования этого языка для достижения тех целей, которые стоят перед большинством разработчиков приложений.

C++ также является чрезвычайно мощным языком. С помощью его действительно эффективных инструментов, подобных макросам препроцессора, шаблонам, перегрузке операторов, можно даже создать собственный язык в пределах C++. Если предоставленный разработчикам исключительно широкий набор функциональных возможностей будет использоваться продуманно, то это позволит создавать очень ясные и простые в сопровождении программы. Однако проблема состоит в том, что множество разработчиков не может противостоять искушению чрезмерного и неоправданного использования существующих возможностей, что

часто приводит к появлению просто ужасных программ. Фактически, на C++ писать плохие программы гораздо легче, чем хорошие, поскольку сам язык не ориентирует разработчика на использование хороших приемов программирования, оставляя эти вопросы полностью на его усмотрение.

Существует два языка, которые, по нашему мнению, очень схожи в том, что в них достигнут оптимальный баланс между сложностью и мощностью. Это — Object Pascal и Java. В обоих языках использован подход, предусматривающий ограничение доступных функциональных возможностей с целью переноса основных усилий разработчика на логическое проектирование приложений. Например, в обоих языках отсутствует "очень объектно-ориентированная", но способствующая различным злоупотреблениям концепция множественного наследования. В обоих случаях она заменяется реализацией в классах нескольких различных интерфейсов. Оба языка исключают изящную, но в то же время весьма опасную функцию перегрузки операторов. Кроме того, в обоих случаях исходные файлы рассматриваются как основные объекты языка, а не как компоненты, которые будут связаны в единое целое редактором связей. Более того, в обоих языках используются преимущества таких мощных функций, как обработка исключений, параметры времени выполнения и строковые ресурсы. Не случайно оба они были созданы не свободными комитетами, а отдельными разработчиками или небольшой их группой в пределах одной организации, имеющих единое мнение о том, что должен представлять собой создаваемый ими язык.

Visual Basic исходно создавался как язык, достаточно простой, чтобы начинающие программисты могли быстро его освоить. Однако по мере добавления в него новых функций, являвшихся ответом на неотложные требования времени, этот язык становился все более и более сложным. Несмотря на все усилия, прилагаемые с целью освобождения разработчиков от обременительных деталей, язык Visual Basic по-прежнему включает несколько серьезных ограничений, которые требуется тем или иным образом обходить при создании достаточно сложных приложений.

Гибкость и масштабируемость используемой архитектуры баз данных

В компании Borland отсутствует собственная линия продуктов управления базами данных. В состав Delphi входит инструментарий, который, по нашему мнению, обеспечивает самую гибкую архитектуру поддержки баз данных, по сравнению со всеми, представленными на рынке. Механизм BDE успешно работает и обеспечивает достаточную для большинства типов приложений производительность при взаимодействии с широким диапазоном локальных, распределенных и ODBC-платформ баз данных. Если вас это не устраивает, можно отказаться от средств BDE и воспользоваться специализированными ADO-компонентами. Если необходимые компоненты ADO отсутствуют, можно разработать собственные классы доступа к базам данных на основе предлагаемых компонентов баз данных многоуровневой абстрактной архитектуры или заказать необходимые компоненты у сторонней фирмы. Более того, предоставляемые средства MIDAS существенно упрощают распределение данных по нескольким логическим или физическим уровням и в то же время обеспечивают доступ к любым из существующих источников данных.

Следует отметить, что инструменты разработки Microsoft логически сфокусированы на поддержке собственных баз данных Microsoft и предоставляют соответствующие решения для доступа к их данным, включая средства ODBC, OLE DB и т.д.

Поддержка средой разработки шаблонов проектирования и использования

Это “чудодейственное снадобье” и “святой грааль” всех технологий разработки программного обеспечения, похоже, совершенно игнорируется другими инструментами разработчика. Хотя все элементы Delphi необходимы и важны, самым важным из них является, все-таки, библиотека VCL. Возможность манипулирования компонентами непосредственно в процессе проектирования, средства разработки собственных компонентов, наследующих элементы своего поведения от других компонентов с помощью различных объектно-ориентированных технологий — все это является важнейшими условиями высокого уровня продуктивности, свойственного среде Delphi. При разработке компонентов VCL всегда можно выбрать подходящую случаю технологию объектно-ориентированного проектирования из числа предоставляемых. Другие среды разработки, поддерживающие работу с компонентами, часто либо слишком жесткие, либо слишком сложные. Например, элементы управления ActiveX предоставляют практически те же самые возможности, что и компоненты VCL, однако создать новый класс, являющийся производным от элемента управления ActiveX, нельзя. Традиционные среды разработки, обеспечивающие работу с классами (например, OWL или MFC), обычно требуют от разработчика глубокого знания их внутренних механизмов. Только в этом случае работа в их среде может быть достаточно продуктивной. Всем им не достает определенных инструментов поддержки функций проектирования. Единственным инструментом, сравнимым по возможностям с библиотекой VCL Delphi, является библиотека MFC (Microsoft Foundation Classes) в среде Visual Java++. Однако на момент написания этой книги будущее пакета Visual Java++ было совершенно туманным — в связи с судебным иском, предъявленным компанией Sun Microsystems по вопросам, касающимся Java.

Немного истории

По своей сути, Delphi — это компилятор языка Pascal. Delphi 5 является очередным шагом в эволюции компиляторов Pascal, продолжающейся с тех времен, когда Андерс Хейлсберг (Anders Hejlsberg) создал первый компилятор Turbo Pascal. С тех пор программисты не устают восхищаться надежностью, изяществом и, конечно же, скоростью работы компиляторов Pascal фирмы Borland. Delphi 5 — не исключение. В нем воплощен более чем десятилетний опыт разработки компиляторов, превративший этот 32-разрядный оптимизирующий компилятор в настоящее произведение искусства. Хотя с ходом времени возможности компиляторов постоянно увеличивались, скорость его работы осталась практически неизменной. Более того, стабильность компилятора Delphi продолжает оставаться эталоном, с которым сравнивают все остальные инструменты разработки.

А теперь мы рассмотрим особенности каждой из выпущенных версий Delphi, а также кратко остановимся на характерных чертах исторического контекста, сложившегося на момент их выпуска.

Delphi 1

Во времена DOS, которые уже стали историей, программисты стояли перед нелегким выбором между продуктивным, но неэффективным BASIC и эффективным, но непродуктивным ассемблером. Появление компилятора Turbo Pascal, который сочетал простоту структурированного языка программирования с эффективностью настоящего компилятора, во многом разрешило

их проблемы. Программисты, работающие под Windows 3.1, оказались перед схожей проблемой: что выбрать — мощный, но сложный и требующий знаний C++ или простой, но крайне ограниченный Visual Basic? Delphi 1 предложил радикально новый подход к разработке приложений в среде Windows: простой язык, визуальная разработка приложений, создание откомпилированных выполняемых файлов, динамических библиотек, баз данных и многое другое. Delphi 1 был первым инструментом разработки Windows-приложений, объединившим в себе оптимизирующий компилятор, визуальную среду программирования и мощные возможности работы с базами данных. Все это вместе впоследствии получило название среды быстрой разработки приложений (Rapid Application Development — RAD).

Сочетание компилятора, RAD-инструментов и быстрого доступа к базам данных было совершенно неотразимым для множества разработчиков в среде Visual Basic, поэтому Delphi приобрел массу почитателей. Многие из разработчиков, работавшие с Turbo Pascal, перешли к работе в среде этого нового и привлекательного инструмента. Распространилось мнение, что Object Pascal — это уже не тот язык, с которым нас заставляли работать в колледже и который оставлял впечатление, что у работающего с ним связаны руки. Многие из разработчиков перешли к Delphi, чтобы воспользоваться преимуществами надежных элементов визуальной разработки, дополненных мощным языком и необходимыми инструментами. Visual Basic компании Microsoft явно проиграл соревнование с Delphi, к выходу которого разработчики Visual Basic оказались совершенно неподготовленными. Медленный, раздутый и ограниченный, Visual Basic 3 ничего не мог противопоставить Delphi 1.

Это был 1995 год. Компания Borland выплатила громадную компенсацию компании Lotus в связи с судебным иском по использованию элементов интерфейса приложения Lotus 1-2-3 в приложении Quattro. Кроме того, компания Borland подвергалась атакам со стороны Microsoft за то, что сделала попытку утвердиться в той области коммерческих приложений, которую Microsoft считала своей собственной. Чтобы разрядить ситуацию, Borland продает права на Quattro компании Novell, и нацеливает разработчиков СУБД dBASE и Paradox на удовлетворение нужд профессиональных разработчиков баз данных, а не на случайных пользователей-непрофессионалов. Пока Borland разбиралась с рынком своих приложений, Microsoft спокойно выравнивала положение дел на собственной платформе, имея целью привлечение к своим продуктам большей части тех разработчиков в среде Windows, которые уже использовали продукты Borland. Когда внимание Borland вновь сосредоточилось на вопросах конкурентной борьбы между приложениями, предназначенными для разработчиков, выяснилось, что она утратила часть рынка, ранее принадлежавшего Delphi и новой версии Borland C++.

Delphi 2

Годом позже в Delphi 2 было предложено все то же, но на новом уровне современной 32-разрядной операционной системы Windows 95 и Windows NT. Кроме того, Delphi 2 предоставил программисту 32-битовый компилятор, создававший более быстрые и эффективные приложения, мощные библиотеки объектов, улучшенную поддержку баз данных, поддержку OLE, средства Visual Form Inheritance, и при этом обеспечивал совместимость со старыми 16-битовыми приложениями. Delphi 2 стал тем мерилом, по которому равнялись другие RAD.

Это был 1996 год. Наиболее важный этап истории ОС Windows после выхода версии 3.0 — 32-разрядная ОС Windows 95, которая вышла в конце 1995 года. Borland твердо намеревалась сделать Delphi пакетом, превосходящим все остальные инструменты разработчика для этой платформы. Интересным историческим фактом является то, что Delphi 2 исходно получил название Delphi32 — это должно было подчеркнуть тот факт, что он создавался специально для 32-разрядной среды Windows. Однако перед самым выпуском название продукта было изменено на Delphi 2. Предполагалось, что это отметит тот факт, что Delphi 2 является самостоятельным законченным продуктом, а не просто вариантом Delphi 1 для новой платформы.

Microsoft сделала попытку ответить на вызов, выпустив Visual Basic 4, однако он обладал низкой производительностью, не обеспечивал совместимость 16- и 32-разрядных приложений, а также имел несколько других заметных недостатков. Тем не менее, впечатляющее количество разработчиков по тем или иным причинам продолжало использовать Visual Basic. Помимо прочего, компания Borland хотела, чтобы Delphi вышел на рынок высокопроизводительных приложений среды клиент-сервер, занятый такими приложениями, как PowerBilder. Однако данная версия не обладала необходимой мощностью, чтобы сколько-нибудь заметным образом потеснить те продукты, которые полностью захватили корпоративный сектор рынка.

К тому времени компания Borland вынуждена была сфокусировать свои интересы на корпоративных пользователях. Это решение во многом было продиктовано сокращением рынка dBASE и Paradox, а также уменьшением доходов, поступающих от продуктов C++. В целях подготовки к такому сложному шагу, Borland допустила ошибку, заключавшуюся в приобретении компании Open Environment Corporation, которая специализировалась на программном обеспечении среднего уровня и выпустила два продукта. Один из них стал предшественником CORBA, а другой, поддерживавший распределенную технологию OLE, был впоследствии вытеснен DCOM.

Delphi 3

В процессе создания Delphi 1, команда разработчиков Delphi была занята преимущественно проектированием и реализацией основных инструментов среды разработки. При создании Delphi 2 основная работа состояла в переходе на 32-разрадную платформу (с одновременным сохранением практически полной обратной совместимости). Кроме того, создавались новые средства поддержки баз данных с архитектурой клиент-сервер, необходимые для выхода на корпоративный рынок. В процессе работы над Delphi 3 команде разработчиков было поручено расширить набор инструментов для того, чтобы обеспечить самые широкие возможности выбора решений тех проблем, с которыми постоянно сталкивались разработчики в среде Windows. В частности, в Delphi 3 было существенно упрощено использование таких сложных технологий, как COM и ActiveX, добавлены средства разработки приложений для World Wide Web, включены средства создания тонких клиентов приложений и поддержка баз данных с многоуровневой архитектурой. Модернизация инструмента Code Insight позволила упростить процесс написания программ, хотя в остальной части используемые для создания приложений технологии остались теми же, что и в Delphi 1.

Это был 1997 год, и конкурентная борьба приняла особенно напряженную форму. На рынок простых приложений Microsoft наконец выпустила достойный внимания продукт — Visual Basic 5. Он включал долгожданный компилятор, призванный разрешить проблемы низкой производительности, отличные средства поддержки технологии COM/ActiveX и еще некоторые важные улучшения. На рынке корпоративных приложений Delphi удалось успешно потеснить такие пакеты, как PowerBilder и Forte.

В ходе создания Delphi 3 команда разработчиков потеряла своего главного члена. Андерс Хейлсберг, ведущий специалист и главный архитектор Delphi, принял решение оставить свой пост и перейти на работу в корпорацию Microsoft. Однако это не нарушило хода работ, поскольку освободившееся место занял Чак Яджевски, долгое время тесно работавший с прежним ведущим специалистом. Из корпорации также ушел глава службы технической поддержки Поль Гросс, однако это больше сказалось на пользователях продуктов, чем на ходе разработки нового программного обеспечения.

Delphi 4

Главная задача Delphi 4 состояла в упрощении процедуры разработки приложений. Новая утилита Module Explorer позволила просматривать и редактировать модули с помощью удобного графического интерфейса. Новые средства навигации в программах и используемых

классах позволяли вести работу над текстом создаваемого приложения с минимальными усилиями. Визуальная среда разработки была перепроектирована и дополнена возможностью пристыковывать панели инструментов и окна, делая процесс разработки более удобным. Существенные улучшения были внесены и в программу отладчика. Возможности Delphi 4 были расширены средствами поддержки корпоративных многопользовательских решений за счет таких современных технологий, как MIDAS, DCOM и CORBA.

Это был 1998 год, и Delphi удавалось эффективно защищать свои позиции на рынке. Общая обстановка стабилизировалась, однако Delphi продолжал медленно расширять свой сектор рынка. Самым модным решением в это время была технология CORBA, и Delphi включал ее поддержку, а его "конкуренты" — нет. Кроме того, некоторый успех отмечался и на рынке простых приложений. Заслужив славу самого стабильного инструмента разработки на рынке, Delphi 4 завоевал хорошую репутацию у постоянных пользователей Delphi, которые не желали отказываться от предоставленных им высококачественных решений и стабильности.

Выпуск Delphi 4 последовал за приобретением компании Visigenic — одного из лидеров в области технологии CORBA. Компания Borland, теперь носившая название Inprise, присвоенное ей для упрощения вхождения на корпоративный рынок, фактически стала лидером в этом секторе рынка — в основном за счет интеграции своих инструментов со средствами технологии CORBA. Для окончательной победы требовалось обеспечить ту же простоту использования средств поддержки CORBA, которая была достигнута в предыдущих версиях продуктов Borland в отношении поддержки COM и средств разработки приложений Internet. Однако по разным причинам необходимой степени интеграции достичь не удалось, и интегрированные средства поддержки CORBA составили лишь незначительную часть общих функциональных возможностей среды разработки.

Delphi 5

В Delphi 5 дальнейшее развитие продукта происходило сразу по нескольким направлениям. Во-первых, в Delphi 5 была продолжена основная линия улучшений, начатая в Delphi 4. В пакет были добавлены новые функции, упрощающие выполнение задач, обычно связанных с большими затратами времени. Это позволяет разработчикам больше сосредотачиваться на том, *что* они хотят написать, а не том, *как* это можно выполнить. В число новых функций входят различные улучшения графической среды разработки и отладчика, пакет поддержки групповой разработки программ TeamSource и инструменты трансляции. Во-вторых, в Delphi 5 включен набор новых функций, предназначенных для упрощения разработки приложений для Internet. Сюда относится мастер объектов Active Server, предназначенный для создания ASP, компоненты InternetExpress, обеспечивающие поддержку XML, и новые функции MIDAS, обеспечивающие весьма гибкую платформу размещения данных в среде Internet. Наконец, разработчики затратили немало усилий на обеспечение самого важного из показателей Delphi 5 — стабильности его работы. Как и при изготовлении лучших вин, при создании высококачественного программного обеспечения не должно быть излишней поспешности. Поэтому Delphi 5 выпустили в свет только после того, как он был окончательно готов.

Выход Delphi 5 в свет состоялся во второй половине 1999 года. Delphi продолжает расширять свое проникновение на корпоративный рынок, а в секторе малых приложений он по-прежнему конкурирует с Visual Basic. В целом, положение остается более-менее стабильным. Компания Inprise приняла решение вернуть свое прежнее название — Borland, что было с радостью воспринято давними почитателями ее продуктов. Кроме того, была проведена определенная реорганизация, основное назначение которой — гарантировать сохранение высокого качества выпускаемых программных продуктов, свойственного прежним разработкам фирмы Borland.

Что дальше?

Хотя история продукта представляет определенный интерес, более важным является то, что ожидает Delphi в будущем. Исходя из исторической перспективы, можно с высокой степенью вероятности предположить, что еще достаточно долго Delphi будет представлять собой прекрасный инструмент разработки приложений в среде Windows. Однако интереснее было бы узнать, появится ли когда-либо версия Delphi для платформ, отличных от Win32. На основании поступающих из компании Borland сведений можно заключить, что этому вопросу уделяется серьезное внимание. В 1998 году на конференции пользователей Borland ведущий специалист Delphi Чак Яджевски продемонстрировал компилятор Delphi, который генерировал байт-код языка Java. Теоретически, подобные файлы могут выполняться на любом компьютере с установленной виртуальной машиной Java. Хотя использование подобной технологии связано с преодолением множества технических препятствий, можно надеяться, что функция "Delphi for Java" будет когда-либо реализована в продукте. На недавней конференции пользователей Borland в 1999 году было объявлено, что планируется развернуть работы по выпуску версии Delphi для платформы Linux.

Интегрированная среда разработки Delphi 5

С целью подтверждения факта преемственности используемой терминологии, на рис 1.2 показан общий вид интегрированной среды разработки (IDE) Delphi 5. На этом рисунке отмечены все основные компоненты среды разработки: главное окно, палитра компонентов, панели инструментов, окно конструктора форм, окно редактора кода, окно инспектора объектов и окно Code Explorer.

Главное окно

Главное окно можно представить как управляющий центр Delphi 5 IDE. В нем содержатся все стандартные элементы и сохранена функциональность других окон Windows. Оно состоит из трех основных частей: главного меню, панелей инструментов и палитры компонентов (Component Palette).

Главное меню

Как и в любой программе Windows, к меню обращаются при необходимости открыть, сохранить или создать новый файл, вызвать мастер, перейти в другое окно, изменить параметры настройки и т.д. Каждый элемент главного меню может быть продублирован соответствующей кнопкой на панели инструментов.

Панели инструментов Delphi

Панели инструментов предоставляют доступ к различным функциям главного меню IDE с помощью единственного щелчка на соответствующей кнопке. Обратите внимание, что для каждой кнопки панели инструментов предусмотрен вывод подсказки, содержащей описание ее назначения. Не считая палитры компонентов, в IDE Delphi имеется пять отдельных панелей инструментов: Debug (Отладка), Desktop (Рабочий стол), Standard (Стандартная), View (Вид) и Custom (Пользовательская). На рис. 1.2 показана конфигурация кнопок этих панелей,

принимаемая по умолчанию. Однако любую из кнопок можно удалить или добавить с помощью команды View⇨Toolbars⇨Customize. На рис. 1.3 показано диалоговое окно Customize, предназначенное для настройки панелей инструментов. Добавление новых кнопок осуществляется путем их перетаскивания из этого окна на любую панель инструментов. Для удаления кнопки достаточно перетащить ее за пределы панели инструментов.

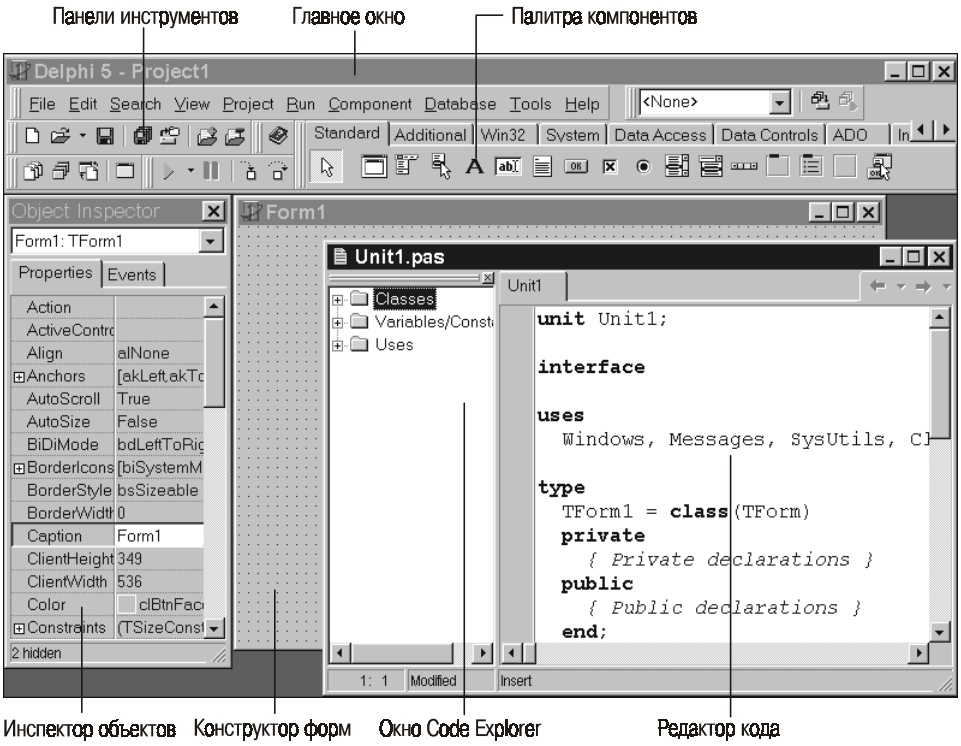


Рис. 1.2. Общий вид интегрированной среды разработки (IDE) Delphi 5

Возможности настройки панелей инструментов не ограничиваются лишь заданием отображаемых на них кнопок. Любую из панелей инструментов, палитру компонентов и панель меню можно перемещать в пределах главного окна IDE. Для этого достаточно поместить указатель мыши на вертикальную серую полоску в левой части панели и перетащить ее в требуемое положение в окне. Если перетащить панель за пределы главного окна, будет использована еще одна из возможностей настройки — панель инструментов отстыкуется от главного окна и станет плавающей, т.е. расположенной в собственном независимом окне. Вид плавающих панелей инструментов показан на рис. 1.4.

Палитра компонентов

Палитра компонентов представляет собой панель инструментов удвоенной высоты, содержащей несколько вкладок, в которых находятся все установленные в среде IDE компоненты VCL и ActiveX. Порядок следования и вид вкладок и компонентов может быть настроен с помощью щелчка правой кнопкой мыши на интересующем объекте или посредством выбора в главном меню команды Component⇨Configure Palette.

Конструктор форм Form Designer

При запуске конструктор форм представляет собой пустое окно, готовое к превращению в окно приложения Windows. Его можно рассматривать как холст художника, предназначенный для создания интерфейса будущего приложения — здесь определяется, как оно будет выглядеть с точки зрения пользователя. Процесс создания заключается в выборе компонентов на палитре и перетаскивании их в форму. Точное размещение и установка размеров компонентов также выполняются с помощью мыши. Кроме того, внешним видом и поведением компонентов можно управлять из окон Object Inspector и Code Editor.

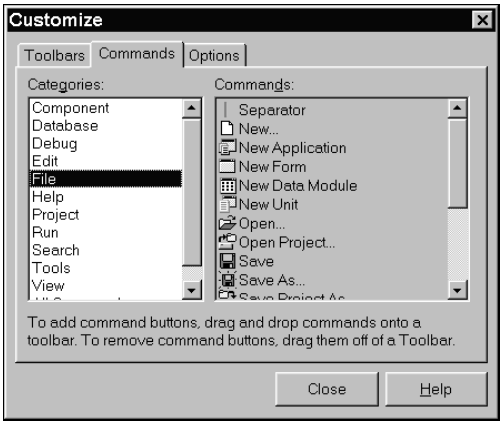


Рис. 1.3. Диалоговое окно Customize предназначено для настройки панелей инструментов

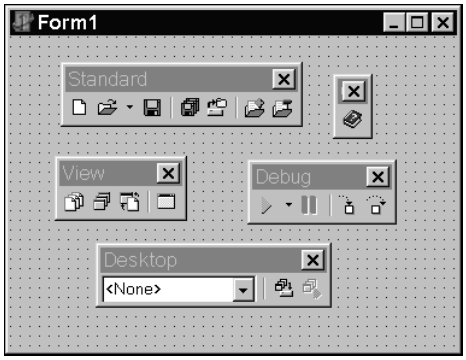


Рис. 1.4. На этом рисунке все панели инструментов представлены в плавающем виде

Инспектор объектов Object Inspector

В окне Object Inspector можно изменять свойства компонентов формы или определять события, на которые будет реагировать эта форма или ее компоненты. *Свойства* (*properties*) представляют собой данные, определяющие, как объект выглядит на экране, — высоту, цвет, шрифт и т.д. *События* (*events*) — это подпрограммы, выполняемые в ответ на некоторые происходящие в вашем приложении действия. Примером события может служить поступление сообщения от мыши или передача сообщения окну с требованием его перерисовки. В окне Object Inspector для переключения между работой с событиями и работой со свойствами используется стандартная технология вкладок — для перехода в ту или иную вкладку достаточно щелкнуть на ее корешке. Инспектор показывает события и свойства, относящиеся к той форме или компоненту, который активен в конструкторе форм в настоящее время.

Новой функцией Delphi 5 является возможность упорядочивать содержимое окна Object Inspector либо по категории, либо по именам (в алфавитном порядке). Для этого достаточно щелкнуть правой кнопкой мыши в любом месте окна Object Inspector и выбрать в раскрывшемся контекстном меню команду Arrange. На рис. 1.5 показаны два расположенных рядом окна Object Inspector. В левом окне объекты упорядочены по категории, а в правом — по именам. Кроме того, с помощью команды View этого же контекстного меню можно определить, какие именно категории объектов вас интересуют в данный момент.

Инспектор объектов тесно связан с контекстной справочной системой. Если особенности какого-то события или свойства вам непонятны, поместите на него указатель мыши и нажмите клавишу <F1>. На экране раскроется окно справочной системы с соответствующим текстом.

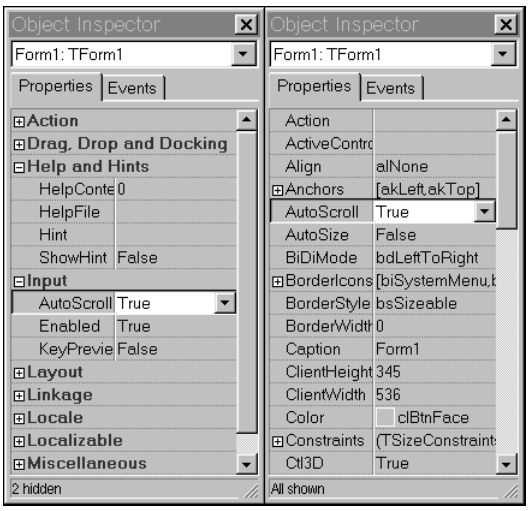


Рис. 1.5. Вывод содержимого окна Object Inspector с упорядочиванием по категории и по имени

Редактор кода Code Editor

Окно редактора текстов программ Code Editor предназначено для ввода команд, определяющих поведение создаваемой программы. Здесь же отображается текст, сгенерированный Delphi для создания компонентов разработанной формы. Окно Code Editor использует технологию вкладок, причем отдельная вкладка создается для каждого модуля или файла. При каждом добавлении в приложение новой формы создается новый модуль, а в окно Code Editor добавляется соответствующая вкладка. Контекстное меню окна Code Editor предоставляет широкий диапазон команд редактирования, включая команды работы с файлами, создания закладок и поиска символов.

Можно работать сразу с несколькими окнами Code Editor. Для открытия нового окна редактора кода следует выбрать в главном меню команду View⇨New Edit Window.

Окно Code Explorer

В окне Code Explorer модули, представленные во вкладках окна Code Editor, можно просматривать в виде древовидной структуры. Подобное представление позволяет легче ориентироваться в модулях, а также добавлять новые или переименовывать уже имеющиеся элементы модулей. Очень важно помнить, что между окнами Code Explorer и Code Editor всегда поддерживается связь типа “один к одному”. Щелчок правой кнопкой мыши на любом из элементов в окне Code Explorer позволяет вывести контекстное меню с командами, доступными этому объекту. Кроме того, можно управлять сортировкой и фильтрацией отображаемых в окне Code Explorer объектов. Для этой цели используются параметры, расположенные во вкладке Explorer диалогового окна Environment Options.

Генератор исходного кода

При работе с визуальными компонентами в конструкторе форм Delphi автоматически генерирует соответствующий код на языке Object Pascal. Простейший путь познакомиться с этой особенностью Delphi — начать новый проект. Выберите в главном окне команду File⇨New Application. В результате в конструкторе форм будет создана новая форма, а в редакторе кода — заготовка исходного текста, представленная в листинге 1.1.

Листинг 1.1. Заготовка исходного текста программы

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
  private
    {  Private declarations }
  public
    {  Public declarations }
  end;

var
  Form1: TForm1;

implementation

{ $R *.DFM}

end.
```

Обратите внимание на то, что исходный код, ассоциированный с некоторой формой, всегда сохраняется в отдельном модуле. Однако, хотя каждая форма имеет собственный модуль, не каждый модуль имеет собственную форму. (Если вы плохо знакомы с языком Pascal и используемой в нем концепцией модульности, обратитесь за пояснениями к главе 2, "Язык программирования Object Pascal". В данной главе содержится краткое описание языка Object Pascal, предназначенное для тех, кто перешел к Delphi после работы с C++, Visual Basic, Java и другими языками программирования.)

Рассмотрим код заготовки более подробно. Ниже приведена часть ее текста.

```
type
  TForm1 = class(TForm)
  private
  {  Private declarations }
  public
  {  Public declarations }
  end;
```

Этот фрагмент указывает, что объект TForm1 порожден от объекта TForm. Тут же комментариями отмечены места, предназначенные для ввода собственного кода — как доступного извне (public), так и скрытого (private) от внешних программ. Не беспокойтесь, если вы пока не знаете точного значения терминов "объект", "скрытый" и т.д. Подробно об этих и других понятиях языка Object Pascal рассказывается в главе 2, "Язык программирования Object Pascal".

Очень важна следующая строка программы:

```
{$R *.DFM}
```

В Object Pascal директива $R используется для загрузки внешнего файла ресурсов. Приведенная строка связывает файл с расширением .dfm (расширение файлов форм Delphi) с выполняемым модулем. Файл .dfm содержит бинарное представление созданной с помощью Form Designer формы. Символ шаблона "*" в данном случае означает, что имя файла должно быть тем же, что и имя модуля. В нашем примере имя модуля определено как Unit1, следовательно, его исходный текст будет находиться в файле Unit1.pas, а значение *.DFM в директиве соответствует файлу Unit1.dfm.

На заметку

В IDE Delphi 5 появилась новая функция — возможность сохранения создаваемых файлов DFM в текстовом, а не двоичном виде. Именно этот режим теперь устанавливается по умолчанию. Однако его можно отменить, для чего следует сбросить флажок опции New forms as text во вкладке Preferences диалогового окна Environment Options. Хотя сохранение форм в текстовом формате менее эффективно с точки зрения размера создаваемых файлов, данный вариант следует считать очень удобным сразу по нескольким причинам. Во-первых, это позволяет очень легко вносить незначительные изменения в текст описания формы в окне любого текстового редактора. Во-вторых, если файл по какой-либо причине был поврежден, восстановить содержимое текстового файла гораздо проще, чем двоичного. Не забывайте, что предыдущие версии Delphi работают только с двоичными файлами .DFM, поэтому этот режим следует отменить, если создаваемый проект будет обрабатываться и в других версиях Delphi.

Файл проекта приложения также заслуживает хотя бы беглого взгляда. Расширение файла проекта — .dpr (от **D**elphi **PR**oject). Он представляет собой обычный исходный файл Pascal, но с некоторыми расширениями. В этом файле содержится основная часть (с точки зрения Object Pascal) вашей программы. В отличие от других версий Pascal, которые, возможно, вам знакомы, основная работа программ Delphi проходит в модулях, а не в главном файле программы. Текст файла проекта можно отобразить в окне редактора с помощью команды Project⇨View Source. Вот текст файла нашего нового проекта:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' { Form1} ;

{ $R *.RES}

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

При добавлении модулей и форм в приложение они будут появляться в описании uses файла проекта. Обратите внимание, что в описании uses сгенерированный код содержит указания (в виде комментария) того, какая форма находится в подключаемом модуле. Если у вас когда-либо возникнут сомнения, в каком из модулей содержится та или иная форма, их легко можно будет развеять, воспользовавшись окном Project Manager, которое выводится на экран с помощью команды View⇨Project Manager.

На заметку Каждая форма связана с собственным модулем, кроме того, в программу могут входить модули, содержащие только код и не связанные с какой-либо формой. В Delphi основная работа осуществляется в модулях, и вряд ли вам когда-либо придется непосредственно работать с файлом проекта .dpr.

Создание простейшего приложения

Простейшее действие, заключающееся в перетаскивании некоторого компонента (например, кнопки) в форму, вызывает генерацию дополнительного кода, необходимого для создания вставленного в форму элемента:

```
type:
  TForm1 = class(TForm)
    Button1: TButton;
  private
    {  Private declarations }
  public
    {  Public declarations }
  end;
```

Как видите, помещенная в форму кнопка представляет собой экземпляр объекта TButton в форме TForm1. Напомним, что при обращении к этой кнопке за пределами модуля данной формы имя экземпляра этой формы потребуется указывать в качестве области видимости: Form1.Button1. Подробно об областях видимости рассказывается в главе 2, "Язык программирования Object Pascal".

Когда кнопка выбрана в конструкторе форм, ее функциональность можно изменить в окне Object Inspector. Предположим, что требуется установить ее исходную ширину равной 100 пикселям и обеспечить, чтобы во время работы программы щелчок на кнопке приводил к удвоению ее высоты. Для изменения ширины кнопки выберите в окне Object Browser свойство Width и установите его значение равным 100. Внесенное изменение вступит в силу и будет отображено в конструкторе форм только после нажатия клавиши <Enter> или выхода из свойства Width. Для того чтобы обеспечить реакцию кнопки на щелчок, в окне Object Inspector перейдите во вкладку Events. В ней содержится список событий, на которые может реагировать объект. Дважды щелкните на пункте списка OnClick, при этом Delphi сгенерирует заготовку процедуры, которая будет вызываться по щелчку на данной кнопке, и перенесет фокус ввода в окно с ее текстом (в нашем случае это процедура TForm1.Button1Click()). Все, что осталось сделать — это ввести в заготовку оператор удвоения высоты кнопки:

```
Button1.Height := Button1.Height * 2;
```

Для того чтобы проверить, как работает созданная программа, нажмите клавишу <F9> и оцените полученные результаты.

На заметку

Delphi постоянно поддерживает целостность связей между сгенерированными процедурами и теми управляющими элементами, которым они соответствуют. При компиляции или сохранении исходного текста модуля Delphi просматривает код и удаляет все пустые заготовки процедур. Это означает, что, если вы не внесете ни одной строки кода в заготовку процедуры `TForm1.Button1Click()`, Delphi полностью удалит ее из модуля. Основной вывод, который следует из сказанного, — никогда не удаляйте сгенерированные Delphi заготовки процедур самостоятельно. Если процедура вам не нужна, просто удалите ее тело, после чего Delphi удалит оболочку процедуры собственными средствами.

После того как кнопка в созданной вами форме примет достаточно большие размеры, завершите работу программы и вернитесь в среду IDE Delphi. Следует заметить, что обеспечить реакцию кнопки на щелчок мышью можно сразу же после помещения этого управляющего элемента в форму. Двойной щелчок мышью на любом компоненте формы автоматически вызывает раскрытие окна редактора с текстами подпрограмм, связанных с данным компонентом. В большинстве случаев будет автоматически сгенерирована заготовка текста подпрограммы того события, которое первым упоминается в списке событий окна Object Inspector.

События и сообщения

Если вам уже приходилось создавать приложения Windows традиционными методами, то вы, безусловно, сможете оценить простоту использования событий Delphi. Исключается всякая необходимость организации перехвата сообщений Windows, их расшифровки, анализа и т.д. Если же вы не знаете, о чем идет речь, обратитесь за помощью к главе 5, “Сообщения Windows”.

Как правило, события Delphi запускаются сообщениями Windows. Так, например, событие `OnMouseDown` объекта `TButton` фактически инкапсулирует сообщения Windows `WM_xBUTTONDOWN`. Обратите внимание на то, что событие выполняет предварительную обработку сообщения и предоставляет вам информацию о том, какая из кнопок мыши была нажата и где находился курсор мыши, когда это произошло. Подобную информацию — но уже о нажатии клавиши — предоставляет событие `OnKeyDown`. Вот код сгенерированной Delphi заготовки для обработки этого события:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
  begin
  end;
```

Вся необходимая для написания обработчика события информация — уже в ваших руках. Если вы имеете опыт программирования в Windows, то сможете оценить, насколько эта форма удобнее самостоятельного разбора параметров сообщения `LParams` или `WParams`. Вся эта работа делается вместо вас, и вам требуется только воспользоваться ее результатами, которые значительно понятнее, чем стандартные параметры сообщений Windows. Сведения о том, как в Delphi функционирует внутренняя служба обработки сообщений, будут приведены в главе 5, “Сообщения Windows”.

Необязательность программирования

Одним из самых значительных (хотя и спорных) достоинств системы событий Delphi (по сравнению со стандартной системой обработки Windows) является то, что вы не обязаны программировать обработчики событий. В отличие от обработки стандартных сообщений

Windows, не требуется ни вызывать обработчик по умолчанию, ни передавать какую-либо информацию назад в Windows после обработки сообщения.

Обратная сторона медали заключается в том, что нельзя сделать больше того, что вам позволено. Вы находитесь полностью во власти того, кто спроектировал это событие, поскольку именно этот человек определил тот уровень контроля, который будет предоставлен вам при обработке данного события в приложении. Так, в обработчике `OnKeyPress` можно изменить или просто удалить информацию о нажатых клавишах, в то время как обработчик `OnResize` только извещает вас о происшедшем событии, не позволяя ни предотвратить его, ни повлиять на устанавливаемые размеры объекта.

Тем не менее существует возможность сделать все, что потребуется, работая с сообщениями Windows не через события, а непосредственно. Естественно, знания и уровень программирования при этом должны быть выше, но ведь и ваша ответственность за работоспособность системы в этом случае возрастает. Полную власть над сообщениями Windows можно получить с помощью ключевого слова `message` (подробнее об этом рассказывается в главе 5, "Сообщения Windows").

В этом — весь Delphi. Он универсален и предоставляет каждому свое, обеспечивая простоту и ясность при высокоуровневом программировании и оставляя, в случае необходимости, возможность эффективного и нестандартного низкоуровневого программирования.

Упрощение разработки прототипов

Поработав с Delphi некоторое время, можно заметить, что предъявляемые этим пакетом требования к уровню знаний достаточно размыты. Создание в Delphi даже самого первого приложения приносит новичкам заметные дивиденды. Это выражается в сокращении времени разработки и получении надежных устойчивых продуктов. Delphi обладает просто неограниченными возможностями в том аспекте разработки приложений, который доставлял программистам в среде Windows больше всего неприятностей. Речь идет о создании интерфейса пользователя (User Interface — UI).

Иногда процедуру разработки интерфейса пользователя и общего макета окна программы называют "созданием прототипа". В невизуальной среде программирования создание прототипа приложения часто требует существенно бо́льших усилий, чем создание собственно рабочей части программы или ее "основного плана". Тем не менее именно реализация основного плана программы является главной целью ее создания. Безусловно, создание привлекательного и интуитивно понятного интерфейса пользователя также является важнейшим условием успеха. Однако, кому будет нужна коммуникационная программа с превосходно выполненными диалоговыми окнами и удобным интерфейсом, но не способная выполнить посылку данных через модем? Программы во многом подобны людям — приятная внешность всегда располагает, однако этого мало для того, чтобы данный человек занял сколько-нибудь заметное место в нашей жизни.

Delphi позволяет создавать программу, вкладывая максимум сил и умения в ее рабочую часть, сократив до минимума затраты времени на реализацию пользовательского интерфейса. Потратив некоторое время на обучение работе с Delphi, вы сможете легко и просто разрабатывать пользовательские интерфейсы, не сравнимые ни с какими другими, созданными с помощью традиционного инструментария. Более того, они будут не просто элегантны, но и оригинальны, т.е. будут обладать "изюминкой" в виде разработанных вами новых управляющих элементов.

Расширяемость Delphi

Благодаря объектно-ориентированной природе Delphi можно создавать собственные компоненты не только с нуля, но и на основе существующего богатого набора стандартных компонентов. В главе 21 второго тома, "Создание пользовательских компонентов в Delphi", мы более подробно поговорим о методах создания новых компонентов на базе уже имеющихся, а в главе 7, "Использование элементов управления ActiveX в Delphi" — об использовании в приложениях управляющих элементов ActiveX.

Помимо создания и интеграции в среду разработки новых компонентов, Delphi также допускает интегрировать в IDE готовые подпрограммы, называемые *экспертами*. Expert Interface позволяет дополнять визуальную среду разработки своими пунктами меню и диалоговыми окнами, предназначенными для расширения ее функциональности. Примером подобного эксперта является утилита Database Form Expert, команда вызова которой находится в меню Database главного окна. Процесс создания собственного эксперта и интеграции его в Delphi IDE описывается в главе 26 второго тома, "Использование интерфейса Open Tools API".

Десять важнейших функций графической среды разработки Delphi

Прежде чем позволить читателю продолжить чтение этой книги, авторы хотят предоставить ему возможность овладеть всеми основными инструментами, которые могут потребоваться в работе. С этой целью ниже приводится краткое описание десяти функций графической среды разработки Delphi, которые мы считаем наиболее важными и необходимыми.

1. Функция дополнения класса (Class Completion)

Больше всего времени разработчики затрачивают на ввод текста программ. Как часто мы точно знаем, что именно хотим написать, но недостаточная скорость ввода текста с клавиатуры невыносимо затягивает достижение желаемого результата! В Delphi имеется специальная функция, позволяющая существенно сократить объем рутинной работы.

Безусловно, самым важной особенностью функции дополнения класса является то, что она не имеет собственного интерфейса. Достаточно просто ввести часть объявления класса и нажать комбинацию клавиш <Ctrl+Shift+C>, для того чтобы функция дополнения класса предприняла попытку определить, что именно предполагалось ввести, и сгенерировала соответствующий текст. Например, если поместить в класс объявление процедуры `Foo`, после этого нажать упомянутую выше комбинацию клавиш, то описание этого метода автоматически будет внесено в раздел реализации данного модуля. Если объявить новое свойство, значение которого считывается из поля, а записывается с помощью метода, то после вызова функции дополнения класса будет автоматически создано определение поля, а также текст описания и реализации метода.

Если приведенные выше примеры вас не убедили, попробуйте эту функцию на практике. Очень скоро она станет вам просто необходимой.

2. Функция навигации AppBrowser

Достаточно часто, глядя на некоторую строку текста программы, приходится мучительно вспоминать, где именно объявляется используемый в ней метод. Самый простой способ выяснить это состоит в нажатии клавиши <Ctrl> и последующем щелчке мышью на интересующей лексеме. Подпрограммы IDE используют заранее собранную компилятором отладочную информацию для перехода к строке объявления указанной лексемы. Как и в случае обычного Web-броузера, функция ведет свой собственный стек переходов, который можно использовать для перемещения в обоих направлениях. Переходы выполняются посредством щелчков мышью на стрелках, размещенных на правом корешке вкладки окна редактора текстов.

3. Перемещение между разделами объявления и реализации

Для перемещения между объявлением и реализацией метода достаточно поместить на его имя курсор ввода и нажать комбинацию клавиш <Ctrl+Shift+Стрелка вверх> или <Ctrl+Shift+Стрелка вниз>.

4. Состыковка окон

Визуальная среда разработки позволяет организовать на экране единое окно, составленное из нескольких состыкованных окон, каждое из которых будет выглядеть как его отдельная панель. Полученное составное окно можно перемещать по экрану как единое целое. Отличить составное окно от обычного очень просто — при перемещении на экране оно особым образом пульсирует. Окно редактора текста позволяет стыковать другие окна с трех его сторон — слева, снизу и справа. Стыковка окна осуществляется путем перетаскивания одного окна вплотную к границе или в центр другого. Завершив перекомпоновку, не забудьте сохранить результаты с помощью кнопок панели инструментов Desktops. Для предотвращения случайной состыковки окон при их перемещении нажмите и удерживайте клавишу <Ctrl>. Кроме того, можно щелкнуть в окне правой кнопкой мыши и сбросить в раскрывшемся контекстном меню флажок опции Dockable.

Имеется еще одна дополнительная возможность. Если щелкнуть правой кнопкой мыши на корешке панели состыкованного окна, то его можно будет переместить в верхнюю, нижнюю, левую или правую часть общего окна.

5. Броузер объектов

В Delphi версий 1–4 использовался очень примитивный броузер объектов. Если вы о нем даже не слышали, то это не удивительно — многие вообще предпочитали им не пользоваться, поскольку возможности его были весьма ограничены. В Delphi 5 броузер объектов полностью переработан. Его окно, которое выводится на экран с помощью команды View⇨Browser, показано на рис. 1.6. В этом окне отображается древовидная структура, позволяющая получать доступ к глобальным переменным, классам и модулям, а также контролировать области видимости, цепочки наследования и ссылки на символы.

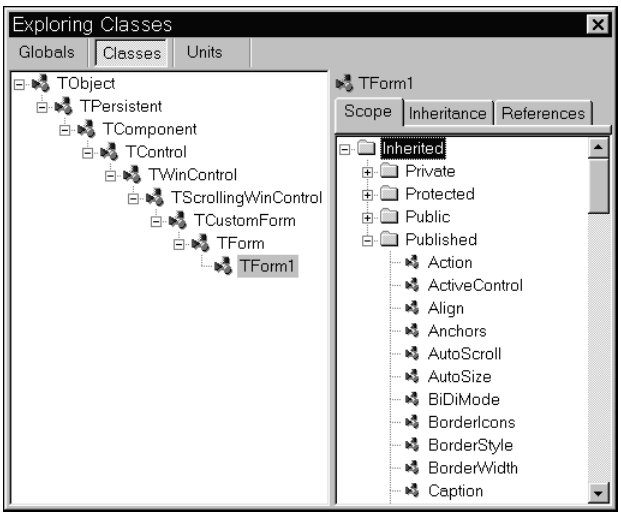


Рис. 1.6. Общий вид окна броузера объектов

6. Создание нового GUID

Нажатие комбинации клавиш <Ctrl+Shift+G> позволяет поместить в окно редактора кода новое уникальное значение GUID (уникального глобального идентификатора интерфейса). Эта возможность оказывается весьма полезной при создании нового интерфейса.

7. Подсветка синтаксиса C + +

Если в процессе работы в среде Delphi вам часто приходится просматривать файлы с программами на языке C++ (например, файлы заголовков SDK), то вы сможете по достоинству оценить эту встроенную функцию редактора кодов. Нужно просто загрузить в редактор файл с текстом программы на C++ — все остальное будет сделано автоматически.

8. Список To Do

Используйте список “To Do List” для управления ходом работы над файлами с программами. Вывести содержимое этого списка можно с помощью команды View⇨To Do List. В этот список автоматически помещаются любые комментарии из создаваемых программ, которые начинаются словом “TODO”. Окно To Do Items может использоваться для определения владельца, приоритета и категории каждого из объектов “To Do”. Общий вид этого окна, пристыкованного к нижней части окна редактора текстов, показан на рис. 1.7.

9. Использование менеджера проектов

Окно Project Manager может оказаться очень удобным инструментом при работе над большими проектами — особенно над теми, которые предусматривают создание нескольких файлов EXE или DLL. Однако многие пользователи просто забывают о его существовании. Вывести это окно на экран можно с помощью команды View⇨Project Manager. В Delphi 5

функциональные возможности этого окна были расширены. В частности, появилась возможность вставки и копирования элементов с одного проекта в другой с помощью операций перетаскивания.

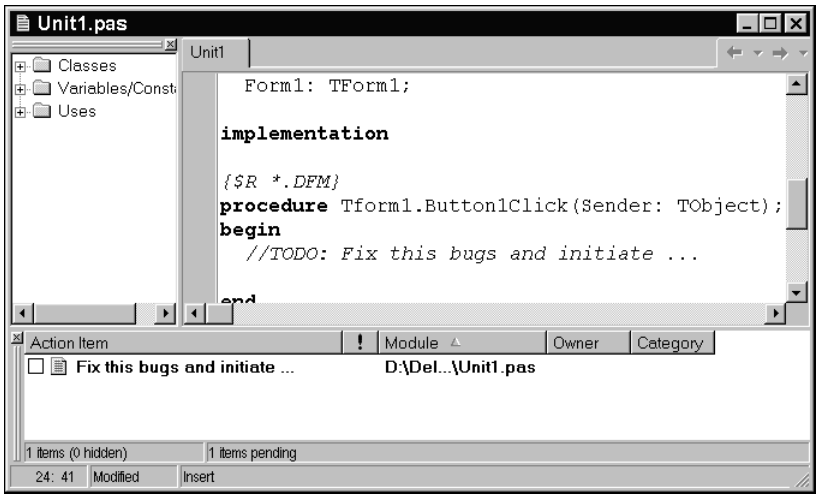


Рис. 1.7. Окно To Do Items пристыковано к нижней части окна редактора текстов

10. Использование функции Code Insight для завершения объявлений и параметров

При вводе ключевого слова `Identifier` после завершающей точки на экран будет автоматически выведено окно со списком свойств, методов, событий и полей, доступных для этого идентификатора. С помощью щелчка правой кнопкой мыши можно отсортировать этот список по именам или по области видимости. Если окно исчезло с экрана до того, как вы успели прочитать его содержимое, для повторного его вывода нужно нажать комбинацию клавиш <Ctrl+Пробел>.

Запоминание всех параметров функций может оказаться слишком обременительным требованием, поэтому функция Code Insight автоматически выводит окно подсказки со списком параметров, как только в окне редактора текстов будет введено любое *`имя_функции(`*. Если это окно исчезло с экрана слишком быстро, можно повторно вывести его с помощью комбинации клавиш <Ctrl+Shift+Пробел>.

Резюме

В этой главе вы познакомились с характеристиками каждого из продуктов серии Delphi 5, освоились с графической средой разработки Delphi, а также получили представление о предлагаемых методах разработки приложений для Windows. Основное назначение этой главы состоит в том, чтобы научить вас ориентироваться в среде Delphi и предоставить несколько базовых концепций, необходимых для освоения материала последующих глав книги. Далее разговор пойдет о сугубо технических аспектах работы с Delphi, поэтому убедитесь, что вы полностью освоились в предлагаемой графической среде разработки и уже можете самостоятельно выполнять небольшие проекты.

Язык программирования Object Pascal

Глава

2

В этой главе не рассматриваются разнообразные визуальные элементы среды разработки Delphi. Она посвящена языку программирования Object Pascal, лежащему в основе этого пакета. Сначала речь пойдет о базовых элементах языка, его правилах и конструктах, а затем будут рассмотрены более сложные вопросы — классы и обработка исключительных ситуаций. Поскольку эта книга не предназначена для начинающих, предполагается, что читатель знаком с другими языками программирования высокого уровня, такими, например, как C, C++ или хотя бы Visual Basic. Мы будем часто проводить сравнение тех или иных положений языка Object Pascal с другими языками программирования. К концу главы вы будете иметь представление о реализации основных концепций программирования (переменных, типах, операторах и многом другом) в Object Pascal и их отличиях от концепций таких языков, как C++ и Visual Basic.

Даже если вы — опытный программист на Pascal, эта глава не будет для вас бесполезной, так как только в ней вы сможете найти полную информацию о синтаксисе и семантике Object Pascal.

Комментарии

Начнем с того, каким образом вносить комментарии в программу на языке Object Pascal. Этот язык поддерживает три типа комментариев — с использованием фигурных скобок, пар скобка–звездочка и двойной наклонной черты (в стиле C++). Вот примеры комментариев Object Pascal:

```
{  Комментарий с использованием фигурных скобок }
(* Комментарий с использованием пары скобка+звездочка *)
// Комментарий в стиле C++
```

Первые два типа комментариев идентичны. Object Pascal считает комментарием все, находящееся между открывающим и закрывающим комментарий символами (или парой символов). Комментарий в стиле C++ начинается с двойной косой черты и продолжается до конца строки.

На заметку

В Object Pascal нельзя использовать вложенные комментарии одного и того же типа. И хотя синтаксис Object Pascal позволяет создавать вложенные комментарии различного типа, мы не рекомендуем пользоваться этой возможностью.

```
{ (* Допустимо *) }
(* { Допустимо } *)
(* (* Недопустимо *) *)
{ { Недопустимо } }
```

Новые возможности процедур и функций

Поскольку без процедур и функций не обходится ни один язык программирования, мы не будем слишком углубляться в эту тему и отметим лишь малоизвестные или новые возможности Object Pascal в этой области.

Скобки

Хотя добавление скобок при вызове функции или процедуры без параметров не является новинкой в Delphi 5, тем не менее эта возможность мало известна. В Object Pascal считаются корректными оба варианта вызова:

```
Form1.Show;
Form1.Show();
```

Это, конечно, возможность не из тех, что потрясают воображение, однако программисты, вынужденные делить свое рабочее время между разными языками, например, такими как Delphi и C++, оценят ее, ибо им не придется специально вспоминать о различном синтаксисе вызова функций в разных языках.

Возможность перегрузки

В Delphi 4 была реализована концепция перегрузки функций (overloading), которая позволяет иметь несколько различных функций или процедур с одинаковым именем, но с разными списками параметров. Такие процедуры и функции должны быть описаны с применением директивы `overload`:

```
procedure Hello(I: Integer); overload;
procedure Hello(S: String);  overload;
procedure Hello(D: Double);  overload;
```

Заметим, что правила перегрузки методов класса, сформулированные в соответствующей главе книги, несколько отличаются от правил перегрузки обычных процедур и функций. Хотя внедрения возможности перегрузки имен функций программисты требовали с момента выхода в свет Delphi 1, тем не менее это не самая безопасная возможность языка. Наряду с уже имевшейся возможностью использования в разных модулях функций с одним и тем же именем, применение перегруженных функций может стать неиссякаемым источником трудноуловимых ошибок в программе. Поэтому сто раз подумайте, прежде чем использовать перегрузку. Помните золотое правило: “Следует потреблять, но не злоупотреблять!”

Значения параметров по умолчанию

В Delphi 4 была введена еще одна полезная возможность — использование значений параметров по умолчанию. Она позволяет установить принимаемое по умолчанию значение параметра процедуры или функции. Это значение будет использоваться в тех случаях, когда вызов процедуры или функции производится без указания значения данного параметра. В объявлении процедуры или функции принимаемое по умолчанию значение параметра указывается после знака равенства, следующего после его имени. Поясним это на следующем примере:

```
procedure HasDefVal(S: String; I: Integer = 0);
```

Подобное объявление означает, что процедура `HasDefVal` может быть вызвана двумя путями. В первом случае — как обычно, с указанием обоих параметров:

```
HasDefVal('hello',26);
```

Во втором случае можно задать только значение параметра S, а для параметра I использовать значение, установленное по умолчанию:

```
HasDefVal('hello'); // Для параметра I используется значение по умолчанию
```

При использовании значений параметров по умолчанию следует помнить о нескольких приведенных ниже правилах.

- Параметры, имеющие значения по умолчанию, должны располагаться в конце списка параметров. Параметр без значения по умолчанию не должен встречаться в списке после параметра, имеющего значение по умолчанию.
- Значения по умолчанию могут присваиваться только параметрам обычных типов, указателям или множествам.
- Значение по умолчанию может передаваться только по значению либо с модификатором const. Оно не может быть ссылкой или нетипизированным параметром.

Одним из важных преимуществ применения значений параметров по умолчанию является простота расширения функциональных возможностей уже имеющихся процедур и функций с соблюдением обратной совместимости. Предположим, на рынок программных продуктов была выпущена программа, ключевым звеном которой является функция сложения двух целых величин:

```
function AddInts(I1, I2: Integer) : Integer;
begin
  Result := I1 + I2;
end;
```

Предположим также, что маркетинговые исследования показали целесообразность добавления в программу возможности сложения трех чисел. Однако замена имеющейся функции функцией сложения трех чисел приведет к тому, что вам придется переправлять немало текста, который перестанет компилироваться из-за внесения в функцию еще одного параметра. Однако при использовании значений параметров по умолчанию проблема решается легко и просто. Достаточно изменить объявление функции так, как показано ниже.

```
function AddInts(I1, I2: Integer; I3: Integer = 0) : Integer;
begin
  Result := I1 + I2 + I3;
end;
```

Переменные

Если вы пишете программы примерно так: “Здесь мне понадобится еще одна переменная — ну что ж, опишем ее прямо среди кода, в том месте, где она понадобилась”, — значит, вы программируете на C++. Однако при использовании переменных в Object Pascal вам следует четко запомнить: переменные должны быть описаны до начала процедуры, функции или программы в целом. Предположим, на C++ текст функции выглядит так:

```
void foo(void)
{
  int x = 1;
```

```
  x++;
  int y = 2:
  float f;
  // и так далее
}
```

Аналогичная ей функция на Object Pascal должна выглядеть следующим образом:

```
Procedure Foo;
var
  x, y : Integer;
  f    : Double;

begin
  x:=1;

  inc(x)
  y := 2;
  // и так далее
end;
```

На заметку

Object Pascal (как и Visual Basic) в отличие от C и C++ не чувствителен к регистру символов. Использование строчных и прописных символов сводится к тому, чтобы сделать текст более удобочитаемым. Так, сложно воспринять имя процедуры, записанное как `procedure thisprocedurenamemakenosense;`. Значительно понятнее имена процедур, записанные в таком виде: `procedure ThisProcedureNameIsMoreClear;`
Описание используемого в этой книге стиля записи программ приведено в главе 6, "Стандарты программирования, принятые в этой книге".

Вы можете спросить: "В чем же в данном случае состоят достоинства и удобства Object Pascal?" Вскоре вы убедитесь, что такая строгая структуризация программы делает ее более понятной, удобочитаемой и, как следствие, приводит к написанию программ, которые проще отлаживать и сопровождать.

Отметим, что Object Pascal позволяет группировать в одной строке объявления нескольких переменных одного типа, например:

```
VarName1, VarName2 : SomeType;
```

Не забывайте, что объявление переменной в Object Pascal состоит из ее имени, двоеточия и типа переменной.

Начиная с Delphi 2.0, язык позволяет инициализировать глобальные переменные в блоке их объявления `var`. Приведенный ниже пример демонстрирует синтаксис такой инициализации.

```
var
  I: Integer = 10;
  S: String  = 'Hello world';
  D: Double  = 3.141579;
```

На заметку

Инициализация допустима только в случае глобальных переменных; инициализировать локальные переменные процедур или функций нельзя.

Компилятор Delphi автоматически инициализирует все глобальные переменные значением 0. Таким образом, при запуске вашей программы все глобальные переменные целого типа примут значение 0, переменные с плавающей точкой — значение 0.0, строки будут пустыми, а указатели иметь значение `nil`. Поэтому специальная инициализация глобальных переменных нулевым значением не требуется.

Константы

Константы в Object Pascal определяются в выражении `const`, действие которого подобно описанию `const` в языке C. Вот пример объявления трех констант в языке C:

```
const float ADecimalNumber = 3.14;
const int I = 10;
const char * ErrorString    = "Danger, Danger, Danger!";
```

Основное отличие между константами в языках C и Object Pascal заключается в том, что в Object Pascal (как и в Visual Basic) вам не требуется указывать тип константы. Компилятор автоматически выделяет необходимую память, основываясь на инициализирующем значении (а в случае скалярных констант, таких как целое значение, он просто подставляет его в нужные места в программе, не выделяя памяти вовсе). Объявления тех же констант в Object Pascal выглядят следующим образом:

```
const
  ADecimalNumber = 3.14;
  i = 10;
  ErrorString    = 'Danger, Danger, Danger!';
```

На заметку

Выделение памяти для констант происходит таким образом: целые значения размещаются в наименьшей, но достаточной для этого памяти (так, значение 10 будет размещено как переменная типа `ShortInt`, 32000 — как `SmallInt` и т.д.). Алфавитно-цифровые значения, а попросту — строки, размещаются в типе `Char` или текущем типе `String`, определенном директивой `$H`. Значения с плавающей точкой размещаются как переменные `extended`, кроме значений, имеющих менее четырех десятичных знаков, — в этом случае используется тип `Comp`. Массивы целых (`Integer`) или символьных (`Char`) данных хранятся в соответствующем представлении.

Object Pascal не запрещает определять тип константы, что дает вам возможность контроля над тем, каким образом компилятор будет трактовать константу во время работы над программой:

```
const
  ADecimalNumber : Double  = 3.14;
  i : Integer = 10;
  ErrorString : String  = 'Danger, Danger, Danger!';
```

При объявлении констант (`const`) и переменных (`var`) в языке Object Pascal допускается использование функций, вычисляемых во время компиляции. К этим функциям относятся `Ord()`, `Chr()`, `Trunc()`, `Round()`, `High()`, `Low()` и `SizeOf()`. Так, все объявления в следующем примере абсолютно корректны:

```
type
  A = array [1..2] of Integer;

const
  w: Word = SizeOf(Byte);

var
  I : Integer = 8;
  j : SmallInt = Ord('a');
  L : LongInt = Trunc(3.14159);
  x : ShortInt = Round(2.71828);
  B1 : Byte = High(A);
  B2 : Byte = Low(A);
  C : Char = Chr(46);
```

Если вы попытаетесь присвоить константе некоторое значение, компилятор выдаст сообщение об ошибке, поясняющее, что значение константы не может быть изменено. Исходя из того факта, что константы доступны только для чтения, компилятор Object Pascal оптимизирует использование памяти, располагая константы в незаполненных частях страниц кода. Более детальную информацию о страницах кода и данных можно найти в главе 3, "Win32 API".

Поведение констант в 32-битовом Delphi отличается от такового в 16-битовом Delphi 1.0. В Delphi 1.0 константа рассматривалась как инициализированная переменная, называемая *типизированной константой*. Начиная с Delphi 2.0 константы стали истинными константами, а для обратной совместимости была введена директива компилятора `$J`. (Кроме того, этот режим может быть задан с помощью флажка опции Assignable typed constants, расположенном во вкладке Compiler окна Project Options.) Однако мы рекомендуем не полагаться на поддержку обратной совместимости, а потратить некоторое время и усилия, чтобы избавиться от устаревших типизированных констант в ваших программах.

На заметку

Object Pascal не имеет препроцессора, как C или C++. Таким образом, в Object Pascal отсутствует концепция макросов, а значит, и эквивалент использования `#define` в C для определения констант. Хотя язык Object Pascal позволяет выполнять условную компиляцию с использованием директивы компилятора `$define`, в целом подобной директиве `#define` в C, ее нельзя применять для определения констант. Там, где в языке C конструкция `#define` использовалась для определения постоянной, в языке Object Pascal мы рекомендуем применять описание `const`.

Операторы

Операторами являются те символы в тексте программы, с помощью которых выполняются определенные действия с данными различных типов. Простейшим примером могут служить операторы сложения, вычитания, умножения и деления арифметических типов данных; другим примером может быть оператор для доступа к определенному элементу массива. В этом разделе вы познакомитесь с операторами Object Pascal и узнаете о некоторых их отличиях от операторов C и Visual Basic.

Оператор присвоения

Это один из чаще всего используемых операторов. Для того чтобы присвоить значение переменной, применяется оператор `:=`, аналогичный оператору `=` в языках C или Visual Basic. Иногда программисты называют его оператором *получения* (gets) (или назначения). Рассмотрим пример оператора присвоения:

```
Number := 5;
```

Эту строку программы можно прочитать как "переменная `Number` получает значение `5`" либо как "переменной `Number` присвоено значение `5`".

Операторы сравнения

Если вы работали с Visual Basic, то использование операторов сравнения в Object Pascal не вызовет никаких проблем, так как они идентичны соответствующим операторам Visual Basic.

Object Pascal использует оператор `=` для выполнения сравнения двух выражений или значений. В языке C ему аналогичен оператор `==`, пример использования которого показан ниже.

```
if ( x == y )
```

На языке Object Pascal этот же оператор будет выглядеть следующим образом:

```
if x = y
```

На заметку Твердо запомните, что в Object Pascal оператор `=` используется только для сравнения. Для присвоения значений следует пользоваться оператором `:=`.

В Delphi оператор "не равно" выглядит так: `<>`. В языке C ему аналогичен оператор `!=`. Вот пример проверки того, что два значения не равны друг другу:

```
if x <> y then DoSomething
```

Логические операторы

В качестве логических операторов "и" и "или" в языке Object Pascal используются ключевые слова `and` и `or` (в языке C для этой цели применяются операторы `&&` и `||`). В основном эти операторы применяются как элементы оператора `if` или цикла. Например:

```
if (Condition1) and (Condition2) then DoSomething;
while (Condition1) or (Condition2) do DoSomething;
```

Оператор "нет" в Object Pascal выглядит как `not` (он аналогичен оператору `!` в языке C). В основном оператор `not` применяется в составе оператора `if`; это демонстрирует следующий пример:

```
if not (условие) then doSome; //Если условие ложно, то
```

В табл. 2.1 приведен список операторов Object Pascal и соответствующие им операторы языков C и Visual Basic.

Таблица 2.1. Операторы присвоения, сравнения и логические

Оператор	Object Pascal	C	Visual Basic
Присвоение	:=	=	=
Сравнение	=	==	= или is*
Неравенство	<>	!=	<>
Меньше, чем	<	<	<
Больше, чем	>	>	>
Меньше или равно	<=	<=	<=
Больше или равно	>=	>=	>=
Логическое *и*	and	&&	and
Логическое *или*	or	||	or
Логическое *нет*	not	!	not

*Оператор сравнения `is` используется только для объектов, тогда как оператор сравнения = применяется для всех остальных типов данных.

Арифметические операторы

Большинство арифметических операторов Object Pascal должно быть вам знакомо, так как они идентичны операторам, используемым в языках C, C++ и Visual Basic. В табл. 2.2 приведены арифметические операторы Object Pascal и соответствующие им операторы языков C и Visual Basic.

Таблица 2.2. Арифметические операторы

Оператор	Object Pascal	C	Visual Basic
Сложение	+	+	+
Вычитание	-	-	-
Умножение	*	*	*
Деление с плавающей точкой	/	/	/
Деление целых чисел	div	/	\
Деление по модулю	mod	%	Mod
Возведение в степень	Отсутствует	Отсутствует	^

Как видите, основное различие между Object Pascal и другими языками программирования состоит в наличии различных операторов для деления целых чисел и чисел с плавающей точкой. Оператор `div` автоматически отсекает остаток при делении двух целых выражений.

На заметку

Не забывайте использовать правильный оператор деления для типов выражений, с которыми вы работаете. Компилятор выдаст сообщение об ошибке при попытке деления двух чисел с плавающей точкой с помощью оператора `div` или двух целых чисел с помощью оператора `/`, например:

```
var
  i: Integer;
  r: Real;
begin
  i := 4 / 3;        //Здесь компилятор сообщит об ошибке
  f := 3.4 div 2.3;  //Эта строка также содержит ошибку
end;
```

В большинстве других языков программирования не делается различия между делением целочисленным и делением с плавающей точкой. Как правило, всегда выполняется деление с плавающей точкой, а, при необходимости, результат конвертируется в целое число. Однако при таком подходе возможно существенное снижение производительности программы. Оператор `div` языка Pascal является более специализированным, а потому и выполняется быстрее.

Побитовые операторы

Это операторы, позволяющие работать с отдельными битами заданной переменной. Чаще всего побитовые операторы используются для сдвига битов влево или вправо, их инверсии, а также побитовых операций "и", "или" и "исключающее или" между двумя числами. Операторы сдвига влево и вправо в Object Pascal имеют вид `shl` и `shr` соответственно, они аналогичны операторам `<<` и `>>` в языке C. Запомнить остальные операторы тоже достаточно легко — это `not`, `and`, `or` и `xor`. В табл. 2.3 приведены побитовые операторы и соответствующие им операторы языков C и Visual Basic.

Процедуры увеличения и уменьшения

Процедуры увеличения и уменьшения генерируют оптимизированный код для добавления или вычитания единицы из целой переменной. Object Pascal не предоставляет таких широких возможностей, как постфиксные и префиксные операторы `++` и `--` в языке C. Тем не менее процедуры `Inc()` и `Dec()` обычно компилируются в одну команду процессора.

Таблица 2.3. Побитовые операторы

Оператор	Object Pascal	C	Visual Basic
И	`and`	`&`	`And`
Не	`not`	`~`	`Not`
Или	`or`	`|`	`Or`
Исключающее или	`xor`	`^`	`Xor`
Сдвиг влево	`shl`	`<<`	Нет
Сдвиг вправо	`shr`	`>>`	Нет

Процедуры `Inc()` и `Dec()` можно вызывать как с одним, так и с двумя параметрами. Ниже приведен пример их вызова с одним параметром.

```
Inc(variable);
Dec(variable);
```

Эти операторы будут скомпилированы в ассемблерные инструкции `inc` и `dec`. Вот пример вызова этих процедур с двумя параметрами:

```
Inc(variable,3);
Dec(variable,3);
```

В данном случае выполняется увеличение (уменьшение) переменной `variable` на 3, которое будет реализовано с помощью инструкции `add` или `sub`.

В табл. 2.4 приведены данные об операторах увеличения и уменьшения в разных языках программирования.

Таблица 2.4. Операторы увеличения и уменьшения

Оператор	Object Pascal	C	Visual Basic
Увеличение	`Inc()`	`++`	Нет
Уменьшение	`Dec()`	`--`	Нет

На заметку Если для компилятора установлен режим оптимизации, функции `Inc()` и `Dec()` обычно порождают такой же машинный код, что и выражения `variable:=variable + 1`. Поэтому для увеличения или уменьшения значения целой переменной можно использовать тот вариант записи, который покажется вам более удобным.

Типы данных Object Pascal

Одно из основных и важнейших свойств языка Object Pascal — строгая типизация данных. В частности, это означает, что все реальные переменные, передаваемые в качестве параметров в функцию или процедуру, должны абсолютно точно соответствовать типу формальных параметров в объявлении этой функции или процедуры. В Object Pascal вы не увидите предупреждений о проведении подозрительного преобразования типа указателя, с которыми хорошо знакомы программисты на языке C. Компилятор Object Pascal не допустит вызова функции с типом указателя, отличным от того, который описан в объявлении этой функции. (Однако в функцию, параметр которой описан как нетипизированный указатель, можно передавать указатели любых типов.) В целом строгое типизирование данных в языке Pascal имеет целью выполнение проверок и выдачу предупреждений о попытках применения квадратных пробок для затыкания круглых дырок.

Сравнение типов данных

Основные типы данных Object Pascal схожи с типами данных языков C или Visual Basic. В табл. 2.5 приведено сравнение типов данных этих языков. Настоятельно рекомендуем пометить эту страницу закладкой — она содержит отличный справочный материал, который будет полезен при вызове функций из динамически компонуемых библиотек, созданных другими компиляторами.

Таблица 2.5. Сравнение типов разных языков программирования

Тип переменной	Object Pascal	C / C++	Visual Basic
8-битовое целое со знаком	ShortInt	char	Нет
8-битовое целое без знака	Byte	unsigned char	Нет
16-битовое целое со знаком	SmallInt	short	Short
16-битовое целое без знака	Word	unsigned short	Нет
32-битовое целое со знаком	Integer, Longint	int, long	Integer
32-битовое целое без знака	Cardinal, LongWord	unsigned long	Нет
64-битовое целое со знаком	Int64	__int64	Нет
4-байтовое вещественное	Single	float	Single
6-байтовое вещественное	Real48	Нет	Нет
8-байтовое вещественное	Double	double	Double
10-байтовое вещественное	Extended	long double	Нет
64-битовое денежное	currency	Нет	Currency
8-битовое дата/время	TDataTime	Нет	Data
16-байтовый вариант	Variant, OleVariant, TVarData	Variant*, OleVariant*	По умолчанию
1-байтовый символ	Char	char	Нет
2-байтовый символ	WideChar	WCHAR	Нет
Строка фиксированной длины	ShortString	Нет	Нет
Динамическая строка	AnsiString	AnsiString*	$
Строка с завершающим нулевым символом	PChar	char*	Нет
Строка 2-байтовых символов с завершающим нулевым символом	PWideChar	LPCWSTR	Нет
Динамическая 2-байтовая строка	WideString	WideString*	Нет
1-байтовое булево	Boolean, ByteBool	(Любое 1-байтовое)	Нет
2-байтовое булево	WordBool	(Любое 2-байтовое)	Boolean
4-байтовое булево	BOOL, LongBool	BOOL	Нет

* Классы Borland C++ Builder для эмуляции соответствующих типов Object Pascal.

На заметку

Если вы переносите 16-битовый код из Delphi 1.0, помните, что размер типов `Integer` и `Cardinal` вырос с 16 до 32 бит. Впрочем, это утверждение не совсем точно: в Delphi 2 и 3 тип `Cardinal` трактовался как 31-битовое целое без знака — в соответствии с теми результатами, которые могли быть получены при выполнении целочисленных операций. Однако уже в Delphi 4 `Cardinal` представляет собой истинное 32-битовое целое без знака.

В Delphi 1, 2 и 3 тип `Real` определял 6-байтовое вещественное число. Такой тип присущ только языку Pascal и не совместим с другими языками программирования. В Delphi 4 этот тип стал синонимом типа `Double`. Старый 6-байтовый тип остался в языке под именем `Real48`, однако вы можете заставить компилятор трактовать тип `Real` как 6-байтовый с помощью директивы компилятора `{$REALCOMPATIBILITY ON}`.

Символьные типы

В Delphi существует три символьных типа.

- `AnsiChar` — стандартный 1-байтовый символ, хорошо знакомый всем программистам.
- `WideChar` — 2-байтовый символ Unicode.
- `Char` — сейчас это тип, эквивалентный `AnsiChar`, однако Borland предупреждает, что в последующих версиях он может измениться и стать эквивалентным `WideChar`.

В дальнейшем не рекомендуется полагаться на казавшийся столь незыблемым факт, что размер символа всегда равен одному байту. Теперь везде, где используется длина символа, вместо единицы следует указывать выражение `SizeOf(Char)`.

На заметку

Функция `SizeOf()` — стандартная, она возвращает размер того или иного типа либо его экземпляра, выраженный в байтах.

Различные строковые типы

Строки представляют собой типы данных, используемые для представления групп символов. Каждый язык по-своему решает проблему размещения в памяти и использования строк. Object Pascal имеет несколько различных типов строк, и вы можете выбирать тот или иной тип строки исходя из конкретной ситуации.

- `AnsiString` — основной тип строки в Object Pascal, состоящей из символов `AnsiChar` и теоретически не имеющий ограничений по длине. Этот тип совместим со строками с завершающим нулевым символом.
- `ShortString` — остался в языке для обратной совместимости Delphi 1. Максимальная длина этой строки — 255 символов.
- `WideString` — по своей сути сходен с `AnsiString`. Единственное отличие состоит в том, что данная строка состоит из символов типа `WideChar`.
- `PChar` — представляет собой указатель на строку с завершающим нулевым символом, состоящую из символов типа `Char`. Аналог типов `char*` или `lpstr` в языке C.
- `PAnsiChar` — указатель на строку `AnsiChar` с завершающим нулевым символом.
- `PWideChar` — указатель на строку `WideChar` с завершающим нулевым символом.

По умолчанию при объявлении строковой переменной компилятор полагает, что создается строка типа AnsiString:

```
var
  S: string;  // Переменная S имеет тип AnsiString
```

Для изменения принимаемого по умолчанию типа строки используется директива компилятора $H. Ее положительное (по умолчанию) значение определяет использование в качестве стандартного строчного типа AnsiString, отрицательное — ShortString. Вот пример использования директивы $H для изменения строчного типа, выбираемого по умолчанию:

```
var
  {$H-}
  S1: String;    // Переменная S1 имеет тип ShortString
  {$H+}
  S2: String;    // Переменная S2 имеет тип AnsiString
```

Исключением из этого правила являются строки, объявленные с установленным фиксированным размером. Если заданная длина не превышает 255 символов, такие строки всегда имеют тип ShortString:

```
var
  S: String[63]; // Это строка типа ShortString размером 63 символа
```

Тип AnsiString

Тип AnsiString, известный как "длинная строка", был введен в Delphi 2.0 в ответ на требования пользователей отменить 255-символьное ограничение на длину строки.

Хотя в использовании строка AnsiString практически ничем не отличается от своей предшественницы, память для нее выделяется динамически, а для ее освобождения применяется технология "сборки мусора" (garbage collect). Благодаря этому AnsiString является типом *с управляемым временем жизни* (lifetime-managed). Object Pascal сам автоматически выделяет память для временных строк, так что вам не придется беспокоиться об этом, как в случае языков C/C++, а также сам заботится о том, чтобы строка всегда была с завершающим нулевым символом (и тем самым обеспечивает возможность использования ее с Win32 API). На самом деле тип AnsiString представляет собой определенную структуру в памяти, показанную на рис. 2.1.

Рис. 2.1. Размещение AnsiString в памяти

Полный внутренний формат длинной строки не документирован фирмой Borland, а следовательно, она оставляет за собой право изменять его в дальнейших версиях языка. Поэтому приведенная здесь информация о структуре AnsiString служит только для того, чтобы облегчить понимание ее функционирования, и вы не должны использовать эти сведения при написании программ.

Разработчики программ, которые избегали использования внутренней структуры строк, при переходе от Delphi 1 к Delphi 2 смогли перекомпилировать свои программы без проблем. Те же, кто опирались на внутренний формат строки (например на то, что нулевой символ строки содержит ее длину), должны были соответствующим образом изменить свой код.

Как показано на рис. 2.1, в строке AnsiString имеется счетчик ссылок, показывающий количество строковых переменных, ссылающихся на одно и то же место в памяти. При этом такая операция, как копирование, по сути, сводится к копированию указателя и увеличению счетчика. Реальное копирование строки, замедляющее работу программы, происходит только в необходимых случаях, как показано в следующем примере:

```
var
  S1, S2: String;
begin
  //Сохраняем строку в S1, при этом счетчик S1 становится равным 1
  S1 := 'Некоторая строка';
  S2 := S1;  //Реального копирования строки не
             //происходит – копируется указатель;
             //значение счетчика становится равным 2
  //Далее значение S2 изменяется, поэтому для строки
  //S2 выделяется память, туда копируется содержимое
  //строки S1, счетчик которой уменьшается на 1
  S2 := S2 + ' теперь изменена';
```

Типы с управляемым временем жизни

Помимо AnsiString, в Delphi существует несколько других типов данных с управляемым временем жизни. Это динамические массивы WideString, Variant, OleVariant, interface и dispinterface. Далее в этой главе они будут описаны подробнее, а сейчас нас интересует только один вопрос: что такое управляемое время жизни и как это работает?

Такие типы, которые иногда называют типами "со сборкой мусора", используют некоторые ресурсы компьютера и автоматически освобождают их при выходе из области видимости. Естественно, то, какие именно ресурсы требуются тому или иному типу, зависит исключительно от него. Так, тип AnsiString использует память компьютера для хранения своих данных, и при выходе из области видимости эта память должна быть освобождена.

Для глобальных переменных решение этой проблемы достаточно тривиально — соответствующий код освобождения ресурсов помещается в завершающий код приложения. Поскольку при начальной инициализации происходит инициализация глобальных переменных нулевыми значениями, т.е. фактически значениями "не используется", завершающий код может принимать решение о необходимости освобождения тех или иных ресурсов по состоянию глобальной переменной и выполнять сборку мусора только там, где это необходимо.

Для локальных переменных этот процесс заметно сложнее. Во-первых, компилятору необходимо внести код, который инициализирует переменную всякий раз при входе в функцию или процедуру. Во-вторых, компилятор генерирует блок обработки исключительных ситуаций try finally вокруг всего тела функции. И, наконец, компилятор помещает код сборки мусора в часть finally блока (об обработке исключительных ситуаций вы узнаете в соответствующем разделе этой главы). Чтобы вы могли лучше представить себе эти действия, рассмотрим простейший пример:

```
procedure foo;
var
  S: String;
begin
  // Тело процедуры с использованием S
end;
```

Хотя процедура и выглядит очень просто, де-факто компилятор рассматривает ее как

```
procedure foo;
var
    S: String;
begin
    S := '';
    trya
        // Тело процедуры с использованием S
    finally
        // Освобождение занятых для S ресурсов
    end;
end;
```

Строковые операции

Добавить одну строку в конец другой можно с помощью оператора + или функции `Concat()`. Предпочтительнее использовать оператор +, так как функция `Concat()` сохранена в основном из соображений обратной совместимости. Вот примеры использования оператора и функции:

```
{ Использование оператора + }
var
  S, S2: string
begin
  S := 'Cookie ':
  S2 := 'Monster';
  S := S + S2;   { Cookie Monster }
end.
{ Использование функции Concat() }
var
  S, S2: string;
begin
  S := 'Cookie ';
  S2 := 'Monster';
  S := Concat(S, S2);  { Cookie Monster }
end.
```

`Concat()` — одна из множества особых функций и процедур (подобно `ReadLn()` и `WriteLn()`), которые не имеют объявления в языке Object Pascal. Функции и процедуры такого типа отличаются тем, что могут принимать необязательные параметры или иметь неопределенное количество параметров. Это делает невозможным определение таких функций средствами самого языка. Поэтому компилятор обрабатывает такие функции особым образом и генерирует вызов одной из *вспомогательных функций* модуля `System`. Как правило, эти вспомогательные функции написаны на языке ассемблера — для обхода ограничений Object Pascal.

В дополнение к таким функциям поддержки работы со строками имеется также ряд других функций модуля `SysUtils`, предназначенных для упрощения работы со строками. О них вы можете узнать в разделе “String-handling routines (Pascal-style)” справочной системы Delphi.

На заметку Не забывайте использовать *одинарные* кавычки (`'Это строка'`) при работе со строками в Object Pascal.

Длина и размещение в памяти

При первом объявлении строка `AnsiString` не имеет длины, а значит, память для ее значения не выделяется. Чтобы выделить память строке, следует либо присвоить ей некоторое строковое значение (например, литерал), либо использовать процедуру `SetLength()`, как показано ниже.

```
var
  S: string;   // Вначале строка не имеет длины
begin
  S := 'Doh!'; // Выделяется память, необходимая для размещения строки
  { или }
  S := OtherString; // Увеличивается счетчик ссылок строки OtherString
                    // (Предполагается, что строка OtherString уже имеет
значение)
  { или }
  SetLength(S, 4);  // Строке выделяется память, необходимая для
                    // размещения четырех символов
end;
```

Строка `AnsiString` может рассматриваться как массив символов, однако будьте осторожны — индекс элемента массива не может превышать длину строки. Так, приведенный ниже код вызовет ошибку.

```
var
  S: string;
begin
  S[1] := 'a'; // Ошибка - память для S не выделена
end;
```

Корректно работать будет следующий вариант этого же примера:

```
var
  S: string;
begin
  SetLength(S, 1);
  S[1] := 'a'; // Теперь S уже выделена память, необходимая
               // для размещения одного символа
end;
```

Совместимость с API Win32

Как указывалось ранее, строка типа `AnsiString` всегда завершается нулевым символом. Поэтому такая строка вполне совместима с функциями API Win32 или любыми другими, использующими параметры типа `PChar`. Все, что в данном случае необходимо, — это преобразование типа `string` в тип `PChar` (о преобразовании типов в Object Pascal можно узнать в раз-

деле “Приведение и преобразование типов” этой же главы). Приведенный ниже пример демонстрирует вызов функции Win32 GetWindowsDirectory(), которой передается в качестве параметров указатель на буфер PChar и его размер.

```
var
  S: string;
begin
  SetLength(S, 256);  // Важно! Сначала выделяем память для строки
  GetWindowsDirectory(PChar(S), 256);
end;
```

После использования AnsiString в качестве аргумента PChar необходимо вручную установить длину строковой переменной равной длине строки с завершающим нулевым символом. Для этого можно использовать процедуру RealizeLength() из упоминавшегося ранее модуля STRUTILS.

```
procedure RealizeLength(var S: string);
begin
  SetLength(S, StrLen(PChar(S)));
end;
```

Вызов этой процедуры завершает использование строки AnsiString как строки PChar:

```
var
  S: string;
begin
  SetLength(S, 256);  // Важно! Выделяем память для строки
  GetWindowsDirectory(PChar(S), 256);
  RealizeLength(S);   // Устанавливаем длину S
end;
```

Проявляйте определенную осторожность при приведении типов string к типу PChar. Поскольку string — это тип с управляемым временем жизни, следует обратить внимание на область видимости соответствующих переменных. Так, если вы делаете присвоение типа P := PChar(Str) и область видимости P больше, чем Str, результат может оказаться плачевным.

Вопросы переносимости

При переносе на новую 32-битовую платформу старых 16-битовых приложений, созданных с помощью Delphi 1.0, вам необходимо помнить следующее.

- Там, где применялся тип PString (указатель на ShortString), следует использовать тип String. Помните, что AnsiString представляет собой указатель на строку.
- Вы больше не можете использовать нулевой элемент строки для получения ее длины. Вместо этого следует прибегнуть к функции Length() и процедуре SetLength() для определения или установки нового значения длины строки.
- Для приведения типов строк между String и PChar больше не требуются функции StrPas() и StrPCopy(). Как уже указывалось, можно выполнять прямое преобразование типа AnsiString в PChar. При копировании содержимого PChar в AnsiString можно использовать обычное присвоение: StringVar := PCharVar;.

Не забывайте, что для установки длины строк типа AnsiString должна использоваться процедура SetLength(). Прежняя практика прямого обращения к нулевому элементу коротких строк здесь не применима. Это замечание следует помнить при переносе 16-разрядных приложений Delphi 1.0 в 32-разрядную среду.

Тип ShortString

Если вы — не новичок в Delphi, то наверняка знаете, что ShortString — это тип String в Delphi 1.0, он же тип с байтом длины строки. Еще раз напомним, что директива компилятора $H позволяет выбрать, какой именно тип будет пониматься под именем String — AnsiString или ShortString.

В памяти строка ShortString представляет собой массив символов, причем нулевой элемент этого массива содержит длину строки. Последующие символы составляют содержание строки, при этом ее максимальная длина не может превышать 255 байт, а вся строка в памяти не может занимать более 256 байт (255 байт строки и один байт длины). Как и в случае с типом AnsiString, при работе со строками ShortString вам не нужно беспокоиться о выделении памяти для временных строк или ее освобождении — всю эту работу компилятор берет на себя.

На рис. 2.2 показано размещение строки ShortString в памяти.

Рис. 2.2. Размещение строки типа ShortString в памяти

Ниже приведено объявление и инициализация строки типа ShortString.

```
var
  S: ShortString;
begin
  S := 'Bob the cat.';
end.
```

При необходимости строке можно выделить меньше 255 байт памяти, для чего в ее объявление следует поместить явное указание длины:

```
var
    S: string[45];  { 45-символьная строка }
begin
    S := 'В этой строке не более 45 символов.';
end.
```

В этом примере будет создана строка типа ShortString вне зависимости от состояния директивы компилятора $H. Следовательно, максимальное допустимое значение модификатора длины составляет 255 символов.

Не следует сохранять в строке символов больше, чем ее реальная длина. Если строка была описана как String[8], то при попытке сохранить в ней значение 'Windows suxx, OS/2 rulez' строка будет обрезана до восьми символов, а остальная часть данных потеряна.

При обращении к отдельным символам в строке ShortString как к элементам массива необходимо следить за тем, чтобы индекс элемента находился строго в допустимых пределах запрошенной памяти. При попытке использовать индекс элемента, превосходящий объявленный размер строки, полученные результаты будут либо ошибочными, либо вызовут разрушение памяти. Предположим, что переменная была объявлена следующим образом:

```
var
  Str: string[8]
```

Если выполнить запись в десятый элемент этой строки, то в результате, вероятнее всего, будет испорчена память, выделенная другой переменной.

```
Var
  Str: string[8];
  i: Integer;
begin
  1:=10;
  Str[i] := s;  // Этот оператор вызовет разрушение памяти
```

Можно потребовать от компилятора сгенерировать код проверки выхода индекса за допустимые пределы в процессе выполнения программы. Для этого следует установить флажок опции Range Checking во вкладке Compiler диалогового окна Project Options.

Хотя включение в программу логики проверки допустимости значений индексов упрощает поиск ошибок в работе со строками, выполнение проверок существенно сказывается на производительности приложения. В связи с этим данный режим следует применять только на этапе отладки и тестирования. Готовый программный продукт рекомендуется транслировать без использования этого режима.

В отличие от строк типа AnsiString, строки ShortString не совместимы со строками в формате с завершающим нулевым символом. Чтобы использовать их в функциях Win32 API необходима определенная предварительная обработка. Для этого можно воспользоваться функцией ShortStringAsPChar(), которая входит в состав уже упоминавшегося модуля STRUTILS. Текст данной функции приведен ниже.

```
function ShortStringAsPChar(var S: ShortString): PChar;
{ Функция добавляет к исходной строке завершающий нулевой символ.
  Длина строки не должна превышать 254 символа }
begin
  if Length(S) = High(S) then Dec(S[0]);
  { Обрезка строки, если она слишком длинна }
  S[Ord(Length(S)) + 1] := #0;
  { Добавление нулевого символа в конец строки }
  Result := @S[1]; { Возвращается указатель PChar }
end;
```

Функции и процедуры Win32 API работают только со строками в формате с завершающим нулем. Не пытайтесь передать на их вход строки типа ShortString непосредственно — программа просто не будет компилироваться. Можно существенно облегчить себе жизнь, если при работе с функциями API использовать только длинные строки типа AnsiString.

Тип WideString

Это тип строки с управляемым временем жизни, подобный типу AnsiString. Оба этих типа являются динамически распределяемыми. Они подвергаются процедурам сборки мусора и даже по назначению схожи. Однако между типами WideString и AnsiString имеются три важных отличия.

- Строка WideString состоит из символов WideChar, что делает ее совместимой со строками Unicode.

- Строки WideString используют память, выделенную с помощью функции API SysAllocStrLen(), что делает их совместимыми со строками OLE BSTR.
- В строках WideString отсутствует счетчик ссылок, поэтому любое присвоение одной строки другой приводит к выделению памяти и копированию строки. Это делает данный тип строк менее эффективным, чем AnsiString, с точки зрения производительности и использования памяти.

Компилятор способен автоматически конвертировать строки типов WideString и AnsiString при присвоении, как показано в следующем примере:

```
var
  W: WideString;
  S: string;
begin
  W := 'Margaritaville';
  S := W;  // Преобразование строки WideString в строку AnsiString
  S := 'Come Monday';
  W := S;  // Преобразование строки AnsiString в строку WideString
end;
```

Для упрощения работы со строками этого типа Object Pascal перегружает функции Concat(), Copy(), Insert(), Length(), Pos() и SetLength(), а также операторы +, = и <> для работы с WideString. Таким образом, следующий код синтаксически безупречен:

```
var
  W1, W2: WideString;
  P: Integer;
begin
  W1 := 'Enfield';
  W2 := 'field';
  if W1 <> W2 then
    P := Pos(W1, W2);
end;
```

Как и в случае строк типа AnsiString или ShortString, для ссылок на отдельные символы строки WideString можно использовать квадратные скобки:

```
var
  W: WideString;
  C: WideChar;
begin
  W := 'Ebony and Ivory living in perfect harmony';
  C := W[Length(W)];  // Теперь C содержит последний символ строки W
end;
```

Строки с завершающим нулевым символом

Ранее в этой главе упоминалось, что в Delphi имеется три различных типа строк с завершающим нулевым символом: PChar, PAnsiChar и PWideChar. Это строки с завершающим нулем для каждого из трех строковых типов Delphi. В данной главе на все эти типы строк мы

будем ссылаться как на тип PChar. Тип PChar в основном обеспечивает обратную совместимость с Delphi 1.0 и работу с Win32 API, широко использующим строки с завершающим нулевым символом. Тип PChar определяется как указатель на строку с завершающим нулевым символом. (Подробно указатели будут рассматриваться ниже, в этой же главе). В отличие от строк AnsiString и WideString, память для строк типа PChar автоматически не выделяется (и не освобождается). Это значит, что для тех строк, на которые указывают данные указатели, память потребуется выделять вручную — с помощью одной из существующих в Object Pascal функций выделения памяти. Теоретически, длина строки PChar может достигать 4 Гбайт. Ее размещение в памяти показано на рис. 2.3.

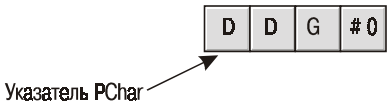


Рис. 2.3. Размещение строки типа PChar в памяти

Поскольку в большинстве случаев вместо PChar может использоваться тип AnsiString, мы рекомендуем применять его везде, где это возможно. При этом удается избежать множества неприятностей, связанных с выделением и освобождением памяти, поскольку при работе с AnsiString компилятор берет заботу об этом на себя.

Как уже упоминалось, переменные типа PChar требуют ручного выделения и освобождения буферов, содержащих соответствующие строки. Обычно выделение памяти осуществляется с помощью функции StrAlloc(), однако для создания буферов PChar можно применять и другие функции, такие, например, как AllocMem(), GetMem(), StrNew() или даже функции API VirtualAlloc(). Каждой функции соответствует функция освобождения памяти (табл. 2.6).

Таблица 2.6. Функции выделения и освобождения памяти

Функция выделения памяти	Функция освобождения памяти
AllocMem()	FreeMem()
GlobalAlloc()	GlobalFree()
GetMem()	FreeMem()
New()	Dispose()
StrAlloc()	StrDispose()
StrNew()	StrDispose()
VirtualAlloc()	VirtualFree()

Вот пример техники выделения памяти при работе с данными типа PChar и String:

```
var
  P1, P2: PChar;
  S1, S2: string;
begin
  P1 := StrAlloc(64 * SizeOf(Char));
      // P1 указывает на выделенные 63 байта памяти
  StrPCopy(P1, 'Delphi 5 ');  // Копирование литерала в строку P1
  S1 := 'Developer''s Guide'; // Помещение текста в строку S1
  P2 := StrNew(PChar(S1));    // P2 указывает на копию S1
```

```
  StrCat(P1, P2);             // Объединение P1 и P2
  S2 := P1;                   // Теперь S2 содержит 'Delphi 5 Developer's Guide'
  StrDispose(P1);             // Освобождение буфера P1
  StrDispose(P2);             // Освобождение буфера P2
end.
```

Обратите особое внимание на то, что при выделении памяти для P1 использовалась функция SizeOf(Char). Вполне вероятно, что в будущих версиях Delphi размер символа может измениться с одного до двух байт, а такой способ выделения памяти для строки будет корректно работать при любых изменениях в размере символа.

Для объединения двух строк использовалась функция StrCat() — при работе с PChar нельзя применять оператор +, как это делается в случае со строками типа AnsiString или ShortString.

Функция StrNew() была использована с целью выделения памяти для строки P2 и копирования в нее строки S1. Однако следует помнить, что при этом память выделяется в количестве, необходимом только для хранения конкретного значения строки. Попытка записи в этот буфер более длинной строки приведет к разрушению памяти. Следующий пример иллюстрирует подобную ситуацию.

```
var
P1, P2: Pchar;
begin
  // Выделяется память, необходимая для размещения строк в P1 и P2
  P1 := StrNew('Hello ');
  P2 := StrNew('World');
  StrCat(P1, P2);   // Попытка записи за пределы выделенной памяти
  .
  .
  .
end;
```

Как и в случае с другими типами строк, для работы с типом PChar Object Pascal предоставляет различные функции и процедуры. Подробные сведения о них можно найти в разделе “String-handling routines (null-terminated)” справочной системы Delphi.

Тип Variant

В Delphi 2.0 был введен новый мощный тип данных — Variant. В основном его назначение состояло в поддержке автоматизации OLE, где тип данных Variant используется очень широко. Фактически, тип Variant языка Object Pascal является инкапсуляцией *вариантов* OLE. Как мы вскоре увидим, реализация в Delphi вариантов оказалась полезной и с точки зрения других аспектов программирования. Object Pascal является единственным компилируемым языком, в котором для работы с вариантами OLE введен специализированный тип данных, представляемый как динамический во время выполнения программы и как статический во время ее компиляции.

В Delphi 3 для этой же цели был введен еще один новый тип данных — OleVariant, полностью идентичный типу Variant, за исключением того, что он может вмещать только те типы, которые являются совместимыми с OLE. Дальше в этом разделе мы подробно рассмотрим тип Variant, а также обсудим те отличия, которые имеются между типами данных OleVariant и Variant.

Динамическое изменение типа

Основное назначение типа Variant — получить возможность определять переменную, тип которой не может быть установлен во время компиляции. Это означает, что реальный тип такой переменной может изменяться в процессе выполнения программы. Например, приведенная ниже программа корректна и будет компилироваться и выполняться без ошибок.

```
var
  V: Variant;               // Переменная типа Variant
begin
  V := 'Delphi 5 is Great!'; // V содержит строку
  V := 1;                   // V содержит целое число
  V := 123.34;              // V содержит вещественное число
  V := True;                // V содержит логическое значение
  V := CreateOleObject('Word.Basic');
                            // V содержит объект OLE
end;
```

Варианты могут поддерживать все простые типы Object Pascal, такие как целые и вещественные числа, строки, значения даты и времени и т.п. Кроме того, они также могут содержать объекты автоматизации OLE. Однако отметим, что они не могут быть ссылками на объекты Object Pascal. Кроме того, варианты могут ссылаться на неоднородные массивы, состоящие из переменного количества элементов, имеющих различный тип (в том числе они могут быть и другими массивами вариантов).

Структура определения данных типа Variant

Структура определения данных типа Variant описана в модуле System и выглядит следующим образом:

```
type
  PVarData = ^TVarData;
  TVarData = packed record
    VType: Word;
    Reserved1, Reserved2, Reserved3: Word;
    case Integer of
      varSmallint: (VSmallint: Smallint);
      varInteger:  (VInteger: Integer);
      varSingle:   (VSingle: Single);
      varDouble:   (VDouble: Double);
      varCurrency: (VCurrency: Currency);
      varDate:     (VDate: Double);
      varOleStr:   (VOleStr: PWideChar);
      varDispatch: (VDispatch: Pointer);
      varError:    (VError: LongWord);
      varBoolean:  (VBoolean: WordBool);
      varUnknown:  (VUnknown: Pointer);
      varByte:     (VByte: Byte);
      varString:   (VString: Pointer);
      varAny:      (VAny: Pointer);
      varArray:    (VArray: PVarArray);
      varByRef:    (VPointer: Pointer);
  end;
```

Структура TVarData занимает 16 байт памяти. Первых два байта этой структуры содержат слово, значение которого определяет, на какой именно тип данных ссылается вариант. Ниже приведены конкретные значения, соответствующие различным типам данных, которые могут помещаться в поле VType записи TVarData. Следующих 6 байт записи не используются. Последних 8 байт содержат либо действительные данные, либо указатель на данные, представляемые этим вариантом. Следует отметить, что данная структура точно соответствует требованиям, предъявляемым к вариантам OLE.

```
{ Коды типов Variant }
const
  varEmpty    = $0000;
  varNull     = $0001;
  varSmallint = $0002;
  varInteger  = $0003;
  varSingle   = $0004;
  varDouble   = $0005;
  varCurrency = $0006;
  varDate     = $0007;
  varOleStr   = $0008;
  varDispatch = $0009;
  varError    = $000A;
  varBoolean  = $000B;
  varVariant  = $000C;
  varUnknown  = $000D;
  varByte     = $0011;
  varStrArg   = $0048;
  varString   = $0100;
  varAny      = $0101;
  varTypeMask = $0FFF;
  varArray    = $2000;
  varByRef    = $4000;
```

На заметку Из приведенного выше списка видно, что Variant не может содержать ссылку на данные типа Pointer или class.

Из описания структуры записи TVarData видно, что она действительно может содержать данные любого типа. Следует отличать приведенную выше запись TVarData от действительных данных типа Variant. Хотя запись переменного состава и данное типа вариант внешне схожи по своему назначению, они представляют две совершенно разные конструкции. Запись TVarData позволяет хранить данные различных типов в одной и той же области памяти (подобно объединениям — union языка C/C++). Подробнее об этом речь пойдет дальше, в разделе "Записи" этой главы. Оператор case в описании записи TVarData служит для определения типов данных, которые может представлять вариант. Так, если в поле VType содержится значение varInteger, то только четыре из восьми байтов данных записи будут содержать хранимое переменной целое значение. Подобным образом, если в поле VType содержится значение varByte, только один из восьми байтов области данных будет использоваться для хранения значения.

Однако, если в поле VType содержится значение varString, то восемь байтов данных в записи содержат не реальную строку, а лишь указатель на нее. Это важно четко понимать, поскольку при работе с вариантами можно иметь непосредственный доступ к значениям любого поля, как показано в приведенном ниже примере.

```
var
  V: Variant;
begin
  TVarData(V).VType    := varInteger;
  TVarData(V).VInteger := 2;
end;
```

Следует отдавать себе отчет в том, что такое использование варианта — опасная игра, так как при этом, например, можно легко разрушить указатель на строку или другой объект с управляемым временем жизни. В результате объект станет недоступным для процессов сборки мусора, что приведет к утечке памяти и других ресурсов приложения. Разговор о процессах сборки мусора пойдет в следующем разделе.

Варианты являются объектами с управляемым временем жизни

Delphi автоматически выделяет и освобождает память для данных типа Variant. Рассмотрим приведенный ниже пример, в котором варианту присваивается строка.

```
procedure ShowVariant(S: string);
var
  V: Variant
begin
  V := S;
  ShowMessage(V);
end;
```

Как уже отмечалось выше в этой главе, в отношении объектов с управляемым временем жизни, при выполнении данной программы осуществляется несколько действий, которые не вполне очевидны. Прежде всего Delphi инициализирует вариант пустым значением. Затем, в ходе присвоения значения, в поле VType помещается значение varString, а в поле VString копируется указатель на строку. При этом счетчик ссылок строки S увеличивается. Когда вариант покидает область видимости (в данном случае — при выходе из процедуры), он очищается и счетчик ссылок строки S уменьшается. Delphi выполняет это посредством помещения тела процедуры в блок try finally, как показано ниже.

```
procedure ShowVariant(S: string);
var
  V: Variant
begin
  V := Unassigned;  // Инициализация варианта
  try
    V := S;
    ShowMessage(V);
  finally
```

```
    // Теперь можно освободить ресурсы, связанные с вариантом
  end;
end;
```

Точно такое же неявное освобождение ресурсов происходит и при присвоении варианту данных нового типа. Рассмотрим следующий пример:

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  V := S;
  V := 34;
end;
```

Эти действия Delphi будет выполнять таким образом (последующий текст написан с помощью псевдокода):

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  Вариант V инициализируется пустым значением
  try
  V.VType := varString; V.VString := S; Inc(S.RefCount);
  Освобождение ресурсов, связанных с вариантом V
  V.VType := varInteger; V.VInteger := 34;
  finally
    Освобождение ресурсов, связанных с вариантом V
  end;
end;
```

Если вы разобрались в том, как работает приведенный выше пример, то сможете понять, почему не рекомендуется непосредственно манипулировать полями записи `TVarData`, как в данном случае:

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  V := S;
  TVarData(V).VType    := varInteger;
  TVarData(V).VInteger := 32;
  V := 34;
end;
```

Хотя все кажется вполне корректным, это далеко не так — ведь уменьшение счетчика ссылок строки `S` не выполняется, что приводит к утечке памяти. Одним словом, никогда не работайте с полями `TVarData` напрямую, а если уж вы делаете это, то должны четко понимать, что именно вы делаете и какие последствия это может вызвать.

Преобразование типов для вариантов

Можно явно преобразовать выражение к типу Variant. Так, выражение Variant(X) дает в результате вариант, тип которого соответствует результату выражения X. Последнее должно быть целым, вещественным, строкой, символом, денежным или булевым типом.

Можно также преобразовывать тип Variant к другим типам. Рассмотрим вариант V := 1.6 и различные возможные для него преобразования типов:

```
S := string(V);   // S будет содержать строку '1.6';
I := Integer(V);  // I будет равным ближайшему целому, т.е. 2
B := Boolean(V);  // B содержит False, если V содержит 0, иначе B равно True
D := Double(V);   // D получает значение 1.6
```

Результаты всех этих преобразований получены в соответствии с определенными правилами приведения типов, применимых к переменным типа Variant. Детально преобразования типов описаны в руководстве Delphi "Object Pascal Language Guide".

Заметим, что в предыдущем примере явное преобразование типов не является необходимым. Приведенный ниже код по своей функциональности идентичен предыдущему.

```
V := 1.6;
S := V;
I := V;
B := V;
D := V;
```

В этом примере преобразование типов происходит неявно. Однако за счет того, что преобразование осуществляется непосредственно в процессе выполнения программы, в ее состав включается соответствующая логика. Какой метод предпочесть, зависит от вас. Если тип реальных данных варианта вам известен на стадии компиляции, лучше использовать явное преобразование типов, которое более эффективно (так как выполняется на стадии компиляции). Особенно справедливо это замечание в тех случаях, когда вариант используется в выражениях, речь о которых пойдет ниже.

Использование вариантов в выражениях

Вы можете использовать варианты в выражениях, составленных с использованием следующих операторов: +, -, =, *, /, div, mod, shl, shr, and, or, xor, not, :=, <>, <, >, <=, >=.

При использовании вариантов в выражениях Delphi принимает решение о том, как должны выполняться операторы, на основании текущего типа содержимого варианта. Например, если варианты V1 и V2 содержат целые числа, то результатом выражения V1+V2 станет их сумма. Если же они содержат строки, то результатом будет их объединение. А что произойдет, если типы данных различны? В этом случае Delphi использует некоторые правила с целью определения того, какие преобразования должны быть выполнены. Так, если V1 содержит строку '4.5', а V2 — вещественное число, то V1 будет конвертировано в число 4.5 и сложено со значением V2. Рассмотрим следующий пример:

```
var
  V1, V2, V3: Variant;
begin
  V1 := '100';  // Строка
```

```
  V2 := '50';   // Строка
  V3 := 200;    // Целое
  V1 := V1 + V2 + V3;
end;
```

Исходя из сделанного выше замечания можно предположить, что в результате будет получено целое значение `350`. Однако это не так. Поскольку вычисление выражений выполняется слева направо, при первом сложении складываются две *строки* и в результате должна получиться тоже *строка*, имеющая значение `'10050'`. А уже затем полученный результат будет преобразован в целое значение и просуммирован с третьим целочисленным операндом, в результате чего будет получено значение `10250`.

В Delphi для успешного выполнения вычислений данные типа `Variant` всегда преобразуются к самому высокому типу данных, присутствующих в выражении. Однако, когда в операции участвуют два варианта, для которых Delphi не в состоянии подобрать подходящий тип, генерируется исключительная ситуация невозможности преобразования типа варианта. Вот простейший пример такой ситуации:

```
var
  V1, V2: Variant;
begin
  V1 := 77;
  V2 := 'hello';
  V1 := V1 / V2;  // Генерируется исключительная ситуация
end;
```

Неплохая идея при использовании преобразования типов варианта — задание явного преобразования во время компиляции. Так, в случае даже такой обычной операции, как `V4 := V1*V2/V3`, в процессе выполнения Delphi рассматривает множество вариантов преобразования типов для того, чтобы найти наиболее подходящий. Явное же указание типов, например `V4 := Integer(V1)*Double(V2)/Integer(V3)`, позволяет принять решение еще на стадии компиляции, избежав тем самым излишних затрат времени при работе программы. Правда, при этом, как уже упоминалось, вы должны точно знать, какой тип данных содержится в варианте.

Пустое значение и значение `Null`

Следует отдельно обсудить два специальных значения поля `VType`. Первое — `varEmpty` — означает, что варианту пока не назначено никакого значения. Это начальное значение варианта, которое компилятор устанавливает при входе переменной в область видимости. Второе значение — `varNull` — отличается от `varEmpty` тем, что оно представляет реально существующее значение переменной, которое равно `Null`. Это отличие особо важно при работе с базами данных, где отсутствие значения и значение `Null` — абсолютно разные вещи. В главе 27 второго тома, "Разработка приложений CORBA в Delphi", вы узнаете о применении вариантов в контексте приложений для работы с базами данных.

Еще одно отличие этих значений состоит в том, что любая попытка вычисления выражений с пустым вариантом будет приводить к возникновению исключительной ситуации: "некорректная операция с вариантом". Однако этого не будет при использовании в выражении варианта со значением `Null` — результат вычисления любого выражения, в состав которого входит значение `Null`, всегда будет равен `Null`.

Если потребуется присвоить или сравнить вариант с одним из этих специальных значений, то для этого в модуле System имеется два предопределенных варианта — Unassigned и Null, у которых поля VType соответственно имеют значения varEmpty и varNull.

За все в этой жизни приходится расплачиваться, и варианты — не исключение. Удобство работы и высокая гибкость достигаются ценой увеличения размера и замедления работы вашего приложения. Кроме того, повышается сложность сопровождения создаваемого программного обеспечения. Естественно, бывают ситуации, когда без вариантов трудно обойтись. В частности, благодаря их гибкости, они достаточно широко применяются в визуальных компонентах, особенно в элементах управления ActiveX и компонентах для работы с базами данных. Тем не менее в большинстве случаев рекомендуется работать с обычными типами данных. Старайтесь использовать варианты только в тех ситуациях, когда без них действительно нельзя обойтись и когда увеличение размера и замедление работы приложения — разумная плата за гибкость. Не забывайте, что использование неоднозначных типов данных приводит к появлению неоднозначных ошибок.

Варианты-массивы

Ранее упоминалось, что варианты могут представлять гетерогенные массивы. Так, следующий код синтаксически абсолютно корректен:

```
var
    V: Variant;
    I, J: Integer;
begin
    I := V[J];
end;
```

Однако, хотя код и скомпилируется без ошибки, при его выполнении будет сгенерирована исключительная ситуация, так как вариант V не содержит массива. Object Pascal предоставляет несколько функций, позволяющих создать вариант-массив, в частности, функций VarArrayCreate() и VarArrayOf().

Функция VarArrayCreate()

Функция VarArrayCreate() определена в модуле System следующим образом:

```
function VarArrayCreate(const Bounds: array of Integer;
  VarType: Integer): Variant;
```

При использовании функции в качестве параметров ей передаются границы создаваемого массива и код типа варианта элементов создаваемого массива. (Первый параметр представляет собой открытый массив, речь о котором пойдет ниже, в разделе “Передача параметров” этой же главы.) Например, в приведенном ниже фрагменте кода создается вариант-массив целых чисел, после чего его членам присваиваются требуемые значения.

```
var
  V: Variant;
begin
  V := VarArrayCreate([1, 4], varInteger);  // Создание массива из 4 элементов
  V[1] := 1;
  V[2] := 2;
```

```
  V[3] := 3;
  V[4] := 4;
end;
```

Однако это еще не все — передав в качестве кода тип varVariant, вы создаете вариант-массив вариантов. Это — массив, элементы которого могут содержать значения разных типов. Кроме того, точно так же можно создавать и многомерные массивы, для чего достаточно просто указать дополнительные значения границ. Например, в приведенном ниже фрагменте кода создается двухмерный вариант-массив размерностью [1..4, 1..5].

```
V := VarArrayCreate([1, 4, 1, 5], varInteger);
```

Функция VarArrayOf()

Эта функция определена в модуле System следующим образом:

```
function VarArrayOf(const Values: array of Variant): Variant;
```

Она возвращает одномерный массив, элементы которого были заданы в параметре Values. Ниже приведен пример создания массива из трех элементов: целого, строки и вещественного числа.

```
V := VarArrayOf([1, 'Delphi', 2.2]);
```

Функции и процедуры для работы с вариантами-массивами

Помимо функций VarArrayCreate() и VarArrayOf(), имеется еще ряд других функций и процедур, предназначенных для работы с вариантами-массивами. Они определены в модуле System следующим образом:

```
procedure VarArrayRedim (var A: Variant; HighBound: Integer);
function  VarArrayDimCount(const A: Variant): Integer;
function  VarArrayLowBound(const A: Variant; Dim: Integer): Integer;
function  VarArrayHighBound(const A: Variant; Dim: Integer): Integer;
function  VarArrayLock(const A: Variant): Pointer;
procedure VarArrayUnlock(const A: Variant);
function  VarArrayRef(const A: Variant): Variant;
function  VarIsArray (const A: Variant): Boolean;
```

Функция VarArrayRedim() позволяет изменять верхнюю границу размерности самого варианта-массива. Функция VarArrayDimCount() возвращает размерность варианта-массива, а функции VarArrayLowBound() и VarArrayHighBound() — соответственно возвращают нижнюю и верхнюю границы варианта-массива. Специализированные функции VarArrayLock() и VarArrayUnlock() подробнее обсуждаются в следующем разделе.

Функция VarArrayRef() необходима для передачи вариантов-массивов серверу автоматизации OLE. Проблемы возникают в том случае, если в метод автоматизации в качестве параметра передается вариант, содержащий вариант-массив. Например:

```
Server.PassVariantArray(VA)
```

Между передачей варианта-массива и передачей варианта, содержащего вариант-массив, имеются существенные различия. Если серверу передать вариант-массив, а не ссылку на подобный объект, то при использовании приведенной выше записи сервер выдаст сообщение об ошибке. Функция `VarArrayRef()` предназначена для приведения передаваемого значения к виду и типу, ожидаемому сервером:

```
Server.PassVariantArray(VarArrayRef(VA))
```

Функция `VarIsArray()` представляет собой простую проверку, возвращающую значение `True`, если передаваемый ей параметр представляет собой массив.

Инициализация большого массива с помощью функции `VarArrayLock()` и процедуры `VarArrayUnlock()`

Варианты-массивы широко используются в средствах автоматизации OLE, так как они представляют собой единственный способ передачи серверу автоматизации произвольных двоичных данных. (Отметим, что указатели не являются допустимым типом данных в среде автоматизации OLE (детальнее об этом речь пойдет в главе 23 второго тома, "COM-ориентированные технологии").) Однако при неверном использовании варианты-массивы могут оказаться весьма неэффективным средством обмена данными. Рассмотрим небольшой пример:

```
V := VarArrayCreate([1, 10000], VarByte);
```

Здесь создается вариант-массив размером в 10 000 байт. Предположим, что существует другой, обычный, массив того же размера и необходимо скопировать его содержимое в созданный вариант-массив. Естественный путь такого копирования — цикл, подобный тому, который приведен ниже.

```
begin
  V := VarArrayCreate([1, 10000], VarByte);
  for i := 1 to 10000 do
    V[i] := A[i];
end;
```

Проблема при этом заключается в огромной и совершенно излишней работе программы по определению типа и совместимости каждого элемента, его инициализации и др. Суть в том, что присвоение значений элементам массива осуществляется во время выполнения программы Для исключения выполнения всех ненужных проверок при работе программы предназначены функция `VarArrayLock()` и процедура `VarArrayUnlock()`.

Функция `VarArrayLock()` блокирует массив в памяти так, что нельзя будет переместить его или изменить его размеры. Она возвращает указатель на данные массива. Процедура `VarArrayUnlock()` деблокирует массив, разрешая его перемещение в памяти и изменение его размеров. После блокировки массива для заполнения его данными можно воспользоваться более эффективными методами, например, такими как процедура `Move()`. Приведенный ниже пример позволяет инициализировать тот же массив значительно более эффективным методом.

```
begin
  V := VarArrayCreate([1, 10000], VarByte);
  P := VarArrayLock(V);
  try
    Move(A, P^, 10000);
```

```
  finally
    VarArrayUnlock(V);
  end;
end;
```

Функции для работы с вариантами

Существует еще ряд функций и процедур, предназначенных для работы с вариантами. Ниже приведены их определения в модуле System.

```
procedure VarClear(var V: Variant);
procedure VarCopy(var Dest: Variant; const Source: Variant);
procedure VarCast(var Dest: Variant; const Source: Variant; VarType: Integer);
function  VarType(const V: Variant): Integer;
function  VarAsType(const V: Variant; VarType: Integer): Variant;
function  VarIsEmpty(const V: Variant): Boolean;
function  VarIsNull(const V: Variant): Boolean;
function  VarToStr(const V: Variant): string;
function  VarFromDateTime(DateTime: TDateTime): Variant;
function  VarToDateTime(const V: Variant): TDateTime;
```

Процедура VarClear() очищает вариант и помещает в поле VType значение varEmpty. Процедура VarCopy() копирует вариант Source в вариант Dest. Процедура VarCast() предназначена для преобразования варианта к некоторому типу и сохранения результата в другом варианте. Функция VarType() возвращает значение кода varXXXX для заданного варианта. Функция VarAsType() имеет то же самое назначение, что и процедура VarCast(). Функция VarIsEmpty() возвращает значение True, если поле VType варианта равно varEmpty, а функция VarIsNull() возвращает это же значение, если данное поле равно varNull. Функция VarToStr() конвертирует вариант в строку (пустую, если вариант пуст, или нулевой).

С помощью функции VarFromDateTime() создается вариант, содержащий заданное значение типа TDateTime. Функция VarToDateTime() возвращает значение типа TDateTime, содержащееся в указанном варианте.

Тип данных OleVariant

Тип данных OleVariant практически во всем идентичен рассмотренному выше типу Variant, за одним исключением — он допускает только те типы данных, которые совместимы со средствами автоматизации OLE. В настоящее время единственное отличие проявляется при работе со строками (тип varString предназначен для строк типа AnsiString). В то время как тип Variant работает с подробными строками, при присвоении строкового значения типа AnsiString переменной типа OleVariant происходит автоматическое конвертирование этой строки в тип OLE BSTR и сохранение ее в варианте как тип varOleStr.

Тип данных Currency

Этот тип впервые был введен в Delphi 2 и представляет собой десятичное число с фиксированной точкой, имеющее 15 значащих цифр до десятичной точки и 4 после. Этот формат идеально подходит для финансовых вычислений, поскольку свободен от ошибок ок-

ругления, свойственных операциям с плавающими числами. Настоятельно рекомендуем изменить все участвующие в финансовых расчетах переменные типов `Single`, `Real`, `Double` и `Extended` на тип `Currency`.

Пользовательские типы данных

Таких типов, как целые, строки и вещественные числа, зачастую недостаточно для адекватного представления данных, с которыми приходится иметь дело при решении реальных задач. Часто необходимо использовать и другие типы данных, более точно отражающие реальную действительность, моделируемую конкретной программой. В Object Pascal подобные пользовательские типы данных обычно принимают вид записей или объектов. Объявление этих типов производится с помощью ключевого слова `Type`.

Массивы

Object Pascal позволяет создавать массивы переменных любого типа (кроме файлов). Например, ниже объявляется переменная, представляющая собой массив из восьми целых чисел.

```
var
  A: Array[0..7] of Integer;
```

Такой оператор эквивалентен следующему объявлению в языке C:

```
int A[8];
```

Соответствующий оператор языка Visual Basic выглядит следующим образом:

```
Dim A(8) as Integer
```

Массивы Object Pascal имеют одно существенное отличие от массивов языков C или Visual Basic (и многих других языков) — они не должны начинаться с определенного номера элемента. Например, можно определить массив из трех элементов, начинающийся с элемента под номером 28:

```
var
  A: Array[28..30] of Integer;
```

Поскольку в Object Pascal элементы массива не обязательно начинаются с нулевого или первого элемента, следует принимать необходимые меры при организации итераций, например, в цикле `for`. Для этого компилятор предоставляет две встроенные функции — `High()` и `Low()`, возвращающие верхнюю и нижнюю границы заданного массива. Программа будет более устойчива к ошибкам и возможным изменениям описания массива, если в ней будут использоваться эти функции. Например:

```
var
  A: array[28..30] of Integer;
  i: Integer;
begin
  for i := Low(A) to High(A) do  // Не используйте в параметрах цикла
                                 // конкретных числовых значений!
    A[i] := i;
end;
```

Всегда начинайте массивы символов с нулевого элемента — такие массивы могут быть переданы в качестве параметров функциям, использующим параметры с типом PChar. Это — особая дополнительная возможность, предоставляемая компилятором.

Для определения многомерных массивов в описании используется перечисление описания размерностей, отделяемых запятыми:

```
var
  // Двухмерный массив целых чисел
  A: Array[1..2, 1..2] of Integer;
```

Для доступа к элементам такого массива используются индексы, задаваемые через запятую:

```
I := A[1,2];
```

Динамические массивы

Динамические массивы — это массивы, память для которых выделяется динамически и размерность которых не известна заранее, во время компиляции. Для объявления такого массива используется обычное описание, но без указания размерности массива:

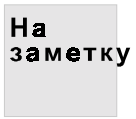
```
var
  // Динамический массив строк
  SA: array of string;
```

Перед использованием такого массива необходимо определить его размер с помощью процедуры SetLength(), которая выполнит распределение памяти для массива:

```
begin
  // Выделение памяти для 33 элементов
  SetLength(SA, 33);
```

После этого можно работать с элементами данного массива так же, как с элементами любого другого массива:

```
SA[0] := 'Pooh likes hunny';
OtherString := SA[0];
```



Динамические массивы всегда начинаются с нулевого элемента.

Динамические массивы — это тип данных с управляемым временем жизни, поэтому можно не заботиться о своевременном освобождении выделенной им памяти. Она будет освобождена автоматически, когда данная переменная покинет область видимости. Однако это не значит, что нельзя самостоятельно освободить память при необходимости (например, если массив использовал слишком большое ее количество). Для этого достаточно просто присвоить массиву значение nil:

```
SA := nil;    //Освобождение выделенной массиву памяти
```

Для создания динамических массивов используется технология ссылок (как, например, в случае типа AnsiString). Вот маленький тест для вас: "Чему равен элемент A1[0] после выполнения приведенного ниже фрагмента?"

```
var
  A1, A2: array of Integer;
begin
  SetLength(A1, 4);
  A2 := A1;
  A1[0] := 1;
  A2[0] := 26;
```

Правильный ответ — 26. Поскольку присвоение A2 := A1 приводит к тому, что и A1 и A2 ссылаются на один и тот же массив в памяти, то изменение элемента массива A2 приводит к изменению соответствующего элемента массива A1 (впрочем, более точным будет утверждение, что это просто один и тот же элемент). Если же требуется создать именно копию массива, воспользуйтесь стандартной процедурой Copy():

```
A2 := Copy(A1);
```

После этого будет создана копия массива, изменения в которой не оказывают никакого влияния на данные в исходном массиве. Можно также скопировать не весь массив, а только некоторый диапазон его элементов, как показано в приведенном примере, где копируются два элемента, начиная с первого:

```
A2 := Copy(A1, 1, 2);  // Копируются два элемента, начиная с первого
```

Динамические массивы могут быть многомерными. Для определения такого массива добавьте дополнительное описание array of для каждой дополнительной размерности:

```
var
  // Двухмерный динамический массив целых чисел
  IA: array of array of Integer;
```

При выделении памяти для многомерного динамического массива функции SetLength() следует передать дополнительный параметр:

```
begin
  // IA будет массивом целых чисел размерностью 5 x 5
  SetLength(IA, 5, 5);
```

Обращение к элементам многомерного динамического массива ничем не отличается от обращения к элементам обычного массива:

```
IA[0,3] := 28;
```

Записи

Структуры, определяемые пользователем, в Object Pascal называются *записями* (record). Они эквивалентны типам данных struct в языке C или Type — в языке Visual Basic:

```
{ Pascal }
Type
  MyRec = record
    i: Integer;
```

```
    d: Double;
  end;

/* C */
typedef struct {
  int i;
  double d;
} MyRec;

'Visual Basic
Type MyRec
  i As Integer
  d As Double
End Type
```

При работе с записями доступ к их полям обеспечивается с помощью точки:

```
var
  N: MyRec;
begin
  N.i := 23;
  N.d := 3.4;
end;
```

Object Pascal также поддерживает *вариантные записи* (*variant records*), которые обеспечивают хранение разнотипных данных в одной и той же области памяти. Не путайте эти записи с рассмотренным выше типом `Variant` — вариантные записи позволяют независимо получать доступ к каждому из перекрывающихся полей данных. Если вы знакомы с языком C или C++, то можете понимать вариантные записи как аналог концепции `union` в структурах языка C или C++. Приведенный ниже код показывает вариантную запись, в которой поля типа `Double`, `Integer` и `Char` занимают одну и ту же область памяти.

```
type
  TVariantRecord = record
    NullStrField: PChar;
    IntField: Integer;
    case Integer of
      0: (D: Double);
      1: (I: Integer);
      2: (C: char);
  end;
```

На заметку Вариантная часть записи не может быть представлена типом данных с управляемым временем жизни.

Вот эквивалент приведенного кода в языке C++:

```
struct TUnionStruct
{
  char * StrField;
```

```
  int IntField;
  union
  {
    double D;
    int i;
    char c;
  } ;
} ;
```

Множества

Множества — уникальный для языка Pascal тип данных, который не имеет аналогов в языках Visual Basic, C или C++ (хотя в Borland C++ Builder реализован шаблонный класс `Set`, эмулирующий поведение множеств в Pascal). Множества обеспечивают эффективный способ представления коллекций чисел, символов или других перечислимых значений. Новый тип множества можно определить с помощью ключевого слова `set of` с указанием перечислимого типа или диапазона в некотором множестве допустимых значений:

```
type
  TCharSet = set of char; // Допустимые элементы: #0 - #255
  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday);
  TEnumSet = set of TEnum;  // Может содержать любую комбинацию членов TEnum
  TSubrangeSet = set of 1..10;  // Допустимые элементы: 1 - 10
  TAlphaSet = set of 'A'..'z'; // Допустимые элементы: 'A' - 'z'
```

Заметим, что множество может содержать только до 256 элементов. Кроме того, в множествах после ключевого слова `set of` могут указываться только перечислимые типы данных. Таким образом, следующие объявления некорректны:

```
type
  TIntSet = set of Integer;  // Слишком много элементов
  TStrSet = set of string;  // Неперечислимый тип данных
```

Внутренне элементы множеств хранятся как отдельные биты, что делает их весьма эффективными в плане скорости обработки и использования памяти. Множества, насчитывающие менее чем 32 базовых элемента, могут храниться и обрабатываться в регистрах процессора, что способствует еще большей эффективности. Множества с 32 (или более) элементами (например, множество символов `char` из 255 элементов) хранятся в памяти. Поэтому для достижения максимальной скорости обработки целесообразно определять множества не более чем с 32 базовыми элементами.

Использование множеств

Для ссылки на элемент множества используются квадратные скобки. Приведенный ниже пример демонстрирует объявление множества и присвоение значений его элементам.

```
type
  TCharSet = set of char; // Допустимые члены: #0 - #255
  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
  TEnumSet = set of TEnum;  // Может содержать любую комбинацию членов TEnum
```

```
var
  CharSet: TCharSet;
  EnumSet: TEnumSet;
  SubrangeSet: set of 1..10;  // Допустимые члены: 1 - 10
  AlphaSet: set of 'A'..'z';  // Допустимые члены: 'A' - 'z'
begin
  CharSet := ['A'..'J', 'a', 'm'];
  EnumSet := [Saturday, Sunday];
  SubrangeSet := [1, 2, 4..6];
  AlphaSet := [];  // Пустое множество
end;
```

Операторы работы с множествами

Object Pascal предоставляет несколько операторов для работы с множествами (например, для определения принадлежности к множеству, добавления или удаления элементов и пересечения множеств).

Проверка принадлежности к множеству

Оператор `in` используется для определения, входит ли данный элемент в состав того или иного множества. Например, следующий код используется для определения того, содержится ли символ `'S'` в множестве `CharSet`:

```
if 'S' in CharSet then
  // Некоторые действия;
```

В приведенном ниже примере осуществляется проверка отсутствия члена `Monday` в множестве `EnumSet`.

```
if not (Monday in EnumSet) then
  // Некоторые действия;
```

Добавление и удаление элементов множеств

Операторы `+` и `-` или процедуры `Include()` и `Exclude()` могут быть использованы для добавления или удаления элементов множеств:

```
Include(CharSet, 'a');            // Добавление 'a' в множество
CharSet := CharSet + ['b'];       // Добавление 'b' в множество
Exclude(CharSet, 'x');            // Удаление 'x' из множества
CharSet := CharSet - ['y', 'z'];  // Удаление 'y' и 'z' из множества
```

Везде, где только это возможно, для добавления или удаления элементов из множества вместо операторов `+` и `-` используйте процедуры `Include()` и `Exclude()`. Каждая из этих процедур реализуется одной машинной командой, в то время как для реализации операторов `+` и `-` требуется по `13+6n` команд (здесь `n` — количество битов в множестве).

Пересечение множеств

Оператор `*` используется для вычисления пересечения двух множеств. В результате операции `Set1 * Set2` получается множество с элементами, содержащимися в обоих множест-

вах одновременно. Приведенный ниже код является эффективным способом определения того, содержит ли множество несколько заданных элементов.

```
if ['a', 'b', 'c'] * CharSet = ['a', 'b', 'c'] then
  // Некоторые действия
```

Объекты

Объекты в Object Pascal можно представить как записи, которые помимо данных содержат функции и процедуры. Но поскольку объектная модель Delphi детально обсуждается в разделе "Использование объектов Delphi" настоящей главы, здесь мы рассмотрим только основы синтаксиса объявления объектов Object Pascal. Объект объявляется следующим образом:

```
Type
  TChildObject = class(TParentObject);
    SomeVar: Integer;
    procedure SomeProc;
  end;
```

Хотя объекты Delphi не совсем идентичны объектам языка C++, это объявление очень близко к описанию, используемому в языке C++:

```
class TChildObject : public TParentObject
{
  int SomeVar;
  void SomeProc();
};
```

Методы определяются так же, как и обычные процедуры и функции (о которых речь пойдет позднее, в разделе "Процедуры и функции" этой же главы). Единственное отличие — это добавление имени объекта и точки перед именем метода:

```
procedure TChildObject.SomeProc;
begin
  { код процедуры }
end;
```

Символ точки в Object Pascal по своей функциональности похож на оператор "**.**" в языке Visual Basic или на оператор "**::**" в языке C++. Следует отметить, что, хотя все три языка позволяют использовать классы, только Object Pascal и C++ позволяют создавать новые классы, полностью соответствующие парадигме объектно-ориентированного программирования.

На заметку

Размещение объектов Object Pascal в памяти отличается от размещения классов C++, и поэтому невозможно использовать объекты C++ непосредственно в Delphi или наоборот. В главе 13, "Дополнительный инструментарий разработчика" показана технология разделения объектов между C++ и Delphi.
Исключением являются описанные как `__declspec(delphiclass)` классы Borland C++ Builder, которые совместимы с объектами Delphi (но не совместимы с обычными классами C++).

Указатели

Указатель (pointer) представляет собой переменную, содержащую местоположение (адрес) ячейки памяти. С примером указателя мы уже сталкивались в этой главе, когда рассматривали тип PChar. Общий тип указателя в Object Pascal имеет название Pointer — нетипизированный указатель, который содержит адрес некоего места в памяти, при этом компилятору ничего не известно о данных, располагающихся по этому адресу. Однако применение таких "указателей вообще" противоречит концепции строгого контроля типов, а потому вам в основном придется иметь дело с типизированными указателями, т.е. с указателями на данные конкретного типа.

На заметку Указатели — тема достаточно сложная для новичка, и можно писать приложения Delphi, не будучи досконально с ней знакомым. Однако, по мере ее освоения указатели могут стать одним из самых мощных инструментов программирования, доступных в Delphi.

Типизированные указатели объявляются в разделе Type программы с использованием символа ^ (или ключевого слова Pointer) — оператора указателя. Типизированные указатели позволяют компилятору отслеживать, с какими данными вы работаете и не выполняете ли вы над ними некорректные действия. Вот примеры объявлений типизированных указателей:

```
Type
  PInt = ^Integer;   // PInt - указатель на Integer
  Foo = record       // Тип - запись
    GobbledyGook: string;
    Snarf: Real;
  end;
  PFoo = ^Foo;        // PFoo - указатель на объект типа foo
var
  P: Pointer;         // Нетипизированный указатель
  P2: PFoo;           // Указатель на экземпляр Foo
```

На заметку Программисты на языке C++ могут заметить схожесть оператора ^ Object Pascal и оператора * языка C++. Тип Pointer в Object Pascal соответствует типу void * в языке C.

Запомните, что переменная типа Pointer всегда содержит только адрес памяти. О выделении памяти для той структуры, на которую будет указывать указатель, должен позаботиться сам программист (для этого можно воспользоваться одной из функций, перечисленных в табл. 2.6).

На заметку Когда указатель не указывает ни на какое место в памяти, его значение равно 0. При этом говорят, что его значение равно Nil, а сам указатель — нулевой.

Получить доступ к данным, на которые указатель указывает, можно с помощью оператора ^, следующего за именем этой переменной. Этот метод называется "разрешением указателя". Ниже приведен пример работы с указателями.

```
Program PtrTest;
Type
  MyRec = record
```

```
    I: Integer;
    S: string;
    R: Real;
  end;
  PMyRec = ^MyRec;
var
  Rec : PMyRec;
begin
  New(Rec);      // Выделяем память для новой записи Rec
  Rec^.I := 10;  // Помещаем в нее некоторые данные
                 // Обратите внимание на оператор ^
  Rec^.S := 'Теперь помещаем в нее данные другого типа';
  Rec^.R := 6.384;
  { Запись Rec заполнена }
  Dispose(Rec);  // Не забывайте освобождать память!
end.
```

Когда следует использовать функцию New()

Функция New() используется при выделении памяти для указателя на структуру данных известного размера. Так как компилятору известен размер структуры, для которой требуется выделить память, при выполнении функции New() будет распределено требуемое количество байтов, причем такой способ выделения более корректен и безопасен, чем вызов функции GetMem() или AllocMem(). В тоже время, никогда не используйте функцию New() для выделения памяти для типов Pointer или PChar, так как в этом случае компилятору не известно, какое количество памяти должно быть выделено. И не забывайте использовать функцию Dispose() для освобождения памяти, выделенной с помощью функции New().
Для выделения памяти структурам, размер которых на этапе компиляции еще не известен, используются функции GetMem() и AllocMem(). Например, компилятор не может определить заранее, сколько памяти потребуется выделить для структур, задаваемых переменными типа PChar или Pointer, что связано с самой природой этого типа данных. Самое важное — не пытаться манипулировать количеством памяти, большим, чем было выделено реально, поскольку наиболее вероятным результатом таких действий будет ошибка доступа к памяти (Access Violation). Для освобождения памяти, выделенной с помощью упомянутых выше функций, используйте функцию FreeMem(). Кстати, для распределения памяти лучше пользоваться функцией AllocMem(), так как она всегда инициализирует выделенную память нулевыми значениями.

Один из аспектов работы с указателями в Object Pascal, который существенно отличается от работы с ними в языке C, — это их строжайшая типизация. Так, в приведенном ниже примере переменные a и b несовместимы по типу.

```
var
  a: ^Integer;
  b: ^Integer;
```

В то же время в эквивалентном описании на языке C эти переменные вполне совместимы по типу:

```
int *a;
int *b;
```

Object Pascal создает уникальный тип для каждого объявления указателя на тип, поэтому для совместимости по типу следует объявить не только переменные, но и их тип:

```
type
  PtrInteger = ^Integer;  // Создаем тип данных

var
  a, b: PtrInteger;       // Теперь a и b совместимы по типу
```

Псевдонимы типов

Object Pascal позволяет давать новые имена уже имеющимся типам — создавать их псевдонимы (aliases). Так, если потребуется присвоить новое имя `MyReallyNiftyInteger` обычному типу `Integer`, можно использовать такой код:

```
type
  MyReallyNiftyInteger = Integer;
```

Новый тип совместим с оригиналом. Это означает, что везде, где вы использовали `Integer`, вы можете применять `MyReallyNiftyInteger`.

Но можно создать и строго типизированный псевдоним (т.е. псевдоним, не совместимый с оригиналом). Для этого используется дополнительное ключевое слово `type`.

```
type
  MyOtherNeatInteger = type Integer;
```

При этом новый тип будет преобразовываться в оригинальный при таких операциях, как, например, присвоение, однако как параметр он будет несовместим с оригиналом. Следующий код синтаксически корректен:

```
var
  MONI: MyOtherNeatInteger;
  I: Integer;
begin
  I := 1;
  MONI := I;
```

Однако пример, приведенный ниже, вызовет появление ошибки несовместимости типов:

```
procedure Goon(var Value: Integer);
begin
  // некий программный текст
end;

var
  M: MyOtherNeatInteger;
begin
  M := 29;
  Goon(M);  // Ошибка: тип переменной M не совместим с Integer
```

Помимо усиленного контроля за совместимостью типа данных, для строго типизированных псевдонимов компилятор также генерирует информацию о типе во время выполнения программы. Это позволяет создавать уникальные редакторы свойств для простых типов, о чем подробнее рассказывается в главе 22 второго тома, “Сложные методики работы с компонентами”.

Приведение и преобразование типов

Приведение типов — это технология, посредством которой компилятор можно заставить рассматривать переменную одного типа, как переменную некоторого другого типа. Pascal является строго типизированным языком, который весьма требователен в отношении соответствия типов формальных параметров функции и реальных параметров, задаваемых при ее вызове. Поэтому в некоторых случаях переменную одного типа необходимо привести к некоторому другому типу, чтобы требования компилятора были удовлетворены. Предположим, необходимо присвоить символьное значение переменной типа byte:

```
var
  c: char;
  b: byte;
begin
  c := 's';
  b := c;   // Для этой строки компилятор выдаст сообщение об ошибке
end.
```

В приведенном выше примере необходимо преобразовать переменную c к типу byte. Фактически, выполнение приведения типа явно указывает компилятору, что программист точно знает, что он делает, требуя преобразовать один тип данных в другой:

```
var
  c: char;
  b: byte;
begin
  c := 's';
  b := byte(c);  // Теперь компилятор не обнаружит ошибки в этой строке
end.
```

На заметку

Вы можете использовать приведение типов в том случае, если размеры данных обоих типов одинаковы. Так, нельзя привести тип Double в тип Integer — для подобного преобразования придется воспользоваться функцией Trunc() или Round(). Однако для преобразования целого в вещественное число можно просто использовать оператор присвоения: FloatVar := IntVar.

Object Pascal также предоставляет очень широкие возможности по приведению типов между объектами с использованием оператора as. Подробно эти возможности описаны ниже, в разделе "Информация о типе времени выполнения" настоящей главы.

Строковые ресурсы

В Delphi 3 появилась возможность внесения строковых ресурсов непосредственно в исходный код языка Object Pascal, для чего используется выражение resourcestring. Строковые ресурсы представляют собой литеральные строки (обычно это — сообщения программы пользователю), которые физически расположены в присоединенных к приложению или биб-

лиотеке ресурсах, а не внедрены в исходный текст программы. В частности, подобное отделение строк от исходного кода упрощает перевод приложения на другой язык — для этого достаточно просто присоединить к приложению строковые ресурсы на необходимом языке, без ретрансляции самого приложения. Строковые ресурсы описываются в виде пар значений *идентификатор = строковый литерал*, как показано ниже.

```
resourcestring
  ResString1 = 'Resource string 1';
  ResString2 = 'Resource string 2';
  ResString3 = 'Resource string 3';
```

В программе строковые ресурсы используются так же, как и обычные строковые константы:

```
resourcestring
  ResString1 = 'hello';
  ResString2 = 'world';

var
  String1: string;

begin
  String1 := ResString1 + ' ' + ResString2;
  .
  .
  .
end;
```

Условные операторы

В этом разделе конструкции `if` и `case` языка Object Pascal сравниваются с соответствующими конструкциями языков C и Visual Basic. Предполагается, что читатель уже знаком с подобными конструкциями, а потому мы не будем тратить время на подробное объяснение их назначения и функционирования.

Оператор условного перехода `if`

Оператор `if` позволяет проверить, выполняется ли некоторое условие, и, в зависимости от результатов этой проверки, выполнить тот или иной блок кода. В качестве примера ниже приведен простейший фрагмент кода с использованием оператора `if` и его аналоги на языках C и Visual Basic:

```
{ Pascal }
if x = 4 then y := x;
/* C */
if (x == 4) y = x;
'Visual Basic
If x = 4 Then y = x
```

На заметку

При наличии оператора `if` с несколькими условиями поместите каждое сравнение в скобки — как минимум, это сделает ваш код более ясным и удобочитаемым. Например:

```
if (x = 7) and (y = 8) then...
```

Избегайте записи, подобной следующей (поскольку это может вызвать осложнения в работе транслятора):

```
if x = 7 and y = 8 then...
```

В текстах на языке Pascal ключевые слова `begin` и `end` можно использовать в качестве операторных скобок для определения блока кода — аналогично заданию фигурных скобок в операторе `if` языков C и C++. Вот пример использования такой конструкции:

```
if x = 6 then begin
  DoSomething;
  DoSomethingElse;
  DoAnotherThing;
end;
```

Кроме того, проверку нескольких условий можно комбинировать с помощью конструкции `if else`, как показано на следующем примере:

```
if x =100 then
  SomeFunction
else if x = 200 then
  SomeOtherFunction
else begin
  SomethingElse;
  Entirely;
end;
```

Оператор case

Оператор `case` в языке Pascal подобен операторам `switch` в языках C или C++. Он позволяет сделать выбор одного из нескольких возможных вариантов без использования сложных конструкций, состоящих из нескольких вложенных операторов `if else`. Вот пример выполнения такого выбора:

```
case SomeIntegerVariable of
  101 : DoSomething;
  202 : begin
      DoSomething;
      DoSomethingElse;
    end;
  303 : DoAnotherThing;
  else DoTheDefault;
end;
```

На заметку

В операторе `case` управляющая переменная должна иметь перечислимый тип. В частности, не допускается использование переменных такого неперечислимого типа, как `string`.

А вот как выглядит этот же пример на языке C:

```
switch (SomeIntegerVariable)
{
  case 101: DoSomeThing; break;
  case 202: DoSomething;
            DoSomethingElse; break
  case 303: DoAnotherThing; break;
  default : DoTheDefault;
}
```

Циклы

Цикл представляет собой конструкцию, которая позволяет выполнять некоторые действия неоднократно. Циклы языка Pascal во многом подобны циклам других языков, потому мы не будем останавливаться на деталях их построения и работы. В этом разделе описываются все конструкции циклов, существующие в языке Pascal.

Цикл `for`

Цикл `for` больше всего подходит для организации повторения некоторых действий заранее известное число раз. Например, ниже приведен текст цикла `for` (хотя и не слишком полезного), в котором к значению переменной десять раз прибавляется счетчик цикла (дальше следуют аналоги этого цикла на языках C и Visual Basic).

```
{ Pascal }
var
  I, X: Integer;
begin
  X := 0;
  for I := 1 to 10 do
    inc(X, I);
end.
```

```
/* C */
void main(void) {
  int x, i;
  x = 0;
  for(i=1; i<=10; i++)
    x += i;
}
```

```
' Visual Basic
X = 0
For I = 1 to 10
  X = X + I
Next I
```

Примечание для знатоков Delphi 1: начиная с версии Delphi 2 присвоение значений счетчику цикла в теле этого цикла недопустимо. Это связано с оптимизацией циклов в 32-битовом компиляторе.

Цикл while

Этот тип цикла позволяет повторять выполнение некоторого блока кода до тех пор, пока заданное условие остается истинным. Условие проверяется перед выполнением блока кода, поэтому классическим примером использования данной конструкции является выполнение некоторых действий с файлом (например, чтение его данных), продолжающееся до тех пор, пока не будет достигнут его конец. Ниже приведен пример подобного цикла, в котором при каждом выполнении его тела одна строка файла считывается и выводится на дисплей:

```
Program FileIt;

{$APPTYPE CONSOLE}

var
  f: TextFile;  // Текстовый файл
  s: string;
begin
  AssignFile(f, 'foo.txt');
  Reset(f);
  while not EOF(f) do begin
    readln(f, S);
    writeln(S);
  end;
  CloseFile(f);
end.
```

Цикл while в языке Pascal по своим свойствам не отличается от цикла while в языке C и цикла Do While в языке Visual Basic.

Цикл repeat...until

Этот цикл выполняет ту же задачу, что и цикл while, но несколько иначе. С его помощью некоторый блок кода выполняется до тех пор, пока заданное условие не станет истинным; причем вначале выполняется блок кода, а уже затем проверяется истинность условия. Этот цикл схож с циклом do while в языке C.

Вот пример использования такого цикла — счетчик X будет увеличиваться до тех пор, пока его значение не превысит 100:

```
var
  x: Integer;
begin
  X := 1;
  repeat
    inc(x);
  until x > 100;
end.
```

Процедура Break()

Вызов процедуры Break() внутри цикла while, for или repeat...until организует немедленный выход программы из цикла. Подобный метод полезен в тех случаях, когда во время выполнения цикла возникло некоторое обстоятельство, требующее немедленного завершения его выполнения. Эта процедура аналогична оператору break в языке C и оператору exit в языке Visual Basic. В приведенном ниже примере выполнение цикла прекращается после выполнения пяти итераций:

```
var
  i: Integer;
begin
  for i := 1 to 1000000 do
  begin
    MessageBeep(0);   // Подача звукового сигнала
    if i = 5 then Break;
  end;
end;
```

Процедура Continue()

Вызов процедуры Continue() в некотором цикле приводит к опусканию оставшейся части блока кода цикла и переходу к началу следующей итерации. В частности, в приведенном ниже примере при первой итерации второй оператор вывода строки выполнен не будет:

```
var
  i: Integer;
begin
  for i := 1 to 3 do
  begin
    writeln(i, '. Before continue');
    if i = 1 then Continue;
    writeln(i, '. After continue');
  end;
end;
```

Процедуры и функции

Как программист, вы должны быть хорошо знакомы с основными положениями использования функций и процедур. Процедура представляет собой отдельную часть программы, которая решает некоторую задачу, а затем возвращает управление в точку вызова. Функция работает практически точно так же, за одним исключением — она возвращает некоторое значение, которое может быть использовано в вызвавшем функцию коде.

Если вы знакомы с языком программирования C или C++, то можете рассматривать процедуру языка Pascal как функцию языка C, возвращающую значение void. В этом аспекте функция языка Pascal аналогична функции языка C, возвращающей некоторое конкретное значение.

В листинге 2.1 приведена небольшая программа, в которой используются процедура и функция.

Листинг 2.1. Пример использования функций и процедур

```
Program FuncProc;

{$APPTYPE CONSOLE}

procedure BiggerThanTen(i: Integer);
  { Выводит на экран сообщение, если I больше 10 }
begin
  if I > 10 then
    writeln('Funky.');
end;

function IsPositive(I: Integer): Boolean;
  { Возвращает True, если I больше или равно 0 }
begin
  if I < 0 then
    Result := False
  else
    Result := True;
end;

var
  Num: Integer;
begin
  Num := 23;
  BiggerThanTen(Num);
  if IsPositive(Num) then
    writeln(Num, 'Is positive.')
  else
    writeln(Num, 'Is negative.');
end.
```

На заметку

Локальная переменная `Result` в функции `IsPositive()` имеет специальное назначение. В каждой функции языка Object Pascal существует локальная переменная с этим именем, предназначенная для размещения возвращаемого значения. Обратите внимание на то, что, в отличие от языков C и C++, выполнение функции не прекращается при присвоении значения переменной `Result`.

Кроме того, вернуть значение из функции можно также путем присвоения значения переменной, имеющей то же имя, что и данная функция. Это стандартный синтаксис языка Pascal, сохранившийся от его предыдущих версий. При использовании в теле функции переменной с ее именем не забывайте, что существуют большие отличия в обработке этого имени — все зависит от того, где оно расположено — в левой части оператора присвоения или же в любом другом месте текста функции. Если имя функции указано в левой части оператора присвоения, то предполагается, что назначается возвращаемое функцией значение. Во всех других случаях предполагается, что осуществляется рекурсивный вызов этой функции!

Учтите, что использование переменной `Result` недопустимо при сброшенном флажке опции Extended Syntax, расположенном во вкладке Compiler диалогового окна Project Options, или при указании директивы компилятора {`$X-`}.

Передача параметров

Pascal позволяет передавать параметры в функции и процедуры либо по значению, либо по ссылке. Передаваемый параметр может иметь любой встроенный или пользовательский тип либо являться открытым массивом (они рассматриваются ниже в этой главе). Параметр также может быть константой, если его значение в процедуре или функции не изменяется.

Передача параметров по значению

Этот режим передачи параметров принимается по умолчанию. Если параметр передается по значению, создается локальная копия данной переменной, которая и предоставляется для обработки в процедуру или функцию. Рассмотрим следующий пример:

```
procedure Foo(s: string);
```

При вызове указанной процедуры будет создана копия передаваемой ей в качестве параметра строки `s`, с которой и будет работать процедура `Foo()`. При этом все внесенные в полученную строку изменения никак не отразятся на исходной строке `s`.

Передача параметров по ссылке

Pascal позволяет также передавать параметры в функции или процедуры по ссылке — такие параметры называются параметрами-переменными. Передача параметра по ссылке означает, что функция или процедура сможет изменить полученные значения параметров. Для передачи параметров по ссылке используется ключевое слово `var`, помещаемое в список параметров вызываемой процедуры или функции:

```
procedure ChangeMe(var x: longint);
begin
  x := 2;  { параметр x изменен вызванной процедурой }
end;
```

Вместо создания копии переменной `x`, ключевое слово `var` требует передачи адреса самой переменной `x`, что позволяет процедуре непосредственно изменять ее значение.

Для передачи параметров по ссылке в языке C++ используется оператор `&`.

Параметры-константы

Если нет необходимости изменять передаваемые функции или процедуре данные, можно описать параметр как константу. Ключевое слово `const` не только защищает параметр от изменения, но и позволяет компилятору сгенерировать более оптимальный код передачи строк и записей. Вот пример объявления параметра-константы:

```
procedure Goon(const s: string);
```

Использование открытых массивов

Открытый массив параметров позволяет передавать в функцию или процедуру различное количество параметров. В качестве параметра можно передать либо открытый массив элементов одинакового типа, либо массивы констант различного типа. В приведенном ниже примере объявляется функция, которой в качестве параметра должен передаваться открытый массив целых чисел:

```
function AddEmUp(A: array of Integer): Integer;
```

В открытом массиве можно передавать переменные, константы или выражения из констант. Ниже приведен пример, который демонстрирует вызов функции AddEmUp() с передачей ей нескольких различных элементов.

```
var
  i, Rez: Integer;
const
  j = 23;
begin
  i := 8;
  Rez := AddEmUp([i, 50, j, 89]);
```

Для получения информации о фактически передаваемом массиве параметров в функции или процедуре могут использоваться функции High(), Low() и SizeOf(). Для иллюстрации их использования ниже приведен текст функции AddEmUp(), которая возвращает сумму всех переданных ей элементов массива A.

```
function AddEmUp(A: array of Integer): Integer;
var
  i: Integer;
begin
  Result := 0;
  for i := Low(A) to High(A) do
    inc(Result, A[i]);
end;
```

Object Pascal также поддерживает тип array of const, который позволяет передавать в одном массиве данные различных типов. Синтаксис объявления функций или процедур, использующих такой массив для получения параметров, следующий:

```
procedure WhatHaveIGot(A: array of const);
```

Вызвать объявленную выше функцию можно, например, с помощью такого оператора:

```
WhatHaveIGot(['Tabasco', 90, 5.6, @WhatHaveIGot, 3.14159, True, 's']);
```

При передаче функции или процедуре массива констант все передаваемые параметры компилятор неявно конвертирует в тип TVarRec. Тип данных TVarRec объявлен в модуле System следующим образом:

```
type
PVarRec = ^TVarRec;
  TVarRec = record
    case Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
      vtChar:       (VChar: Char);
      vtExtended:   (VExtended: PExtended);
      vtString:     (VString: PShortString);
      vtPointer:    (VPointer: Pointer);
      vtPChar:      (VPChar: PChar);
      vtObject:     (VObject: TObject);
      vtClass:      (VClass: TClass);
```

```
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
      vtVariant:    (VVariant: PVariant);
      vtInterface:  (VInterface: Pointer);
      vtWideString: (VWideString: Pointer);
      vtInt64:      (VInt64: PInt64);
  end;
```

Поле VType определяет тип содержащихся в данном экземпляре записи TVarRec данных и может принимать одно из приведенных ниже значений.

```
const
  { значения поля TVarRec.VType }
  vtInteger    = 0;
  vtBoolean    = 1;
  vtChar       = 2;
  vtExtended   = 3;
  vtString     = 4;
  vtPointer    = 5;
  vtPChar      = 6;
  vtObject     = 7;
  vtClass      = 8;
  vtWideChar   = 9;
  vtPWideChar  = 10;
  vtAnsiString = 11;
  vtCurrency   = 12;
  vtVariant    = 13;
  vtInterface  = 14;
  vtWideString = 15;
  vtInt64      = 16;
```

Поскольку массив констант способен передавать данные различных типов, это может вызвать определенные затруднения при создании обрабатывающей полученные параметры функции или процедуры. В качестве примера работы с таким массивом рассмотрим реализацию процедуры WhatHaveIGot(), которая просматривает элементы полученного массива параметров и выводит их тип.

```
procedure WhatHaveIGot(A: array of const);
var
  i: Integer;
  TypeStr: string;
begin
  for i := Low(A) to High(A) do
  begin
    case A[i].VType of
      vtInteger    : TypeStr := 'Integer';
      vtBoolean    : TypeStr := 'Boolean';
      vtChar       : TypeStr := 'Char';
      vtExtended   : TypeStr := 'Extended';
      vtString     : TypeStr := 'String';
```

```
      vtPointer    : TypeStr := 'Pointer';
      vtPChar      : TypeStr := 'PChar';
      vtObject     : TypeStr := 'Object';
      vtClass      : TypeStr := 'Class';
      vtWideChar   : TypeStr := 'WideChar';
      vtPWideChar  : TypeStr := 'PWideChar';
      vtAnsiString : TypeStr := 'AnsiString';
      vtCurrency   : TypeStr := 'Currency';
      vtVariant    : TypeStr := 'Variant';
      vtInterface  : TypeStr := 'Interface';
      vtWideString : TypeStr := 'WideString';
      vtInt64      : TypeStr := 'Int64';
    end;
    ShowMessage(Format('Array item %d is a %s', [i, TypeStr]));
  end;
end;
```

Область видимости

Область видимости — это некоторая часть программы, в которой данная функция или переменная известна компилятору. Глобальные константы видимы в любой точке программы, в то время как локальные переменные видны только в процедуре, в которой они объявлены. Рассмотрим листинг 2.2.

Листинг 2.2. Пример, иллюстрирующий понятие области видимости

```
program Foo;

{$APPTYPE CONSOLE}

const
  SomeConstant = 100;

var
  SomeGlobal: Integer;
  R: Real;

procedure SomeProc(var R: Real);
var
  LocalReal: Real;
begin
  LocalReal := 10.0;
  R := R - LocalReal;
end;

begin
  SomeGlobal := SomeConstant;
  R := 4.593;
  SomeProc(R);
end.
```

Здесь переменные SomeConstant, SomeGlobal и R имеют глобальную область видимости, поэтому их значения известны компилятору в любой точке программы. Процедура SomeProc() имеет две собственные локальные переменные: R и LocalReal. Любая попытка обращения к переменной LocalReal вне процедуры SomeProc() вызовет появление сообщения об ошибке. Обращение к переменной R внутри функции SomeProc() приведет к считыванию значения ее локальной переменной, тогда как обращение к переменной R вне этой функции вызовет считывание значения глобальной переменной с этим же именем.

Модули

Модули представляют собой отдельные единицы исходного текста, вся совокупность которых составляет программу на языке Object Pascal. В модуле обычно размещается некоторая группа функций и процедур, которые могут быть вызваны из основной программы. Для того чтобы считаться модулем, файл с исходным текстом должен состоять как минимум из трех частей.

- Описание unit. Каждый модуль должен начинаться со строки, объявляющей, что данный блок текста является модулем, и задающей имя этого модуля. Имя модуля всегда должно соответствовать имени его файла. Например, если файл называется FooBar, его первая строка должна содержать следующий оператор:

  ```
  unit FooBar;
  ```

- Описание интерфейса. После определения модуля следующей функциональной строкой должен быть оператор interface. Все, что находится между этой строкой и оператором implementation данного модуля, доступно извне и может использоваться другими модулями и программами. Именно в этой части описываются типы данных, константы, переменные, процедуры и функции, которые должны быть доступны создаваемой программе или другим модулям. В этой части допустимы только объявления (но не реализация!) функций и процедур. Сам оператор interface занимает отдельную строку и содержит единственное ключевое слов — interface.
- Раздел реализации. Следует за описанием интерфейса и начинается с оператора implementation. Хотя основное содержимое этой части модуля составляют тела описанных ранее процедур и функций, здесь также можно определять типы данных, константы и переменные, однако они будут доступны только в пределах данного модуля. Сам оператор implementation занимает отдельную строку и содержит единственное ключевое слов — implementation.

Кроме того, модуль может иметь еще две необязательные части.

- Раздел инициализации (initialization). Располагается после области реализации и содержит программный текст, необходимый для инициализации работы данного модуля. Этот текст будет выполнен только один раз — перед началом выполнения основной программы.
- Раздел завершения (finalization). Располагается между разделом инициализации и оператором end. модуля. Он содержит программный текст, необходимый для завершения работы модуля. Этот текст будет выполнен только один раз — при завершении работы программы. Раздел завершения впервые появился в Delphi 2.0. В Delphi 1.0 для выполнения завершающих работу модуля действий следовало создать особую процедуру завершения и зарегистрировать ее посредством вызова функции AddExitProc().

На заметку

При наличии разделов initialization/finalization сразу в нескольких модулях выполнение их подпрограмм происходит в том порядке, в котором эти модули открываются компилятором (первый модуль в описании uses основной программы, затем первый модуль в описании uses этого модуля и т.д.). Однако не следует создавать подпрограммы инициализации/завершения модулей, работа которых полагается на некоторый жесткий порядок их выполнения, — это плохой стиль программирования. Малейшее изменение в любом операторе uses может вызвать появление ошибки, найти которую будет чрезвычайно трудно!

Описание uses

В описании uses перечисляются модули, которые будут включены в данную программу (или в модуль). Например, если в программе FooProg используются функции или типы данных из двух модулей — UnitA и UnitB, то описание uses этой программы должно выглядеть следующим образом:

```
Program FooProg;

uses UnitA, UnitB;
```

Модули могут содержать два описания uses: одно — в разделе interface, а другое — в разделе implementation. Вот пример модуля с описаниями uses:

```
Unit FooBar;

interface

uses BarFoo;

  { Общедоступные описания }

implementation

uses BarFly;

  { Закрытые описания }

initialization
  { Инициализация модуля }
finalization
  { Завершение работы модуля }
end.
```

Взаимные ссылки

Иногда может возникнуть ситуация, когда модуль UnitA использует модуль UnitB, а тот в свою очередь — модуль UnitA. Обычно наличие таких взаимных ссылок говорит о просчетах, допущенных на этапе проектирования структуры приложения, — рекомендуется избегать появления в модулях взаимных ссылок. Устранить данную проблему можно путем создания

третьего модуля, в который выносятся необходимые для работы обоих модулей функции и процедуры. Если же по каким-либо соображениям это невозможно, попробуйте в одном модуле разместить ссылку в части `interface`, а в другом — в части `implementation`. Зачастую это решает проблему взаимных ссылок.

Пакеты

Пакеты (packages) Delphi позволяют размещать части некоторого приложения в различных модулях, которые затем могут совместно использоваться несколькими приложениями. Для разработок, выполненных в среде Delphi 1 и 2, преимуществами использования пакетов можно воспользоваться без внесения каких-либо изменений в существующие исходные тексты программ.

Пакеты можно понимать как коллекции модулей, сохраняемых в отдельных файлах-библиотеках (Borland Package Library, BPL), подобных DLL-файлам. Приложение можно связать с пакетированными модулями непосредственно во время выполнения, а не при его трансляции. Естественно, при этом размер выполняемого файла уменьшается, так как часть кода и данных располагается в BPL-файле. Delphi позволяет создавать пакеты четырех типов.

1. Исполнимый пакет. Этот тип пакетов содержит модули, используемые программой во время ее выполнения. Скомпилированное для работы с некоторым пакетом, приложение не будет работать при его отсутствии. Примером пакета такого типа может служить пакет Delphi `VCL50.BPL`.
2. Конструкторский пакет. Пакет этого типа содержит элементы, необходимые для конструирования приложения (например, компоненты, редакторы свойств и компонентов, программы-эксперты). Эти пакеты могут быть включены в библиотеку компонентов Delphi с помощью команды Component⇨Install Package. В качестве примера можно привести пакет Delphi `DCL*.BPL`. Подробно пакеты этого типа описываются в главе 21 второго тома, “Создание пользовательских компонентов в Delphi”.
3. Исполнимые и конструкторские пакеты. Такие пакеты применяются как пакеты одновременно обоих указанных выше типов. Использование подобных пакетов упрощает создание и распространение приложения. Однако это тип пакетов менее эффективен, поскольку помимо необходимой для работы исполняемой части они содержат также конструкторскую поддержку.
4. Пакеты, не относящиеся ни к одному из перечисленных типов. Такие пакеты могут применяться только другими пакетами и не используются непосредственно приложением или средой конструирования.

Использование пакетов Delphi

Создание приложений, работающих с пакетами Delphi, представляет собой несложную задачу. Для этого достаточно установить флажок опции Build with Runtime Packages во вкладке Packages диалогового окно Project Options. Эта опция требует от компилятора создания приложения, динамически скомпонованного с исполнимыми пакетами, вместо статической компоновки всех модулей в общий `EXE`- или `DLL`-файл. В результате размер файла приложения станет намного меньше. Однако не забывайте, что в этом случае при распространении приложение потребуется дополнить необходимыми пакетами.

Синтаксис описания пакетов

Чаще всего пакеты создаются с помощью редактора Package Editor, который вызывается командой File⇨New⇨Package. Этот редактор генерирует исходный файл Delphi Package Source (DPK), который затем компилируется в пакет. Синтаксис, используемый при создании DPK-файла, очень прост и имеет следующий формат:

```
package Имя_пакета

requires Package1, Package2, ...;

contains
  Unit1 in Unit1.pas,
  Unit2 in Unit2.pas,
 ...;

end.
```

Пакеты, перечисленные в предложении `requires`, необходимы для работы текущего пакета. Обычно эти пакеты содержит модули, используемые теми модулями, которые перечислены в предложении `contains` (последние будут скомпилированы в данный пакет). При этом следует помнить, что модули, перечисленные в предложении `contains` этого пакета, не должны упоминаться в предложениях `contains` пакетов, перечисленных в предложении `requires` данного модуля. Заметим также, что в пакет будет неявно включен любой модуль, используемый другим модулем, указанным в предложении `contains` пакета (если только он уже не содержится в одном из пакетов, указанных в предложении `requires`).

Объектно-ориентированное программирование

Объектно-ориентированному программированию (ООП) посвящены многие тома, в дискуссиях о нем сломано немало копий, и уже начинает казаться, что ООП — не методология программирования, а религия. Мы не ортодоксы от ООП и не будем пытаться обратить вас в ту или иную веру. Поэтому мы просто излагаем здесь основные принципы ООП, которые положены в основу языка Object Pascal.

ООП представляет собой парадигму программирования, использующую в качестве строительных элементов программы дискретные объекты, содержащие данные и код. Задача упрощения написания и сопровождения программ не являлась основной целью разработки парадигмы ООП, это, скорее, — побочный эффект применения ООП на практике. Кроме того, хранение данных и кода в одном объекте позволяет минимизировать воздействие одного объекта на другой, а следовательно, и облегчить поиск и исправление ошибок в программе.

Традиционно объектно-ориентированные языки реализуют, по меньшей мере, три основные концепции ООП.

Инкапсуляция — Работа с данными и детали ее реализации скрыты от внешнего пользователя объекта. Достоинства инкапсуляции включают модульность и изоляцию кода разных объектов.

Наследование — Возможность создания новых объектов, которые обладают свойствами и поведением родительских объектов. Такая концепция позволяет создавать иерархии объектов (например, VCL), включающие наборы объектов, порожденных от одного общего предка и обладающих все большей специализацией и функциональностью по сравнению со своими предшественниками. Достоинства наследования состоят, в первую очередь, в разделении общего кода многими объектами. На рис. 2.4 представлен пример наследования: один корневой объект `fruit` (фрукты) служит предком для всех плодов, включая бахчевые, у которых, в свою очередь, объект `melon` служит предком таких объектов, как арбуз (`watermelon`) и зимняя дыня (`honeydew`).

Полиморфизм — Слово *полиморфизм* означает "много форм". В нашем случае под этим подразумевается, что вызов метода переменной объекта будет приводить к вызову кода, соответствующего конкретному экземпляру объекта, хранящемуся в переменной.

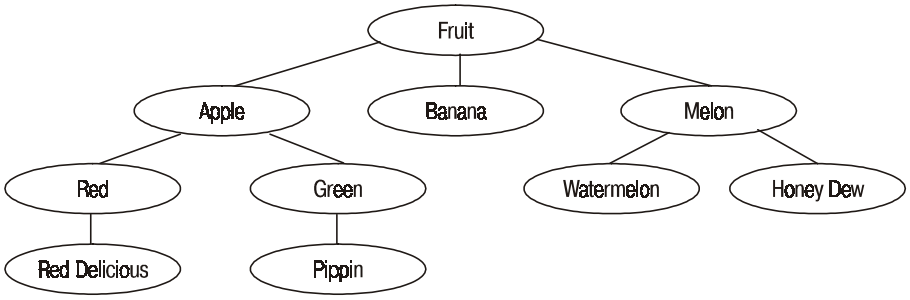


Рис. 2.4. Пример иерархии наследования

Замечания о множественном наследовании

Object Pascal не поддерживает функции множественного наследования, присутствующей в языке C++. *Множественное наследование* означает, что некоторый объект может быть наследником двух и более других различных объектов, т.е. содержать в себе все данные и код объектов-предков.

В рамках приведенной на рис. 2.4 иллюстрации это можно представить как класс "яблочное варенье", предками которого окажутся уже имеющийся класс `apple` (яблоко) и некоторый класс "варенье". Хотя подобная функциональность на первый взгляд кажется полезной, она часто вносит в программы множество проблем и служит причиной снижения их эффективности.

Object Pascal дает два возможных пути решения проблемы множественного наследования. Первый состоит в создании объекта одного класса, который содержит объект другого класса (такое решение использовалось при построении VCL). В рамках яблочно-вареньевой аналогии можно представить себе некий объект "варенье", который содержится в объекте "яблоко". Второе решение заключается в использовании интерфейсов, о чем речь пойдет ниже в этой главе. При использовании интерфейсов появляется возможность создать объект, который будет иметь оба интерфейса — и "яблока", и "варенья".

Для понимания дальнейшего материала нам следует разобраться с тем, что означают термины "поле", "метод" и "свойство".

Поле (Field) — Поля, именуемые также переменными экземпляра, представляют собой переменные с данными, содержащиеся внутри объекта. Поле в объекте сходно с полем в записи Pascal. В языке C++ для них иногда используется термин *члены-данные*

Метод (Method)	Процедуры и функции, принадлежащие объекту. В языке C++ для них используется термин *члены-функции*
Свойство (Property)	Свойства представляют собой сущности, обеспечивающие доступ к данным и коду, содержащемуся в объекте. Тем самым они изолируют конечного пользователя от деталей реализации объекта

На заметку В ООП непосредственный доступ к полям объекта в целом считается плохим стилем программирования. В первую очередь, это связано с тем, что детали реализации объекта могут со временем измениться. Предпочтительнее работать со свойствами, представляющими собой стандартный интерфейс объекта, скрывающий его конкретную реализацию.

Объектно-основанное или объектно-ориентированное программирование

Некоторые языки программирования дают возможность работать с отдельными сущностями, которые можно считать объектами, однако не позволяют создавать собственные объекты. Хорошим примером этого могут служить элементы управления ActiveX (бывшие OCX) в языке Visual Basic. Хотя элементы управления ActiveX можно использовать в приложениях Visual Basic, их нельзя создавать заново или порождать один элемент управления ActiveX из другого. Среда программирования с такими свойствами называется объектно-основанной (object-based).

Delphi представляет собой полностью объектно-ориентированную среду. Это означает, что в Delphi новые объекты можно создавать либо с нуля, либо используя уже имеющиеся. Это относится ко всем объектам Delphi — как визуальным, так и невизуальным (и даже к формам).

Использование объектов Delphi

Как упоминалось ранее, объекты (именуемые также *классами*) представляют собой сущности, которые могут содержать данные и код. Объекты Delphi предоставляют программисту все основные возможности объектно-ориентированного программирования, такие как наследование, инкапсуляция и полиморфизм.

Объявление и создание

Безусловно, перед тем как объект использовать, его следует объявить. В Object Pascal это делается с помощью ключевого слова `class`. Объявления объектов помещаются в раздел объявления типов модуля или программы:

```
type
  TFooObject = class;
```

После объявления типа объекта можно выполнить объявление переменных (называемых также экземплярами) этого типа в разделе `var`:

```
var
  FooObject: TFooObject;
```

В Object Pascal экземпляр объекта создается с помощью вызова одного из *конструкторов* этого объекта. Конструктор отвечает за создание экземпляра объекта, а также за выделение памяти и необходимую инициализацию полей. Он не только создает объект, но и приводит его в состояние, необходимое для его дальнейшего использования. Каждый объект содержит по крайней мере один конструктор `Create()`, который может иметь различное количество параметров разного типа — в зависимости от типа объекта. В этой главе рассматривается только простейший конструктор `Create()` (без параметров).

В отличие от языка C++, конструкторы в Object Pascal не вызываются автоматически. Создание каждого объекта с помощью вызова его конструктора входит в обязанности программиста. Синтаксис вызова конструктора следующий:

```
FooObject := TFooObject.Create;
```

Обратите внимание на уникальную особенность вызова конструктора — он вызывается с помощью ссылки на тип, а не на экземпляр типа (в отличие от других методов, которые вызываются с помощью ссылки на экземпляр). На первый взгляд это может показаться нелепостью, однако в этом есть глубокий смысл — ведь экземпляр объекта `FooObject` в момент вызова конструктора еще не создан. Однако код конструктора класса `TFooObject` статичен и находится в памяти. Он относится к типу, а не его экземпляру, поэтому такой вызов вполне корректен.

Процедуру вызова конструктора для создания экземпляра объекта часто называют *реализацией класса*.

На заметку

При создании экземпляра объекта с помощью конструктора компилятор гарантирует, что все поля экземпляра будут инициализированы. Все числовые поля будут обнулены, указатели примут значение `nil`, а строки будут пусты.

Уничтожение

По окончании использования экземпляра объекта следует освободить выделенную для него память с помощью метода `Free()`. Этот метод сначала проверяет, не равен ли экземпляр объекта значению `Nil`, и затем вызывает *деструктор* объекта — метод `Destroy()`. Понятно, что действие деструктора обратно действию конструктора, т.е. он освобождает всю выделенную память и выполняет другие действия по освобождению захваченных конструктором объекта ресурсов. Синтаксис вызова метода `Free()` прост:

```
FooObject.Free;
```

Обратите внимание, что, в отличие от вызова конструктора, вызов метода `Free()` выполняется с помощью ссылки на экземпляр, а не на тип. Кроме этого, запомните еще один совет — никогда не используйте непосредственный вызов метода `Destroy()`. Более безопасно и корректно вызвать метод `Free()`.

В языке C++ деструктор экземпляра статически созданного объекта вызывается автоматически, когда этот экземпляр покидает область видимости. В случае динамического создания экземпляра (с помощью оператора `new`) вы должны сами уничтожить объект, вызвав оператор `delete`. То же правило действует и в Object Pascal, но с одной поправкой: в нем все экземпляры объекта — динамические, и программист должен самостоятельно их уничтожить. Возьмите себе за правило уничтожать и освобождать все, что

было создано вами в программе. Исключением из этого правила являются объекты, принадлежащие другим объектам (подробнее об этом вы узнаете в главе 20 второго тома, "Ключевые элементы VCL и информация о типах времени выполнения"). Этот тип объектов уничтожается автоматически. Еще одним исключением являются объекты с управляемым временем жизни, имеющие собственный счетчик ссылок (например, производные от классов `TInterfacedObject` или `TComObject`), которые автоматически удаляются после ликвидации последней ссылки на них.

У вас может возникнуть вопрос: "Откуда берутся все эти конструкторы и деструкторы? Неужели необходимо всегда описывать их — даже для самого маленького объекта?" Конечно, нет. Дело в том, что все классы Object Pascal неявно наследуют функциональные возможности базового класса `TObject`, причем, независимо от того, указываете вы это наследование явно или нет. Рассмотрим следующее объявление класса:

```
Type TFoo = Class;
```

Он полностью эквивалентно следующему объявлению этого класса:

```
Type TFoo = Class(TObject);
```

Методы

Методы представляют собой процедуры и функции, принадлежащие заданному объекту. Можно сказать, что методы определяют поведение объекта. Выше были рассмотрены два важнейших метода объектов: конструктор и деструктор. Можно самостоятельно создать произвольное количество любых других методов, необходимых для решения конкретных задач.

Создание метода — процесс двухшаговый. Сначала следует описать метод в объявлении типа, а затем создать текст его реализации. Вот пример описания и определения метода:

```
type
  TBoogieNights = class
    Dance: Boolean;
    procedure DoTheHustle;
  end;

procedure TBoogieNights.DoTheHustle;
begin
  Dance := True;
end;
```

Отметим, что при определении тела метода необходимо использовать его полное имя с указанием объекта. Вторая важная деталь — к любому полю объекта `Dance` его метод может обратиться непосредственно.

Типы методов

Методы объекта могут быть описаны как статические (`static`), виртуальные (`virtual`), динамические (`dinamic`) или как методы-сообщения (`message`). Рассмотрим следующий пример:

```
TFoo = class
  procedure IAmAStatic;
  procedure IAmAVirtual; virtual;
  procedure IAmADynamic; dynamic;
  procedure IAmAMessage(var M: TMessage); message wm_SomeMessage;
end;
```

Статические методы

Статический метод `IAmAStatic` работает подобно обычной процедуре или функции. Этот тип методов принимается по умолчанию. Адрес такого метода известен уже на стадии компиляции, и компилятор в тексте программы оформляет все вызовы данного метода как статические. Такие методы работают быстрее других, однако не могут быть перегружены с целью поддержки полиморфизма объектов.

Виртуальные методы

Метод `IAmAVirtual` объявлен как *виртуальный*. Вызов таких методов, благодаря возможности их перегрузки, немного сложнее, чем вызов статического метода, так как во время компиляции адрес конкретного вызываемого метода не известен. Для решения этой задачи компилятор строит таблицу виртуальных методов (Virtual Method Table, VMT), которая обеспечивает определение адреса метода в процессе выполнения программы. VMT содержит все виртуальные методы предка и виртуальные методы самого объекта, и потому виртуальные методы используют несколько бо́льшую память, чем методы динамические, но при этом их вызов происходит быстрее, чем вызов динамических методов.

Динамические методы

Динамический метод `IAmADynamic` в целом подобен виртуальным методам, но обслуживается другой диспетчерской системой. Каждому динамическому методу компилятор назначает уникальное число и использует его вместе с адресом метода для построения таблицы динамических методов (Dynamic Method Table — DMT). В отличие от VMT, DMT содержит только методы данного объекта, благодаря этому обеспечивается экономия используемой памяти, но одновременно замедляется вызов метода, поскольку для поиска его адреса, скорее всего, будет пересмотрена не одна DMT в иерархии объектов.

Методы-сообщения

Методом обработки сообщений является `IAmAMessage`. Значение после ключевого слова `message` определяет сообщение, в ответ на которое вызывается данный метод. Такие методы создаются для реакции на те или иные сообщения Windows. Они никогда не вызываются непосредственно из программы. Более подробно обработка сообщений обсуждается в главе 5, “Сообщения Windows”.

Переопределение методов

Переопределение (overriding) метода в Object Pascal реализует концепцию полиморфизма ООП. Оно позволяет изменять поведение метода от наследника к наследнику. Переопределение метода возможно только в том случае, если первоначально он был объявлен как `virtual` или

dynamic. Для переопределения метода при его объявлении вместо ключевых слов virtual или dynamic следует указать ключевое слово override. Ниже приведен пример переопределения методов IAmAVirtual и IAmADynamic.

```
TFooChild = class(TFoo)
  procedure IAmAVirtual; override;
  procedure IAmADynamic; override;
  procedure IAmAMessage(var M: TMessage); message wm_SomeMessage;
end;
```

Директива override вызывает замещение строки описания исходного метода в VMT строкой описания нового метода. Если объявить новые функции с описателями virtual или dynamic, а не override, то вместо замещения старых будут созданы новые методы. Если переопределить статический метод, то новый вариант просто полностью заменит статический метод родителя.

Перегрузка метода

Подобно обычным процедурам и функциям, методы могут быть перегружены так, что класс будет содержать несколько методов с одним именем, но с различными списками параметров. Перегруженные методы должны быть объявлены с указанием директивы overload (использовать эту директиву при описании первого перегруженного метода необязательно). Вот пример объявления объекта с перегруженными методами:

```
type
  TSomeClass = class
    procedure AMethod(I: Integer); overload;
    procedure AMethod(S: string); overload;
    procedure AMethod(D: Double); overload;
  end;
```

Дублирование имен методов

Может случиться так, что к одному из классов потребуется добавить метод, замещающий метод с тем же именем, но принадлежащий предку этого класса. В данном случае требуется не переопределить исходный метод, а полностью его заменить. Если просто добавить такой метод в новый класс, компилятор выдаст предупреждение о том, что новый метод скрывает метод базового класса с тем же именем. Для устранения этой ошибки в новом методе укажите директиву reintroduce:

```
type
  TSomeBase = class
    procedure Cooper;
  end;

  TSomeClass = class
    procedure Cooper; reintroduce;
  end;
```

Переменная Self

Во всех методах объекта доступна неявная переменная Self, представляющая собой указатель на тот экземпляр объекта, который был использован при данном вызове этого метода. Переменная Self передается методу компилятором в качестве скрытого параметра.

Свойства

Свойства объекта — это специализированные средства доступа к полям объекта, позволяющие изменять его данные и выполнять его код. По отношению к компонентам свойства являются теми элементами, сведения о которых отображаются в окне Object Inspector. Вот простой пример некоторого объекта, для которого определено свойство:

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
public
  property Value: Integer read SomeValue write SetSomeValue;
end;

procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if SomeValue <> AValue then
    SomeValue := AValue;
end;
```

Класс `TMyObject` представляет собой объект, содержащий одно поле — целое с именем `SomeValue`, один метод — процедуру `SetSomeValue`, а также одно свойство с именем `Value`. Назначение процедуры `SetSomeValue` состоит в присвоении полю `SomeValue` некоторого значения. Свойство `Value` не содержит данных — оно используется в качестве средства доступа к полю `SomeValue`. Когда вы запрашиваете значение свойства `Value`, оно считывает его из поля `SomeValue` и возвращает вам. При попытке присвоить свойству `Value` некоторое значение, вызывается процедура `SetSomeValue`, предназначенная для изменения значения поля `SomeValue`. Вся эта технология имеет два основных преимущества. Во-первых, она позволяет представить конечному пользователю некий интерфейс, полностью скрывающий реализацию объекта и обеспечивающий контроль за доступом к объекту. Во-вторых, она дает программисту возможность замещать методы в классах-потомках с целью обеспечения полиморфного поведения объектов.

Определение области видимости

Object Pascal предоставляет дополнительный контроль над доступностью членов классов (полей и методов) с помощью директив `protected`, `private`, `public`, `published` и `automated`. Синтаксис использования этих директив следующий:

```
TSomeObject = class
private
  APrivateVariable: Integer;
  AnotherPrivateVariable: Boolean;
protected
  procedure AProtectedProcedure;
  function ProtectMe: Byte;
public
  constructor APublicContructor;
  destructor APublicKiller;
published
  property AProperty read APrivateVariable write APrivateVariable;
end;
```

За каждой из директив может следовать любое необходимое количество объявлений полей или методов. Требования хорошего стиля предполагают наличие отступа — аналогично тому, как это делается для имен классов. Что же означают эти директивы?

`private` Эта часть объекта доступна только для кода, находящегося в одном модуле с другим кодом данного объекта. Директива `private` скрывает особенности реализации объекта от пользователей и защищает члены этого объекта от непосредственного доступа и изменения извне.

`protected` Члены объекта, описанные как `protected`, доступны для производных объектов. Это позволяет скрыть внутреннее устройство объекта от пользователя, и в то же время обеспечить необходимую гибкость и эффективность доступа к полям и методам объекта для его потомков.

`public` Описанные подобным образом члены объекта доступны в любом месте данной программы. Конструкторы и деструкторы всегда должны быть описаны как `public`.

`published` Для этой части объекта при компиляции будет сгенерирована информация о типе времени исполнения (Runtime Type Information — RTTI). Это даст возможность другим частям приложения получать информацию о части объекта, описанной как `published`. В частности, подобная информация используется утилитой Object Inspector при построении списков свойств объектов.

`automated` Этот описатель сохранен только для обеспечения обратной совместимости с Delphi 2. Более полную информацию по этому вопросу можно найти в главе 23 второго тома, "COM-ориентированные технологии".

Ниже приведено объявление класса `TMyObject` (использовавшееся в разделе, посвященном свойствам объекта) с добавлениями, повышающими его целостность и логичность.

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
published
  property Value: Integer read SomeValue write SetSomeValue;
end;

procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if SomeValue <> AValue then
    SomeValue := AValue;
end;
```

Теперь ни один из пользователей этого класса не сможет изменить значение `SomeValue` непосредственно и вынужден будет использовать только интерфейс, предоставляемый свойством `Value`.

Дружественные классы

В языке C++ реализовано концепция *дружественных классов*, т.е. классов, которым разрешен доступ к данным и функциям другого класса, защищенным описателем `private`. В языке C++ этот механизм реализуется с помощью ключевого слова `friend`. Хотя, строго говоря, в Object Pascal нет

ни такого ключевого слова, ни такого понятия, тем не менее, фактически, того же эффекта можно добиться, просто описав объекты в одном модуле. Описанные в одном модуле объекты имеют право доступа к закрытым данным и функциям друг друга, благодаря чему и обеспечивается "дружественность" в пределах модуля.

Внутреннее представление объектов

Все экземпляры классов Object Pascal на самом деле представляют собой 32-битовые указатели на данные этого экземпляра объекта, расположенные в динамической памяти. При доступе к полям, методам или свойствам объекта компилятор автоматически выполняет некие скрытые действия, приводящие к разрешению этих ссылок. В результате для постороннего взгляда объект всегда выглядит как статическая переменная. Однако использование подобного механизма означает, что, в отличие от языка C++, язык Object Pascal предоставляет реальные методы для размещения данных класса не в динамической памяти, а в сегменте данных приложения.

Класс `TObject`: родительский объект всех объектов

Все объекты в языке Object Pascal являются наследниками базового объекта `TObject`, а следовательно, автоматически наследуют все его методы. В результате любой объект способен, например, сообщить вам свое имя, тип и даже указать, является он наследником некоторого класса или нет. Самым замечательным во всем этом механизме является тот факт, что прикладной программист избавлен от необходимости беспокоиться о реализации стандартных функций — он может просто использовать их по своему усмотрению.

`TObject` является особым объектом, объявление которого расположено в модуле `System` и всегда доступно компилятору:

```
type
  TObject = class
    constructor Create;
    procedure Free;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer;
    class function InstanceSize: Longint;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: ShortString): Pointer;
    class function MethodName(Address: Pointer): ShortString;
    function FieldAddress(const Name: ShortString): Pointer;
    function GetInterface(const IID: TGUID; out Obj): Boolean;
    class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    function SafeCallException(ExceptObject: TObject;
      ExceptAddr: Pointer): HResult; virtual;
    procedure AfterConstruction; virtual;
```

```
    procedure BeforeDestruction; virtual;
    procedure Dispatch(var Message); virtual;
    procedure DefaultHandler(var Message); virtual;
    class function NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    destructor Destroy; virtual;
  end;
```

Описание каждого из перечисленных методов можно найти в интерактивной справочной системе Delphi.

Обратите внимание на методы, объявление которых начинается со слова `class`. Это означает, что метод может быть вызван как обычная процедура или функция без создания экземпляра класса, членом которого является данный метод. (Аналогами таких методов в языке C++ являются методы, описанные как `static`.) Будьте осторожны при создании подобных методов — они не должны использовать никакой информации экземпляра класса.

Интерфейсы

Возможно, наиболее важным дополнением к языку Object Pascal стала поддержка *интерфейсов*, введенная в Delphi 3. Интерфейс определяет набор функций и процедур, которые могут быть использованы для взаимодействия программы с объектом. Определение конкретного интерфейса известно как разработчику, так и пользователю интерфейса, и воспринимается как соглашение о правилах объявления и использования этого интерфейса. В классе может быть реализовано несколько интерфейсов. В результате объект становится "многоликим", являя клиенту каждого интерфейса свое особое лицо.

Интерфейсы представляют собой только определения того, каким образом могут сообщаться между собой объект и его клиент. Эта концепция схожа с концепцией чисто виртуальных классов языка C++. Класс, включающий поддержку некоторого интерфейса, должен обеспечить реализацию всех его функций и процедур.

В этой главе речь пойдет лишь о языковых элементах поддержки интерфейсов. Более детально тема интерфейсов обсуждается в главе 23 второго тома, "COM-ориентированные технологии".

Определение интерфейса

Так же как все классы языка Object Pascal являются наследниками класса `TObject`, все интерфейсы явно или неявно являются производными от интерфейса `IUnknown`, объявление которого в модуле `System` выглядит следующим образом:

```
type
  IUnknown = interface
    ['{ 00000000-0000-0000-C000-000000000046} ']
    function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

Как видите, синтаксис определения интерфейса очень похож на синтаксис определения класса. Основное отличие между этими двумя определениями заключается в том, что интерфейс может быть связан с глобальным уникальным идентификатором (Globally Unique Identifier — GUID). Определение интерфейса `IUnknown` "происходит" из модели многокомпонентных объектов Microsoft (Component Object Model — COM), более детально описанной в главе 23 второго тома, "COM-ориентированные технологии".

Определение пользовательского интерфейса не представляет особых сложностей, если вы знаете, как определяются классы Delphi. Ниже приведен пример определения нового интерфейса IFoo, который реализует единственный метод F1().

```
type
  IFoo = interface
    ['{ 2137BF60-AA33-11D0-A9BF-9A4537A42701} ']
    function F1: Integer;
  end;
```

Получение нового GUID в интегрированной среде разработчика Delphi обеспечивается комбинацией клавиш <Ctrl+Shift+G>.

Вот определение нового интерфейса IBar, являющегося наследником только что определенного IFoo:

```
type
  IBar = interface(IFoo)
    ['{ 2137BF61-AA33-11D0-A9BF-9A4537A42701} ']
    function F2: Integer;
  end;
```

Реализация интерфейсов

Приведенный ниже фрагмент текста демонстрирует реализацию интерфейсов IFoo и IBar в классе TFooBar.

```
type
  TFooBar = class(TInterfacedObject, IFoo, IBar)
    function F1: Integer;
    function F2: Integer;
  end;

function TFooBar.F1: Integer;
begin
  Result := 0;
end;

function TFooBar.F2: Integer;
begin
  Result := 0;
end;
```

Обратите внимание на то, что после указания класса-предка в первой строке объявления класса может быть перечислено несколько интерфейсов. Связывание функции интерфейса с определенной функцией класса осуществляется компилятором тогда, когда он обнаруживает метод класса, сигнатура которого совпадает с сигнатурой метода интерфейса. Класс должен реализовать все методы интерфейса путем объявления и реализации собственных методов. Если в классе не реализуется хотя бы один из методов интерфейса, компилятор выдаст соответствующее сообщение об ошибке и программа откомпилирована не будет.

Если класс реализует несколько интерфейсов, которые имеют одинаковые методы, избежать неоднозначности позволяет назначение методам имен-псевдонимов, как показано в следующем примере:

```
type
  IFoo = interface
    ['{ 2137BF60-AA33-11D0-A9BF-9A4537A42701} ']
    function F1: Integer;
  end;

  IBar = interface
    ['{ 2137BF61-AA33-11D0-A9BF-9A4537A42701} ']
    function F1: Integer;
  end;

  TFooBar = class(TInterfacedObject, IFoo, IBar)
    // Определение псевдонимов
    function IFoo.F1 = FooF1;
    function IBar.F1 = BarF1;
    // Методы класса с именами псевдонимов
    function FooF1: Integer;
    function BarF1: Integer;
  end;

function TFooBar.FooF1: Integer;
begin
  Result := 0;
end;

function TFooBar.BarF1: Integer;
begin
  Result := 0;
end;
```

Директива implements

В Delphi 4 введена директива implements, которая позволяет делегировать реализацию методов интерфейса другим классам или интерфейсам. Она всегда указывается как последняя директива в объявлении свойства класса или интерфейса:

```
type
  TSomeClass = class(TInterfacedObject, IFoo)
    // Описания
    function GetFoo: TFoo;
    property Foo: TFoo read GetFoo imlements IFoo;
    // Описания
  end;
```

В приведенном примере использование директивы implements говорит о том, что методы, реализующие интерфейс IFoo следует искать в свойстве Foo. Тип свойства должен быть классом, содержащим методы IFoo, интерфейс IFoo или его потомков. После директивы

implements вы можете использовать не один, а целый список интерфейсов (элементы списка должны отделяться запятыми). В этом случае класс, используемый для представления свойства, должен содержать методы, реализующие все приведенные в списке интерфейсы.

Директива implements позволяет выполнять бесконфликтное объединение. Объединение представляет концепцию COM, относящуюся к комбинации нескольких классов для выполнения одной задачи (более подробно об этом рассказывается в главе 23 второго тома, "COM-ориентированные технологии"). Другое немаловажное достоинство применения директивы implements заключается в том, что она позволяет отложить использование необходимых для реализации интерфейса ресурсов до того момента, когда они потребуются реально. Предположим, что для реализации некоторого интерфейса необходим мегабайт памяти для хранения растрового изображения, но этот интерфейс требуется пользователю крайне редко. Вряд ли можно считать эффективным решение постоянно выделять мегабайт памяти, который будет использоваться лишь время от времени. Директива implements позволяет реализовать интерфейс в отдельном классе, экземпляр которого будет создаваться только по прямому запросу пользователя.

Использование интерфейсов

При использовании переменных интерфейсного типа в приложениях необходимо помнить о нескольких важных правилах. Во-первых, не забывайте, что интерфейс — это тип данных с управляемым временем жизни. А это означает, что он всегда инициализируется значением nil, обладает счетчиком ссылок и автоматически уничтожается при выходе за пределы области видимости. Вот небольшой пример, иллюстрирующий управление временем жизни интерфейсной переменной:

```
var
  I: ISomeInterface;
begin
  // I инициализируется значением nil
  I := Функция_возвращающая_интерфейс;  // Счетчик ссылок увеличивается на 1
  I.SomeFunc; // Счетчик ссылок уменьшается на 1
  // При обнулении счетчика объект уничтожается
end;
```

Во-вторых, не забывайте, что интерфейсные переменные совместимы по присвоению с классами, реализующими интерфейсы. В приведенном ниже примере корректно используется объявленный ранее класс TFooBar.

```
procedure Test(FB: TFooBar)
var

  F: IFoo;
begin

  F := FB;  // Корректно, так как FB поддерживает IFoo
  .
  .
  .
```

Последнее правило гласит, что оператор преобразования типов as может быть использован для вызова метода QueryInterface данной интерфейсной переменной с целью получения адреса другой интерфейсной переменной (не отчаивайтесь, если это правило вам пока неяс-

но, — детально эта тема будет рассмотрена в главе 23 второго тома, "COM-ориентированные технологии"). Данное правило можно проиллюстрировать следующим примером:

```
var
  FB: TFooBar;
  F: IFoo;
  B: IBar;
begin
  FB := TFooBar.Create
  F := FB;   // Допустимо, так как FB поддерживает IFoo
  B := F as IBar;  // Вызов функции QueryInterface F для IBar
  .
  .
  .
```

Если требуемый интерфейс не поддерживается, будет сгенерирована исключительная ситуация.

Структурированная обработка исключений

Структурированная обработка исключений (Structured exception handling — SEH) представляет собой метод обработки ошибок, благодаря которому можно восстановить нормальную работу приложения после сбоя в работе программы, который в противном случае был бы фатальным. Исключения были введены в язык Object Pascal в Delphi 1.0, но только начиная с Delphi 2.0 они стали частью Win32 API. Представление исключительных ситуаций с помощью классов, содержащих информацию о том, где и какая ошибка произошла, делает использование системы обработки ошибок в Object Pascal простым и понятным.

Delphi имеет предопределенные классы исключительных ситуаций для обработки стандартных ошибок, таких как нехватка памяти, деление на нуль, числовое переполнение, исчезновение значности или ошибки ввода-вывода. Delphi также позволяет создавать собственные классы исключительных ситуаций, предназначенные для обработки ошибок в приложениях.

В листинге 2.3 приведен пример обработки исключительной ситуации при файловых операциях.

Листинг 2.3. Обработка ошибок файлового ввода-вывода

```
Program FileIO;

uses Classes, Dialogs;
{$APPTYPE CONSOLE}

var
  F: TextFile;
  S: string;
begin
  AssignFile(F, 'FOO.TXT');
  try
    Reset(F);
    try
```

```
      ReadLn(F, S);
    finally
      CloseFile(F);
    end;
  except
    on EInOutError do
      ShowMessage('Error Accessing File!');
  end;
end.
```

В данном листинге внутренний блок `try finally` используется для того, чтобы файл был закрыт в любом случае, т.е. независимо от того, была ошибка или нет. На "человеческом" языке это звучит так: "Дружище, выполни код между операторами `try` и `finally`. Возникла при этом некоторая ошибка или нет, не имеет значения — в любом случае выполни операторы между `finally` и `end`. А теперь, если ошибка была, переходи к следующему блоку обработки ошибок". Это означает, что в нашем случае файл будет закрыт независимо от того, произошла ошибка (и какая именно) или нет.

На заметку

Операторы после `finally` в блоке `try finally` выполняются независимо от того, возникла ли ошибка в процессе выполнения этого блока или нет. При написании этого кода убедитесь, что он не зависит от того, имела ли место некая ошибка или нет. Кроме того, так как оператор `finally` не прекращает перемещения сообщения об ошибке, выполнение программы продолжится переходом в следующий блок обработки ошибок.

Внешний блок `try except` используется для обработки исключительной ситуации, которая может произойти в программе. После закрытия файла во внутреннем блоке `finally` блок `except` выводит сообщение об ошибке для пользователя.

Одно из главных преимуществ системы обработки исключительных ситуаций по сравнению с традиционными методами обработки ошибок состоит в отделении процедуры обнаружения ошибки от процедуры ее обработки. Это позволяет сделать код более удобным для чтения и понимания и в каждом случае сосредоточиться на отдельном аспекте работы программы.

Весьма примечателен тот факт, что блок `try finally` не позволяет перехватывать некоторую конкретную исключительную ситуацию. Помещая в программу блок `try finally`, программист заботится не об обнаружении некоторой определенной ошибки. Цель состоит в том, чтобы выполнить все необходимые действия для корректного выхода из *любой* ошибочной ситуации, которая может возникнуть при выполнении этого фрагмента программы. Блок `finally` является идеальным местом для выполнения действий по освобождению любых распределенных ресурсов, поскольку он выполняется всегда, включая и случай возникновения ошибки. Тем не менее во многих ситуациях обработка ошибок может быть связана с выполнением определенных действий, зависящих от типа возникшей ошибки. Для этой цели предназначен блок `try except`, пример использования которого показан в листинге 2.4.

Листинг 2.4. Блок обработки ошибок `try except`

```
Program HandleIt;

{$APPTYPE CONSOLE}

var
  R1, R2: Double;
```

```
begin
  while True do begin
  try
    Write('Enter a real number: ');
    ReadLn(R1);
    Write('Enter another real number: ');
    ReadLn(R2);
    Writeln('I will now divide the first number by the second...');
    Writeln('The answer is: ', (R1 / R2):5:2);
  except
    On EZeroDivide do
      Writeln('You cannot divide by zero!');
    On EInOutError do
      Writeln('That is not a valid number!');
  end;
  end;
end.
```

В блоке `try except`, помимо перехвата конкретных исключительных ситуаций, с помощью конструкции `else` можно организовать перехват всех остальных типов ошибок, как показано в следующем примере:

```
try
  Операторы
except
  On ESomeException do Something;
else
  { Некоторая стандартная обработка всех типов ошибок }
end;
```

При использовании конструкции `try except else` следует учитывать, что в блоке `else` будут перехватываться *все* типы ошибок, включая и те, которых вы не ожидаете, например ошибки выделения памяти или некоторые типы ошибок времени выполнения программы. Поэтому при использовании фразы `else` проявляйте определенную осторожность. Если вы не в состоянии обработать некоторую ошибку, передайте ее дальше. Подробнее это тема будет рассмотрена ниже в этой главе.

Того же самого эффекта (перехват всех типов исключительных ситуаций) можно достичь и с помощью конструкции `try except` без указания типа обрабатываемой ошибки:

```
try
  Операторы
except
  Обработчик  // Перехват всех исключительных ситуаций
end;
```

Классы исключительных ситуаций

Исключения (exceptions) представляют собой специальные экземпляры объектов. Они создаются при возникновении исключительных ситуаций и уничтожаются, когда ситуация обработана. Базовым для всех классов исключений является класс `Exception`, объявляемый следующим образом:

```
type
  Exception = class(TObject)
  private
    FMessage: string;
    FHelpContext: Integer;
  public
    constructor Create(const Msg: string);
    constructor CreateFmt(const Msg: string; const Args: array of const);
    constructor CreateRes(Ident: Integer); overload;
    constructor CreateRes(ResStringRec: PResStringRec); overload
    constructor CreateResFmt(Ident: Integer; const Args: array of const);
overload;
    constructor CreateResFmt(ResStringRec: PResStringRec;
      const Args: array of const); overload;
    constructor CreateHelp(const Msg: string; AHelpContext: Integer);
    constructor CreateFmtHelp(const Msg: string; const Args: array of const;
      AHelpContext: Integer);
    constructor CreateResHelp(Ident: Integer; AHelpContext: Integer); overload;
    constructor CreateResHelp(ResStringRec: PResStringRec;
      AHelpContext: Integer); overload;
    constructor CreateResFmtHelp(Ident: Integer; const Args: array of const;
      AHelpContext: Integer); overload;
    constructor CreateResFmtHelp(ResStringRec: PResStringRec;
      const Args: array of const; AHelpContext: Integer); overload;
    property HelpContext: Integer read FHelpContext write FHelpContext;
    property Message: string read FMessage write FMessage;
  end;
```

Важным элементом объекта `Exception` является свойство `Message`, имеющее строковый тип. Оно содержит дополнительную информацию или разъяснение сути происшедшей ошибки. Вид информации в свойстве `Message` зависит от типа возникшей исключительной ситуации.

При определении собственного объекта исключения обязательно объявляйте его как производный от уже существующего объекта исключения. Это может быть класс `Exception` или любой его потомок. Суть подобного требования заключается в том, что в этом случае новый объект исключения гарантированно будет перехватываться стандартными обработчиками ошибок.

При обработке конкретной исключительной ситуаций в блоке `except` обработчик будет перехватывать также все исключительные ситуации, порожденные данной. Например, исключение `EMathError` является родительским объектом для многих других типов исключений — в частности, для исключительных ситуаций `EZeroDivide` и `EOverflow`. Все они будут перехватываться обработчиком для исключительной ситуации `EMathError`:

```
try
  Операторы
except
  on EMathError do  // Перехват исключения EMathError и всех его потомков
    Обработка
end;
```

Любое исключение, которое не будет перехвачено и обработано самой программой, будет перехвачено и обработано стандартным обработчиком исключений, входящим в состав библиотеки времени выполнения Delphi. Этот обработчик выводит диалоговое окно с сообщением, извещающим пользователя о возникшей исключительной ситуации. В главе 4, "Строение приложения и концепции конструирования", мы обсудим, каким образом можно переопределить стандартный обработчик ошибок Delphi.

При обработке исключительной ситуации может потребоваться доступ к самому экземпляру объекта исключения — например, для получения дополнительной информации о случившемся, помещаемой в свойство `Message` этого объекта. Для этого есть два пути: использование необязательного идентификатора в конструкции `ESomeException` или обращение к функции `ExceptObject()`.

Посредством добавления дополнительного идентификатора к фразе `ESomeException` блока `except` можно получить идентификатор, обозначающий тот объект исключения, который был создан при возникновении текущей исключительной ситуации. Синтаксис задания дополнительного идентификатора состоит в указании этого идентификатора и двоеточия непосредственно *перед* типом исключительной ситуации:

```
try
  Операторы
except
  on E:ESomeException do
    ShowMessage(E.Message);
end;
```

В этом случае идентификатор (в нашем примере это `E`) обозначает экземпляр объекта исключения, сгенерированного при возникновении текущей исключительной ситуации. Данный идентификатор всегда имеет тот же тип, что и объект исключения, который он обозначает.

Кроме того, можно воспользоваться функцией `ExceptObject()`, которая возвращает экземпляр объекта исключения, созданного для текущей исключительной ситуации. Однако следует заметить, что тип возвращаемого функцией значения всегда равен `TObject`, и для работы с ним потребуется выполнить преобразование типов. Вот пример использования описываемой функции:

```
try
  Операторы
except
  on ESomeException do
    ShowMessage(ESomeException(ExceptObject).Message);
end;
```

При вызове функции `ExceptObject()` в случае отсутствия активного исключения возвращается значение `Nil`.

Синтаксис генерации исключительной ситуации подобен синтаксису создания экземпляра объекта. Так, для генерации определенной пользователем исключительной ситуации под названием `EBadStuff` следует использовать следующий синтаксис:

```
Raise EBadStuff.Create('Описание возникшей ошибки');
```

Обработка исключительных ситуаций

После генерации исключительной ситуации и создания объекта исключения, нормальный ход выполнения программы прерывается и управление передается от одного обработчика ошибок к другому до тех пор, пока исключительная ситуация не будет обработана, а экземп-

ляр объекта исключения — уничтожен. Этот процесс построен на обработке стека вызовов и, следовательно, носит глобальный характер в пределах всей программы, а не только в рамках текущей процедуры или модуля. В листинге 2.5 приведен пример, иллюстрирующий указанный принцип обработки исключений. Здесь представлен основной модуль некоторого приложения Delphi, содержащего единственную форму, в которую помещена одна кнопка. По щелчку на кнопке метод `Button1Click()` вызывает процедуру `Proc1()`, которая, в свою очередь, вызывает процедуру `Proc2()`, вызывающую процедуру `Proc3()`. Исключение генерируется именно в этой, последней, наиболее глубоко вложенной процедуре `Proc3()`, что позволяет проследить прохождение процесса обработки исключительной ситуации через каждый из блоков `try...finally` до тех пор, пока данная исключительная ситуация не будет обработана в подпрограмме метода `Button1Click()`.

Если вы запускаете эту программу из интегрированной среды разработки Delphi, то проследить процесс обработки исключительной ситуации будет проще, если предварительно отключить встроенный перехватчик исключительных ситуаций отладчика. Для этого следует сбросить флажок опции Stop on Delphi Exceptions во вкладке Language Exceptions диалогового окна Debugger Options. Доступ к этому диалоговому окну осуществляется по команде Tools⇨Debugger Options.

Листинг 2.5. Приложение, демонстрирующее процедуру обработки исключительных ситуаций

```
unit Main;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
implementation

{$R *.DFM}

type
  EBadStuff = class(Exception);

procedure Proc3;
```

```
begin
  try
    raise EBadStuff.Create('Up the stack we go!');
  finally
    ShowMessage('Exception raised. Proc3 sees the exception');
  end;
end;

procedure Proc2;
begin
  try
    Proc3;
  finally
    ShowMessage('Proc2 sees the exception');
  end;
end;

procedure Proc1;
begin
  try
    Proc2;
  finally
    ShowMessage('Proc1 sees the exception');
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
const
  ExceptMsg = 'Exception handled in calling procedure. The message is "%s"';
begin
  ShowMessage('This method calls Proc1 which calls Proc2 which calls Proc3');
  try
    Proc1;
  except
    on E:EBadStuff do
     ShowMessage(Format(ExceptMsg, [E.Message]));
  end;
end;

end.
```

Регенерация исключений

Если вам необходимо выполнить некоторые действия при возникновении исключения во внутреннем блоке `try except`, но не прекратить на этом обработку, а передать исключение дальше, внешнему обработчику, вы можете воспользоваться технологией *регенерации* исключений. В листинге 2.6 приведен пример этой технологии.

Листинг 2.6. Регенерация исключений

```
try              // Внешний блок
  { операторы }
  { операторы }
  { операторы }
  try            // Внутренний блок
    { Некоторые действия, требующие специальной обработки исключения }
  except
    on ESomeException do
    begin
      { Специальная обработка исключения }
      raise;     // Регенерация исключения
    end;
  end;
except
  // Внешний блок со стандартной обработкой
  on ESomeException do Something;
end;
```

Информация о типе времени выполнения

Информация о типе времени выполнения (Runtime Type Information — RTTI) представляет собой функцию языка, обеспечивающую предоставление приложениям Delphi информации об объектах непосредственно во время выполнения программы. Эта же функция используется для обмена информацией между компонентами Delphi и графической средой разработки.

Будучи потомками `TObject`, все объекты Object Pascal содержат указатель на информацию о типе и некоторые встроенные методы для работы с ней, обеспечивающие функциональные возможности RTTI. Ниже приведены некоторые методы класса `TObject`, предназначенные для получения информации о конкретном экземпляре объекта.

Функция	Возвращаемый тип	Информация
`ClassName()`	`string`	Имя класса объекта
`ClassType()`	`TClass`	Тип объекта
`InheritsFrom()`	`Boolean`	Сведения (да/нет) о том, является ли данный класс потомком указанного класса
`ClassParent()`	`TClass`	Тип предка объекта
`InstanceSize()`	`word`	Размер экземпляра объекта в байтах
`ClassInfo()`	`Pointer`	Указатель на RTTI объекта

Object Pascal предоставляет программисту два оператора — `is` и `as`, — которые позволяют проводить сравнение и преобразование типов с помощью средств RTTI.

Оператор as обеспечивает новую форму преобразования типов, которая позволяет привести низкоуровневый родительский объект к типу-потомку (при невозможности такого преобразования будет сгенерирована исключительная ситуация). Предположим, что существует процедура, которой может быть передан в качестве параметра объект любого типа. Ниже приведен возможный вариант объявления такой процедуры:

```
procedure Foo(AnObject: TObject);
```

Если требуется не просто получить объект в качестве параметра, а сделать с его помощью что-либо полезное, необходимо суметь привести его к типу-потомку, обладающему необходимой функциональностью. Предположим, что полученный объект AnObject опознан как потомок класса TEdit и необходимо изменить текст, содержащийся в этом объекте. (В Delphi класс TEdit является элементом управления VCL, представляющим собой окно текстового редактора.). Для этого можно воспользоваться таким кодом:

```
(Foo as TEdit).Text := 'Hello World!';
```

Кроме того, для непосредственного сравнения типов объектов можно использовать оператор is. Этот оператор позволяет сравнить характеристики неизвестного объекта с характеристиками известного типа данных или экземпляра объекта. На основе этого сравнения можно будет сделать выводы о возможных свойствах и поведении неизвестного объекта. Так, до выполнения приведения типа (см. предыдущий пример) можно проверить, является ли полученный объект AnObject совместимым с классом TEdit:

```
If (Foo is TEdit) then
  TEdit(Foo).Text := 'Hello World!';
```

Обратите внимание, что в этом примере во второй строке вместо оператора as использовано стандартное приведение типа. Это сделано для повышения эффективности работы программы, так как в первой строке кода мы уже установили, что тип объекта Foo — TEdit, и дополнительная проверка допустимости преобразования, выполняемая при использовании оператора as, уже не нужна.

Резюме

В этой главе вы ознакомились с основами синтаксиса и семантики языка Object Pascal, в том числе с его переменными, операторами, функциями, процедурами, типами данных, основными программными конструкциями и другими компонентами. Кроме того, теперь вам известны базовые концепции ООП, включая понятия объектов, полей, свойств, методов и многое другое.

После того как читатель получил представление о языке в целом, мы перейдем к более конкретным темам, касающимся Win32 API и библиотеки Visual Component Library.

Глава

3

Win32 API

Данная глава служит введением в Win32 API и в целом в систему Win32. Здесь обсуждаются возможности этой системы и ее ключевые отличия от 16-битовой версии. Материал этой главы нельзя считать исчерпывающим и детальным описанием системы Win32, его назначение скромнее — предоставить читателю сведения только об основных принципах функционирования системы. Четкое понимание этих основных принципов позволит вам воспользоваться любыми возможностями системы Win32, когда в этом возникнет реальная необходимость.

Объекты и еще раз объекты

Понятие *объект* весьма многогранно и зависит от контекста. Обсуждая архитектуру системы Win32, мы вновь сталкиваемся с ним, однако здесь объект — совсем не то, что подразумевается под этим термином в теории объектно-ориентированного программирования или в технологии COM. Здесь объекты — нечто совсем другое; чтобы окончательно вас запутать, добавим, что объекты в среде Win32 отличаются от объектов в 16-битовых Windows. Попробуем разобраться, что же такое объекты в среде Win32.

В среде Win32 существует два основных типа объектов — объекты ядра (kernel) и объекты GDI/USER (графического интерфейса и интерфейса пользователя).

Объекты ядра

Объекты ядра представляют собой базовые объекты Win32 и включают события, отображения файлов, файлы, почтовые гнезда, блоки совместного использования, каналы, семафоры, процессы и потоки. Для каждого типа объекта ядра в Win32 API имеются свои специфические функции. Прежде чем перейти к непосредственному рассмотрению этих объектов, целесообразно обсудить понятие процесса, что совершенно необходимо для понимания принципов работы с объектами в среде Win32.

Процессы и потоки

Процесс можно представить как выполняющееся приложение или отдельный экземпляр приложения. В среде Win32 одновременно могут быть активными сразу несколько процессов. Каждый процесс получает собственное 4-гигабайтовое адресное пространство для кода и данных. В этих 4 гигабайтах памяти размещается вся распределяемая приложением память, все его потоки, карты файлов и многое другое. Кроме того, в это же адресное пространство процесса загружаются и все динамически подключаемые им библиотеки (DLL). Более детально работа с памятью в среде Win32 описана в разделе “Управление памятью в Win32”, ниже в этой главе.

Сами по себе процессы инертны, т.е. они ничего не выполняют. Однако каждый процесс имеет *первичный поток* (primary thread), в рамках которого и выполняется программный код, присутствующий в контексте данного процесса. Процесс может иметь несколько потоков, однако только один из них является первичным, или главным.

На заметку

Поток (thread) — это объект операционной системы, который представляет независимую последовательность выполнения программного кода, имеющую место в рамках некоторого процесса. Каждое приложение Win32 имеет минимум один поток, часто именуемый первичным, или главным. Однако при необходимости приложение может создавать и другие потоки, предназначенные для решения отдельных задач. Детально потоки описываются в главе 11, “Создание многопоточных приложений”.

При создании процесса операционная система автоматически создает его главный поток. В свою очередь, этот поток при необходимости может создавать дополнительные потоки. Система Win32 распределяет время центрального процессора между всеми потоками, выделяя каждому из них для выполнения отдельные кванты времени (time slices).

В табл. 3.1 описаны основные функции Win32 API для работы с процессами.

Таблица 3.1. Функции Win32 API для работы с процессами

Функция	Описание
`CreateProcess()`	Создание нового процесса и его первичного потока. Эта функция замещает функцию `WinExec()`, использовавшуюся в Windows 3.11
`ExitProcess()`	Выход из текущего процесса, прекращение его выполнения и остановка всех его потоков
`GetCurrentProcess()`	Получение псевдодескриптора текущего процесса. Псевдодескриптор представляет собой специальный дескриптор, интерпретируемый как дескриптор текущего процесса. Настоящий дескриптор может быть получен с помощью функции `DuplicateHandle()`
`DuplicateHandle()`	Создание копии дескриптора объекта ядра
`GetCurrentProcessID()`	Получение идентификатора текущего процесса, который уникальным образом идентифицирует процесс в системе во время его работы
`GetExitCodeProcess()`	Получение статуса завершения указанного процесса
`GetPriorityClass()`	Получение класса приоритета указанного процесса. Это значение и значение приоритета каждого из потоков процесса определяют основной уровень приоритета каждого потока
`GetStartupInfo()`	Получение содержимого структуры `TStartupInfo`, инициируемой при создании процесса
`OpenProcess()`	Получение дескриптора существующего процесса, задаваемого его идентификатором
`SetPriorityClass()`	Установка класса приоритета процесса
`TerminateProcess()`	Прекращение выполнения процесса и всех его потоков
`WaitForInputIdle()`	Приостановка процесса на время ожидания ввода пользователя

Одни функции Win32 API требуют указания дескриптора экземпляра приложения, а другие — дескриптора модуля. В 16-битовых версиях Windows эти значения отличались, однако в среде Win32 это не так. Каждый процесс получает свой собственный дескриптор экземпляра. В приложениях Delphi это значение хранится в глобальной переменной `HInstance`. Так как переменная `HInstance` и дескриптор модуля приложения представляют одно и то же значение, можно передавать переменную `HInstance` в качестве параметра и функциям, требующим указания дескриптора модуля — например, функции `GetModuleFileName()`, возвращающей имя файла модуля, заданного его дескриптором.

Переменная `HInstance` не будет содержать дескриптор модуля для кода, откомпилированного в пакет. В этом случае дескриптор основного модуля программы будет содержаться в переменной `MainInstance`, а в переменную `HInstance` будет помещен дескриптор модуля, содержащего исполняемый код.

Еще одно отличие между Win32 и 16-битовой версией Windows связано с использованием глобальной переменной `HPrevInst`. В 16-битовых версиях Windows она содержит дескриптор ранее запущенного экземпляра этого же приложения. Данную переменную можно было использовать, например, для предотвращения повторного запуска приложения. Однако подобная технология не

работает в среде Win32 — здесь каждый процесс имеет собственное адресное пространство в 4 Гбайта и не видит других процессов. Поэтому значение переменной `HPrevInst` в среде Win32 всегда равно 0. Однако это не значит, что задача определения наличия запущенных ранее экземпляров этого же приложения неразрешима принципиально (более полную информацию о методах ее решения можно найти в главе 13, "Дополнительный инструментарий разработчика").

Типы объектов ядра

В Win32 существует несколько различных типов объектов ядра. При создании такой объект размещается в адресном пространстве породившего его процесса, которому передается дескриптор вновь созданного объекта. Этот дескриптор не может быть передан другому процессу или повторно использоваться последующими процессами с целью доступа к одному и тому же объекту ядра. Однако с помощью специальных функций Win32 API другой процесс может получить собственный дескриптор уже существующего объекта ядра. Например, функция `CreateMutex()` создает именованный или неименованный блок совместного использования и возвращает его дескриптор. Функция `OpenMutex()` возвращает новый дескриптор для существующего поименованного блока совместного использования. Этой функции передается имя блока совместного использования, дескриптор которого уже был зарегистрирован в системе.

На заметку

При создании объектам ядра могут быть назначены имена, представленные строкой с завершающим нулевым символом, переданной в качестве параметра в соответствующую функцию `CreateXXX()`. Это имя регистрируется в операционной системе, а объект называется именованным. Другие процессы могут получить доступ к этому же объекту ядра посредством вызова функции `OpenXXX()` с передачей ей имени требуемого объекта. Пример использования этой технологии будет приведен в главе 13, "Дополнительный инструментарий разработчика", при демонстрации методов защиты от запуска нескольких экземпляров приложения одновременно.

Если необходимо разрешить доступ к блоку совместного использования со стороны других процессов, то основной процесс должен создать его с помощью функции `CreateMutex()`. В качестве параметра этой функции обязательно следует передать имя, которое будет присвоено создаваемому блоку. Все остальные процессы должны использовать функцию `OpenMutex()`, передавая ей то же самое имя, которое было использовано первым процессом. Функция `OpenMutex()` возвращает дескриптор для того блока совместного использования, имя которого было ей указано. Доступ других процессов к уже существующим объектам ядра может быть тем или иным образом ограничен. Требуемый набор ограничений устанавливается при создании блока с помощью функции `CreateMutex()`. Описание возможных ограничений доступа для каждого типа объекта ядра можно найти в интерактивной справочной системе.

Так как к объектам ядра могут иметь доступ несколько процессов, управление такими объектами осуществляется с помощью счетчика обращений. Значение этого счетчика увеличивается всякий раз, когда другое приложение получает доступ к данному объекту. Завершая работу с внешним объектом, приложение должно вызвать функцию `CloseHandle()`, которая уменьшает счетчик обращений соответствующего объекта.

Объекты GDI и User

В 16-битовых версиях Windows объектами назывались сущности, обращение к которым осуществлялось с помощью дескрипторов. Сказанное не относится к объектам ядра, поскольку в 16-битовых версиях Windows их просто не существовало.

В 16-битовых версиях Windows типов объектов было два: объекты, расположенные в локальной памяти модулей GDI и USER, и объекты, размещенные в глобальной памяти. Примерами объектов GDI (графического интерфейса) могут быть перья, кисти, шрифты, палитры, изображения и области. Примерами объектов USER (интерфейса пользователя) являются окна, классы окон, атомарные объекты и меню.

В этой модели существовала непосредственная связь между объектом и его дескриптором. Дескриптор объекта фактически представлял собой селектор, который при конвертации в указатель адресовал структуру данных, описывающую объект. Эта структура, в зависимости от типа объекта, находилась в сегменте данных модуля GDI или USER. Кроме того, дескриптор объекта ссылался на глобальную память как селектор сегмента глобальной памяти. Поэтому при конвертации в указатель последний указывал на соответствующий блок памяти.

Следствием подобного построения объектов в 16-битовых версиях Windows являлась возможность их разделения. Доступная глобально, таблица локальных дескрипторов (Local Descriptor Table — LDT) содержала указатели на эти объекты. Кроме того, всем приложениям и библиотекам DLL были доступны также сегменты данных GDI и USER. Таким образом, любое приложение или динамическая библиотека могли получить доступ к объекту, используемому другим приложением. Во многих приложениях эти особенности успешно использовались, например, для организации совместного использования памяти.

В среде Win32 работа с объектами GDI и USER проводится иначе, поэтому некоторые ранее успешно работавшие технологии теперь являются несостоятельными. Начнем с того, что в Win32 появились объекты ядра, уже рассмотренные нами выше. Кроме того, в Win32 объекты GDI и USER реализованы иначе, чем в 16-битовых версиях Windows

В среде Win32 объекты GDI больше не используются совместно, как это было в 16-битовых версиях. Объекты GDI хранятся в адресных пространствах отдельных процессов, а не в глобально доступном блоке памяти. (Напомним, что каждому процессу выделяется собственное адресное пространство размером в 4 Гбайта.) Кроме того, каждый процесс имеет собственную таблицу дескрипторов, в которую помещаются и дескрипторы объектов GDI данного процесса. Очень важно помнить об этом постоянно, чтобы не предпринимать попыток передачи дескрипторов объектов GDI в другие процессы.

Ранее мы упоминали, что таблица LDT была доступна всем приложениям. В среде Win32 распределение адресного пространства каждого процесса фиксируется в его собственной LDT. Следовательно, в среде Win32 использование локальных таблиц в точности отвечает их названию и ограничивается только локальным процессом-владельцем.

Хотя имеется возможность вызова функции `SelectObject()` с дескриптором, полученным из другого процесса, успешное использование его — дело случая. В различных процессах объекты с одинаковым дескриптором могут иметь совершенно разное назначение, поэтому мы не рекомендуем применять подобный метод.

Управление дескрипторами GDI осуществляется подсистемой Win32, которая включает проверку корректности объектов GDI, а также освобождение и повторное использование дескрипторов.

Объекты USER работают подобно объектам GDI и управляются своей подсистемой Win32. Основное отличие заключается в том, что таблицы дескрипторов по-прежнему управляются модулем USER и не находятся в адресных пространствах процессов (как в случае таблиц GDI). Поэтому объекты окон, классов окон, атомарных областей и т.д. могут разделяться различными процессами.

Многозадачность и многопоточность

Многозадачность (multitasking) — термин, используемый для описания способности операционной системы поддерживать работу одновременно нескольких приложений. Система обеспечивает такую работу, выделяя каждому приложению определенную порцию квантов времени процессора. В этом смысле многозадачность скорее означает переключение между задачами, чем истинное выполнение ряда задач параллельно. Другими словами, реально операционная система не позволяет нескольким приложениям работать в один и тот же момент времени, а просто организует их последовательное выполнение в течение установленных интервалов времени. Однако, поскольку эти интервалы очень малы, у пользователя создается впечатление параллельной работы различных приложений.

Концепция многозадачности не нова для Windows и была реализована и в предыдущих версиях этой системы. Основное отличие между реализацией многозадачности в Win32 и предыдущих версиях состоит в использовании в новой версии алгоритма вытесняющей многозадачности. В предыдущих версиях многозадачность была невытесняющей — это означает, что при выделении приложениям квантов времени операционная система не использовала системный таймер. Выделение кванта времени следующему приложению происходило только после того, как текущее приложение сообщало о завершении или приостановке своей работы. Иногда это вызывало проблемы, так как отдельное приложение с интенсивной загрузкой процессора продолжительными вычислениями могло практически парализовать работу системы. В этих случаях единственный выход состоял в организации в приложении специального программного контроля с целью исключения монополизации процессора.

Система Win32 гарантирует, что все потоки каждого приложения получат свою порцию времени процессора. Выделение квантов времени ЦП каждому из потоков осуществляется на основе их приоритетов. Более детальное описание принципов работы вытесняющей многозадачности можно найти в главе 11, “Создание многопоточных приложений”.

На заметку

Реализация Win32 в операционных системах Windows NT\2000 позволяет получить истинную многозадачность при работе на многопроцессорных компьютерах. В этом случае кванты времени каждому из приложений могут выделяться на специально отведенном для этого приложения процессоре. Возможен и другой режим, когда каждому из потоков квант времени ЦП может быть выделен на любом доступном процессоре, из числа установленных в многопроцессорном компьютере.

Многопоточность (multithreading) характеризует способность приложения к многозадачности в рамках одного процесса. Это означает, что приложение способно выполнять несколько различных задач одновременно. Процесс может иметь несколько *потоков* (thread), каждый из которых выполняет свой собственный код. Потоки могут зависеть один от другого и, следовательно, нуждаться в синхронизации. Например, часто необходимо гарантировать, что один потоков завершит свою работу до того, как результаты его работы будут использованы другими потоками. Для подобной координации действий потоков используются специальные технологии *синхронизации потоков*. Более детальное описание принципов работы с потоками можно найти в главе 11, “Создание многопоточных приложений”.

Управление памятью в Win32

Система Win32 вводит вас в мир 32-разрядной плоской модели памяти. Теперь у программистов на языке Pascal появилась реальная возможность создавать большие массивы без риска возникновения ошибки компиляции:

```
BigArray = array [1..100000] of Integer;
```

В последующих разделах мы обсудим основные принципы принятой в среде Win32 модели памяти и методов управления ею.

Плоская модель памяти

В 16-разрядной среде использовалась сегментированная модель памяти. Все адреса в ней представлялись в виде пары значений *сегмент:смещение*. Сегмент задает некий базовый адрес, а смещение — количество байтов от базового адреса до требуемого. Такая модель памяти нередко сбивала с толку неподготовленных программистов, а также крайне осложняла работу с объектами размером более 64 Кбайт, которые не могли быть размещены в одном сегменте.

В случае плоской модели памяти все эти ограничения снимаются. Каждый процесс получает собственное 4-гигабайтовое адресное пространство и может размещать в нем объекты огромных размеров. Кроме того, каждый адрес представляется единственным уникальным числовым значением.

Работа системы управления памятью Win32

Весьма вероятно, что на вашем компьютере установлено менее 4 Гбайт памяти. Каким же образом система Win32 позволяет процессам использовать больше памяти, чем реально установлено в компьютере? Дело в том, что 32-разрядные адреса не указывают непосредственно на ячейки физической памяти. На самом деле процессы в системе Win32 работают с *виртуальными адресами*.

Использование виртуальной памяти позволяет выделить каждому процессу 4-гигабайтовое адресное пространство, в котором область верхних адресов размером в 2 Гбайт выделена для использования Windows, а нижние 2 Гбайт — это область, где размещается ваше приложение и где вы можете выделять память. Одно из главных достоинств такой схемы — полное разделение памяти процессов: один процесс не может получить непосредственный доступ к памяти другого процесса. Например, виртуальный адрес `$54545454` в одном процессе указывает на совершенно иную физическую область памяти, чем тот же адрес в другом процессе.

Очень важно понимать, что на самом деле процесс не получает 4 Гбайт памяти, он просто имеет возможность обратиться к памяти в этом диапазоне адресов. Количество памяти, реально доступное процессу, зависит от того, какое количество ОЗУ имеется в компьютере и какой участок диска доступен файлу страничного обмена. Для предоставления памяти приложениям доступные операционной системе физическая память и файл страничного обмена разбиваются на страницы. Размер страницы зависит от используемой платформы. На платформе Intel используются страницы размером в 4 Кбайт; в случае процессоров Alpha — 8 Кбайт; стандартный размер страниц PowerPC и MIPS также равен 4 Кбайт. Операционная система при необходимости перемещает страницы между файлом страничного обмена и оперативной памятью. При этом для преобразования виртуального адреса процесса в реальный физический адрес используется специальная карта страниц. Мы не будем вдаваться в подробности этого процесса — достаточно иметь общее представление о механизме использования виртуальной памяти.

В среде Win32 разработчику предлагается три различных способа использования памяти — работа с виртуальной памятью, отображаемыми файлами и кучей.

Виртуальная память

Система Win32 предоставляет набор низкоуровневых функций, позволяющих манипулировать виртуальной памятью процесса. Эта память может находиться в одном из трех состояний.

Свободная	Память, которая может быть зарезервирована или размещена.
Зарезервированная	Память в некотором диапазоне адресов, которая зарезервирована для будущего использования. Эта память защищена от последующих за-

просов на выделение памяти. Однако она остается недоступной процессу, поскольку физическая память этому диапазону адресов будет выделена только при его размещении. Для резервирования памяти используется функция `VirtualAlloc()`.

Размещенная — Память, которая была зарезервирована и которой выделена физическая память. Такая память может использоваться процессом. Размещение памяти выполняется также с помощью функции `VirtualAlloc()`.

Для работы с виртуальной памятью Win32 предоставляет набор функций группы `VirtualXXX()`, краткие сведения о которых приведены в табл. 3.2. Подробное описание всех этих функций имеется в интерактивной справочной системе.

Таблица 3.2. Функции для работы с виртуальной памятью

Функция	Описание
`VirtualAlloc()`	Резервирует и/или размещает страницы памяти в виртуальном адресном пространстве процесса
`VirtualFree()`	Освобождает страницы памяти в виртуальном адресном пространстве процесса
`VirtualLock()`	Блокирует диапазон виртуальных адресов процесса, защищая его от выталкивания в файл страничного обмена на диске
`VirtualUnlock()`	Деблокирует указанный диапазон адресов памяти, разрешая его выталкивание в файл страничного обмена
`VirtualQuery()`	Возвращает информацию о диапазоне страниц в виртуальном адресном пространстве вызывающего процесса
`VirtualQueryEx()`	Возвращает ту же информацию, что и функция `VirtualQuery()`, но для некоторого указанного процесса
`VirtualProtect()`	Изменяет права доступа к определенному диапазону размещенной памяти в адресном пространстве вызывающего процесса
`VirtualProtectEx()`	Изменяет права доступа к определенному диапазону размещенной памяти в адресном пространстве некоторого указанного процесса

На заметку Функции `xxxEx()`, описанные в табл. 3.2, могут использоваться только процессами, имеющими отладочные привилегии по отношению к определенному процессу. Использовать эти функции в обычной практике нет необходимости.

Отображаемые в памяти файлы

Отображаемые в памяти файлы (memory-mapped files) позволяют обращаться к данным дисковых файлов так же, как к динамически выделенной памяти. Это осуществляется путем отображения всего файла или его части в диапазоне адресов процесса. После этого доступ к данным в файле может быть получен с помощью обычного указателя. Более детально эти объекты рассматриваются в главе 12, “Работа с файлами”.

Кучи

Кучи (heaps) представляют собой непрерывные блоки памяти, в пределах которых могут быть выделены меньшие блоки. Кучи позволяют более эффективно выделять динамическую память и манипулировать ею. Для работы с кучами система Win32 предоставляет набор функций `HeapXXX()`, краткие сведения о которых приведены в табл. 3.3, а подробное описание — в интерактивной справочной системе Delphi.

Таблица 3.3. Функции работы с кучей

Функция	Описание
`HeapCreate()`	Резервирует непрерывный блок в виртуальном адресном пространстве вызывающего процесса и выделяет физическую память для определенной начальной части этого блока
`HeapAlloc()`	Выделяет в куче блок неперемещаемой памяти
`HeapReAlloc()`	Позволяет изменить размер выделенного блока памяти кучи, что позволяет изменить ее свойства
`HeapFree()`	Освобождает блок памяти, выделенный из кучи с помощью функции `HeapAlloc()`
`HeapDestroy()`	Уничтожает объект кучи, созданный с помощью функции `HeapCreate()`

На заметку

Следует иметь в виду различия в реализации функций Win32 в средах Windows NT/2000 и Windows 95/98. В основном они касаются безопасности и скорости. Например, менеджер памяти Windows 95/98 менее мощный, чем менеджер Windows NT/2000, который хранит существенно большее количество информации о блоках памяти.
При разработке программ с использованием функций Win32 API постоянно помните о подобных различиях. Подробности о платформенно-зависимых различиях в использовании функций Win32 можно найти в интерактивной справочной системе — не забывайте к ней своевременно обращаться.

Обработка ошибок в Win32

Большинство функций Win32 API возвращает булево значение `True` или `False`, указывающее, соответственно, на удачное или неудачное ее выполнение. Если функция вернула значение `False` (вызов неудачен), следует использовать функцию `GetLastError()`, позволяющую получить код последней ошибки, происшедшей в данном потоке.

На заметку

К сожалению, не все функции Win32 API устанавливают код ошибки, который будет доступен функции `GetLastError()`. В частности, это касается многих функции GDI.

Код ошибки сохраняется для каждого потока в отдельности, поэтому функция `GetLastError()` должна быть вызвана в том же контексте потока, в котором произошла ошибка. Вот пример использования этой функции:

```
if not CreateProcess(CommandLine, nil, nil, nil, False,
  NORMAL_PRIORITY_CLASS, nil, nil, StartupInfo, ProcessInfo) then
```

```
raise Exception.Create('Error creating process:' +
  IntToStr(GetLastError));
```

Модуль SYSUTILS Delphi 5 содержит стандартный класс исключительной ситуации и вспомогательную функцию для конвертирования системных ошибок в исключения. Речь идет о функциях Win32Check() и RaiseLastWin32Error(), генерирующих исключительную ситуацию EWin32Error. Используйте эти вспомогательные функции вместо написания собственного кода проверки результатов вызова функций Win32 API.

В приведенном примере кода предпринимается попытка создать процесс, определяемый строкой с завершающим нулевым символом CommandLine. Обсуждение метода CreateProcess() мы оставим для последующих глав, а сейчас сосредоточим внимание на функции GetLastError(). Если выполнение функции CreateProcess() завершится неудачно, генерируется исключительная ситуация. Это исключение будет отражать код последней зафиксированной при выполнении вызываемой функции ошибки, полученный с помощью функции GetLastError(). Подобный подход вы можете использовать в своих приложениях.

Коды ошибок, возвращаемые функцией GetLastError(), как правило, описаны в статье интерактивной справочной системы по той функции, при выполнении которой была зафиксирована ошибка. Другими словами, коды ошибок, которые могут возникнуть при выполнении функции CrateMutex(), описаны в статье интерактивной справочной системы Win32, посвященной данной функции.

Резюме

В этой главе вы познакомились с Win32 API, узнали об объектах ядра и о том, как Win32 управляет памятью. Кроме того, мы обсудили, какие существуют возможности работы с памятью в приложениях. Программистам Delphi необязательно знать все тонкости функционирования системы Win32. Однако понимание основ работы системы Win32 и знание функциональных возможностей ее API позволит более осознанно, а значит, и более успешно создавать собственные приложения.

Строение приложения и концепции конструирования

Глава

4

В этой главе речь пойдет об управлении проектами и архитектуре приложений, принятой в среде Delphi. Вы узнаете, как правильно использовать формы в создаваемом приложении и как управлять их поведением и характеристиками. Обсуждаемые в этой главе технологии включают инициализацию приложения, повторное использование и наследование форм, а также расширения пользовательского интерфейса. Здесь же мы рассмотрим базовые классы приложений Delphi 5 — `Tapplication`, `Tform`, `Tframe` и `Tscreen`. Мы покажем, почему понимание этих концепций так важно для выбора правильной архитектуры создаваемого приложения.

Среда Delphi и архитектура проекта

Для правильного построения и управления проектами Delphi 5 необходимы по меньшей мере два важных элемента — отличное знание среды разработки Delphi и понимание присущей приложениям Delphi архитектуры. Мы не будем останавливаться на работе интегрированной среды разработки Delphi — это достаточно хорошо освещено в документации. Вместо этого мы обсудим, как наиболее эффективно использовать среду разработки для управления создаваемыми проектами. Мы поговорим также об архитектуре, свойственной всем приложениям Delphi, и о том, как правильно применять полученные знания при разработке приложений.

Предлагаем начать со знакомства со средой разработки Delphi 5. Книга написана в предположении, что вы уже знакомы с Delphi 5 IDE и прочли стандартную документацию по Delphi 5. Также настоятельно рекомендуем вам пройтись по всем пунктам меню интегрированной среды Delphi и раскрыть каждое диалоговое окно. Если вы встретите нечто незнакомое, вызовите справку по интересующему вас вопросу и прочтите ее (заодно убедитесь, насколько хорошо создана справочная система Delphi). Поверьте, не будет потрачено напрасно затраченное на эту работу время.

Справочная система Delphi 5 стоит отдельного упоминания. Это одно из наиболее мощных средств получения необходимой информации, находящееся в вашем распоряжении, а потому вы просто обязаны уметь свободно с ним обращаться.

Справочная система Delphi 5 содержит информацию буквально обо всем — от использования интегрированной среды разработки до тонкостей вызова той или иной функции Win32 API и формата сложных структур Win32. Контекстная помощь вызывается с помощью комбинации клавиш <Ctrl+F1> в тот момент, когда курсор в окне редактора находится на соответствующем слове, означающем ту или иную тему или название функции. Второй способ вызова контекстной справки — кнопка Help или клавиша <F1>, которые вызовут справку о компоненте, имеющем в настоящий момент фокус ввода. И, наконец, вы можете просто перемещаться по справочной системе, начав путь с выбора команды Help⇨Help.

Файлы, входящие в состав проекта Delphi 5

Проект Delphi 5 состоит из нескольких связанных файлов. Одни из них создаются при построении форм, другие отсутствуют до момента компиляции проекта. Для эффективного управления проектами Delphi 5 необходимо знать назначение каждого из указанных файлов (об этом можно прочесть как в стандартной документации, так и в оперативной справочной системе). Неплохая мысль — обратиться к документации прямо сейчас, чтобы освежить в памяти соответствующие разделы перед дальнейшим чтением этой главы.

Файл проекта

Файл проекта создается во время работы над проектом и имеет расширение `.dpr`. Этот файл содержит исходный текст главной (main) программы приложения. Именно здесь создается экземпляр главной формы приложения, а также инициируются все прочие автоматически создаваемые формы. Вам редко потребуется редактировать этот файл, за исключением случаев использования особых подпрограмм инициализации, предназначенных для вывода на экран фирменной заставки или осуществления других действий, которые должны быть выполнены непосредственно при старте программы. Вот код типичного файла проекта:

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' { Form1} ;
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(Tform1, Form1);
  Application.Run;
end.
```

Программисты на языке Pascal без труда узнают в нем обычный файл с текстом программы. Обратите внимание на перечисление в операторе `uses` списка модулей форм (сейчас список состоит из единственного модуля `unit1` с единственной формой). Подобным образом в файле проекта перечисляются все используемые в приложении модули форм.

Строка `{$R *.RES}` описывает используемый приложением файл ресурсов. Эта строка сообщает компилятору, что он должен связать с приложением файл ресурсов, имеющий то же имя, что и основной файл приложения, но с расширением имени `.res`. Этот файл содержит, например, пиктограмму приложения и информацию о версии.

Наконец, блок `begin … end` представляет собой основной выполняемый код — в нашем примере здесь создается главная форма `Form1`, которая выводится на дисплей посредством вызова метода `Application.Run()`. В этот блок можно добавить собственный код, как будет показано ниже, в этой же главе.

Файлы модулей проекта

Модули представляют собой файлы с исходными текстами программ на языке Pascal и имеют расширения имени `.pas`. Существует три основных типа этих файлов: модули форм/данных или кадров, модули компонентов и модули общего назначения.

- *Модули форм/данных или кадров* автоматически генерируются Delphi 5 для каждой отдельно создаваемой формы или кадра. Невозможно определить в одном модуле две формы и при этом иметь возможность работать с ними обеими в конструкторе форм. При обсуждении формата файлов форм мы не будем делать различия между модулями форм, данных или кадров.
- *Модули компонентов* представляют собой модули, генерируемые Delphi 5 при создании каждого нового компонента.

- *Модули общего назначения* могут быть созданы для различных типов данных, переменных, процедур и классов, которые требуется сделать доступными различным приложениям.

С детальным описанием модулей мы познакомимся ниже в этой главе.

Файлы форм

Файлы форм содержат двоичное представление форм. При создании новой формы Delphi 5 записывает файл формы с расширением `.dfm` одновременно с модулем формы на языке Pascal (с расширением `.pas`). В тексте модуля любой созданной формы обязательно будет присутствовать строка, подобная следующей:

```
{$R *.DFM}
```

Эта строка указывает компилятору на необходимость связать с проектом соответствующий файл формы, имеющий то же имя, что и файл модуля формы, но с расширением имени `.dfm`.

Как правило, самостоятельно редактировать текст файла формы вам не потребуется (хотя такая возможность и имеется). Для редактирования файла формы в текстовом представлении достаточно просто загрузить его в окне редактора Delphi 5. Выберите команду File⇨Open и установите в раскрывшемся окне режим отображения только файлов форм Delphi (`.dfm`). Другой вариант — щелкнуть правой кнопкой мыши в поле конструктора форм и выбрать в раскрывшемся контекстном меню команду View as Text. В любом случае на экран будет выведено окно редактора Delphi с текстовым представлением файла формы.

Просмотр текстового представления бывает удобен, в частности, тем, что при этом вы видите все свойства и компоненты формы. Редактирование текстового представления позволяет вам, например, изменять тип того или иного компонента. Предположим, например, что файл формы содержит следующее определение компонента `Tbutton`:

```
object Button1: Tbutton
  Left = 8
  Top = 8
  Width = 75
  Height = 25
  Caption = 'Button1'
  TabOrder = 0
end
```

Если в первой строке фразу `object Button1: Tbutton` вы замените фразой `object Button1: Tlabel`, то тем самым вы смените тип объекта и при просмотре формы в конструкторе на месте кнопки увидите надпись.

На заметку

Изменение типа компонентов в файле формы может привести к ошибкам при чтении свойств того или иного компонента. Например, изменение типа `Tbutton` (который имеет свойство `TabOrder`) на тип `Tlabel` (который такового свойства не имеет) приведет к указанной ошибке. Однако, если вы работаете в Delphi, особого повода для беспокойства нет, так как Delphi устранит ссылку на свойство при очередном сохранении формы.

При прямом редактировании файла формы следует проявлять исключительную осторожность, так как его можно повредить настолько серьезно, что в Delphi 5 больше не сможет его открыть.

На заметку

В Delphi 5 появилась новая возможность сохранять файлы форм в текстовом формате. Это было сделано с целью предоставления возможности редактирования файлов форм с помощью других общераспространенных инструментов, например, таких как `Notepad.exe`. Для этого достаточно щелкнуть в поле конструктора форм правой кнопкой мыши и выбрать в раскрывшемся контекстном меню команду Text DFM.

Файлы ресурсов

Файлы ресурсов содержат двоичные данные, называемые также *ресурсами*, которые связаны с выполняемым файлом приложения. Файл `.res`, автоматически создаваемый Delphi 5, содержит пиктограмму приложения, информацию о версии приложения и другие сведения. Добавить другие ресурсы в новое приложение можно посредством создания отдельного файла ресурсов с последующим связыванием его с проектом. Такой файл ресурсов может быть создан с помощью редактора ресурсов — например, входящей в состав Delphi 5 утилиты Image Editor или приложения Resource Workshop.

Не редактируйте файлы ресурсов, автоматически создаваемые Delphi во время компиляции. Такие изменения будут потеряны при следующей же компиляции приложения. Если вы хотите добавить ресурсы в свое приложение, создайте файл ресурсов с отличным от имеющихся именем, а затем свяжите его с проектом с помощью директивы `$R`, как показано ниже.

```
{$R MYRESFIL.RES}
```

Файлы опций проекта и установок рабочего стола

Файлы опций проекта (с расширениями `.dof`) хранят установки, определенные во вкладках окна Project Options. (Раскрыть это окно можно с помощью команды Project⇨Options.) Такой файл создается при первом же сохранении проекта и обновляется при каждом последующем.

Файл установок рабочего стола (с расширением `.dsk`) хранит параметры, установленные во вкладках окна Environment Options и определяющие характеристики рабочего стола Delphi IDE. (Раскрыть это окно можно с помощью команды Tools⇨Environment Options.) Параметры рабочего стола отличаются от параметров проекта, так как последние являются специфическими для отдельного проекта, а первые определяют настройку визуальной среды проектирования Delphi.

Повреждение файлов `.dsk` и `.dof` может привести к непредсказуемым последствиям — например, к ошибке защиты памяти при компиляции проекта. Если это произойдет, просто удалите оба указанных файла — при этом опции будут вновь установлены по умолчанию, а удаленные файлы перезаписаны при первом же сохранении проекта.

Файлы резервных копий

Delphi 5 создает файлы резервных копий для файла проекта и файлов модулей при их втором и всех последующих сохранениях. Файлы резервных копий содержат предыдущую версию содержимого соответствующего файла и имеют расширения `.~dp` и `.~pa` соответственно для файлов проекта и файлов модулей.

Файл резервной копии создается и для двоичного файла формы при втором и всех последующих его сохранения. Он имеет расширение имени `.~df`.

Если удалить эти файлы, ничего страшного не произойдет. В худшем случае будет утеряна возможность возврата к предыдущей версии программы. Можно вообще отказаться от их создания. Для этого следует сбросить флажок опции Create Backup Files во вкладке Display диалогового окна Editor Options.

Файлы пакетов

Пакеты представляют собой обычные библиотеки DLL, содержащие программный код, который может совместно использоваться многими приложениями. Однако применение пактов в Delphi отличается тем, что они позволяют разделять компоненты, классы, данные и код также между модулями приложения. Это означает, что можно существенно сократить размеры файлов приложения за счет динамического использования сохраняемых в пакетах компонентов, вместо прямого помещения их в файлы приложения. О пакетах речь пойдет в последующих главах, а пока заметим только, что исходные файлы пакетов имеют расширение `.dpk`. При трансляции из них создаются `.bpl`-файлы (файлы BPL функционально аналогичны файлам DLL). Файлы `.bpl` могут быть составлены из нескольких модулей или DCU-файлов (скомпилированных модулей). Двоичное представление DPK-файла содержит все включаемые модули и заголовок пакета и представляет собой файл с расширением имени `.dcp`. Не расстраивайтесь, если изложенная выше информация покажется вам слишком запутанной — все необходимые пояснения будут даны позже.

Советы по управлению проектом

Есть множество путей оптимизации процесса разработки приложений с использованием специальных приемов улучшения организации и повторного использования кода. О некоторых из них и пойдет речь в следующих разделах.

Один проект — один каталог

Не следует смешивать создаваемые проекты. Файлы одного проекта не должны быть в том же каталоге, что и файлы другого проекта. Это один из простейших советов, но не последний по важности.

Отметим, что каждый проект, помещенный на компакт-диск, прилагаемый ко второму тому (см. также `www.williamspublishing.com`.), размещен в отдельном каталоге. Рекомендуем всегда придерживаться этого простого принципа.

Соглашения по присвоению имен

Очень полезно выработать некоторое соглашение о присвоении стандартных имен файлам, входящим в состав создаваемых проектов. Полезные рекомендации можно найти в документе DDG “Coding Standard Document”, сведения о котором приводятся в главе 6, “Стандарты программирования, принятые в этой книге”. Предлагаемые в нем решения использовались авторами во всех проектах, представленных в данной книге.

Модули с разделяемым кодом

Можно организовать одновременное использование некоторых подпрограмм в различных приложениях, переместив их в отдельные модули, доступные для различных проектов. Обычно для таких модулей выделяется собственный каталог. При необходимости исполь-

зования определенной подпрограммы, помещенной в один из модулей подобного каталога, достаточно просто указать имя требуемого модуля в объявлении uses создаваемой программы.

Для облегчения поиска модуля внесите соответствующий каталог в поле Search Path, расположенном во вкладке Directories/Conditionals диалогового окна Project Options.

При использовании Project Manager в существующий проект можно добавить модуль и из другого каталога. Необходимые меры по добавлению пути будут предприняты автоматически.

Для пояснения методов использования модулей-утилит в листинге 4.1 приведен текст небольшого модуля StrUtils.pas, включающего единственную функцию обработки строк. На практике в модуле обычно содержится несколько функций, однако для нашего примера будет достаточно и одной. Назначение самой функции поясняется в комментариях.

Листинг 4.1. Модуль StrUtils.pas

```
unit strutils;
interface
function ShortStringAsPChar(var S: ShortString): PChar;
implementation
function ShortStringAsPChar(var S: ShortString): PChar;
{ Эта функция завершает стандартную строку нулевым
символом для использования ее в качестве аргумента PChar }
begin
  if Length(S) = High(S) then Dec(S[0]); {Обрезка слишком длинных строк}
  S[Ord(Length(S)) + 1] := #0; {Помещение символа null в конец строки}
  Result := @S[1]; {Возвращение вновь созданной строки типа PChar}
end;
end.
```

Предположим теперь, что имеется модуль SomeUnit.pas, в котором необходимо использовать эту функцию. Для этого достаточно просто добавить имя StrUtils в список uses:

```
unit SomeUnit;
interface
...
implementation
uses
  strutils;
...
end.
```

Теперь мы должны обеспечить Delphi 5 возможность найти требуемый модуль, указав путь к нему в окне свойств проекта, раскрываемом с помощью команды Project⇨Options.

Выполнив эту подготовительную работу, можно будет использовать функцию ShortStringAsPChar() в любом месте раздела реализации модуля SomeUnit.pas. Точно так же следует поместить StrUtils в списки uses всех модулей, где требуется использование функции ShortStringAsPChar(). Недостаточно добавить описание модуля StrUtils только в один из модулей проекта или даже в файл проекта (DPR) приложения, чтобы сделать его доступным в пределах всего приложения.

Так как функция `ShortStringAsPChar()` весьма удобна и, вероятно, будет часто применяться, она является первым претендентом на повторное использование и размещение в модуле-утилите.

Модули для глобальных идентификаторов

Модули являются весьма удобным механизмом описания глобальных идентификаторов проекта. Как уже упоминалось ранее, проект обычно состоит из множества модулей — форм, компонентов и общего назначения. Что делать, если некоторая переменная должна быть доступна повсюду в вашей программе, т.е. во всех ее модулях? Ниже приведена процедура создания модуля для хранения таких глобальных идентификаторов.

1. Создайте новый модуль Delphi 5.
2. Присвойте ему имя, указывающее, что в нем хранятся глобальные идентификаторы (например, `Globals.pas` или `ProjGlob.pas`).
3. Поместите переменные, типы и прочую информацию в раздел `interface` создаваемого модуля. В результате эти идентификаторы станут доступными другим модулям приложения.
4. Для того чтобы сделать глобальные идентификаторы доступными в некотором модуле, просто добавьте имя модуля с глобальными идентификаторами в его список `uses`.

Обеспечение взаимной доступности форм

Каждая форма содержится внутри собственного модуля, а значит, не имеет доступа к переменным, свойствам и методам других форм. Delphi генерирует исходный текст формы в соответствующем `.pas`-файле, объявляя экземпляр такой формы глобальной переменной. Поэтому все, что необходимо для обеспечения доступа, — это добавить имя модуля с некоторой формой в список `uses` модуля той формы, которой необходим доступ к первой форме. Например, если форме `Form1`, определенной в модуле `Unit1.pas`, требуется доступ к форме `Form2`, определенной в модуле `Unit2`, то достаточно просто внести значение `Unit2` в список `uses` модуля `Unit1`:

```
unit Unit1;
interface
...
implementation
uses
  Unit2;
...
end.
```

В результате ссылки на `Form2` можно будет использовать в любом месте раздела `implementation` модуля `Unit1`.

На заметку

При компиляции проекта вам будет предложено автоматически включить имя модуля `Unit2` в предложение `uses` модуля `Unit1`, если в этом модуле имеется хотя бы одна ссылка на экземпляр формы `Form2`.

Множественное управление проектами (группы проектов)

Зачастую создаваемый продукт состоит из нескольких проектов, которые зависят один от другого. Примером подобных проектов могут служить программы отдельных уровней в многоуровневом приложении среды клиент/сервер. Кроме того, как части общего проекта могут рассматриваться библиотеки DLL, используемые в других проектах, — даже в том случае, если каждая из этих библиотек представляет собой отдельный проект.

Delphi 5 предоставляет мощные средства для работы с такими группами проектов. Встроенный менеджер проектов (Project Manager) позволяет скомбинировать несколько проектов Delphi в одну группу. Нет необходимости детально останавливаться на этом процессе, так как он достаточно хорошо описан в стандартной документации Delphi. Мы только хотим подчеркнуть важность возможности работы с группами проектов и наличие специализированного менеджера проектов, предназначенного для этой цели.

И в данном случае правило "один проект — один каталог" остается в силе. Все разделяемые модули, формы и другие элементы одного проекта лучше разместить в отдельном каталоге. Любые разделяемые модули, формы и прочие элементы должны быть помещены в общий каталог, доступ к которому обеспечивается в каждом из отдельных проектов. Ниже приведен пример построения дерева каталогов группы проектов.

```
\DDGBugProduct
\DDGBugProduct\BugReportProject
\DDGBugProduct\BugAdminTool
\DDGBugProduct\CommonFiles
```

В этой структуре двум отдельным проектам выделены собственные каталоги (`BugReportProject` и `BugAdminTool`) и еще один каталог отведен для размещения разделяемых модулей и форм (`CommonFiles`).

Организация дерева каталогов группы проектов важна из соображений эффективности работы с ними, особенно при работе над проектами целой команды разработчиков. Настоятельно советуем выработать стандартные правила размещения файлов проекта еще до начала работы с ним — это предотвратит появление многих ошибок и избавит вас от излишней работы. Для разработки соответствующей структуры можно воспользоваться средствами менеджера проектов.

Базовые классы проектов Delphi 5

Абсолютное большинство проектов Delphi 5 включает по меньшей мере один экземпляр объекта класса `TForm`. Любые VCL-приложения содержат по одному экземпляру объектов классов `TApplication` и `TScreen`. Три упомянутых класса играют важную роль в определении функциональных возможностей любых проектов Delphi 5. В последующих разделах мы познакомимся с их назначением и особенностями, что позволит вам при необходимости изменять поведение этих объектов, принимаемое по умолчанию.

Класс `TForm`

Класс `TForm` — это фокусная точка приложений Delphi 5. В большинстве случаев работа всего приложения строится вокруг использования его главной формы. Как правило, любые другие формы открываются с помощью команды меню или щелчка мышью на кнопках в главной форме.

В Delphi 5 можно использовать функцию автоматического создания форм, что избавит вас от необходимости беспокоиться об их инициализации и уничтожении. Кроме того, можно установить режим динамического создания форм непосредственно в процессе выполнения приложения.

На заметку В Delphi можно создавать и приложения, не использующие форм, — например, консольные приложения, службы и серверы COM. Таким образом, класс `TForm` все же не всегда является той осью, вокруг которой вращается работа всего приложения.

Отображение форм на экране пользователя может быть осуществлено двумя методами: модальным и немодальным. Выбор метода определяется тем, каким образом пользователь должен взаимодействовать с данной формой по отношению к другим формами, выведенным на экран в данный момент.

Отображение модальной формы

Модальная форма выводится на экран таким образом, что при работе с ней пользователь не получит доступа к остальным частям приложения до тех пор, пока эта форма не будет закрыта. Обычно модальные формы связаны с диалоговыми окнами и используются гораздо чаще немодальных. Для вызова формы в модальном режиме используется метод `ShowModal()`. Приведенный ниже код — это пример создания экземпляра формы `TModalForm` и вывода ее на экран как модальной.

```
Begin
  // Создание экземпляра формы
  ModalForm := TModalForm.Create(Application);
  try
    if ModalForm.ShowModal = mrOk then  // Отображение формы в модальном режиме
     { do something };  // Выполнение некоторых действий
  finally
    ModalForm.Free;  // Освобождение объекта формы
    ModalForm := nil; // Уничтожение объекта формы
  end;
end;
```

Здесь показано, как динамически создать экземпляр объекта `TModalForm` и присвоить его переменной `ModalForm`. Следует заметить, что при создании объекта формы динамически необходимо удалить ее из списка доступных форм Auto-Create, расположенного во вкладке Forms диалогового окна Project Options. (Это окно выводится на экран с помощью команды Project⇨Options.) В том случае, когда экземпляр формы уже создан, вывести ее на экран как модальную можно просто посредством вызова метода `ShowModal()`. При этом весь окружающий текст следует удалить:

```
begin
  if ModalForm.ShowModal = mrOk then  // Форма ModalForm уже создана
    {  do something }  // Выполнение некоторых действий
end;
```

Метод `ShowModal()` возвращает значение, назначенное свойству `ModalResult`. По умолчанию это значение равно нулю и соответствует предопределенной константе `mrNone`. Присвоение свойству `ModalResult` ненулевого значения приводит к немедленному закрытию формы и возвращению этого значения как результата выполнения метода `ShowModal()`.

Свойство ModalResult имеется и у объектов кнопок. Если ему заранее назначить некоторое значение, то при выполнении на кнопке щелчка мышью установленное значение будет присвоено свойству ModalResult формы. Это вызовет ее закрытие (если установленное значение не равно mrNone) и возврат в вызывающую функцию того значения, которое было назначено свойству кнопки.

Значение свойству ModalResult формы можно присвоить и во время работы программы:

```
begin
  ModalForm.ModalResult := 100;
    // Присвоение значения ModalResult
    // Это приведет к закрытию формы
end;
```

В табл. 4.1 показаны предопределенные значения свойства ModalResult.

Таблица 4.1. Значения свойства ModalResult

Константа	Значение
mrNone	0
mrOk	idOk
mrCancel	idCancel
mrAbort	idAbort
mrRetry	idRetry
mrIgnore	idIgnore
mrYes	idYes
mrNo	idNo
mrAll	mrNo+1

Вывод немодальной формы

Немодальная форма отображается с помощью вызова метода Show(). В случае отображения формы как немодальной (в отличие от модальных форм) пользователь сможет переключаться между нею и другими формами приложения. Назначение немодальных форм состоит в предоставлении пользователю возможности одновременно работать с различными частями одного и того же приложения, представленными различными формами. Вот пример динамического создания и использования немодальной формы:

```
Begin
// Проверка на наличие немодальной формы Modeless
  if not Assigned(Modeless) then begin
    Modeless := TModeless.Create(Application);  // Создание формы
    Modeless.Show  // Вывод формы как немодальной
  end;

end;  // Форма не выводится - ее объект уже существует
```

В приведенном коде также показано, каким образом предупредить создание нескольких экземпляров одной и той же формы. Ведь при работе немодальной формы пользователь может взаимодействовать с другими частями приложения, а следовательно, ничто не мешает ему вновь

обратиться к команде меню, которая выводит уже выведенную форму TModeless. Очень важно корректным образом организовать процессы создания и удаления объектов форм.

Еще один немаловажный момент — при закрытии немодальной формы (с помощью команды меню или щелчка на кнопке закрытия в верхнем правом углу формы) форма автоматически не освобождается. Экземпляр формы остается в памяти до закрытия главной формы, т.е. фактически до прекращения работы приложения. В предыдущем примере оператор в фразе then условного выражения обеспечивает создание и вывод формы только в том случае, если она еще не была создана. В противном случае эти действия блокируются, поскольку объект формы уже существует (хотя сама форма уже может быть закрыта). Все это прекрасно, если приложение должно работать именно таким образом. Однако, если же необходимо освобождать форму всякий раз, когда пользователь ее закрывает, потребуется подготовить подпрограмму обработки ее события OnClose, параметру Action которого должно быть присвоено значение caFree. Это укажет подпрограммам библиотеки VCL на необходимость освободить форму при ее закрытии:

```
procedure TModeless.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;  // Освобождение экземпляра объекта формы при ее закрытии
end;
```

Приведенный выше пример решает проблему освобождения формы при ее закрытии. Однако существует еще одна проблема. Рассмотрим выполнение следующей строки из приведенного выше примера отображения немодальной формы:

```
if not Assigned(Modeless) then begin
```

Этот оператор проверяет наличие экземпляра объекта класса TModeless по значению ссылающейся на него переменной Modeless. Фактически проверяется равенство переменной Modeless значению nil. Хотя значение переменной Modeless при первом выполнении этой подпрограммы действительно будет равно nil, однако при всех последующих входах в данную подпрограмму после очередного удаления формы TModeless оно уже не будет равно этому значению. Дело в том, что подпрограммы библиотеки VCL автоматически не присваивают значение nil переменной Modeless при освобождении формы. Поэтому это действие должно выполняться самой программой.

В отличие от модальной формы, программа не может прямо установить момент уничтожения немодальной формы. Следовательно, нельзя уничтожать форму в пределах той подпрограммы, которая ее создает. Пользователь может закрыть форму в любой момент работы с приложением. Поэтому присвоение переменной Modeless значения nil должно выполняться самим классом TModeless. Наилучшим местом для выполнения подобных действий является подпрограмма обработки события OnDestroy класса TModeless:

```
procedure TModeless.FormDestroy(Sender: TObject);
begin
  Modeless := nil;  // Сброс значения переменной при уничтожении формы
end;
```

И теперь при каждом уничтожении объекта формы переменной Modeless будет присваиваться значение nil и результаты выполнения метода Assigned() всегда будут корректны. Тем не менее не забывайте осуществлять контроль за тем, чтобы в каждый момент времени в программе существовало не более одного экземпляра класса TModeless, как это сделано в приведенном ранее примере.

Избегайте следующей ошибки при использовании немодальных форм:

```
begin
  Form1 := TForm1.Create(Application);
  Form1.Show();
end;
```

Подобное решение может вызвать бесполезный расход памяти, поскольку при каждом создании экземпляра формы перекрывается текущее значение указателя `Form1`. В результате неоднократного выполнения этого кода вы получите формы, не связанные ни с какими переменными. Хотя доступ к каждому из созданных экземпляров форм может быть осуществлен с помощью списка `Screen.Forms`, рекомендуется избегать подобных решений. Передача же в конструктор `Create()` параметра `nil` приведет к тому, что даже такой метод доступа к потерянным формам будет невозможен.

На компакт-диске, прилагаемом ко второму тому (см. также `www.williamspublishing.com`.), имеется проект `ModState.dpr`, иллюстрирующий использование как модальных, так и немодальных форм.

Управление пиктограммами и рамками форм

Класс `TForm` имеет свойство `BorderIcons`, представляющее собой множество, которое может содержать значения `biSystemMenu`, `biMinimize`, `biMaximize` и `biHelp`. С помощью установки из этих значений в `False` можно удалять из формы соответственно системное меню, кнопки максимизации, минимизации и справки. (Тем не менее любые формы всегда будут иметь стандартную кнопку Windows 95/98 Close.)

Кроме того, изменяя значения свойства `BorderStyle`, можно изменять тип рамки формы. Это свойство объявляется следующим образом:

```
TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog,
↳bsSizeToolWin,bsToolWindow);
```

Установка свойству `BorderStyle` того или иного значения придает форме следующие характеристики:

- `bsDialog` — неизменяемый размер, только одна кнопка Close;
- `bsNone` — неизменяемый размер, отсутствие рамки, отсутствие кнопок;
- `bsSingle` — неизменяемый размер; наличие всех стандартных кнопок. Если одно из свойств — `biMinimize` или `biMaximize` — равны в `False`, в форме появляются обе кнопки, но только одна из них будет активна. Если оба свойства равны `False`, кнопки в форме отсутствуют. Если свойство `biSystemMenu` равно `False`, кнопок в форме не будет;
- `bsSizeable` — изменяемый размер. Могут присутствовать все кнопки. Поведение и появление кнопок определяется теми же правилами, которые применяются в случае значения `bsSingle`;
- `bsSizeToolWin` — изменяемый размер; присутствует только кнопка Close; маленькая полоса заголовка;
- `bsToolWindow` — неизменяемый размер; присутствует только кнопка Close; маленькая полоса заголовка.

На заметку

Изменения значений свойств BorderIcon и BorderStyle не проявляются во время конструирования. Они будут отображены только во время работы приложения. То же самое справедливо и для большинства других свойств класса TForm. Такое поведение легко объяснимо — интересно было бы представить вашу работу с формой, если бы изменение ее свойства Visible в False сработало при конструировании!

Надоедливые заголовки

Вероятно, вы уже заметили, что ни одно сочетание описанных свойств не позволяет создать форму без заголовка, но с изменяемыми размерами. Однако это не значит, что такой возможности не существует, — просто следует использовать некоторые особые, еще не рассмотренные методы. Необходимо переопределить метод CreateParams() объекта формы и установить набор стилей, необходимых для вывода окна с подобными свойствами. Как это сделать, показано в следующем примере:

```
unit Nocapu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs;
type
  TForm1 = class (TForm)
public
  { Переопределение метода CreateParams }
  procedure CreateParams(var Params: TCreateParams); override;
end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params); {Вызов унаследованного метода}
  Params.Style := WS_THICKFRAME or WS_POPUP or WS_BORDER;
end;
end.
```

Дополнительную информацию о методе CreateParams можно найти в главе 21 второго тома, "Создание пользовательских компонентов в Delphi".

Пример использования формы с изменяемым размером, но без рамки, можно найти в проекте NoCaption.dpr, имеющемся на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.). Кроме того, в этом приложении демонстрируется, как можно перехватить сообщение WM_NCHITTEST с целью поддержки возможности перетаскивания формы без заголовка с помощью мыши.

Взгляните на проект BrdrIcon.dpr, также помещенный на компакт-диск, прилагаемый ко второму тому (см. также www.williamspublishing.com.). В нем демонстрируются методы изменения значений свойств BorderIcon и BorderStyle непосредственно в ходе выполнения программы, что позволяет получить интересные визуальные эффекты. В листинге 4.2 представлен текст описания главной формы этого проекта.

Листинг 4.2. Главная форма проекта `BorderStyle/BorderIcon`

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    gbBorderIcons: TGroupBox;
    cbSystemMenu: TCheckBox;
    cbMinimize: TCheckBox;
    cbMaximize: TCheckBox;
    rgBorderStyle: TRadioGroup;
    cbHelp: TCheckBox;
    procedure cbMinimizeClick(Sender: TObject);
    procedure rgBorderStyleClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
procedure TMainForm.cbMinimizeClick(Sender: TObject);
var
  IconSet: TBorderIcons;  // Временная переменная для размещения значений
begin
  IconSet := [];  // Инициализация пустого множества
  if cbSystemMenu.Checked then
    IconSet := IconSet + [biSystemMenu];  // Добавление кнопки biSystemMenu
  if cbMinimize.Checked then
    IconSet := IconSet + [biMinimize];  // Добавление кнопки biMinimaze
  if cbMaximize.Checked then
    IconSet := IconSet + [biMaximize];  // Добавление кнопки biMaximaize
  if cbHelp.Checked then
    IconSet := IconSet + [biHelp];

  BorderIcons := IconSet;               // Помещение результата в
                                        // свойство BorderIcon формы
end;

procedure TMainForm.rgBorderStyleClick(Sender: TObject);
begin
  BorderStyle := TBorderStyle(rgBorderStyle.ItemIndex);
end;

end.
```

На заметку Одни свойства в окне Object Inspector вызывают изменение внешнего вида формы, другие влияют только на ее поведение. Проведите серию экспериментов со значениями свойств, назначение которых вам неясно. Для получения дополнительной информации обратитесь к интерактивной справочной системе Delphi 5.

Повторное использование и наследование визуальных форм

Весьма полезной концепцией, реализованной в Delphi 5, является *наследование визуальных форм*. В первой версии Delphi можно было создать форму и сохранить ее как шаблон, однако это не было истинным наследованием с возможностью доступа к компонентам, методам и свойствам предка. С появлением механизма наследования, все формы-потомки получили возможность использовать тот же код, что и их предок. Единственными дополнительными расходами в этом случае являются лишь вновь добавляемые в классы-потомки методы. Таким образом, помимо прочих преимуществ, наследование позволяет уменьшить размер создаваемого приложения. Еще одним важным достоинством является то, что изменения в работе исходной формы будут немедленно применены и ко всем ее потомкам.

Хранилище объектов

Delphi 5 имеет весьма полезное средство поддержки разработки проектов, позволяющее программистам обеспечивать повторное использование форм, диалоговых окон, модулей данных и шаблонов проектов. Это средство называется *хранилищем объектов* (Object Repository). С помощью хранилища объектов разработчик получает возможность использовать различные элементы, которые были созданы для других проектов. Кроме того, хранилище объектов позволяет максимально увеличить повторное использование разработанных программ за счет их наследования из состава хранящихся в нем объектов. Подробно использование хранилища объектов описано в документации по Delphi 5.

В сетевом окружении можно использовать шаблоны проектов, созданные другими программистами. Это стало возможным благодаря использованию разделяемого хранилища объектов. Его местонахождение определяется в поле Shared Repository вкладки Preference диалогового окна Environment Options (оно выводится на экран с помощью команды Tools⇨Environment Options). Каждый программист на своем компьютере должен назначить одно и то же сетевое устройство, которое будет соответствовать расположению этого каталога. В результате появится возможность доступа к находящимся в нем шаблонам с помощью команды File⇨New.

Наследование формы из другой формы в Delphi 5 — простой процесс, обеспечиваемый встроенными функциями среды разработки. Для создания формы, порожденной другой формой, следует выбрать команду File⇨New. В результате на экране раскроется диалоговое окно New Items, позволяющее просматривать объекты, помещенные в хранилище объектов. Во вкладке Forms этого окна перечислены все формы, помещенные в хранилище.

На заметку Для наследования формы необязательно обращаться в хранилище объектов. Наследовать формы можно также из форм данного проекта. Выберите команду File⇨New, а затем перейдите в раскрывшемся диалоговом окне во вкладку Project. Для целей наследования можно выбрать любую из показанных здесь форм проекта. Отображаемые в этой вкладке формы не помещены в хранилище объектов.

Все отображаемые во вкладке Forms формы были ранее помещены в хранилище объектов. Для извлечения и использования этих форм в вашем проекте предназначены три опции: Copy, Inherit и Use.

Установка переключателя в положение Copy вызовет добавление в проект точной копии выбранной формы. Если помещенная в хранилище объектов форма будет впоследствии изменена, это никак не отразится на ее копии в вашем проекте.

Установка переключателя в положение Inherit вызовет порождение нового класса на базе выбранной в хранилище объектов формы и добавление его в проект. При этом изменения, внесенные впоследствии в сохраняемую в хранилище объектов исходную форму, отразятся и на унаследованных формах, используемых в проекте. Именно этот режим чаще всего используется разработчиками.

Установка переключателя в положение Use приведет к добавлению выбранной формы в проект — так, как будто она была создана как часть этого проекта. При этом внесенные впоследствии в форму изменения будут проявляться во всех проектах, использующих данную форму или ее наследников.

Класс TApplication

Каждая использующая формы программа Delphi 5 включает глобальную переменную Application с типом TApplication. Класс TApplication инкапсулирует саму программу и неявно выполняет множество действий, обеспечивающих работу приложения в среде Windows. Эти функции включают подготовку определения класса окна, создание главного окна приложения, активизацию приложения, обработку сообщений, вывод контекстной справки, обработку быстрых клавиш, назначенных командам меню, а также обработку объектов исключений библиотеки VCL.

На заметку Глобальную переменную Application имеют только приложения, основанные на использовании форм. Другие типы приложений — например, консольные приложения, не включают объектов Application библиотеки VCL.

Как правило, вмешиваться в работу класса TApplication вам не потребуется. Однако в некоторых ситуациях без такого вмешательства не обойтись.

Поскольку класс TApplication не отображается в окне инспектора объектов (Object Inspector), провести модификацию его свойств с помощью этого инструмента не удастся. Вместо этого можно выбрать команду Project⇨Options и перейти в раскрывшемся диалоговом окне во вкладку Application. В этой вкладке можно установить значения некоторых свойств класса TApplication. Кроме того, экземпляр Application класса TApplication создается в ходе выполнения программы, поэтому можно устанавливать его свойства и назначать обработчики событий непосредственно во время работы приложения.

Свойства класса TApplication

Класс TApplication имеет ряд свойств, к которым можно обращаться в процессе работы приложения. Ниже обсуждаются некоторые из этих свойств, а также их использование для изменения принимаемого по умолчанию поведения объекта TApplication с целью расширения возможностей приложения. Заметим также, что свойства TApplication хорошо документированы в интерактивной справочной системе Delphi 5.

Свойство TApplication.ExeName

Свойство ExeName содержит полный путь доступа и имя файла приложения. В процессе выполнения программы данное свойство доступно только для чтения — изменять его значе-

ние нельзя. Вот пример использования этого свойства (после выполнения приведенного ниже кода заголовок приложения будет содержать полное имя файла приложения):

```
Application.MainForm.Caption := Application.ExeName;
```

Для получения имени файла из строки, содержащей полное имя, можно воспользоваться функцией `ExtractFileName()`:

```
ShowMessage(ExtractFileName(Application.ExeName));
```

Для получения только пути используется функция `ExtractFilePath()`:

```
ShowMessage(ExtractFilePath(Application.ExeName));
```

И, наконец, для получения расширения имени файла можно воспользоваться функцией `ExtractFileExt()`:

```
ShowMessage(ExtractFileExt(Application.ExeName));
```

Свойство TApplication.MainForm

В предыдущем разделе был приведен пример использования этого свойства для отображения названия файла приложения в заголовке его главной формы. Свойство `MainForm` всегда указывает на экземпляр класса `TForm`. В результате появляется возможность использовать все свойства и методы этого объекта. Кроме того, пользуясь соответствующим преобразованием типов, можно получить доступ и к свойствам его потомков, например:

```
(MainForm as TForm1).SongTitle := 'The Flood';
```

Свойство `MainForm` доступно только для чтения. Определить, какая из форм будет главной формой создаваемого приложения, можно во вкладке Forms диалогового окна Project Options.

Свойство TApplication.Handle

Свойство `Handle` представляет собой `HWND` (в терминах Win32 API — дескриптор окна). Этот дескриптор окна является владельцем всех окон более высокого уровня данного приложения. Именно свойство `Handle` используется для организации модального доступа к некоторому окну из числа всех прочих открытых окон данного приложения. Доступ к этому свойству вам потребуется в том случае, если понадобится изменить стандартное поведение приложения, обеспечиваемое Delphi по умолчанию. Кроме того, оно может использоваться при обращении к функциям Win32 API, требующим указания дескриптора окна приложения. Подробнее свойство `Handle` мы обсудим ниже в этой главе.

Свойства TApplication.Icon и TApplication.Title

Свойство `Icon` содержит пиктограмму, представляющую приложение в свернутом состоянии. Можно изменить эту пиктограмму, для чего следует подобрать или создать другую пиктограмму, а затем назначить ее свойству `Application.Icon`. Подробнее о добавлении в проект ресурсов (к которым относятся и пиктограммы) речь пойдет ниже, в разделе “Добавление ресурсов в проект”.

Текст, который появляется возле пиктограммы на панели задач Windows 95/98, определяется свойством `Title` объекта приложения. При работе в среде Windows NT этот текст будет выведен под пиктограммой приложения. Изменение данного текста осуществляется путем простого присвоения свойству `Title` нового значения:

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

Другие свойства

Свойство Active, доступное только для чтения, имеет тип Boolean и определяет, обладает ли приложение фокусом ввода.

Свойство ComponentCount определяет количество компонентов, содержащихся в объекте Application. В основном это — формы и экземпляры объектов класса THintWindow (в случае, если свойство Application.ShowHint равно True). Значение свойства ComponentIndex любых компонентов, которые не имеют владельца, всегда равно -1. Следовательно, свойство TApplication.ComponentIndex всегда будет равно -1. Это свойство в основном применяется к формам и их компонентам.

Свойство Components представляет собой массив компонентов, которые принадлежат объекту Application. Количество содержащихся в этом массиве элементов определяется значением свойства TApplication.ComponentCount. Вот пример, показывающий, как можно поместить в список TListBox имена классов всех компонентов, на которые есть ссылки в свойстве ComponentCount:

```
var
  i: integer;
begin
  for i := 0 to Application.ComponentCount - 1 do
    ListBox1.Items.Add(Application.Components[i].ClassName);
end;
```

Свойство HelpFile определяет имя файла справки Windows, который используется данным приложением в качестве интерактивного справочного файла.

Свойство TApplication.Owner всегда равно nil, так как объект TApplication не может принадлежать какому-либо другому компоненту.

Свойство ShowHint разрешает или запрещает вывод подсказок во всем приложении. Поэтому, даже если в определенном компоненте свойство ShowHint будет равно True, присвоение свойству Application.ShowHint значения False запретит вывод подсказок и в этом компоненте (как и во всех других компонентах приложения).

Свойство Terminated равно True, если работа приложения была завершена закрытием главной формы или вызовом метода TApplication.Terminate().

Методы класса TApplication

Класс TApplication имеет ряд методов, которые следует хорошо знать и уметь ими пользоваться. Рассмотрим некоторые из них.

Метод TApplication.CreateForm()

Этот метод определяется так:

```
procedure CreateForm(InstanceClass: TComponentClass; var Reference)
```

Данный метод создает экземпляр формы с типом, определенным параметром InstanceClass, и назначает его переменной Reference. Мы уже встречались с этим методом, когда рассматривали .dpr-файлы. В следующем примере создается экземпляр формы Form1 типа TForm1:

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

Эта строка генерируется Delphi автоматически, если имя Form1 содержится в списке автоматически создаваемых форм. Данный метод можно вызвать в любом месте программы, если необходимо создать экземпляр формы, не входящей в список автоматически создаваемых. Этот вызов будет подобен вызову метода Create() самой формы, но за одним исключением — метод TApplication.CreateForm() всегда проверяет, не равно ли свойство TApplication.MainForm значению nil. Если это так, то свойству Application.MainForm назначается имя вновь созданной формы. Все последующие вызовы метода CreateForm() уже не окажут влияния на значение этого свойства. Как правило, прямой вызов метода CreateForm() не используется — в большинстве случаев для создания формы применяется ее собственный метод Create().

Метод TApplication.HandleException()

С помощью метода HandleException() объект TApplication выводит информацию о возникших при работе приложения исключительных ситуациях. Эта информация выводится в стандартном окне сообщений библиотеки VCL. Можно переопределить стандартную обработку исключений, для чего следует создать собственный обработчик события Application.OnException. О том, как это сделать, рассказывается в разделе "Переопределение методов обработки исключений в приложении", ниже в этой главе.

Методы HelpCommand(), HelpContext() и HelpJump()

Эти методы используются для организации взаимодействия вашего приложения со справочной системой Windows, работа которой обеспечивается приложением WINHELP.EXE. Метод HelpCommand() позволяет вызывать макрокоманды WinHelp и макросы, определенные в файле справки приложения. Метод HelpContext() осуществляет вывод требуемой страницы файла справки, указанного свойством TApplication.HelpFile. Номер страницы определяется значением параметра Context, передаваемого в метод HelpContext(). Метод HelpJump() подобен методу HelpContext(), за одним исключением — в качестве параметра ему передается строка.

Метод TApplication.ProcessMessages()

Этот метод позволяет приложению выбирать из очереди и обрабатывать переданные ему сообщения. Это может оказаться полезным при необходимости обработки сообщений (например, о щелчке на кнопке отмены) внутри жесткого цикла обработки. В противоположность методу TApplication.HandleMessage(), который переводит приложение в состояние бездействующего ожидания, если сообщения в очереди отсутствуют, метод ProcessMessages() в подобном случае не останавливает работу приложения. Пример использования метода ProcessMessages() приведен в главе 10, "Печать в Delphi 5".

Метод TApplication.Run()

Delphi 5 автоматически помещает вызов метода Run() в основной блок файла проекта. Вам никогда не потребуется вызывать этот метод непосредственно, однако необходимо знать, где он вызывается и как работает, на тот случай, если потребуется вносить изменения в файл проекта. В ходе выполнения метод TApplication.Run() сначала определяет процедуру выхода из приложения, обеспечивающую освобождение всей распределенной памяти по завершении его работы. Затем метод входит в цикл, получающий сообщения и вызывающий соответствующие процедуры их обработки вплоть до завершения выполнения приложения.

Метод TApplication.ShowException()

Этот метод получает в качестве параметра класс-исключение и выводит на экран окно с сообщением о возникшей исключительной ситуации. О замене данного метода собственной подпрограммой обработки ошибок речь пойдет в разделе "Переопределение методов обработки исключений в приложении" этой главы.

Другие методы

Метод TApplication.Create() создает экземпляр объекта класса TApplication. Этот метод не должен вызываться непосредственно; Delphi 5 выполняет это самостоятельно при инициализации приложения.

Метод TApplication.Destroy() уничтожает экземпляр объекта класса TApplication. Этот метод также вызывается Delphi 5 автоматически и не должен вызываться программистом.

Метод TApplication.MessageBox() позволяет вывести стандартное окно сообщения Windows (однако он не требует передачи ему в качестве параметра дескриптора окна, в отличие от стандартной функции Windows MessageBox()).

Метод TApplication.Minimize() сворачивает окно приложения, а метод TApplication.Restore() восстанавливает его предыдущее состояние.

Метод TApplication.Terminate() служит для прекращения работы приложения. В отличие от Halt(), этот метод корректно завершает работу приложения посредством вызова функции PostQuitMessage.

На заметку

Для завершения работы приложения используйте метод TApplication.Terminate(). Этот метод вызывает функцию Windows API PostQuitMessage(), которая посылает соответствующее сообщение в очередь сообщений приложения. На это сообщение подпрограммы библиотеки VCL отвечают корректным освобождением всех созданных в приложении объектов и завершением работы приложения. Следует понимать, что приложение не прекращает свою работу немедленно при вызове метода Terminate() — оно продолжает нормальное выполнение до получения сообщения WM_QUIT. Метод Halt(), напротив, прекращает выполнение приложения немедленно, больше не возвращая управления приложению. При этом не происходит корректного освобождения захваченных приложением ресурсов (не считая их освобождения самой операционной системой).

События класса TApplication

Класс TApplication включает несколько событий, для которых можно предоставить собственные подпрограммы обработки. В предыдущих версиях Delphi эти события были не доступны в окне инспектора объектов. Для добавления обработчика события переменной Application требовалось сначала определить его как метод, а затем, уже при выполнении программы, назначить этот метод как обработчик соответствующего события. В Delphi 5 во вкладку Additional палитры компонентов добавлен новый компонент под именем TApplicationEvents. Этот компонент позволяет назначать обработчики событий глобальному объекту Application прямо на этапе проектирования. События класса TApplication описаны в табл. 4.2.

На заметку

Событие TApplication.OnIdle позволяет приложению выполнять некоторую обработку во время ожидания действий пользователя. Одно из применений этого события — обновление состояния меню и кнопок панелей инструментов с учетом текущего состояния приложения.

Таблица 4.2. События класса TApplication

Событие	Описание
`OnActivate`	Происходит, когда приложение становится активным. Событие `OnDeactivate` происходит, когда приложение перестает быть активным — например, при переходе пользователя к другому приложению
`OnException`	Происходит при возникновении необработанной исключительной ситуации. Для их перехвата можно подготовить собственный обработчик. Событие происходит после того, как объект исключения пройдет свой обычный путь по всем перехватчикам исключений в приложении. Как правило, обработка любой исключительной ситуации должна быть выполнена выбираемым по умолчанию обработчиком до того, как исключение попадет на уровень объекта `Application`. Если собственный обработчик исключительной ситуации перехватывает неизвестное ему сообщение, следует регенерировать эту исключительную ситуацию, предварительно убедившись, что создаваемый объект содержит полное описание ошибочной ситуации. Это описание будет выведено пользователю обработчиком исключительных ситуаций, используемым по умолчанию
`OnHelp`	Происходит при вызове справочной системы — например, при нажатии клавиши <F1> или вызове методов `HelpCommand()`, `HelpContext()` либо `HelpJump()`
`OnMessage`	Позволяет перехватить системное сообщение до начала его обработки тем элементом управления, которому оно адресовано. Перехватываются все сообщения для всех управляющих элементов приложения. Будьте внимательны при работе с этим событием, поскольку подпрограмма его обработки легко может превратиться в узкое место, тормозящее работу всего приложения
`OnHint`	Позволяет выводить подсказки при нахождении указателя мыши над тем или иным управляющим элементом. Текст подсказки может быть помещен, например, в строку состояния приложения
`OnIdle`	Происходит при переходе приложения в состояние простоя, связанного с ожиданием очередного события. Это событие не вызывается постоянно — перейдя в состояние простоя, приложение выйдет из него только после получения очередного сообщения

Мы еще вернемся к обсуждению объекта `Application` ниже в этой главе, а также при рассмотрении различных проектов в других главах.

Класс `TScreen`

Этот класс инкапсулирует состояние экрана, на котором выполняется ваше приложение. Класс `TScreen` не является компонентом, добавляемым в формы Delphi 5 во время разработки или динамически создаваемым в процессе выполнения приложения. Delphi 5 автоматически создает один глобальный экземпляр объекта класса `TScreen` под именем `Screen`, который доступен в любом месте приложения. Класс `TScreen` включает несколько перечисленных в табл. 4.3 свойств, которые могут оказаться вам полезными.

Таблица 4.3. Свойства класса TScreen

Свойство	Описание
`ActiveControl`	Свойство с доступом “только для чтения”, определяющее, какой управляющий элемент в данный момент имеет фокус ввода. При перемещении фокуса ввода от одного элемента к другому свойство получает новое значение до окончания события `OnExit` объекта, теряющего фокус

Окончание табл. 4.3

Свойство	Описание
`ActiveForm`	Указывает форму, имеющую в настоящее время фокус ввода. Свойство устанавливается при передаче фокуса ввода другой форме или при получении фокуса ввода от другого приложения
`Cursor`	Форма курсора, глобальная для всего приложения. Значение по умолчанию — `crDefault`. Каждый компонент имеет собственное свойство `Cursor`, которое может модифицироваться при работе программы. Однако когда курсор приобретает вид, отличный от `crDefault`, все остальные компоненты будут отражать это изменение до тех пор, пока свойство `Screen.Cursor` вновь не получит значение `crDefault`. Другими словами, если значение свойства `Screen.Cursor` равно `crDefault`, то о форме курсора следует запросить элемент, в котором он изображается. Если значение свойства не равно `crDefault`, то опрос производить не требуется
`Cursors`	Список всех курсоров, доступных устройству экрана
`DataModules`	Список всех модулей данных приложения
`DataModuleCount`	Количество модулей данных, имеющихся в приложении
`FormCount`	Количество форм, доступных в приложении
`Forms`	Список форм, доступных в приложении
`Fonts`	Список имен шрифтов, доступных устройству экрана
`Height`	Высота экрана в пикселях
`PixelsPerInch`	Относительный масштаб системного шрифта
`Width`	Ширина экрана в пикселях

Архитектура приложений и использование хранилища объектов

Delphi настолько упрощает разработку, что 60 процентов текста создаваемого приложения будет готово еще до того, как вы приступите к работе над ним. Это позволяет уделить больше времени и усилий для работы над архитектурой приложения. Главная проблема разработчиков заключается в том, что они слишком торопятся приступить к кодированию, не уделив должного внимания продумыванию конструкции приложения. Именно это часто бывает основной причиной неудач при разработке проекта.

Выбор архитектуры приложения

Данная книга не посвящена вопросам разработки архитектуры приложений или объектно-ориентированному анализу и конструированию. Однако мы полагаем, что это — один из важнейших аспектов разработки приложений, наряду с общей постановкой задачи, разработкой детального проекта и других важнейших операций, выполняемых до начала написания программ.

Ссылки на дополнительные источники по вопросам разработки и проектирования приложений, которые мы считаем наиболее полезными, приведены в приложении В во втором томе, "Рекомендуемая литература". Настоятельно рекомендуем вам тщательно проработать материал этого раздела, прежде чем приступать к кодированию программ создаваемого приложения.

Приведем лишь несколько вопросов из числа тех, на которые необходимо ответить при обдумывании архитектуры нового приложения.

- Должна ли выбранная архитектура обеспечивать повторное использование кода?
- Выполнена ли должным образом локализация кода и данных по отдельным модулям?
- Допускает ли выбранная архитектура простое внесение изменений?
- Позволяет ли локализация интерфейса пользователя и фоновых процессов их независимую замену?
- Допускает ли выбранная архитектура совместную работу команды программистов над проектом? Другими словами, может ли команда легко и просто работать над отдельными модулями без пересечения действий?

Все это только некоторая часть требований, которые должны учитываться на стадии проектирования.

О разработке архитектуры приложения написаны многие тома, и мы не намерены соревноваться с ними в полноте охвата этой темы. Однако мы надеемся, что нам удастся вызвать у вас достаточный интерес и вы обратитесь к соответствующей литературе. В последующих разделах предлагаются простые методы разработки архитектуры типичного приложения для доступа к базе данных с соответствующим интерфейсом пользователя и обсуждается, как среда Delphi может упростить решение этой задачи.

Архитектура приложений Delphi

Как мы уже отмечали, для того чтобы разрабатывать приложения в среде Delphi, не нужно уметь создавать отдельные компоненты. Это действительно так, однако подобное умение сделает вас более квалифицированным программистом.

Причина в том, что способный к созданию компонентов программист отлично представляет себе объектную модель и принятую архитектуру приложений Delphi. Значит, такой программист в большей степени способен к использованию всей мощи и достоинств объектной модели Delphi в своих приложениях. Возможно, вам уже приходилось слышать, что сама система Delphi была создана в среде Delphi. Поэтому визуальная среда разработки Delphi является прекрасным примером, по которому можно судить о принятой в Delphi архитектуре приложений. Ее можно использовать и в ваших приложениях.

Даже если вам не придется создавать собственные компоненты, время, потраченное на их изучение, не пропадет зря. Попробуйте стать настоящим специалистом во всем — в библиотеке VCL, языке Object Pascal и операционной системе Win32.

Пример архитектуры приложения

Для демонстрации мощи наследования форм и использования хранилища объектов попробуем воспользоваться ими на практике, создав, если не само приложение в целом, то по крайней мере его скелет. Основные цели, которые мы будем преследовать, — это достижение возможности повторного использования кода, обеспечение простоты внесения изменений, гарантия согласованности проекта, а также поддержка групповой разработки.

Рассматриваемая иерархия классов форм состоит из форм, предназначенных для работы с базами данных. Эти формы типичны для большинства приложений, в которых необходим доступ к базам данных. Формы должны обеспечивать выполнение основных операций с данными, таких как редактирование, добавление и просмотр. Они также должны содержать управляющие элементы, необходимые для выполнения этих функций (например, панель инструментов или строку состояния). Состояние объектов управления должно изменяться в соответствии с изменением состояния самой формы. Кроме того, такие формы должны обеспечивать генерацию события при изменении режима работы формы.

Создаваемая иерархия классов должна допускать возможность работы команды программистов над различными частями приложения, которые в этом случае должны быть в достаточной степени обособлены. В противном случае велика вероятность ситуации, когда одни и те же файлы будут модифицироваться несколькими программистами одновременно.

Рассматриваемая иерархия состоит из трех уровней. Подробное их обсуждение будет проведено ниже.

В табл. 4.4 приведен полный список используемых форм и даны сведения о их назначении.

Таблица 4.4. Формы для работы с базами данных

Форма	Назначение
`TChildForm = class(TForm)`	Обеспечивает способность вставки в другое окно в качестве дочернего
`TDBModeForm = class(TchildForm)`	Обеспечивает функции работы с данными (редактирование, добавление, просмотр) и предоставляет событие, генерируемое при изменении состояния формы
`TDBNavStatForm = class(TDBBaseForm)`	Типичная форма ввода данных, содержащая стандартную навигационную панель и строку состояния — компоненты, используемые практически во всех приложениях, работающих с базами данных

Класс `TChildForm`

Класс `TChildForm` представляет собой базовый класс для форм, которые могут выполняться как независимые модальные или немодальные формы, а также быть дочерними окнами в любом другом окне.

Эта способность данного класса упрощает работу команды разработчиков над отдельными частями приложения, позволяя отлаживать каждую форму как самостоятельную часть приложения, не зависящую от других частей. Кроме того, этот класс предоставляет удобную функцию интерфейса пользователя, позволяющую ему запускать каждую форму как отдельный элемент приложения, хотя подобный метод и нельзя считать общепринятым способом работы с формами. В листинге 4.3 приведен исходный код класса `TChildForm`. Текст этой и всех других форм находится в хранилище объектов, размещенном в каталоге `/Code` на компакт-диске, прилагаемом ко второму тому (см. также `www.williamspublishing.com`.).

Листинг 4.3. Исходный текст класса `TChildForm`

```
unit ChildFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
```

```
  Forms, Dialogs, StdCtrls, ExtCtrls, Menus;
type
  TChildForm = class(TForm)
  private
    FAsChild: Boolean;
    FTempParent: TWinControl;
  protected
    procedure CreateParams(var Params: TCreateParams); override;
    procedure Loaded; override;
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create(AOwner: TComponent;
      AParent: TWinControl); reintroduce; overload;

    // Этот метод должен быть перегружен и возвращать
    // либо главное меню формы, либо nil.
    function GeTFormMenu: TMainMenu; virtual; abstract;
    function CanChange: Boolean; virtual;
  end;

implementation

{$R *.DFM}
constructor TChildForm.Create(AOwner: TComponent);
begin
  FAsChild := False;
  inherited Create(AOwner);
end;

constructor TChildForm.Create(AOwner: TComponent; AParent: TWinControl);
begin
  FAsChild := True;
  FTempParent := aParent;
  inherited Create(AOwner);
end;

procedure TChildForm.Loaded;
begin
  inherited;
  if FAsChild then
  begin
    align := alClient;
    BorderStyle := bsNone;
    BorderIcons := [];
    Parent := FTempParent;
    Position := poDefault;
  end;
end;

procedure TChildForm.CreateParams(var Params: TCreateParams);
Begin
```

```
  Inherited CreateParams(Params);
  if FAsChild then
    Params.Style := Params.Style or WS_CHILD;
end;

function TChildForm.CanChange: Boolean;
begin
  Result := True;
end;

end.
```

В приведенном выше листинге вы можете увидеть некоторые специфичные приемы: использование возможностей перегрузки языка Object Pascal и определение формы в качестве дочернего окна.

Использование второго конструктора

Вероятно, вы уже обратили внимание на то, что данная форма имеет два конструктора. Первый из них используется при создании обычной формы. Это конструктор с одним параметром. Второй конструктор имеет два параметра и объявлен как перегружаемый. Он используется при создании формы в качестве дочернего окна. Родительское окно передается в конструктор как параметр `AParent`. Обратите внимание на использование описания `reintroduce`, которое необходимо для того, чтобы избежать выдачи предупреждения о сокрытии виртуального конструктора.

Первый конструктор просто устанавливает значение переменной `FAsChild` равным `False` — это указывает, что форма была создана обычным образом. Второй конструктор устанавливает эту переменную равной `True` и присваивает значение параметра `AParent` переменной `FTempParent`. Позже это значение используется в методе `Loaded()` — как родительский объект дочерней формы.

Форма как дочернее окно

Для того чтобы форма стала дочерним окном, следует приложить определенные усилия. Прежде всего следует корректно установить некоторые свойства формы, что и делается в методе `TChildForm.Loaded()`. Приведенный в листинге 4.3 код обеспечивает форме внешний вид, отличный от диалогового окна, при ее вызове в качестве дочернего окна. Это осуществляется посредством удаления рамки и всех размещенных на ней пиктограмм. Кроме того, выполняется выравнивание формы относительно родительского окна, задаваемого переменной `FTempParent`. Если форма всегда используется в качестве дочернего окна, все эти свойства могут быть установлены еще во время конструирования формы. Однако наша форма может вызываться и как обычная, поэтому описанные свойства устанавливаются только в том случае, если значение переменной `FAsChild` равно `True`.

Кроме того, следует переопределить метод `CreateParams()`, чтобы указывать Windows о необходимости создания дочернего окна. Это осуществляется путем установки стиля `WS_CHILD` в свойстве `Params.Style`.

Возможности применения этой типовой формы не ограничиваются лишь приложениями для работы с базами данных. Фактически, она может использоваться для создания любой формы, которая должна поддерживать функцию представления в виде дочернего окна. Демонстрационный пример использования этой формы как нормальной и как дочернего окна содержится в проекте `ChildTest.DPR`, размещенном в каталоге `\Form Framework` на компакт-диске, прилагаемом ко второму тому (см. также `www.williamspublishing.com`.).

На заметку

В Delphi 5 в состав библиотеки VCL были введены кадры. Они построены таким образом, что допускают внедрение в форму. Поскольку кадры служат контейнерами для прочих компонентов, их функции весьма напоминают поведение дочерних форм, речь о которых шла выше. Подробное обсуждение кадров будет представлено вашему вниманию чуть позднее.

Класс TDBModeForm

Класс TDBModeForm является производным от класса TChildForm. Его назначение — работа с таблицей данных (просмотр, добавление, редактирование). Кроме того, класс TDBModeForm включает событие, вызываемое при изменении режима работы.

Исходный текст класса TDBModeForm приведен в листинге 4.4.

Листинг 4.4. Исходный текст класса TDBModeForm

```
unit DBModeFrm;

interface

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

type

  TFormMode = (fmBrowse, fmInsert, fmEdit);

  TDBModeForm = class(TChildForm)
  private
    FFormMode      : TFormMode;
    FOnSetFormMode : TNotifyEvent;
  protected
    procedure SetFormMode(AValue: TFormMode); virtual;
    function  GetFormMode: TFormMode; virtual;
  public
    property FormMode: TFormMode read GetFormMode write SetFormMode;
  published
    property OnSetFormMode: TNotifyEvent read FOnSetFormMode write
FOnSetFormMode;

  end;

var
  DBModeForm: TDBModeForm;

implementation

{$R *.DFM}

procedure TDBModeForm.SetFormMode(AValue: TFormMode);
begin
```

```
  FFormMode := AValue;
  if Assigned(FOnSetFormMode) then
    FOnSetFormMode(self);
end;

function TDBModeForm.GetFormMode: TFormMode;
begin
  Result := FFormMode;
end;

end.
```

Реализация формы TDBModeForm предельно проста и ясна. Хотя здесь и используются некоторые еще не обсуждавшиеся нами приемы, очень легко проследить, как работает приведенный в листинге код. Сперва определяется перечислимый тип TFormMode, представляющий возможные состояния формы. Затем объявляется свойство FormMode и методы его чтения и записи. (Подробно вопросы создания свойств и методов их чтения/записи будут обсуждаться в главе 21 второго тома, "Создание пользовательских компонентов в Delphi".)

Проект FormModeTest.DPR, демонстрирующий работу формы TDBModeForm, можно найти в каталоге \Form Framework на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Класс TDBNavStatForm

Класс TDBNavStatForm обеспечивает основную функциональность приложения в описываемой нами иерархии. Эта форма содержит общий набор компонентов, используемых в приложениях для работы с базами данных. В частности, она содержит навигационную панель и строку состояния, которые автоматически обновляются при изменении состояния формы. Например, при переводе формы в состояние fsBrowse (просмотр) кнопки Accept и Cancel находятся в пассивном состоянии. Однако при переводе формы в состояние fsInsert (вставка) или fsEdit (редактирование) эти кнопки становятся активными. Текущее состояние формы всегда отражается в ее строке состояния.

В листинге 4.5 приведен исходный текст класса TDBNavStatForm. Обратите внимание: из этого листинга удален список компонентов. Полный текст класса можно увидеть после загрузки демонстрационного проекта этой формы.

Листинг 4.5. Исходный текст класса TDBNavStatForm (в сокращении)

```
unit DBNavStatFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBMODEFRM, ComCtrls, ToolWin, Menus, ExtCtrls, ImgList;

type
  TDBNavStatForm = class(TDBModeForm)
    { Компоненты, не вошедшие в листинг }
    procedure sbAcceptClick(Sender: TObject);
    procedure sbInsertClick(Sender: TObject);
```

```
    procedure sbEditClick(Sender: TObject);
  private
    { Private declarations }
  protected
    procedure Setbuttons; virtual;
    procedure SetStatusBar; virtual;
    procedure SetFormMode(AValue: TFormMode); override;
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create(AOwner: TComponent; AParent: TWinControl); overload;
    procedure SetToolBarParent(AParent: TWinControl);
    procedure SetStatusBarParent(AParent: TWinControl);
  end;
var
  DBNavStatForm: TDBNavStatForm;

implementation

{$R *.DFM}

{ TDBModeForm3 }

procedure TDBNavStatForm.SetFormMode(AValue: TFormMode);
begin
  inherited SetFormMode(AValue);
  SetButtons;
  SetStatusBar;
end;

procedure TDBNavStatForm.Setbuttons;

  procedure SetBrowseButtons;
  begin
    sbAccept.Enabled  := False;
    sbCancel.Enabled  := False;

    sbInsert.Enabled  := True;
    sbDelete.Enabled  := True;
    sbEdit.Enabled    := True;

    sbFind.Enabled    := True;
    sbBrowse.Enabled  := True;

    sbFirst.Enabled   := True ;
    sbPrev.Enabled    := True ;
    sbNext.Enabled    := True ;
    sbLast.Enabled    := True ;
  end;

  procedure SetInsertButtons;
  begin
    sbAccept.Enabled  := True;
```

```
    sbCancel.Enabled  := True;

    sbInsert.Enabled  := False;
    sbDelete.Enabled  := False;
    sbEdit.Enabled    := False;

    sbFind.Enabled    := False;
    sbBrowse.Enabled  := False;

    sbFirst.Enabled   := False;
    sbPrev.Enabled    := False;
    sbNext.Enabled    := False;
    sbLast.Enabled    := False;
  end;

  procedure SetEditButtons;
  begin
    sbAccept.Enabled  := True;
    sbCancel.Enabled  := True;
    sbInsert.Enabled  := False;
    sbDelete.Enabled  := False;
    sbEdit.Enabled    := False;

    sbFind.Enabled    := False;
    sbBrowse.Enabled  := True;

    sbFirst.Enabled   := False;
    sbPrev.Enabled    := False;
    sbNext.Enabled    := False;
    sbLast.Enabled    := False;
  end;

begin
  case FormMode of
    fmBrowse: SetBrowseButtons;
    fmInsert: SetInsertButtons;
    fmEdit:   SetEditButtons;
  end; {  case }

end;

procedure TDBNavStatForm.SetStatusBar;
begin
  case FormMode of
    fmBrowse: stbStatusBar.Panels[1].Text := 'Browsing';
    fmInsert: stbStatusBar.Panels[1].Text := 'Inserting';
    fmEdit:   stbStatusBar.Panels[1].Text := 'Edit';
  end;

  mmiInsert.Enabled := sbInsert.Enabled;
  mmiEdit.Enabled   := sbEdit.Enabled;
  mmiDelete.Enabled := sbDelete.Enabled;
```

```
  mmiCancel.Enabled := sbCancel.Enabled;
  mmiFind.Enabled   := sbFind.Enabled;

  mmiNext.Enabled   := sbNext.Enabled;
  mmiPrevious.Enabled := sbPrev.Enabled;
  mmiFirst.Enabled  := sbFirst.Enabled;
  mmiLast.Enabled   := sbLast.Enabled;

end;

procedure TDBNavStatForm.sbAcceptClick(Sender: TObject);
begin
  inherited;
   FormMode := fmBrowse;
end;

procedure TDBNavStatForm.sbInsertClick(Sender: TObject);
begin
  inherited;
  FormMode := fmInsert;
end;

procedure TDBNavStatForm.sbEditClick(Sender: TObject);
begin
  inherited;
  FormMode := fmEdit;
end;

constructor TDBNavStatForm.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FormMode := fmBrowse;
end;

constructor TDBNavStatForm.Create(AOwner: TComponent; AParent: TWinControl);
begin
  inherited Create(AOwner, AParent);
  FormMode := fmBrowse;
end;

procedure TDBNavStatForm.SetStatusBarParent(AParent: TWinControl);
begin
  stbStatusBar.Parent := AParent;
end;

procedure TDBNavStatForm.SetToolBarParent(AParent: TWinControl);
begin
  tlbNavigationBar.Parent := AParent;
end;

end.
```

Большинство обработчиков событий различных кнопок TToolButtons просто устанавливает форму в соответствующее состояние. В свою очередь, изменение состояния формы с помощью вызова процедур SetButtons() и SetStatusBar() отражается на состояниях кнопок и строки состояния. В процедуре SetButtons() выполняется активизация и деактивизация кнопок панели инструментов в соответствии с новым состоянием формы.

Обратите внимание на две процедуры, предназначенные для изменения родительского окна компонентов формы TToolBar и TStatusBar. Это сделано для того, чтобы в случае вызова формы как дочернего окна, можно было в качестве родительского окна для этих компонентов указать главную форму. Зачем это нужно, станет вам понятно после запуска демонстрационного примера, помещенного в каталог \Form Framework, расположенный на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Как отмечалось ранее, форма TDBNavStatForm унаследовала функциональные возможности, которые позволяют ей выступать как в качестве независимой формы, так и в качестве дочернего окна. В демонстрационной программе форма TDBNavStatForm вызывается как независимая с помощью следующего кода:

```
procedure TMainForm.btnNormalClick(Sender: TObject);
var
  LocalNavStatForm: TNavStatForm;
begin
  LocalNavStatForm := TNavStatForm.Create(Application);
  try
    LocalNavStatForm.ShowModal;
  finally
    LocalNavStatForm.Free;
  end;
end;
```

Для вывода формы в качестве дочернего окна использован следующий код:

```
procedure TMainForm.btnAsChildClick(Sender: TObject);
begin
  if not Assigned(FNavStatForm) then
  begin
    FNavStatForm := TNavStatForm.Create(Application, pnlParent);
    FNavStatForm.SetToolBarParent(self);
    FNavStatForm.SetStatusBarParent(self);
    mmMainMenu.Merge(FNavStatForm.mmFormMenu);
    FNavStatForm.Show;
    pnlParent.Height := pnlParent.Height - 1;
  end;
end;
```

При этом не только форма определяется как дочернее окно по отношению к форме pnlParent, но и ее компоненты TToolBar и TStatusBar становятся дочерними по отношению к главной форме. Также обратите внимание на слияние меню основной и дочерней форм с помощью процедуры TMainForm.mmMainMenu.Merge(). Естественно, при освобождении дочерней формы следует обязательно вызвать процедуру TMainForm.mmMainMenu.UnMerge(), как показано ниже.

```
procedure TMainForm.btnFreeChildClick(Sender: TObject);
begin
  if Assigned(FNavStatForm) then
  begin
    mmMainMenu.UnMerge(FNavStatForm.mmFormMenu);
    FNavStatForm.Free;
    FNavStatForm := nil;
  end;
end;
```

Взгляните на работу демонстрационной программы, имеющейся на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.). На рис. 4.1 показаны оба варианта формы TDBNavStatForm — как внедренное окно и как отдельная форма. (Чтобы форма, представленная как дочернее окно, имела видимые границы, в нее был помещен рисунок — компонент TImage.)

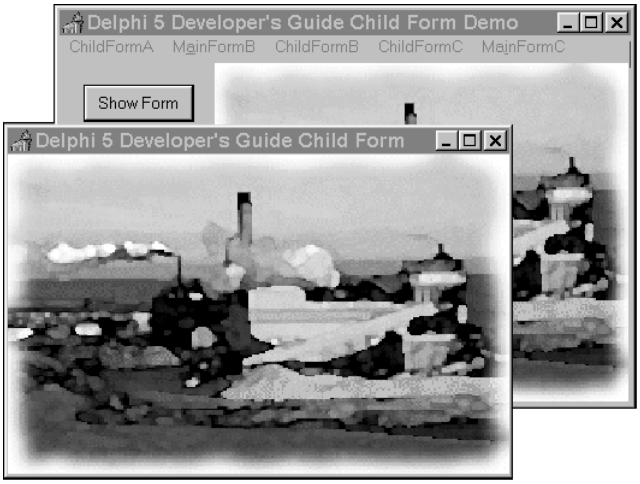


Рис. 4.1. Вид формы TDBNavStatForm в качестве независимой формы и дочернего окна

Позже мы доведем начатую здесь работу до создания полностью функционального приложения, предназначенного для работы с базой данных.

Использование кадров при разработке проектов

В Delphi 5 была добавлена новая возможность работы с кадрами. Кадры (frame) предназначены для создания контейнеров компонентов, внедряемых в ту или иную форму. В целом это напоминает тот механизм, работу которого мы только что продемонстрировали с помощью формы TChildForm. Отличие кадров состоит в том, что они позволяют выполнять подобные действия непосредственно на этапе разработки, а также добавлять полученные объекты в палитру компонентов с целью их повторного использования. В листинге 4.6 показан исходный текст главной формы проекта, по своим возможностям сходного с предыдущим демонстрационным примером. Отличие состоит только в том, что в этом случае в программе используются кадры.

Листинг 4.6. Пример программы, использующей кадры

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    spltrMain: TSplitter;
    pnlParent: TPanel;
    pnlMain: TPanel;
    btnFrame1: TButton;
    btnFrame2: TButton;
    procedure btnFrame1Click(Sender: TObject);
    procedure btnFrame2Click(Sender: TObject);
  private
    { Private declarations }
    FFrame: TFrame;
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses Frame1Fram, Frame2Fram;
{$R *.DFM}

procedure TMainForm.btnFrame1Click(Sender: TObject);
begin
  if FFrame <> nil then
    FFrame.Free;
  FFrame := TFrame1.Create(pnlParent);
  FFrame.Align := alClient;
  FFrame.Parent := pnlParent;
end;

procedure TMainForm.btnFrame2Click(Sender: TObject);
begin
  if FFrame <> nil then
    FFrame.Free;
  FFrame := TFrame2.Create(pnlParent);
  FFrame.Align := alClient;
  FFrame.Parent := pnlParent;
end;

end.
```

В листинге 4.6 показан текст главной формы, содержащей две области, оформленные как две отдельные панели. Правая панель предназначена для размещения кадра. В программе определяются два различных кадра. Закрытое поле FFrame ссылается на объект класса TFrame. Поскольку оба используемых в программе кадра являются производными непосредственно от класса TFrame, переменная FFrame может содержать ссылку на любой из них. Две кнопки в главной форме предназначены для создания различных объектов TFrame и назначения их переменной FFrame. Полученный результат будет аналогичен использованию формы класса TChildForm. Демонстрационный проект FrameDemo.dpr можно найти на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Дополнительные возможности управления проектом

В этом разделе рассматриваются некоторые дополнительные полезные приемы управления проектом, которые вы сможете использовать в ваших приложениях.

Добавление ресурсов в проект

Ранее уже упоминалось о том, что .res-файлы представляют собой подключаемые к проектам ресурсы. Кроме того, вам должно быть уже известно, что именно могут представлять собой ресурсы Windows. Можно добавить к проекту любое количество ресурсов, создавая отдельные .res-файлы с изображения, пиктограммами, курсорами и т.п.

Для создания .res-файлов следует использовать редактор ресурсов. После создания очередного файла он связывается с приложением с помощью директивы $R, помещаемой в .dpr-файл:

```
{$R MYFILE.RES}
```

Эта запись может следовать непосредственно за директивой, связывающей с приложением файл ресурсов с тем же именем, что и у самого приложения:

```
{$R *.RES}
```

Если все сделано правильно, в процессе работы приложения ресурсы могут быть загружены, например, с помощью методов TBitmap.LoadFromResourceName() или TBitmap.LoadFromResourceID(). В листинге 4.7 приведен пример загрузки из .res-файлов таких ресурсов, как изображение, пиктограмма и курсор. Данный проект — Resource.dpr — имеется на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.). Обратите внимание на используемые в этом проекте функции Win32 API LoadIcon() и LoadCursor() (подробнее о них можно узнать в интерактивной справочной системе по функциям Win32 API).

На заметку

Для загрузки растрового изображения из файла ресурса Windows API предоставляет функцию LoadBitmap(). Однако эта функция не возвращает палитры цветов и, следовательно, не может использоваться для загрузки 256-цветных изображений. Вместо нее следует использовать функции TBitmap.LoadFromResourceName() или TBitmap.LoadFromResourceID().

Листинг 4.7. Пример загрузки ресурсов из .res-файла

```
unit MainFrm;
interface
uses
  Windows, Forms, Controls, Classes, StdCtrls, ExtCtrls;

const
  crXHair = 1; //Константа для нового курсора. Это число должно быть
               // либо положительным, либо меньше -20.
type

  TMainForm = class(TForm)
    imgBitmap: TImage;
    btnChemicals: TButton;
    btnClear: TButton;
    btnChangeIcon: TButton;
    btnNewCursor: TButton;
    btnOldCursor: TButton;
    btnOldIcon: TButton;
    btnAthena: TButton;
    procedure btnChemicalsClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
    procedure btnChangeIconClick(Sender: TObject);
    procedure btnNewCursorClick(Sender: TObject);
    procedure btnOldCursorClick(Sender: TObject);
    procedure btnOldIconClick(Sender: TObject);
    procedure btnAthenaClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnChemicalsClick(Sender: TObject);
begin
  { Загрузка изображения. Его имя должно быть указано прописными буквами! }
  imgBitmap.Picture.Bitmap.LoadFromResourceName(hInstance, 'CHEMICAL');
end;
procedure TMainForm.btnClearClick(Sender: TObject);
begin
  imgBitmap.Picture.Assign(nil); // Очистка изображения
end;

procedure TMainForm.btnChangeIconClick(Sender: TObject);
begin
  { Загрузка пиктограммы. Ее имя должно быть указано прописными буквами! }
  Application.Icon.Handle := LoadIcon(hInstance, 'SKYLINE');
end;
```

```
procedure TMainForm.btnNewCursorClick(Sender: TObject);
begin
  { Внесение нового курсора в массив курсоров }
  Screen.Cursors[crXHair] := LoadCursor(hInstance, 'XHAIR');
  Screen.Cursor := crXHair;  // Изменение курсора
end;

procedure TMainForm.btnOldCursorClick(Sender: TObject);
begin
  // Возврат к стандартному курсору
  Screen.Cursor := crDefault;
end;

procedure TMainForm.btnOldIconClick(Sender: TObject);
begin
  { Загрузка пиктограммы. Ее имя должно быть указано прописными буквами! }
  Application.Icon.Handle := LoadIcon(hInstance, 'DELPHI');
end;

procedure TMainForm.btnAthenaClick(Sender: TObject);
begin
  { Загрузка изображения. Его имя должно быть указано прописными буквами! }
  imgBitmap.Picture.Bitmap.LoadFromResourceName(hInstance, 'ATHENA');
end;

end.
```

Изменение курсора

Вероятно, одним из чаще всего используемых свойств класса `Tscreen` является свойство `Cursor`, позволяющее изменять вид курсора, общий для всего приложения. Например, следующий фрагмент кода придает курсору вид песочных часов, показывающих пользователю, что он должен подождать окончания некоторого длительного процесса:

```
Screen.Cursor := crHourGlass
{ Некоторые длительные действия }
Screen.Cursor := crDefault;
```

Идентификатор `crHourGlass` — это предопределенная константа, задающая индекс в массиве `Cursors`. Есть и другие предопределенные константы, например `crBeam` или `crSize`. Все предопределенные константы соответствуют диапазону чисел от 0 до -20 (от `crDefault` до `crHelp`). Весь вписок доступных значений приведен в интерактивной справочной системе Delphi. При необходимости любое из этих значений может быть назначено свойству `Screen.Cursor`.

Кроме того, можно создать собственный курсор и добавить его в массив `Cursors`. Для этого необходимо определить константу, описывающую новый курсор и еще не занятую другим курсором. Предопределенные курсоры задаются значениями от -20 до 0. Отрицательные значения констант зарезервированы Borland, поэтому для собственных курсоров приложений следует использовать только положительные значения. Например:

```
crCrossHair := 1;
```

Для создания самого курсора можно использовать любой редактор ресурсов — например, входящую в состав Delphi 5 утилиту Image Editor. Курсор должен быть сохранен в файле ресурсов (.res-файле). Здесь следует обратить внимание на одну важную деталь: имя созданного вами файла ресурса должно отличаться от имени проекта. Дело в том, что файл ресурсов с именем, соответствующим имени проекта, создается Delphi 5 автоматически при каждой компиляции проекта и может просто перекрыть подготовленный вами файл. Однако собственные .res-файлы должны находиться в том же каталоге, что и исходные файлы проекта, чтобы компилятор мог найти и связать их с создаваемым приложением. Указать Delphi 5 на необходимость присоединения к приложению дополнительного файла ресурсов можно с помощью следующей директивы, помещаемой в .dpr-файл:

```
{$R CrossHairRes.RES}
```

И, наконец, для загрузки курсора и внесения его в список Cursors следует использовать следующие операторы:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Screen.Cursors[crCrossHair] := LoadCursor (hInstance, 'CROSSHAIR');
  Screen.Cursor := crCrossHair;
end;
```

Здесь для загрузки курсора использована функция Win32 API LoadCursor(), которой передаются два параметра — дескриптор модуля, содержащий курсор, и имя курсора в .res-файле. Учтите, что имя курсора должно быть указано прописными буквами!

Переменная hInstance указывает на текущее приложение. Далее возвращаемое функцией LoadCursor() значение назначается элементу массива Cursors с индексом, равным значению crCrossHair. Наконец, значение crCrossHair назначается свойству Screen.Cursor, тем самым обеспечивается требуемое изменение формы курсора.

На компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.), имеется соответствующий пример — приложение CrossHair.dpr. В этом проекте особый курсор загружается и выводится на экран с помощью изложенных выше действий. Сам образ нового курсора сохраняется в файле ресурсов под именем CrossHairRes.res.

Чтобы понять, как создается курсор, с помощью команды Tools⇨Image Editor запустите утилиту Image Editor и откройте в ее окне файл CrossHairRes.res.

Предупреждение создания нескольких экземпляров формы

Если для создания форм используются методы Application.CreateForm() или TForm.Create(), следует убедится, что параметр Reference не указывает на уже существующий экземпляр этой формы. Вот пример такой технологии:

```
begin
  if not Assigned(SomeForm) then begin
    Application.CreateForm(TSomeForm, SomeForm);
    try
      SomeForm.ShowModal;
    finally
```

```
      SomeForm.Free;
      SomeForm := nil;
    end;
  end
  else
    SomeForm.ShowModal;
end;
```

В этом варианте при уничтожении формы, переменной `SomeForm` обязательно следует присвоить значение `nil`. В противном случае метод `Assigned()` будет работать неверно и требуемые цели не будут достигнуты. В случае немодальной формы невозможно определить момент ее закрытия, а потому присвоение значения `nil` ее переменной следует выполнять в обработчике события `OnDestroy`. Этот метод уже рассматривался нами выше, в разделе "Вывод немодальной формы".

Добавление кода в `.dpr`-файл

Можно добавить некоторый код в файл проекта до вывода главной формы приложения. Это может быть выполнение процедур инициализации, вывод заставки, открытие базы данных или любые другие действия, которые необходимо выполнить до вывода главной формы. В случае необходимости можно даже прекратить выполнение приложения, не создавая его главную форму. Так, в листинге 4.8 приведен пример проекта, в котором для получения доступа к приложению пользователь должен ввести пароль. В случае некорректного ввода пароля приложение прекратит работу до создания его главной формы.

Листинг 4.8. Пример выполнения действий по инициализации проекта

```
program Initialize;

uses
  Forms,
  Dialogs,
  Controls,
  MainFrm in 'MainFrm.pas' {MainForm} ;

{$R *.RES}

var
  Password: String;
begin
  if InputQuery('Password', 'Enter your password', PassWord) then
    if Password = 'D5DG' then
    begin
      // Прочие действия по инициализации
      Application.CreateForm(TMainForm, MainForm);
    Application.Run;
    end
    else
      MessageDlg('Incorrect Password, terminating program',
                 mtError, [mbok], 0);
end.
```

Переопределение методов обработки исключений в приложении

По умолчанию при возникновении в приложении исключительной ситуации объект Application автоматически обрабатывает ее и выводит пользователю стандартное окно с соответствующим сообщением.

Однако в больших приложениях обычно создаются собственные классы исключительных ситуаций, которые будут некорректно обрабатываться стандартными средствами Delphi 5. В этом случае необходимо организовать специальную обработку для каждого конкретного типа исключения. Как правило, в подобных ситуациях потребуется переопределить и стандартные методы обработки исключительных ситуаций класса TApplication, заменив их подпрограммами собственной разработки.

Класс TApplication имеет стандартный обработчик события OnException, в который можно внести собственный код. Этот обработчик автоматически вызывается при возникновении исключительной ситуации. В нем можно выполнить любую дополнительную обработку и исключить выдачу стандартного сообщения об ошибке.

Однако свойства объекта TApplication не доступны в окне инспектора объектов. Поэтому для добавления в приложение специализированного обработчика исключений потребуется воспользоваться компонентом TApplicationEvents.

В листинге 4.9 показано, как именно следует переопределять стандартный обработчик исключительных ситуаций приложения.

Листинг 4.9. Текст главной формы приложения с переопределенным обработчиком исключений

```
unit MainFrm;

interface

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

type

  ENotSoBadError = class(Exception);
  EBadError      = class(Exception);
  ERealBadError  = class(Exception);

  TMainForm = class(TForm)
    btnNotSoBad: TButton;
    btnBad: TButton;
    btnRealBad: TButton;
    appevnMain: TApplicationEvents;
    procedure btnNotSoBadClick(Sender: TObject);
    procedure btnBadClick(Sender: TObject);
    procedure btnRealBadClick(Sender: TObject);
    procedure appevnMainException(Sender: TObject; E: Exception);
  public
  end;
```

```
var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnNotSoBadClick(Sender: TObject);
begin
  raise ENotSoBadError.Create('This isn''t so bad!');
end;

procedure TMainForm.btnBadClick(Sender: TObject);
begin
  raise EBadError.Create('This is bad!');
end;

procedure TMainForm.btnRealBadClick(Sender: TObject);
begin
  raise ERealBadError.Create('This is real bad!');
end;

procedure TMainForm.appevnMainException(Sender: TObject; E: Exception);
var
  rslt: Boolean;
begin
  if E is EBadError then
  begin
  { Вывод собственного окна с сообщением об ошибке и
    предложением завершить работу приложения. }
    rslt := MessageDlg(Format('%s %s %s %s %s', ['An', E.ClassName,
        'exception has occurred.', E.Message, 'Quit App?']),
         mtError, [mbYes, mbNo], 0) = mrYes;
    if rslt then
       Application.Terminate;
  end
  else if E is ERealBadError then
  begin // Вывод собственного сообщения
        // и завершение работы приложения.
    MessageDlg(Format('%s %s %s %s %s', ['An', E.ClassName,
        'exception has occured.', E.Message, 'Quitting Application']),
         mtError, [mbOK], 0);
    Application.Terminate;
  end
  else // Выполнение стандартной обработки исключительных ситуаций
    Application.ShowException(E);
end;

end.
```

В листинге 4.9 метод appevrMainException() является подпрограммой обработки события OnException компонента TApplicationEvent. В этом обработчике события для проверки типа возникшей исключительной ситуации используется информация о типе времени выполнения (RTTI). На основании полученных сведений о типе объекта исключения в подпрограмме предпринимаются те или иные действия. Работа этой подпрограммы объясняется в комментариях. Проект, в котором используется этот обработчик событий, носит имя OnException.dpr и располагается на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Если во вкладке Language Exceptions диалогового окна Debugger Options (вывести его можно с помощью команды Tools⇨Debugger Options) установлен флажок опции Stop on Delphi Exceptions, то отладчик IDE Delphi 5 будет выводить сведения о возникших исключительных ситуациях в своем собственном окне, не давая приложению возможности выполнить их обработку своими силами. Хотя этот режим может оказаться весьма полезным в режиме отладки, установка упомянутого флажка опции может помешать анализу работы собственных подпрограмм обработки исключений вашего проекта. Для возобновления нормальной работы приложения этот флажок необходимо сбросить.

Вывод заставки

Предположим, что до начала работы приложения требуется вывести некоторую заставку. Подобная форма может быть выведена на экран сразу же после запуска приложения и оставаться на нем до завершения процесса подготовки приложения к работе. Выполнить это несложно. Ниже приведены основные этапы создания подобной формы-заставки.

1. После создания главной формы приложения создайте другую форму, представляющую собой заставку. Назовите ее SplashForm.
2. С помощью команды Project⇨Options убедитесь, что форма SplashForm отсутствует в списке автоматически создаваемых форм.
3. Присвойте свойству BorderStyle формы-заставки значение bsNone, а свойству BorderIcons — значение [ ].
4. Поместите в форму SplashForm компонент TImage и установите значение его свойства Align равным alClient.
5. Загрузите в компонент TImage изображение, присвоив соответствующее значение его свойству Picture.

Теперь, когда у вас имеется форма-заставка, осталось только вывести ее на экран, для чего достаточно слегка изменить .dpr-файл. Пример такого изменения показан в листинге 4.10, содержащем текст файла проекта приложения Splash.dpr, присутствующего на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Листинг 4.10. Текст .dpr-файла, предусматривающего вывод заставки

```
program splash;

uses
  Forms,
  MainFrm in 'MainFrm.pas' { MainForm} ,
  SplashFrm in 'SplashFrm.pas' { SplashForm} ;
```

```
{$R *.RES}
begin
  Application.Initialize;
  { Создание формы-заставки }
  SplashForm := TSplashForm.Create(Application);
  SplashForm.Show;   // Вывод заставки
  SplashForm.Update; // Обеспечение ее появления на экране

  { Следующий цикл просто создает видимость некоторой
     длительной работы по инициализации приложения }
  while SplashForm.tmMainTimer.Enabled do
    Application.ProcessMessages;

  Application.CreateForm(TMainForm, MainForm);
  SplashForm.Hide;  // Удаление заставки
  SplashForm.Free;  // Освобождение памяти
  Application.Run;
end.
```

Обратите внимание на следующие две строки листинга:

```
while SplashForm.tmMainTimer.Enabled do
  Application.ProcessMessages;
```

Это просто имитация некоторого длительного процесса, чтобы заставка не исчезла с экрана, едва на нем появившись. Компонент `TTimer` был помещен в форму `SplashForm`, и его свойство `Interval` установлено равным `3000`. По истечении трех секунд происходит событие `OnTimer`, в обработчике которого содержится следующая строка:

```
tmMainTimer.Enabled := False;
```

В результате этого действия выполнение приведенного выше цикла прекращается.

Ограничение изменений размера окна формы

Для иллюстрации того, каким образом можно управлять изменением размера формы, подготовлен проект, в котором основная форма синего цвета содержит панель, на которой размещены остальные компоненты. При любых изменениях размера формы панель всегда остается размещенной по центру формы. Кроме того, пользователь не может уменьшить размер формы так, чтобы она стала меньше содержащейся в ней панели. Исходный текст этой формы приведен в листинге 4.11.

Листинг 4.11. Исходный текст формы приложения `TempDemo`

```
unit BlueBackFrm;

interface

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

```
type
  TBlueBackForm = class(TForm)
    pnlMain: TPanel;
    bbtnOK: TBitBtn;
    bbtnCancel: TBitBtn;
    procedure FormResize(Sender: TObject);
  private
    Procedure CenterPanel;
    { Обработчик сообщения WM_WINDOWPOSCHANGING }
    procedure WMWindowPosChanging(var Msg: TWMWindowPosChanging);
    message WM_WINDOWPOSCHANGING;
  end;

var
  BlueBackForm: TBlueBackForm;

implementation
uses Math;
{$R *.DFM}

procedure TBlueBackForm.CenterPanel;
{  Эта процедура центрирует главную панель в форме}
begin
  { Горизонтальное центрирование }
  if pnlMain.Width < ClientWidth then
    pnlMain.Left := (ClientWidth - pnlMain.Width) div 2
  else
    pnlMain.Left := 0;

  { Вертикальное центрирование }
  if pnlMain.Height < ClientHeight then
    pnlMain.Top := (ClientHeight - pnlMain.Height) div 2
  else
    pnlMain.Top := 0;
end;

procedure TBlueBackForm.WMWindowPosChanging(var Msg: TWMWindowPosChanging);
var
  CaptionHeight: integer;
begin
  { Вычисление высоты заголовка }
  CaptionHeight := GetSystemMetrics(SM_CYCAPTION);
  { Эта процедура не учитывает высоту и ширину рамки формы.
    Для учета этих величин можно воспользоваться методом GetSystemMetrics() }

  // Защита окна от уменьшения до размеров, меньших размеров панели
  Msg.WindowPos^.cx := Max(Msg.WindowPos^.cx, pnlMain.Width+20);
  Msg.WindowPos^.cy := Max(Msg.WindowPos^.cy, pnlMain.Height+20+CaptionHeight);

  inherited;
```

```
end;
procedure TBlueBackForm.FormResize(Sender: TObject);
begin
  CenterPanel; // Центрирование панели при изменении размера формы
end;

end.
```

В этом листинге демонстрируется перехват сообщений окна, в частности сообщения `WM_WINDOWPOSCHANGING`, которое генерируется при выполнении пользователем попытки изменить размеры окна. Его обработка позволяет предотвратить изменение размеров формы меньше допустимых. Дополнительную информацию о сообщениях Windows можно найти в главе 5, "Сообщения Windows". Данный демонстрационный проект носит название `TempDemo.dpr`. Он присутствует на компакт-диске, прилагаемом ко второму тому (см. также `www.williamspublishing.com`.).

Проекты, не имеющие форм

Формы представляют собой центральные звенья, вокруг которых строится большинство приложений Delphi 5. Однако, в принципе, ничто не запрещает создать приложение, не использующее форм. Файл проекта `.dpr` является обычным файлом с программным текстом, в котором используются дополнительные модули с формами и другими объектами. В этом программном файле, безусловно, может выполняться любая обработка, не требующая использования форм. Для того чтобы увидеть, как будет выглядеть подобное приложение, просто создайте новое приложение Delphi и удалите главную форму из проекта с помощью команды File⇨Remove From Project. Теперь `.dpr`-файл этого приложения будет выглядеть так:

```
program Project1;
uses
 Forms;
{$R *.RES}
begin
  Application.Initialize;
  Application.Run;
end.
```

Более того, из этого файла можно удалить даже оператор `uses` и вызов методов `Application.Initialize` и `Application.Run`:

```
program Project1;
begin
end.
```

Полученная программа совершенно бесполезна, однако не следует забывать, что в блок `begin..end` можно поместить любое количество каких угодно операторов. По сути, этот блок представляет собой точку входа консольного приложения Win32.

Выход из Windows

Выход из Windows, осуществляемый непосредственно из приложения, может понадобиться, например, в ситуации, когда это приложение вносит в систему изменения, которые вступят в силу только после перезагрузки Windows. Можно, конечно, предложить

пользователю самому перезагрузить Windows, но лучше спросить у пользователя, желает ли он произвести перезагрузку системы немедленно, после чего выполнить эту работу без его участия. В любом случае, постарайтесь в создаваемых приложениях избегать перезагрузки Windows без крайней необходимости.

Для выхода из Windows можно воспользоваться одной из двух функций Windows API — `ExitWindows()` или `ExitWindowsEx()`.

Функция `ExitWindows()` унаследована от 16-битовой Windows. В предыдущих версиях Windows можно было передавать ей различные параметры и выполнять перезагрузку системы по завершении ее работы. В среде Win32 она просто прекращает текущий сеанс работы пользователя и позволяет войти в систему другому пользователю.

В Win32 функция `ExitWindows()` была заменена новой функцией `ExitWindowsEx()`. Она позволяет закончить сеанс работы, завершить работу Windows и перезагрузить операционную систему. В листинге 4.12 показан пример использования обеих этих функций.

Листинг 4.12. Использование функций `ExitWindows()` и `ExitWindowsEx()`

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    btnExit: TButton;
    rgExitOptions: TRadioGroup;
    procedure btnExitClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnExitClick(Sender: TObject);
begin
  case rgExitOptions.ItemIndex of
    0: Win32Check(ExitWindows(0, 0)); // Выход из системы
                                      // и вход под другим именем
    1: Win32Check(ExitWindowsEx(EWX_REBOOT, 0));  // Перезагрузка системы
    2: Win32Check(ExitWindowsEx(EWX_SHUTDOWN, 0));  // Выход и выключение питания
    3: Win32Check(ExitWindowsEx(EWX_LOGOFF, 0));
               // Выход из системы и вход под другим именем
  end;
end;

end.
```

В листинге 4.12 для определения варианта выхода из Windows используется значение переключателя rgExitOptions. Первая опция использует функцию ExitWindows(), которая прекращает сеанс работы текущего пользователя и позволяет войти в систему другому пользователю.

Оставшиеся опции работают с функцией ExitWindowsEx(). Вторая опция перезагружает Windows, а третья позволяет пользователю выключить компьютер. Четвертая опция выполняет те же действия, что и первая, однако использует функцию ExitWindowsEx().

Функции ExitWindows() и ExitWindowsEx() возвращают значение True при успешном завершении и False при неудаче. В последнем случае можно воспользоваться функцией Win32Check() из модуля SysUtils.pas, которая вызывает функцию Win32 API GetLastError() и выводит более подробную информацию о возникшей ошибке.

На заметку

При работе в среде Windows NT функция ExitWindowsEx() не сможет остановить работу системы, поскольку это действие требует специальных привилегий. Предварительно потребуется вызвать функцию Win32 API AdjustTokenPrivleges(), чтобы с ее помощью получить привилегию SE_SHUTDOWN_NAME. Подробнее о получении необходимых привилегий можно узнать из интерактивной справочной системы Win32.

Приведенный пример текста входит в состав проекта ExitWin.dpr, который присутствует на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Защита от выхода из Windows

А теперь поговорим о том, как можно защититься от перезагрузки — ведь в тот момент, когда некоторое приложение попытается перезагрузить компьютер, ваше приложение может работать с открытыми файлами и располагать несохраненной информацией. Если вашему приложению не удастся каким-либо образом перехватить запрос на перезагрузку системы, могут оказаться безвозвратно потерянными ценные данные. Однако перехватить запрос на перезагрузку несложно. Для этого требуется лишь обеспечить обработку события OnCloseQuery главной формы приложения. Текст подпрограммы обработки этого события может выглядеть следующим образом:

```
procedure TMainForm.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  if MessageDlg('Shutdown?', mtConfirmation, mbYesNoCancel, 0) = mrYes then
    CanClose := True
  else
    CanClose := False;
end;
```

Присвоение переменной CanClose значения False защитит приложение от перезагрузки Windows. В другом варианте можно присвоить переменной CanClose значение True только после вывода пользователю предложения сохранить открытые файлы. Демонстрационный пример по этой теме носит название NoClose.dpr и присутствует на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

На заметку

В случае упоминавшегося ранее проекта, не использующего формы, приложение должно перехватывать системное сообщение WM_QUERYENDSESSION. Оно посылается всем работающим приложениям при выполнении каким-либо из них процедур ExitWindows() или ExitWindowsEX(). Возвратив нулевое значение в ответ на это сообщение, можно указать Windows, что ваше приложение не завершило свою работу, и поэтому перезагрузка невозможна. Подробнее об обработке сообщений Windows речь пойдет в главе 5, “Сообщения Windows”.

Резюме

В этой главе рассматривались вопросы управления проектами и выбора архитектуры приложений. Было проведено обсуждение ключевых компонентов, входящих в состав большинства проектов Delphi 5: классов `TForm`, `TApplication` и `TScreen`. Мы показали, что разработку приложений следует начинать с выбора требуемой архитектуры проекта. Кроме того, вашему вниманию было предложено множество полезных технических приемов, которые можно использовать в создаваемых приложениях.

Глава

5

Сообщения Windows

Хотя компоненты библиотеки VCL отображают множество сообщений Win32 в события языка Object Pascal, понимание системы сообщений Windows необходимо любому программисту, имеющему дело с Win32.

Потребности программиста на Delphi практически полностью удовлетворяются возможностями работы с событиями, предоставляемыми VCL. Однако при создании серьезных нестандартных приложений и особенно при разработке компонентов Delphi вам, безусловно, потребуется непосредственно обрабатывать сообщения Windows, после чего генерировать события, соответствующие этим сообщениям.

Что такое сообщение

Сообщение (*message*) представляет собой извещение о некотором имевшем место событии, посылаемое системой Windows в адрес приложения. Любые действия пользователя — щелчок мышью, изменение размеров окна приложения, нажатие клавиши на клавиатуре — вынуждают Windows отправить приложению сообщение, извещающее о том, что произошло в системе.

Сообщение представляет собой определенную запись, передаваемую приложению системой Windows. Эта запись содержит сведения о том, что именно произошло, а также дополнительную информацию, зависящую от типа события. Например, в случае щелчка мышью запись дополнительно содержит координаты указателя мыши в момент щелчка. Тип данных, используемый Delphi для представления сообщений Windows, называется `TMsg`. Он представляет собой запись и объявляется в модуле Windows следующим образом:

```
type
  TMsg = packed record
    hwnd: HWND;     // Дескриптор окна-получателя
    message: UINT;  // Идентификатор сообщения
    WParam: WPARAM; // 32 бита дополнительной информации
    LParam: LPARAM; // 32 бита дополнительной информации
    time: DWORD;    // Время создания сообщения
    pt: TPoint;     // Положение указателя мыши в момент создания сообщения
  end;
```

Что содержится в сообщении?

Если приведенных в описании записи комментариев вам показались недостаточно, то ниже приводится более подробное описание назначения полей сообщения Windows.

`hwnd`	32-битовый дескриптор окна, которому предназначено сообщение. Окно может быть практически любым экранным объектом, так как Win32 поддерживает дескрипторы окон для большинства визуальных объектов (окон, диалоговых окон, кнопок, полей ввода и т.п.).
`message`	Константа, представляющая тип сообщения. Она может быть определена системой Windows и описана в модуле `Windows`, либо определена программистом как определяемое пользователем сообщение.
`Wparam`	Это поле часто содержит некоторую константу, значение которой определяется типом сообщения. Кроме того, оно может содержать идентификатор окна или элемента управления, связанного с данным сообщением.
`Lparam`	Это поле чаще всего хранит индекс или указатель на некоторые данные в памяти. Так как поля `WParam`, `LParam` и `Pointer` имеют один и тот же размер, равный 32 битам, между ними допускается взаимное преобразование типов.

Итак, мы познакомились с тем, что представляют собой сообщения Windows в целом, в последующих разделах будут подробно рассмотрены различные типы этих сообщений.

Типы сообщений

В Win32 API каждому сообщению Windows ставится в соответствие определенное значение, которое заносится в поле `message` записи `TMsg`. В Delphi все эти константы определяются в модуле `messages`, а большая их часть дополнительно описана в интерактивной справочной системе. Обратите внимание на то, что имя каждой константы, представляющей тип сообщения, начинается с символов *WM* (т.е. ***W****indows* ***M****essage*). В табл. 5.1 представлены некоторые наиболее распространенные сообщения Windows и их числовые коды.

Таблица 5.1. Основные типы сообщений Windows

Идентификатор сообщения	Значение	Сообщает окну о том, что…
`WM_ACTIVATE`	`$0006`	Это окно активизируется или деактивизируется
`WM_CHAR`	`$0102`	Для некоторой клавиши были посланы сообщения `WM_KEYDOWN` или `WM_KEYUP`
`WM_CLOSE`	`$0010`	Данное окно должно быть закрыто
`WM_KEYDOWN`	`$0100`	На клавиатуре была нажата клавиша
`WM_KEYUP`	`$0101`	Клавиша на клавиатуре была отпущена
`WM_LBUTTONDOWN`	`$0201`	Пользователь нажал левую кнопку мыши
`WM_MOUSEMOVE`	`$0200`	Указатель мыши переместился
`WM_PAINT`	`$000F`	Необходимо перерисовать клиентскую область данного окна
`WM_TIMER`	`$0113`	Произошло событие таймера
`WM_QUIT`	`$0012`	Программа должна быть завершена

Принципы работы системы сообщений Windows

Система сообщений Windows состоит из трех компонентов.

- *Очередь сообщений* (Message queue). Windows содержит отдельную очередь сообщений для каждого приложения. Приложение Windows должно получать сообщения из этой очереди и передавать их соответствующему окну.
- *Цикл сообщений* (Message loop). Группа циклически выполняемых операторов приложения, осуществляющая выборку сообщения из очереди и передачу его соответствующему окну для обработки. Затем из очереди выбирается следующее сообщение, которое также передается по назначению, и так — до окончания работы приложения.
- *Процедура окна* (Window procedure). Каждое окно приложения имеет собственную процедуру, которая получает все передаваемые этому окну сообщения. В ответ на полученное сообщение процедура должна выполнить определенные действия. Эта процедура является процедурой обратного вызова и обычно возвращает Windows некоторое значение по завершении обработки сообщения.

На заметку *Функция обратного вызова* (callback function) представляет собой функцию в вашей программе, вызываемую Windows или некоторым другим внешним модулем.

Путешествие сообщения из пункта А (где произошло породившее его событие) в пункт Б (окно в вашем приложении, отвечающее на это сообщение) состоит из пяти этапов.

1. В системе происходит некоторое событие.
2. Windows превращает это событие в соответствующее сообщение и помещает его в очередь сообщений вашего приложения.
3. Приложение получает сообщение из очереди и помещает его в запись типа `TMsg`.
4. Приложение передает сообщение процедуре соответствующего окна вашего приложения.
5. Процедура окна выполняет некоторые действия в ответ на сообщение.

Этапы 3 и 4 выполняются циклом сообщений, который фактически представляет собой "сердце" программы Windows. Именно этот цикл обеспечивает взаимодействие программы с внешним миром, получая сообщения из очереди и передавая их соответствующему окну приложения. Если очередь сообщения данного приложения будет исчерпана, система Windows обратится к другим приложениям и позволит им выполнить обработку сообщений в их очередях. Все описанные выше действия показаны на рис. 5.1.

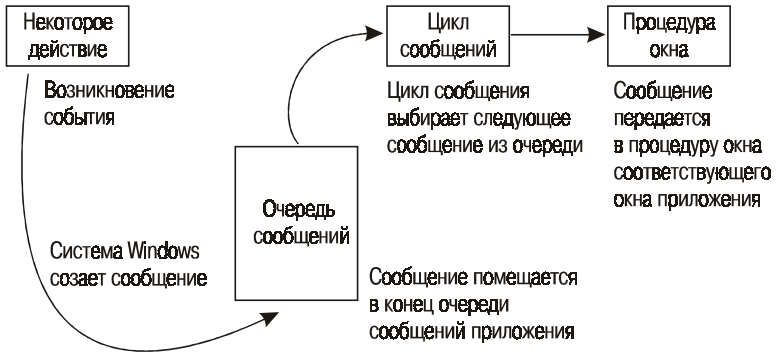


Рис. 5.1. Схема обработки сообщений Windows

Система обработки сообщений в Delphi

Подпрограммы библиотеки VCL выполняют существенную часть обработки сообщений Windows в приложении. В частности, цикл сообщений встроен в модуль `Forms` библиотеки VCL, поэтому прикладному программисту не придется беспокоиться о выборке сообщений из очереди и передаче их соответствующим процедурам окон. Кроме того, Delphi помещает информацию из записи типа `TMsg` в собственную запись типа `TMessage`, описание которой приведено ниже.

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

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

Обратите внимание, что в записи `TMessage` содержится меньше информации, чем в исходной записи `TMsg`. Причина в том, что Delphi берет часть обработки сообщений Windows на себя, и в запись `TMessage` помещается только та часть информации, которая необходимая для дальнейшей обработки.

Важно отметить, что в записи `TMessage` содержится поле `Result`. Как уже упоминалось ранее, некоторые сообщения требуют возврата значения из процедуры обработки окна после завершения их обработки. В Delphi эта задача решается совсем просто — достаточно поместить возвращаемое значение в поле `Result` записи `TMessage`. Подробнее этот процесс мы обсудим в разделе "Возврат результата обработки сообщения", ниже в этой главе.

Специализированные записи

В дополнение к общей записи типа `TMessage` в Delphi определен набор специализированных записей для всех типов сообщений Windows. Назначение этих записей состоит в предоставлении программисту всей содержащейся в исходном сообщении Windows информации, но без необходимости дешифровки значений полей `wParam` и `lParam`. Описания всех этих записей находятся в модуле `Messages`. Ниже приведен пример такой специализированной записи, используемой для большинства типов сообщений Windows о событиях мыши.

```
type
  TWMMouse = record
    Msg: Cardinal;
    Keys: Longint;
    case Integer of
      0: (
        XPos: Smallint;
        YPos: Smallint);
      1: (
        Pos: TSmallPoint;
        Result: Longint);
  end;
```

Все типы записей для конкретных сообщений о событиях мыши (например, `WM_LBUTTONDOWN` или `WM_RBUTTONUP`) просто определяются равными записи типа `TWMMouse`, как показано ниже.

```
TWMRButtonUp = TWMMouse;
TWMLButtonDown = TWMMouse;
```

На заметку

Специализированные записи сообщений определены практически для всех стандартных сообщений Windows. Соглашение о присвоении имен требует, чтобы имя записи было равно имени сообщения с префиксом *T* и без символа подчеркивания. Например, запись для сообщения `WM_SETFONT` должна иметь имя `TWMSetFont`.
Следует отметить, что запись типа `TMessage` создается для всех типов сообщений Windows и в любых ситуациях. Однако работать с этими записями менее удобно, чем со специализированными записями сообщений.

Обработка сообщений

Обработка сообщений означает, что приложение тем или иным образом реагирует на получаемые от операционной системы сообщения. В стандартном приложении Windows обработка сообщений сосредоточивается в процедурах окна. Однако Delphi, частично обрабатывая сообщения, упрощает работу программиста, позволяя вместо одной процедуры для обработки всех типов сообщений создавать независимые процедуры для обработки сообщений различных типов. Любая процедура обработки сообщений должна отвечать трем требованиям.

- Процедура должна быть методом объекта.
- Процедуре должен передаваться один передаваемый по ссылке (с помощью описания `var`) параметр с типом `TMessage` или с типом любого другого специализированного сообщения.
- Описание процедуры должно включать ключевое слово `message` за которым должна следовать константа, задающая тип обрабатываемого сообщения.

Вот пример объявления процедуры, обрабатывающей сообщение `WM_PAINT`:

```
procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
```

На заметку

Соглашение по присвоению имен требует присваивать обработчику сообщений то же имя, что и имя обрабатываемого им сообщения, но без символа подчеркивания и с указанием первых знаков имени прописными буквами.

В качестве примера напишем простую процедуру обработки сообщения `WM_PAINT`, которая вместо перерисовки сообщения будет подавать звуковой сигнал.

Создайте новый пустой проект. Добавьте в окне редактора заголовок для функции `WMPaint`, поместив его в раздел `private` объекта `TForm1`:

```
procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
```

Теперь в раздел `implementation` модуля добавьте определение процедуры. В этом случае указание ключевого слова `message` не требуется. Не забудьте определить область видимости новой процедуры, указав с помощью оператора “точка”, что она является методом класса `TForm1`.

```
procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
  Beep;
  inherited;
end;
```

Обратите внимание на использование в процедуре ключевого слова inherited, которое позволяет передать сообщение обработчику этого сообщения, принадлежащему классу-предку. В нашем случае по ключевому слову inherited сообщение передается обработчику сообщения WM_PAINT класса TForm.

На заметку

В отличие от вызова обычных унаследованных методов, в данном случае имя метода-предка не указывается. Дело в том, что при обработке сообщений имя метода не имеет значения, так как Delphi отыскивает процедуру обработки сообщения по значению, задаваемому в интерфейсе класса ключевым словом message.

В листинге 5.1 приведен пример простой формы, в которой будет обрабатываться сообщение WM_PAINT. Подготовить данный проект очень просто — достаточно создать новый проект и добавить текст процедуры WMPaint в описание объекта TForm.

Листинг 5.1. Пример обработки сообщения

```
unit GMMain;

interface

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

type
  TForm1 = class(TForm)
  private
    procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
  end;

var
  Form1: TForm1;

implementation

{ $R *.DFM}

procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
  MessageBeep(0);
  inherited;
end;

end.
```

Всякий раз при поступлении в приложение сообщения WM_PAINT будет вызываться описанная в листинге процедура WMPaint. Она будет информировать вас об этом звуковым сигналом, издаваемым с помощью процедуры MessageBeep(), после чего передаст управление обработчику данного сообщения класса-предка.

Процедура MessageBeep() — отладчик для бедных

Раз уж мы упомянули процедуру MessageBeep(), позволим себе небольшое отступление от основной темы. Эта процедура является одним из самых простых и полезных элементов Win32 API. Она очень проста в использовании — достаточно лишь передать ей некоторую заранее определенную константу, и система Windows пошлет звуковой сигнал на динамик компьютера (или, при наличии звуковой платы, воспроизведет соответствующий WAV-файл). "Ну и что?" — спросите вы. Хотя это и не очевидно с первого взгляда, но процедура MessageBeep() очень удобна для отладочных целей.

Если требуется быстро, без привлечения отладчика и задания точек останова, уточнить, достигает ли программа в ходе своего выполнения некоторого места в ее тексте, поместите туда вызов процедуры MessageBeep(). Эта процедура не требует передачи ей в качестве параметра дескриптора окна или некоторого другого ресурса Windows, поэтому ее можно вызвать практически из любого места программы. Если в компьютере имеется звуковая плата, можно передавать этой процедуре несколько заранее определенных значений параметра (они описаны в интерактивной справочной системе Win32 API) и определять по воспроизводимой мелодии, что же именно произошло.

Если вы не хотите каждый раз писать такое длинное название процедуры, используйте процедуру Beep() из модуля SysUtils, которая представляет собой вызов процедуры MessageBeep() с параметром 0.

Обработка сообщений — условие обязательное

В отличие от событий Delphi, обработка сообщений является обязательной и не может выполняться или не выполняться по усмотрению программиста. Обычно, если в программе объявлено о проведении собственной обработки сообщений, система Windows ожидает выполнения некоторых предопределенных действий, связанных с обработкой сообщения. Чаще всего компоненты VCL выполняют бо́льшую часть обработки своими встроенными средствами, для доступа к которым программисту достаточно лишь вызвать обработчик класса-предка с помощью директивы inherited. Общая схема выглядит примерно так: собственно в обработчике сообщения должны выполняться действия, которые необходимы для приложения, а для выполнения тех действий, которые ожидаются системой Windows, следует вызвать метод inherited.

На заметку

Однако простого вызова унаследованного обработчика сообщения в некоторых случаях может оказаться недостаточно. При обработке сообщений Windows на подпрограмму их обработки в определенных случаях могут быть наложены дополнительные ограничения. Например, при обработке сообщения WM_KILLFOCUS передача фокуса ввода другому управляющему элементу вызовет аварийное прекращение работы приложения.

Для демонстрации важности вызова унаследованного обработчика попробуйте убрать вызов метода inherited из рассмотренного ранее примера. Подпрограмма обработки при этом будет выглядеть так:

```
procedure Tform1.WMPaint(var Msg: TWMPaint);
begin
  MessageBeep(0);
end;
```

Поскольку системе Windows никогда не предоставляется возможности выполнить основную обработку сообщения WM_PAINT, форма этого приложения никогда не будет перерисовываться.

Впрочем, в некоторых случаях вызов метода inherited действительно может быть нежелателен. Например, можно использовать этот подход при обработке сообщения WM_SYSCOMMAND с целью защиты окна от минимизации или распахивания.

Возврат результата обработки сообщения

При обработке определенных сообщений Windows ожидает возврата некоторого результирующего значения. Классическим примером может служить сообщение WM_CTLCOLOR. От подпрограммы обработки этого сообщения Windows ожидает получить дескриптор кисти, с помощью которой должно быть прорисовано диалоговое окно или элемент управления. (Для своих компонентов Delphi предоставляет свойство Color, обеспечивающее выполнение упомянутых действий, так что пример носит лишь иллюстративный характер). Можно и самостоятельно вернуть требуемый дескриптор кисти из процедуры обработки сообщений, назначив соответствующее значение полю Result записи Tmessage (или записи сообщения другого типа) после вызова метода inherited. Например, при обработке сообщения WM_CTLCOLOR требуемый тип кисти можно указать Windows с помощью следующих операторов:

```
procedure Tform1.WMCtlColor(var Msg: TWMCtlColor);
var
  BrushHand: hBrush;
begin
  inherited;
  { Создание дескриптора кисти и назначение его переменной BrushHand }
  Msg.Result := BrushHand;
end;
```

Событие OnMessage класса Tapplication

Другой метод обработки сообщений заключается в использовании события OnMessage класса Tapplication. При назначении процедуры этому событию она будет вызываться всякий раз при получении сообщения из очереди и подготовке его к обработке. Обработчик события OnMessage всегда вызывается до того, как система Windows получит возможность обработать сообщение. Подпрограмма Tapplication.OnMessage должна иметь тип TMessageEvent и объявляться со списком параметров, показанным ниже.

```
Procedure SomeObject.AppMessageHandler(var Msg: TMsg;
  var Handled: Boolean);
```

Все параметры сообщения передаются обработчику события OnMessage в параметре Msg. (Этот параметр имеет тип TMsg записи сообщения Windows и был описан ранее в этой главе). Параметр Handled имеет тип Boolean и используется для указания того, обработано сообщение или нет.

Первым шагом в создании обработчика события OnMessage является создание метода, принимающего те же параметры, что и процедура TMessageEvent. Ниже приведен пример метода, предназначенного для подсчета количества полученных приложением сообщений.

```
Var
  NumMessages: Integer;

procedure Form1.AppMessageHandler(var Msg: TMsg; var Handled: Boolean);
begin
  Inc(NumMessages);
  Handled := False;
end;
```

Второй (и последний) шаг состоит в создании обработчика события посредством назначения подготовленной процедуры в качестве обработчика события Application.OnMessage где-либо в тексте программы. В частности, это может быть сделано непосредственно в .dpr-файле проекта, после создания главной формы приложения, но перед вызовом метода Application.Run:

```
Application.OnMessage := Form1.AppMessageHandler;
```

Основное ограничение события OnMessage состоит в том, что оно перехватывает только сообщения, выбранные из очереди, и не работает с сообщениями, переданными непосредственно процедурам окна различных окон приложения. В главе 13, "Дополнительный инструментарий разработчика", вы узнаете о том, что обойти эти ограничения можно, заменив стандартные процедуры окна собственными подпрограммами.

Событие OnMessage перехватывает все сообщения, направляемые в адрес всех идентификаторов окон, относящихся к данному приложению. Обработчик этого события будет самой загруженной подпрограммой приложения, поскольку этих событий очень много — до тысяч в секунду. Поэтому избегайте выполнения в этой процедуре любых продолжительных действий, иначе работа всего приложения может существенно замедлиться. В частности, это одно из самых неподходящих мест для размещения точек останова в процессе отладки.

Использование собственных типов сообщений

Аналогично тому, как система Windows посылает свои сообщения различным окнам приложения, в самом приложении также может появиться необходимость обмена сообщениями между его окнами и управляющими элементами. Delphi предоставляет разработчику несколько способов осуществления обмена сообщениями в пределах приложения: метод Perform() (работающий независимо от Windows API) и функции Win32 API SendMessage() и PostMessage().

Метод Perform()

Этим методом обладают все потомки класса Tcontrol, входящие в состав библиотеки VCL. Метод Perform() позволяет послать сообщение любой форме или управляющему элементу, заданному именем экземпляра требуемого объекта. Методу Perform() передается три параметра — само сообщение и соответствующие ему параметры Lparam и Wparam. Определение этого метода имеет следующий вид:

```
function Tcontrol.Perform(Msg: Cardinal; Wparam, Lparam: Longint):
  Longint;
```

Для посылки сообщения форме или управляющему элементу используется следующий синтаксис:

```
RetVal := ControlName.Perform(MessageID, Wparam, Lparam);
```

При вызове Perform() управление вызывающей программе не будет возвращено до тех пор, пока сообщение не будет обработано. В методе переданные ему параметры собираются в запись типа Tmessage, после чего вызывается метод Dispatch() указанного объекта, минуя систему обработки сообщений Windows API. Подробнее о методе Dispatch() поговорим немного позже.

Функции API SendMessage() и PostMessage()

В некоторых случаях может потребоваться послать сообщение окну, для которого нет соответствующего экземпляра объекта Delphi. Например, иногда необходимо послать сообщение окну, созданному не Delphi, и вся информация, которая доступна программе, — это его дескриптор. Для подобных целей используются две функции Windows API — SendMessage() и PostMessage(), — которые практически идентичны, но имеют одно существенное различие. Функция SendMessage() подобна методу Perform() — она посылает сообщение непосредственно процедуре окна и ожидает окончания его обработки. В отличие от нее, функция PostMessage() просто помещает сообщение в очередь сообщений Windows и немедленно возвращает управление вызвавшей ее программе, не дожидаясь результатов обработки сообщения.

Функции определены следующим образом:

```
function SendMessage(hWnd: HWND; Msg: UINT; Wparam: WPARAM;
  Lparam: LPARAM): LRESULT; stdcall;
function PostMessage(hWnd: HWND; Msg: UINT; Wparam: WPARAM;
  Lparam: LPARAM): BOOL; stdcall;
```

где

- hWnd — дескриптор окна-получателя сообщения;
- Msg — идентификатор сообщения;
- Wparam — 32 бита дополнительной уточняющей информации;
- Lparam — еще 32 бита дополнительной уточняющей информации.

На заметку

Хотя функции SendMessage() и PostMessage() очень похожи, они возвращают различные по своему смыслу значения. Функция SendMessage() возвращает значение, полученное в результате обработки сообщения, тогда как функция PostMessage() возвращает значение типа Boolean, указывающее, удалось ли поместить сообщение в очередь сообщений соответствующего окна.

Нестандартные сообщения

До сих пор мы обсуждали только стандартные сообщения Windows, идентификаторы которых имеют вид WM_*XXX*. Однако есть еще две обширные категории сообщений, которые следует рассмотреть подробнее, — извещающие сообщения и пользовательские сообщения.

Извещающие сообщения

Извещающие сообщения (notification messages) представляют собой сообщения, посылаемые родительскому окну в том случае, когда в одном из его дочерних элементов управления происходит нечто, заслуживающее внимания родителя. Эти сообщения генерируются только стандартными элементами управления Windows (кнопки, списки, раскрывающиеся списки, поля редактирования) и общими элементами управления Windows (дерево объектов, список объектов и т.п.). Щелчок или двойной щелчок на управляющем элементе, выбор текста в поле редактирования или перемещение ползунка полосы прокрутки — вот примеры событий, вызывающих генерацию извещающих сообщений.

Обработка извещающих сообщений осуществляется с помощью соответствующих процедур обработки, принадлежащих той форме, в которой содержится некоторый управляющий элемент. В табл. 5.2 приведен список извещающих сообщений для стандартных управляющих элементов Win32.

Таблица 5.2. Извещающие сообщения стандартных элементов управления Windows

Извещение	Смысл сообщения
Извещения от кнопок	
BN_CLICKED	Пользователь щелкнул на кнопке
BN_DISABLE	Кнопка переведена в неактивное состояние
BN_DOUBLECLICKED	Пользователь дважды щелкнул на кнопке
BN_HILITE	Пользователь выделил кнопку
BN_PAINT	Кнопка должна быть перерисована
BN_UNHILITE	Выделение кнопки должно быть отменено
Извещения раскрывающихся списков	
CBN_CLOSEUP	Раскрытый список был закрыт пользователем
CBN_DBLCLK	Пользователь дважды щелкнул на строке
CBN_DROPDOWN	Список был сдвинут вниз
CBN_EDITCHANGE	Пользователь изменил текст в поле ввода
CBN_EDITUPDATE	Требуется вывести измененный текст
CBN_ERRSPACE	Элементу управления списка не хватает памяти
CBN_KILLFOCUS	Список потерял фокус ввода
CBN_SELCHANGE	Выбран новый элемент списка
CBN_SELENDCANCEL	Пользователь отказался от сделанного им выбора
CBN_SELENDOK	Выбор пользователя корректен
CBN_SETFOCUS	Список получил фокус ввода
Извещения полей ввода	
EN_CHANGE	Требуется обновление после внесения изменений
EN_ERRSPACE	Элементу управления не хватает памяти
EN_HSCROLL	Пользователь щелкнул на горизонтальной полосе прокрутки
EN_KILLFOCUS	Поле ввода потеряло фокус ввода
EN_MAXTEXT	Введенный текст был обрезан
EN_SETFOCUS	Поле ввода получило фокус ввода
EN_UPDATE	Требуется отображение введенного текста
EN_VSCROLL	Пользователь щелкнул на вертикальной полосе прокрутки
Извещения списков	
LBN_DBLCLK	Пользователь дважды щелкнул на строке
LBN_ERRSPACE	Элементу управления не хватает памяти

Окончание табл. 5.2

Извещение	Смысл сообщения
Извещения списков	
LBN_KILLFOCUS	Список потерял фокус ввода
LBN_SELCANCEL	Отмена выделения
LBN_SELCHANGE	Изменение выделения
LBN_SETFOCUS	Список получил фокус ввода

Внутренние сообщения компонентов VCL

Компоненты библиотеки VCL используют обширный набор собственных внутренних и извещающих сообщений. Хотя в создаваемых вами приложениях вряд ли потребуется работать с этими сообщениями непосредственно, тем не менее разработчику компонентов познакомиться с ними будет весьма полезно. Имена этих сообщений всегда начинаются с префикса `CM_` (*сообщения компонентов* — component messages) или `CN_` (*извещения компонентов* — component notification). Они используются компонентами библиотеки VCL для управления состоянием их внутренних свойств — например, для передачи фокуса, установки цвета, изменения состояния видимости, выдачи требования перерисовки окна, поддержки операций перетаскивания и т.д. Полный список этих сообщений можно найти в разделе интерактивной справочной системы Delphi, посвященном созданию пользовательских компонентов.

Пользовательские сообщения

При разработке приложений может возникнуть ситуация, когда приложению потребуется послать сообщение либо самому себе, либо другому приложению из числа созданных вами. У многих предыдущее утверждение вызовет недоумение — зачем приложению посылать сообщение самому себе, если можно просто вызвать соответствующую процедуру? Это хороший вопрос, на который можно дать сразу несколько ответов. Во-первых, использование сообщений представляет собой механизм поддержки реального полиморфизма, поскольку не требует наличия каких-либо знаний о типе объекта — получателя сообщения. Таким образом, технология работы с сообщениями имеет ту же мощь, что и механизм виртуальных методов, но обладает существенно большей гибкостью. Во-вторых, сообщения допускают необязательную обработку — если объект-получатель не обработает поступившее сообщение, то ничего страшного не произойдет. И, в-третьих, сообщения позволяют широковещательное обращение к нескольким получателям и организацию параллельного прослушивания, что весьма трудно реализовать с помощью механизма вызова процедур.

Использование сообщений внутри приложения

Заставить приложение послать сообщение самому себе очень просто — достаточно воспользоваться функцией Win32 API `SendMessage()` или `PostMessage()` либо вызвать метод `Perform()`. Сообщение должно иметь идентификатор в диапазоне от `WM_USER+100` до `$7FFFF` (этот диапазон Windows резервирует для сообщений пользователя). Например:

```
const
  SX_MYMESSAGE = WM_USER + 100;

begin
```

```
  SomeForm.Perform(SX_MYMESSAGE, 0, 0);
  { или }
  SendMessage(SomeForm.Handle, SX_MYMESSAGE, 0, 0);
  { или }
  PostMessage(SomeForm.Handle, SX_MYMESSAGE, 0, 0);
  .
  .
  .
end;
```

Для получения подобного сообщения необходимо создать процедуру-обработчик в той форме, которой это сообщение предназначается:

```
TForm1 = class(TForm)
  .
  .
  .
private
  procedure SXMyMessage(var Msg: TMessage); message SX_MYMESSAGE;
end;

procedure TForm1.SXMyMessage(var Msg: TMessage);
begin
  MessageDlg('She turned me into a newt!', mtInformation, [mbOk], 0);
end;
```

Как видите, различия в обработке собственного сообщения и стандартного сообщения Windows невелики. Они заключаются в использовании идентификаторов в диапазоне от `WM_USER+100` и выше, а также в присвоении каждому сообщению имени, которое каким-то образом будет отражать его смысл.

Никогда не посылайте сообщения из диапазона `WM_USER – $7FFF`, если не уверены, что получатель способен его корректно обработать. Поскольку каждое окно может независимо выбирать используемые им значения, весьма вероятно появление трудноуловимых ошибок, если только заранее не составить таблицы идентификаторов сообщений, с которыми будут работать все отправители и получатели сообщений.

Обмен сообщениями между приложениями

При необходимости организовать обмен сообщениями между двумя или более приложениями следует использовать в них функцию Win32 API `RegisterWindowMessage()`. Этот метод гарантирует, что для заданного типа сообщений каждое приложение будет использовать один и тот же номер.

Функция `RegisterWindowMessage()` принимает в качестве параметра строку с завершающим нулем и возвращает идентификатор для нового сообщения в диапазоне `$C000 – $7FFF`. Это означает, что вызова этой функции с одной и той же строкой в качестве параметра в каждом приложении будет достаточно, чтобы гарантировать одинаковые номера сообщений во всех приложениях, принимающих участие в обмене. Еще одно преимущество этой функции состоит в том, что система гарантирует уникальность идентификатора, назначенного любой заданной строке. Это позволяет посылать широковещательные сообщения всем существующим в системе окнам, не опасаясь нежелательных побочных эффектов. Недостатком данного метода является некоторое усложнение обработки подобных сообщений. Суть в том, что идентификатор сообщения становится известным только при работе приложения, поэтому использование стандартных процедур обработки сообще-

ний оказывается невозможным. Для работы с подобными сообщениями потребуется переопределить стандартный метод WndProc() или DefaultHandler() соответствующих компонентов либо создать подкласс существующей процедуры окна. Пример использования данной технологии можно найти в главе 13, "Дополнительный инструментарий разработчика".

На заметку Возвращаемый функцией RegisterWindowMessage() идентификатор выделяется динамически и может принимать различные значения в разных сессиях Windows, а значит, не может быть определен до момента выполнения программы.

Широковещательные сообщения

Класс, производный от класса TWinControl, может с помощью метода Broadcast() послать широковещательное сообщение каждому управляющему элементу, владельцем которого он является. Эта технология используется в тех случаях, когда требуется послать одно и то же сообщение группе компонентов. Например, чтобы послать пользовательское сообщение с именем um_Foo всем управляющим элементам, принадлежащим объекту Panel1, можно воспользоваться следующим кодом:

```
var
  M: TMessage;
begin
  with M do
  begin
    Message := UM_FOO;
    WParam := 0;
    LParam := 0;
    Result := 0;
  end;
  Panel1.Broadcast(M);
end;
```

Анатомия системы сообщений библиотеки VCL

Система сообщений библиотеки VCL — это нечто большее, чем просто обработка сообщений с помощью директивы message. После того как Windows отправит сообщение, оно пройдет несколько последовательных этапов обработки, прежде чем достигнет процедуры обработки сообщений вашего приложения. (Кроме того, оно может пройти еще несколько этапов обработки и после выполнения этой процедуры.) Подпрограммы библиотеки VCL обеспечивают доступ к сообщению на протяжении всего этого долгого пути.

Любое сообщение Windows, посланное без требования получения результатов обработки, первоначально обрабатывается методом Application.ProcessMessage(), в котором реализован главный цикл обработки сообщений средствами библиотеки VCL. Следующий этап состоит в передаче сообщения обработчику события Application.OnMessage. Событие OnMessage генерируется при выборке сообщения из очереди приложения в методе ProcessMessage(). Поскольку сообщения, посылаемые без требования получения результатов обработки, не помещаются в очередь, событие OnMessage для них не генерируется.

Для обработки сообщений, посылаемых с требованием предоставления результатов их обработки, неявно вызывается функция Win32 API `DispatchMessage()`, которая, в свою очередь, передает каждое сообщение функции `StdWndProc()`. Для отправляемых сообщений эта функция вызывается непосредственно системой Win32. Функция `StdWndProc()` представляет собой написанную на ассемблере функцию низкого уровня, которая принимает сообщение от Windows и переправляет его назначенному объекту.

Метод объекта, который получает сообщение, называется `MainWndProc()`. Начиная с этого момента в программе можно выполнять любую специальную обработку сообщения, какая только может потребоваться. Обычно в обработку сообщения на этом этапе вмешиваются только для того, чтобы не допустить стандартной обработки сообщения средствами библиотеки VCL.

По завершении работы метода `MainWndProc()` сообщение передается методу `WndProc()` объекта, а затем поступает в распоряжение механизма диспетчеризации сообщений. Этот механизм, функционирующий в рамках метода `Dispatch()`, переправляет сообщение дальше, некоторой конкретной процедуре обработки сообщений, определенной программистом или уже имеющейся в составе средств библиотеки VCL.

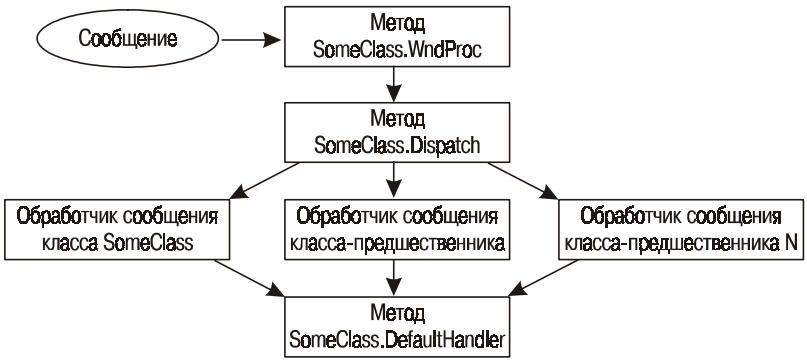


Рис. 5.2. Схема механизма обработки сообщений средствами библиотеки VCL

Наконец, сообщение достигает определенной в программе специализированной процедуры обработки сообщений данного типа. После выполнения всех действий, предусмотренных пользовательской процедурой обработки и всеми унаследованными процедурами, вызываемыми с помощью ключевого слова `inherited`, сообщение передается методу `DefaultHandler()`. Этот метод осуществляет любые завершающие действия по обработке сообщения и передает его функции Win32 API `DefWindowProc()` (или другой стандартной процедуре обработки, например `DefMDIProc()`) для стандартной обработки средствами Windows. Общая схема механизма обработки сообщений средствами библиотеки VCL представлена на рис. 5.2.

На заметку

При обработке сообщений всегда следует вызывать метод `inherited`, за исключением тех случаев, когда имеется абсолютная уверенность в том, что стандартная обработка сообщения не требуется.

Совет

Поскольку все необработанные сообщения доставляются обработчику `DefaultHandler()`, этот метод представляет собой самое подходящее место для выполнения обработки сообщений из других приложений, идентификаторы которых назначены с помощью процедуры `RegisterWindowMessage()`.

Для лучшего понимания схемы обработки сообщений средствами библиотеки VCL рассмотрим небольшую программу, в которой определенная обработка сообщений выполняется в методах `Application.OnMessage` и `WndProc()`, в процедуре обработки сообщений и в методе `DefaultHandler()`. Этот проект носит имя `CatchIt`. Общий вид его главной формы показан на рис. 5.3.

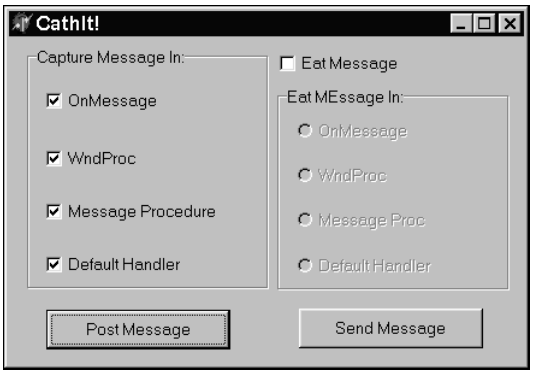


Рис. 5.3. Главная форма приложения `CatchIt`

Ниже приведен текст обработчиков события `OnClick` для объектов кнопок `PostMessButton` и `SendMessButton`. В первом из них для отправки пользовательского сообщения в адрес формы приложения используется функция `PostMessage()`, а во втором для этой же цели применяется функция `SendMessage()`. Для того чтобы в приложении можно было отличить сообщения, посланные с помощью функции `PostMessage()`, от сообщений, посланных с помощью функции `SendMessage()`, нужно в первом случае в поле `WParam` сообщения поместить значение `1`, а во втором — `0`.

```
procedure TMainForm.PostMessButtonClick(Sender: TObject);
{ Отправка форме сообщения без требования предоставления результатов обработки }
begin
  PostMessage(Handle, SX_MYMESSAGE, 1, 0);
end;

procedure TMainForm.SendMessButtonClick(Sender: TObject);
{ Отправка форме сообщения с требованием предоставления результатов обработки }
begin
  SendMessage(Handle, SX_MYMESSAGE, 0, 0);
end;
```

Это приложение предоставляет пользователю возможность “проглотить” сообщение в обработчике события `OnMessage`, методе `WndProc()`, методе специализированной обработки сообщения или в методе `DefaultHandler()`. В любом случае не будет выполнена унаследованная обработка сообщения (запускаемая с помощью ключевого слова `inherited`) и, следовательно, цикл прохождения сообщением всей цепочки механизма обработки сообщений средствами библиотеки VCL будет незавершенным. В листинге 5.2 приведен полный текст главного модуля приложения, иллюстрирующего процессы обработки сообщений в приложениях Delphi.

Листинг 5.2. Исходный текст модуля `CIMain.Pas`

```
unit CIMain;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Menus;

const
  SX_MYMESSAGE = WM_USER; // Идентификатор пользовательского сообщения
  MessString = '%s message now in %s.'; // Оповещение пользователя о сообщении

type
  TMainForm = class(TForm)
    GroupBox1: TGroupBox;
    PostMessButton: TButton;
    WndProcCB: TCheckBox;
    MessProcCB: TCheckBox;
    DefHandCB: TCheckBox;
    SendMessButton: TButton;
    AppMsgCB: TCheckBox;
    EatMsgCB: TCheckBox;
    EatMsgGB: TGroupBox;
    OnMsgRB: TRadioButton;
    WndProcRB: TRadioButton;
    MsgProcRB: TRadioButton;
    DefHandlerRB: TRadioButton;
    procedure PostMessButtonClick(Sender: TObject);
    procedure SendMessButtonClick(Sender: TObject);
    procedure EatMsgCBClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure AppMsgCBClick(Sender: TObject);
  private
    { Обработка сообщений на уровне объекта Application }
    procedure OnAppMessage(var Msg: TMsg; var Handled: Boolean);
    { Обработка сообщений на уровне процедуры WndProc }
    procedure WndProc(var Msg: TMessage); override;
    { Обработка сообщений после диспетчеризации }
    procedure SXMyMessage(var Msg: TMessage); message SX_MYMESSAGE;
    { Обработка сообщений по умолчанию }
    procedure DefaultHandler(var Msg); override;
  end;

var
  MainForm: TMainForm;

implementation

{ $R *.DFM}
```

```
const
  // Строки для указания способа посылки сообщения
 SendPostStrings: array[0..1] of String = ('Sent', 'Posted');

procedure TMainForm.FormCreate(Sender: TObject);
{ Обработчик события OnCreate главной формы }
begin
  // Определение метода OnAppMessage как обработчика события OnMessage
  Application.OnMessage := OnAppMessage;
  // Использование свойства Tag объекта флажка опции для размещения
  // ссылки на связанную с ним кнопку переключателя
  AppMsgCB.Tag := Longint(OnMsgRB);
  WndProcCB.Tag := Longint(WndProcRB);
  MessProcCB.Tag := Longint(MsgProcRB);
  DefHandCB.Tag := Longint(DefHandlerRB);
  // Использование свойства Tag объекта кнопки переключателя для
  // размещения ссылки на связанный с ней объект флажка опции
  OnMsgRB.Tag := Longint(AppMsgCB);
  WndProcRB.Tag := Longint(WndProcCB);
  MsgProcRB.Tag := Longint(MessProcCB);
  DefHandlerRB.Tag := Longint(DefHandCB);
end;

procedure TMainForm.OnAppMessage(var Msg: TMsg; var Handled: Boolean);
{ Обработчик события OnMessage объекта Application }
begin
  // Проверка, является ли данное сообщение собственным сообщением формы
  if Msg.Message = SX_MYMESSAGE then
  begin
    if AppMsgCB.Checked then
    begin
      // Извещение пользователя о поступлении сообщения и
      // установка соответствующего флажка Handled
      ShowMessage(Format(MessString, [SendPostStrings[Msg.WParam],
        'Application.OnMessage']));
      Handled := OnMsgRB.Checked;
    end;
  end;
end;

procedure TMainForm.WndProc(var Msg: TMessage);
{ Процедура WndProc объекта формы}
var
  CallInherited: Boolean;
begin
  // Предполагаем, что унаследованные методы будут вызываться
  CallInherited := True;
  // Проверка получения собственного сообщения формы
  if Msg.Msg = SX_MYMESSAGE then
  begin
    if WndProcCB.Checked then  // Если флажок WndProcCB установлен...
```

```
    begin
      // Извещение пользователя о поступлении сообщения
      ShowMessage(Format(MessString, [SendPostStrings[Msg.WParam],
        'WndProc']));
      { Вызов унаследованных методов, в случае,
         если сообщение не "проглатывается" }
      CallInherited := not WndProcRB.Checked;
    end;
  end;
  if CallInherited then inherited WndProc(Msg);
end;

procedure TMainForm.SXMyMessage(var Msg: TMessage);
{ Процедура обработки пользовательского сообщения }
var
  // Предполагаем, что унаследованные методы будут вызываться
  CallInherited: Boolean;
begin
  CallInherited := True;
  if MessProcCB.Checked then  // Если флажок MessProcCB установлен...
  begin
    // Извещение пользователя о поступлении сообщения
    ShowMessage(Format(MessString, [SendPostStrings[Msg.WParam],
      'Message Procedure']));
    { Вызов унаследованных методов, в случае,
      если сообщение не "проглатывается" }
    CallInherited := not MsgProcRB.Checked;
  end;
  if CallInherited then Inherited;
end;

procedure TMainForm.DefaultHandler(var Msg);
{ Обработка сообщения по умолчанию }
var
  CallInherited: Boolean;
begin
  // Предполагаем, что унаследованные методы будут вызываться
  CallInherited := True;
  // Проверка поступления собственного сообщения приложения
  if TMessage(Msg).Msg = SX_MYMESSAGE then
  begin
    if DefHandCB.Checked then  // Если флажок DefHandCB установлен...
    begin
      // Извещение пользователя о поступлении сообщения
      ShowMessage(Format(MessString, [SendPostStrings[TMessage(Msg).WParam],
        'DefaultHandler']));
      { Вызов унаследованных методов, в случае,
        если сообщение не "проглатывается" }
      CallInherited := not DefHandlerRB.Checked;
    end;
  end;
```

```
  if CallInherited then inherited DefaultHandler(Msg);
end;

procedure TMainForm.PostMessButtonClick(Sender: TObject);
{ Отправление сообщения по методу post }
begin
  PostMessage(Handle, SX_MYMESSAGE, 1, 0);
end;

procedure TMainForm.SendMessButtonClick(Sender: TObject);
{ Отправление сообщения по методу send }
begin
  SendMessage(Handle, SX_MYMESSAGE, 0, 0);
end;

procedure TMainForm.AppMsgCBClick(Sender: TObject);
{ Активизация/деактивизация кнопок переключателей формы по установке флажка}
begin
  if EatMsgCB.Checked then
  begin
    with TRadioButton((Sender as TCheckBox).Tag) do
    begin
      Enabled := TCheckbox(Sender).Checked;
      if not Enabled then Checked := False;
    end;
  end;
end;

procedure TMainForm.EatMsgCBClick(Sender: TObject);
{ Активизация/деактивизация переключателей формы соответственно флажкам }
var
  i: Integer;
  DoEnable, EatEnabled: Boolean;
begin
  // Определение состояния соответствующего флажка
  EatEnabled := EatMsgCB.Checked;
  // Итерация по дочерним элементам управления объекта GroupBox
  // для активизации кнопок и установки состояния флажков
for i := 0 to EatMsgGB.ControlCount - 1 do
    with EatMsgGB.Controls[i] as TRadioButton do
    begin
      DoEnable := EatEnabled;
      if DoEnable then DoEnable := TCheckbox(Tag).Checked;
      if not DoEnable then Checked := False;
      Enabled := DoEnable;
    end;
end;

end.
```

Хотя в процедуре обработки сообщения для передачи сообщения на обработку в унаследованные методы достаточно указать ключевое слово `inherited`, этот метод не пригоден в случае процедур `WndProc()` и `DefaultHandler()`. В этих случаях необходимо явно указать имя вызываемой унаследованной процедуры, например: `inherited WndProc(Msg);`.

Обратите внимание: процедура `DefaultHandler()` несколько отличается от других — ей требуется передавать один *нетипизированный* `var`-параметр. Суть в том, что в процедуре `DefaultHandler()` предполагается, что первое слово в значении параметра будет идентификатором сообщения, тогда как остальная часть сообщения процедурой игнорируется. В результате можно легко извлечь из сообщения его идентификатор с помощью явного приведения параметра к типу `TMessage`.

Связь между сообщениями и событиями

Теперь вы уже знаете достаточно, чтобы понять, что система событий Delphi представляет собой интерфейс для взаимодействия с сообщениями Windows, по крайней мере — с некоторой их частью. Многие события компонентов библиотеки VCL непосредственно связаны с сообщениями Windows типа `WM_`*`XXX`*. В табл. 5.3 приведены основные события компонентов библиотеки VCL и связанные с ними сообщения Windows.

Таблица 5.3. Соответствие событий компонентов библиотеки VCL сообщениям Windows

Событие VCL	Сообщение Windows	Событие VCL	Сообщение Windows
`OnActivate`	`WM_ACTIVATE`	`OnKeyPress`	`WM_CHAR`
`OnClick`	`WM_XBUTTONDOWN`	`OnKeyUp`	`WM_KEYUP`
`OnCreate`	`WM_CREATE`	`OnPaint`	`WM_PAINT`
`OnDblClick`	`WM_XBUTTONDBLCLICK`	`OnResize`	`WM_SIZE`
`OnKeyDown`	`WM_KEYDOWN`	`OnTimer`	`WM_TIMER`

Табл. 5.3 можно рассматривать как краткий справочник, полезный при поиске событий, непосредственно соответствующих тем или иным сообщениям Windows.

Не стоит писать обработчик сообщения Windows, если для него уже имеется предопределенное событие того же назначения. Поскольку обработка событий не является обязательной, это существенно упрощает их использование по сравнению с прямой обработкой сообщений Windows.

Резюме

В этой главе мы говорили о работе системы сообщений Win32 и инкапсуляции обработки этих сообщений средствами библиотеки VCL. Хотя система событий в Delphi — достаточно мощная, знание механизмов обработки сообщений в Windows является важным требованием к любому программисту, работающему в среде Win32.

Дополнительную информацию о работе системы сообщений Windows можно найти в главе 21 второго тома, “Создание пользовательских компонентов в Delphi”. Здесь речь пойдет о методах практического применения тех знаний, которые были приобретены при чтении данной главы.

Глава

6

Стандарты программирования, принятые в этой книге

В этой главе описаны стандарты программирования в Delphi, принятые в данной книге, которая (в соответствии с ее названием) представляет собой руководство для разработчика в Delphi 5. В целом здесь представлены практически все неписаные правила, используемые фирмой Borland International. Эта глава включена в данную книгу с целью демонстрации метода, с помощью которого разработчики, совместно работающие над проектами, смогут придерживаться единого стиля программирования. Благодаря использованию единого стиля написанные разными программистами программы станут для всех более понятными и удобочитаемыми.

Эту главу ни в коем случае нельзя рассматривать как исчерпывающий источник информации о стандартах программирования. Однако предлагаемых здесь сведений вполне достаточно, чтобы приступить к работе. Пользуясь предложенными стандартами, чувствуйте себя свободно и изменяйте их в соответствии со своими потребностями. Однако мы не рекомендуем слишком сильно отклоняться от стандартов, применяемых разработчиками фирмы Borland. Суть в том, что, если в вашу команду придет новый программист, вероятнее всего, он будет знать именно стандарты фирмы Borland. Большинство документов, содержащих описание стандартов программирования, с течением времени эволюционирует — и эта глава не является исключением. Поэтому, если вас интересует более новая версия данного документа, обращайтесь по адресу: `www.xapware.com/ddg`.

В этой главе не описаны стандарты, связанные с интерфейсом пользователя. Это отдельная тема, которая по важности не уступает данной. Ей посвящено множество книг, не говоря о документации фирмы Microsoft, и потому мы решили не дублировать содержащуюся в них информацию. Каждый может обратиться непосредственно к Microsoft Developers Network или к другим источникам, которые могут оказаться полезными в этом отношении.

Общие правила форматирования исходного кода

Отступы

Для организации отступов используйте по два пробела для каждого уровня. Не следует прибегать к символам табуляции, поскольку пользователи могут по-разному установить параметры табуляции, и в результате реальная ширина отступов будет различной. Кроме того, разные системные утилиты (печати, архивирования, управления версиями и др.) также имеют различную ширину табуляции.

Можно запретить сохранение символов табуляции в исходных файлах, сбросив флажки опций Use Tab Character и Optimal Fill во вкладке General диалогового окна Editor Properties, доступ к которому можно получить с помощью команды Tools⇨Editor Options.

Ширина поля

Устанавливайте ширину поля равной 80 символам. В общем случае исходный текст должен укладываться в указанную длину строки — за исключением тех ситуаций, когда необходимо закончить слово. Однако эта рекомендация не является слишком уж жесткой. Часть инструкции, которая не помещается в текущей строке, должна быть перенесена (если это возможно) на следующую строку. Можно разрывать инструкцию после любого символа запятой или некоторого оператора. Если инструкция занимает несколько строк, то продолжение на последующих строках оформляется с помощью отступа, равного двум символам, от ее начальной позиции.

Блок begin..end

Инструкция begin занимает отдельную строку. В следующих двух строках первая — неверна, а вторая служит примером правильного оформления инструкций программы:

```
for I := 0 to 10 do begin  // Неверно, так как инструкция begin находится
                           // в одной строке со словом for

for I := 0 to 10 do
begin   // Верно, так как инструкция begin находится в отдельной строке
```

Исключением из этого правила является случай, когда инструкция begin оказывается частью условного предложения else, например:

```
if инструкция1 = then
begin
...
end
else begin
  инструкция2;
end;
```

Инструкция end всегда занимает отдельную строку.

Если инструкция begin не является частью предложения else, то соответствующая ей инструкция end всегда имеет тот же отступ, что и begin.

Язык Object Pascal

Круглые скобки

Между открывающей круглой скобкой и следующим символом не должно быть пробела, как и между закрывающей круглой скобкой и предыдущим символом, например:

```
CallProc ( Aparameter );    // Неверно
CallProc (Aparameter);      // Верно
```

Никогда не включайте в инструкцию лишних круглых скобок. Их нужно использовать только в тех случаях, когда они позволяют получить требуемое значение исходного кода, например:

```
if (I = 42) then               // Неверно – лишние круглые скобки
if (I = 42) or (J = 42) then   // Верно – круглые скобки необходимы
```

Зарезервированные и ключевые слова

В языке Object Pascal зарезервированные и ключевые слова всегда должны записываться строчными буквами.

Процедуры и функции (подпрограммы)

Присвоение имен и форматирование

Имена подпрограмм всегда должны начинаться с прописной буквы. С прописных букв также должны начинаться и составные части имен — это облегчит их восприятие. Например, имя следующей процедуры отформатировано неверно (опустим тот факт, что для именования процедур должны использоваться только символы английского алфавита):

```
procedure этонеудачноеимядляпроцедуры;
```

Совсем другое дело, когда составные части имени процедуры начинаются с прописной буквы:

```
procedure ЭтоИмяПроцедурыПрочитатьГораздоЛегче;
```

Подпрограммам следует присваивать имена, которые отражают их назначение. Имена процедур, выполняющих некоторое действие, должны начинаться со слова, обозначающего это действие, например:

```
procedure FormatHardDrive;    // Процедура форматирования жесткого диска
```

Имена процедур, устанавливающих значения, должны начинаться со слова `set`, например:

```
procedure SetUserName;    // Процедура установки имени пользователя
```

Имена подпрограмм, осуществляющих выборку значений, следует начинать со слова `get`, например:

```
function GetUserName: string;    // Функция получения имени пользователя
```

Формальные параметры

Форматирование

Формальные параметры одного и того же типа следует по возможности объединять в одну инструкцию:

```
procedure Foo(Param1, Param2, Param3: integer; Param4: string);
```

Присвоение имен

Все имена формальных параметров должны соответствовать их назначению. Как правило, их следует образовывать на основе имени идентификатора, который передается в подпрограмму. Когда это уместно, имена параметров предваряются символом `A`, например:

```
procedure SomeProc(AUserName: string; AUserAge: Integer);
```

Префикс `A` применяется для устранения неоднозначности, если имя параметра совпадает с именем свойства или поля в классе.

Порядок следования параметров

Порядок следования параметров имеет важное значение, поскольку здесь используются преимущества регистровых вызовов.

Чаще всего используемые (вызывающей стороной) параметры следует указывать в начале списка, а те, которые применяются реже — в конце, т.е. снижение частоты использования параметров должно происходить в порядке слева направо.

Список входных параметров должен предшествовать списку выходных (также в порядке слева направо).

Более общие параметры следует размещать перед параметрами узкого назначения в порядке слева направо, например: `SomeProc(APlanet, AContinent, ACountry, AState, ACity)`.

В правиле следования параметров возможны исключения, например при использовании обработчиков событий, когда параметр с именем `Sender` (отправитель) типа `TObject` часто передается первым.

Параметры-константы

Если параметры записи, массива, строки `ShortString` или типа интерфейса не модифицируются подпрограммой, то формальные параметры для этой подпрограммы следует помечать зарезервированным словом `const`. В этом случае компилятор сгенерирует программный код, согласно которому эти немодифицируемые параметры будут передаваться самым эффективным образом.

Параметры других типов, если они не модифицируются подпрограммой, могут либо помечаться как `const`, либо нет — на эффективность это не окажет никакого влияния. Однако использование слова `const` предоставит источнику вызова этой подпрограммы больше информации о характере использования данного параметра.

Коллизии, связанные с именами

При использовании двух модулей, содержащих подпрограммы с одинаковым именем, будет вызываться та подпрограмма, модуль которой в предложении `uses` расположен последним. Во избежание подобных неоднозначностей, зависящих от структуры предложения `uses`, всегда предваряйте вызовы таких методов или подпрограмм именем соответствующего модуля, например:

```
SysUtils.FindClose(SR);
```

или

```
Windows.FindClose(Handle);
```

Переменные

Присвоение имен и форматирование

Переменным следует присваивать имена, соответствующие их назначению.

Переменным управления циклом обычно присваивается имя, состоящее из одного символа, например `I`, `J` или `K`. Но ничто не мешает использовать для них и гораздо более выразительные имена типа `UserIndex`.

Имена булевых переменных должны быть достаточно наглядными, чтобы их значения `True` и `False` были вполне ясными.

Локальные переменные

Локальные переменные, используемые внутри процедур, подчиняются тем же соглашениям об именах, которые распространяются на все остальные переменные. Временные переменные должны иметь соответствующие имена.

При необходимости инициализация локальных переменных выполняется сразу при входе в подпрограмму. Локальные переменные типа `AnsiString` автоматически инициализируются значениями, равными пустой строке, локальные переменные типа интерфейса и диспинтерфейса — значениями `nil`, а локальные переменные типа `Variant` и `OleVariant` — значениями `Unassigned`.

Использование глобальных переменных

Использовать глобальные переменные не рекомендуется. Но в случае необходимости с ними можно работать, не изменяя контекста их использования. Например, глобальная переменная может быть глобальной только внутри области видимости раздела реализации одного модуля.

Глобальные данные, предназначенные для использования несколькими модулями, следует перенести в открытый модуль, доступный всем объектам.

Глобальные данные могут быть инициализированы любыми значениями в разделе `var`. Но имейте в виду, что глобальные данные автоматически инициализированы нулями, поэтому не следует прибегать к таким "пустым" значениям, как `0`, `nil`, `''`, `Unassigned` и т.п. Дело в том, что глобальные данные, инициализированные нулями, не занимают места в `.exe`-файле. Они хранятся в виртуальном сегменте данных, который размещается в памяти только при запуске приложения. Глобальные данные, инициализированные ненулевыми значениями, занимают место в `.exe`-файле на диске.

Типы

Соглашение о выделении прописными буквами

Названия типов, которые являются зарезервированными словами, должны быть полностью написаны строчными буквами. Типы Win32 API обычно полностью пишутся прописными буквами; при этом вам следует соблюдать соглашение для определенного имени типа, предложенное в модуле `Windows.pas` или любом другом модуле API. Другие имена типов переменных должны начинаться с прописной буквы, кроме того, с прописной должны начинаться и составные части имени, что облегчит его восприятие. Вот несколько примеров:

```
var
  MyString: string;        // Зарезервированное слово
  WindowHandle: HWND;      // Тип Win32 API
  I: Integer;              // Идентификатор типа, введенный в модуле System
```

Типы значений с плавающей точкой

Использовать тип `Real` не рекомендуется, поскольку он существует только для обратной совместимости с программами, написанными на более старых версиях языка Pascal. Хотя он совершенно аналогичен типу `Double`, использование его может вызвать сомнения у других разработчиков. В общем случае для работы с данными в плавающем формате следует использовать тип `Double`. Это как раз тот тип, под который оптимизированы инструкции процессора и шины. Кроме того, тип `Double` является форматом данных, определенным стандартом IEEE представления чисел с плавающей точкой. Тип `Extended` используйте только в тех случаях,

когда для представления данных требуется диапазон больший, чем предлагает тип Double. Тип Extended определен фирмой Intel и не поддерживается в языке Java. Используйте тип Single только тогда, когда важен физический размер байта вещественной переменной (например, при использовании библиотек DLL, написанных на других языках).

Перечислимые типы

Имена для перечислимых типов следует выбирать в соответствии с назначением перечисления. Имя типа должно начинаться с буквы T, которая указывает на объявление типа. Список идентификаторов перечислимого типа должен содержать набранный строчными буквами двух- или трехсимвольный префикс, указывающий на имя исходного перечислимого типа, например:

```
TSongType = (stRock, stClassical, stCountry, stAlternative, stHeavyMetal, stRB);
```

Экземплярам переменных перечислимого типа следует присваивать имена, совпадающие с именем типа, но без префикса T (SongType), если нет причины назвать переменную более конкретным именем, например FavoriteSongType1, FavoriteSongType2 и т.д.

Типы Variant и OleVariant

Использовать типы Variant и OleVariant в основном не рекомендуется, однако они необходимы в тех случаях, когда типы данных становятся известны только во время выполнения программы, что зачастую характерно для приложений с доступом к базам данных и при использовании COM. Используйте тип OleVariant для таких COM-ориентированных задач, как автоматизация и элементы управления ActiveX. А тип Variant лучше применять для задач, не ориентированных на COM. Дело в том, что тип Variant может эффективно хранить собственные строки Delphi (подобно типу string), а тип OleVariant преобразует все строки к типу WideChar.

Структурированные типы

Массивы

Имена, подбираемые для типов массивов, должны отражать назначение массива. Имя типа начинается с буквы T. Если объявляется указатель на тип массива, то его имя начинается с буквы P и это объявление находится непосредственно перед объявлением типа массива, например:

```
type
  PCycleArray = ^TCycleArray;
  TCycleArray = array[1..100] of integer;
```

Как правило, экземплярам переменных типа массива присваиваются имена, совпадающие с именем типа, но без префикса T.

Записи

Имена, используемые для типов записей, должны отражать их назначение. Имя типа начинается с буквы T. Если объявляется указатель на тип записи, то его имя начинается с буквы P и это объявление находится непосредственно перед объявлением типа записи. Объявление типа для каждого элемента можно произвольно располагать в столбце справа, например:

```
type
  PEmployee = ^TEmployee;
  TEmployee = record
```

```
    EmployeeName: string
    EmployeeRate: Double;
  end;
```

Инструкции

Инструкции if

Наиболее вероятный вариант выполнения инструкции if/then/else следует размещать в предложении then, а наименее вероятные варианты — в предложении (предложениях) else.

Старайтесь избегать нагромождения инструкций if — вместо них лучше использовать (если это возможно) инструкции case.

Не следует допускать вложения инструкций if на глубину более пяти уровней. Найдите более красивое решение.

Не используйте лишних круглых скобок в инструкции if.

Если с помощью инструкции if проверяется несколько условий, то их следует расположить слева направо в порядке возрастания интенсивности вычислений. При этом программа сможет использовать преимущества встроенной в компилятор логики оценки вычислений по краткой схеме. Например, если условие Condition1 проверяется быстрее, чем условие Condition2, а условие Condition2 — быстрее, чем условие Condition3, то инструкция if должна быть построена следующим образом:

```
if Condition1 and Condition2 and Condition3 then
```

Инструкции case

Общие замечания

Отдельные случаи в инструкции case должны быть упорядочены либо по числам, либо по алфавиту.

Инструкции действий каждого рассматриваемого случая желательно не усложнять (они не должны превышать четырех-пяти программных строк). Если выполняемые действия более сложные, то этот фрагмент кода следует оформить в виде отдельной процедуры или функции.

Предложение else инструкции case следует использовать только для значений, устанавливаемых по умолчанию, или для обнаружения ошибок.

Форматирование

Инструкции case подчиняются тем же правилам форматирования, связанным с формированием отступов и соглашением по присвоению имен, что и другие конструкции.

Инструкции while

Использовать процедуру Exit для выхода из цикла while не рекомендуется. Если возможно, выходите из цикла только с помощью условия выполнения цикла.

Все инструкции инициализации для цикла while выполняются непосредственно перед входом в цикл; при этом среди них не должно встречаться инструкций, не связанных с инициализацией этого цикла.

Любые вспомогательные инструкции, связанные с результатом работы цикла, необходимо располагать сразу после цикла.

Инструкции for

Инструкции for следует использовать вместо инструкций while в тех случаях, если некоторый программный код должен выполниться известное число раз.

Инструкции repeat

Инструкции repeat аналогичны циклам while, и при работе с ними следует придерживаться тех же общих правил.

Инструкции with

Общие замечания

Инструкции with необходимо применять при достаточных на то основаниях и с большой осторожностью. Не следует злоупотреблять использованием инструкций with вообще и количеством применяемых в них объектов и записей в частности. Например, программные строки

```
with Record1, Record2 do
```

могут ввести программиста в заблуждение и послужить причиной появления ошибок, которые впоследствии будет трудно выявить.

Форматирование

Инструкции with подчиняются тем же правилам форматирования, связанным с формированием отступов и соглашением по присвоению имен, которые описаны в этом документе.

Структурированная обработка исключительных ситуаций

Общие замечания

Обработку исключительных ситуаций следует широко использовать как для исправления ошибок, так и для защиты ресурсов. Это значит, что во всех случаях назначения ресурсов необходимо использовать конструкцию try..finally для их корректного освобождения. Исключениями из этого правила являются случаи, когда ресурсы назначаются или освобождаются в разделе инициализации/завершения модуля или в конструкторе/деструкторе объекта.

Использование конструкции try..finally

Везде, где это возможно, каждому назначению ресурсов должна соответствовать конструкция try..finally. Например, следующий код может привести к возникновению ошибок:

```
SomeClass1 := TSomeClass.Create
SomeClass2 := TSomeClass.Create;
try
  { Некоторый код }
finally
  SomeClass1.Free;
  SomeClass2.Free;
end;
```

Более безопасный подход к приведенному выше назначению ресурсов будет выглядеть следующим образом:

```
SomeClass1 := TSomeClass.Create
try
  SomeClass2 := TSomeClass.Create;
  try
    { Некоторый код }
  finally
    SomeClass2.Free;
  end;
finally
  SomeClass1.Free;
end;
```

Использование конструкции try..except

Используйте конструкцию try..except только в том случае, когда при возникновении исключительной ситуации требуется выполнить некоторые важные действия. В общем случае не следует использовать блоки try..except лишь для того, чтобы просто отобразить на экране сообщение об ошибке, поскольку это будет сделано автоматически в контексте приложения объектом Application. Если вы хотите активизировать стандартную обработку исключительной ситуации после выполнения некоторой задачи в блоке except, используйте ключевое слово raise, чтобы повторно сгенерировать данную исключительную ситуацию для следующего обработчика.

Использование конструкции try..except..else

Использовать предложение else вместе с конструкцией try..except не рекомендуется, поскольку оно заблокирует все исключительные ситуации, в том числе не предусмотренные вами.

Классы

Присвоение имен и форматирование

Имена типов для классов должны отвечать назначению классов. Имя типа должно начинаться с буквы T, которая служит для указания определения типа, например:

```
type
  TCustomer = class(TObject)
```

Имена экземпляров для классов должны совпадать с именем типа класса, но не содержать букву T, например:

```
var
  Customer: TCustomer;
```

На заметку Для получения дополнительной информации о присвоении имен компонентам обращайтесь к разделу “Стандарты присвоения имен типам компонентов”.

Поля

Присвоение имен и форматирование

Имена полей класса подчиняются соглашениям о присвоении имен, предусмотренным для идентификаторов переменных, за исключением того, что они должны начинаться с префикса `F`, указывающего на имя поля (англ. ***f****ield*).

Видимость

Все поля должны быть закрытыми (`private`). К полям, которые должны быть доступны вне области видимости класса, следует предоставлять доступ только с помощью свойств.

Методы

Присвоение имен и форматирование

Имена методов подчиняются тем же соглашениям о присвоении имен, которые описаны в этом документе для процедур и функций.

Использование статических методов

Используйте статические методы, если вы не хотите, чтобы некоторый метод был переопределен классами-потомками.

Использование виртуальных/динамических методов

Если предполагается, что некоторый метод обязательно будет переопределяться классами-потомками, объявляйте его как виртуальный. Динамические методы следует использовать только в классах, для которых предполагается существование многочисленных потомков (прямых или непрямых). Например, класс, содержащий один редко переопределяемый метод и сто классов-потомков, должен сделать этот метод динамическим, чтобы уменьшить использование памяти сотней классов-потомков.

Использование абстрактных методов

Не используйте абстрактные методы в классах, для которых будут создаваться экземпляры. Применяйте ключевое слово `abstract` только к методам базовых классов, для которых не предполагается создание экземпляров.

Методы доступа к свойствам

Все методы доступа размещаются в закрытых или защищенных разделах определения класса.

Соглашения о присвоении имен методам доступа к свойствам содержат те же правила, которые описаны в этом документе для процедур и функций. Имя метода доступа "для чтения" (метода чтения) должно начинаться со слова `Get`. Имя метода доступа "для записи" (метода записи) должно начинаться со слова `Set`. Параметр метода записи должен иметь имя `Value`, а его тип должен совпадать с типом свойства, которое он представляет, например:

```
TSomeClass = class(TObject)
private
  FSomeField: Integer;
protected
  function GetSomeField: Integer;
  procedure SetSomeField( Value: Integer);
public
  property SomeField: Integer read GetSomeField write SetSomeField;
end;
```

Свойства

Присвоение имен и форматирование

Имена свойств, которые служат в качестве средства доступа к закрытым полям, должны совпадать с полями, которые они представляют, но без начальной буквы `F`.

Имена свойств должны быть существительными, а не глаголами. Свойства представляют данные, а методы — действия.

Имена свойств массивов имеют форму множественного числа, в то время как имена обычных свойств — единственного.

Использование методов доступа

Даже если это явно не требуется, рекомендуется создавать, как минимум, методы записи для всех свойств, которые представляют закрытое поле.

Файлы

Файлы проекта

Присвоение имен

Файлам проектов следует присваивать описательные имена. Например, проект *The Delphi 5 Developer's Guide Bug Manager* будет иметь имя `DDGBugs.dpr`. Программа отображения системной информации может иметь такое имя, как `SysInfo.dpr`.

Файлы форм

Присвоение имен

Файлам форм следует присваивать имена, описывающие назначение соответствующих форм; завершать их нужно тремя символами `Frm`. Например, файлу формы About (О программе…) следует присвоить имя `AboutFrm.dpr`, а файлу формы Main — имя `MainFrm.dpr`.

Файлы модулей данных

Присвоение имен

Модулям данных следует присваивать имена, описывающие их назначение; завершать их нужно двумя символами `DM`. Например, модуль данных `Customers` будет иметь имя файла `CustomersDM.dfm`.

Файлы модулей удаленных данных

Присвоение имен

Модулям удаленных данных следует присваивать имена, описывающие их назначение; завершать их нужно символами `RDM`. Например, модуль удаленных данных `Customers` будет иметь имя файла `CustomersRDM.dfm`.

Файлы модулей

Общая структура модулей

Имя модуля

Файлам модулей следует давать описательные имена. Например, модуль, содержащий главную форму приложения, будет называться `MainFrm.pas`.

Предложения `uses`

Предложение `uses` в разделе интерфейса должно содержать только те модули, которые требует программный код в этом разделе. Удалите любые лишние имена модулей, которые могли быть автоматически вставлены системными средствами Delphi.

Предложение `uses` в разделе реализации должно содержать только те модули, которые требуются программным кодом в этом разделе. Любые лишние имена модулей следует удалить.

Раздел интерфейса

Этот раздел должен включать объявления только тех типов и переменных, а также упреждающие объявления тех процедур/функций и методов, к которым должен быть обеспечен доступ со стороны внешних модулей. Все остальные объявления необходимо перенести в раздел реализации.

Раздел реализации

Данный раздел содержит любые объявления для типов, переменных, процедур/функций (и тому подобные) данного модуля, которые должны быть закрытыми.

Раздел инициализации

В разделе инициализации модуля не следует размещать программный код, требующий больших затрат времени. Это создаст впечатление инертности приложения при его запуске.

Раздел завершения

Убедитесь, что вы освободили все элементы, назначенные в разделе инициализации.

Модули форм

Присвоение имен

Файлу модуля формы присваивается имя, совпадающее с именем соответствующего файла формы. Например, форма About должна иметь имя файла модуля `AboutFrm.pas`, а главная форма Main — имя файла модуля `MainFrm.pas`.

Модули данных

Файлы модулей данных должны иметь те же имена, что и соответствующие им файлы форм. Например, модуль данных `Customers` будет иметь имя `CustomersDM.pas`.

Модули общего назначения

Присвоение имен

Модулям общего назначения необходимо присваивать имена в соответствии с их назначением. Например, модулю утилит можно присвоить имя `BugUtilities.pas`. Модуль, содержащий глобальные переменные, заслуживает имени `CustomersGlobals.pas`.

Имейте в виду, что имена модулей должны быть уникальными во всех пакетах, используемых данным проектом. Давать модулям групповые имена не рекомендуется.

Модули компонентов

Присвоение имен

Модули компонентов необходимо размещать в отдельном каталоге, чтобы модули определения компонентов или наборов компонентов не смешивались с другими модулями. Никогда не размещайте их в одном каталоге с проектом. Имя такого модуля должно соответствовать его содержимому.

На заметку Для получения дополнительной информации о стандартах присвоения имен компонентам обращайтесь к разделу “Компоненты, определяемые пользователем”.

Заголовки файлов

Для всех исходных файлов, файлов проекта, модулей и тому подобных рекомендуется использовать информационный заголовок файла, который должен содержать следующую информацию:

```
{
Copyright © ГОД АВТОРЫ
}
```

Формы и модули данных

Формы

Стандарт присвоения имен типам форм

Типам форм следует присваивать имена, описывающие назначение форм и начинающиеся с буквы `T`. За этим префиксом должно следовать описательное имя, которое завершается словом `Form`. Например, имя типа для формы About будет следующим:

```
TAboutForm = class(TForm)
```

Определение главной формы имеет вид

```
TMainForm = class(TForm)
```

Имя формы для ввода данных о клиенте будет таким:

```
TCustomerEntryForm = class(TForm)
```

Стандарт присвоения имен экземплярам форм

Экземпляры форм должны получать имена, совпадающие с именами соответствующих типов, но без префикса T. Например, имена экземпляров для типов описанных выше форм будут иметь следующий вид:

Имя типа	Имя экземпляра
TAboutForm	AboutForm
TMainForm	MainForm
TCustomerEntryForm	CustomerEntryForm

Формы, создаваемые автоматически

Автоматически будет создаваться только главная форма (если не существует веской причины, чтобы было по-другому). Все остальные формы должны быть удалены из списка автоматического создания, расположенного в диалоговом окне Project Options. За дополнительной информацией обращайтесь к следующему разделу.

Функции реализации модальных форм

Все модули форм должны содержать функцию реализации формы, которая предназначена для создания, настройки, отображения формы в модальном режиме и ее последующего освобождения. Эта функция возвращает результат, формируемый самой формой. Параметры, передаваемые в данную функцию, должны отвечать стандарту “передачи параметров”, который определяется этим документом. Функция должна быть инкапсулирована, что позволит многократно использовать ее программный код и упростит ее сопровождение.

Переменная формы должна быть удалена из модуля и объявлена локально в функции реализации формы. Обратите внимание на то, что при этом придется удалить форму из списка автоматического создания в диалоговом окне Project Options.

Приведенный ниже пример иллюстрирует создание подобной функции для формы GetUserData.

```
unit UserDataFrm;

interface

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

type
  TUserDataForm = class(TForm)
    edtUserName: TEdit;
    edtUserID: TEdit;
  private
    { Закрытые объявления }
  public
```

```
    { Открытые объявления }
  end;

function GetUserData(var aUserName: String; var aUserID: Integer): Word;

implementation
{$R *.DFM}

function GetUserData(var aUserName: String; var aUserID: Integer): Word;
var
  UserDataForm: TUserDataForm;
begin
  UserDataForm := TUserDataForm.Create(Application);
  try
    UserDataForm.Caption := 'Получение данных от пользователя';
    Result := UserDataForm.ShowModal;
    if ( Result = mrOK ) then begin
      aUserName := UserDataForm.edtUserName.Text;
      aUserID   := StrToInt(UserDataForm.edtUserID.Text);
    end;
  finally
    UserDataForm.Free;
  end;
end;

end.
```

Модули данных

Стандарт присвоения имен модулям данных

Типам модулей данных следует присваивать имена, описывающие их назначение и начинающиеся с буквы `T`. За этим префиксом должно следовать описательное имя, к которому присоединяется слово `DataModule`. Например, имя типа для модуля данных `Customer` должно выглядеть следующим образом:

```
TCustomerDataModule = class(TDataModule)
```

Аналогично, модуль данных `Orders` следует назвать так:

```
TOrdersDataModule = class(TDataModule)
```

Стандарт присвоения имен экземплярам модулей данных

Экземпляры модулей данных следует называть так же, как и соответствующие им типы, но без префикса `T`. Например, для типов описанных выше форм имена экземпляров будут иметь следующий вид:

Имя типа	Имя экземпляра
`TCustomerDataModule`	`CustomerDataModule`
`TOrdersDataModule`	`OrdersDataModule`

Пакеты

Пакеты времени выполнения и пакеты времени разработки

Пакеты времени выполнения должны содержать только те модули и компоненты, которые требуются другими компонентами в этом пакете. Другие модули, содержащие редакторы свойств или компонентов и другие средства разработки программного кода, должны размещаться в пакете времени конструирования. Модули регистрации также нужно размещать в пакете времени разработки.

Стандарты присвоения имен файлам

Пакеты нужно называть в соответствии со следующими шаблонами:

- *iii*lib*vv*.dpk — пакет времени конструирования;
- *iii*std*vv*.dpk — пакет времени выполнения.

Здесь символы *iii* означают трехсимвольный префикс идентификации, который можно использовать для указания компании, лица или любой другой идентификационной информации.

Символы *vv* означают версию пакета, соответствующую версии Delphi, для которой предназначен данный пакет.

Обратите внимание на то, что имя пакета содержит сочетание lib или std, означающие пакет времени выполнения (загрузочный пакет) либо пакет времени разработки.

В тех случаях, если существуют оба типа пакетов, файлы должны называться по одной схеме. Например, пакеты для данной книги (*Delphi 5 Developer's Guide*) имеют следующие имена:

- DdgLib50.dpk — пакет времени разработки;
- DdgStd50.dpk — пакет времени выполнения.

Компоненты

Компоненты, определяемые пользователем

Стандарты присвоения имен типам компонентов

Компоненты следует называть аналогично классам (см. раздел "Классы"), с той лишь разницей, что их имена должны предваряться трехсимвольным идентификационным префиксом. Этот префикс можно использовать для обозначения компании, разработчика или какой-либо другой категории. Например, компонент, отвечающий за работу часов (clock), который написан для данной книги (*Delphi 5. Руководство разработчика*), определяется следующим образом:

```
TddgClock = class(TComponent)
```

Заметьте, что в трехсимвольном префиксе использованы все строчные буквы.

Модули компонентов

Модуль компонента должен содержать только один основной компонент. *Основной компонент* — это такой компонент, который отображается в палитре компонентов (Component Palette). В одном модуле с основным компонентом могут располагаться любые вспомогательные компоненты и объекты.

Использование модулей регистрации

Процедуру регистрации для компонентов необходимо удалить из модуля компонента и разместить в отдельном модуле, который предназначен для регистрации любых компонентов, редакторов свойств, редакторов компонентов, экспертов и т.п.

Регистрация компонентов выполняется только в пакетах времени конструирования, поэтому модуль регистрации должен быть включен в состав пакета разработки, а не в состав пакета времени выполнения.

Предлагается модули регистрации называть следующим образом:

*Xxx*Reg.pas

Здесь *Xxx* — это трехсимвольный префикс, используемый для идентификации компании, разработчика или другой категории. Например, модулю регистрации для компонентов этой книги можно было бы присвоить имя DdgReg.pas.

Соглашения о присвоении имен экземплярам компонентов

Всем компонентам необходимо давать описательные имена. Не следует оставлять для компонентов имена, присвоенные им Delphi по умолчанию. Для построения имен компонентов следует использовать один из вариантов венгерской нотации. В соответствии с этим соглашением имя компонента должно состоять из двух частей: префикса типа и квалификатора имени.

Префиксы типов компонентов

Префикс типа компонента указывается строчными буквами и определяет тип компонента. Например, ниже представлены допустимые префиксы типа для некоторых компонентов.

Tbutton	btn
TEdit	edt
TSpeedButton	spdbtn
TListBox	lstbx

Как следует из приведенного примера, префикс типа компонента создается посредством модификации имени типа компонента (TButton или TEdit) по следующему правилу:

1. Удалите из имени компонента префикс T. Например, TButton превращается в Button.
2. Из полученного значения удалите все гласные буквы, за исключением первых букв слова. Например, Button превращается в bttn, а Edit превращается в edt.
3. Удалите сдвоенные согласные буквы. Например, bttn превращается в btn.
4. Если в результате возникает конфликт имен, возвращайте в полученное промежуточное значение гласные буквы — по одной, слева направо. Например, если появится новый компонент TBatton, его префикс типа войдет в конфликт с префиксом типа компонента TButton. Следовательно, для нового компонента следует установить префикс типа batn.

Квалифицированное имя компонента

Квалифицированное имя компонента должно соответствовать назначению компонента. Например, компоненту типа `TButton`, предназначенному для закрытия формы, следует присвоить имя `btnClose`, а компоненту типа `TEdit`, предназначенному для ввода фамилии, — имя `edtFirstName`.

Сведения об изменении стандартов программирования

Приведенные выше соглашения регулярно обновляются с целью отражения дополнений и расширений языка Object Pascal и библиотеки VCL. Самую свежую версию этого документа всегда можно получить, обратившись по адресу: `http://www.xapware.com/ddg`.

Глава

7

Использование элементов управления ActiveX в Delphi

Одно из важных достоинств Delphi заключается в простоте интеграции в пользовательские приложения стандартизованных элементов управления ActiveX (ранее известных как элементы управления OCX и OLE). В отличие от собственных компонентов Delphi, элементы управления ActiveX не зависят от конкретных средств разработки. Это значит, что вы можете пользоваться услугами разных поставщиков, предлагающих разнообразные по свойствам и функциям решения ActiveX.

Поддержка элементов управления ActiveX в 32-разрядной системе Delphi реализована аналогично поддержке компонентов VBX в 16-разрядной системе Delphi 1. От вас требуется лишь выбрать в главном меню IDE Delphi или редактора пакетов Package Editor команду добавления новых элементов управления ActiveX — и Delphi построит Object Pascal-оболочку для этого элемента управления, который затем будет скомпилирован в пакет и добавлен в палитру компонентов IDE Delphi. После помещения в палитру компонентов новый элемент управления ActiveX объединяется с другими компонентами VCL и ActiveX. В дальнейшем, чтобы добавить этот элемент управления ActiveX в приложение, пользователю достаточно щелкнуть на нем и затем перетащить его из палитры в создаваемую форму. В этой главе речь пойдет об интеграции элементов управления ActiveX в среду разработки Delphi, использовании элементов управления ActiveX в приложении и особенностях распространения приложений, оснащенных элементами ActiveX.

На заметку

Delphi 1 была последней версией Delphi, которая поддерживала элементы управления VBX (Visual Basic Extension). Если у вас есть проект, который опирается на один или несколько таких элементов, проконсультируйтесь со специалистами фирм — поставщиков компонентов VBX и узнайте о равноценной замене компонентов VBX элементами ActiveX, которые можно будет использовать в среде 32-разрядной Delphi.

Что представляет собой элемент управления ActiveX

Элементы управления ActiveX представляют собой нестандартные элементы управления, разработанные для 16- и 32-разрядных приложений Windows, в которых используются преимущества технологий COM, OLE и ActiveX. В отличие от элементов управления VBX, которые были разработаны для 16-разрядной системы Visual Basic (а следовательно, им были присущи и все ограничения Visual Basic), в разработку элементов ActiveX с самого начала была заложена идея независимости от конкретных приложений. Грубо говоря, элементы ActiveX можно представить себе как результат слияния простой в применении технологии VBX с открытым стандартом ActiveX. Для понимания материала этой главы можно не углубляться в различия между OLE и ActiveX, но те, кого этот вопрос интересует, могут обратиться к главе 23 второго тома, “COM-ориентированные технологии”.

Если копнуть глубже, то в действительности элемент ActiveX представляет собой сервер ActiveX, который в одном пакете может предоставить всю мощь ActiveX, включая все функции и сервис OLE, визуальное редактирование, поддержку операций перетаскивания и автоматизацию OLE. Подобно всем серверам ActiveX, элементы управления ActiveX регистрируются в системном реестре. Элементы ActiveX можно разрабатывать с помощью таких продуктов, как Delphi, Borland C++ Builder, Visual C++ и Visual Basic.

Фирма Microsoft активно пропагандирует элементы ActiveX как одну из ведущих технологий создания пользовательских элементов управления, не зависящих от приложений. Фирма Microsoft заявила, что технология VBX не будет поддерживаться в операционных системах Win32 и выше. Поэтому при создании 32-разрядных приложений разработчикам следует обратить свои взоры на элементы управления ActiveX, а не на элементы VBX.

На заметку Более полное описание технологии ActiveX содержится в главе 25 второго тома, “Создание элементов управления ActiveX”.

Когда следует использовать элемент управления ActiveX

Обычно приводят две причины, заставляющие использовать элементы управления ActiveX вместо компонентов Delphi. Первая — это отсутствие компонентов Delphi, которые могли бы удовлетворять имеющимся конкретным требованиям. Поскольку рынок элементов управления ActiveX шире, чем рынок компонентов VCL, пользователь, вероятно, найдет в нем более широкий ассортимент таких полнофункциональных и конкурентоспособных элементов, как текстовые процессоры, броузеры World Wide Web или электронные таблицы. Вторая причина состоит в том, что в случае разработки с применением нескольких языков программирования и необходимости тестирования своего продукта с конкретными элементами управления на нескольких платформах, использование элементов управления ActiveX предпочтительнее, чем компонентов библиотеки VCL Delphi.

Несмотря на то что элементы управления ActiveX естественным образом интегрируются в IDE Delphi, однако существуют некоторые недостатки, связанные с использованием элементов ActiveX в приложениях. Самый очевидный из них состоит в том, что если компоненты Delphi встраиваются непосредственно в выполняемый модуль приложения, то элементы управления ActiveX обычно требуют одного или нескольких дополнительных загрузочных файлов, которые должны поставляться вместе с выполняемым модулем. Другая проблема заключается в том, что элементы управления ActiveX взаимодействуют с приложениями через промежуточный уровень COM, в то время как компоненты Delphi вступают в непосредственное взаимодействие с приложениями и другими компонентами. Это значит, что хорошо написанный компонент Delphi обычно работает эффективнее, чем хорошо написанный элемент управления ActiveX. Не слишком явно выраженным недостатком элементов управления ActiveX является и то, что их можно назвать решением типа “наименьшего общего знаменателя”, а потому они не способны реализовать все возможности тех средств разработки, в которых используются.

Внесение элемента управления ActiveX в палитру компонентов

Использование конкретного элемента управления ActiveX в приложении Delphi начинается с добавления его в палитру компонентов IDE Delphi. В результате этой операции пиктограмма элемента управления ActiveX появится в палитре компонентов среди других элементов Delphi и ActiveX, после чего ее можно будет перетаскивать в любую форму и использовать данный компонент подобно любому другому элементу управления Delphi.

Чтобы внести элемент управления ActiveX в палитру компонентов, выполните следующие действия.

1. Выберите в главном меню команду Component⇨Import ActiveX Control. Откроется диалоговое окно Import ActiveX, показанное на рис. 7.1.

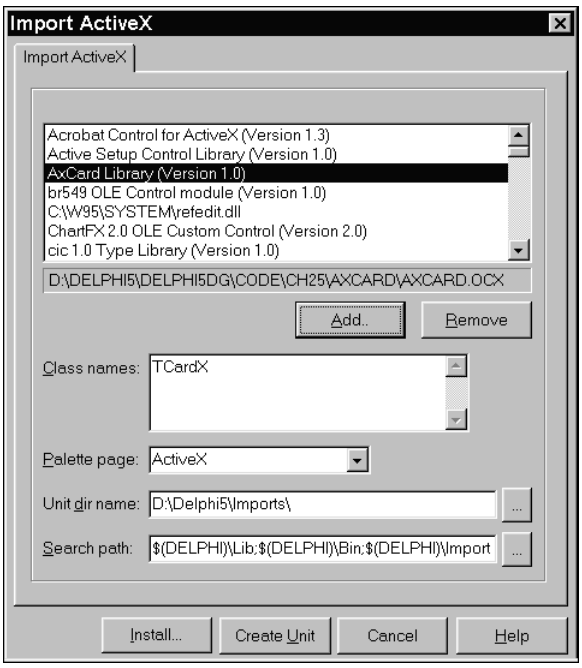


Рис. 7.1. Диалоговое окно Import ActiveX

2. Диалоговое окно Import ActiveX разделено на две части. В верхней содержится список зарегистрированных элементов управления ActiveX, а также две кнопки, Add и Remove, с помощью которых можно зарегистрировать или вычеркнуть элемент управления. В нижней части этого диалогового окна можно задать параметры для создания компонента Delphi и указать модуль, в котором данный элемент будет инкапсулирован.
3. Если имя элемента управления ActiveX, который вы хотите использовать, уже содержится в списке, расположенном в верхней части этого диалогового окна, переходите к п. 4. В противном случае щелкните на кнопке Add, чтобы зарегистрировать новый элемент в системе. По щелчку на кнопке Add откроется диалоговое окно Register OLE Control, показанное на рис. 7.2. Выберите имя OCX- или DLL-файла, который представляет элемент управления ActiveX, предназначенный для добавления в систему, и щелкните на кнопке Open. При этом выбранный элемент будет зарегистрирован в системном реестре, а диалоговое окно Register OLE Control, выполнив свою функцию, закроется.
4. В верхней части диалогового окна Import ActiveX Control выберите имя элемента управления ActiveX, который требуется добавить в палитру компонентов. В нижней части этого диалогового окна содержатся элементы, предназначенные для редактирования имени файла модуля, страницы палитры и маршрута поиска, а также мемо-поле, в котором отображается список классов, содержащихся внутри OCX-файла. Путь, выведенный в поле редактирования Unit Dir Name, определяет маршрут к компоненту оболочки Delphi, создаваемому для взаимодействия с элементом управления ActiveX. По умолчанию имя этого файла совпадает с именем `.OCX`-файла (с расширением `.pas`), а путь указывает на подкаталог `\Delphi 5\Imports`. И хотя использовать предложенные стандартные установки весьма удобно, при необходимости путь и имя файла можно отредактировать.

5. В поле редактирования Palette Page диалогового окна Import ActiveX содержится имя страницы палитры компонентов, на которую будет помещен данный элемент управления. По умолчанию предлагается страница ActiveX. Но существует возможность выбрать и другую имеющуюся страницу. Если ввести в это поле новое имя, в палитре компонентов будет создана соответствующая страница.

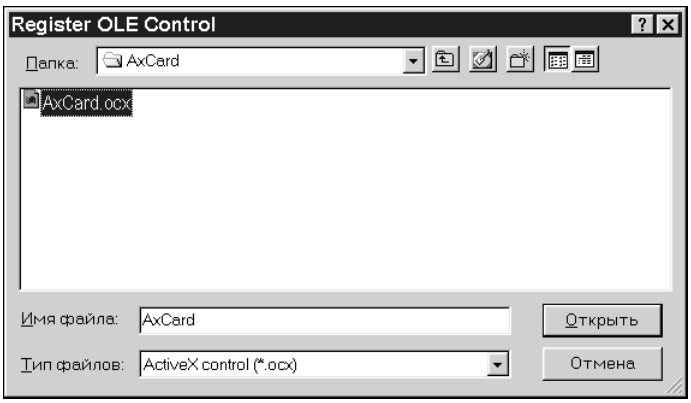


Рис. 7.2. Диалоговое окно Register OLE Control

6. В поле Class Names диалогового окна Import ActiveX содержатся имена новых объектов, созданных в этом элементе управления. Обычно данные имена оставляют неизмененными, если, конечно, нет веской причины поступить иначе. Причиной для вмешательства может служить конфликт имен между добавляемым классом и классом другого компонента, уже установленного в среде IDE.
7. На этом этапе в диалоговом окне Import ActiveX можно щелкнуть либо на кнопке Install, либо на кнопке Create Unit. По щелчку на кнопке Create Unit будет сгенерирован исходный текст программы, который войдет в модуль для оболочки компонента данного элемента управления ActiveX. Щелчок на кнопке Install вызовет генерацию программного кода оболочки, а затем откроет диалоговое окно Install, позволяющее выбрать пакет, в который будет инсталлирован новый компонент. Это окно показано на рис. 7.3.

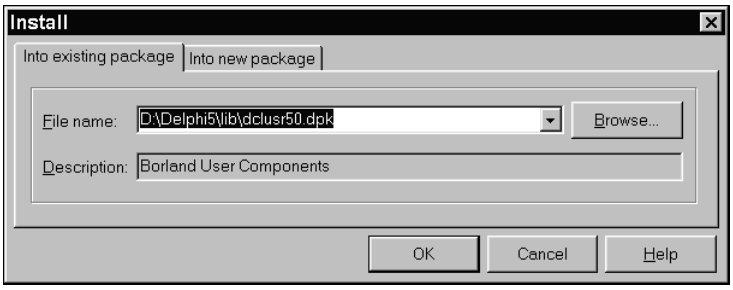


Рис. 7.3. Диалоговое окно Install

8. В диалоговом окне Install можно выбрать вариант либо добавления элемента управления к уже существующему пакету, либо создания совершенно нового пакета, который будет инсталлирован в палитру компонентов. Щелкните на кнопке OK, и компонент будет установлен в палитре.

Теперь новый элемент ActiveX находится в палитре компонентов и готов к работе.

Оболочка компонентов Delphi

Настало время обратить внимание на оболочку Object Pascal, создаваемую для инкапсуляции элемента управления ActiveX. Это прольет некоторый свет на то, как осуществляется поддержка технологии ActiveX в среде Delphi, и поможет вам получить четкое представление о возможностях и ограничениях, присущих элементам управления ActiveX. В листинге 7.1 демонстрируется текст модуля `Card_TLB.pas`, сгенерированного Delphi. Этот модуль инкапсулирует элемент ActiveX `AxCard.ocx`.

На заметку Описание разработки элемента управления ActiveX `AxCard.ocx` приведено в главе 25 второго тома, "Создание элементов управления ActiveX".

Листинг 7.1. Модуль оболочки компонента Delphi для элемента управления AxCard.ocx

```
unit AxCard_TLB;

// ************************************************************************ //
// ВНИМАНИЕ
// --------
// Типы, объявленные в этом файле, были сгенерированы на основании данных,
// считанных из библиотеки типов. Если эта библиотека типов явно или неявно
// (через другую библиотеку типов, ссылающуюся на эту библиотеку) будет
// импортирована заново или при ее редактировании в окне редактора библиотеки
// типов будет выбрана команда 'Refresh', содержимое этого файла будет
// сгенерировано заново, а все вручную внесенные в него изменения - потеряны.
// ************************************************************************ //

// PASTLWTR : $Revision:   1.88  $
// File generated on 08.24.99 12:38:15 from Type Library described below.

// ************************************************************************ //
// Type Lib: C:\DELPHI5\DELPHI5DG\CODE\CH25\AXCARD\AXCARD.OCX (1)
// IID\LCID: {7B33D940-0A2C-11D2-AE5C-04640BC10000}\0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\W95\SYSTEM\STDOLE2.TLB)
//   (2) v4.0 StdVCL, (C:\W95\SYSTEM\STDVCL40.DLL)
// ************************************************************************ //
{$TYPEDADDRESS OFF} // Модуль должен компилироваться без проверки типов
                    // указателей.
Interface

uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL;

// ************************************************************************//
// GUIDS объявлен в библиотеке типов. Используются следующие префиксы:
//   Библиотеки типов   : LIBID_xxxx
//   Классы компонентов : CLASS_xxxx
```

```
//   DISP-интефейсы     : DIID_xxxx
//   Non-DISP интерфейсы: IID_xxxx
// *********************************************************************//
const
  // Библиотеки типов версий Major и minor
  AxCardMajorVersion = 1;
  AxCardMinorVersion = 0;

  LIBID_AxCard: TGUID = '{7B33D940-0A2C-11D2-AE5C-04640BC10000}';

  IID_ICardX: TGUID = '{7B33D941-0A2C-11D2-AE5C-04640BC10000}';
  DIID_ICardXEvents: TGUID = '{7B33D943-0A2C-11D2-AE5C-04640BC10000}';
  CLASS_CardX: TGUID = '{7B33D945-0A2C-11D2-AE5C-04640BC10000}';

// *********************************************************************//
// Объявления перечислений, определенных в библиотеке типов
// *********************************************************************//
// Константы для перечисления TxDragMode
type
  TxDragMode = TOleEnum;
const
  dmManual = $00000000;
  dmAutomatic = $00000001;

// Константы для перечисления TxCardSuit
type
  TxCardSuit = TOleEnum;
const
  csClub = $00000000;
  csDiamond = $00000001;
  csHeart = $00000002;
  csSpade = $00000003;

// Константы для перечисления TxCardValue
type
  TxCardValue = TOleEnum;
const
  cvAce = $00000000;
  cvTwo = $00000001;
  cvThree = $00000002;
  cvFour = $00000003;
  cvFive = $00000004;
  cvSix = $00000005;
  cvSeven = $00000006;
  cvEight = $00000007;
  cvNine = $00000008;
  cvTen = $00000009;
  cvJack = $0000000A;
  cvQueen = $0000000B;
  cvKing = $0000000C;
```

```
// Константы для перечисления TxMouseButton
type
  TxMouseButton = TOleEnum;
const
  mbLeft = $00000000;
  mbRight = $00000001;
  mbMiddle = $00000002;

// Константы для перечисления TxAlignment
type
  TxAlignment = TOleEnum;
const
  taLeftJustify = $00000000;
  taRightJustify = $00000001;
  taCenter = $00000002;

// Константы для перечисления TxBiDiMode
type
  TxBiDiMode = TOleEnum;
const
  bdLeftToRight = $00000000;
  bdRightToLeft = $00000001;
  bdRightToLeftNoAlign = $00000002;
  bdRightToLeftReadingOnly = $00000003;

type

// ************************************************************************//
// Предварительное объявление интерфейсов, определяемых в библиотеке типов
// ************************************************************************//
  ICardX = interface;
  ICardXDisp = dispinterface;
  ICardXEvents = dispinterface;

// ************************************************************************//
// Объявление классов компонентов, определяемых в библиотеке типов
// (Замечание: Здесь устанавливается соответствие каждого класса компонента
//              своему стандартному интерфейсу)
// ************************************************************************//
  CardX = ICardX;


// ************************************************************************//
// Interface: ICardX
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {7B33D941-0A2C-11D2-AE5C-04640BC10000}
// ************************************************************************//
  ICardX = interface(IDispatch)
    ['{7B33D941-0A2C-11D2-AE5C-04640BC10000}']
    function  Get_BackColor: OLE_COLOR; safecall;
```

```
procedure Set_BackColor(Value: OLE_COLOR); safecall;
function  Get_Color: OLE_COLOR; safecall;
procedure Set_Color(Value: OLE_COLOR); safecall;
function  Get_DragCursor: Smallint; safecall;
procedure Set_DragCursor(Value: Smallint); safecall;
function  Get_DragMode: TxDragMode; safecall;
procedure Set_DragMode(Value: TxDragMode); safecall;
function  Get_FaceUp: WordBool; safecall;
procedure Set_FaceUp(Value: WordBool); safecall;
function  Get_ParentColor: WordBool; safecall;
procedure Set_ParentColor(Value: WordBool); safecall;
function  Get_Suit: TxCardSuit; safecall;
procedure Set_Suit(Value: TxCardSuit); safecall;
function  Get_Value: TxCardValue; safecall;
procedure Set_Value(Value: TxCardValue); safecall;
function  Get_DoubleBuffered: WordBool; safecall;
procedure Set_DoubleBuffered(Value: WordBool); safecall;
procedure FlipChildren(AllLevels: WordBool); safecall;
function  DrawTextBiDiModeFlags(Flags: Integer): Integer; safecall;
function  DrawTextBiDiModeFlagsReadingOnly: Integer; safecall;
function  Get_Enabled: WordBool; safecall;
procedure Set_Enabled(Value: WordBool); safecall;
function  GetControlsAlignment: TxAlignment; safecall;
procedure InitiateAction; safecall;
function  IsRightToLeft: WordBool; safecall;
function  UseRightToLeftAlignment: WordBool; safecall;
function  UseRightToLeftReading: WordBool; safecall;
function  UseRightToLeftScrollBar: WordBool; safecall;
function  Get_BiDiMode: TxBiDiMode; safecall;
procedure Set_BiDiMode(Value: TxBiDiMode); safecall;
function  Get_Visible: WordBool; safecall;
procedure Set_Visible(Value: WordBool); safecall;
function  Get_Cursor: Smallint; safecall;
procedure Set_Cursor(Value: Smallint); safecall;
function  ClassNameIs(const Name: WideString): WordBool; safecall;
procedure AboutBox; safecall;
property BackColor: OLE_COLOR read Get_BackColor write Set_BackColor;
property Color: OLE_COLOR read Get_Color write Set_Color;
property DragCursor: Smallint read Get_DragCursor write Set_DragCursor;
property DragMode: TxDragMode read Get_DragMode write Set_DragMode;
property FaceUp: WordBool read Get_FaceUp write Set_FaceUp;
property ParentColor: WordBool read Get_ParentColor write Set_ParentColor;
property Suit: TxCardSuit read Get_Suit write Set_Suit;
property Value: TxCardValue read Get_Value write Set_Value;
property DoubleBuffered: WordBool read Get_DoubleBuffered
  write Set_DoubleBuffered;
property Enabled: WordBool read Get_Enabled write Set_Enabled;
property BiDiMode: TxBiDiMode read Get_BiDiMode write Set_BiDiMode;
property Visible: WordBool read Get_Visible write Set_Visible;
property Cursor: Smallint read Get_Cursor write Set_Cursor;
```

```
  end;

// ************************************************************************//
// DispIntf:  ICardXDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {7B33D941-0A2C-11D2-AE5C-04640BC10000}
// ************************************************************************//
  ICardXDisp = dispinterface
    ['{7B33D941-0A2C-11D2-AE5C-04640BC10000}']
    property BackColor: OLE_COLOR dispid 1;
    property Color: OLE_COLOR dispid -501;
    property DragCursor: Smallint dispid 2;
    property DragMode: TxDragMode dispid 3;
    property FaceUp: WordBool dispid 4;
    property ParentColor: WordBool dispid 5;
    property Suit: TxCardSuit dispid 6;
    property Value: TxCardValue dispid 7;
    property DoubleBuffered: WordBool dispid 10;
    procedure FlipChildren(AllLevels: WordBool); dispid 11;
    function  DrawTextBiDiModeFlags(Flags: Integer): Integer; dispid 14;
    function  DrawTextBiDiModeFlagsReadingOnly: Integer; dispid 15;
    property Enabled: WordBool dispid -514;
    function  GetControlsAlignment: TxAlignment; dispid 16;
    procedure InitiateAction; dispid 18;
    function  IsRightToLeft: WordBool; dispid 19;
    function  UseRightToLeftAlignment: WordBool; dispid 24;
    function  UseRightToLeftReading: WordBool; dispid 25;
    function  UseRightToLeftScrollBar: WordBool; dispid 26;
    property BiDiMode: TxBiDiMode dispid 27;
    property Visible: WordBool dispid 28;
    property Cursor: Smallint dispid 29;
    function  ClassNameIs(const Name: WideString): WordBool; dispid 33;
    procedure AboutBox; dispid -552;
  end;

// ************************************************************************//
// DispIntf:  ICardXEvents
// Flags:     (4096) Dispatchable
// GUID:      {7B33D943-0A2C-11D2-AE5C-04640BC10000}
// ************************************************************************//
  ICardXEvents = dispinterface
    ['{7B33D943-0A2C-11D2-AE5C-04640BC10000}']
    procedure OnClick; dispid 1;
    procedure OnDblClick; dispid 2;
    procedure OnKeyPress(var Key: Smallint); dispid 7;
  end;


// ************************************************************************//
// Объявление Proxy-класса элемента управления OLE
```

```
// Имя элемента управления : TCardX
// Справочная строка       : CardX Control
// Стандартный интерфейс   : ICardX
// Станд. интерф. для DISP?: No
// Интерфейс событий       : ICardXEvents
// TypeFlags               : (34) CanCreate Control
// **********************************************************************//
  TCardXOnKeyPress = procedure(Sender: TObject; var Key: Smallint) of object;

  TCardX = class(TOleControl)
  private
    FOnClick: TNotifyEvent;
    FOnDblClick: TNotifyEvent;
    FOnKeyPress: TCardXOnKeyPress;
    FIntf: ICardX;
    function  GetControlInterface: ICardX;
  protected
    procedure CreateControl;
    procedure InitControlData; override;
  public
    procedure FlipChildren(AllLevels: WordBool);
    function  DrawTextBiDiModeFlags(Flags: Integer): Integer;
    function  DrawTextBiDiModeFlagsReadingOnly: Integer;
    function  GetControlsAlignment: TxAlignment;
    procedure InitiateAction;
    function  IsRightToLeft: WordBool;
    function  UseRightToLeftAlignment: WordBool;
    function  UseRightToLeftReading: WordBool;
    function  UseRightToLeftScrollBar: WordBool;
    function  ClassNameIs(const Name: WideString): WordBool;
    procedure AboutBox;
    property  ControlInterface: ICardX read GetControlInterface;
    property  DefaultInterface: ICardX read GetControlInterface;
    property DoubleBuffered: WordBool index 10 read GetWordBoolProp write
      SetWordBoolProp;
    property Enabled: WordBool index -514 read GetWordBoolProp write
      SetWordBoolProp;
    property BiDiMode: TOleEnum index 27 read GetTOleEnumProp write
      SetTOleEnumProp;
    property Visible: WordBool index 28 read GetWordBoolProp write
      SetWordBoolProp;
  published
    property  TabStop;
    property  Align;
    property  ParentShowHint;
    property  PopupMenu;
    property  ShowHint;
    property  TabOrder;
    property  OnDragDrop;
    property  OnDragOver;
```

```
    property  OnEndDrag;
    property  OnEnter;
    property  OnExit;
    property  OnStartDrag;
    property BackColor: TColor index 1 read GetTColorProp write
      SetTColorProp stored False;
    property Color: TColor index -501 read GetTColorProp write
      SetTColorProp stored False;
    property DragCursor: Smallint index 2 read GetSmallintProp write
      SetSmallintProp stored False;
    property DragMode: TOleEnum index 3 read GetTOleEnumProp write
      SetTOleEnumProp stored False;
    property FaceUp: WordBool index 4 read GetWordBoolProp write
      SetWordBoolProp stored False;
    property ParentColor: WordBool index 5 read GetWordBoolProp write
      SetWordBoolProp stored False;
    property Suit: TOleEnum index 6 read GetTOleEnumProp write
      SetTOleEnumProp stored False;
    property Value: TOleEnum index 7 read GetTOleEnumProp write
      SetTOleEnumProp stored False;
    property Cursor: Smallint index 29 read GetSmallintProp write
      SetSmallintProp stored False;
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    property OnDblClick: TNotifyEvent read FOnDblClick write OnDblClick;
    property OnKeyPress: TCardXOnKeyPress read FOnKeyPress write FOnKeyPress;
  end;

procedure Register;

implementation

uses ComObj;

procedure TCardX.InitControlData;
const
  CEventDispIDs: array [0..2] of DWORD = (
    $00000001, $00000002, $00000007);
  CControlData: TControlData2 = (
    ClassID: '{7B33D945-0A2C-11D2-AE5C-04640BC10000}';
    EventIID: '{7B33D943-0A2C-11D2-AE5C-04640BC10000}';
    EventCount: 3;
    EventDispIDs: @CEventDispIDs;
    LicenseKey: nil (*HR:$00000000*);
    Flags: $00000009;
    Version: 401);
begin
  ControlData := @CControlData;
  TControlData2(CControlData).FirstEventOfs :=
    Cardinal(@@FOnClick) - Cardinal(Self);
end;
```

```
procedure TCardX.CreateControl;

  procedure DoCreate;
  begin
    FIntf := IUnknown(OleObject) as ICardX;
  end;

begin
  if FIntf = nil then DoCreate;
end;

function TCardX.GetControlInterface: ICardX;
begin
  CreateControl;
  Result := FIntf;
end;

procedure TCardX.FlipChildren(AllLevels: WordBool);
begin
  DefaultInterface.FlipChildren(AllLevels);
end;

function  TCardX.DrawTextBiDiModeFlags(Flags: Integer): Integer;
begin
  Result := DefaultInterface.DrawTextBiDiModeFlags(Flags);
end;

function  TCardX.DrawTextBiDiModeFlagsReadingOnly: Integer;
begin
  Result := DefaultInterface.DrawTextBiDiModeFlagsReadingOnly;
end;

function  TCardX.GetControlsAlignment: TxAlignment;
begin
  Result := DefaultInterface.GetControlsAlignment;
end;

procedure TCardX.InitiateAction;
begin
  DefaultInterface.InitiateAction;
end;

function  TCardX.IsRightToLeft: WordBool;
begin
  Result := DefaultInterface.IsRightToLeft;
end;

function  TCardX.UseRightToLeftAlignment: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftAlignment;
```

```
end;

function  TCardX.UseRightToLeftReading: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftReading;
end;

function  TCardX.UseRightToLeftScrollBar: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftScrollBar;
end;

function  TCardX.ClassNameIs(const Name: WideString): WordBool;
begin
  Result := DefaultInterface.ClassNameIs(Name);
end;

procedure TCardX.AboutBox;
begin
  DefaultInterface.AboutBox;
end;

procedure Register;
begin
  RegisterComponents('ActiveX',[TCardX]);
end;

end.
```

Теперь, ознакомившись с программным текстом, сгенерированным редактором библиотеки типов, рассмотрим детали работы механизма импортирования библиотек типов.

Откуда берутся файлы оболочек

Первое, что бросается в глаза, — это окончание `_TLB` в имени файла. Кроме того, можно заметить, что в сгенерированном исходном файле есть несколько ссылок на библиотеки. Оба этих факта наводят на мысль, что первоисточником для построения файла оболочки служит библиотека типов конкретного элемента управления. Библиотека типов любого элемента управления ActiveX представляет собой специальную информацию, которая связывается с ним в виде ресурса, описывающего различные элементы этого компонента. В частности, библиотеки типов содержат информацию об интерфейсах, поддерживаемых данным элементом управления, свойствах, методах и событиях, а также используемых перечислимых типах. Первая запись в файле оболочки содержит уникальный идентификационный номер GUID библиотеки типов данного элемента управления.

На заметку

Библиотеки типов используются достаточно широко — для объектов автоматизации OLE любого типа. Подробнее о библиотеках типов и их использовании можно узнать в главе 23 второго тома, "COM-ориентированные технологии".

Перечисления

Сразу за идентификационным номером GUID библиотеки типов располагаются перечислимые типы, используемые данным элементом управления. Обратите внимание, что перечисления объявляются как простые константы, а не как настоящие перечислимые типы. Дело в том, что перечисления библиотеки типов (как и перечисления языка C), могут начинаться не с нуля, а порядковые номера, присваиваемые элементам, могут не быть последовательными. Подобные вольности в объявлении перечислимых типов в Object Pascal не разрешены, поэтому перечисления здесь должны объявляться как константы.

Интерфейсы элементов управления

Далее в файле оболочки описывается исходный интерфейс элемента управления. Здесь вы найдете все свойства и методы элемента управления ActiveX. Свойства еще раз объявляются в разделе `dispinterface`, что дает возможность элементу управления применяться в качестве двойного интерфейса. События объявляются отдельно, в следующем разделе `dispinterface`. Чтобы использовать элементы управления ActiveX в своих приложениях, вам не нужно много знать об интерфейсах, поэтому здесь мы не будем слишком углубляться в детали. Более подробную информацию об интерфейсах вообще вы найдете в главе 23 второго тома, “COM-ориентированные технологии”, а об интерфейсах элементов управления ActiveX — в главе 25 второго тома, “Создание элементов управления ActiveX”.

Потомок класса `TOleControl`

Далее в файле модуля следует определение класса для оболочки элемента управления. По умолчанию имя оболочки элемента ActiveX имеет вид `TXX`, где *`X`* — это имя класса компонента элемента управления в библиотеке типов. Этот объект, подобно всем оболочкам элементов ActiveX, является производным от класса `TOleControl`. Это компонент, который позволяет выполнять определенную обработку дескриптора окна, и является потомком класса `TWinControl`. Класс `TOleControl` инкапсулирует всю сложность отображения функций элементов управления ActiveX на компоненты Delphi, что позволяет элементам ActiveX успешно работать в среде Delphi. Класс `TOleControl` является *абстрактным*, а это значит, что создать его экземпляр невозможно, но зато его очень удобно использовать в качестве отправной точки для создания других классов.

Методы

Первой в листинге 7.1 приведена процедура `InitControlData()`. Она объявлена в описании объекта `TOleControl` и должна переопределяться во всех его потомках. Помимо прочей информации, связанной с конкретным элементом управления, в этой процедуре устанавливается уникальный класс OLE и идентификационные номера событий. В частности, этот метод уведомляет объект `TOleControl` о таких важных деталях, как идентификатор класса, разнообразные флажки управления и номер лицензии (если данный элемент управления лицензирован). Этот метод расположен в защищенной (`protected`) части описания класса, поскольку для пользователей данного класса он бесполезен и представляет ценность только для самого класса.

Метод `InitControlInterface()` переопределен с целью инициализации закрытого поля интерфейса `FIntf` значением указателя на интерфейс `ICardsX` элемента управления.

Элемент управления ActiveX CardX предоставляет еще только один метод — AboutBox(). При его вызове открывается диалоговое окно About, содержащее описание назначения данного элемента управления, что является стандартной практикой. Этот метод вызывается посредством интерфейса vTable с помощью свойства ControlInterface, которое считывается из поля FIntf.

На заметку

Помимо вызовов интерфейса vTable, использующего свойство ControlInterface, методы Control можно активизировать через автоматизацию с помощью свойства OLEObject класса TOleControl. Но, как вы узнаете в главе 23 второго тома, "COM-ориентированные технологии", обычно более эффективно вызывать методы через интерфейс vTable, а не через автоматизацию.

Свойства

Возможно, вы уже заметили, что класс TCardX имеет две отдельные группы свойств. Для членов одной группы не заданы методы чтения или записи значений. Эта группа представляет собой стандартные свойства и события компонентов Delphi, унаследованные от классов-предков TWinControl и TComponent. Для всех членов другой группы заданы индексы, а также методы чтения (get) и записи (set). Эту группу составляют свойства элементов управления ActiveX, инкапсулированные классом TOleControl.

Специализированные методы get и set для инкапсулированных свойств служат своего рода связующим мостом, проложенным между свойствами элементов управления ActiveX и компонентами Object Pascal. Обратите внимание на методы чтения и записи, которые получают и устанавливают свойства для каждого заданного типа, например GetBoolProp(), SetBoolProp(), GetString(), SetStringProp() и т.д. И хотя методы get и set существуют для каждого типа свойства, все они действуют по сходному принципу. В следующем фрагменте программного кода демонстрируются общие методы get и set для класса TOleControl, которые и определяют работу данного свойства типа *X*:

```
function TOleControl.Get X Prop(Index: Integer): X;
var
  Temp: TVarData;
begin
  GetProperty(Index, Temp);
  Result := Temp.V X;
end;

procedure TOleControl.Set X Prop(Index: Integer; Value: X);
var
  Temp: TVarData;
begin
  Temp.VType := var X;
  Temp.V X := Value;
  SetProperty(Index, Temp);
end;
```

В этом программном фрагменте индекс свойства (обозначенный директивой index в свойствах компонента TCardsCtrl) неявно передается процедурам. Переменная типа *X* упаковывается в запись данных TVarData (запись, представляющая тип Variant), а затем эти пара-

метры передаются методу GetProperty() или SetProperty() класса TOleControl. Каждое свойство элемента управления ActiveX имеет уникальный индекс, который действует подобно идентификатору. С помощью этого индекса и переменной Temp типа TVarData методы GetProperty() и SetProperty() используют автоматизацию OLE для получения и установки значений свойств внутри элемента управления ActiveX.

Если вам приходилось сталкиваться с другими пакетами разработки, то вы уже знаете, что Delphi обеспечивает упрощенный доступ не только к свойствам конкретного элемента ActiveX, но также и к обычным свойствам и методам объекта TWinControl. Это позволяет использовать любой элемент управления ActiveX подобно другим управляющим элементам в Delphi и дает пользователям возможность применять объектно-ориентированные принципы для переопределения поведения элемента управления ActiveX путем создания настраиваемых потомков элементов ActiveX в среде Delphi.

Использование элементов управления ActiveX в приложениях

Встроив оболочку своего элемента управления ActiveX в библиотеку компонентов, вы можете считать, что бо́льшую часть дела уже сделали. После размещения элемента управления ActiveX в палитре компонентов работа с ним практически не отличается от работы с обычными компонентами Delphi. На рис. 7.4 показана среда Delphi с элементом TCardX, активизированным в окне конструктора форм. Обратите внимание на свойства элемента TCardX, перечисленные в окне инспектора объектов.

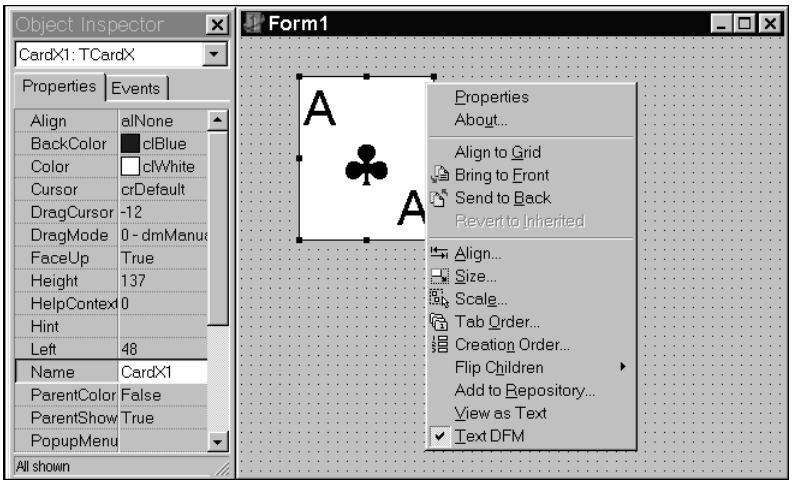


Рис. 7.4. Работа с элементом управления ActiveX в среде Delphi

Помимо возможности устанавливать свойства с помощью инспектора объектов, для некоторых элементов управления ActiveX также предусмотрено диалоговое окно Properties, которое открывается при выборе команды Properties в контекстном меню в дизайнере форм Delphi. Это контекстное меню, также показанное на рис. 7.4, открывается по щелчку правой кнопкой мыши на соответствующем элементе управления. На рис. 7.5 показано диалоговое окно Properties (Свойства) элемента ActiveX TCardX.

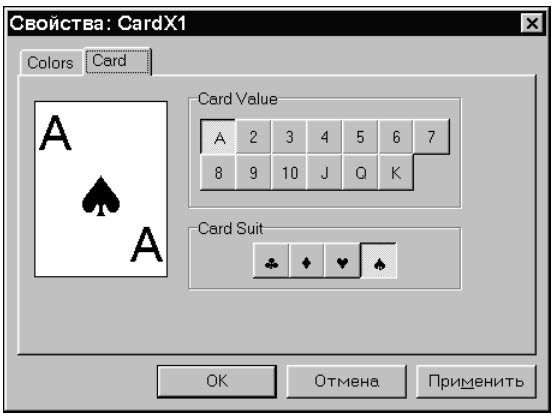


Рис. 7.5. Диалоговое окно Properties *элемента управления ActiveX* `TCardX`

Как вы, наверное, догадываетесь, данный элемент управления оснащен специфическими свойствами, позволяющими устанавливать масть карты, ее значение, цвет или изображение, нанесенное на ее оборотную сторону, а также стандартными свойствами, определяющими позицию, порядок обхода по нажатию клавиши <Tab> и т.п. У карты, показанной на рис. 7.5, свойство Value (Значение) установлено равным 1 (Туз), а свойство Suit (Масть) — 3 (Пики).

Распространение приложений, оснащенных элементами управления ActiveX

Когда приложение, оснащенное элементами управления ActiveX, будет готово к сдаче в эксплуатацию, передавая его элементы и соответствующие файлы заказчикам, обязательно учитывайте следующие замечания.

- Необходимо передать заказчику OCX- или DLL-файлы, содержащие элементы управления ActiveX, используемые в вашем приложении. OCX-файлы, являющиеся библиотеками динамической компоновки (DLL), не объединяются с загрузочным модулем приложения. Кроме того, прежде чем пользователь сможет работать с вашим приложением, каждый передаваемый элемент управления ActiveX должен быть зарегистрирован в системном реестре компьютера этого пользователя. Процесс регистрации элементов управления ActiveX рассматривается в следующем разделе.
- Работа некоторых элементов управления ActiveX невозможна без одной или нескольких внешних библиотек DLL или других файлов. Обратитесь к документации по элементам управления ActiveX, разработанным сторонними производителями, чтобы точно знать, нужно ли передавать заказчику какие-либо дополнительные файлы. Обратитесь также к главе 25 второго тома, “Создание элементов управления ActiveX”, за информацией о том, какие дополнительные файлы могут понадобиться для сопровождения элементов управления, созданных в среде Delphi.

- Многие элементы управления ActiveX поставляются с лицензионным файлом, который необходим в том случае, если вы собираетесь использовать данный элемент в процессе разработки. Этот файл выдает фирма — изготовитель элемента управления. Он призван защитить продаваемые производителем элементы управления от использования в целях разработки конечными пользователями, которым вы передаете эти элементы со своими приложениями. Вы не должны передавать своим заказчикам `LIC`-файлы вместе с приложением, если не хотите, чтобы пользователи этого приложения применяли лицензированные элементы управления в средствах разработки, и если у вас нет соответствующей лицензии на такое перераспределение.

Регистрация элементов управления ActiveX

Прежде чем какой-либо элемент управления ActiveX можно будет использовать в любой системе (включая тех же заказчиков или клиентов, которым будут переданы приложения), он должен быть зарегистрирован в системном реестре. Чаще всего это выполняется с помощью приложения `RegSvr32.exe`, которое входит в состав большинства версий Windows. В качестве альтернативного варианта можно использовать утилиту регистрации `TRegSvr.exe`, работающую в режиме командной строки (эта утилита находится в подкаталоге `bin` каталога Delphi). Вероятно, вы хотели бы иметь возможность регистрировать элементы управления более очевидным способом, чтобы ваше приложение казалось полностью интегрированным. К счастью, совсем не трудно интегрировать регистрацию (как и дерегистрацию) элементов ActiveX в ваше приложение. Фирма Inprise предоставляет исходный код для утилиты `TRegSvr` в качестве примера приложения, которое является великолепной демонстрацией того, как регистрировать серверы ActiveX и библиотеки типов.

BlackJack: пример приложения с компонентом ActiveX

Лучший способ продемонстрировать использование элемента управления ActiveX в приложении — показать, как можно написать какое-либо полезное приложение, включающее этот элемент. В нашем примере используется элемент управления `TCardX`, а каким образом можно лучше продемонстрировать использование элемента управления в виде игральной карты, если не в игре blackjack? Для убедительности предположим, что все программисты — крутые игроки, которым не нужно объяснять правила игры (разве вы не знали при покупке, что эта книга принадлежит к легкому жанру?), чтобы вы могли сконцентрироваться именно на программировании.

Нетрудно догадаться, что бо́льшая часть программного текста создаваемого приложения будет связана с логикой игры в blackjack. Вся программа приведена далее в этой главе, а пока мы обсудим лишь ту ее часть, которая касается непосредственно работы с элементами управления ActiveX. Данный проект называется `BJ`. Чтобы получить представление о том, для чего предназначен тот или иной программный фрагмент, взгляните на рис. 7.6, на котором показана игра DDG BJ в действии.

Колода карт

Прежде чем писать саму игру, необходимо сначала создать объект, который инкапсулирует колоду игральных карт. В отличие от настоящей колоды карт (в которой карты снимаются с верха перетасованной колоды), этот объект содержит неперетасованную колоду, а для выбора из нее случайной карты служат псевдослучайные числа. Такой подход возможен благо-

даря тому, что на каждой карте делается отметка о том, была ли она уже использована. Это значительно упрощает процедуру тасования карт, поскольку требуется всего лишь сбросить для каждой карты признак использования. В листинге 7.2 приведен текст программы для модуля `PlayCard.pas`, который содержит объект `TCardDeck`.

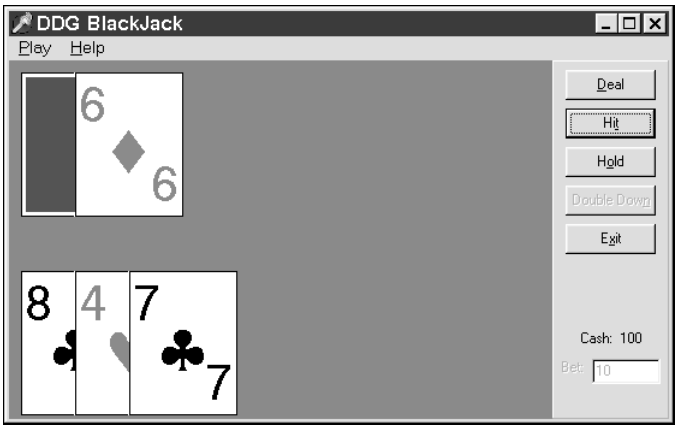


Рис. 7.6. Игра BJ

Листинг 7.2. Модуль `PlayCard.pas`

```
unit PlayCard;

interface

uses SysUtils, Cards;

type
  ECardError = class(Exception); // Общая исключительная ситуация для карт
  TPlayingCard = record          // представляет одну карту
    Face: TCardValue;            // Значение фигуры карты
    Suit: TCardSuit;             // Значение масти карты
  end;

  {  Массив из 52 карт, представляющих одну колоду }
  TCardArray = array[1..52] of TPlayingCard;

  { Объект, который представляет колоду из 52 УНИКАЛЬНЫХ карт. }
  { Более того, этот объект - перетасованная колода из 52 карт }
  { которая "помнит", сколько карт пользователь уже с нее снял.}
  TCardDeck = class
  private
    FCardArray: TCardArray;
    FTop: integer;
    procedure InitCards;
    function GetCount: integer;
```

```
  public
    property Count: integer read GetCount;
    constructor Create; virtual;
    procedure Shuffle;
    function Draw: TPlayingCard;
  end;

{ Метод GetCardValue возвращает числовое значение любой карты }
function GetCardValue(C: TPlayingCard): Integer;

implementation

function GetCardValue(C: TPlayingCard): Integer;
{ Возвращает числовое значение карты }
begin
  Result := Ord(C.Face) + 1;
  if Result > 10 then Result := 10;
end;

procedure TCardDeck.InitCards;
{ Инициализирует колоду путем назначения каждой }
{ карте уникальной комбинации значение/масть.   }
var
  i: integer;
  AFace: TCardValue;
  ASuit: TCardSuit;
begin
  AFace := cvAce;                        // Начало с туза
  ASuit := csClub;                       // начало с треф
  for i := 1 to 52 do                    // для каждой карты в колоде
  begin
    FCardArray[i].Face := AFace;         // Присвоение значения
    FCardArray[i].Suit := ASuit;         // Присвоение масти
    if (i mod 4 = 0) and (i <> 52) then  // Через каждые четыре карты
      inc(AFace);                        // увеличивается значение
    if ASuit <> High(TCardSuit) then     // Постоянное увеличение масти
      inc(ASuit)
    else
      ASuit := Low(TCardSuit);
  end;
end;

constructor TCardDeck.Create;
{ Конструктор для объекта TCardDeck. }
begin
  inherited Create;
  InitCards;
  Shuffle;
end;
```

```
function TCardDeck.GetCount: integer;
{ Возвращает число неиспользованных карт }
begin
  Result := 52 - FTop;
end;

procedure TCardDeck.Shuffle;
{ Перемешивание карт и установка номера верхней карты равным 0. }
var
  i: integer;
  RandCard: TPlayingCard;
  RandNum: integer;
begin
  for i := 1 to 52 do
  begin
    RandNum := Random(51) + 1;            // Получение случайного числа
    RandCard := FCardArray[RandNum];      // Обмен следующей карты на
    FCardArray[RandNum] := FCardArray[i]; // случайную карту в колоде
    FCardArray[i] := RandCard;
  end;
  FTop := 0;
end;

function TCardDeck.Draw: TPlayingCard;
{ Выбор следующей карты из колоды. }
begin
  inc(FTop);
  if FTop = 53 then
    raise ECardError.Create('Колода пуста');
  Result := FCardArray[FTop];
end;

initialization
  Randomize;   // Необходимо инициализировать генератор случайных чисел
end.
```

Создание игры

В игре blackjack обращение к объекту `TCardX` встречается главным образом в трех процедурах. Вызов одной из них, `Hit()`, происходит в случае, если игрок решает взять еще одну карту. Другая процедура, `DealerHit()`, вызывается, когда еще одну карту хочет взять тот, кто сдает карты (дилер). И, наконец, процедура `FreeCards()` вызывается, когда все отображенные на экране карты отдаются для подготовки к новой сдаче карт другим игроком.

Процедура `Hit()` работает с элементом управления ActiveX `TCardX` особым образом. Прежде всего, вместо использования элементов управления из палитры компонентов все они (элементы управления) в данной процедуре создаются динамически. Кроме того,

здесь нигде не применяется переменная экземпляра типа TCardX — вместо нее используются преимущества конструкции with..do для создания и использования этого объекта в одном шаге. Процедура Hit() представлена в следующем программном фрагменте:

```
procedure TMainForm.Hit;
{ Ход игрока }
begin
  CurCard := CardDeck.Draw;            // Взятие карты из колоды
  with TCardX.Create(Self) do          // Создание карты OCX
  begin
    Left := NextPlayerPos;             // Установка позиции
    Top := PYPos;
    Suit := Ord(CurCard.Suit);         // Установка масти
    Value := Ord(CurCard.Face);        // Установка значения
    Parent := Self;                    // Назначение родителя
    Inc(NextPlayerPos, Width div 2);   // Отслеживание позиции
    Update;                            // Отображение карты
  end;
  DblBtn.Enabled := False;
  if CurCard.Face = cvAce then PAceFlag := True; // Установка признака туза
  Inc(PlayerTotal, GetCardValue(CurCard));       // Накопление текущей суммы
  PlayLbl.Caption := IntToStr(PlayerTotal);      // Отображение суммы очков
  if PlayerTotal > 21 then                       // Банкротство
  begin
    ShowMessage('Busted!');
    ShowFirstCard;
    ShowWinner;
  end;
end;
```

В этой процедуре случайная карта, представленная переменной CurCard, берется из колоды, т.е. из объекта CardDeck, имеющего тип TCardDeck. Затем создается элемент управления ActiveX TCardX, для которого устанавливаются соответствующие значения свойств. Переменная NextPlayerPos предназначена для отслеживания позиции для следующей карты по оси X. Константа PYPos определяет позицию по оси Y для "руки" игрока. Свойствам Suit и Value присваиваются значения, которые соответствуют значениям Suit и Face случайной карты CurCard. Основная форма MainForm назначается родительской формой (Parent) этого элемента управления, а значение переменной NextPlayerPos увеличивается на половину ширины карты. После всего этого значение переменной PlayerTotal увеличивается на значение карты с целью подсчета суммы очков игрока.

Процедура DealerHit() действует аналогично процедуре Hit(). Вот ее код:

```
procedure TMainForm.DealerHit(CardVisible: Boolean);
{ Ход сдающего }
begin
  CurCard := CardDeck.Draw;     // Сдающий берет из колоды карту
  with TCardX.Create(Self) do   // Создание элемента управления ActiveX
  begin
    Left := NextDealerPos;      // Размещение карты в форме
    Top := DYPos;
    Suit := Ord(CurCard.Suit);  // Присвоение масти
```

```
    FaceUp := CardVisible;
    Value := Ord(CurCard.Face); // Присвоение значения карты
    Parent := Self;             // Присвоение родительской формы для OCX
    Inc(NextDealerPos, Width div 2); // Позиция для размещения следующей карты
    Update;     // Отображение карты
  end;
  if CurCard.Face = cvAce then DAceFlag := True; // Установка признака туза
  Inc(DealerTotal, GetCardValue(CurCard));  // Вычисление суммы очков
  DealLbl.Caption := IntToStr(DealerTotal); // Отображение суммы очков на экране
  if DealerTotal > 21 then                  // Банкротство сдающего
    ShowMessage('Dealer Busted!');
end;
```

Этому методу передается параметр `CardVisible` типа `Boolean`, который определяет, должна ли карта ложиться "лицом" вверх. Дело в том, что согласно правилам игры в blackjack первая карта дилера должна оставаться "лицом" вниз до тех пор, пока игрок решит не брать больше карт или не обанкротится. Для соблюдения этого правила при первом обращении к процедуре `DealerHit()` через переменную `CardVisible` передается значение `False`.

В обязанности процедуры `FreeCards()` входит удаление всех элементов управления `TCardX`, находящихся на главной форме. Поскольку в этом приложении для управления картами на экране не поддерживается массив или набор переменных типа `TCardX`, в данной процедуре выполняется просмотр элементов массива (образующих свойство формы `Control`) на предмет поиска элементов типа `TCardX`. Обнаруженный элемент управления искомого типа удаляется из памяти путем вызова метода `Free`. Освобождение элементов управления подобным образом происходит корректно благодаря *обратному* обходу массива. Если бы просмотр элементов массива происходил в прямом направлении, то существовала бы вероятность изменения порядка следования элементов управления в массиве, что привело бы к возникновению ошибок. Ниже приводится текст процедуры `FreeCards()`.

```
procedure TMainForm.FreeCards;
{ Удаление всех карт с экрана }
var
  i: integer;
begin
  for i := ControlCount - 1 downto 0 do  // В обратном направлении!
    if Controls[i] is TCardX then
      Controls[i].Free;
end;
```

На этом завершается обсуждение основных частей программы, благодаря которым происходят манипуляции с элементами управления ActiveX. Полный текст главного модуля этого приложения `Main.pas` приведен в листинге 7.3.

Листинг 7.3. Модуль Main.pas для проекта BJ

```
unit Main;

interface

uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  OleCtrls, Card_TLB, Cards, PlayCard, StdCtrls, ExtCtrls, Menus;

type
  TMainForm = class(TForm)
    Panel1: TPanel;
    MainMenu1: TMainMenu;
    Play1: TMenuItem;
    Deal1: TMenuItem;
    Hit1: TMenuItem;
    Hold1: TMenuItem;
    DoubleDown1: TMenuItem;
    N1: TMenuItem;
    Close1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    Panel2: TPanel;
    Label3: TLabel;
    CashLabel: TLabel;
    BetLabel: TLabel;
    HitBtn: TButton;
    DealBtn: TButton;
    HoldBtn: TButton;
    ExitBtn: TButton;
    BetEdit: TEdit;
    DblBtn: TButton;
    CheatPanel: TPanel;
    DealLbl: TLabel;
    PlayLbl: TLabel;
    Label4: TLabel;
    Label6: TLabel;
    Cheat1: TMenuItem;
    N2: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure About1Click(Sender: TObject);
    procedure Cheat1Click(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
    procedure DblBtnClick(Sender: TObject);
    procedure DealBtnClick(Sender: TObject);
    procedure HitBtnClick(Sender: TObject);
    procedure HoldBtnClick(Sender: TObject);
  private
    CardDeck: TCardDeck;
    CurCard: TPlayingCard;
    NextPlayerPos: integer;
    NextDealerPos: integer;
    PlayerTotal: integer;
    DealerTotal: integer;
    PAceFlag: Boolean;
```

```
    DAceFlag: Boolean;
    PBJFlag: Boolean;
    DBJFlag: Boolean;
    DDFlag: Boolean;
    Procedure Deal;
    procedure DealerHit(CardVisible: Boolean);
    procedure DoubleDown;
    procedure EnableMoves(Enable: Boolean);
    procedure FreeCards;
    procedure Hit;
    procedure Hold;
    procedure ShowFirstCard;
    procedure ShowWinner;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses AboutU;

const
  PYPos = 175;        // Начальная позиция y для карт игрока
  DYPos = 10;         // Начальная позиция y для карт дилера

procedure TMainForm.FormCreate(Sender: TObject);
begin
  CardDeck := TCardDeck.Create;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  CardDeck.Free;
end;

procedure TMainForm.About1Click(Sender: TObject);
{ Создание и активизация информационного окна "About" }
begin
  with TAboutBox.Create(Self) do
    try
      ShowModal;
    finally
      Free;
    end;
end;

procedure TMainForm.Cheat1Click(Sender: TObject);
```

```
begin
  Cheat1.Checked := not Cheat1.Checked;
  CheatPanel.Visible := Cheat1.Checked;
end;

procedure TMainForm.ExitBtnClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.DblBtnClick(Sender: TObject);
begin
  DoubleDown;
end;

procedure TMainForm.DealBtnClick(Sender: TObject);
begin
  Deal;
end;

procedure TMainForm.HitBtnClick(Sender: TObject);
begin
  Hit;
end;

procedure TMainForm.HoldBtnClick(Sender: TObject);
begin
  Hold;
end;

procedure TMainForm.Deal;
{ Новая партия для дилера и игрока }
begin
  FreeCards;                        // Удаление всех карт с экрана
  BetEdit.Enabled := False;         // Запрещение изменений ставок
  BetLabel.Enabled := False;        // Запрещение информации о ставках
  if CardDeck.Count < 11 then       // Перетасовка колоды, если меньше 11 карт
  begin
    Panel1.Caption := 'Reshuffling and dealing...';
    CardDeck.Shuffle;
  end
  else
    Panel1.Caption := 'Dealing...';
  Panel1.Show;                      // Отображение панели "dealing"
  Panel1.Update;                    // Гарантия ее видимости
  NextPlayerPos := 10;              // Установка позиции карт по горизонтали
  NextDealerPos := 10;
  PlayerTotal := 0;                 // Обнуление сумм очков
  DealerTotal := 0;
  PAceFlag := False;                // Сброс признаков
```

```
  DAceFlag := False;
  PBJFlag := False;
  DBJFlag := False;
  DDFlag := False;
  Hit;                              // Ход игрока
  DealerHit(False);                 // Ход дилера
  Hit;                              // Ход игрока
  DealerHit(True);                  // Ход дилера
  Panel1.Hide;                      // Сокрытие панели
  if (PlayerTotal = 11) and PAceFlag then
    PBJFlag := True;                // Проверка на blackjack у игрока
  if (DealerTotal = 11) and DAceFlag then
    DBJFlag := True;                // Проверка на blackjack у дилера
  if PBJFlag or DBJFlag then        // Если blackjack получился
  begin
    ShowFirstCard;                  // Открывается карта дилера
    ShowMessage('Blackjack!');
    ShowWinner;                     // Определение победителя
  end
  else
    EnableMoves(True); // Разрешение хода без открывания двух карт кряду
end;

procedure TMainForm.DealerHit(CardVisible: Boolean);
{ Ход дилера }
begin
  CurCard := CardDeck.Draw;     // Дилер берет карту из колоды
  with TCardX.Create(Self) do   // Создание элемента управления ActiveX
  begin
    Left := NextDealerPos;      // Размещение карты на форме
    Top := DYPos;
    Suit := Ord(CurCard.Suit);  // Присвоение масти
    FaceUp := CardVisible;
    Value := Ord(CurCard.Face); // Присвоение значения карты
    Parent := Self;   // Назначение родительской формы для элемента ActiveX
    Inc(NextDealerPos, Width div 2); // Вычисление позиции следующей карты
    Update;                     // Отображение карты
  end;
if CurCard.Face = cvAce then DAceFlag := True; // Установка признака туза
  Inc(DealerTotal, GetCardValue(CurCard));  // Вычисление текущей суммы очков
  DealLbl.Caption := IntToStr(DealerTotal);    // Отображение суммы очков
  if DealerTotal > 21 then                     // Банкротство дилера
    ShowMessage('Dealer Busted!');
end;

procedure TMainForm.DoubleDown;
{ Вызывается при взятии двух неоткрытых карт }
begin
  DDFlag := True; // Установка признака "слепой игры" для корректировки ставок
  Hit;      // Взятие одной карты
```

```
  Hold;     // Разрешение дилеру взять свои карты
end;

procedure TMainForm.EnableMoves(Enable: Boolean);
{ Разрешает/запрещает действия кнопок/элементов меню }
begin
  HitBtn.Enabled := Enable;              // Кнопка хода
  HoldBtn.Enabled := Enable;             // Кнопка блокировки
  DblBtn.Enabled := Enable;              // Кнопка двойного хода
  Hit1.Enabled := Enable;                // Элемент меню хода
  Hold1.Enabled := Enable;               // Элемент меню блокировки
  DoubleDown1.Enabled := Enable;         // Элемент меню двойного хода
end;

procedure TMainForm.FreeCards;
{ Освобождение всех карт на экране }
var
  i: integer;
begin
  for i := ControlCount - 1 downto 0 do  // В обратном направлении!
    if Controls[i] is TCardX then
      Controls[i].Free;
end;

procedure TMainForm.Hit;
{ Ход игрока }
begin
  CurCard := CardDeck.Draw;          // Взятие карты
  with TCardX.Create(Self) do // Создание карты, т.е. элемента управления ActiveX
  begin
    Left := NextPlayerPos;           // Установка позиции
    Top := PYPos;
    Suit := Ord(CurCard.Suit);       // Установка масти
    Value := Ord(CurCard.Face);      // Установка значения
    Parent := Self;                  // Назначение родительской формы
    Inc(NextPlayerPos, Width div 2); // Позиция следующей карты
    Update;                          // Отображение карты
  end;
  DblBtn.Enabled := False;           // Запрещение двойного хода "вслепую"
  if CurCard.Face = cvAce then PAceFlag := True;  // Установка признака туза
  Inc(PlayerTotal, GetCardValue(CurCard));  // Накопление текущей суммы очков
  PlayLbl.Caption := IntToStr(PlayerTotal); // Отображение суммы очков
  if PlayerTotal > 21 then                  // Банкротство
  begin
    ShowMessage('Busted!');
    ShowFirstCard;
    ShowWinner;
  end;
end;
```

```
procedure TMainForm.Hold;
{ Игрок прекращает брать из колоды карты. С помощью этой процедуры
  ход передается дилеру. }
begin
  EnableMoves(False);
  ShowFirstCard;                  // Отображение карты дилера,
  if PlayerTotal <= 21 then       // если у игрока не перебор...
  begin
    if DAceFlag then              // если у дилера есть туз...
    begin
      { Дилер должен сделать ход }
      while (DealerTotal <= 7) or ((DealerTotal >= 11) and
        (DealerTotal < 17)) do
        DealerHit(True);
    end
    else
      // Если выпал не туз, ходить, пока не накопится 17 очков
      while DealerTotal < 17 do DealerHit(True);
  end;
  ShowWinner;                     // Определение победителя
end;

procedure TMainForm.ShowFirstCard;
var
  i: integer;
begin
  // Отобразить все карты
  for i := 0 to ControlCount - 1 do
    if Controls[i] is TCardX then
    begin
      TCardX(Controls[i]).FaceUp := True;
      TCardX(Controls[i]).Update;
    end;
end;

procedure TMainForm.ShowWinner;
{ Определение стороны-победителя }
var
  S: string;
begin
  if DAceFlag then                 // Если у дилера есть туз...
  begin
    if DealerTotal + 10 <= 21 then  // Вычисление наилучшего хода
      inc(DealerTotal, 10);
  end;
  if PACeFlag then                // Если есть туз у игрока...
  begin
    if PlayerTotal + 10 <= 21 then // Вычисление наилучшего хода
      inc(PlayerTotal, 10);
  end;
```

```
  if DealerTotal > 21 then         // При переборе сброс счета в ноль
    DealerTotal := 0;
  if PlayerTotal > 21 then
    PlayerTotal := 0;
  if PlayerTotal > DealerTotal then // Если игрок выигрывает...
  begin
    S := 'You win!';
    if DDFlag then                 // Плата 2:1 при двойном ходе "вслепую"
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) +
        StrToInt(BetEdit.Text) * 2)
    else                           // Обычно плата 1:1
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) +
        StrToInt(BetEdit.Text));
    if PBJFlag then                // Плата 1.5:1 при выпадании blackjack
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) +
        StrToInt(BetEdit.Text) div 2)
  end
  else if DealerTotal > PlayerTotal then    // Если выигрывает дилер...
  begin
    S := 'Dealer wins!';
    if DDFlag then           // Потеря двойной ставки при двойном ходе "вслепую"
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) -
        StrToInt(BetEdit.Text) * 2)
    else                                // Обычная потеря
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) -
        StrToInt(BetEdit.Text));
  end
  else
    S := 'Push!';                       // Никто не выигрывает
  if MessageDlg(S + #13#10'Do you want to play again with the same bet?',
                                       // Вы будете платить по той же ставке?
    mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    Deal;
  BetEdit.Enabled := True;          // Разрешение изменить ставку
  BetLabel.Enabled := True;
end;

end.
```

Активизация метода элемента управления ActiveX

В листинге 7.3, как вы могли заметить, главная форма содержит метод, который создает и отображает диалоговое окно About, показанное на рис. 7.7.

В этом диалоговом окне предусмотрена кнопка, по щелчку на которой отображается окно About элемента управления ActiveX `Cards`, что осуществляется с помощью вызова метода `AboutBox()`. Вид этого окна показан на рис. 7.8.

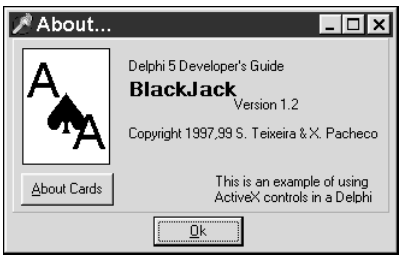


Рис. 7.7. Диалоговое окно About *игры BJ*

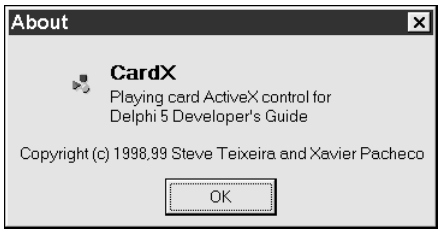


Рис. 7.8. Диалоговое окно About *элемента управления ActiveX* `Cards`

Вот как выглядит программный код, выполняющий эту задачу (забегая вперед, отметим, что здесь использован тот же метод, который применяется при создании обращений интерфейса `vTable` к серверам автоматизации OLE (читайте главу 23 второго тома, "COM-ориентированные технологии")):

```
procedure TAboutBox.CardBtnClick(Sender: TObject);
begin
  Card.AboutBox;
end;
```

Резюме

Прочитав эту главу, вы получили представление обо всех важных аспектах использования элементов управления ActiveX в среде Delphi 5. Мы обсудили интеграцию элемента ActiveX в Delphi и рассмотрели, как работает оболочка элемента управления ActiveX, написанная на языке Object Pascal. Вы узнали, как передавать своим пользователям приложения с элементами ActiveX, как зарегистрировать элемент ActiveX, а также о том, как можно внедрить элементы ActiveX в любое создаваемое приложение. Широкий рынок элементов управления ActiveX позволяет существенно увеличить продуктивность приложений, однако, учитывая некоторые присущие им принципиальные недостатки, не следует забывать и о собственных компонентах библиотеки VCL.

Профессиональное программирование

ЧАСТЬ II

Глава

8

GDI, шрифты и графика

В предыдущих главах нам уже приходилось работать со свойством, именуемым `Canvas` (в переводе с английского означает "холст" или "канва"). Это название вполне оправдывает себя, поскольку любое окно можно сравнить с чистым холстом художника, на котором изображаются различные объекты Windows. Каждая кнопка (окно или курсор) — это не что иное, как совокупность пикселей, в которых цвета подобраны так, чтобы придать ей определенный внешний вид. Вполне логично представлять себе каждое окно в виде отдельной поверхности, на которой нарисованы те или иные компоненты. Следуя этой аналогии, закономерно представить себя художником. Но чтобы написать картину, художнику необходимы различные инструменты. Ему не обойтись без палитры с красками, различных кистей и других рисовальных принадлежностей. И, наконец, он должен владеть специфическими художественными приемами и методами. Все эти средства и технологии программисту предоставляет интерфейс системы Win32. С помощью предлагаемых им инструментов и методов можно нарисовать самые разнообразные объекты, с которыми будет взаимодействовать пользователь. Доступ к этим инструментам осуществляется через графический интерфейс устройств, известный как GDI (Graphics Device Interface).

Интерфейс GDI используется в системе Win32 для рисования или раскраски изображений, которые пользователь видит на экране компьютера. До появления Delphi, в традиционном программировании для среды Windows, программисты работали непосредственно с функциями и инструментами GDI. В настоящее время объект `TCanvas` инкапсулирует и упрощает использование этих функций, инструментов и методов. В данной главе вы узнаете, как использовать объект `TCanvas` для выполнения полезных графический операций. Кроме того, мы покажем, как создавать сложные графические проекты с использованием Delphi 5 и интерфейса Win32 GDI. Для этой цели будут разработаны проекты программы рисования и программы анимации.

Представление рисунков в Delphi: класс `TImage`

Компонент `TImage` доступен в палитре компонентов Delphi 5 и представляет графическое изображение, которое может быть отображено в любом месте формы. С помощью компонента `TImage` можно загрузить и отобразить на экране любой растровый файл (`.bmp`), 16-разрядный метафайл Windows (`.wmf`), 32-разрядный метафайл расширенного формата (`.emf`), файл пиктограммы (`.ico`), файл формата JPEG (`.jpg`, `jpeg`) либо файлы других форматов, поддерживаемых надстройками класса `TGraphic`. Реально графические данные сохраняются в свойстве `Picture` объекта `TImage`, которое имеет тип `TPicture`.

Графические изображения: растры, метафайлы и пиктограммы

Растры

Растровые файлы (bitmap) в системе Win32 представляют собой двоичную информацию, организованную в виде набора битов, узор из которых и создает графический образ. В частности, в этих битах хранятся элементы информации о цвете, называемые *пикселями*. Существует два типа растров: зависимые от устройств (device-dependent bitmaps — DDB) и независимые от них (device-independent bitmaps — DIB).

Как программисту для среды Win32, вам, вероятно, вряд ли придется иметь дело с DDB-растрами, поскольку этот формат оставлен исключительно ради обратной совместимости. Такие растровые файлы зависят от типа конкретного устройства, на котором они были созданы. При сохранении данных в этом формате не запоминается ни информация, связанная с используемой цветовой палитрой, ни информация, связанная с разрешением.

Независимые от устройств растровые файлы (DIB) хранят всю необходимую информацию, позволяющую отобразить графические образы на любом устройстве без внесения существенных изменений в их внешний вид.
В памяти оба формата (как DDB, так и DIB) представлены практически одинаковыми структурами. Ключевое различие состоит в том, что в формате DDB используется палитра, предоставляемая системой, в то время как файл формата DIB содержит собственную палитру. Файл DDB — это, попросту, неструктурированное хранилище данных, содержимое которого обрабатывается программами видеодрайверов и специальными аппаратными видеосредствами. Файл DIB содержит данные в стандартизированных пиксельных форматах, обрабатываемые общими функциями GDI и хранящиеся в общей глобальной памяти. Некоторые видеоплаты используют пиксельные форматы DIB в качестве собственных хранилищ — тогда формат DDB эквивалентен формату DIB. Но в общем случае формат DIB предоставляет большую гибкость и простоту, иногда за счет незначительных потерь в производительности. Формат DDB всегда работает быстрее, но менее удобен.

Метафайлы
В отличие от растров, *метафайлы* представляют векторно-ориентированные графические изображения. Это файлы, в которых хранится набор функций GDI, что дает возможность сохранять обращения функций GDI к диску, чтобы можно было позже повторно отобразить данный графический образ. Это также позволяет использовать функции рисования совместно с другими программами, освобождая от необходимости вызова специальных функций GDI в каждой программе. Еще одним преимуществом метафайлов является возможность произвольного изменения размеров, что никоим образом не влияет на гладкость линий и дуг — тогда как для растровых изображений это не характерно. Это явилось одной из причин того, что при выполнении заданий печати механизм печати системы Win32 опирается на использование метафайлов расширенного формата.
Существует два формата метафайлов: стандартный, обычно использующий файлы с расширением `.wmf`, и расширенный, использующий `.emf`-файлы.
Стандартные метафайлы достались в наследство от системы Win16. Поскольку расширенные метафайлы более устойчивы к ошибкам и обладают большей точностью, при создании метафайлов для собственных приложений лучше использовать формат EMF. Но если вы собираетесь экспортировать свои метафайлы в более старые программы, не способные работать с расширенным форматом, используйте 16-разрядный формат WMF. При этом, однако, следует иметь в виду, что "опустившись" до 16-разрядного формата WMF, вы теряете некоторые GDI-примитивы, которые в формате EMF поддерживаются, а в формате WMF — нет. Классу `TMetaFile` Delphi 5 известно о существовании двух типов метафайлов.

Пиктограммы
Это ресурсы системы Win32, которые обычно хранятся в файле с расширением `.ico`. Кроме того, они могут располагаться и в файлах ресурсов (с расширением `.res`). В Windows существует два типичных размера пиктограмм: крупные (32×32 пикселя) и мелкие (16×16 пикселей). Во всех приложениях Windows используются значки обоих размеров. Мелкие значки отображаются в верхнем левом углу главного окна приложения, а также в элементах управления Windows 95, представляющих списки. Для инкапсуляции этого элемента управления в Delphi служит компонент `TListView`, который располагается во вкладке Win32 палитры компонентов.
Пиктограммы составляются из двух растров. Один из них, обычно называемый *образом* (image), представляет собой реальное изображение, которое появляется на экране. Другой же, именуемый *маской* (mask), позволяет добиться прозрачности при отображении пиктограммы. Пиктограммы используются в различных целях. Например, такие значки размещаются на панели задач приложения и в окнах сообщений (в этом случае для привлечения внимания в них также помещаются значки с изображением вопросительного и восклицательного знака или стоп-сигнала).

Класс `TPicture` представляет собой контейнерный класс для инкапсуляции абстрактного класса `TGraphic`. Название *контейнерный класс* означает, что класс `TPicture` может хранить ссылку на объекты `TBitmap`, `TMetaFile`, `TIcon` или другие типы классов, производных от

класса Tgraphic, и отображать их, не выясняя конкретно, с чем именно он имеет дело в данный момент. Для загрузки файлов изображений в компонент TImage используйте свойства и методы класса TImage.Picture, как показано, например, в следующей инструкции:

```
MyImage.Picture.LoadFromFile('FileName.bmp');
```

Аналогичную инструкцию можно использовать и для загрузки файлов пиктограмм или метафайлов. Например, с помощью следующего кода выполняется загрузка метафайла Win32:

```
MyImage.Picture.LoadFromFile('FileName.emf');
```

А эта строка поможет загрузить файл пиктограммы Win32:

```
MyImage.Picture.LoadFromFile('FileName.ico');
```

В Delphi 5 класс TPicture приобрел новую возможность загружать изображения в формате JPEG. Для этого используется тот же метод, что и при загрузке растровых изображений:

```
MyImage.Picture.LoadFromFile('FileName.jpeg');
```

Сохранение изображений

Для сохранения любого изображения используйте метод SaveToFile():

```
MyImage.Picture.SaveToFile('FileName.bmp');
```

Класс TBitmap инкапсулирует объекты растра и палитры системы Win32 и предоставляет методы для загрузки, сохранения, отображения и копирования растровых изображений. Данный класс также автоматически поддерживает реализацию палитры. Это значит, что с помощью класса TBitmap Delphi 5 утомительную задачу управления растрами можно в значительной степени упростить. Он позволяет сосредоточить внимание на использовании растра, не вникая во все детали реализации.

На заметку Класс TBitmap не является единственным объектом, в обязанности которого входит реализация палитры. Такие компоненты, как TImage, TMetaFile и любой другой потомок класса TGraphic, также реализуют палитры своих растров по запросу. Если вы построите компоненты, содержащие объекты TBitmap, которые могут иметь изображения, использующие 256 цветов, то вам придется переопределить метод своего компонента GetPalette() для возврата к цветовой палитре этого растра.

Для создания экземпляра класса TBitmap и загрузки в него, например, растрового файла используйте следующие команды:

```
MyBitmap := TBitMap.Create;
MyBitmap.LoadFromFile('MyBMP.BMP');
```

На заметку Другой метод загрузки растров в приложение состоит в их загрузке из файла ресурсов. Речь об этом пойдет далее.

Для копирования одного растра в другой используется метод TBitmap.Assign(), как показано в данном примере:

```
Bitmap1.Assign(Bitmap2);
```

Кроме того, с помощью метода CopyRect() можно скопировать часть растрового изображения из одного экземпляра объекта TBitmap в другой или даже на канву формы:

```
var
  R1: TRect;
begin
  with R1 do
  begin
    Top := 0;
    Left := 0;
    Right := BitMap2.Width div 2;
    Bottom := BitMap2.Height div 2;
  end;
  Bitmap1.Canvas.CopyRect(ClientRect, BitMap2.Canvas, R1);
end;
```

В приведенном выше фрагменте программного кода сначала вычисляются соответствующие значения в записи TRect, а затем метод TCanvas.CopyRect() используется для копирования части растра. Запись TRect определяется следующим образом:

```
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
  end;
```

Этот метод используется в программе рисования, обсуждаемой ниже в этой главе. Метод CopyRect() автоматически растягивает скопированную порцию канвы, чтобы она заполнила всю площадь объекта назначения.

Следует учитывать существенное различие в расходовании ресурсов при копировании растров в двух предыдущих примерах. При использовании метода CopyRect() потребление памяти удваивается за счет существования в ней двух отдельных копий изображения. В то же время метод Assign() не нуждается в лишних затратах, поскольку два растровых объекта будут совместно использовать ссылку на один и тот же графический образ в памяти. Если вам понадобится модифицировать один из растровых объектов, подпрограммы библиотеки VCL автоматически выполнят клонирование этого изображения.

Существует еще один метод, позволяющий копировать целый растр на канву формы со сжатием или расширением изображения так, чтобы оно помещалось и полностью заполняло пространство внутри границ канвы. Речь идет о методе StretchDraw():

```
Canvas.StretchDraw(R1, MyBitmap);
```

Подробнее методы класса TCanvas рассматриваются ниже в этой главе.

Использование свойств класса TCanvas

Классы высокого уровня, производные от классов TForm и TGraphicControl, обладают свойством Canvas. Этот объект — канву — можно рассматривать как поверхность, на которой рисуются все компоненты любой формы. В качестве инструментов рисования объект Canvas использует перья, кисти и шрифты.

Использование перьев

В этом разделе мы сначала обсудим, как использовать свойства объекта TPen, а затем рассмотрим пример проекта, в котором описываемые свойства будут реально использоваться.

С помощью перьев на канве рисуются линии. Доступ к перьям осуществляется через свойство Canvas.Pen. Для изменения способа рисования линий следует модифицировать свойства объекта пера: Color, Width, Style и Mode.

Свойство Color определяет цвет пера. В Delphi 5 предусмотрены предопределенные цветовые константы, соответствующие многим общеупотребительным цветам. Например, константы clRed и clYellow соответствуют красному и желтому цветам. Кроме того, в Delphi 5 определены константы для представления системных цветов экранных элементов системы Win32. Например, константы clActiveCaption и clHighlightText соответствуют цветам активных заголовков и выделенного текста в данной системе Win32. Приведенная ниже программная строка назначает перу канвы синий цвет.

```
Canvas.Pen.color := clblue;
```

А с помощью следующей программной строки свойству Pen объекта Canvas можно назначить случайный цвет:

```
Pen.Color := TColor(RGB(Random(255), Random(255), Random(255)));
```

Функция RGB() и тип TColor

В интерфейсе системы Win32 цвета представлены в виде целых значений, в которых младших три байта означают уровни яркости соответственно красного, зеленого и синего цветов. Комбинация этих трех значений образует цвет Win32. Функции RGB(R,G,B) передается три параметра, указывающих уровень интенсивности красного, зеленого и синего цветов. Эта функция возвращает цвет Win32 как целое значение. В Delphi такое представление цветов поддерживается типом TColor. Для каждого уровня интенсивности существует 255 возможных значений, следовательно, функция RGB() может возвратить приблизительно 16 млн цветов. Например, вызов RGB(0,0,0) возвращает значение для черного цвета, в то время как вызов RGB(255,255,255) — для белого. Вызовы RGB(255,0,0), RGB(0,255,0) и RGB(0,0,255) возвращают цветовые значения для красного, зеленого и синего цветов соответственно. Варьируя значения, передаваемые функции RGB(), можно получить любой цвет из доступного цветового спектра.

Тип данных TColor является специфическим для библиотеки VCL и ссылается на константы, определенные в модуле Graphics.pas. Эти константы соответствуют либо ближайшему соответствующему цвету в системной палитре, либо определенному цвету, устанавливаемому в окне панели управления Windows. Например, константа clBlue соответствует синему цвету, тогда как константа clBtnFace — цвету, установленному в системе для лицевых поверхностей кнопок. Кроме трех байтов, определяющих

собственно цвет, в данных типа TColor старший байт используется для обозначения того, что именно представляет собой этот цвет. Если значение этого байта равно $00, данный цвет представляет ближайший соответствующий цвет в системной палитре. Значение $01 представляет ближайшее соответствие в используемой в данный момент палитре. Наконец, значение старшего байта $02 определяет ближайший цвет в логической палитре текущего контекста устройства. Дополнительную информацию можно найти в статье "TColor type" интерактивной справочной системы Delphi.

Для преобразования системных цветов Win32 (например, clWindow) в действующие RGB-цвета используйте функцию ColorToRGB(). Эта функция описана в интерактивной справочной системе Delphi 5.

С помощью пера можно также рисовать линии, отличающиеся друг от друга стилем, который определяется свойством Style. В табл. 8.1 показаны различные стили, которые можно устанавливать для свойства Pen.Style:

Таблица 8.1. Стили пера

Стиль	Что рисуется
psClear	Невидимая линия
psDash	Линия, состоящая из штрихов
psDashDot	Линия, состоящая из чередующихся штрихов и точек
psDashDotDot	Линия, состоящая из сочетаний штрих-точка-точка
psDot	Линия, состоящая из точек
psInsideFrame	Линия внутри рамки замкнутой формы, определяющей ограничивающий прямоугольник
psSolid	Сплошная линия

Следующая строка демонстрирует, как изменять стиль рисования пером:

```
Canvas.Pen.Style := psDashDot;
```

На рис. 8.1 показано, как выглядят различные стили пера при рисовании на канве формы. Одно замечание: цвет в промежутках штриховых линий зависит от цвета кисти. Если требуется провести черную штриховую линию по красной поверхности, следует установить свойство Canvas.Brush.Color равным константе clRed либо свойство Canvas.Brush.Style равным константе bsClear. Установка цвета пера и кисти позволяет достичь любого желаемого вида линий (например, провести на белой поверхности красную и синюю пунктирные линии).

Свойство Pen.Width позволяет указать ширину линии в пикселях, проводимую пером при рисовании. При установке большего значения перо рисует более толстую линию.

На заметку

Стиль штриховых линий применяется для перьев шириной, равной 1 пикселю. При установке ширины, равной 2, перо будет рисовать сплошную линию. Такое поведение унаследовано от 16-разрядного GDI, действия которого система Win32 эмулирует в целях совместимости. В Windows 95/98 не рисуются жирные пунктирные линии, однако в Windows NT/2000 это возможно, но при условии, что используется только расширенный набор средств GDI.

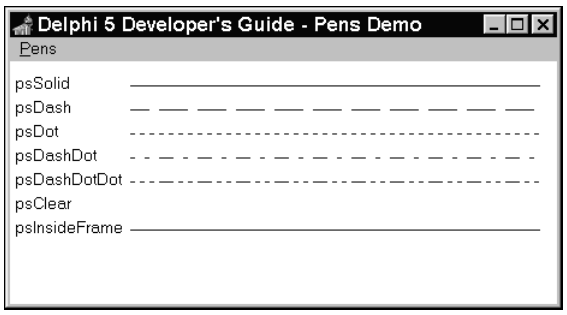


Рис. 8.1. Различные стили пера

Существует три фактора, которые определяют, как система Win32 рисует пиксели или линии на поверхности канвы: цвет пера, цвет поверхности или конечный цвет, а также побитовая операция, которую интерфейс системы Win32 выполняет для двуцветных значений. Эта операция называется *растровой* (ROP). Тип растровой операции задается с помощью свойства `Pen.Mode`, применяемого к данной канве. В системе Win32 определены 16 режимов, перечисленных в табл. 8.2.

Таблица 8.2. Режимы работы пера в системе Win32 с учетом исходного цвета Pen.Color (S) и конечного цвета (D)

Режим	Результирующий цвет пикселя	Логическая операция
`pmBlack`	Всегда черный	0
`pmWhite`	Всегда белый	1
`pmNOP`	Неизменяемый	D
`pmNOT`	Инверсия цвета D	Не D
`pmCopy`	Цвет S	S
`pmNotCopy`	Инверсия цвета S	Не S
`pmMergePenNot`	Объединение цвета S и инверсии D	S или не D
`pmMaskPenNot`	Объединение цветов, общих для S и инверсии D	S и не D
`pmMergeNotPen`	Сочетание цвета D и инверсии S	Не S или D
`pmMaskNotPen`	Объединение цветов, общих для D и инверсии S	Не S и D
`pmMerge`	Объединение цветов S и D	S или D
`pmNotMerge`	Инверсия результата операции `pmMerge` для цветов S и D	Не (S или D)
`pmMask`	Объединение цветов, общих для S и D	S и D
`pmNotMask`	Инверсия результата операции `pmMask` для цветов S и D	Не (S и D)
`pmXor`	Либо цвет S, либо D, но не оба	S XOR D (исключающее ИЛИ)
`pmNotXor`	Инверсия результата операции `pmXor` для цветов S и D	Не (S XOR D)

По умолчанию свойство Pen.mode установлено равным значению pmCopy. Это говорит о том, что перо рисует цветом, заданным его свойством Color. Предположим, вы хотите нарисовать черные линии на белом фоне. Если какая-нибудь линия пересечет нарисованную ранее, то в месте пересечения должен остаться белый цвет, а не черный.

Один из возможных путей решения этой задачи состоит в проверке цвета области, в которой вы собираетесь рисовать. Если она белая, установите свойство pen.Color равным значению, соответствующему черному цвету, а если черная, то свойство pen.Color должно задавать белый цвет. Несмотря на то что этот способ работает, он слишком громоздкий и медленный. Было бы гораздо лучше установить свойство pen.Color равным константе clBlack, а свойство Pen.Mode — константе pmNot. Это приведет к тому, что рисование пером выльется в инверсию операции объединения цвета пера и поверхности. На рис. 8.2 показан результат этой операции при рисовании черным пером.

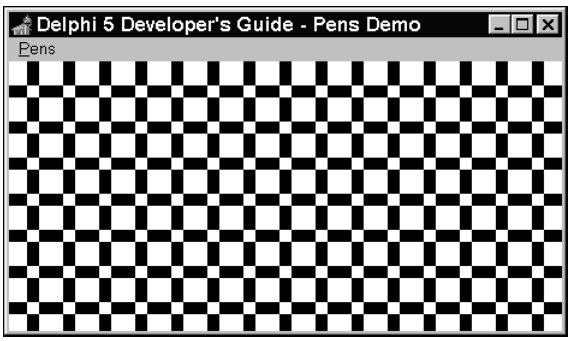


Рис. 8.2. Результат операции pmNot

В листинге 8.1 приведен пример программы, результат работы которой показан на рис. 8.1 и 8.2.

Листинг 8.1. Примеры операций с пером

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiPens: TMenuItem;
    mmiStyles: TMenuItem;
    mmiPenColors: TMenuItem;
    mmiPenMode: TMenuItem;
    procedure mmiStylesClick(Sender: TObject);
    procedure mmiPenColorsClick(Sender: TObject);
    procedure mmiPenModeClick(Sender: TObject);
  private
```

```
    { Закрытые объявления }
  public
    { Открытые объявления }
    procedure ClearCanvas;
    procedure SetPenDefaults;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.ClearCanvas;
var
  R: TRect;
begin
  // Очистка содержимого канвы
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    Canvas.FillRect(ClientRect);
  end;
end;

procedure TMainForm.SetPenDefaults;
begin
  with Canvas.Pen do
begin
    Width := 1;
    Mode := pmCopy;
    Style := psSolid;
    Color := clBlack;
  end;
end;

procedure TMainForm.mmiStylesClick(Sender: TObject);
var
  yPos: integer;
  PenStyle: TPenStyle;
begin
  ClearCanvas;     // Первая очистка содержимого канвы
  SetPenDefaults;
  // yPos представляет координату Y
  YPos := 20;
  with Canvas do
  begin
    for PenStyle := psSolid to psInsideFrame do
```

```
    begin
      Pen.Color := clBlue;
      Pen.Style := PenStyle;
      MoveTo(100, yPos);
      LineTo(ClientWidth, yPos);
      inc(yPos, 20);
    end;

    // Вывод названий для различных стилей пера
    TextOut(1, 10, ' psSolid ');
    TextOut(1, 30, ' psDash ');
    TextOut(1, 50, ' psDot ');
    TextOut(1, 70, ' psDashDot ');
    TextOut(1, 90, ' psDashDotDot ');
    TextOut(1, 110, ' psClear ');
    TextOut(1, 130, ' psInsideFrame ');
  end;
end;

procedure TMainForm.mmiPenColorsClick(Sender: TObject);
var
  i: integer;
begin
  ClearCanvas;    // Очистка содержимого канвы
  SetPenDefaults;
  with Canvas do
  begin
    for i := 1 to 100 do
    begin
      // Получить случайное цветовое значение и нарисовать линию этого цвета
      Pen.Color := RGB(Random(255),Random(255), Random(255));
      MoveTo(random(ClientWidth), Random(ClientHeight));
      LineTo(random(ClientWidth), Random(ClientHeight));
    end
  end;
end;
procedure TMainForm.mmiPenModeClick(Sender: TObject);
var
  x,y: integer;
begin
  ClearCanvas;  // Очистка содержимого канвы
  SetPenDefaults;
  y := 10;
  canvas.Pen.Width := 20;
  while y < ClientHeight do
  begin
    canvas.MoveTo(0, y);
    // Нарисовать линию и увеличить значение координаты Y
    canvas.LineTo(ClientWidth, y);
    inc(y, 30);
```

```
  end;
  x := 5;

  canvas.pen.Mode := pmNot;
  while x < ClientWidth do
  begin
    Canvas.MoveTo(x, 0);
    canvas.LineTo(x, ClientHeight);
    inc(x, 30);
  end;
end;

end.
```

В листинге 8.1 приведены три примера использования пера канвы. Две вспомогательные функции, `ClearCanvas()` и `SetPenDefaults()`, применяются для очистки содержимого канвы главной формы и установки свойств объекта `Canvas.Pen` равными их стандартным значениям, поскольку эти свойства модифицируются каждым из трех обработчиков событий.

Функция `ClearCanvas()` — это очень полезный инструмент удаления содержимого любого компонента, имеющего свойство `Canvas`. В функции `ClearCanvas()` используется кисть, наносящая сплошной белый цвет для удаления всего, что было ранее нанесено на канву. Метод `FillRect()` отвечает за окрашивание прямоугольной области, задаваемой переданной ему в качестве параметра структурой типа `TRect` с именем `ClientRect`.

Метод `mmiStylesClick()` демонстрирует, как выглядят различные стили пера при рисовании горизонтальных линий на канве формы (см. рис. 8.1). Процесс рисования линий на канве реализуется с помощью методов `TCanvas.MoveTo()` и `TCanvas.LineTo()`.

На примере метода `mmiPenColorsClick()` проиллюстрировано рисование линий перьями различных цветов. Здесь для получения случайного значения цвета, присваиваемого свойству `TPen.Color`, используется функция `RGB()`. Три значения, передаваемые функции `RGB()`, являются случайными числами в интервале от 0 до 255. Результат работы этого метода показан на рис. 8.3.

Наконец, метод `mmiPenModeClick()` иллюстрирует рисование линий с использованием различных режимов работы пера. В данном случае для выполнения описанных выше действий применяется режим `pmNot` (см. рис. 8.2).

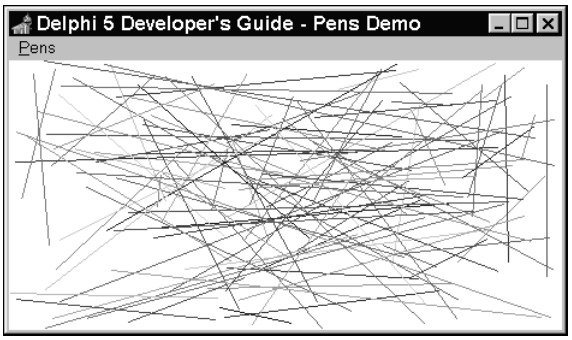


Рис. 8.3. Результат работы метода `mmiPenColorsClick()`

Использование свойства Pixels объекта TCanvas

Свойство TCanvas.Pixels представляет собой двухмерный массив, в котором каждый элемент является значением пикселя TColor на поверхности формы или области клиента. Левый верхний угол поверхности рисования формы определяется так:

```
Canvas.Pixels[0,0]
```

А правый нижний угол определяется так:

```
Canvas.Pixels[clientWidth, clientHeight];
```

Вряд ли вам когда-либо придется обращаться к отдельным пикселям вашей формы. Как правило, свойство Pixels не применяется по причине его медлительности, однако его используют такие функции GDI, как GetPixel() и SetPixel(), которые (по признанию фирмы Microsoft) недоработаны и неэффективны. Суть в том, что в обеих функциях используются 24-битовые значения формата RGB. Если контекст устройства не работает с подобным представлением цвета, то выполнение этих функций будет сопряжено с массовыми операциями преобразования данных RGB в формат пикселя контекста устройства. Для быстрых операций с пикселями лучше использовать свойство-массив TBitmap.ScanLine. Однако для считывания или установки всего одного-двух пикселей подойдет и свойство Pixels.

Использование кисти

В этом разделе описаны свойства объекта TBrush и в качестве примера предложен проект, в котором показано, как эти свойства можно использовать.

Использование свойств объекта TBrush

Если перо (объект TPen) позволяет рисовать на канве линии, то с помощью кисти (объекта TBrush) выполняется закрашивание областей и фигур, нарисованных на этой канве. При этом могут использоваться различные цвета, стили и узоры (орнаменты).

Объект TBrush обладает тремя важными свойствами, Color, Style и Bitmap, которые определяют, как кисть будет закрашивать поверхность канвы. Свойство Color определяет цвет кисти, Style — узор фона кисти, а Bitmap задает растр, который можно использовать для создания пользовательских орнаментов, служащих фоном кисти.

Свойство Style кисти может принимать одно из восьми допустимых значений: bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross или bsDiagCross. По умолчанию цвет кисти устанавливается равным константе clWhite, стиль — константе bsSolid, а растр не задается. В вашей власти изменять стандартные цвет и стиль работы кисти при выполнении закрашивания областей различными узорами. В следующем примере иллюстрируется использование каждого из свойств объекта TBrush.

Пример использования объекта TBrush

В листинге 8.2 представлен модуль проекта, который иллюстрирует использование рассмотренных выше свойств объекта TBrush.

Листинг 8.2. Пример работы с объектом `TBrush`

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms,
  Dialogs, Menus, ExtCtrls;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiBrushes: TMenuItem;
    mmiPatterns: TMenuItem;
    mmiBitmapPattern1: TMenuItem;
    mmiBitmapPattern2: TMenuItem;
    procedure mmiPatternsClick(Sender: TObject);
    procedure mmiBitmapPattern1Click(Sender: TObject);
    procedure mmiBitmapPattern2Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    FBitmap: TBitmap;
  public
    procedure ClearCanvas;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
procedure TMainForm.ClearCanvas;
var
  R: TRect;
begin
  // Очистка содержимого канвы
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    GetWindowRect(Handle, R);
    R.TopLeft := ScreenToClient(R.TopLeft);
    R.BottomRight := ScreenToClient(R.BottomRight);
    FillRect(R);
  end;
end;

procedure TMainForm.mmiPatternsClick(Sender: TObject);
```

```
begin
  ClearCanvas;
  with Canvas do
  begin
    // Вывод названий различных стилей кисти
    TextOut(120, 101, 'bsSolid');
    TextOut(10, 101, 'bsClear');
    TextOut(240, 101, 'bsCross');
    TextOut(10, 221, 'bsBDiagonal');
    TextOut(120, 221, 'bsFDiagonal');
    TextOut(240, 221, 'bsDiagCross');
    TextOut(10, 341, 'bsHorizontal');
    TextOut(120, 341, 'bsVertical');

    // Рисование прямоугольника с использованием различных стилей кисти

    Brush.Style := bsClear;
    Rectangle(10, 10, 100, 100);
    Brush.Color := clBlack;

    Brush.Style := bsSolid;
    Rectangle(120, 10, 220, 100);

    {  Демонстрация прозрачной кисти рисованием окрашенного прямоугольника,
       поверх которого будет нарисован другой прямоугольник. }

    Brush.Style := bsSolid;
    Brush.Color := clRed;
    Rectangle(230, 0, 330, 90);

    Brush.Style := bsCross;
    Brush.Color := clBlack;
    Rectangle(240, 10, 340, 100);
    Brush.Style := bsBDiagonal;
    Rectangle(10, 120, 100, 220);

    Brush.Style := bsFDiagonal;
    Rectangle(120, 120, 220, 220);

    Brush.Style := bsDiagCross;
    Rectangle(240, 120, 340, 220);

    Brush.Style := bsHorizontal;
    Rectangle(10, 240, 100, 340);

    Brush.Style := bsVertical;
    Rectangle(120, 240, 220, 340);

  end;
end;
```

```
procedure TMainForm.mmiBitmapPattern1Click(Sender: TObject);
begin
  ClearCanvas;
  // Загрузка растра с диска
  FBitMap.LoadFromFile('pattern.bmp');
  Canvas.Brush.Bitmap := FBitmap;
  try
    { Рисование прямоугольника, охватывающего всю область клиента,
      с использованием растрового узора кисти. }
    Canvas.Rectangle(0, 0, ClientWidth, ClientHeight);
  finally
    Canvas.Brush.Bitmap := nil;
  end;
end;
procedure TMainForm.mmiBitmapPattern2Click(Sender: TObject);
begin
  ClearCanvas;
  // Загрузка растра с диска
  FBitMap.LoadFromFile('pattern2.bmp');
  Canvas.Brush.Bitmap := FBitmap;
  try
    { Рисование прямоугольника, охватывающего всю область клиента,
      с использованием растрового узора кисти. }
    Canvas.Rectangle(0, 0, ClientWidth, ClientHeight);
  finally
    Canvas.Brush.Bitmap := nil;
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  FBitmap := TBitmap.Create;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FBitmap.Free;
end;

end.
```

Используемый здесь метод `ClearCanvas` является удобной подпрограммой для различных служебных модулей. Можно определить этот метод таким образом, чтобы ему передавались параметры `TCanvas` и `TRect`, указывающие объект, к которому применяется процедура очистки:

```
procedure ClearCanvas(ACanvas: TCanvas; ARect: TRect);
var
  // Очистка содержимого канвы
  with ACanvas do
  begin
```


```
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    FillRect(ARect);
  end;
end;
```

С помощью метода `mmiPatternsClick()` иллюстрируется рисование с применением к объекту `TBrush` различных узоров. Сначала выводятся названия стилей, а затем на канве формы рисуются прямоугольники, закрашенные всеми возможными узорами. На рис. 8.4 показан результат работы этого метода.

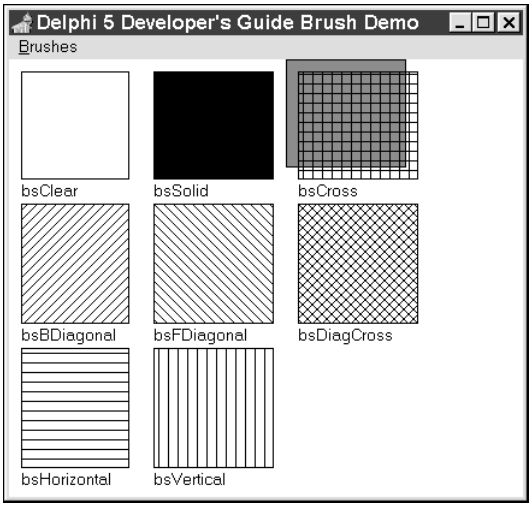


Рис. 8.4. Узоры, которые может поддерживать кисть

На примере методов `mmiBitmapPattern1Click()` и `mmiBitmapPattern2Click()` показано, как использовать в качестве инструмента кисти растровый узор. Свойство `TCanvas.Brush` содержит, в свою очередь, свойство `TBitmap`, которому можно назначить растровый узор. Этот узор будет использован для заполнения закрашиваемой кистью области вместо обычного узора, задаваемого свойством `TBrush.Style`. Однако при использовании этого способа заливки нужно следовать нескольким правилам. Во-первых, назначьте этому свойству действительный растровый объект. Во-вторых, по окончании операции обязательно присвойте значение `nil` свойству `Brush.Bitmap`, поскольку кисть не получает право собственности на растровый объект при назначении ей какого-нибудь растра. На рис. 8.5 и 8.6 показаны результаты работы методов `mmiBitmapPattern1Click()` и `mmiBitmapPattern2Click()` соответственно.

На заметку

Windows ограничивает растры, предназначенные для создания узоров кисти, размерами 8×8 пикселей, причем эти растры должны быть зависимыми от устройств. Windows 95 забракует растры, размер которых превышает 8×8 пикселей. Windows NT примет их в работу, но будет использовать только верхнюю левую часть, образующую квадрат 8×8 пикселей.

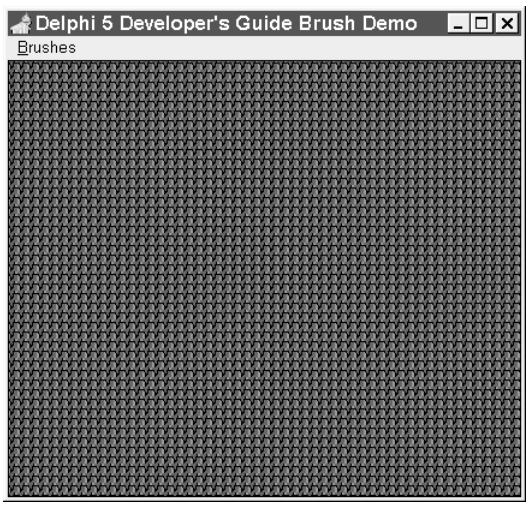


Рис. 8.5. Результат работы метода `mmiBitmapPattern1Click()`

Растровый узор, используемый для заливки некоторой области канвы, может быть применен не только к канве самой формы, но также и к любому компоненту, который содержит свойство `Canvas`. Достаточно получить доступ к методам и/или свойствам нужного компонента, а не формы. Например, для узорной заливки компонента `Image` воспользуйтесь следующим кодом:

```
Image1.Canvas.Brush.Bitmap := SomeBitmap;
  try
    Image1.Canvas.Rectangle(0, 0, Image1.Width, Image1.Height);
  finally
    Image1.Canvas.Brush.Bitmap := nil;
  end;
```

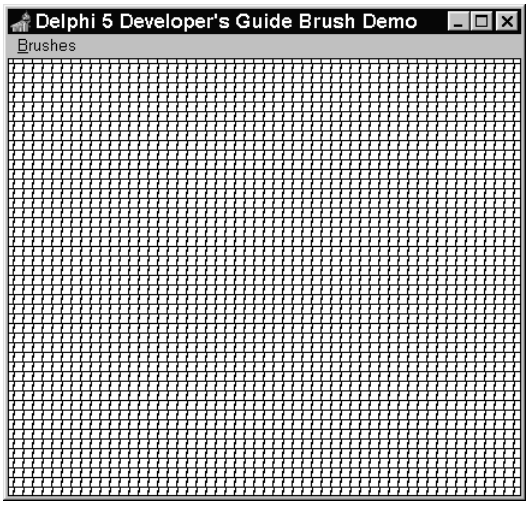


Рис. 8.6. Результат работы метода `mmiBitmapPattern2Click()`

Использование шрифтов

С помощью свойства Canvas.Font можно выводить на канву текст, используя любой из доступных в интерфейсе системы Win32 шрифтов. При этом существует возможность изменять внешний вид помещаемого на канву текста путем модификации такого свойства шрифта, как Color, Name, Size, Height или Style.

Свойству Font.Color можно присвоить любой определенный в Delphi 5 цвет. Например, с помощью следующей строки можно получить шрифт красного цвета:

```
Canvas.Font.Color := clRed;
```

Свойство Name предназначено для указания имени шрифта Windows. Так, используя следующую строку, в качестве шрифта канвы можно установить любой требуемый шрифт:

```
Canvas.Font.Name := 'Times New Roman';
```

Свойство Canvas.Font.Size задает размер шрифта в пунктах.

Свойство Canvas.Font.Style представляет собой множество, которое может состоять из одного или из произвольной комбинации стилей, приведенных в табл. 8.3.

Таблица 8.3. Стили шрифтов

Значение	Стиль
fsBold	Полужирный
fsItalic	Курсив
fsUnderline	Подчеркивание
fsStrikeOut	Перечеркнутый текст

Для объединения двух стилей используйте следующий синтаксис:

```
Canvas.Font.Style := [fsBold, fsItalic];
```

Для выбора определенного шрифта в системе Win32 и присвоения его свойству TMemo.Font можно использовать объект TFontDialog:

```
if FontDialog1.Execute then
  Memo1.Font.Assign(FontDialog1.Font);
```

Тот же результат будет достигнут, если присвоить шрифт, выбранный с помощью объекта TFontDialog, свойству объекта Canvas, отвечающему за шрифт:

```
Canvas.Font.Assign(FontDialog1.Font);
```

Кроме того, шрифту объекта Canvas можно присвоить отдельные атрибуты, принадлежащие шрифту, выбранному с помощью объекта TFontDialog:

```
Canvas.Font.Name := Font.Dialog1.Font.Name;
Canvas.Font.Size := Font.Dialog1.Font.Size;
```

Не забывайте при копировании экземпляров переменных типа TBitmap, TBrush, TIcon, TMetaFile, TPen или TPicture использовать метод Assign(). Рассмотрим следующую инструкцию:

```
MyBrush1 := MyBrush2
```

Хотя она и выглядит корректной, но выполняет прямое копирование указателя, в результате чего оба экземпляра будут указывать на один и тот же объект кисти, а это может привести к утечке памяти. При использовании же метода Assign() гарантируется, что предыдущие ресурсы будут освобождены.
Упомянутая выше опасность не возникает при выполнении присвоения свойств Tfont:

```
Form1.Font := Form2.Font
```

Эта инструкция вполне допустима, поскольку свойство TForm.Font является свойством, метод записи которого неявно вызывает функцию Assign() для копирования данных из соответствующего объекта шрифта. Но будьте осторожны: такую инструкцию допустимо применять при назначении *свойств* TFont, а не *переменных* типа TFont. Чтобы не допустить ошибку, лучше всегда явно использовать метод Assign().

На этом мы пока завершим рассмотрение возможностей канвы, касающихся работы со шрифтами. Дополнительную информацию о шрифтах вы найдете в конце главы.

Использование свойства CopyMode

Свойство TCanvas.CopyMode определяет, как изображение копируется с одной канвы на другую. Например, если свойство CopyMode имеет значение cmSrcCopy, это значит, что исходное изображение будет полностью скопировано на область назначения. Значение, задаваемое константой CmSrcInvert, требует сочетать пиксели изображений источника и приемника с учетом действия поразрядного оператора XOR (исключающее ИЛИ). Свойство CopyMode используется для получения различных эффектов при копировании одного растра на другой. Обычно замена стандартного значения свойства CopyMode, равного константе cmSrcCopy, каким-нибудь другим значением оказывается полезной при написании анимационных приложений. О том, как создается анимация, речь пойдет ниже в этой главе.

Чтобы понять, как используется свойство CopyMode, взгляните на рис. 8.7.

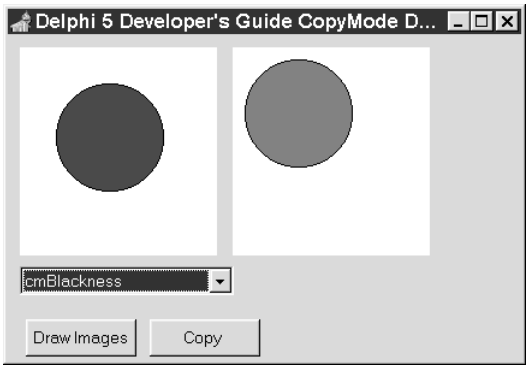


Рис. 8.7. Форма, содержащая два изображения, является хорошей иллюстрацией использования свойства CopyMode

На рис. 8.7 показана форма с двумя графическими изображениями, на которых нарисованы эллипсы. При выборе из списка (представленного объектом TComboBox) различных значений свойства CopyMode можно получать различные результаты копирования одного изображения поверх другого (копирование происходит по щелчку на кнопке Copy). На рис. 8.8 и 8.9 показано, какие эффекты получаются при копировании объекта imgFromImage в объект imgToImage с использованием констант режима копирования cmSrcAnd и cmSrcInvert.

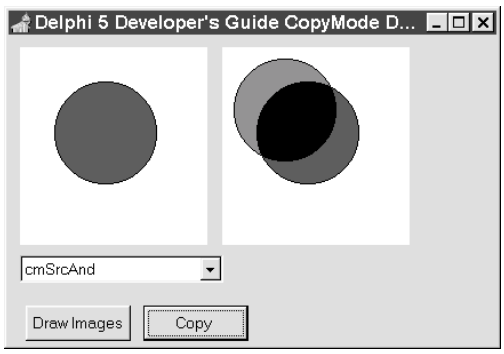


Рис. 8.8. Операция копирования с использованием значения `cmSrcAnd`

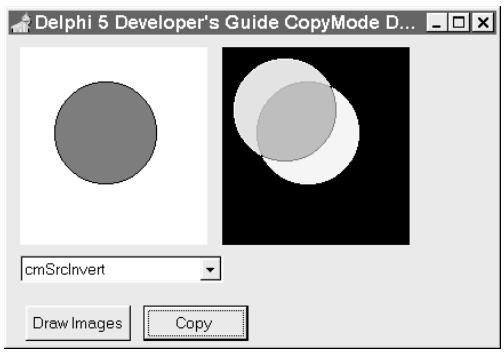


Рис. 8.9. Операция копирования с использованием значения `cmSrcInvert`

В листинге 8.3 представлен исходный текст проекта, иллюстрирующего установки различных режимов копирования.

Листинг 8.3. Проект, иллюстрирующий использование свойства `CopyMode`

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;
type
TMainForm = class(TForm)
    imgCopyTo: TImage;
    imgCopyFrom: TImage;
    cbCopyMode: TComboBox;
    btnDrawImages: TButton;
    btnCopy: TButton;
    procedure FormShow(Sender: TObject);
    procedure btnCopyClick(Sender: TObject);
    procedure btnDrawImagesClick(Sender: TObject);
  private
    procedure DrawImages;
    procedure GetCanvasRect(AImage: TImage; var ARect: TRect);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.GetCanvasRect(AImage: TImage; var ARect: TRect);
```

```
var
  R: TRect;
  R2: TRect;
begin
  R := AImage.Canvas.ClipRect;
  with AImage do begin
    ARect.TopLeft     := Point(0, 0);
    ARect.BottomRight := Point(Width, Height);
  end;
  R2 := ARect;
  ARect := R2;
end;

procedure TMainForm.DrawImages;
var
  R: TRect;
begin
  // Рисование эллипса внутри первого прямоугольника
  with imgCopyTo.Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    GetCanvasRect(imgCopyTo, R);
    FillRect(R);
    Brush.Color := clRed;
    Ellipse(10, 10, 100, 100);
  end;

  // Рисование эллипса внутри второго прямоугольника
  with imgCopyFrom.Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    GetCanvasRect(imgCopyFrom, R);
    FillRect(R);
    Brush.Color := clBlue;
    Ellipse(30, 30, 120, 120);
  end;

end;

procedure TMainForm.FormShow(Sender: TObject);
begin
  // Инициализация комбинированного списка с выбором первого элемента
  cbCopyMode.ItemIndex := 0;
  DrawImages;
end;

procedure TMainForm.btnCopyClick(Sender: TObject);
var
  cm: Longint;
```

```
  CopyToRect,
  CopyFromRect: TRect;
begin
  // Определение режима копирования на основе выбора элемента в списке
  case cbCopyMode.ItemIndex of
   0:  cm := cmBlackNess;
   1:  cm := cmDstInvert;
   2:  cm := cmMergeCopy;
   3:  cm := cmMergePaint;
   4:  cm := cmNotSrcCopy;
   5:  cm := cmNotSrcErase;
   6:  cm := cmPatCopy;
   7:  cm := cmPatInvert;
   8:  cm := cmPatPaint;
   9:  cm := cmSrcAnd;
   10: cm := cmSrcCopy;
   11: cm := cmSrcErase;
   12: cm := cmSrcInvert;
   13: cm := cmSrcPaint;
   14: cm := cmWhiteness;
   else
     cm := cmSrcCopy;
  end;

  // Присвоение выбранного режима копирования св-ву CopyMode 1-го изображения
  imgCopyTo.Canvas.CopyMode := cm;
  GetCanvasRect(imgCopyTo, CopyToRect);
  GetCanvasRect(imgCopyFrom, CopyFromRect);

  // Копирование второго изображения в первое с использованием
  // значения CopyMode для первого
  imgCopyTo.Canvas.CopyRect(CopyToRect, imgCopyFrom.Canvas, CopyFromRect);
end;

procedure TMainForm.btnDrawImagesClick(Sender: TObject);
begin
  DrawImages;
end;

end.
```

Этот проект сначала рисует эллипс на двух компонентах `TImage` (`imgFromImage` и `imgToImage`), а по щелчку на кнопке Copy объект `imgFromImage` копируется в объект `imgToImage1` с использованием для свойства `CopyMode` значения, полученного из компонента `cbCopyMode`.

Другие свойства

Объект `TCanvas` обладает и другими свойствами, которые будут подробно рассмотрены ниже — по мере их использования в примерах программ. В этом разделе мы ограничимся только их кратким описанием.

Свойство TCanvas.ClipRect представляет область канвы, в которой может быть выполнено рисование. Это свойство используется для выделения определенной области данной канвы, за пределами которой рисование невозможно.

По умолчанию свойство ClipRect представляет всю область канвы, что может навести вас на мысль об использовании этого свойства для получения границ канвы. Но вот тут-то и можно попасть впросак: свойство ClipRect не всегда представляет общий размер своего компонента, и его значение может быть меньше области отображения объекта Canvas.

Свойство TCanvas.Handle предоставляет доступ к реальному контексту устройства, который инкапсулируется экземпляром класса TCanvas. Контексты устройств будут рассмотрены ниже в этой главе.

Свойство TCanvas.PenPos просто хранит значения координат x, y позиции пера канвы. Изменить позицию пера можно с помощью таких методов класса TCanvas, как MoveTo(), LineTo(), PolyLine(), TextOut() и др.

Использование методов класса TCanvas

Класс TCanvas инкапсулирует многие функции рисования интерфейса GDI. Используя методы класса TCanvas, можно рисовать линии и фигуры, писать текст, копировать целые области из одной канвы в другую и даже растягивать некоторую область канвы, чтобы заполнить ее содержимым бо́льшую область.

Рисование линий с помощью методов класса TCanvas

Метод TCanvas.MoveTo() изменяет позицию рисования объекта Canvas.Pen на поверхности канвы. При выполнении следующей строки позиция рисования перемещается в верхний левый угол канвы:

```
Canvas.MoveTo(0, 0);
```

Метод TCanvas.LineTo() предназначен для проведения прямой линии из текущей позиции на поверхности канвы в позицию, заданную параметрами функции LineTo(). Совместное использование функций MoveTo() и LineTo() позволяет рисовать прямые линии в любом месте канвы. С помощью следующих строк можно провести прямую линию из верхней левой позиции клиентской области формы в ее нижний правый угол:

```
Canvas.MoveTo(0, 0);
Canvas.LineTo(ClientWidth, ClientHeight);
```

Использование методов MoveTo() и LineTo() было продемонстрировано ранее — в разделе, посвященном свойству TCanvas.Pen.

На заметку

Delphi в настоящее время способен поддерживать текст и расположение элементов управления с ориентацией справа налево; некоторые элементы управления изменяют систему координат канвы, переворачивая ось X. Поэтому при запуске Delphi в ближневосточной (Middle Eastern) версии Windows приведенный вызов MoveTo() может перенести позицию рисования пером в верхний правый угол окна.

Рисование фигур с помощью методов класса TCanvas

В классе TCanvas предусмотрены различные методы воспроизведения фигур на канве: Arc(), Chord(), Ellipse(), Pie(), Polygon(), PolyLine(), Rectangle() и RoundRect(). Чтобы нарисовать эллипс, используйте метод Ellipse(), как показано в следующей строке:

```
Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
```

Можно также выполнить заливку любой области канвы с помощью узора кисти, заданного в свойстве Canvas.Brush.Style. При выполнении следующего фрагмента будет нарисован эллипс, закрашенный в соответствии со значением свойства Canvas.Brush.Style:

```
Canvas.Brush.Style := bsCross;
Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
```

Вы уже знаете, как вместо значения свойства Canvas.Brush.Style использовать растровый узор, определяемый свойством TCanvas.Brush.Bitmap. Выше было показано, как с помощью этого узора выполняется заливка некоторой области канвы. Точно так же этот растровый узор применяется и для заливки фигур (мы продемонстрируем это позже).

В некоторые методы рисования фигур на канве передаются дополнительные параметры, предназначенные для описания фигуры, которую нужно нарисовать. Например, метод PolyLine() принимает массив записей TPoint, определяющих позиции (или координаты пикселей) на канве, по которым будет проходить линия, соединяющая указанные точки. В Delphi 5 объект TPoint — это запись, содержащая координаты (x, y). Он определяется следующим образом:

```
TPoint = record
  X: Integer;
  Y: Integer;
end; ()
```

Пример программы рисования фигур

В листинге 8.4 содержится текст приложения, иллюстрирующего использование различных методов класса TCanvas, предназначенных для рисования фигур на канве. Полный текст этого проекта содержится на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Листинг 8.4. Иллюстрация рисования фигур

```
unit MainFrm;

interface

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

```
type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiShapes: TMenuItem;
    mmiArc: TMenuItem;
    mmiChord: TMenuItem;
    mmiEllipse: TMenuItem;
    mmiPie: TMenuItem;
    mmiPolygon: TMenuItem;
    mmiPolyline: TMenuItem;
    mmiRectangle: TMenuItem;
    mmiRoundRect: TMenuItem;
    N1: TMenuItem;
    mmiFill: TMenuItem;
    mmiUseBitmapPattern: TMenuItem;
    procedure mmiFillClick(Sender: TObject);
    procedure mmiArcClick(Sender: TObject);
    procedure mmiChordClick(Sender: TObject);
    procedure mmiEllipseClick(Sender: TObject);
    procedure mmiUseBitmapPatternClick(Sender: TObject);
    procedure mmiPieClick(Sender: TObject);
    procedure mmiPolygonClick(Sender: TObject);
    procedure mmiPolylineClick(Sender: TObject);
    procedure mmiRectangleClick(Sender: TObject);
    procedure mmiRoundRectClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure mmiPolyBezierClick(Sender: TObject);
  private
    FBitmap: TBitmap;
  public
    procedure ClearCanvas;
    procedure SetFillPattern;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.ClearCanvas;
begin
 // Очистка содержимого канвы
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
```

```
    FillRect(ClientRect);
  end;
end;

procedure TMainForm.SetFillPattern;
begin
  { Определение того, нужно ли рисовать фигуру с использованием растрового
    узора. Если да, то нужно загрузить файл растра. В противном случае
    следует использовать стандартный узор кисти. }
  if mmiUseBitmapPattern.Checked then
    Canvas.Brush.Bitmap := FBitmap
  else
    with Canvas.Brush do
    begin
      Bitmap := nil;
      Color := clBlue;
      Style := bsCross;
    end;
end;

procedure TMainForm.mmiFillClick(Sender: TObject);
begin
  mmiFill.Checked := not mmiFill.Checked;
  { Если признак mmiUseBitmapPattern был установлен, эту установку нужно
    отменить и установить свойство использования растра кисти равным nil. }
  if mmiUseBitmapPattern.Checked then
  begin
    mmiUseBitmapPattern.Checked := not mmiUseBitmapPattern.Checked;
    Canvas.Brush.Bitmap := nil;
  end;
end;

procedure TMainForm.mmiUseBitmapPatternClick(Sender: TObject);
begin
  { Установка признаков mmiFill.Checked и mmiUseBitmapPattern.Checked,
    которая приведет к вызову процедуры SetFillPattern. Однако, если
    признак mmiUseBitmapPattern станет равным True, свойству
    Canvas.Brush.Bitmap присваивается значение nil. }
  mmiUseBitmapPattern.Checked := not mmiUseBitmapPattern.Checked;
  mmiFill.Checked := mmiUseBitmapPattern.Checked;
  if not mmiUseBitmapPattern.Checked then
    Canvas.Brush.Bitmap := nil;
end;

procedure TMainForm.mmiArcClick(Sender: TObject);
begin
  ClearCanvas;
  with ClientRect do
    Canvas.Arc(Left, Top, Right, Bottom, Right, Top, Left, Top);
```

```
end;

procedure TMainForm.mmiChordClick(Sender: TObject);
begin
  ClearCanvas;
  with ClientRect do
  begin
    if mmiFill.Checked then
      SetFillPattern;
    Canvas.Chord(Left, Top, Right, Bottom, Right, Top, Left, Top);
  end;
end;

procedure TMainForm.mmiEllipseClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
end;
procedure TMainForm.mmiPieClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.Pie(0, 0, ClientWidth, ClientHeight, 50, 5, 300, 50);
end;

procedure TMainForm.mmiPolygonClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.Polygon([Point(0, 0), Point(150, 20), Point(230, 130),
                  Point(40, 120)]);
end;

procedure TMainForm.mmiPolylineClick(Sender: TObject);
begin
  ClearCanvas;
  Canvas.PolyLine([Point(0, 0), Point(120, 30), Point(250, 120),
    Point(140, 200), Point(80, 100), Point(30, 30)]);
end;

procedure TMainForm.mmiRectangleClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
```

```
  Canvas.Rectangle(10 , 10, 125, 240);
end;

procedure TMainForm.mmiRoundRectClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.RoundRect(15, 15, 150, 200, 50, 50);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  FBitmap := TBitmap.Create;
  FBitMap.LoadFromFile('Pattern.bmp');
  Canvas.Brush.Bitmap := nil;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FBitmap.Free;
end;

procedure TMainForm.mmiPolyBezierClick(Sender: TObject);
begin
  ClearCanvas;
  Canvas.PolyBezier([Point(0, 100), Point(100,0), Point(200, 50),
    Point(300, 100)]);
end

end.
```

Функции рисования фигур выполняются обработчиками событий главного меню. Два открытых метода, `ClearCanvas()` и `SetFillPattern()`, служат для этих обработчиков событий вспомогательными функциями. При выборе одного из первых восьми элементов меню вызывается соответствующая функция рисования фигуры на канве формы, а с помощью двух последних элементов меню, Fill (Залить) и Use Bitmap Pattern (Использовать растровый узор), определяется, каким образом нужно выполнить заливку формы одним из стандартных узоров кисти или с применением файла растрового узора соответственно.

Назначение и возможности метода `ClearCanvas()` вам уже должны быть знакомы. Метод `SetFillPattern()` предназначен для определения способа заливки фигур, нарисованных другими методами: с помощью одного из стандартных узоров кисти либо растрового файла узора. Если выбран растровый файл, он назначается свойству `Canvas.Brush.Bitmap`.

Все обработчики событий, приступающие к рисованию фигуры, прежде всего обращаются к функции `ClearCanvas()` для стирания всего, что было нарисовано ранее. Затем, если свойство `mmiFill.Checked` установлено равным `True`, они вызывают функцию `SetFillPattern()`. И только после всего этого выполняется рисование соответствующей фигуры путем вызова нужного метода класса `TCanvas`. Комментарии, приведенные в исходном тексте программы, разъясняют назначение каждой функции. Тем не менее метод `mmiPolylineClick()` заслуживает отдельного обсуждения.

В процессе рисования фигур их внутреннюю область, ограниченную замкнутым контуром, бывает необходимо залить либо стандартным узором кисти, либо узором из растрового файла. И хотя, казалось бы, ничто не мешает использовать методы `PolyLine()` и `FloodFill()` для создания замкнутой границы, заполненной каким-нибудь узором, прибегать к этому способу все-таки не рекомендуется. Метод `PolyLine()` применятся исключительно для рисования линий. Если же вам нужно нарисовать закрашенные многоугольники, вызовите метод `TCanvas.Polygon()`. Неточность в 1 пиксель при рисовании линий с помощью функции `PolyLine()` вызовет при обращении к функции `FloodFill()` заливку всей канвы. При применении для рисования фигур с заливкой функции `Polygon()` используются математические методы, которые защищены от подобных колебаний в позиционировании пикселей.

Вывод текста с помощью методов класса `TCanvas`

Класс `TCanvas` инкапсулирует функции интерфейса Win32 GDI для вывода текста на поверхность рисования. Их использованию и будет уделено внимание в следующих разделах. Кроме того, вы узнаете, как использовать другие функции, которые не инкапсулированы классом `TCanvas`.

Использование функций вывода текста, принадлежащих классу `TCanvas`

Как отмечалось в предыдущих главах, для вывода текста на поверхность рисования используется определенная в классе `TCanvas` функция `TextOut()`. Однако класс `TCanvas` содержит и некоторые другие полезные методы (`TextWidth()` и `TextHeight()`), предназначенные для определения размера текста в строке (в пикселях), использующей шрифт, воспроизводимый классом `TCanvas`. В следующем фрагменте программы определяется ширина и высота строки “Delphi 5 -- Yes!”:

```
var
  S: String;
  w, h: Integer;
begin
  S := 'Delphi 5 -- Yes!';
  w := Canvas.TextWidth(S);
  h := Canvas.TextHeight(S);
end.
```

Существует также метод `TextRect()`, который записывает текст на форму, но при этом помещает его только внутрь прямоугольника, заданного структурой `TRect`. Текст, не поместившийся в пределах границ `TRect`, усекается. Рассмотрим следующий пример:

```
Canvas.TextRect(R,0,0,'Delphi 5 Yes!');
```

Здесь строка “`Delphi 5 -- Yes!`” записывается на канву, начиная с позиции `0,0`. Однако та часть строки, которая выпадает за пределы координат, заданных параметром `R` (имеющим тип структуры `TRect`), отсекается.

В листинге 8.5 демонстрируется использование некоторых функций вывода текста.

Листинг 8.5. Операции вывода текста

```
unit MainFrm;

interface

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

const
  DString = 'Delphi 5 YES!';
  DString2 = 'Delphi 5 Rocks!';

type

  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiText: TMenuItem;
    mmiTextRect: TMenuItem;
    mmiTextSize: TMenuItem;
    mmiDrawTextCenter: TMenuItem;
    mmiDrawTextRight: TMenuItem;
    mmiDrawTextLeft: TMenuItem;
    procedure mmiTextRectClick(Sender: TObject);
    procedure mmiTextSizeClick(Sender: TObject);
    procedure mmiDrawTextCenterClick(Sender: TObject);
    procedure mmiDrawTextRightClick(Sender: TObject);
    procedure mmiDrawTextLeftClick(Sender: TObject);
  public
    procedure ClearCanvas;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.ClearCanvas;
begin
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    FillRect(ClipRect);
  end;
end;
```

```
procedure TMainForm.mmiTextRectClick(Sender: TObject);
var
  R: TRect;
  TWidth, THeight: integer;
begin
  ClearCanvas;
  Canvas.Font.Size := 18;
  // Вычисление ширины/высоты строки текста
  TWidth := Canvas.TextWidth(DString);
  THeight := Canvas.TextHeight(DString);

  { Инициализация структуры TRect. Высота этого прямоугольника будет
    равна половине высоты строки текста, чтобы проиллюстрировать
    усечение текста границами прямоугольника. }
  R := Rect(1, THeight div 2, TWidth + 1, THeight+(THeight div 2));
  // Рисование прямоугольника на основе размеров текста
  Canvas.Rectangle(R.Left-1, R.Top-1, R.Right+1, R.Bottom+1);
  // Вывод текста внутри прямоугольника
  Canvas.TextRect(R,0,0,DString);
end;
procedure TMainForm.mmiTextSizeClick(Sender: TObject);
begin
  ClearCanvas;
  with Canvas do
  begin
    Font.Size := 18;
    TextOut(10, 10, DString);
    TextOut(50, 50, 'TextWidth = '+IntToStr(TextWidth(DString)));
    TextOut(100, 100, 'TextHeight = '+IntToStr(TextHeight(DString)));
  end;
end;

procedure TMainForm.mmiDrawTextCenterClick(Sender: TObject);
var
  R: TRect;
begin
  ClearCanvas;
  Canvas.Font.Size := 10;
  R := Rect(10, 10, 80, 100);
  // Рисование прямоугольника, обводящего объект TRect границей,
  // отстоящей на 2 пикселя
  Canvas.Rectangle(R.Left-2, R.Top-2, R.Right+2, R.Bottom+2);
  // Вывод текста по центру путем задания опции dt_Center
  DrawText(Canvas.Handle, PChar(DString2), -1, R, dt_WordBreak or
 dt_Center);
end;

procedure TMainForm.mmiDrawTextRightClick(Sender: TObject);
var
  R: TRect;
```

```
begin
  ClearCanvas;
  Canvas.Font.Size := 10;
  R := Rect(10, 10, 80, 100);
  // Рисование прямоугольника, обводящего объект TRect границей,
  // отстоящей на 2 пикселя
  Canvas.Rectangle(R.Left-2, R.Top-2, R.Right+2, R.Bottom+2);
  // Вывод текста с выравниванием вправо путем задания опции dt_Right
  DrawText(Canvas.Handle, PChar(DString2), -1, R, dt_WordBreak or dt_Right);
end;

procedure TMainForm.mmiDrawTextLeftClick(Sender: TObject);
var
 R: TRect;
begin
  ClearCanvas;
  Canvas.Font.Size := 10;
  R := Rect(10, 10, 80, 100);
  // Рисование прямоугольника, обводящего объект TRect границей,
  // отстоящей на 2 пикселя
  Canvas.Rectangle(R.Left-2, R.Top-2, R.Right+2, R.Bottom+2);
  // Вывод текста с выравниванием влево путем задания опции dt_Left
  DrawText(Canvas.Handle, PChar(DString2), -1, R, dt_WordBreak or dt_Left);
end;

end.
```

Как и все предыдущие проекты, данный проект содержит метод `ClearCanvas()`, используемый для стирания содержимого канвы формы.

Различные методы главной формы являются обработчиками событий для главного меню формы.

В методе `mmiTextRectClick()` иллюстрируется использование метода `TCanvas.TextRect()`. Здесь сначала определяется ширина и высота заданного текста, а затем этот текст выводится внутри прямоугольника с высотой, равной половине высоты заданного размера шрифта. Результат работы этого метода показан на рис. 8.10.

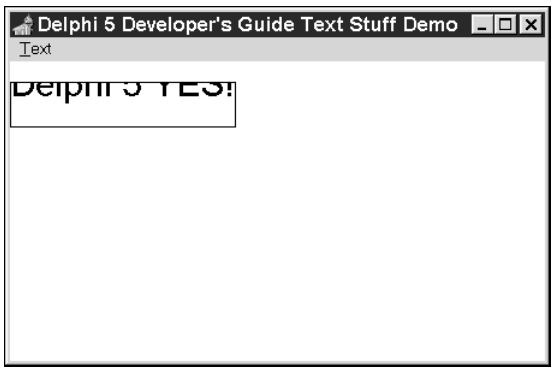


Рис. 8.10. Результат работы метода `mmiTextRectClick()`

На примере функции mmiTextSizeClick() показано, как определяется размер текстовой строки с помощью методов TCanvas.TextWidth() и TCanvas.TextHeight(). Результат работы этой функции показан на рис. 8.11.

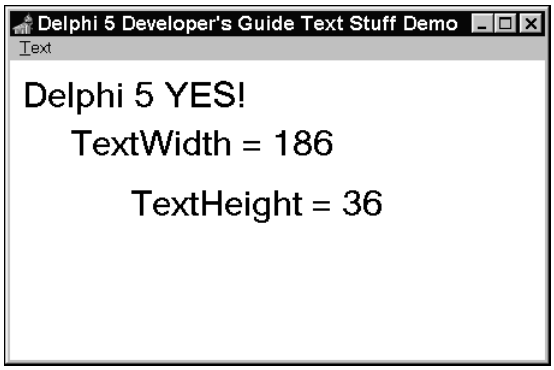


Рис. 8.11. Результат работы функции mmiTextSizeClick()

Использование GDI-функций вывода текста, не принадлежащих классу TCanvas

В приведенном выше примере с помощью методов mmiDrawTextCenter(), mmiDrawTextRight() и mmiDrawTextLeft() демонстрируется использование функции интерфейса Win32 GDI DrawText(), которая *не* инкапсулируется классом TCanvas.

Тем самым показано, что инкапсуляция в Delphi 5 интерфейса Win32 GDI посредством класса TCanvas не мешает использовать все богатство функций интерфейса Win32 GDI. Класс TCanvas лишь упрощает применение наиболее популярных функций, по-прежнему позволяя вызывать любую другую функцию интерфейса Win32 GDI, которая может потребоваться.

Из описания таких функций GDI, как BitBlt() или DrawText(), следует, что одним из обязательных их параметров является контекст устройства (device context — DC). Доступ к контексту устройства осуществляется через свойство канвы Handle, т.е. свойство TCanvas.Handle содержит контекст устройства (DC) для данной канвы.

Контексты устройств

Контексты устройств (Device Context) представляют собой дескрипторы, предусмотренные интерфейсом Win32 для идентификации связи приложения Win32 с такими выходными устройствами, как монитор, принтер или плоттер, осуществляемой через драйвер устройства. В традиционном Windows-ориентированном программировании для прорисовки поверхности окна программист должен был сам позаботиться о запросе на получение DC. По завершении операции он был обязан вернуть DC системе. В Delphi 5 процедура работы с DC упрощена посредством инкапсуляции управления DC в классе TCanvas. Этот класс даже кэширует используемые DC, сохраняя их для последующих применений, с целью уменьшения частоты запросов к системе Win32 и, следовательно, увеличения общего быстродействия программы.

Чтобы показать, как класс TCanvas используется совместно с функциями интерфейса Win32 GDI, в приведенном выше листинге была применена функция GDI DrawText(), предназначенная для вывода текста с дополнительными возможностями форматирования. Этой функции передается пять описанных ниже параметров.

Параметр	Описание
DC	Контекст устройства поверхности рисования
Str	Указатель на буфер, который содержит подлежащий выводу текст. Если параметр Count равен -1, то это должна быть строка с завершающим нуль-символом
Count	Число байтов в строке Str. Если это значение равно -1, то Str должен указывать на строку с завершающим нуль-символом
Rect	Указатель на структуру TRect, содержащую координаты прямоугольника, в котором форматируется текст
Format	Битовое поле с признаками, определяющими различные опции форматирования для параметра Str

В приведенной выше программе структура TRect инициализируется с помощью функции Rect(). Эта структура используется для рисования прямоугольника вокруг текста, выводимого с использованием функции DrawText(). Каждый из трех методов передает этой функции различный набор признаков форматирования. Признаки dt_WordBreak и dt_Center используются для центрирования текста в прямоугольнике, заданном переменной R типа TRect. Объединенные операцией логического ИЛИ признаки форматирования dt_WordBreak и dt_Right служат для выравнивания текста в прямоугольнике по правому краю, а dt_WordBreak и dt_Left — по левому краю. Спецификатор dt_WordBreak предназначен для переноса слов на новую строку в пределах границы, обусловленной шириной (которая задается соответствующим параметром прямоугольника), и для модификации высоты прямоугольника после реализации этого переноса слов.

Результат работы метода mmiDrawTextCenterClick() показан на рис. 8.12.

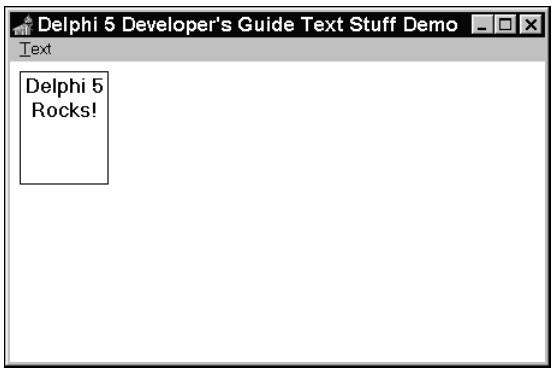


Рис. 8.12. Результат работы метода mmiDrawTextCenterClick()

Класс TCanvas также содержит методы Draw(), Copy(), CopyRect() и StretchDraw(), с помощью которых можно рисовать, растягивать и сжимать изображения или их части либо копировать их на другую канву. Об использовании функции CopyRect() мы поговорим далее в этой главе, когда обратимся к созданию программы рисования. Кроме того, в главе 16, "MDI-приложения", показано, как применять метод StretchDraw() для растяжения растрового изображения в заданной области клиента формы.

Координатные системы и режимы отображения

Большинство GDI-функций рисования требует указывать набор координат, который задает позицию для рисования. Единицей измерения этих координат является пиксель. Кроме того, GDI-функции предполагают определенную ориентацию осей координат, т.е. напрвление, в котором значения координат каждой из осей возрастают или убывают. При выполнении функций рисования интерфейс Win32 опирается на два базовых фактора: на координатную систему и на режим отображения области рисования.

Координатные системы Win32 в общем случае не отличаются от других координатных систем. Вы определяете координаты для осей X и Y, а система Win32 отмечает указанную позицию точкой на поверхности рисования с учетом действующей ориентации. В системе Win32 используется три системы координат для определения областей на поверхностях рисования, которые называются координатами *устройства*, *логическими* и *мировыми*. Windows 95 не поддерживает выполнение преобразований в мировых координатах (под преобразованиями подразумеваются такие операции, как вращение растра, сдвиг, скручивание и т.п.). Поэтому в этой главе рассматриваются только две первые системы координат.

Координаты устройства

Как следует из названия, эти координаты связаны с устройством, на котором работает интерфейс Win32. Их единицами измерения являются пиксели, а ориентация горизонтальной и вертикальной осей такова, что значения координат увеличиваются слева направо и сверху вниз. Например, если запустить Windows на дисплее с разрешением 640×480, то координаты точки в верхнем левом углу вашего устройства будут равны (0,0), а в нижнем правом углу — (639,479).

Логические координаты

Под логическими понимают координаты системы, используемые любой областью в системе Win32, обладающей контекстом устройства (DC). Такими областями являются экран, форма или клиентская область формы. Различие между координатами устройства и логическими координатами разъясняется ниже, а пока мы остановимся на координатах экрана, окна формы и области клиента.

Координаты экрана

Координаты экрана имеют прямое отношение к дисплею и поэтому измеряются в пикселях. Для дисплея с разрешением 640×480 свойства `Screen.Width` и `Screen.Height` равны 640 и 480 пикселей соответственно. Чтобы получить контекст устройства экрана, используйте функцию Win32 API `GetDC()`. При этом любая функция, считывающая контекст устройства, должна работать совместно с функцией `ReleaseDC()`; эта согласованность иллюстрируется следующим фрагментом:

```
var
  ScreenDC: HDC;
begin
```

```
  Screen DC := GetDC(0);
  try
    { Любые действия, использующие контекст устройства ScreenDC }
  finally
     ReleaseDC(0, ScreenDC);
  end;
end;
```

Координаты формы

Координаты формы можно считать синонимом термина *координаты окна*, когда рассматривается вся форма (или окно) целиком, включая строку заголовка и ограничивающую рамку. В Delphi 5 не предусмотрено такого свойства формы, которое бы содержало значение DC для области рисования формы, но его можно получить с помощью функции Win32 API `GetWindowDC()`:

```
MyDC := GetWindowDC(Form1.Handle);
```

Эта функция возвращает контекст устройства для дескриптора окна, переданного ей в качестве параметра.

На заметку

Для инкапсуляции значений контекстов устройств, получаемых при обращениях к функциям `GetDC()` и `GetWindowDC()`, можно использовать объект `TCanvas`, что позволит применять методы класса `TCanvas` вместо явных значений контекстов устройств. Для этого достаточно создать экземпляр класса `TCanvas`, а затем присвоить результат вызова функции `GetDC()` или `GetWindowDC()` свойству `TCanvas.Handle`. Работоспособность этого подхода достигается благодаря тому, что класс приобретает право собственности на присвоенный ему дескриптор, после чего несет за него полную ответственность, освобождая DC при освобождении канвы. Вот фрагмент, иллюстрирующий описанный метод:

```
var
  c: TCanvas;
begin
  c := TCanvas.Create;
  try
    c.Handle := GetDC(0);
    c.TextOut(10, 10, 'Hello World');
  finally
    c.Free;
  end;
end;
```

Координаты клиентской области формы относятся к области клиента формы, контекст устройства (DC) которой равен значению свойства `Handle` канвы (объекта `Canvas`) формы. Размеры клиентской области можно получить с помощью свойств `Canvas.ClientWidth` и `Canvas.ClientHeight`.

Отображение координат

Что же мешает при работе с функциями рисования вместо логических координат использовать координаты устройства? Рассмотрим следующую строку кода:

```
Form1.Canvas.TextOut(0, 0, 'Upper Left Corner of Form');
```

При выполнении этой программной строки заданная строка текста размещается в верхнем левом углу формы. Заданные координаты (0,0) указывают на позицию (0,0) в контексте устройства формы, т.е. являются логическими. Но позиция (0,0) для данной формы имеет совершенно другие значения в системе координат устройства и зависит от конкретного расположения формы на экране. Если верхний левый угол формы совпадает с верхним левым углом экрана, то координаты формы (0,0) могут и в самом деле совпасть с позицией (0,0) в координатах устройства. Однако при перемещении формы на другое место точка с координатами формы (0,0) будет соответствовать совершенно другой позиции, определяемой в координатах устройства.

Для преобразования логических координат точки в координаты устройства и наоборот используйте функции Win32 API `ClientToScreen()` и `ScreenToClient()` соответственно. Эти функции являются также методами класса `TControl`. Обратите внимание на то, что они работают только с контекстами устройств экрана, связанными с видимыми элементами управления. Для контекста устройства принтера или метафайла, которые не имеют отношения к экрану, преобразование логических пикселей в пиксели устройства и обратно выполняется с помощью функций Win32 API `LPtoDP()` и `DPtoLP()` соответственно.

При выполнении метода `Canvas.TextOut()` на самом деле используются координаты устройства. Для этого интерфейс Win32 должен преобразовать логические координаты конкретного DC в координаты устройства. Это достигается за счет использования определенного режима отображения, связанного с DC.

Другая причина использования логических координат — нежелание использовать пиксели для работы с функциями рисования. Вполне вероятно, что при рисовании вам удобнее работать с такими единицами измерения, как дюймы или миллиметры. Система Win32 позволяет изменять единицы измерения путем изменения режима отображения, как будет показано ниже.

Режимы отображения определяют два атрибута для DC: преобразование, которое система Win32 использует для пересчета логических единиц измерения в единицы измерения устройства, и ориентацию осей X, Y для данного DC.

На заметку

Возможно, функции рисования, режимы отображения, ориентация и тому подобные вещи не кажутся связанными столь уж очевидным образом с контекстом устройства, поскольку в Delphi 5 для рисования используется канва. Напомним, что класс `TCanvas` является оболочкой для DC. Эта связь становится более явной при сравнении функций интерфейса Win32 GDI с эквивалентными методами объекта `Canvas`:

метод класса `TCanvas`: `Canvas.Rectangle(0, 0, 50, 50);`

функция GDI: `Rectangle(ADC, 0, 0, 50, 50);`

При использовании функции GDI требуется явная передача контекста устройства (DC) конкретной функции, в то время как метод канвы использует DC неявным образом — за счет инкапсуляции.

В системе Win32 предусмотрена возможность задания режима отображения для DC или свойства `TCanvas.Handle`. Определено восемь режимов отображения, которые перечислены вместе со своими атрибутами в табл. 8.4. Ниже, в листинге 8.6, приведен текст примера иллюстрирующего некоторые особенности использования режимов отображения.

Таблица 8.4. Режимы отображения системы Win32

Режим отображения	Размер логической единицы измерения	Ориентация (X,Y)
MM_ANISOTROPIC	Произвольно (x <> y или x = y)	Задаваемая/Задаваемая
MM_HIENGLISH	0.001 дюйма	Направо/Вверх
MM_HIMETRIC	0.01 мм	Направо/Вверх
MM_ISOTROPIC	Произвольно (x = y)	Задаваемая/Задаваемая
MM_LOENGLISH	0.01 дюйма	Направо/Вверх
MM_LOMETRIC	0.1 мм	Направо/Вверх
MM_TEXT	1 пиксель	Направо/Вниз
MM_TWIPS	1/1440 дюйма	Направо/Вверх

В системе Win32 определено несколько функций, с помощью которых можно изменить или получить информацию о режимах отображения для данного DC.

SetMapMode()	Устанавливает режим отображения для данного контекста устройства
GetMapMode()	Получает режим отображения для данного контекста устройства
SetWindowOrgEx()	Определяет начало координат окна для данного контекста устройства (точка с координатами 0,0)
SetViewPortOrgEx()	Определяет начало координат окна проекции для данного контекста устройства (точка с координатами 0,0)
SetWindowExtEx()	Определяет величины X,Y для DC данного окна. Эти значения используются вместе с величинами X,Y окна для выполнения перевода логических единиц измерения в единицы измерения устройства
SetViewPortExtEx()	Определяет величины X,Y для DC данного окна проекции. Эти значения используются вместе с величинами X,Y окна для выполнения перевода логических единиц измерения в единицы измерения устройства

Обратите внимание на то, что в названии этих функций содержится либо слово *Window*, либо слово *ViewPort*, а в описании употребляется соответственно либо слово *окно*, либо словосочетание *окно проекции*. Эти слова просто означают средства, с помощью которых интерфейс Win32 GDI может выполнять преобразование из логических единиц измерения в единицы измерения устройства. Функции со словом *Window* связаны с логической системой координат, а со словом *ViewPort* — с системой координат устройства. За исключением таких режимов отображения, как MM_ANISOTROPIC и MM_ISOTROPIC, вам вряд ли придется слишком беспокоиться об их установке. Следует отметить, что по умолчанию в системе Win32 используется режим MM_TEXT.

На заметку

Установленный по умолчанию режим отображения MM_TEXT напрямую (1:1) переводит логические координаты в координаты устройства. Поэтому, если вы не изменяли режим отображения, всегда используйте координаты устройства для всех контекстов устройств. Есть несколько функций API, для которых этот момент очень важен. Например, высота шрифта всегда определяется в пикселях устройства, а не в логических пикселях.

Установка режима отображения

Вы, вероятно, заметили, что все режимы отображения используют различные размеры логических единиц измерения. Поэтому в разных случаях будет удобно использовать различные режимы отображения. Например, если независимо от разрешения выходного устройства вам понадобится провести линию шириной в 2 дюйма, то режим MM_LOENGLISH окажется весьма кстати.

Чтобы в качестве примера нарисовать квадрат со стороной в один дюйм, установите сначала свойство Form1.Canvas.Handle равным значению MM_HIENGLISH или MM_LOENGLISH:

```
SetMapMode(Canvas.Handle, MM_LOENGLISH);
```

Затем нарисуйте требуемый квадрат, используя соответствующие единицы измерения. Поскольку режим MM_LOENGLISH использует 1/100 дюйма, просто передайте значение 100:

```
Canvas.Rectangle(0, 0, 100, 100);
```

Поскольку при установке режима отображения MM_TEXT в качестве единиц измерения используются пиксели, то для считывания информации, необходимой для преобразования пикселей в дюймы или миллиметры, можно использовать функцию GetDeviceCaps(). Затем можно выполнить собственные вычисления (об этом речь идет в главе 10, “Печать в Delphi 5”). Режимы отображения можно рассматривать как средство, позволяющее интерфейсу Win32 работать вместо вас. Однако, вероятнее всего, вам никогда не удастся добиться точных размеров объектов, отображаемых на экране дисплея. На это есть несколько причин. Во-первых, Windows не умеет определять размеры экрана — ей остается только предполагать их. Кроме того, в Windows обычно завышается масштаб отображения, чтобы улучшить читабельность текста на мониторах. И поэтому, например, 10-пунктовый шрифт на экране имеет такую же высоту, как 12- или даже 14-пунктовый шрифт на бумаге.

Установка величин Window/Viewport

Функции SetWindowExtEx() и SetViewPortExtEx() позволяют указать способ преобразования логических единиц измерения в единицы устройства. Эти функции действуют только при установке режима отображения окна равным MM_ANISOTROPIC или MM_ISOTROPIC. В любом другом случае они игнорируются. Таким образом, следующие две строки кода означают, что на одну логическую единицу приходится две единицы устройства (два пикселя):

```
SetWindowExtEx(Canvas.Handle, 1, 1, nil)
SetViewportExtEx(Canvas.Handle, 2, 2, nil);
```

Аналогично, следующие строки кода означают, что на пять логических единиц приходится десять единиц устройства:

```
SetWindowExtEx(Canvas.Handle, 5, 5, nil)
SetViewportExtEx(Canvas.Handle, 10, 10, nil);
```

Вероятно, вы заметили, что последний пример эквивалентен предыдущему. В обоих случаях сохраняется одно и то же отношение логических единиц к единицам устройства, а именно 1:2. В следующем примере демонстрируется, как этот подход можно использовать для изменения единиц в форме:

```
SetWindowExtEx(Canvas.Handle, 500, 500, nil)
SetViewportExtEx(Canvas.Handle, ClientWidth, ClientHeight, nil);
```

На заметку Изменение режима отображения для контекста устройства, представленного канвой класса библиотеки VCL, ни к чему особенному вас не обязывает, т.е. при необходимости вы можете спокойно вернуться к исходному режиму. В общем случае в процессе рисования режим отображения должен быть установлен внутри дескриптора.

Поэтому, несмотря на изменение размеров формы, вы сможете работать с формой, ширина и высота области клиента которой составляют 500×500 единиц (не пикселей!).

С помощью функций `SetWindowOrgEx()` и `SetViewPortOrgEx()` можно изменить расположение начала координат, т.е. позицию `(0,0)`, которая по умолчанию находится в верхнем левом углу области клиента формы для режима отображения `MM_TEXT`. Как правило, вам придется модифицировать только начало координат окна проекции. Например, с помощью следующей программной строки устанавливается четырехквадрантная система координат, показанная на рис. 8.13:

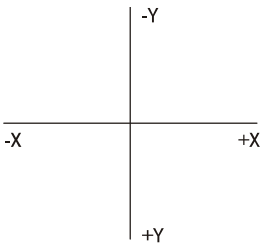


Рис. 8.13. Система координат с четырьмя квадрантами

```
SetViewportOrgEx(Canvas.Handle, ClientWidth div 2,
ClientHeight div 2, nil);
```

Обратите внимание на то, что в качестве последнего параметра функциям `SetWindowOrgEx()`, `SetViewPortOrgEx()`, `SetWindowExtEx()` и `SetViewPortExtEx()` передается значение `nil`. Функциям `SetWindowOrgEx()` и `SetViewPortOrgEx()` передается параметр типа `TPoint`, которому присваивается последнее значение начала координат, чтобы при необходимости его можно было восстановить для данного DC. Функциям `SetWindowExtEx()` и `SetViewPortExtEx()` передается структура `TSize`, с целью сохранения исходных величин для DC — по той же причине.

Пример использования режимов отображения

В листинге 8.6 демонстрируется, как устанавливать режимы отображения, начала координат окна и окна проекции, а также предельные величины окна и окна проекции. Кроме того, здесь иллюстрируется процесс рисования различных фигур с помощью методов `TCanvas`. Файлы этого проекта можно найти на компакт-диске, прилагаемом ко второму тому (см. также `www.williamspublishing.com`.).

Листинг 8.6. Иллюстрация использования режимов отображения

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, DB, DBCGrids, DBTables;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiMappingMode: TMenuItem;
    mmiMM_ISOTROPIC: TMenuItem;
    mmiMM_ANISOTROPIC: TMenuItem;
```

```
    mmiMM_LOENGLISH: TMenuItem;
    mmiMM_HIINGLISH: TMenuItem;
    mmiMM_LOMETRIC: TMenuItem;
    mmiMM_HIMETRIC: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure mmiMM_ISOTROPICClick(Sender: TObject);
    procedure mmiMM_ANSITROPICClick(Sender: TObject);
    procedure mmiMM_LOENGLISHClick(Sender: TObject);
    procedure mmiMM_HIINGLISHClick(Sender: TObject);
    procedure mmiMM_LOMETRICClick(Sender: TObject);
    procedure mmiMM_HIMETRICClick(Sender: TObject);
  public
    MappingMode: Integer;
    procedure ClearCanvas;
    procedure DrawMapMode(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation
{$R *.DFM}

procedure TMainForm.ClearCanvas;

begin
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    FillRect(ClipRect);
  end;
end;

procedure TMainForm.DrawMapMode(Sender: TObject);
var
  PrevMapMode: Integer;
begin
  ClearCanvas;
  Canvas.TextOut(0, 0, (Sender as TMenuItem).Caption);
// Установить режим отображения MM_LOENGLISH и сохранить предыдущий режим
  PrevMapMode := SetMapMode(Canvas.Handle, MappingMode);
  try
    // Установить начало координат окна проекции в левом нижнем углу
    SetViewPortOrgEx(Canvas.Handle, 0, ClientHeight, nil);
    { Нарисовать несколько фигур для иллюстрации рисования с помощью
      различных режимов, заданных параметром MappingMode. }
    Canvas.Rectangle(0, 0, 200, 200);
    Canvas.Rectangle(200, 200, 400, 400);
    Canvas.Ellipse(200, 200, 400, 400);
```

```
    Canvas.MoveTo(0, 0);
    Canvas.LineTo(400, 400);
    Canvas.MoveTo(0, 200);
    Canvas.LineTo(200, 0);
  finally
    // Восстановить предыдущий режим отображения
    SetMapMode(Canvas.Handle, PrevMapMode);
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  MappingMode := MM_TEXT;
end;

procedure TMainForm.mmiMM_ISOTROPICClick(Sender: TObject);
var
  PrevMapMode: Integer;
begin
  ClearCanvas;
  // Установить режим отображения MM_ISOTROPIC и сохранить предыдущий режим
  PrevMapMode := SetMapMode(Canvas.Handle, MM_ISOTROPIC);
  try
    // Установить размеры окна равными 500 x 500
    SetWindowExtEx(Canvas.Handle, 500, 500, nil);
    // Установить размеры окна проекции равными области клиента окна
    SetViewportExtEx(Canvas.Handle, ClientWidth, ClientHeight, nil);
    // Установить начало координат окна проекции в центре области клиента
    SetViewportOrgEx(Canvas.Handle, ClientWidth div 2, ClientHeight div 2, nil);
    // Нарисовать прямоугольник с использованием текущих установок
    Canvas.Rectangle(0, 0, 250, 250);
    { Установить другой размер окна проекции и нарисовать другой прямоугольник.
      Выполнить это еще три раза, чтобы нарисованный прямоугольник
      можно было использовать для представления одной плоскости проекции
      в системе с четырьмя квадрантами. }
    SetViewportExtEx(Canvas.Handle, ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);

    SetViewportExtEx(Canvas.Handle, -ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);

    SetViewportExtEx(Canvas.Handle, -ClientWidth, ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);
    // Нарисовать эллипс в центре области клиента
    Canvas.Ellipse(-50, -50, 50, 50);
  finally
    // Восстановить предыдущий режим отображения
    SetMapMode(Canvas.Handle, PrevMapMode);
  end;
end;
```

```
procedure TMainForm.mmiMM_ANSITROPICClick(Sender: TObject);
var
  PrevMapMode: Integer;

begin
  ClearCanvas;
  // Установить режим отображения MM_ANISOTROPIC и сохранить предыдущий режим
  PrevMapMode := SetMapMode(Canvas.Handle, MM_ANISOTROPIC);
  try
    // Установить размеры окна равными 500 x 500
    SetWindowExtEx(Canvas.Handle, 500, 500, nil);
    // Установить размеры окна проекции равными этой области клиента окна
    SetViewportExtEx(Canvas.Handle, ClientWidth, ClientHeight, nil);
    // Установить начало координат окна проекции в центре области клиента
    SetViewportOrgEx(Canvas.Handle, ClientWidth div 2,
      ClientHeight div 2, nil);
   // Нарисовать прямоугольник с использованием текущих установок
    Canvas.Rectangle(0, 0, 250, 250);
    { Установить другой размер окна проекции и нарисовать другой прямоугольник.
      Выполнить это еще три раза, чтобы нарисованный прямоугольник
      можно было использовать для представления одной плоскости проекции
      в системе с четырьмя квадрантами. }
    SetViewportExtEx(Canvas.Handle, ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);
    SetViewportExtEx(Canvas.Handle, -ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);

    SetViewportExtEx(Canvas.Handle, -ClientWidth, ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);
    // Нарисовать эллипс в центре области клиента
    Canvas.Ellipse(-50, -50, 50, 50);
  finally
    //Восстановить предыдущий режим отображения
    SetMapMode(Canvas.Handle, PrevMapMode);
  end;
end;
procedure TMainForm.mmiMM_LOENGLISHClick(Sender: TObject);
begin
  MappingMode := MM_LOENGLISH;
  DrawMapMode(Sender);
end;

procedure TMainForm.mmiMM_HIINGLISHClick(Sender: TObject);
begin
  MappingMode := MM_HIENGLISH;
  DrawMapMode(Sender);
end;

procedure TMainForm.mmiMM_LOMETRICClick(Sender: TObject);
begin
```

```
  MappingMode := MM_LOMETRIC;
  DrawMapMode(Sender);
end;

procedure TMainForm.mmiMM_HIMETRICClick(Sender: TObject);
begin
  MappingMode := MM_HIMETRIC;
  DrawMapMode(Sender);
end;

end.
```

Поле главной формы `MappingMode` используется для хранения текущего режима отображения, который инициализируется в методе `FormCreate()` установкой значения `MM_TEXT`. Эта переменная устанавливается при вызове методов `MMLOENGLISH1Click()`, `MMHIENGLISH1Click()`, `MMLOMETRIC1Click()` и `MMHIMETRIC1Click()` из соответствующих меню. В этих методах затем выполняется обращение к методу `DrawMapMode()`, который сначала устанавливает режим отображения главной формы равным значению, определяемому переменной `MappingMode`, а затем рисует несколько фигур и прямых, используя для задания их размеров значения констант. Если в процессе рисования фигур используются разные режимы отображения, то размеры этих фигур будут различаться из-за разных единиц измерения, используемых в контексте заданного режима отображения. На рис. 8.14 и 8.15 показаны результаты работы метода `DrawMapMode()` в режимах отображения `MM_LOENGLISH` и `MM_LOMETRIC`.

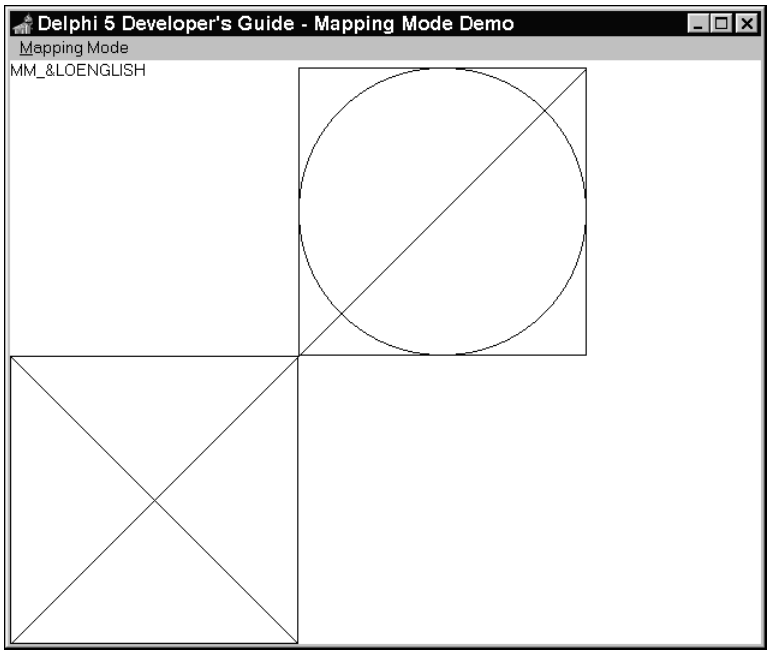


Рис. 8.14. Результат работы метода `DrawMapMode()` с использованием режима отображения `MM_LOENGLISH`

С помощью метода mmiMM_ISOTROPICClick() демонстрируется процесс рисования на канве формы в режиме MM_ISOTROPIC. В этом методе сначала устанавливается указанный режим отображения, а затем предельные величины окна проекции устанавливаются равными размерам области клиента формы. После этого в центре области клиента формы располагается начало координат, что позволяет хорошо просматривать все четыре квадранта системы координат.

После выполнения подготовительных операций этот метод рисует в каждой плоскости проекции прямоугольник, а в центре области клиента — эллипс. Обратите внимание на то, что для рисования различных областей канвы используются одни и те же значения параметров, передаваемых функции Canvas.Rectangle(). Это достигается за счет отрицательного значения либо параметра X, либо параметра Y, либо сразу обоих параметров, передаваемых функции SetViewPortExt().

Цель включения в проект метода mmiMM_ANISOTROPICClick(), выполняющего те же самые операции (за исключением использования режима MM_ANISOTROPIC), состоит в демонстрации принципиальных различий между режимами отображения MM_ISOTROPIC и MM_ANISOTROPIC.

При использовании режима MM_ISOTROPIC интерфейс Win32 гарантирует по обеим осям один и тот же физический размер и выполняет необходимые выравнивания. Но в режиме MM_ANISOTROPIC используются физические размеры, которые в общем случае могут и не быть одинаковыми. Это хорошо видно на рис. 8.16 и 8.17. Вполне очевидно, что режим MM_ISOTROPIC обеспечивает равенство двух осей, в то время как тот же программный код, использующий режим MM_ANISOTROPIC, не гарантирует этого равенства. Более того, при работе в режиме MM_ISOTROPIC можно быть уверенным в том, что равные логические координаты будут преобразованы в координаты устройства с сохранением этого равенства, даже если система координат устройства не является квадратной.

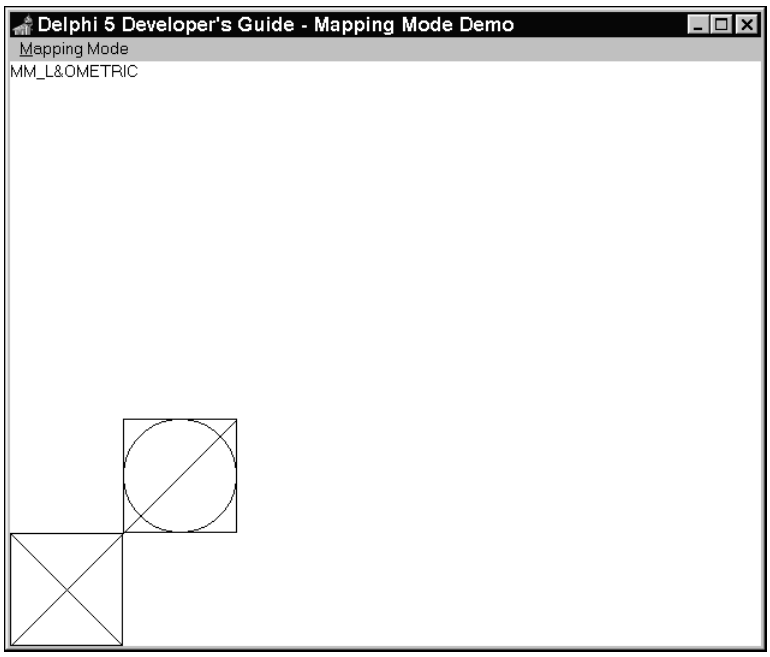


Рис. 8.15. Результат работы метода DrawMapMode() с использованием режима отображения MM_LOMETRIC

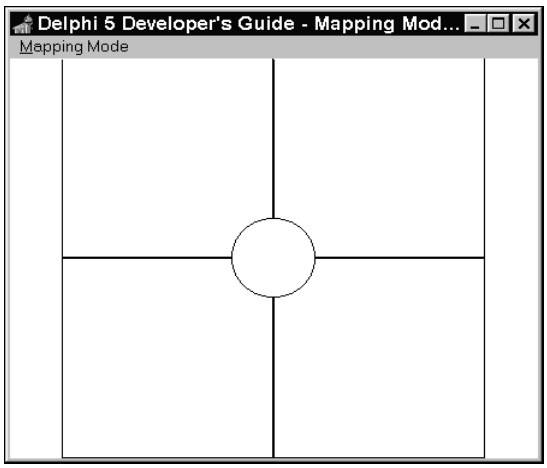


Рис. 8.16. Результат использования режима MM_ISOTROPIC

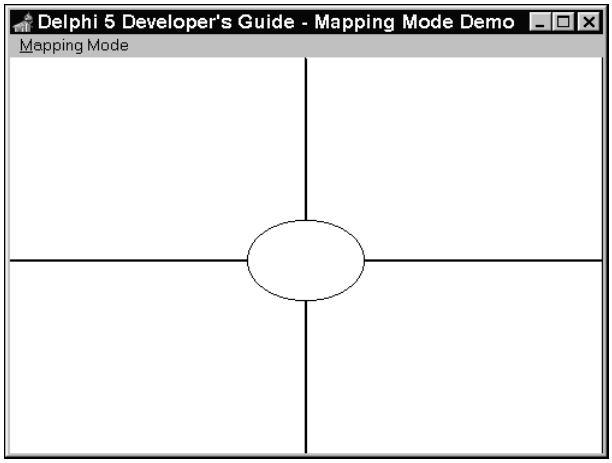


Рис. 8.17. Результат использования режима MM_ANISOTROPIC

Создание программы рисования

На примере приводимой ниже программы рисования демонстрируется использование нескольких более сложных методов работы с интерфейсом GDI и графическими изображениями. Исходный код проекта представлен в листинге 8.7. Полный текст проекта DDGPaint.dpr находится на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Листинг 8.7. Программа рисования DDGPaint

```
unit MainFrm;

interface

uses
```

```
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Buttons, ExtCtrls, ColorGrd, StdCtrls, Menus,
  drwpnel, ComCtrls, DdeMan;

const
  crMove = 1;
type

  TDrawType = (dtLineDraw, dtRectangle, dtEllipse, dtRoundRect,
               dtClipRect, dtCrooked);

  TMainForm = class(TForm)
    sbxMain: TScrollBox;
    imgDrawingPad: TImage;
    pnlToolBar: TPanel;
    sbLine: TSpeedButton;
    sbRectangle: TSpeedButton;
    sbEllipse: TSpeedButton;
    sbRoundRect: TSpeedButton;
    pnlColors: TPanel;
    cgDrawingColors: TColorGrid;
    pnlFgBgBorder: TPanel;
    pnlFgBgInner: TPanel;
    Bevel1: TBevel;
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    N2: TMenuItem;
    mmiSaveAs: TMenuItem;
    mmiSaveFile: TMenuItem;
    mmiOpenFile: TMenuItem;
    mmiNewFile: TMenuItem;
    mmiEdit: TMenuItem;
    mmiPaste: TMenuItem;
    mmiCopy: TMenuItem;
    mmiCut: TMenuItem;
    sbRectSelect: TSpeedButton;
    SaveDialog: TSaveDialog;
    OpenDialog: TOpenDialog;
    stbMain: TStatusBar;
    pbPasteBox: TPaintBox;
    sbFreeForm: TSpeedButton;
    RgGrpFillOptions: TRadioGroup;
    cbxBorder: TCheckBox
    procedure FormCreate(Sender: TObject);
    procedure sbLineClick(Sender: TObject);
    procedure imgDrawingPadMouseDown(Sender: TObject; Button:
      TMouseButton; Shift: TShiftState; X, Y: Integer);
    procedure imgDrawingPadMouseMove(Sender: TObject;
      Shift: TShiftState; X, Y: Integer);
    procedure imgDrawingPadMouseUp(Sender: TObject; Button: TMouseButton;
```

```
    Shift: TShiftState; X, Y: Integer);
  procedure cgDrawingColorsChange(Sender: TObject);
  procedure mmiExitClick(Sender: TObject);
  procedure mmiSaveFileClick(Sender: TObject);
  procedure mmiSaveAsClick(Sender: TObject);
  procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
  procedure mmiNewFileClick(Sender: TObject);
  procedure mmiOpenFileClick(Sender: TObject);
  procedure mmiEditClick(Sender: TObject);
  procedure mmiCutClick(Sender: TObject);
  procedure mmiCopyClick(Sender: TObject);
  procedure mmiPasteClick(Sender: TObject);
  procedure pbPasteBoxMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
  procedure pbPasteBoxMouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
  procedure pbPasteBoxMouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
  procedure pbPasteBoxPaint(Sender: TObject);
  procedure FormDestroy(Sender: TObject);
  procedure RgGrpFillOptionsClick(Sender: TObject);
public
  { Открытые объявления }
  MouseOrg: TPoint;    // Хранит информацию о мыши
  NextPoint: TPoint;   // Хранит информацию о мыши
  Drawing: Boolean;    // Признак выполнения рисования
  DrawType: TDrawType; // Хранит информацию о типе рисования: TDrawType
  FillSelected,        // Признак заливки фигур
  BorderSelected: Boolean;  // Признак рисования фигур без рамки
  EraseClipRect: Boolean;     // Указывает, удалять или нет
                              // вырезаемый прямоугольник
  Modified: Boolean;     // Признак модификации изображения
  FileName: String;      // Хранит имя файла изображения
  OldClipViewHwnd: Hwnd; // Хранит старое окно просмотра буфера обмена
  { Переменные вставки изображения }
  PBoxMoving: Boolean;   // Признак перемещения объекта PasteBox
  PBoxMouseOrg: TPoint;  // Хранит координаты мыши для
                         // перемещения объекта PasteBox
  PasteBitMap: TBitmap;  // Хранит растровое изображение
                         // вставленных данных
  Pasted: Boolean;       // Признак вставленных данных
  LastDot: TPoint;       // Хранит координату TPoint для рисования
                         // произвольной линии
  procedure DrawToImage(TL, BR: TPoint; PenMode: TPenMode);
  { Эта процедура рисует изображение, заданное полем DrawType
    для компонента изображения imgDrawingPad. }
  procedure SetDrawingStyle;
  { В этой процедуре устанавливаются различные стили пера и
    кисти(Pen/Brush) на основе значений, заданных с помощью
    элементов управления формы. Для установки этих значений
    используются панели инструментов и цветовая сетка. }
```

```
    procedure CopyPasteBoxToImage;
    { Эта процедура копирует данные, вставленные из системного буфера
     обмена Windows в компонент главного изображения imgDrawingPad. }
    procedure WMDrawClipBoard(var Msg: TWMDrawClipBoard);
       message WM_DRAWCLIPBOARD;
    { Этот обработчик сообщений перехватывает сообщения WM_DRAWCLIPBOARD,
      которые посылаются всем окнам, добавленным в цепочку просмотра
      буфера обмена. С помощью функции Win32 API SetClipBoardViewer()
      любое приложение может добавить себя в эту цепочку подобно тому,
      как это сделано в функции FormCreate(). }
    procedure CopyCut(Cut: Boolean);
    { Этот метод копирует часть главного изображения imgDrawingPad
      в буфер обмена Windows. }
  end;

var
  MainForm: TMainForm;

implementation
uses ClipBrd, math;

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
{ В этом методе поля формы устанавливаются равными значениям, действующим
  по умолчанию. Затем для компонента imgDrawingPad создается растровое
  изображение, на котором выполняется рисование. И, наконец, с помощью
  функции SetClipBoardViewer() данное приложение добавляется как часть
  цепочки просмотра буфера обмена Windows. Благодаря этому форме разрешается
  получать сообщения WM_DRAWCLIPBOARD, которые посылаются всем окнам
  упомянутой цепочки при модификации данных буфера обмена. }
begin
  Screen.Cursors[crMove] := LoadCursor(hInstance, 'MOVE');

  FillSelected := False;
  BorderSelected := True;

  Modified := False;
  FileName := '';
  Pasted := False;
  pbPasteBox.Enabled := False;

  // Создание растра для компонента imgDrawingPad и установка его границ
  with imgDrawingPad do
  begin
    SetBounds(0, 0, 600, 400);
    Picture.Graphic := TBitMap.Create;
    Picture.Graphic.Width := 600;
    Picture.Graphic.Height := 400;
  end;
  // Создание растрового изображения для сохранения вставленных данных
```

```
  PasteBitmap := TBitmap.Create;
  pbPasteBox.BringToFront;
  { Добавление формы в цепочку просмотра буфера обмена Windows.
    Сохранение дескриптора следующего окна в цепочке, для того чтобы его
    можно было восстановить с помощью функции Win32 API ChangeClipboardChain()
    в методе этой формы FormDestroy(). }
  OldClipViewHwnd := SetClipBoardViewer(Handle);
end;
procedure TMainForm.WMDrawClipBoard(var Msg: TWMDrawClipBoard);
begin
  { Этот метод вызывается при любом изменении данных буфера обмена.
    Поскольку в цепочку просмотра буфера обмена была добавлена главная
    форма, она получит сообщение WM_DRAWCLIPBOARD, сигнализирующее
    об изменении данных буфера обмена. }
  inherited;
  { Убедитесь, что данные, содержащиеся в буфере обмена, действительно
    являются данными растрового изображения. }
  if ClipBoard.HasFormat(CF_BITMAP) then
    mmiPaste.Enabled := True
  else
    mmiPaste.Enabled := False;
  Msg.Result := 0;
end;

procedure TMainForm.DrawToImage(TL, BR: TPoint; PenMode: TPenMode);
{ В этом методе выполняется операция рисования, заданная полем DrawType. }
begin
  with imgDrawingPad.Canvas do
  begin
    Pen.Mode := PenMode;

    case DrawType of
      dtLineDraw:
        begin
          MoveTo(TL.X, TL.Y);
          LineTo(BR.X, BR.Y);
        end;
      dtRectangle:
        Rectangle(TL.X, TL.Y, BR.X, BR.Y);
      dtEllipse:
        Ellipse(TL.X, TL.Y, BR.X, BR.Y);
      dtRoundRect:
        RoundRect(TL.X, TL.Y, BR.X, BR.Y,
          (TL.X - BR.X) div 2, (TL.Y - BR.Y) div 2);
      dtClipRect:
        Rectangle(TL.X, TL.Y, BR.X, BR.Y);
    end;
  end;
end;

procedure TMainForm.CopyPasteBoxToImage;
```

```
{ Этот метод копирует изображение, вставленное из буфера обмена Windows
  в компонент imgDrawingPad. Сначала удаляется ограничивающий прямоугольник,
  нарисованный объектом pbPasteBox, который является экземпляром компонента
  PaintBox. Затем из объекта pbPasteBox данные копируются в компонент
  imgDrawingPad в позиции, в которую перетащили объект pbPasteBox через
  компонент imgDrawingPad. Поскольку при перетаскивании части объекта
  pbPasteBox за пределы видимой области Windows не отображает ее, приходится
  не копировать содержимое канвы объекта pbPasteBox, а использовать
  канву объекта PasteBitmap. Следовательно, нужно использовать вставленный
  растр из невидимого растрового изображения. }
var
  SrcRect, DestRect: TRect;
begin
  // Сначала стираем прямоугольник, нарисованный объектом pbPasteBox
  with pbPasteBox do
  begin
    Canvas.Pen.Mode := pmNotXOR;
    Canvas.Pen.Style := psDot;
    Canvas.Brush.Style := bsClear;
    Canvas.Rectangle(0, 0, Width, Height);
    DestRect := Rect(Left, Top, Left+Width, Top+Height);
    SrcRect := Rect(0, 0, Width, Height);
  end;
  { Здесь вместо объекта pbPasteBox нужно использовать объект
    PasteBitmap, поскольку объект pbPasteBox отсечет все, выходящее
    за пределы видимой области. }
  imgDrawingPad.Canvas.CopyRect(DestRect, PasteBitmap.Canvas, SrcRect);
  pbPasteBox.Visible := False;
  pbPasteBox.Enabled := False;
  Pasted := False;  // Операция вставки выполнена
end;

procedure TMainForm.imgDrawingPadMouseDown(Sender: TObject; Button:
  TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Modified := True;
  // Удалить прямоугольник отсечения, если он был нарисован
  if (DrawType = dtClipRect) and EraseClipRect then
    DrawToImage(MouseOrg, NextPoint, pmNotXOR)
  else if (DrawType = dtClipRect) then
    EraseClipRect := True; // Восстановление разрешения на стирание
                           // прямоугольника отсечения

  { Если растр был вставлен из буфера обмена, скопируйте его на
    изображение и удалите объект PaintBox. }
  if Pasted then
    CopyPasteBoxToImage;

  Drawing := True;
  // Сохранение информации, связанной с мышью
  MouseOrg := Point(X, Y);
```

```
  NextPoint := MouseOrg;
  LastDot := NextPoint; //Обновление переменной Lastdot из-за перемещения мыши
  imgDrawingPad.Canvas.MoveTo(X, Y);
end;
procedure TMainForm.imgDrawingPadMouseMove(Sender: TObject;
  Shift: TShiftState;  X, Y: Integer);
{ В этом методе определяется предстоящая операция рисования, а затем
  либо выполняется рисование произвольной линии, либо вызывается метод
  DrawToImage для рисования заданной фигуры. }
begin
  if Drawing then
  begin
    if DrawType = dtCrooked then
    begin
      imgDrawingPad.Canvas.MoveTo(LastDot.X, LastDot.Y);
      imgDrawingPad.Canvas.LineTo(X, Y);
      LastDot := Point(X,Y);
    end
    else begin
      DrawToImage(MouseOrg, NextPoint, pmNotXor);
      NextPoint := Point(X, Y);
      DrawToImage(MouseOrg, NextPoint, pmNotXor)
    end;
  end;
  // Обновление строки состояния в соответствии с текущей позицией мыши
  stbMain.Panels[1].Text := Format('X: %d, Y: %D', [X, Y]);
end;

procedure TMainForm.imgDrawingPadMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  { Защита прямоугольника отсечения от разрушения уже имеющихся изображений. }
    if not (DrawType = dtClipRect) then
      DrawToImage(MouseOrg, Point(X, Y), pmCopy);
  Drawing := False;
end;

procedure TMainForm.sbLineClick(Sender: TObject);
begin
  // Сначала удалим прямоугольник отсечения, если это текущий тип рисования
  if DrawType = dtClipRect then
     DrawToImage(MouseOrg, NextPoint, pmNotXOR);

  { Теперь установим поле DrawType в соответствии со значением объекта
    TSpeedButton, переданным в этот метод. Значения Tag компонента
    TSpeedButton совпадают с конкретным значением TDrawType, и поэтому
    с помощью операции приведения типов полю DrawType успешно
    назначается действительное значение типа TDrawType. }
  if Sender is TSpeedButton then
    DrawType := TDrawType(TSpeedButton(Sender).Tag);
```

```
  // Позаботимся, чтобы стиль dtClipRect не стирал нарисованное ранее
  if DrawType = dtClipRect then begin
    EraseClipRect := False;
  end;
  // Установим стиль рисования
  SetDrawingStyle;
end;

procedure TMainForm.cgDrawingColorsChange(Sender: TObject);
{ Этот метод рисует прямоугольник, представляя цвета заливки и обводки
  для индикации выбора пользователем обоих цветов. Объекты pnlFgBgInner
  и pnlFgBgBorder, имеющие тип TPanels, для достижения желаемого
  эффекта располагаются один поверх другого. }
begin
  pnlFgBgInner.Color := cgDrawingColors.ForeGroundColor;
  pnlFgBgInner.Color := cgDrawingColors.BackGroundColor;
  SetDrawingStyle;
end;

procedure TMainForm.SetDrawingStyle;
{ В этом методе устанавливаются различные стили рисования на основе
  выбора, сделанного на панели pnlFillStyle (объект типа TPanel)
  для стилей заливки и обводки. }
begin
  with imgDrawingPad do
  begin
    if DrawType = dtClipRect then
    begin
      Canvas.Pen.Style := psDot;
      Canvas.Brush.Style := bsClear;
      Canvas.Pen.Color := clBlack;
    end
    else if FillSelected then
      Canvas.Brush.Style := bsSolid;
    else
      Canvas.Brush.Style := bsClear;

    if BorderSelected then
      Canvas.Pen.Style := psSolid;
    else
      Canvas.Pen.Style := psClear;

    if FillSelected and (DrawType <> dtClipRect) then
     Canvas.Brush.Color := pnlFgBgInner.Color;
    if DrawType <> dtClipRect then
      Canvas.Pen.Color := pnlFgBgInner.Color;
  end;
end;

 procedure TMainForm.mmiExitClick(Sender: TObject);
begin
```

```
  Close; // Завершение работы приложения
end;

procedure TMainForm.mmiSaveFileClick(Sender: TObject);
{ В этом методе изображение сохраняется в файле, заданном переменной
  FileName. Но если имя файла оказывается пустым, для получения имени
  файла вызывается метод SaveAs1Click.}
begin
  if FileName = '' then
    mmiSaveAsClick(nil)
  else begin
    imgDrawingPad.Picture.SaveToFile(FileName);
    stbMain.Panels[0].Text := FileName;
    Modified := False;
  end;
end;

procedure TMainForm.mmiSaveAsClick(Sender: TObject);
{ Этот метод активизирует диалоговое окно SaveDialog для получения
  имени файла, предназначенного для сохранения содержимого изображения. }
begin
  if SaveDialog.Execute then
  begin
    FileName := SaveDialog.FileName;  // Сохранение имени файла
    mmiSaveFileClick(nil)
  end;
end;

procedure TMainForm.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
{ Если пользователь попытается закрыть форму, не сохранив изображение,
  в этом методе ему будет предложено все-таки сделать это. }
var
  Rslt: Word;
begin
  CanClose := False; // Предположим промах пользователя
  if Modified then begin
    Rslt := MessageDlg('File has changed, save?',
      mtConfirmation, mbYesNOCancel, 0);
    case Rslt of
      mrYes: mmiSaveFileClick(nil);
      mrNo: ;  // Не нужны никакие действия
      mrCancel: Exit;
    end
  end;
  CanClose := True;    // Разрешение закрыть приложение
end;

procedure TMainForm.mmiNewFileClick(Sender: TObject);
{ В этом методе стирается любой рисунок, нанесенный на главное изображение.
  Однако это делается после предложения пользователю сохранить рисунок в
  файле. В случае его согласия вызывается обработчик событий mmiSaveFileClick. }
```

```
var
  Rslt: Word;
begin
  if Modified then begin
    Rslt := MessageDlg('File has changed, save?', mtConfirmation,
      mbYesNOCancel, 0);
    case Rslt of
      mrYes: mmiSaveFileClick(nil);
      mrNo: ;  // Не нужны никакие действия
      mrCancel: Exit;
    end
  end;

   with imgDrawingPad.Canvas do begin
     Brush.Style := bsSolid;
     Brush.Color := clWhite;  // Цвет clWhite – для стирания
     FillRect(ClipRect);      // Стирание изображения
     FileName := '';
     stbMain.Panels[0].Text := FileName;
   end;
   SetDrawingStyle;   // Восстановление предыдущего стиля рисования
   Modified := False;
end;

procedure TMainForm.mmiOpenFileClick(Sender: TObject);
{ В этом методе открывается растровый файл, заданный свойством
  OpenDialog.FileName. Если файл уже создан, пользователю предлагается
  сохранить файл, для чего и вызывается обработчик событий mmiSaveFileClick. }
var
  Rslt: Word;
begin

  if OpenDialog.Execute then
  begin

    if Modified then begin
      Rslt := MessageDlg('File has changed, save?',
        mtConfirmation, mbYesNOCancel, 0);
      case Rslt of
        mrYes: mmiSaveFileClick(nil);
        mrNo: ;  // Не нужны никакие действия
        mrCancel: Exit;
      end
    end;

    imgDrawingPad.Picture.LoadFromFile(OpenDialog.FileName);
    FileName := OpenDialog.FileName;
    stbMain.Panels[0].Text := FileName;
    Modified := False;
  end;
end;
```

```
procedure TMainForm.mmiEditClick(Sender: TObject);
{ Таймер используется для выяснения того, обведена ли какая-нибудь область
  главного изображения ограничивающим прямоугольником. Если да, то элементы
  меню Copy (Копировать) и Cut (Вырезать) активны. В противном случае
  они недоступны. }
var
  IsRect: Boolean;
begin
  IsRect := (MouseOrg.X <> NextPoint.X) and (MouseOrg.Y <> NextPoint.Y);
  if (DrawType = dtClipRect) and IsRect then
  begin
    mmiCut.Enabled := True;
    mmiCopy.Enabled := True;
  end
  else begin
    mmiCut.Enabled := False;
    mmiCopy.Enabled := False;
  end;
end;

procedure TMainForm.CopyCut(Cut: Boolean);
{ В этом методе копируется часть главного изображения в буфер обмена.
  Копируемая область задается ограничивающим прямоугольником на главном
  изображении. Если выбрана команда Cut, то область, заключенная
  в ограничивающий прямоугольник, стирается. }
var
   CopyBitMap: TBitmap;
   DestRect, SrcRect: TRect;
   OldBrushColor: TColor;
begin
  CopyBitMap := TBitMap.Create;
  try
    { Установить размеры копируемого растра (объекта CopyBitmap) на основе
      координат ограничивающего прямоугольника. }
    CopyBitMap.Width := Abs(NextPoint.X - MouseOrg.X);
    CopyBitMap.Height := Abs(NextPoint.Y - MouseOrg.Y);
    DestRect := Rect(0, 0, CopyBitMap.Width, CopyBitmap.Height);
    SrcRect := Rect(Min(MouseOrg.X, NextPoint.X)+1,
                    Min(MouseOrg.Y, NextPoint.Y)+1,
                    Max(MouseOrg.X, NextPoint.X)-1,
                    Max(MouseOrg.Y, NextPoint.Y)-1);
    { Копирование части главного изображения, обведенной ограничивающим
      прямоугольником в буфер обмена Windows. }
    CopyBitMap.Canvas.CopyRect(DestRect, imgDrawingPad.Canvas, SrcRect);
    { В предыдущих версиях Delphi нужно было использовать свойство
      растра Handle, чтобы обеспечить доступ к этому растру. Это требование
      было вызвано кэшированием растровых изображений. Поэтому следующая
      инструкция может быть избыточной. }
    CopyBitMap.Handle;
    // Назначение изображения буферу обмена
    ClipBoard.Assign(CopyBitMap);
```

```
    { Если была задана команда вырезки, выполняется стирание части
      главного изображения, обведенной ограничивающим прямоугольником. }
    if Cut then
      with imgDrawingPad.Canvas do
      begin
        OldBrushColor := Brush.Color;
        Brush.Color := clWhite;
        try
          FillRect(SrcRect);
        finally
          Brush.Color := OldBrushColor;
        end;
      end;
  finally
    CopyBitMap.Free;
  end;
end;

procedure TMainForm.mmiCutClick(Sender: TObject);
begin
  CopyCut(True);
end;

procedure TMainForm.mmiCopyClick(Sender: TObject);
begin
  CopyCut(False);
end;

procedure TMainForm.mmiPasteClick(Sender: TObject);
{ Этот метод выполняет вставку данных, содержащихся в буфере обмена,
  в растр вставки (объект PasteBitmap), который является "внеэкранным"
  растром. Это делается для того, чтобы пользователь мог переместить
  вставляемое изображение в любое место главного изображения. Это реализуется
  с помощью объекта pbPasteBox, экземпляра компонента TPaintBox. Когда
  пользователь размещает в желаемом месте объект вставки (pbPasteBox),
  содержимое объекта TPasteBitmap копируется в объект imgDrawingPad
  в позиции, заданной расположением объекта pbPasteBox.}
begin
  { Очистка ограничивающего прямоугольника }

  pbPasteBox.Enabled := True;
  if DrawType = dtClipRect then
  begin
    DrawToImage(MouseOrg, NextPoint, pmNotXOR);
    EraseClipRect := False;
  end;

  PasteBitmap.Assign(ClipBoard);   // Захват данных из буфера обмена
  Pasted := True;
  // Установка позиции для вставленного изображения в верхнем левом углу
  pbPasteBox.Left := 0;
```

```
  pbPasteBox.Top := 0;
  // Установка размеров объекта pbPasteBox равными размерам
  // объекта PasteBitmap
  pbPasteBox.Width := PasteBitmap.Width;
  pbPasteBox.Height := PasteBitmap.Height;

  pbPasteBox.Visible := True;
  pbPasteBox.Invalidate;
end;

procedure TMainForm.pbPasteBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
{ В этом методе образуется объект pbPasteBox типа TPaintBox в
  процессе перемещения мыши при нажатой ее левой кнопке. }
begin
  if Button = mbLeft then
  begin
    PBoxMoving := True;
    Screen.Cursor := crMove;
    PBoxMouseOrg := Point(X, Y);
  end
  else
    PBoxMoving := False;
end;

procedure TMainForm.pbPasteBoxMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
{ Этот метод перемещает объект pbPasteBox, если признак PBoxMoving
  равен True, а это значит, что пользователь удерживает левую кнопку
  мыши и перетаскивает объект PaintBox. }
begin
  if PBoxMoving then
  begin
    pbPasteBox.Left := pbPasteBox.Left + (X - PBoxMouseOrg.X);
    pbPasteBox.Top := pbPasteBox.Top + (Y - PBoxMouseOrg.Y);
  end;
end;

procedure TMainForm.pbPasteBoxMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
{ Этот метод запрещает перемещение объекта pbPasteBox, когда
  пользователь отпускает левую кнопку мыши. }
  if PBoxMoving then
  begin
     PBoxMoving := False;
     Screen.Cursor := crDefault;
  end;
  pbPasteBox.Refresh; // Перерисовка объекта pbPasteBox
end;
```

```
procedure TMainForm.pbPasteBoxPaint(Sender: TObject);
{ Область рисования образуется, когда пользователь выбирает в меню
  команду вставки (Paste). В область вставки (объект pbPasteBox класса
  TPaintBox) копируется содержимое объекта PasteBitmap, в котором хранится
  изображение, полученное из буфера обмена. Это делается для того,
  чтобы пользователь мог также перемещать этот объект по поверхности
  главного изображения. Другими словами, объект pbPasteBox можно либо
  передвигать, либо скрыть при необходимости. }
var
  DestRect, SrcRect: TRect;
begin
  // Отображение области рисования только при выборе операции вставки
  if Pasted then
  begin
    { Если область рисования больше не перемещается, то сначала копируется
      содержимое объекта PasteBitmap с помощью функции канвы CopyRect.
      Это позволяет уменьшить мерцание. }
    if not PBoxMoving then
    begin
      DestRect := Rect(0, 0, pbPasteBox.Width, pbPasteBox.Height);
      SrcRect := Rect(0, 0, PasteBitmap.Width, PasteBitmap.Height);
      pbPasteBox.Canvas.CopyRect(DestRect, PasteBitmap.Canvas, SrcRect);
    end;
    { Теперь копируем ограничивающий прямоугольник, чтобы показать,
      что pbPasteBox является движущимся объектом. При этом используется
      режим пера pmNotXOR, поскольку, когда пользователь скопирует
      содержимое объекта PaintBox на главное изображение, нужно стереть
      этот прямоугольник, но сохранить при этом его исходное содержимое. }
    pbPasteBox.Canvas.Pen.Mode := pmNotXOR;
    pbPasteBox.Canvas.Pen.Style := psDot;
    pbPasteBox.Canvas.Brush.Style := bsClear;
    pbPasteBox.Canvas.Rectangle(0, 0, pbPasteBox.Width, pbPasteBox.Height);
  end;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  // Удаление формы из цепочки буфера обмена
  ChangeClipBoardChain(Handle, OldClipViewHwnd);
  PasteBitmap.Free; // Освобождение экземпляра PasteBitmap
end;

end.
```

Как работает программа рисования

Программа рисования имеет внушительные размеры. Поскольку было бы трудно разъяснять работу программы, не видя ее текста, мы вставили обширные комментарии прямо в ее исходный код. В этом разделе описываются общие принципы функционирования программы. На рис. 8.18 показана ее главная форма.

На заметку

Обратите внимание на то, что в качестве поверхности рисования для этой программы был использован объект типа `TImage`. Имейте в виду, что это допустимо только в случае, если для изображения используется объект типа `TBitmap`.

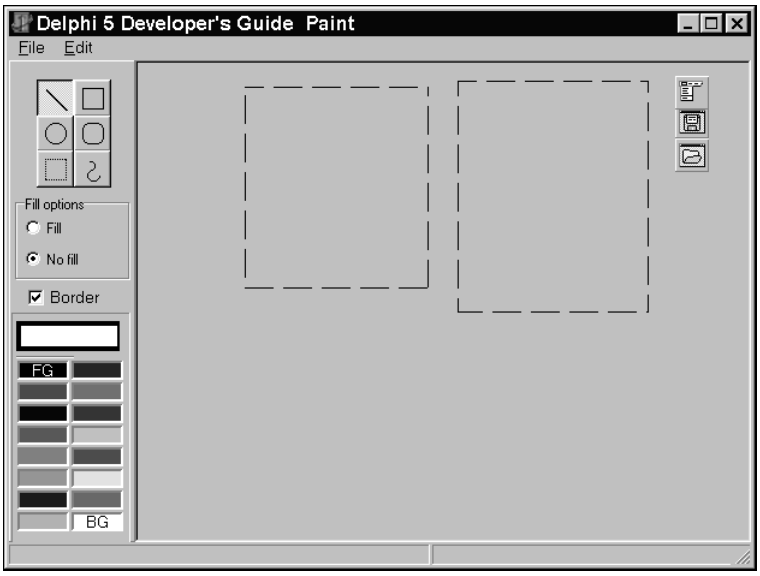


Рис. 8.18. Главная форма для программы рисования

Главная форма содержит компонент главного изображения `imgDrawingPad`, который размещается на компоненте типа `TScrollBox`. Компонент `imgDrawingPad` предоставляет поверхность, на которой пользователь выполняет рисование. Выделенная кнопка на панели инструментов формы определяет тип рисования.

Пользователь может рисовать прямые, прямоугольники, эллипсы, скругленные прямоугольники, а также линии произвольной формы. Кроме того, можно выделить любую часть главного изображения и скопировать ее в стандартный буфер обмена Windows для последующей вставки в другое приложение, которое может обрабатывать растровые данные. И точно так же эта программа рисования может принимать растровые данные из буфера обмена Windows.

Работа с панелью инструментов

Стиль заливки и тип границ (обводки) задается с помощью переключателя Fill Options. Цвет заливки и обводки устанавливаются с помощью палитры цветов (`ColorPanel`), также показанной на рис. 8.18.

Вставка растровых данных из буфера обмена

Для вставки данных из буфера обмена используется внеэкранный растр `PasteBitMap`, предназначенный для хранения вставляемых данных. А экземпляр компонента `pbPasteBox`, имеющий тип `TPaintBox`, служит для последующей перерисовки этих данных из растра

PasteBitMap. Благодаря использованию перевалочного пункта в виде компонента TPaintBox пользователь может перемещать объект pbPasteBox по всей поверхности главного изображения, чтобы определить конкретное место копирования вставляемых данных.

Присоединение к цепочке просмотра буфера обмена системы Win32

В программе рисования показано, как любое приложение может включиться в цепочку просмотра буфера обмена системы Win32. Это включение реализовано в методе FormCreate() посредством обращения к функции Win32 API SetClipboardViewer(). Функция SetClipboardViewer() принимает дескриптор окна, присоединяющего себя к этой цепочке, и возвращает дескриптор, указывающий на следующее окно в цепи. Возвращаемое значение подлежит обязательному сохранению, чтобы после закрытия этого приложения можно было восстановить предыдущее состояние цепочки с помощью метода ChangeClipboardChain(), которому передается удаляемый и сохраненный дескрипторы. Программа рисования восстанавливает цепочку в методе главной формы FormDestroy(). После присоединения к цепочке обозревателя буфера обмена приложение получает сообщения WM_DRAWCLIPBOARD при каждой модификации данных в буфере обмена. Этим фактом можно и нужно воспользоваться, перехватывая эти сообщения и делая доступным элемент меню Paste (Вставить), если измененные данные в буфере обмена являются растровыми. Реализация этих действий доверена методу WMDrawClipBoard().

Копирование растра

Операции копирования выполняются в методах CopyCut(), pbPasteBoxPaint() и CopyPasteBoxToImage(). Методом CopyCut() в буфер обмена копируется часть главного изображения, выделенная ограничивающим прямоугольником, с последующим стиранием этой области, если значение переданного параметра Cut равно True. В противном случае эта область остается нетронутой.

Метод PbPasteBoxPaint() копирует содержимое "внеэкранного" растра в объект pbPasteBox.Canvas, но поскольку пользователю разрешено передвигать по форме объект pbPasteBox, копирование (для уменьшения мерцания) происходит только в том случае, когда объект pbPasteBox не перемещается.

Метод CopyPasteBoxToImage() выполняет копирование содержимого "внеэкранного" растра в экземпляр компонента главного изображения imgDrawingPad, причем позиция копирования определяется объектом pbPasteBox.

Комментарии программы рисования

Как упоминалось выше, большинство действий, выполняемых программой, поясняется в тексте самой программы с помощью комментариев. Поэтому имеет смысл внимательно рассмотреть ее исходный код, обращая особое внимание на приведенные комментарии. Это позволит лучше понять работу программы.

Программирование анимации

В этом разделе показано, как, объединив классы Delphi 5 с функциями Win32 GDI, можно добиться анимации упрощенного изображения эльфа. В листинге 8.8 представлен текст главной формы проекта Animate.dpr, полный текст которого можно найти на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Листинг 8.8. Главная форма проекта создания анимации

```
unit MainFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, Stdctrls; AppEvents

{$R SPRITES.RES }  // Привязка растровых изображений к выполняемому файлу

type
  TSprite = class
  private
    FWidth: integer;
    FHeight: integer;
    FLeft: integer;
    FTop: integer;
    FAndImage, FOrImage: TBitMap;
  public
    property Top: Integer read FTop write FTop;
    property Left: Integer read FLeft write FLeft;
    property Width: Integer read FWidth write FWidth;
    property Height: Integer read FHeight write FHeight;
    constructor Create;
    destructor Destroy; override;
  end;

  TMainForm = class(TForm)
    mmMain:  TMainMenu
    mmiFile: TMenuItem;
    mmiSlower: TMenuItem;
    mmiFaster: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    appevMain: TApplicationEvents;
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiSlowerClick(Sender: TObject);
    procedure mmiFasterClick(Sender: TObject);
    procedure appevMainIdle(Sender: TObject; var Done: Boolean);
  private
    BackGnd1, BackGnd2: TBitMap;
    Sprite: TSprite;
    GoLeft,GoRight,GoUp,GoDown: boolean;
    FSpeed, FSpeedIndicator: Integer;
    procedure DrawSprite;
  end;
```

```
const

  BackGround = 'BACK2.BMP';

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

constructor TSprite.Create;
begin
  inherited Create;
  { Создание растров для хранения изображений эльфа, которые будут
    использованы при выполнении операции AND/OR для создания анимации. }
  FAndImage := TBitMap.Create;
  FAndImage.LoadFromResourceName(hInstance, 'AND');

  FOrImage := TBitMap.Create;
  FOrImage.LoadFromResourceName(hInstance, 'OR');

  Left := 0;
  Top := 0;
  Height := FAndImage.Height;
  Width := FAndImage.Width;

end;

destructor TSprite.Destroy;
begin
  FAndImage.Free;
  FOrImage.Free;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Создание исходного фонового изображения
  BackGnd1 := TBitMap.Create;
  with BackGnd1 do
  begin
    LoadFromResourceName(hInstance, 'BACK');
    Parent := nil;
    SetBounds(0, 0,  Width, Height);
  end;

  // Создание копии фонового изображения
  BackGnd2 :=TBitMap.Create;
  BackGnd2.Assign(BackGnd1);
  // Создание изображения эльфа
```

```
  Sprite := TSprite.Create;

  // Инициализация переменных направления
  GoRight := True;
  GoDown := True;
  GoLeft := False;
  GoUp := False;

  Fspeed :=0;
  FSpeedIndicator :=0;

  // Установка высоты и ширины области клиента формы
  ClientWidth := BackGnd1.Width;
  ClientHeight := BackGnd1.Height;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  // Освобождение всех объектов, созданных в конструкторе формы FormCreate()
  BackGnd1.Free;
  BackGnd2.Free;
  Sprite.Free;
end;

procedure TMainForm.DrawSprite;
var
  OldBounds: TRect;
begin

  // Сохранение границ эльфа в объекте OldBounds
  with OldBounds do
  begin
    Left := Sprite.Left;
    Top := Sprite.Top;
    Right := Sprite.Width;
    Bottom := Sprite.Height;
  end;

  { Теперь изменяем границы эльфа, чтобы он двигался в одном направлении,
    или изменяем направление при соприкосновении с границами формы. }
  with Sprite do
  begin
    if GoLeft then
      if Left > 0 then
         Left := Left - 1
      else begin
         GoLeft := False;
         GoRight := True;
      end;
```

```
  if GoDown then
    if (Top + Height) < self.ClientHeight then
       Top := Top + 1
    else begin
      GoDown := False;
      GoUp := True;
    end;

  if GoUp then
    if  Top > 0 then
       Top := Top - 1
    else begin
      GoUp := False;
      GoDown := True;
    end;

  if GoRight then
  if (Left + Width) < self.ClientWidth then
    Left := Left + 1
  else begin
    GoRight := False;
    GoLeft := True;
  end;
end;

{ Стираем исходное изображение эльфа на фоне BackGnd2 путем
  копирования прямоугольника из фона BackGnd1. }
with OldBounds do
  BitBlt(BackGnd2.Canvas.Handle, Left, Top, Right, Bottom,
         BackGnd1.Canvas.Handle, Left, Top, SrcCopy);

{ Теперь рисуем эльфа на "внеэкранном" растре,
  тем самым избавляясь от мерцания. }
with Sprite do
begin
  { Создадим черное пятно с силуэтом эльфа с помощью операции логического И,
    выполненной над растрами FAndImage и BackGnd2. }
  BitBlt(BackGnd2.Canvas.Handle, Left, Top, Width, Height,
         FAndImage.Canvas.Handle, 0, 0, SrcAnd);
  // Выполним заливку черного пятна исходными цветами эльфа
  BitBlt(BackGnd2.Canvas.Handle, Left, Top, Width, Height,
         FOrImage.Canvas.Handle, 0, 0, SrcPaint);
end;

{ Копируем эльфа в его новой позиции на канву формы. При этом
  используется прямоугольник, который немного больше, чем нужно для
  фигуры эльфа. Тем самым мы добиваемся эффективного стирания эльфа
  путем его перезаписи, после чего рисуем нового эльфа в новой
  позиции с помощью одного вызова функции BitBlt. }
with OldBounds do
```

```
    BitBlt(Canvas.Handle, Left - 2, Top - 2, Right + 2, Bottom + 2,
           BackGnd2.Canvas.Handle, Left - 2, Top - 2, SrcCopy);
end;

procedure TMainForm.FormPaint(Sender: TObject);
begin
  // Рисуем фоновое изображение при закрашивании формы
  BitBlt(Canvas.Handle, 0, 0, ClientWidth, ClientHeight,
         BackGnd1.Canvas.Handle, 0, 0, SrcCopy);
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.mmiSlowerClick(Sender: TObject);
begin
  Inc(FSpeedIndicator, 100);
end;

procedure TMainForm.mmiFasterClick(Sender: TObject);
begin
  if FSpeedIndicator >= 100 then
    Dec(FSpeedIndicator, 100)
  else
    FSpeedIndicator := 0;
end;

procedure TMainForm.appevMainIdle(Sender: TObject; var Done: Boolean);
begin
  if FSpeed >= FSpeedIndicator then
  begin
    DrawSprite;
    FSpeed := 0;
  end
  else
    inc(FSpeed);

  { Допускает вызов OnIdle, даже когда сообщения в очереди
    событий приложения отсутствуют }
  Done := False;
end;

end.
```

Анимационный проект состоит из фонового изображения и нарисованного на нем эльфа в виде летающего блюдца, которое перемещается в пределах области клиента фона. Фон представлен растровым изображением неба с разбросанными по нему звездами (рис. 8.19).

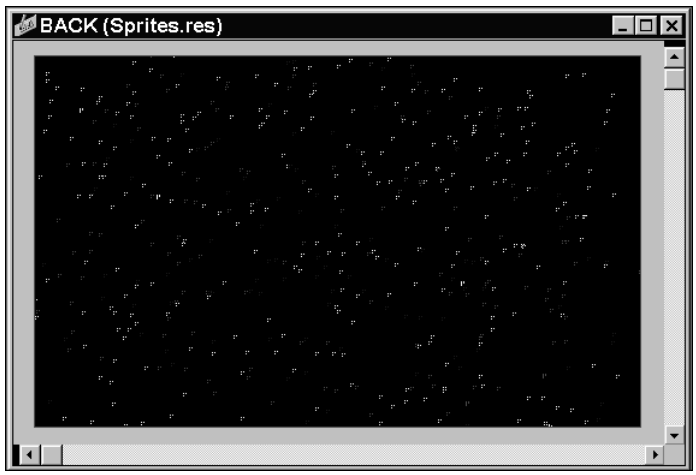


Рис. 8.19. Фон анимационного проекта

Эльф составлен из двух растров размером 64×32. О них речь пойдет ниже, а пока рассмотрим, что происходит в программе.

В приведенном модуле определяется класс `TSprite`, который содержит поля, предназначенные для хранения позиций эльфа на изображении фона, и два объекта типа `TBitmap` для хранения растровых изображений эльфа. Конструктор `TSprite.Create` создает оба экземпляра класса `TBitmap` и загружает в них реальные растры. Оба растровых изображения эльфа и фоновый растр содержатся в файле ресурсов, который привязывается к проекту путем включения в основной модуль следующей инструкции:

```
{ $R SPRITES.RES }
```

После загрузки растра устанавливаются границы изображения эльфа. Деструктор `TSprite.Done` освобождает оба экземпляра растра.

Главная форма содержит два объекта типа `TBitmap`, объект `TSprite` и индикаторы направлений, задающие линию движения эльфа. Кроме того, в главной форме определен метод `DrawSprite()`, предназначенный для рисования изображения эльфа. Компонент `TApplicationEvents` является новым компонентом, он появился только в Delphi 5 и позволяет связываться с событиями объекта приложения. В прежних версиях Delphi подобные действия можно было осуществлять только во время выполнения программы. Теперь, благодаря этому компоненту, все назначения для событий объекта `Application` можно выполнить на этапе проектирования. В данном случае компонент `TApplicationEvents` используется для получения доступа к событию `OnIdle` объекта `Application`.

Обратите внимание на то, что для управления скоростью анимации применяются две закрытые переменные — `FSpeed` и `FSpeedIndicator`. Они используются в методе `DrawSprite()` для замедления процесса анимации на более быстродействующих компьютерах.

Обработчик событий `FormCreate()` создает оба экземпляра класса `TBitmap` и загружает в каждый из них одно и то же растровое изображение (зачем — разберемся чуть ниже). Затем создается экземпляр класса `TSprite` и устанавливаются значения индикаторов направлений. Наконец, обработчик событий формы `FormCreate()` изменяет размеры формы в соответствии с размерами фонового изображения.

Метод FormPaint() выполняет рисование на канве фона BackGnd1, а метод FormDestroy() освобождает экземпляры классов TBitmap и TSprite.

Метод MyIdleEvent() вызывает метод DrawSprite(), который перемещает и рисует эльфа на существующем фоне. Именно этот метод выполняет бо́льшую часть всей работы. Метод applevMainIdle() вызывается всякий раз, когда приложение переходит в состояние ожидания, т.е. когда пользователь не выполняет никаких действий, на которые приложению следовало бы отреагировать.

Метод DrawSprite() изменяет расположение эльфа по отношению к фону. Для этого требуется выполнить немало инструкций — ведь сначала нужно стереть старое изображение эльфа, а затем нарисовать его на новом месте, сохранив цвет фона вокруг реального изображения эльфа. Кроме того, метод DrawSprite() должен выполнить эти действия без мерцания.

Для достижения поставленных целей процесс рисования выполняется на "внеэкранном" растре BackGnd2. Растры BackGnd2 и BackGnd1 являются точными копиями фонового изображения, однако BackGnd1 никогда не модифицируется (поэтому его можно назвать чистой копией фона). По завершении рисования модифицированная область растра BackGnd2 копируется на канву формы. Это позволяет за одно обращение к функции BitBlt() выполнить как стирание на канве формы старого изображения, так и рисование эльфа в новой позиции. Какие же операции выполняются с растром BackGnd2?

Во-первых, из BackGnd1 в BackGnd2 копируется прямоугольный участок, превышающий по своим размерам область, занимаемую самим эльфом. Тем самым гарантируется стирание изображения эльфа с растра BackGnd2. После этого растр FAndImage копируется в BackGnd2 в его новой позиции с помощью поразрядной операции AND (логическое И). Это приводит к созданию черного пятна с силуэтом эльфа, но с сохранением цветов в области растра BackGnd2, окружающей черный силуэт. Растр FAndImage показан на рис. 8.20.

На рис. 8.20 эльф представлен черными пикселями, а изображение вокруг эльфа состоит из белых пикселей. Черный цвет имеет значение, равное 0, а белый — 1. В табл. 8.5 и 8.6 приведены результаты выполнения операции AND соответственно с черным и белым цветами.

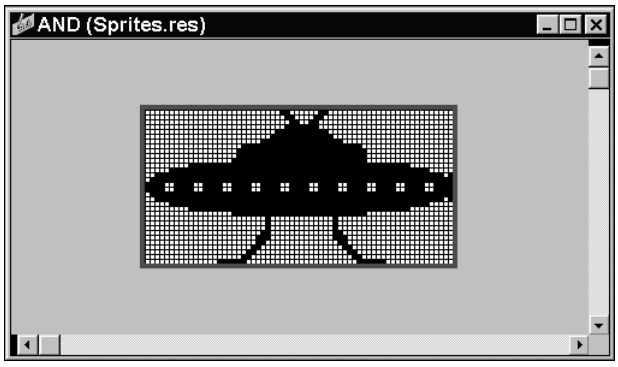


Рис. 8.20. Растр FAndImage для эльфа

Таблица 8.5. Операция AND с черным цветом

Фон	Значение	Цвет
BackGnd2	1001	Некоторый цвет
FAndImage	0000	Черный
Результат	0000	Черный

Таблица 8.6. Операция AND с белым цветом

Фон	Значение	Цвет
BackGnd2	1001	Некоторый цвет
FAndImage	1111	Белый
Результат	1001	Некоторый цвет

Эти таблицы показывают, как выполнение операции логического И приводит к зачернению области, занимаемой эльфом на растре BackGnd2. В табл. 8.5 столбец "Значение" представляет цвет пикселя. Если пиксель на растре BackGnd2 содержит некоторый произвольный цвет, то объединение этого цвета с черным при использовании оператора AND заставит данный пиксель полностью почернеть. Аналогичная операция, выполненная над тем же цветом и абсолютно белым его "коллегой" никак не отразится на исходном цвете, как видно в табл. 8.6. А поскольку цвет фона, на котором находился эльф в растре FAndImage, был белым, то пиксели на растре BackGnd2 копируются без изменения их цветов.

После копирования растра FAndImage в объект BackGnd2 растр FOrImage должен быть скопирован в то же самое место растра BackGnd2, чтобы заполнить черное пятно, созданное объединением растра FAndImage с реальными цветами эльфа. Растр FOrImage также имеет прямоугольник, окружающий реальное изображение эльфа. И вновь мы сталкиваемся с задачей получения цветов эльфа для растра BackGnd2 и одновременным сохранением цветов этого растра в области, окружающей эльф. Это достигается путем объединения растров FOrImage и BackGnd2 с использованием оператора OR (логическое ИЛИ). Растр FOrImage показан на рис. 8.21.

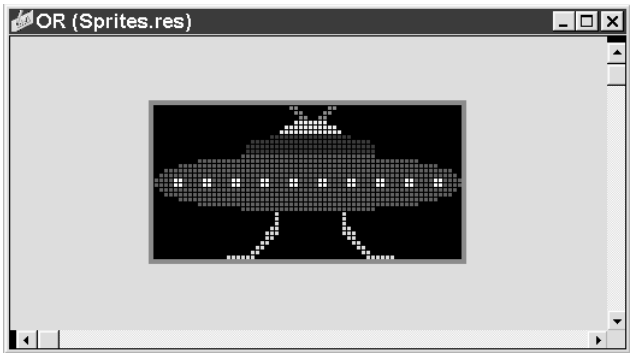


Рис. 8.21. Растр FOrImage

Обратите внимание на то, что область, окружающая изображение эльфа, окрашена в черный цвет. В табл. 8.7 показаны результаты выполнения операции ИЛИ с растрами FOrImage и BackGnd2.

Таблица 8.7. Операция OR с черным цветом

Фон	Значение	Цвет
BackGnd2	1001	Некоторый цвет
FOrImage	0000	Черный
Результат	1001	Некоторый цвет

Из табл. 8.7 следует, что если растр `BackGnd2` содержит произвольный цвет, то после операции логического сложения с черным цветом останется тот же цвет растра `BackGnd2`.

Напомним, что все рисование выполняется на "внеэкранном" растре. По завершении рисования достаточно только одного обращения к функции `BitBlt()`, чтобы стереть и скопировать изображение эльфа.

В описанном способе создания анимации нет ничего необычного. Вам предлагается самим расширить функциональные возможности класса, связанные с перемещением и рисованием на родительской канве.

Шрифты повышенной сложности

Несмотря на то что с помощью библиотеки VCL можно относительно легко манипулировать шрифтами, в ней не предусмотрено столь обширного набора функций отображения шрифтов, какими располагает интерфейс Win32 API. В этом разделе предоставлены базовые сведения о шрифтах и методах манипулирования ими, предоставляемых интерфейсом Win32 API.

Типы шрифтов в системе Win32

Существует два основных типа шрифтов в системе Win32: шрифты GDI и шрифты устройств. Первые сохраняются в файлах ресурсов и имеют расширение `.fon` (для растровых и векторных шрифтов) или `.tot` и `.ttf` (для шрифтов TrueType). Вторые — это специальные шрифты, предназначенные для таких конкретных устройств, как принтеры. Использование в интерфейсе Win32 шрифтов устройства для печати текста снимает с пользователя всякую заботу о печати символов заданным шрифтом — ему нужно лишь послать на устройство символы ASCII, а все остальные проблемы решит само устройство. В случае же использования шрифтов GDI интерфейс Win32 преобразует шрифт в растр или выполняет функцию GDI для рисования заданного шрифта, а это отнимает больше времени, чем использование шрифтов устройства. Но несмотря на бо́льшую скорость последние слишком зависят от конкретного устройства.

Основные элементы шрифтов

Прежде чем научиться пользоваться различными шрифтами в системе Win32, следует познакомиться с терминами и элементами, с ними связанными.

Гарнитура семейства и единицы измерения шрифтов

Представьте себе шрифт только как некую картинку или знак, определяющий символ. Этот символ имеет две характеристики: гарнитуру и размер.

В системе Win32 *гарнитура* шрифта связывается со стилем шрифта и его размером. Возможно, лучшее описание понятия гарнитуры дается в справочном файле системы Win32: "Гарнитура — это семейство символов, имеющих общие характеристики дизайна. Например, Courier является общеупотребительной гарнитурой. Шрифт — это семейство символов, имеющих одну и ту же гарнитуру и размер".

В Win32 различные гарнитуры делятся на пять семейств шрифтов: Decorative, Modern, Roman, Script и Swiss. Отличительными чертами этих шрифтов являются *засечка* и *толщина штриха*.

Засечка — это маленькая линия в начале или конце главных линий шрифта, которые придают ему законченный вид. *Штрих* — это основная линия, определяющая шрифт. Эти две особенности показаны на рис. 8.22.

Рис. 8.22. Засечка и штрих

Различные семейства шрифтов и их типичные представители приведены в табл. 8.8.

Таблица 8.8. Семейства шрифтов и их типичные представители

Семейство шрифтов	Типичные шрифты
Decorative	Недавно выпушенные шрифты: Old English
Modern	Шрифты с постоянной толщиной штриха, имеющие или не имеющие засечки: Pica, Elite, Courier New
Roman	Шрифты с переменной толщиной штриха и засечками: Times New Roman, New Century SchoolBook
Script	Шрифты, которые выглядят как рукописные: Script, Cursive
Swiss	Шрифты с переменной толщиной штриха, но без засечек: Arial, Helvetica

Размер шрифта определяется в пунктах. *Пункт* равен 1/72 дюйма. Высота шрифта складывается из надстрочного и подстрочного элементов, обозначенных на рис. 8.23 значениями `tmAscent` и `tmDescent` соответственно. На том же рисунке показаны и другие параметры, которые также учитываются при определении размеров символов.

Символы располагаются в символьных ячейках — окружающей символ области, заполненной пустым белым пространством. Следует иметь в виду, что под размерами символов часто понимаются как размеры самого значка символа (его видимой части), так и размеры его ячейки. В одних случаях речь может идти только о значке или о ячейке, в других — сразу об обоих этих элементах.

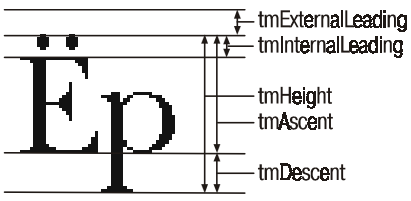


Рис. 8.23. Значения размерных параметров символов

В табл. 8.9 разъясняется значение различных размерных параметров символов.

Таблица 8.9. Размерные параметры символов

Параметр	Значение
Интерлиньяж (External leading)	Расстояние между линиями текста
Плечо (Internal leading)	Разница между высотой значка символа и высотой ячейки
Высота надстрочного элемента	Расстояние от базовой линии до вершины ячейки символа
Высота подстрочного элемента	Расстояние от базовой линии до основания ячейки символа
Размер в пунктах	Высота символа минус значение `tmInternalLeading`
Высота	Сумма значений надстрочного, подстрочного элементов и плеча
Базовая линия	Линия, на которой располагаются символы

Категории шрифтов GDI

Существует три отдельные категории шрифтов GDI: растровые, векторные (которые также называются штриховыми) и шрифты TrueType. Первые две категории существовали и в более старых версиях Windows, а третья впервые появилась только в Windows 3.1.

Растровые шрифты

Растровые шрифты представляют собой битовые матрицы (растры), предусмотренные для конкретного разрешения, или *коэффициента сжатия* (отношение высоты и ширины пикселя данного устройства), и размера шрифта. Поскольку эти шрифты создаются с определенным размером, система Win32 может синтезировать на их основе новый шрифт требуемого размера, но только в том случае, когда из меньшего по размеру шрифта нужно сделать больший. Обратный синтез невозможен, поскольку генерация шрифтов осуществляется в системе Win32 путем удвоения строк и столбцов, составляющих исходный растр шрифта. Растровыми шрифтами удобно пользоваться, когда требуемый размер относится к числу исходных размеров. В этом случае они хорошо выглядят и имеют большую скорость отображения. Однако их внешний вид значительно ухудшается при увеличении размера, как показано на рис. 8.24, на котором приведен пример символа шрифта System системы Win32.

Векторные шрифты

Векторные шрифты, в противоположность растровым, генерируются в системе Win32 с помощью набора линий, создаваемых функциями GDI. Они отличаются гораздо лучшими результатами масштабирования, но для них характерна более низкая плотность при отображении (для кого-то этот фактор может служить плюсом, а для кого-то — минусом). Кроме того, быстродействие векторных шрифтов гораздо ниже по сравнению с растровыми. К векторным шрифтам лучше всего обращаться при использовании плоттеров, но они не подойдут для использования в интерфейсах пользователей. Типичный векторный шрифт показан на рис. 8.25.

Шрифты TrueType

По-видимому, чаще всего предпочтение отдают шрифтам TrueType, поскольку им по силам практически любой стиль и размер (без ущерба для внешнего вида). Система Win32 отображает шрифты TrueType с помощью коллекции точек и алгоритмов, описывающих контур шрифта. Благодаря этим алгоритмам можно изменить масштабированный контур, чтобы улучшить внешний вид шрифта при различных разрешениях. Пример шрифта TrueType показан на рис. 8.26.

Рис. 8.24. Растровый шрифт

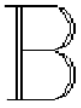


Рис. 8.25. Векторный шрифт

Рис. 8.26. Шрифт TrueType

Отображение различных типов шрифтов

В предыдущих разделах были изложены общие концепции, касающиеся шрифтов Windows. Если вы заинтересованы в более подробном изучении этой темы, обратитесь к интерактивной справочной системе Win32 (раздел “Fonts Overview”). А теперь настало время узнать, как использовать функции Win32 API и структуры Delphi 5 для создания и отображения шрифтов любой формы и размера.

Пример создания шрифта

В приведенном ниже примере демонстрируется процесс реализации различных типов шрифтов в Windows. В этом проекте также показано, как получить информацию о конкретном шрифте. Проект называется `MakeFont.dpr` — текст его находится на компакт-диске, прилагаемом ко второму тому (см. также `www.williamspublishing.com`.).

Как работает этот проект

На главной форме выбираются различные атрибуты шрифтов, которые предполагается использовать при создании нового шрифта. При каждом изменении значения любого из атрибутов шрифта новый вариант его начертания отображается в компоненте `TPaintBox`. (Все компоненты привязаны к обработчику события `FontChanged()` через их события `OnChange` или `OnClick`.) Кроме того, можно просматривать информацию о любом шрифте, щелкнув на кнопке Font Information (Информация о шрифте). На рис. 8.27 показана главная форма этого проекта. В листинге 8.9 содержится исходный текст модуля главной формы проекта.

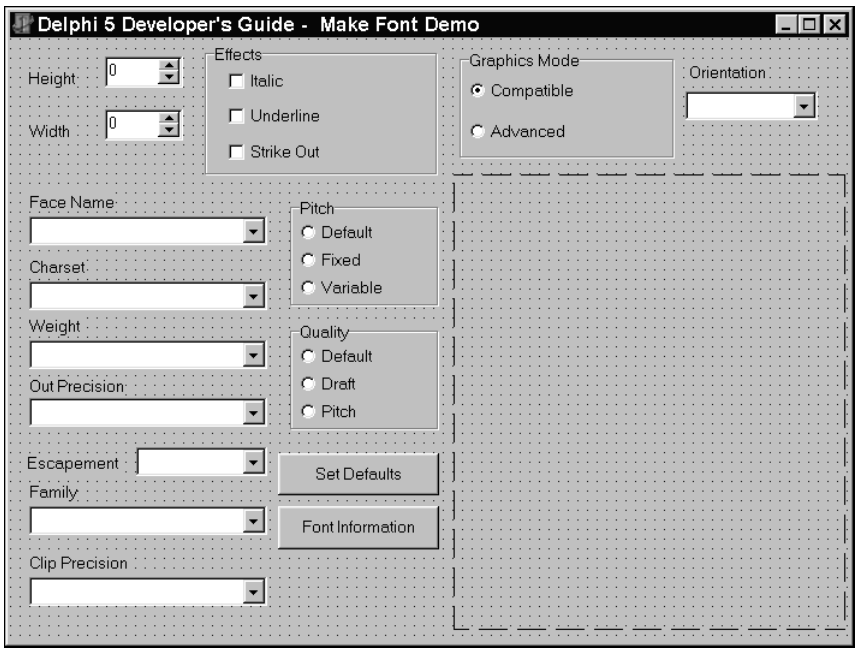


Рис. 8.27. Главная форма создания шрифта

Листинг 8.9. Проект создания шрифта

```
unit MainFrm;

interface

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

const

// Массив значений TLOGFONT.lfCharSet
CharSetArray: array[0..4] of byte = (ANSI_CHARSET, DEFAULT_CHARSET,
   SYMBOL_CHARSET, SHIFTJIS_CHARSET, OEM_CHARSET);

// Массив значений TLOGFONT.lfWeight
WeightArray: array[0..9] of integer =
  (FW_DONTCARE, FW_THIN, FW_EXTRALIGHT, FW_LIGHT, FW_NORMAL, FW_MEDIUM,
   FW_SEMIBOLD, FW_BOLD, FW_EXTRABOLD, FW_HEAVY);

// Массив значений TLOGFONT.lfOutPrecision
OutPrecArray: array[0..7] of byte = (OUT_DEFAULT_PRECIS,
  OUT_STRING_PRECIS, OUT_CHARACTER_PRECIS, OUT_STROKE_PRECIS,
  OUT_TT_PRECIS, OUT_DEVICE_PRECIS, OUT_RASTER_PRECIS,
  OUT_TT_ONLY_PRECIS);

// Массив значений четырех старших разрядов TLOGFONT.lfPitchAndFamily
FamilyArray: array[0..5] of byte = (FF_DONTCARE, FF_ROMAN,
  FF_SWISS, FF_MODERN, FF_SCRIPT, FF_DECORATIVE);

// Массив значений двух младших разрядов TLOGFONT.lfPitchAndFamily
PitchArray: array[0..2] of byte = (DEFAULT_PITCH, FIXED_PITCH,
  VARIABLE_PITCH);

// Массив значений TLOGFONT.lfClipPrecision
ClipPrecArray: array[0..6] of byte = (CLIP_DEFAULT_PRECIS,
  CLIP_CHARACTER_PRECIS, CLIP_STROKE_PRECIS, CLIP_MASK, CLIP_LH_ANGLES,
  CLIP_TT_ALWAYS, CLIP_EMBEDDED);

// Массив значений TLOGFONT.lfQuality
QualityArray: array[0..2] of byte = (DEFAULT_QUALITY, DRAFT_QUALITY,
  PROOF_QUALITY);

type

  TMainForm = class(TForm)
    lblHeight: TLabel;
    lblWidth: TLabel;
    gbEffects: TGroupBox;
    cbxItalic: TCheckBox;
```

```
    cbxUnderline: TCheckBox;
    cbxStrikeOut: TCheckBox;
    cbWeight: TComboBox;
    lblWeight: TLabel;
    lblEscapement: TLabel;
    cbEscapement: TComboBox;
    pbxFont: TPaintBox;
    cbCharSet: TComboBox;
    lblCharSet: TLabel;
    cbOutPrec: TComboBox;
    lblOutPrecision: TLabel;
    cbFontFace: TComboBox;
    rgPitch: TRadioGroup;
    cbFamily: TComboBox;
    lblFamily: TLabel;
    lblClipPrecision: TLabel;
    cbClipPrec: TComboBox;
    rgQuality: TRadioGroup;
    btnSetDefaults: TButton;
    btnFontInfo: TButton;
    lblFaceName: TLabel;
    rgGraphicsMode: TRadioGroup;
    lblOrientation: TLabel;
    cbOrientation: TComboBox;
    seHeight: TSpinEdit;
    seWidth: TSpinEdit;
    procedure pbxFontPaint(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure btnFontInfoClick(Sender: TObject);
    procedure btnSetDefaultsClick(Sender: TObject);
    procedure rgGraphicsModeClick(Sender: TObject);
    procedure cbEscapementChange(Sender: TObject);
    procedure FontChanged(Sender: TObject);
  private
    { Закрытые объявления }
    FLogFont: TLogFont;
    FHFont:   HFont;
    procedure MakeFont;
    procedure SetDefaults;
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation
uses FontInfoFrm;

{$R *.DFM}
```

```
procedure TMainForm.MakeFont;
begin
  // Очищаем содержимое FLogFont
  FillChar(FLogFont, sizeof(TLogFont), 0);
  // Устанавливаем поля TLOGFONT
  with FLogFont do
  begin
    lfHeight         := StrToInt(seHeight.Text);
    lfWidth          := StrToInt(seWidth.Text);
    lfEscapement     :=
      StrToInt(cbEscapement.Items[cbEscapement.ItemIndex]);
    lfOrientation    :=
      StrToInt(cbOrientation.Items[cbOrientation.ItemIndex]);
    lfWeight         := WeightArray[cbWeight.ItemIndex];
    lfItalic         := ord(cbxItalic.Checked);
    lfUnderline      := ord(cbxUnderLine.Checked);
    lfStrikeOut      := ord(cbxStrikeOut.Checked);
    lfCharSet        := CharSetArray[cbCharset.ItemIndex];
    lfOutPrecision   := OutPrecArray[cbOutPrec.ItemIndex];
    lfClipPrecision  := ClipPrecArray[cbClipPrec.ItemIndex];
    lfQuality        := QualityArray[rgQuality.ItemIndex];
    lfPitchAndFamily := PitchArray[rgPitch.ItemIndex] or
      FamilyArray[cbFamily.ItemIndex];
    StrPCopy(lfFaceName, cbFontFace.Items[cbFontFace.ItemIndex]);
  end;
  // Считываем требуемый шрифт
  FHFont := CreateFontIndirect(FLogFont);
  // Присваиваем значение дескриптору Font.Handle
  pbxFont.Font.Handle := FHFont;
  pbxFont.Refresh;
end;

procedure TMainForm.SetDefaults;
begin
  // Устанавливаем элементы управления в исходное состояние для ALogFont
  seHeight.Text           := '0';
  seWidth.Text            := '0';
  cbxItalic.Checked       := False;
  cbxStrikeOut.Checked    := False;
  cbxUnderline.Checked    := False;
  cbWeight.ItemIndex      := 0;
  cbEscapement.ItemIndex  := 0;
  cbOrientation.ItemIndex := 0;
  cbCharset.ItemIndex     := 1;
  cbOutPrec.Itemindex     := 0;
  cbFamily.ItemIndex      := 0;
  cbClipPrec.ItemIndex    := 0;
  rgPitch.ItemIndex       := 0;
  rgQuality.ItemIndex     := 0;
  // Заполняем комбинированный список CBFontFace экранными шрифтами
  cbFontFace.Items.Assign(Screen.Fonts);
```

```
  cbFontFace.ItemIndex := cbFontFace.Items.IndexOf(Font.Name);
end;

procedure TMainForm.pbxFontPaint(Sender: TObject);
begin
  with pbxFont do
  begin
    { Заметьте, что в Windows 95 графические режимы всегда определяются
      значением GM_COMPATIBLE, в то время как режим GM_ADVANCED
      распознается только в Windows NT. }
    case rgGraphicsMode.ItemIndex of
      0: SetGraphicsMode(pbxFont.Canvas.Handle, GM_COMPATIBLE);
      1: SetGraphicsMode(pbxFont.Canvas.Handle, GM_ADVANCED);
    end;
    Canvas.Rectangle(2, 2, Width-2, Height-2);
    // Записываем имя шрифта
    Canvas.TextOut(Width div 2, Height div 2, CBFontFace.Text);
  end;
end;
procedure TMainForm.FormActivate(Sender: TObject);
begin
  SetDefaults;
  MakeFont;
end;

procedure TMainForm.btnFontInfoClick(Sender: TObject);
begin
  FontInfoForm.ShowModal;
end;

procedure TMainForm.btnSetDefaultsClick(Sender: TObject);
begin
  SetDefaults;
  MakeFont;
end;

procedure TMainForm.rgGraphicsModeClick(Sender: TObject);
begin
  cbOrientation.Enabled := rgGraphicsMode.ItemIndex = 1;
  if not cbOrientation.Enabled then
    cbOrientation.ItemIndex := cbEscapement.ItemIndex;
  MakeFont;
end;

procedure TMainForm.cbEscapementChange(Sender: TObject);
begin
  if not cbOrientation.Enabled then
    cbOrientation.ItemIndex := cbEscapement.ItemIndex;
end;
```

```
procedure TMainForm.FontChanged(Sender: TObject);
begin
  MakeFont;
end;

end.
```

В файле MAINFORM.PAS присутствует несколько определений массивов, сопровождающихся краткими пояснениями. Пока заметьте, что для формы объявлены две закрытые (private) переменные: FLogFont и FHFont. Переменная FLogFont имеет тип TLOGFONT и представляет собой структуру записей, используемую для описания создаваемого шрифта. Переменная FHFont содержит дескриптор для создаваемого шрифта. Закрытый (private) метод MakeFont() предназначен для создания шрифта путем первоначального заполнения структуры FLogFont значениями, определяемыми компонентами главной формы, с последующей передачей этой структуры функции Win32 GDI CreateFontIndirect(), которая возвращает дескриптор для нового шрифта. Прежде чем продолжить, рассмотрим структуру TLOGFONT.

Структура TLOGFONT

Как указывалось выше, структура TLOGFONT используется для определения шрифта, который требуется создать. Эта структура определяется в модуле WINDOWS следующим образом:

```
TLogFont = record
  lfHeight: Integer;
  lfWidth: Integer;
  lfEscapement: Integer;
  lfOrientation: Integer;
  lfWeight: Integer;
  lfItalic: Byte;
  lfUnderline: Byte;
  lfStrikeOut: Byte;
  lfCharSet: Byte;
  lfOutPrecision: Byte;
  lfClipPrecision: Byte;
  lfQuality: Byte;
  lfPitchAndFamily: Byte;
  lfFaceName: array[0..lf_FaceSize - 1] of Char;
end;
```

Задача состоит в том, чтобы поместить требуемые значения в те поля структуры TLOGFONT, которые определяют желаемые атрибуты. Все поля представляют различные типы атрибутов. По умолчанию большинство полей можно установить равными нулю, что и делается с помощью кнопки Set Defaults (Установить исходные значения) на главной форме. В этом случае система Win32 выбирает атрибуты для шрифта по собственному усмотрению и возвращает такие значения, которые считает нужными. Основное правило гласит: чем больше полей вы заполните, тем бо́льшую возможность для настройки стиля шрифта получите. В приведенном ниже списке разъясняется, что именно представляет каждое поле структуры TLOGFONT. Некоторым полям в качестве значения можно присвоить константы, определенные в модуле WINDOWS. За полным описанием всех этих значений обращайтесь к справочной системе Win32, поскольку здесь приводятся только самые общеупотребительные из них.

Значение поля	Описание
lfHeight	Высота шрифта. Значение больше нуля означает высоту ячейки, меньше нуля — высоту значка (высота ячейки минус плечо). Нулевое значение позволяет системе Win32 определить высоту по собственному усмотрению
lfWidth	Средняя ширина шрифта. Нулевое значение позволяет системе Win32 выбрать ширину по собственному усмотрению
lfEscapement	Угол (в десятых долях градуса) поворота базовой линии шрифта, т.е. линии, на которой располагаются символы. В Windows 95/98 строка текста и отдельные символы рисуются с использованием одного и того же угла, т.е. значения lfEscapement и lfOrientation одинаковы. В Windows NT текст рисуется независимо от угла ориентации каждого символа в текстовой строке. Чтобы добиться этого, нужно с помощью функции интерфейса Win32 GDI SetGraphicsMode( ) установить графический режим для устройства равным значению GM_ADVANCED. По умолчанию графический режим установлен равным GM_COMPATIBLE, что заставляет Windows NT вести себя подобно Windows 95. Этот эффект действует только в случае шрифтов TrueType
lfOrientation	Позволяет задавать угол, под которым можно рисовать отдельные символы. В Windows 95/98 это значение совпадает со значением поля lfEscapement. В Windows NT эти значения могут быть различны (см. lfEscapement)
lfWeight	Это значение оказывает влияние на плотность шрифта. В модуле WINDOWS определено несколько констант для этого поля, например FW_BOLD и FW_NORMAL. При установке константы FW_DONTCARE система Win32 выбирает толщину по собственному усмотрению
lfItalic	Ненулевое значение означает курсив, а нулевое — не курсив
lfUnderline	Ненулевое значение — подчеркивание, а нулевое — отсутствие подчеркивания
lfStrikeOut	Ненулевое значение — перечеркивание, а нулевое — отсутствие перечеркивания
lfCharSet	В системе Win32 определены следующие наборы символов: ANSI_CHARSET=0, DEFAULT_CHARSET=1, SYMBOL_CHARSET=2, SHIFTJIS_CHARSET=128 и OEM_CHARSET=255. По умолчанию используется DEFAULT_CHARSET
lfOutPrecision	Этим значением определяется, как система Win32 должна сопоставлять запрашиваемый размер и характеристики шрифта с реальным шрифтом. Используйте константу TT_ONLY_PRECIS для указания применения только шрифтов TrueType. Другие константы определены в модуле WINDOWS
lfClipPrecision	Это значение определяет, как система Win32 отсекает символы снаружи области отсечения. Константа CLIP_DEFAULT_PRECIS позволит системе Win32 сделать выбор без вашего участия
lfQuality	Это значение определяет качество шрифта — в том смысле, как интерфейс GDI будет его рисовать. С помощью константы DEFAULT_QUALITY можно предоставить выбор системе Win32. В противном случае используйте константу PROOF_QUALITY или DRAFT_QUALITY
lfPitchAndFamily	В двух младших разрядах указывается тип шрифта, а в четырех старших — семейство. Семейства перечислены в табл. 8.8
lfFaceName	Имя гарнитуры шрифта

В процедуре MakeFont() используются значения, определенные в разделе const модуля MainForm.pas. Эти массивы констант содержат различные заранее определенные значения констант для структуры TLOGFONT. Эти значения расположены в том же порядке, что и элементы в комбинированных списках (компонентах TComboBox) главной формы. Например, порядок следования элементов в списке семейства шрифтов (экземпляр CBFamily компонента TComboBox) совпадает с порядком значений в массиве FamilyArray. Благодаря этому сокращается количество программных инструкций, требуемых для заполнения структуры TLOGFONT. Первая строка в функции MakeFont()

```
fillChar(FLogFont, sizeof(TLogFont), 0);
```

очищает структуру FLogFont до записи в нее каких бы то ни было значений. А после заполнения этой структуры в строке

```
FHFont := CreateFontIndirect(FLogFont)
```

вызывается функция Win32 API CreateFontIndirect(), которой структура FLogFont передается как параметр. Функция возвращает дескриптор для запрашиваемого шрифта. Значение этого дескриптора присваивается свойству Handle экземпляра pbxFont.Font системного компонента TPaintBox. Перед выполнением присвоения подпрограммы Delphi 5 выполняют уничтожение предыдущего шрифта компонента TPaintBox. После присвоения выполняется обновление (перерисовка) объекта pbxFont путем вызова его метода Refresh().

Метод SetDefaults() инициализирует структуру TLOGFONT стандартными значениями. Этот метод вызывается при создании главной формы, а также в тех случаях, когда пользователь щелкает на кнопке Set Defaults. Результат одного из экспериментов с рассматриваемым проектом, выполняемых для демонстрации различных эффектов, которые можно получить при создании шрифтов, показан на рис. 8.28.

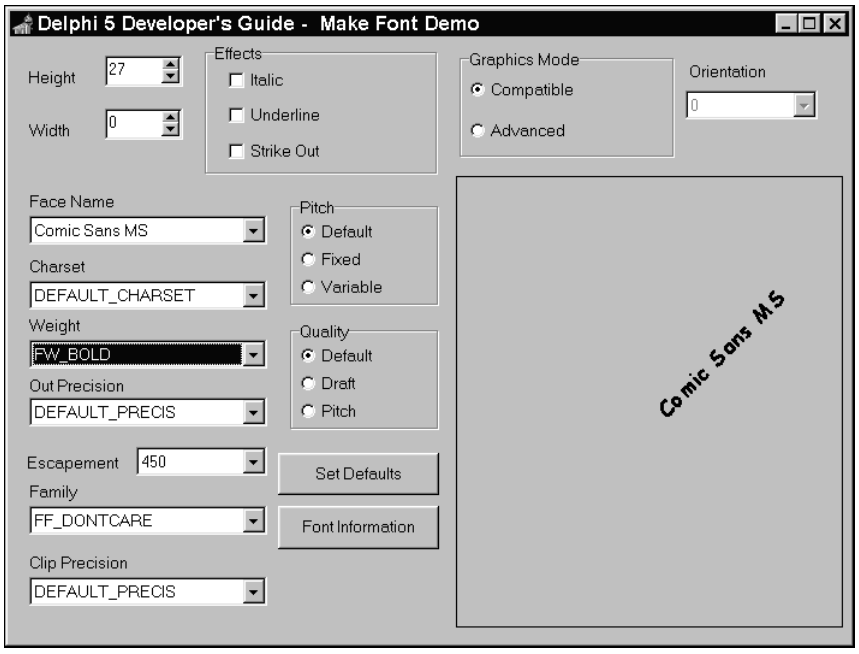


Рис. 8.28. Изображение повернутого шрифта

Отображение информации о шрифтах

По щелчку на кнопке главной формы Font Information (Информация о шрифте) вызывается метод класса FontInfoForm, который отображает информацию о выбранном шрифте. В процессе задания атрибутов шрифта в структуре TLOGFONT система Win32 старается найти шрифт, который наилучшим образом соответствует запрашиваемому. При этом вполне возможно, что шрифт, который вернет вам функция CreateFontIndirect(), будет иметь атрибуты, совершенно отличные от запрашиваемых. Метод FontInfoForm позволяет просмотреть атрибуты выбранного шрифта и для считывания запрошенной информации использует функцию Win32 API GetTextMetrics().

Функции GetTextMetrics() передается два параметра: дескриптор контекста устройства, шрифт которого требуется проанализировать, и ссылка на еще одну структуру системы Win32 — TTEXTMETRIC. Задача функции GetTextMetrics() — обновить структуру TTEXTMETRIC информацией о заданном шрифте. В модуле WINDOWS запись TTEXTMETRIC определена следующим образом:

```
TTextMetric = record
   tmHeight: Integer;
   tmAscent: Integer;
   tmDescent: Integer;
   tmInternalLeading: Integer;
   tmExternalLeading: Integer;
   tmAveCharWidth: Integer;
   tmMaxCharWidth: Integer;
   tmWeight: Integer;
   tmItalic: Byte;
   tmUnderlined: Byte;
   tmStruckOut: Byte;
   tmFirstChar: Byte;
   tmLastChar: Byte;
   tmDefaultChar: Byte;
   tmBreakChar: Byte;
   tmPitchAndFamily: Byte;
   tmCharSet: Byte;
   tmOverhang: Integer;
   tmDigitizedAspectX: Integer;
   tmDigitizedAspectY: Integer;
end;
```

Поля записи TTEXTMETRIC предназначены для хранения информации, бо́льшую часть которой мы уже рассмотрели. Например, вы уже знаете, что понимается под высотой шрифта, средней шириной символа, подчеркиванием, курсивом, перечеркиванием и пр. Более подробную информацию о структуре TTEXTMETRIC можно получить в интерактивной справочной системе Win32 API. В листинге 8.10 содержится пример программы, выполняющей обработку информации о шрифте.

Листинг 8.10. Исходный текст формы вывода информации о шрифте

```
unit FontInfoFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls;

type

  TFontInfoForm = class(TForm)
    lbFontInfo: TListBox;
    procedure FormActivate(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  FontInfoForm: TFontInfoForm;

implementation
uses MainFrm;

{$R *.DFM}

procedure TFontInfoForm.FormActivate(Sender: TObject);
const
  PITCH_MASK: byte = $0F;  // Установка младших 4 разрядов
  FAMILY_MASK: byte = $F0; // Установка старших 4 разрядов
var
  TxMetric: TTextMetric;
  FaceName: String;
  PitchTest, FamilyTest: byte;
begin

  // Выделение памяти для строки FaceName
  SetLength(FaceName, lf_FaceSize+1);

  // Получение информации о шрифте
  with MainForm.pbxFont.Canvas do
  begin
    GetTextFace(Handle, lf_faceSize-1, PChar(FaceName));
    GetTextMetrics(Handle, TxMetric);
  end;

  // Добавление информации о шрифте в список из структуры
```

```
TTEXTMETRIC.
 with lbFontInfo.Items, TxMetric do
 begin
   Clear;
   Add('Font face name:     '+FaceName);
   Add('tmHeight:     '+IntToStr(tmHeight));
   Add('tmAscent:     '+IntToStr(tmAscent));
   Add('tmDescent:     '+IntToStr(tmDescent));
   Add('tmInternalLeading:     '+IntToStr(tmInternalLeading));
   Add('tmExternalLeading:     '+IntToStr(tmExternalLeading));
   Add('tmAveCharWidth:     '+IntToStr(tmAveCharWidth));
   Add('tmMaxCharWidth:     '+IntToStr(tmMaxCharWidth));
   Add('tmWeight:     '+IntToStr(tmWeight));

   if tmItalic <> 0  then
     Add('tmItalic: YES')
   else
     Add('tmItalic: NO');

   if tmUnderlined <> 0 then
     Add('tmUnderlined: YES')
   else
     Add('tmUnderlined: NO');

   if tmStruckOut <> 0 then
     Add('tmStruckOut: YES')
   else
     Add('tmStruckOut: NO');

   // Анализ типа шрифта
   PitchTest := tmPitchAndFamily and PITCH_MASK;
   if (PitchTest and TMPF_FIXED_PITCH) = TMPF_FIXED_PITCH then
     Add('tmPitchAndFamily-Pitch: Fixed Pitch');
   if (PitchTest and TMPF_VECTOR) = TMPF_VECTOR then
     Add('tmPitchAndFamily-Pitch: Vector');
   if (PitchTest and TMPF_TRUETYPE) = TMPF_TRUETYPE then
     Add('tmPitchAndFamily-Pitch: True type');
   if (PitchTest and TMPF_DEVICE) = TMPF_DEVICE then
     Add('tmPitchAndFamily-Pitch: Device');
   if PitchTest = 0 then
     Add('tmPitchAndFamily-Pitch: Unknown');

   // Анализ семейства шрифта
   FamilyTest := tmPitchAndFamily and FAMILY_MASK;
   if (FamilyTest and FF_ROMAN) = FF_ROMAN then
     Add('tmPitchAndFamily-Family: FF_ROMAN');
   if (FamilyTest and FF_SWISS) = FF_SWISS then
     Add('tmPitchAndFamily-Family: FF_SWISS');
   if (FamilyTest and FF_MODERN) = FF_MODERN then
     Add('tmPitchAndFamily-Family: FF_MODERN');
```

```
    if (FamilyTest and FF_SCRIPT) = FF_SCRIPT then
      Add('tmPitchAndFamily-Family: FF_SCRIPT');
    if (FamilyTest and FF_DECORATIVE) = FF_DECORATIVE then
      Add('tmPitchAndFamily-Family: FF_DECORATIVE');
    if FamilyTest = 0 then
      Add('tmPitchAndFamily-Family: Unknown');

    Add('tmCharSet:     '+IntToStr(tmCharSet));
    Add('tmOverhang:     '+IntToStr(tmOverhang));
    Add('tmDigitizedAspectX:     '+IntToStr(tmDigitizedAspectX));
    Add('tmDigitizedAspectY:     '+IntToStr(tmDigitizedAspectY));
  end;
end;

end.
```

В методе `FormActive()` сначала считывается имя шрифта с помощью функции Win32 API `GetTextFace()`, которой в качестве параметров передаются контекст устройства, размер буфера и сам символьный буфер в формате с завершающим нуль-символом. Затем вызывается функция `GetTextMetrics()` для заполнения структуры `TxMetric` (имеющей тип `TTEXTMETRIC`) информацией, соответствующей выбранному шрифту. После этого в обработчике событий полученные значения полей структуры `TxMetric` добавляются в окно списка в качестве отдельных строк. Чтобы отобразить тип шрифта и семейство, к которому он принадлежит, значение `tmPitchAndFamily` маскируется для выделения старших или младших разрядов в зависимости от того, что именно — тип или семейство — определяется в данный момент, после чего значения также добавляются в окно списка. Вид формы отображения информации о шрифте показан на рис. 8.29.

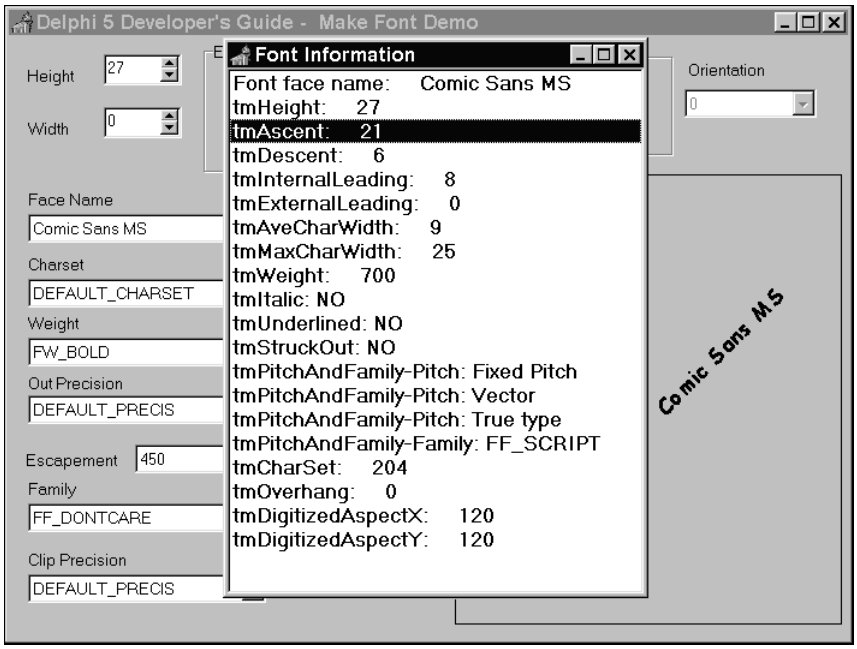


Рис. 8.29. Форма для отображения информации о шрифте

Резюме

В этой главе было представлено множество сведений о возможностях графического интерфейса устройств (GDI) системы Win32. Мы рассмотрели класс `TCanvas` Delphi 5, его свойства и методы рисования. Много внимания уделялось представлению изображений с помощью компонента Delphi 5 `TImage`, а также режимам отображения и координатным системам Win32. Кроме того, обсуждались методы графического программирования для построения программ рисования и выполнения простой анимации, а также работа со шрифтами. Обсуждались методы создания шрифтов и отображения информации о них.

Самое ценное в работе с интерфейсом GDI заключается в том, что эксперименты с его функциями могут доставить вам немало интересных минут. Дерзайте — и все у вас получится!

Динамически компонуемые библиотеки

Глава

9

Эта глава посвящена *динамически компонуемым библиотекам* Win32, или DLL (dynamic link libraries). Они являются ключевым компонентом при написании любого приложения Windows. В этой главе рассматриваются некоторые аспекты использования и создания библиотек DLL. Прежде всего дается обзор принципов работы DLL и обсуждаются способы их создания и применения. Мы рассмотрим различные методы их загрузки и привязки к тем процедурам и функциям, которые они экспортируют. Кроме того, в этой главе уделяется внимание функциям обратного вызова (callback functions) и совместному использованию данных библиотек DLL различными вызывающими процессами.

Что такое библиотека DLL

Динамически компонуемые библиотеки представляют собой программные модули, содержащие код, данные или ресурсы, которые могут совместно использоваться несколькими приложениями Windows. Одно из основных назначений библиотек DLL — позволить приложениям загружать код, который отрабатывается во время выполнения, вместо того чтобы компоновать его в само приложение в процессе компиляции. А это значит, что несколько приложений могут одновременно использовать один и тот же код, содержащийся в библиотеке DLL. Так, файлы библиотек `Kernel32.dll`, `User32.dll` и `GDI32.dll` являются теми тремя китами, на которые опирается система Win32. Файл `Kernel32.dll`, например, отвечает за управление памятью, процессами и потоками. Файл `User32.dll` содержит функции интерфейса пользователя, необходимые для создания окон и обработки сообщений Win32. И, наконец, на файл `GDI32.dll` возложена работа с графикой. Вам также придется сталкиваться с другими системными библиотеками DLL, например `AdvAPI32.dll` и `ComDlg32.dll`, которые предназначены для обеспечения работы с системным реестром и диалоговыми окнами общего назначения.

Другое преимущество использования библиотек DLL состоит в том, что ваше приложение становится модульным. Это упрощает процесс обновления приложений, поскольку при необходимости обновляется не все приложение полностью, а только определенные библиотеки. Типичным примером может служить среда Windows. При каждой установке нового устройства приходится устанавливать и новую библиотеку DLL с драйвером, с помощью которого это устройство сможет общаться с Windows. Преимущество модульности станет очевидным, если представить необходимость повторной инсталляции Windows при каждой установке нового устройства в систему.

С точки зрения хранения на диске файлы библиотек DLL практически ничем не отличаются от EXE-файлов Windows. Разница лишь в том, что файл библиотеки DLL не является независимым выполняемым файлом, хотя может содержать выполняемый код. Чаще всего файлы библиотек DLL имеют расширение `.dll`. Но могут встречаться и другие: `.drv` — для драйверов устройств, `.sys` — для системных файлов, `.fon` — для файлов ресурсов шрифтов, которые не содержат выполняемого кода.

На заметку

В Delphi используются библиотеки DLL специального назначения, называемые *пакетами*. Они применяются не только в среде Delphi, но и в среде Borland C++ Builder. Подробнее о пакетах рассказывается в главе 21 второго тома, “Создание пользовательских компонентов в Delphi”.

Библиотеки DLL могут использовать свой код совместно с другими приложениями благодаря процессу, называемому *динамической компоновкой*, который рассматривается ниже в этой главе. Как правило, когда какое-либо приложение использует библиотеку DLL, система Win32 гарантирует, что в памяти будет размещена только одна копия этой библиотеки. Это достигает-

ся с помощью механизма *отображения файла в память* (memory-mapped files). Суть его состоит в том, что библиотека DLL сначала загружается в глобальную свободную память (кучу) системы Win32, а затем отображается на адресное пространство вызывающего процесса. В системе Win32 каждому процессу выделяется собственное 32-разрядное линейное адресное пространство. Поэтому, когда одна и та же библиотека DLL загружается сразу несколькими процессами, каждый из них получает собственный образ этой библиотеки. Следовательно, процессы не используют одновременно один и тот же физический код, данные или ресурсы, как это было в 16-разрядной Windows. В системе Win32 работа организована так, что библиотека DLL становится как бы реальным кодом, принадлежащим вызывающему процессу. Более подробно о работе системы Win32 поговорим в главе 3, "Win32 API".

Сказанное выше вовсе не означает, что, когда несколько процессов загружает одну и ту же библиотеку DLL, физическая память расходуется на хранение всех необходимых ее копий. Образ DLL можно разместить в адресном пространстве всех процессов, отобразив его из системной глобальной кучи на адресное пространство каждого процесса, использующего эту DLL, — по крайней мере, в идеале (читайте врезку "Установка предпочтительного базового адреса библиотеки DLL").

Установка предпочтительного базового адреса библиотеки DLL

Код библиотеки DLL разделяется между процессами только в том случае, если эту библиотеку можно загрузить в адресное пространство процессов всех заинтересованных клиентов по некоторому предпочтительному базовому адресу библиотеки DLL. Если предпочтительный базовый адрес с учетом размера библиотеки DLL перекрывается с каким-либо другим объектом, уже размещенным в памяти процесса, загрузчик Win32 должен изменить расположение всего образа библиотеки DLL, используя другой базовый адрес. В этом случае ни один из перемещенных образов DLL не используется никакими другими процессами в системе, т.е. каждый перемещенный экземпляр библиотеки DLL занимает блок собственной физической памяти и часть пространства, предназначенного для подкачки файлов.

Важно установить базовый адрес каждой библиотеки DLL равным такому значению, которое не конфликтует или не перекрывается с другими адресными диапазонами, используемыми в создаваемом приложении. Для этой цели служит директива компилятора `$IMAGEBASE`.

Если библиотека DLL предназначена для использования несколькими приложениями, выберите уникальный базовый адрес, который с минимальной вероятностью будет перекрываться с адресами приложения в области младших значений виртуального адресного диапазона или общих библиотек DLL (например, пакетов VCL) в области старших адресов диапазона. По умолчанию базовый адрес всех выполняемых файлов (с расширениями `.EXE` и `.DLL`) равен `$400000`, а это значит, что если не изменить базовый адрес конкретной библиотеки DLL, то конфликт с базовым адресом ее главного EXE-файла будет неизбежен и она никогда не будет совместно использоваться всеми заинтересованными процессами.

В применении базового адреса есть еще один плюс. Поскольку библиотека DLL не требует изменения расположения или внесения исправлений (что обычно и происходит) и сохраняется на локальном диске, страницы ее памяти отображаются прямо в файл DLL на диске. Код DLL не занимает никакого пространства в системном файле страниц (файле подкачки или страничного обмена). Вот почему общее количество и размер выведенных на диск страниц системы может намного превышать размеры системного файла подкачки и объем ОЗУ.

Подробную информацию об использовании директивы компилятора `$IMAGEBASE` можно найти в разделе "Image Base Address" интерактивной справочной системы Delphi 5.

Прежде чем продолжить обсуждение данной темы, необходимо уточнить некоторые термины.

- *Приложение*. Программа Windows, размещенная в `.exe`-файле.
- *Выполняемый файл*. Файл, содержащий выполняемый код (обычно такие файлы имеют расширения `.exe` и `.dll`).

- *Экземпляр*. Если речь идет о приложениях или библиотеках DLL, термин *экземпляр* используется для обозначения отдельного выполняемого файла. Каждому экземпляру соответствует *дескриптор экземпляра*, который назначается системой Win32. Например, когда некоторое приложение запускается дважды, образуется два экземпляра этого приложения и, следовательно, два дескриптора экземпляра. При загрузке библиотеки DLL создается экземпляр этой библиотеки и соответствующий дескриптор экземпляра. Используемый здесь термин *экземпляр* не следует путать с экземпляром класса.
- *Модуль*. В 32-разрядной Windows (в отличие от 16-разрядной) термины *модуль* и *экземпляр* могут использоваться как синонимы. В 16-разрядной Windows для управления модулями существует специальная база данных, в которой для каждого модуля хранится свой дескриптор. В 32-разрядной Windows каждый экземпляр приложения получает собственное адресное пространство, следовательно, нет нужды в образовании отдельного идентификатора модуля. Однако фирма Microsoft в своей документации по-прежнему использует этот термин. Поэтому вам просто нужно иметь в виду, что модуль и экземпляр означают одно и то же.
- *Задача*. Система Windows является многозадачной средой (или средой с переключением задач). Она должна обладать способностью распределять системные ресурсы и время между различными экземплярами приложений, работающими под ее управлением. Это достигается путем ведения базы данных задач, в которой хранятся дескрипторы экземпляров и другая необходимая информация, позволяющая системе выполнять функции переключения задач. Таким образом, задача — это элемент, для которого Windows выделяет ресурсы и блоки времени.

Сравнение статической и динамической компоновки

Статической компоновкой называется метод, с помощью которого компилятор Delphi разрешает вызовы функции или процедуры в выполняемом коде. Код функции может храниться в `.dpr`-файле приложения или в любом модуле, но при компоновке приложений эти функции и процедуры становятся частью конечного выполняемого файла. Другими словами, каждая функция будет занимать на диске определенное место в `.exe`-файле программы.

Расположение функции заранее определено относительно места, занимаемого в памяти программой. Любые обращения к этой функции вызывают передачу управления непосредственно по адресу расположения функции. Далее эта функция выполняется и по завершении возвращает управление в то место, откуда она была вызвана. Относительный адрес функции вычисляется во время процесса компоновки.

Это — приблизительное описание в действительности более сложного процесса, который компилятор Delphi использует для выполнения статической компоновки. Однако задачи, поставленные в этой книге, не требуют от вас понимания неявных операций, выполняемых компилятором с целью повышения эффективности использования библиотек DLL в создаваемых приложениях.

На заметку

В Delphi реализован интеллектуальный компоновщик, который автоматически удаляет функции, процедуры, переменные и типизированные константы, на которые нет ссылок в окончательном варианте проекта. Следовательно, расположенные в больших модулях функции, не используемые в данном проекте, никогда не становятся частью создаваемого `.exe`-файла.

Предположим, имеются два приложения, которые используют одну и ту же функцию, находящуюся в некотором модуле. В обоих приложениях имя этого модуля, безусловно, будет входить в список имен оператора `uses`. Если одновременно запустить оба приложения в среде Windows, текст этой функции будет дважды присутствовать в памяти. Если запустить третье приложение, в памяти появится и третий экземпляр этой функции, а общий расход памяти будет втрое превосходить ее размер. Этот маленький пример иллюстрирует одну из главных причин использования динамической компоновки. Если эту функцию расположить в библиотеке DLL, то при ее загрузке в память одним приложением все другие приложения, которым потребуется ссылка на нее, смогут использовать ее код посредством отображения образа этой библиотеки DLL на адресное пространство их собственных процессов. В результате данная функция будет существовать в памяти только в одном экземпляре — теоретически.

При *динамической компоновке* связь между вызовом функции и ее выполняемым кодом устанавливается во время выполнения приложения посредством создания внешней ссылки на конкретную функцию библиотеки DLL. Подобные ссылки могут быть объявлены в самом приложении, но обычно они размещаются в отдельном модуле `import`. В модуле `import` объявляются импортируемые функции и процедуры, а также определяются различные типы данных, используемые функциями библиотек DLL.

Предположим, например, что имеется библиотека DLL с именем `MaxLib.dll`, содержащая следующую функцию:

```
function Max(i1, I2: integer): integer;
```

Эта функция возвращает большее из двух переданных ей целых чисел. Типичный модуль `import` для этой функции будет выглядеть следующим образом:

```
unit MaxUnit;
interface
function Max(I1, I2: integer): integer;
implementation
function Max; external 'MAXLIB';
end.
```

Вероятно, вы заметили, что, хотя эта функция внешне напоминает типичный модуль, в ней нет определения функции `Max()`. Ключевое слово `external` просто говорит о том, что данная функция находится в файле, имя которого указано сразу за этим ключевым словом. Для использования указанного модуля приложению достаточно иметь в списке инструкции `uses` его имя, т.е. `MaxUnit`. При выполнении приложения требуемая библиотека DLL загружается в память автоматически, и любые обращения к функции `Max()` связываются с функцией `Max()`, находящейся в этой библиотеке.

Этот пример иллюстрирует один из двух способов загрузки библиотек DLL, который называется *неявной загрузкой*. Он требует от Windows автоматически загружать библиотеку DLL при загрузке приложения. Второй способ называется *явной загрузкой* DLL и будет рассматриваться ниже в этой главе.

Зачем нужны библиотеки DLL

Существует несколько причин использования библиотек DLL, часть из которых уже упоминалась выше. Как правило, библиотеки DLL применяются либо для разделения кода или системных ресурсов, либо для сокрытия реализации программного кода или системных функций низкого уровня, либо для разработки пользовательских элементов управления. Эти темы и рассматриваются в последующих разделах.

Совместное использование программного кода, ресурсов и данных несколькими приложениями

Ранее упоминалось, что самой распространенной причиной создания библиотек DLL является разделение кода. В отличие от модулей, которые позволяют совместно использовать исходный код в различных приложениях Delphi, библиотеки DLL позволяют совместно использовать один и тот же исполняемый код любыми приложениями Windows, способными вызывать функции из библиотек DLL.

Кроме того, библиотеки DLL предоставляют способ совместного использования растровых изображений, шрифтов, пиктограмм и других ресурсов, которые обычно входят в состав файла ресурсов и непосредственно связываются с создаваемым приложением. Если эти ресурсы разместить в библиотеке DLL, многие приложения смогут воспользоваться ими, не затрачивая дополнительной памяти, необходимой для загрузки дополнительных экземпляров этих ресурсов.

В 16-разрядной Windows библиотеки DLL использовали собственный сегмент данных, поэтому все приложения, которые обращались к определенной библиотеке DLL, получали доступ к одним и тем же глобальным данным и статическим переменным. В системе Win32 дело обстоит иначе. Поскольку образ библиотеки DLL отображается на адресное пространство каждого процесса, все данные, определенные функциями в DLL, принадлежат отдельному процессу. При этом стоит подчеркнуть одну деталь: несмотря на то что данные библиотек DLL не распределяются между различными *процессами*, они могут разделяться несколькими *потоками* внутри одного и того же процесса. А поскольку потоки выполняются независимо друг от друга, необходимо предпринимать меры предосторожности, позволяющие избежать конфликтов при доступе к глобальным данным библиотек DLL.

Это вовсе не означает, что нет способов заставить несколько процессов совместно использовать данные, доступные через библиотеки DLL. Один из таких способов заключается в создании внутри библиотеки DLL общей области памяти (с помощью отображенного в память файла). В этом случае каждое приложение, использующее такую библиотеку, получает возможность прочитать данные, хранящиеся в общей области памяти. Подробнее данный метод будет описан ниже в этой главе.

Сокрытие реализации

В определенных случаях может возникнуть необходимость скрыть детали реализации программ, помещаемых в библиотеки DLL. Существует множество причин, побуждающих вас скрыть реализацию своего кода. Библиотеки DLL позволяют сделать ваши функции доступными пользователям без раскрытия их исходного текста. От вас потребуется лишь подготовить модуль интерфейса, позволяющий другим пользователям получить доступ к создаваемой библиотеке. Если вы полагаете, что подобная возможность уже давно обеспечивается механизмом откомпилированных модулей Delphi (DCU), то следует заметить, что файл DCU может использоваться лишь другими приложениями Delphi, причем созданными компилятором той же самой версии. Тогда как формат библиотек DLL не зависит от используемого языка программирования, поэтому созданные в Delphi библиотеки можно будет применять в приложениях, написанных на C++, Visual Basic или любом другом языке, который поддерживает работу с файлами DLL.

Модуль Windows является модулем интерфейса с библиотекой Win32 DLL. В поставку Delphi 5 включены исходные файлы модуля Win32 API. Среди них вы найдете файл Windows.pas, содержащий исходный код модуля Windows. В разделе interface файла Windows.pas нетрудно отыскать определения функций, подобные следующему:

```
function ClientToScreen(Hwnd: HWND; var lpPoint: TPoint): BOOL; stdcall;
```

В разделе implementation присутствует соответствующая связка с библиотекой DLL:

```
function ClientToScreen; external user32 name 'ClientToScreen';
```

Эта строка означает, что процедура ClientToScreen() находится в динамически связываемой библиотеке User32.dll и ее имя ClientToScreen.

Пользовательские элементы управления

Пользовательские элементы управления обычно размещаются в библиотеках DLL. Их не следует путать с пользовательскими компонентами Delphi. Пользовательские элементы управления регистрируются в среде Windows и могут использоваться в любой языковой среде разработки для Windows. Они размещаются в библиотеках DLL в целях экономии памяти, достигаемой за счет размещения в памяти только одной копии программного кода даже при использовании в системе нескольких экземпляров конкретного элемента управления.

На заметку Устаревший механизм применения библиотек DLL с пользовательскими элементами управления был исключительно жестким и недоработанным. Поэтому в настоящее время фирма Microsoft использует только элементы управления OLE и ActiveX.

Создание и использование библиотек DLL

В этом разделе демонстрируется процесс реального создания библиотеки DLL в среде Delphi. Мы рассмотрим, как создается модуль интерфейса, позволяющий сделать библиотеку DLL доступной другим программам. Однако прежде чем перейти к более сложным вопросам использования библиотек в Delphi, мы познакомимся с методами помещения в библиотеки DLL форм Delphi.

Подсчет монет (пример простой библиотеки DLL)

В приведенном ниже примере иллюстрируется помещение в библиотеку DLL функции, которую очень любят предлагать студентам профессора компьютерных наук. Эта функция рассчитывает минимальное количество монет достоинством в 1, 5, 10 или 25 центов, необходимых для получения заданной суммы (также в центах).

Базовая библиотека DLL

Создаваемая библиотека состоит из метода PenniesToCoins(). В листинге 9.1 содержится законченный проект библиотеки DLL.

Листинг 9.1. Файл `PenniesLib.dpr` — библиотека DLL, предназначенная для представления суммы в центах в виде набора монет различного достоинства

```
library PenniesLib;
{$DEFINE PENNIESLIB}
uses
  SysUtils,
  Classes,
  PenniesInt;

function PenniesToCoins(TotPennies: word;
    CoinsRec: PCoinsRec): word; StdCall;
  begin
  Result := TotPennies;  // Присвоение значения переменной Result
  { Вычисление количества монет достоинством в 25, 10, 5 и 1 цент соответственно
}
  with CoinsRec^ do
  begin
    Quarters    := TotPennies div 25;
    TotPennies  := TotPennies - Quarters * 25;
    Dimes       := TotPennies div 10;
    TotPennies  := TotPennies - Dimes * 10;
    Nickels     := TotPennies div 5;
    TotPennies  := TotPennies - Nickels * 5;
    Pennies     := TotPennies;
  end;
end;

{ Экспортирование функции по имени }
exports
  PenniesToCoins;
end.
```

Обратите внимание, что в этой библиотеке используется модуль `PenniesInt`, который мы рассмотрим более детально чуть ниже.

Раздел `exports` указывает, какие функции или процедуры экспортируются данной библиотекой DLL и становятся доступными вызывающим приложениям.

Определение модуля интерфейса

С помощью модулей интерфейса пользователи созданной вами библиотеки DLL смогут статически импортировать функции этой библиотеки в свои приложения путем простого размещения имени импортируемого модуля в инструкции `uses` своих модулей. Модули интерфейса также позволяют создателю библиотеки DLL определять общие структуры, используемые как самой библиотекой, так и вызывающим приложением. Продемонстрируем эти возможности на примере модуля интерфейса для библиотеки `PenniesLib.Dll`. В листинге 9.2 содержится исходный текст модуля интерфейса `PenniesInt.pas`.

Листинг 9.2. Модуль интерфейса PenniesInt.pas для библиотеки PenniesLib.Dll

```
unit PenniesInt;
{ Модуль интерфейса для библиотеки PENNIES.DLL }

interface
type

  { Эта запись будет содержать перечень монет после выполнения преобразований }
  PCoinsRec = ^TCoinsRec;
  TCoinsRec = record
    Quarters,
    Dimes,
    Nickels,
    Pennies: word;
  end;

{$IFNDEF PENNIESLIB}
{ Объявление функции с использованием ключевого слова export }

function PenniesToCoins(TotPennies: word; CoinsRec: PCoinsRec): word; StdCall;
{$ENDIF}

implementation

{$IFNDEF PENNIESLIB}
{ Определение импортированной функции }
function PenniesToCoins; external 'PENNIESLIB.DLL' name 'PenniesToCoins';
{$ENDIF}

end.
```

В разделе type этого модуля объявляется запись TCoinsRec, а также указатель на эту запись. Упомянутая запись будет содержать сведения о количестве монет каждого достоинства, необходимых для составления суммы в центах, переданной функции PenniesToCoins(). Этой функции передается два параметра: общая денежная сумма в центах и указатель на переменную TCoinsRec. Результатом выполнения функции является та же самая денежная сумма, которая была передана функции в качестве параметра, но уже в виде некоторого набора монет.

В модуле PenniesInt.pas объявляется функция, которую библиотека PenniesLib.dll экспортирует в своем разделе interface. Определение функции размещается в разделе реализации (implementation). Это определение означает, что функция является внешней и находится в библиотечном файле PenniesLib.dll. Связь с DLL-функцией устанавливается по имени функции. Обратите внимание, что здесь была использована директива компилятора $IFNDEF PENNIESLIB, применяемая для выполнения условной компиляции объявления функции PenniesToCoins(). Это вызвано отсутствием необходимости компоновать данное объявление при компиляции модуля интерфейса для библиотеки. Благодаря этому можно применять объявления типа из модуля интерфейса совместно как с библиотекой, так и с любыми приложениями, которые будут использовать эту библиотеку. Любые изменения в структуру, используемую как библиотекой, так и приложениями, нужно вносить только в модуле интерфейса.

Чтобы определить директиву условной компиляции в масштабе приложения, укажите ее во вкладке Directories/Conditionals диалогового окна Project Options, которое открывается при выборе команды Project⇨Options. После этого следует обязательно заново создать данный проект, чтобы вступили в силу изменения, связанные с условными директивами, поскольку в логике команды Make не предусмотрено переоценки условных определений.

На заметку

Следующее определение показывает один из двух возможных способов импортирования DLL-функции:

```
function PenniesToCoins; external 'PENNIESLIB.DLL' index 1;
```

Этот метод называется импортом по порядковому номеру. Импортировать DLL-функции можно также и другим способом, который называется импортом по имени:

```
function PenniesToCoins; external 'PENNIESLIB.DLL' name
'PenniesToCoins';
```

Нетрудно догадаться, что для определения того, какую функцию следует связать с данной библиотекой DLL, во втором методе используется имя, указываемое после ключевого слова name.

При использовании импорта по порядковому номеру уменьшается время загрузки библиотеки DLL, поскольку в этом случае отпадает необходимость поиска имени функции в ее таблице имен. Тем не менее, в Win32 предпочтение отдается импорту по имени. В этом случае приложения не обладают нежелательной чувствительностью к изменению месторасположения точек входа в функции DLL, возможному при частых обновлениях библиотеки. При импорте по порядковому номеру программа привязана к определенному месту в библиотеке DLL, а при импорте по имени — к имени функции, независимо от ее расположения в библиотеке.

Если рассматриваемая выше библиотека была бы реальной DLL, которую предполагалось использовать и в дальнейшем, следовало бы предоставить пользователям оба файла — и PenniesLib.dll, и PenniesInt.pas. Тогда они смогли бы работать с данной библиотекой, определяя типы и функции в модуле PenniesInt.pas, требуемые для файла PenniesLib.dll. Кроме того, программисты, работающие на других языках (например, C++), могли бы преобразовать модуль PenniesInt.pas в свой язык, что позволило бы им использовать вашу библиотеку DLL в своей среде разработки. Проект, работающий с библиотекой PenniesLib.dll, можно найти на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Отображение модальных форм из библиотек DLL

В этом разделе мы рассмотрим, как сделать модальные формы доступными в библиотеке DLL. Одна из причин помещения часто используемых форм в библиотеку DLL состоит в возможности применения этих форм в других приложениях Windows и даже в другой среде разработки (например, в C++ или в Visual Basic).

Для достижения намеченной цели, прежде всего, удалите будущую DLL-форму из списка автоматически создаваемых форм.

Ранее мы уже создали подходящую форму, в главном окне которой содержится компонент TCalendar. Вызывающее приложение будет обращаться к DLL-функции, запускающей данную форму. Когда пользователь выберет в календаре день, в вызывающее приложение будет возвращена соответствующая дата.

В листинге 9.3 показан исходный текст файла проекта CalendarLib.dpr, а в листинге 9.4 — исходный текст модуля самой DLL-формы DllFrm.pas. Оба этих файла служат иллюстрацией метода инкапсуляции формы в библиотеку DLL.

Листинг 9.3. Исходный текст файла проекта CalendarLib.dpr

```
unit DLLFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, Calendar;

type

  TDLLForm = class(TForm)
    calDllCalendar: TCalendar;
    procedure calDllCalendarDblClick(Sender: TObject);
  end;

{ Объявление экспортируемой функции }
function ShowCalendar(AHandle: THandle; ACaption: String): TDateTime; StdCall;

implementation
{$R *.DFM}

function ShowCalendar(AHandle: THandle; ACaption: String): TDateTime;
var
  DLLForm: TDllForm;
begin
  // Копирование дескриптора приложения в объект TApplication библиотеки DLL
  Application.Handle := AHandle;
  DLLForm := TDLLForm.Create(Application);
  try
    DLLForm.Caption := ACaption;
    DLLForm.ShowModal;
    // Возвращение даты в качестве результата
    Result := DLLForm.calDLLCalendar.CalendarDate; finally
    DLLForm.Free;
  end;
end;

procedure TDLLForm.calDllCalendarDblClick(Sender: TObject);
begin
  Close;
end;

end.
```

Главная форма в этой библиотеке DLL включается в экспортируемую функцию. Обратите внимание на то, что объявление `DLLForm` было удалено из раздела `interface` и перенесено в текст самой функции.

В рассматриваемой DLL-функции свойству `Application.Handle` был присвоен параметр `AHandle`. Напомним, что все проекты Delphi (вспомните главу 4, "Строение приложения и концепции конструирования"), включая и библиотечные проекты, содержат глобальный объект `Application`. В функции библиотеки DLL этот объект отличен от объекта `Application`, который существует в вызывающем приложении. Чтобы DLL-форма правильно работала в качестве модальной формы для вызывающего приложения, необходимо присвоить дескриптор этого вызывающего приложения свойству `Application.Handle` функции из библиотеки DLL, что и было продемонстрировано. Без этого поведение библиотечной формы было бы некорректным, особенно в случае ее минимизации. Кроме того, при создании подобной формы не следует передавать значение, равное `nil`.

После создания DLL-формы ее свойству `Caption` присваивается строка `ACaption`. Затем форма отображается в модальном режиме. Когда форма закрывается, дата, выбранная пользователем в компоненте `TCalendar`, передается обратно в вызывающую функцию. Форма закрывается по двойному щелчку на компоненте `TCalendar`.

Если создаваемая библиотека DLL экспортирует любые процедуры или функции, которые получают в качестве параметров или возвращают в виде результатов выполнения строки или динамические массивы, то первым элементом списка инструкции `uses` этой библиотеки и проекта (выберите команду View⇨Project Source) должен быть модуль `ShareMem`. Это относится ко всем строкам, передаваемым или получаемым от библиотек DLL, даже если эти строки вложены в записи или классы. Модуль `ShareMem` является модулем интерфейса для библиотеки `Borlndmm.dll` (диспетчера разделяемой памяти), который необходимо использовать вместе с этой библиотекой. Чтобы обойтись без модуля `Borlndmn.dll`, передавайте строковую информацию с помощью параметра типа `PChar` или `ShortString`.

Использование модуля `ShareMem` является единственным обязательным условием для организации передачи из одного приложения в другое размещенных в куче строк или динамических массивов. При этом также передается и право собственности на данную строковую память. После преобразования внутренней строки к типу `PChar` (с помощью операции приведения типа) с последующей передачей ее в другой модуль как параметра типа `PChar`, право собственности на строковую память вызываемому модулю не передается, и поэтому модуль интерфейса `ShareMem` не требуется.

Следует учесть, что модуль `ShareMem` применяется только к библиотекам Delphi/BCB DLL, которые передают строки или динамические массивы другим библиотекам Delphi/BCB DLL или EXE-файлам. Никогда не передавайте строки или динамические массивы Delphi (в качестве параметров или результатов выполнения экспортируемых функций библиотек DLL) в функции библиотек DLL или приложения, написанные на другом языке. Суть в том, что внешние приложения не знают, как корректно освободить эти элементы среды Delphi.

Кроме того, модуль `ShareMem` никогда не требуется в отношениях между приложениями, встроенными в пакеты. В этом случае диспетчер памяти распределяет память между членами пакета неявно.

Вот и все, что требуется знать для инкапсуляции модальной формы в функцию библиотеки DLL. В следующем разделе рассматривается помещение в библиотеку DLL немодальной формы.

Отображение немодальных форм из библиотек DLL

Для иллюстрации размещения в библиотеке DLL немодальных форм вновь воспользуемся формой с календарем, к которой мы обращались в предыдущем разделе.

Для отображения из библиотеки DLL немодальных форм, такая DLL должна содержать две функции. На первую возлагается задача создания и отображения формы, а на вторую — ее освобождения. В листинге 9.4 представлен исходный текст, иллюстрирующий вывод немодальной формы функциями DLL.

Листинг 9.4. Немодальная форма в библиотеке DLL

```
unit DLLFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, Calendar;

type

  TDLLForm = class(TForm)
    calDllCalendar: TCalendar;
  end;

{ Объявление экспортируемой функции }
function ShowCalendar(AHandle: THandle; ACaption: String): Longint; stdCall;
procedure CloseCalendar(AFormRef: Longint); stdcall;

implementation
{$R *.DFM}

function ShowCalendar(AHandle: THandle; ACaption: String): Longint;
var
  DLLForm: TDllForm;
begin
  //Копирование дескриптора приложения в объект DLL TApplication
  Application.Handle := AHandle;
  DLLForm := TDLLForm.Create(Application);
  Result := Longint(DLLForm);
  DLLForm.Caption := ACaption;
  DLLForm.Show;
end;

procedure CloseCalendar(AFormRef: Longint);
begin
  if AFormRef > 0 then
    TDLLForm(AFormRef).Release;
end;

end.
```

В этом листинге содержится два метода: ShowCalendar() и CloseCalendar(). Метод ShowCalendar() похож на одноименную функцию из примера с модальной формой тем, что дескриптор вызывающего приложения присваивается свойству Handle объекта приложения функции DLL, после чего создается сама форма. Однако вместо вызова метода ShowModal() эта функция вызывает метод Show() и при этом не освобождает форму. Обратите внимание на то, что она возвращает значение типа longint, в роли которого выступает экземпляр DLLForm. Все дело в том, что ссылка на созданную форму должна быть сохранена, поэтому забота о сохранении ссылки на форму поручается вызывающему приложению. Подобная забота распространяется на все экземпляры формы, связанные с любыми приложениями, вызывающими эту функцию библиотеки DLL и создающими собственные экземпляры данной формы.

В процедуре CloseCalendar() просто проверяется корректность ссылки на форму и вызывается ее метод Release(). Здесь вызывающее приложение должно передать обратно ту же самую ссылку, которая была ранее получена от функции ShowCalendar().

При использовании такого подхода нужно иметь в виду, что библиотека DLL никогда не освобождает форму. Если предпринять подобную попытку (например, попытаться вернуть значение caFree функции CanClose()), то обращение к функции CloseCalendar() приведет к аварийной ситуации.

Демонстрационные примеры модальной и немодальной форм имеются на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Использование библиотек DLL в приложениях Delphi

Как уже отмечалось выше в этой главе, существует два способа загрузки или импортирования функций библиотек DLL — явный и неявный. В этом разделе оба способа будут продемонстрированы на примере созданных ранее библиотек DLL.

Первая библиотека DLL, созданная в этой главе, содержала модуль interface. Воспользуемся этим модулем интерфейса для иллюстрации неявного связывания с библиотекой DLL. Главная форма используемого в качестве примера проекта содержит объекты TMaskEdit, TButton и девять экземпляров компонента TLabel.

В этом приложении пользователь вводит денежную сумму в центах. По щелчку на кнопке программа отображает представление введенной суммы в виде набора монет разного достоинства. Эта информация поступает от экспортированной функции PenniesToCoins() библиотеки PenniesLib.dll.

Главная форма приложения определяется в модуле MainFrm.pas, текст которого показан в листинге 9.5.

Листинг 9.5. Главная форма демонстрационного проекта размена сумм

```
unit MainFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Mask;
type
```

```
  TMainForm = class(TForm)
    lblTotal: TLabel;
    lblQlbl: TLabel;
    lblDlbl: TLabel;
    lblNlbl: TLabel;
    lblPlbl: TLabel;
    lblQuarters: TLabel;
    lblDimes: TLabel;
    lblNickels: TLabel;
    lblPennies: TLabel;
    btnMakeChange: TButton;
    meTotalPennies: TMaskEdit;
    procedure btnMakeChangeClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation
uses PenniesInt;  // Использование модуля интерфейса

{$R *.DFM}

procedure TMainForm.btnMakeChangeClick(Sender: TObject);
var
  CoinsRec: TCoinsRec;
  TotPennies: word;
begin
  { Вызов DLL-функции для определения минимального количества монет,
    составляющих денежную сумму, заданную в центах. }
  TotPennies := PenniesToCoins(StrToInt(meTotalPennies.Text), @CoinsRec);
  with CoinsRec do
  begin
    { Вывод информации о наборе монет }
    lblQuarters.Caption := IntToStr(Quarters);
    lblDimes.Caption    := IntToStr(Dimes);
    lblNickels.Caption  := IntToStr(Nickels);
    lblPennies.Caption  := IntToStr(Pennies);
  end
end;

end.
```

Обратите внимание: в модуле `MainFrm.pas` используется модуль `PenniesInt`. Напомним, что модуль `PenniesInt.pas` содержит внешние объявления для функций, помещенных в библиотечный файл `PenniesLib.dpr`. При запуске этого приложения система Win32 автоматически загружает библиотеку `PenniesLib.dpr` и отображает ее на адресное пространство процесса вызывающего приложения.

Использовать модуль import не обязательно. Вполне можно удалить имя PenniesInt из инструкции uses и вставить объявление external для функции PenniesToCoins() в раздел implementation модуля MainFrm.pas:

```
implementation

function PenniesToCoins(TotPennies: word; ChangeRec: PChangeRec): word;
  ↳StdCall external 'PENNIESLIB.DLL';
```

Вам также придется снова определить в модуле MainFrm.pas объекты PChangeRec и TChangeRec (можно просто скомпилировать это приложение с указанием директивы компилятора PENNIESLIB). Этот метод может с успехом использоваться в тех случаях, когда требуется получить доступ только к немногим функциям из данной библиотеки DLL. Однако чаще всего оказывается, что требуются не только внешние объявления для функций библиотеки DLL, но и доступ к типам, определенным в ее модуле interface.

На заметку

При использовании библиотеки DLL какого-либо стороннего производителя у вас может не оказаться ее интерфейсного модуля на языке Pascal. Часто вместо него поставляется библиотека импортируемых функций C/C++. В таком случае придется перевести эту библиотеку в эквивалентный интерфейсный модуль Delphi.

Упоминаемая в этом разделе демонстрационная программа имеется на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Явная загрузка библиотек DLL

Несмотря на удобство метода неявной загрузки библиотек DLL, он не всегда бывает желательным. Предположим, что существует некоторая библиотека DLL, содержащая множество функций. Вполне вероятно, что в обычном режиме работы приложение никогда не вызовет ни одной из функций этой библиотеки. Получается, что при каждом запуске этого приложения загрузка данной библиотеки приводит к напрасным затратам памяти, особенно при использовании этим приложением сразу нескольких библиотек DLL. Другим примером может служить применение компонентов библиотек DLL для заполнения объектов, имеющих очень большой размер и содержащих списки весьма специализированных функций, предоставляемых на выбор для использования в конкретных ситуациях. (Например, списки доступных драйверов принтеров или подпрограмм преобразования формата файлов.) В такой ситуации имело бы смысл загружать каждую библиотеку DLL только по конкретному требованию, исходящему от приложения. Этот метод и называется *явной* загрузкой библиотек DLL.

Для иллюстрации явной загрузки вернемся к примеру библиотеки DLL с модальной формой. В листинге 9.6 содержится программный текст главной формы приложения, в котором демонстрируется явная загрузка этой DLL. Файл проекта этого приложения можно найти на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Листинг 9.6. Главная форма приложения, демонстрирующего явную загрузку библиотеки DLL

```
unit MainFfm;

interface

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

```
type
  { Сначала определяем процедурный тип данных, который должен
    отражать экспортируемую из библиотеки DLL процедуру. }
  TShowCalendar = function (AHandle: THandle; ACaption: String):
    TDateTime; StdCall;

  { Создаем новый класс исключения для отражения неудачной загрузки DLL }
  EDLLLoadError = class(Exception);

  TMainForm = class(TForm)
    lblDate: TLabel;
    btnGetCalendar: TButton;
    procedure btnGetCalendarClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnGetCalendarClick(Sender: TObject);
var
  LibHandle   : THandle;
  ShowCalendar: TShowCalendar;
begin

  { Делаем попытку загрузить DLL }
  LibHandle := LoadLibrary('CALENDARLIB.DLL');
  try
    { Если загрузка DLL окажется неудачной, переменная LibHandle будет иметь
      нулевое значение. В этом случае возникнет исключительная ситуация. }
    if LibHandle = 0 then
      raise EDLLLoadError.Create('Unable to Load DLL');
    { Если выполнение программы дойдет до этого места, значит, DLL
      загрузилась удачно. Теперь устанавливаем связь с экспортируемой
      функцией DLL, чтобы ее можно было вызывать. }
    @ShowCalendar := GetProcAddress(LibHandle, 'ShowCalendar');
    { Если функция импортируется успешно, устанавливаем свойство lblDate.Caption
      для отображения даты, возвращаемой этой функцией. В противном случае
      генерируется исключительная ситуация. }
    if not (@ShowCalendar = nil) then
      lblDate.Caption := DateToStr(ShowCalendar(Application.Handle, Caption))
    else
      RaiseLastWin32Error;
  finally
    FreeLibrary(LibHandle); // Выгрузка DLL.
  end;
end;

end.
```

В начале этого модуля определяется процедурный тип данных TShowCalendar, который отображает определение функции, вызываемой из библиотеки CalendarLib.dll. Затем определяется специальная исключительная ситуация, которая генерируется при возникновении проблем с загрузкой библиотеки DLL. Обратите внимание на то, что в обработчике событий btnGetCalendarClick() используются три функции Win32 API: LoadLibrary(), FreeLibrary() и GetProcAddress().

Функция LoadLibrary() определяется следующим образом:

```
function LoadLibrary(lpLibFileName: PChar): HMODULE; stdcall;
```

Эта функция загружает экземпляр библиотеки DLL, указанной переменной lpLibFileName, и отображает ее на адресное пространство вызывающего процесса. Если работа функции завершается успешно, она возвращает дескриптор экземпляра DLL, а в случае неудачи — нулевое значение, которое и является сигналом к генерации исключительной ситуации. За подробной информацией о работе функции LoadLibrary() и возвращаемых ей кодах ошибок обратитесь к интерактивной справочной системе.

Функция FreeLibrary() определяется следующим образом:

```
function FreeLibrary(hLibModule: HMODULE): BOOL; stdcall;
```

Функция FreeLibrary() уменьшает счетчик экземпляров библиотеки, заданной значением переменной hLibModule. Если счетчик экземпляров примет нулевое значение, то данная библиотека будет удалена из памяти. Упомянутый счетчик экземпляров отслеживает количество задач, использующих данную библиотеку DLL.

А вот как определяется функция GetProcAddress():

```
function GetProcAddress(hModule: HMODULE; lpProcName: LPCSTR): FARPROC; stdcall
```

Функция GetProcAddress() возвращает адрес функции внутри библиотечного модуля, заданного значением первого параметра hModule. Этот параметр имеет тип THandle, возвращаемым при обращении к функции LoadLibrary(). В случае неуспешного завершения работы функции GetProcAddress() последняя возвращает значение nil. Для получения расширенной информации об ошибках нужно вызвать функцию Win32 API GetLastError().

В обработчике событий Button1.OnClick функция LoadLibrary() вызывается для загрузки библиотеки CALDLL. Если загрузка завершается неудачно, генерируется исключительная ситуация. В случае успешной загрузки будет выполнено обращение к функции окна GetProcAddress(), цель которого — получение адреса функции ShowCalendar(). Наличие символа оператора адреса (@) перед переменной процедурного типа ShowCalendar не позволяет компилятору выдать ошибку о несовпадении типов. Полученный адрес функции ShowCalendar() можно будет использовать уже в виде переменной типа TShowCalendar. И, наконец, внутри блока finally вызывается функция FreeLibrary(), гарантирующая, что загруженная библиотека будет освобождена.

Вы видите, что эта библиотека загружается и освобождается при каждом вызове данной функции. Если за все время работы приложения функция вызывается только раз, то становится ясно, что явная загрузка может сэкономить ценные ресурсы памяти, всегда такие нужные и часто весьма ограниченные. Но в то же время, если эта функция вызывалась бы часто, повторение операций загрузки и выгрузки библиотеки DLL привело бы к существенной дополнительной нагрузке на систему.

Функция входа/выхода для библиотек DLL

В случае необходимости для создаваемой библиотеки DLL можно подготовить функции, выполняющие требуемые операции инициализации и завершения работы. В частности, подобные операции могут потребоваться при запуске и останове работы процессов или потоков.

Функции инициализации и завершения процессов и потоков

К типичным операциям инициализации относятся: регистрация классов Windows, инициализация глобальных переменных и функции входа-выхода. Все эти операции выполняются в методе, обеспечивающем вход в библиотеку DLL, который реализуется в виде функции `DLLEntryPoint`. На самом деле эта функция представлена блоком `begin..end` файла проекта DLL. Именно здесь целесообразно было бы поместить процедуру обработки входа/выхода. Этой процедуре должен передаваться один параметр типа `DWord`.

Глобальная переменная `DLLProc` представляет собой процедурный указатель, которому в качестве значения можно назначить адрес процедуры входа/выхода. Исходно эта переменная имеет значение `nil`, сохраняемое до тех пор, пока в нее не будет помещен адрес некоторой собственной процедуры. После установки адреса процедуры входа/выхода в программе появляется возможность реагировать на события, перечисленные в табл. 9.1.

Таблица 9.1. События входа-выхода для библиотеки DLL

Событие	Назначение
`DLL_PROCESS_ATTACH`	Библиотека DLL присоединяется к адресному пространству текущего процесса при запуске процесса или в результате вызова функции `LoadLibrary()`. При обработке этого события в библиотеке DLL инициализируются экземпляры любых типов данных
`DLL_PROCESS_DETACH`	Библиотека DLL отсоединяется от адресного пространства вызывающего процесса. Эта ситуация возникает при выходе из процесса очистки или при обращении к функции `FreeLibrary()`. При обработке этого события в библиотеке DLL могут деинициализироваться экземпляры данных любых типов
`DLL_THREAD_ATTACH`	Это событие происходит, когда текущий процесс создает новый поток. В подобных случаях система вызывает функцию входной точки для любой библиотеки DLL, присоединенной к процессу. Этот вызов выполняется в контексте нового потока и может быть использован для размещения любых данных, предназначенных конкретному потоку
`DLL_THREAD_DETACH`	Это событие происходит, когда работа потока завершается. В процессе обработки этого события в библиотеке DLL могут освобождаться любые инициализированные данные конкретного потока

В потоках, работа которых завершается ненормально (например, посредством вызова функции `TerminateThread()`), генерация события `DLL_THREAD_DETACH` не гарантируется.

Пример использования подпрограмм входа/выхода в библиотеке DLL

В листинге 9.7 показано, как можно было бы определить процедуру входа/выхода в переменной, связанной с библиотекой DLLProc.

Листинг 9.7. Исходный текст файла проекта DLLEntry.dpr

```
library DllEntry;
uses
  SysUtils,
  Windows,
  Dialogs,
  Classes;
procedure DLLEntryPoint(dwReason: DWord);
begin
  case dwReason of
    DLL_PROCESS_ATTACH: ShowMessage('Attaching to process');
    DLL_PROCESS_DETACH: ShowMessage('Detaching from process');
    DLL_THREAD_ATTACH:  MessageBeep(0);
    DLL_THREAD_DETACH:  MessageBeep(0);
  end;
end;
begin
{ Сначала назначаем процедуру переменной DLLProc }
  DllProc := @DLLEntryPoint;
  { Теперь вызываем эту процедуру для отображения того, что DLL присоединена к
процессу }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

Процедура входа/выхода назначается переменной DLLProc библиотеки DLL в блоке begin..end файла проекта данной библиотеки. Эта процедура, называемая DLLEntryPoint(), проверяет значение параметра типа word с целью определения того, какое событие явилось причиной ее вызова. Возможные события соответствуют событиям, перечисленным в табл. 9.1. Для большей наглядности при загрузке и выгрузке этой библиотеки DLL каждое событие в данном проекте будет сопровождаться выводом окна сообщения. При создании и уничтожении потока в вызывающем приложении вместе с сообщением будет издаваться звуковой сигнал.

Для иллюстрации использования этой библиотеки DLL рассмотрим программный текст, представленный в листинге 9.8.

Листинг 9.8. Пример демонстрационной программы использования функций входа/выхода библиотеки DLL

```
unit MainFrm;

interface

uses
```

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

type

  { Определение потомка класса TThread }
  TTestThread = class(TThread)
    procedure Execute; override;
  end;

  TMainForm = class(TForm)
    btnLoadLib: TButton;
    btnFreeLib: TButton;
    btnCreateThread: TButton;
    btnFreeThread: TButton;
    lblCount: TLabel;
    procedure btnLoadLibClick(Sender: TObject);
    procedure btnFreeLibClick(Sender: TObject);
    procedure btnCreateThreadClick(Sender: TObject);
    procedure btnFreeThreadClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    LibHandle   : THandle;
    TestThread  : TTestThread;
    Counter     : Integer;
    GoThread    : Boolean;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TTestThread.Execute;
begin
  while MainForm.GoThread do
  begin
    Synchronize(SetCaptionData);
    Inc(MainForm.Counter);
  end;
end;

procedure TTestThread.SetCaptionData;
begin
  MainForm.lblCount.Caption := IntToStr(MainForm.Counter);
end;

procedure TMainForm.btnLoadLibClick(Sender: TObject);
{ Эта процедура загружает библиотеку DllEntryLib.DLL }
begin
```

```
  if LibHandle = 0 then
  begin
    LibHandle := LoadLibrary('DLLENTRYLIB.DLL');
    if LibHandle = 0 then
      raise Exception.Create('Unable to Load DLL');
  end
  else
    MessageDlg('Library already loaded', mtWarning, [mbok], 0);
end;

procedure TMainForm.btnFreeLibClick(Sender: TObject);
{ Эта процедура освобождает библиотеку }
begin
  if not (LibHandle = 0) then
  begin
    FreeLibrary(LibHandle);
    LibHandle := 0;
  end;
end;

procedure TMainForm.btnCreateThreadClick(Sender: TObject);
{ Эта процедура создает экземпляр класса TThread. Если DLL загружена,
  будет подан звуковой сигнал. }
begin
  if TestThread = nil then
  begin
    GoThread   := True;
    TestThread := TTestThread.Create(False);
  end;
end;

procedure TMainForm.btnFreeThreadClick(Sender: TObject);
{ При освобождении экземпляра TThread будет подан звуковой сигнал,
  если DLL загружена. }
begin
  if not (TestThread = nil) then
  begin
    GoThread   := False;
    TestThread.Free;
    TestThread := nil;
    Counter    := 0;
  end;

end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  LibHandle  := 0;
  TestThread := nil;
end;

end.
```

Этот проект состоит из главной формы с четырьмя кнопками (компонент TButton). По щелчку на кнопке BtnLoadLib загружается библиотека DllEntryLib.dll. Кнопка BtnFreeLib предназначена для удаления этой библиотеки из процесса. С помощью кнопки BtnCreateThread создается объект, класс которого является производным от класса TThread. Этот объект используется для создания нового потока. По щелчку на кнопке BtnFreeThread объект TThread разрушается. Метка lblCount используется только для демонстрации работы потока.

Обработчик события btnLoadLibClick() вызывает функцию LoadLibrary() для загрузки библиотеки DllEntryLib.dll, в результате чего указанная библиотека загружается и отображается на адресное пространство процесса. Кроме того, в библиотеке DLL выполняется подпрограмма инициализации, содержащаяся в блоке begin..end, которая запускает процедуру входа-выхода для этой библиотеки:

```
begin
  { Сначала назначаем процедуру переменной DLLProc }
  DllProc := @DLLEntryPoint;
  { Затем вызываем эту процедуру для отображения факта присоединения
    библиотеки DLL к процессу }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

На протяжении всей работы процесса раздел инициализации будет вызван только один раз. Если другой процесс загрузит ту же самую библиотеку DLL, этот раздел будет вызван снова, но уже в контексте нового процесса, поскольку процессы не разделяют экземпляры библиотек DLL.

Обработчик события btnFreeLibClick() выгружает библиотеку DLL с помощью вызова функции FreeLibrary(). В этом случае вызывается процедура DLLEntryProc(), на которую указывает переменная DLLProc; в качестве параметра ей передается значение DLL_PROCESS_DETACH.

Обработчик события btnCreateThreadClick() создает объект, являющийся потомком класса TThread. При этом вызывается процедура DLLEntryProc(), и ей в качестве параметра передается значение DLL_THREAD_ATTACH. Обработчик событий btnFreeThreadClick() вызывает ту же процедуру DLLEntryProc(), но на этот раз с параметром, равным DLL_THREAD_DETACH. Несмотря на то, что в нашем примере при возникновении событий выводится всего лишь окно сообщений, эти события можно использовать для любых операций инициализации либо сворачивания любого процесса или потока, которые могут потребоваться в приложении. Ниже рассматривается применение этого метода для установки совместно используемых глобальных данных DLL. Приведенный выше пример содержится в проекте DLLEntryTest.dpr, помещенном на компакт-диск, прилагаемый ко второму тому (см. также www.williamspublishing.com.).

Исключительные ситуации и библиотеки DLL

В этом разделе рассматриваются вопросы, связанные с исключительными ситуациями и их обработкой в библиотеках DLL и системе Win32.

Перехват исключительных ситуаций в 16-разрядной Delphi

В среде 16-разрядной Delphi 1 объекты исключений имели специфический для этой среды формат. Поэтому в случае возникновения исключительных ситуаций при работе функций библиотек DLL их приходилось перехватывать до выхода из подпрограмм DLL. В противном случае они разрушали стек вызова модулей основного приложения, что неминуемо приводило к фатальному исходу. Во избежание таких неприятностей следовало каждую точку входа в библиотеку DLL снабжать собственным обработчиком исключительных ситуаций, как показано в этом примере:

```
procedure SomeDLLProc;
begin
  try
    { Выполнение инструкций }
  except
     on Exception do
        { Чтобы исключительная ситуация не вышла из-под контроля, ее нужно
 обработать, дабы не допустить ее повторного возникновения }
  end;
end;
```

Такое положение вещей не распространилось дальше Delphi 2. Исключительные ситуации Delphi 5 прямо отображаются в исключительные ситуации системы Win32. Исключительные ситуации, возникающие в функциях библиотек DLL, больше не зависят от компилятора Delphi, а обрабатываются средствами системы Win32.

Для реализации последнего утверждения необходимо включить в инструкцию `uses` исходного файла библиотеки DLL модуль `SysUtils`. В противном случае поддержка исключительных ситуаций средствами Delphi внутри библиотеки DLL не действует.

В большинстве приложений Win32 обработка исключительных ситуаций не предусмотрена, поэтому ни перевод исключений Delphi в исключения Win32, ни то, что программе удастся выйти из функции библиотеки DLL в главное приложение, скорее всего, положения не спасет: такое приложение будет закрыто.

Если основное приложение построено с помощью Delphi или C++ Builder, больших проблем не возникнет, но ведь не секрет, что по-прежнему в ходу масса программ, в которых исключительным ситуациям не уделяется должного внимания.

Следовательно, чтобы сделать функции библиотеки DLL защищенными и работоспособными при их использовании в любом приложении, нужно по-прежнему прибегать к тому методу защиты точек входа в DLL, который был принят в 16-разрядн Delphi и предполагал применение блоков `try..except`, предназначенных для перехвата всех исключительных ситуаций, возникающих в функциях библиотеки DLL..

На заметку

Если приложение, написанное на языке, отличном от Delphi, использует созданную в Delphi библиотеку DLL, оно не сможет работать с классами исключительных ситуаций, зависящих от конкретной реализации языка Delphi. Однако такая ситуация может быть обработана как исключительная ситуация системы Win32, представленная кодом `$0EEDFACE`. При этом адрес точки, в которой возникла исключительная ситуация, будет первым элементом массива `ExceptionInformation` в структуре `EXCEPTION_RECORD` системы Win32. Второй элемент этого массива содержит ссылку на объект исключения Delphi. За дополнительной информацией обратитесь к интерактивной справочной системе Delphi (статья о структуре `EXCEPTION_RECORD`).

Исключительные ситуации и директива Safecall

Функции, отмеченные директивой Safecall, используются в модели COM и при обработке исключительных ситуаций. Они гарантируют, что любая исключительная ситуация будет передана источнику вызова данной функции. При этом функция с директивой Safecall преобразует исключение в возвращаемое значение HResult. Кроме того, реализации директивы Safecall заключает в себе поддержку спецификатора StdCall, определяющего соглашение о передаче параметров при вызовах функций. Допустим, что функция с директивой Safecall объявлена так:

```
function Foo(i: integer): string; Safecall;
```

В этом случае компилятор воспримет ее как определенную следующим образом:

```
function Foo(i: integer): string; HResult; StdCall;
```

Затем компилятор неявно поместит все содержимое этой функции в блок try..except, который будет перехватывать любые исключительные ситуации. Данный блок except содержит обращение к функции SafecallExceptionHandler() для преобразования объекта исключения в значение HResult. Такой подход несколько напоминает метод перехвата исключения и передачи значений ошибки, принятый в 16-разрядной Delphi.

Функции обратного вызова

Функция обратного вызова (callback function) — это функция приложения, вызываемая библиотеками DLL Win32 или другими библиотеками DLL. Фактически, в системе Windows есть всего несколько функций API, которые используют функции обратного вызова. При вызове этих функций вы передаете им адрес функции вашего приложения, которую Windows может вызывать. Если вы не понимаете, какое отношение это имеет к библиотекам DLL, то напомню, что реально функции Win32 API экспортируются из системных библиотек DLL. По сути, при передаче адреса функции обратного вызова в функцию Win32 API происходит передача этой функции в библиотеку DLL.

Одной из таких функций является функция Win32 API EnumWindows(), которая регистрирует все окна верхнего уровня. Она передает функциям обратного вызова приложений дескрипторы каждого окна, объединенные в перечисление. От вас требуется лишь передать функции EnumWindows() адрес функции обратного вызова, предварительно определив ее следующим образом:

```
function EnumWindowsProc(Hw: HWnd; lp: lParam): Boolean; stdcall;
```

Пример использования функции EnumWindows() приводится в проекте CallBack.dpr, текст которого показан в листинге 9.9. Этот проект помещен на компакт-диск, прилагаемый ко второму тому (см. также www.williamspublishing.com.).

Листинг 9.9. Пример использования функции обратного вызова

```
unit MainFrm;

interface

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

type

  { Определение записи/класса для хранения имени окна и имени класса для
    каждого окна. Экземпляры этого класса помещаются в список ListBox1 }
  TWindowInfo = class
    WindowName,          // Имя окна
    WindowClass: String; // Имя класса окна
  end;

  TMainForm = class(TForm)
    lbWinInfo: TListBox;
    btnGetWinInfo: TButton;
    hdWinInfo: THeaderControl;
    procedure btnGetWinInfoClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure lbWinInfoDrawItem(Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure hdWinInfoSectionResize(HeaderControl: THeaderControl;
      Section: THeaderSection);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
function EnumWindowsProc(Hw: HWnd; AMainForm: TMainForm):
   Boolean; stdcall;
{ Эта процедура вызывается библиотекой User32.DLL, поскольку она регистрирует
активные окна в системе. }
var
  WinName, CName: array[0..144] of char;
  WindowInfo: TWindowInfo;
begin
  { Возвращаемое значение по умолчанию равно true, и это означает
    незаконченную регистрацию окон }
  Result := True;
  GetWindowText(Hw, WinName, 144); // Получение текста текущего окна
  GetClassName(Hw, CName, 144);    // Получение имени класса окна
```

```
  { Создается экземпляр класса TWindowInfo, поля которого заполняются
    значениями, равными имени окна и имени класса окна. Затем этот объект
    добавляется в массив Objects списка ListBox1. Позже эти значения
    будут отображены в окне списка.}
  WindowInfo := TWindowInfo.Create;
  with WindowInfo do
  begin
    SetLength(WindowName, strlen(WinName));
    SetLength(WindowClass, StrLen(CName));
    WindowName := StrPas(WinName);
    WindowClass := StrPas(CName);
  end;
  // Добавление в массив Objects
  MainForm.lbWinInfo.Items.AddObject('', WindowInfo);
end;

procedure TMainForm.btnGetWinInfoClick(Sender: TObject);
begin
  { Регистрация всех отображаемых окон верхнего уровня. Передача адреса
    функции обратного вызова EnumWindowsProc, которая будет вызвана
    для каждого окна }
  EnumWindows(@EnumWindowsProc, 0);
end;

procedure TMainForm.FormDestroy(Sender: TObject);
var
  i: integer;
begin
  { Освобождение всех экземпляров класса TWindowInfo }
  for i := 0 to lbWinInfo.Items.Count - 1 do
    TWindowInfo(lbWinInfo.Items.Objects[i]).Free
end;

procedure TMainForm.lbWinInfoDrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
begin
  { Сначала очищаем прямоугольник, предназначенный для вывода информации }
  lbWinInfo.Canvas.FillRect(Rect);
  { Теперь выводим строки записи TWindowInfo, хранящейся в списке под
    номером Index. Позиции вывода каждой строки определяются разделами
     HeaderControl1 }
  with TWindowInfo(lbWinInfo.Items.Objects[Index]) do
  begin
    DrawText(lbWinInfo.Canvas.Handle, PChar(WindowName),
      Length(WindowName), Rect,dt_Left or dt_VCenter);
    { Смещаем прямоугольник рисования, используя размер разделов
      HeaderControl1 для определения позиции вывода следующей строки }
    Rect.Left := Rect.Left + hdWinInfo.Sections[0].Width;
    DrawText(lbWinInfo.Canvas.Handle, PChar(WindowClass),
      Length(WindowClass), Rect, dt_Left or dt_VCenter);
```

```
  end;
end;

procedure TMainForm.hdWinInfoSectionResize(HeaderControl:
  THeaderControl; Section: THeaderSection);
begin
  lbWinInfo.Invalidate; // Перерисовка списка ListBox1.
end;

end.
```

В этом приложении функция `EnumWindows()` используется для получения имени окна и имени класса всех окон верхнего уровня, помещаемых в нестандартный список объектов в главной форме. Главная форма использует этот нестандартный список для вывода имени окна и имени класса окна в виде отдельных столбцов. Но прежде чем разбираться в том, как создаются нестандартные списки со столбцами, рассмотрим методы использования функции обратного вызова.

Использование функции обратного вызова

В листинге 9.9 приведено определение процедуры `EnumWindowsProc()`, которой в качестве первого параметра передается дескриптор окна. Вторым параметром указываются определяемые пользователем данные, поэтому в нем можно передавать любые данные, которые вы сочтете нужными, лишь бы их размер был эквивалентен размеру целого типа данных.

`EnumWindowsProc()` — это процедура обратного вызова, адрес которой передается функции Win32 API `EnumWindows()`, поэтому она должна быть объявлена с директивой `StdCall`, указывающей на использование соглашения о вызове, принятого в системе Win32. При передаче адреса этой процедуры в функцию `EnumWindows()` предполагается, что она будет вызвана для каждого существующего в системе окна верхнего уровня, дескриптор которого будет указан в качестве первого параметра. Этот дескриптор окна используется для получения имени окна и имени его класса. Затем создается экземпляр класса `TWindowInfo`, и его поля заполняются полученной информацией. Созданный экземпляр добавляется в массив `lbWinInfo.Objects`. Информация, содержащаяся в объектах этого массива, будет выведена в виде списка, включающего несколько столбцов данных.

Следует помнить, что в обработчике события `OnDestroy` главной формы обязательно должны освобождаться любые созданные экземпляры класса `TWindowInfo`.

В обработчике события `btnGetWinInfoClick()` выполняется вызов процедуры `EnumWindows()` с передачей ей в качестве первого параметра адреса процедуры `EnumWindowsProc()`.

Запустив это приложение и щелкнув на кнопке в его форме, вы увидите полученную от каждого окна информацию, представленную в виде списка.

Отображение нестандартного списка

Имена окон и имена классов всех окон верхнего уровня выводятся в виде отдельных столбцов в объекте под именем `lbWinInfo`. Это — экземпляр класса `TListBox`, в котором свойству `Style` присвоено значение `lbOwnerDraw`. При выборе данного стиля событие `TListBox.OnDrawItem` генерируется всякий раз, когда в компоненте `TListBox` требуется отобразить очередной элемент данных. При этом вся ответственность за отображение данных возлагается на программиста, что позволяет ему самостоятельно выбирать способ их представления.

В листинге 9.9 обработчик событий lbWinInfoDrawItem() содержит весь код, осуществляющий вывод элементов списка. В данном случае выводятся строки, содержащиеся в экземплярах класса TWindowInfo, помещенных в массив lbWinInfo.Objects. Эти значения были получены от функции обратного вызова EnumWindowsProc(). Чтобы разобраться в работе подпрограммы обработки данного события, проанализируйте комментарии, помещенные в ее текст.

Обращение к функциям обратного вызова из библиотеки DLL

Аналогично тому, как адреса функции обратного вызова передаются функциям библиотек DLL, можно организовать и вызов этих функций из программ библиотек DLL. В этом разделе будет показано, как создать библиотеку DLL, в которой экспортируемой функции в качестве параметра передается адрес процедуры обратного вызова. В случае получения от пользователя адреса процедуры обратного вызова последняя будет вызвана функцией библиотеки DLL. В листинге 9.10 содержится исходный код подобной библиотеки DLL.

Листинг 9.10. Исходный текст библиотеки StrSrchLib.dll — пример вызова функций обратного вызова из библиотеки DLL

```
library StrSrchLib;

uses
  Wintypes,
  WinProcs,
  SysUtils,
  Dialogs;

type
 { Объявление типа функции обратного вызова }
 TFoundStrProc = procedure(StrPos: PChar); StdCall;

function SearchStr(ASrcStr, ASearchStr: PChar;  AProc: TFarProc):
  Integer; StdCall;
{ Эта функция выполняет поиск подстроки ASearchStr в строке ASrcStr.
  В случае обнаружения искомой подстроки ASearchStr вызывается процедура
  обратного вызова, заданная параметром Aproc (если он был передан).
  Если в качестве значения этого параметра пользователь передал значение
  nil, эта процедура вызываться не будет. }
var
  FindStr: PChar;
begin
  FindStr := ASrcStr;
  FindStr := StrPos(FindStr, ASearchStr);
  while FindStr <> nil do
  begin
    if AProc <> nil then
      TFoundStrProc(AProc)(FindStr);
    FindStr := FindStr + 1;
```

```
    FindStr := StrPos(FindStr, ASearchStr);
  end;
end;

exports
  SearchStr;
begin

end.
```

В этой библиотеке DLL также определяется процедурный тип TFoundStrProc, предназначенный для функции обратного вызова. Он используется для выполнения приведения типа функции обратного вызова при осуществлении ее вызова.

Собственно вызов функции обратного вызова осуществляется в экспортируемой процедуре SearchStr(). Работа этой процедуры поясняется в комментариях.

Пример использования этой библиотеки DLL приведен в проекте CallBackDemo.dpr, помещенном в каталог \DLLCallBack, расположенный на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.). Исходный текст главной формы этого демонстрационного проекта приведен в листинге 9.11.

Листинг 9.11. Главная форма программы, демонстрирующей работу с библиотекой StrSrchLib.dll

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    btnCallDLLFunc: TButton;
    edtSearchStr: TEdit;
    lblSrchWrd: TLabel;
    memStr: TMemo;
    procedure btnCallDLLFuncClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;
  Count: Integer;

implementation

{$R *.DFM}

{ Определение экспортируемой процедуры библиотеки DLL }
function SearchStr(ASrcStr, ASearchStr: PChar; AProc: TFarProc):
```

```
   Integer; StdCall external 'STRSRCHLIB.DLL';

{ Определение процедуры обратного вызова с обязательным
  использованием директивы StdCall }
procedure StrPosProc(AStrPsn: PChar); StdCall;
begin
  inc(Count); // Увеличение значения переменной Count на единицу
end;

procedure TMainForm.btnCallDLLFuncClick(Sender: TObject);
var
  S: String;
  S2: String;
begin
  Count := 0; // Инициализация переменной Count значением, равным нулю
  { Определение длины текста, в котором ищется подстрока. }
  SetLength(S, memStr.GetTextLen);
  { Копирование этого текста в переменную S }
  memStr.GetTextBuf(PChar(S), memStr.GetTextLen);
  { Копирование искомого текста в строковую переменную, чтобы ее
    можно было передать функции библиотеки DLL }
  S2 := edtSearchStr.Text;
  { Вызов функции библиотеки DLL }
  SearchStr(PChar(S), PChar(S2), @StrPosProc);
  { Вывод сведений о том, сколько раз заданное слово встречается в строке.
    Результат запоминается в переменной Count, которая используется
    в функции обратного вызова }
  ShowMessage(Format('%s %s %d %s', [edtSearchStr.Text, 'occurs', Count,
'times.']));
end;

end.
```

В этом приложении используется элемент управления класса `TMemo` под именем `memStr`. Свойство `EdtSearchStr.Text` компонента класса `TEdit` содержит строку, которая будет отыскиваться в содержимом компонента `memStr`. Содержимое объекта `memStr` передается функции `SearchStr()` библиотеки DLL в качестве исходной строки, а значение свойства `edtSearchStr.Text` — в качестве искомой строки.

Функция `StrPosProc()` — это используемая функция обратного вызова. Она увеличивает на единицу значение глобальной переменной `Count`, предназначенной для хранения числа вхождений искомой строки в текст, помещенный в компонент `memStr`.

Разделение данных библиотеки DLL между различными процессами

В среде 16-разрядной Windows управление памятью библиотек DLL происходило совсем не так, как в среде 32-разрядной Win32. Одна из самых характерных особенностей работы 16-разрядных библиотек DLL заключалась в совместном использовании глобальной памяти раз-

личными приложениями. Иными словами, если в функции 16-разрядной библиотеки DLL объявляется глобальная переменная, то к ней получит доступ любое приложение, использующее эту DLL. Изменения, внесенные в эту переменную одним приложением, будут видны всем остальным приложениям.

В определенных случаях такое поведение могло быть опасным, поскольку одно приложение способно перезаписать данные, от которых зависит работа другого приложения. В других случаях разработчики специально использовали эту особенность системы.

В среде Win32 подобного разделения глобальных данных библиотек DLL больше не существует. Поскольку каждый процесс приложения отображает библиотеку DLL на свое собственное адресное пространство, данные библиотеки также отображаются на это адресное пространство. В результате каждое приложение получает собственный экземпляр данных библиотеки DLL. Поэтому изменения, внесенные в глобальные данные библиотеки DLL одним приложением, не будут видны другому.

Если планируется работа с 16-разрядным приложением, которое опирается на общедоступность глобальных данных DLL, можно (как и прежде) предоставить такому приложению возможность совместного использования данных DLL с другими приложениями. Этот процесс не является автоматическим, и для хранения общих данных используются отображенные в память файлы, о которых речь пойдет в главе 12, “Работа с файлами”. Здесь же мы воспользуемся ими лишь для иллюстрации этого метода. Однако, после того как вы прочтете главу 12 и ближе познакомитесь с отображенными в память файлами, желательно вернуться к этому разделу и просмотреть его еще раз.

Создание библиотек DLL с общим полем памяти

В листинге 9.12 содержится исходный текст файла проекта библиотеки DLL, позволяющей обращающимся к ее функциям приложениям совместно использовать глобальные данные. Эти глобальные данные хранятся в переменной под именем `GlobalData`.

Листинг 9.12. Библиотека DLL `ShareLib`, иллюстрирующая совместное использование глобальных данных

```
library ShareLib;

uses
  ShareMem, Windows, SysUtils, Classes;

const
  cMMFileName: PChar = 'SharedMapData';

{$I DLLDATA.INC}

var
  GlobalData : PGlobalDLLData;
  MapHandle  : THandle;

{ Процедура GetDLLData является экспортируемой функцией DLL }
procedure GetDLLData(var AGlobalData: PGlobalDLLData); StdCall;
begin
  { Указатель AGlobalData будет указывать на тот же адрес памяти,
    на который ссылается и указатель GlobalData. }
```

```
  AGlobalData := GlobalData;
end;

procedure OpenSharedData;
var
  Size: Integer;
begin
  { Определение размера отображаемых данных. }
  Size := SizeOf(TGlobalDLLData);

  { Теперь получаем объект отображенного в память файла. Обратите внимание
    на то, что первый параметр передает значение $FFFFFFFF, или DWord(-1),
    чтобы пространство выделялось из системного файла подкачки.
    Для этого нужно, чтобы имя отображенного в память объекта
    передавалось в качестве последнего параметра. }

  MapHandle := CreateFileMapping(DWord(-1), nil, PAGE_READWRITE, 0,
    Size, cMMFileName);

  if MapHandle = 0 then
    RaiseLastWin32Error;
  { Отображение данных на адресное пространство вызывающего процесса
    и получение указателя на начало этого адреса }
  GlobalData := MapViewOfFile(MapHandle, FILE_MAP_ALL_ACCESS, 0, 0, Size);
  { Инициализация этих данных }
  GlobalData^.S := 'ShareLib';
  GlobalData^.I := 1;
  if GlobalData = nil then
  begin
    CloseHandle(MapHandle);
    RaiseLastWin32Error;
  end;
end;

procedure CloseSharedData;
{ Эта процедура закрывает отображенный в память файл и освобождает
  его дескриптор. }
begin
  UnmapViewOfFile(GlobalData);
  CloseHandle(MapHandle);
end;

procedure DLLEntryPoint(dwReason: DWord);
begin
  case dwReason of
    DLL_PROCESS_ATTACH: OpenSharedData;
    DLL_PROCESS_DETACH: CloseSharedData;
  end;
end;
```

```
exports
  GetDLLData;

begin
  { Сначала назначаем процедуру переменной DLLProc }
  DllProc := @DLLEntryPoint;
  { Теперь вызываем эту процедуру для отражения факта присоединения
    данной библиотеки DLL к процессу }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

Переменная `GlobalData` имеет тип `PGlobalDLLData`, который определяется во включаемом файле `DllData.inc`. Этот файл содержит следующее определение типа (обратите внимание на то, что файл включения присоединяется с помощью директивы включения `$I`):

```
type

  PGlobalDLLData = ^TGlobalDLLData;
  TGlobalDLLData = record
    S: String[50];
    I: Integer;
  end;
```

В этой библиотеке DLL используется тот же процесс, который рассматривался выше в этой главе при обсуждении добавления кода инициализации/завершения библиотеки DLL в виде процедуры входа-выхода. Упомянутая процедура называется `DLLEntryPoint()`, как показано в листинге. Когда процесс загружает эту библиотеку DLL, вызывается метод `OpenSharedData()`, а когда процесс отсоединяется от нее, — метод `CloseSharedData()`.

Мы не будем углубляться в подробности использования отображенных в память файлов — должное внимание им уделяется в главе 12, “Работа с файлами”. Но для того чтобы вы могли понять назначение этой библиотеки DLL, необходимо уже сейчас разъяснить некоторые моменты.

Файлы, отображенные в память, позволяют резервировать некоторую область адресного пространства в системе Win32, для которой назначаются страницы физической памяти. Этот процесс напоминает выделение памяти с возможностью ссылки на нее с помощью указателя. Однако в случае отображенных в память файлов в это адресное пространство можно отобразить любой дисковый файл, а затем с помощью указателя ссылаться на адресное пространство внутри файла так же, как на любую другую область памяти.

Для работы с отображенным в память файлом необходимо сначала получить дескриптор для существующего на диске файла, для которого будет создан объект отображения в память. Затем этот объект отображается на указанный файл. В начале этой главы мы обсудили, как система разделяет библиотеки DLL между несколькими приложениями. Вначале библиотека загружается в память, а затем каждому приложению предоставляется его собственный образ этой DLL. В результате создается впечатление, что каждое приложение загрузило отдельный экземпляр библиотеки. Но в действительности в памяти находится только один экземпляр файла DLL. Эффект множественности достигается за счет использования механизма отображенных в память файлов. Тот же процесс можно использовать и для предоставления доступа к файлам данных. Для этого нужно лишь выполнить необходимые обращения к функциям Win32 API, которые обеспечивают создание и доступ к файлам, отображенным в память.

Теперь рассмотрим следующий сценарий. Предположим, что некоторое приложение (назовем его App1) создает отображенный в память файл, который связывается с дисковым файлом `MyFile.dat`. Приложение App1 может теперь считывать и записывать данные в этот файл. Если во время работы приложения App1 другое приложение, App2, также будет выполнять отображение в тот же файл, то изменения, внесенные App1, будут видны App2. На самом деле все обстоит несколько сложнее: для немедленного переноса на диск внесенных в файл изменений необходимо установить определенные флажки, а также выполнить некоторую другую подготовительную работу. Но в наших рассуждениях все эти подробности не имеют значения. Достаточно сказать, что изменения будут восприняты обоими приложениями, поскольку это возможно.

Один из допустимых вариантов использования отображенных в память файлов — создание файлового отображения на основе файла подкачки (страничного обмена) Win32, а не обычного файла данных. Это значит, что вместо отображения в память файла, реально существующего на диске, можно зарезервировать некоторую область памяти, на которую затем ссылаться как на дисковый файл. Это избавит вас от необходимости создания и уничтожения временного файла, если все, что требуется, заключается лишь в создании некоторого адресного пространства, доступного нескольким процессам. Система Win32 автоматически управляет своим файлом подкачки, поэтому, когда память больше не требуется, она освобождается системой.

Выше был предложен сценарий, иллюстрирующий, как два приложения могут получить доступ к одним и тем же файловым данным с помощью файла, отображенного в память. Тот же сценарий подходит и к разделению данных между приложением и библиотекой DLL. Действительно, если библиотека DLL, загружаемая некоторым приложением, создает отображенный в память файл, то после загрузки этой библиотеки другим приложением она будет продолжать использовать все тот же отображенный в память файл. При этом будут существовать два образа библиотеки DLL (по одному для каждого вызывающего приложения), использующие один и тот же экземпляр файла, отображенного в память. Функции библиотеки DLL могут обеспечить доступ к данным отображенного в память файла любому вызывающему их приложению. Когда одно приложение внесет в эти данные изменения, они будут видны другому приложению, поскольку оба приложения обращаются к одним и тем же данным, но отображенным двумя различными экземплярами отображенных в память библиотек. Описанный метод и используется в примере, представленном в листинге 9.12.

В этом примере создание отображенного в память файла возложено на процедуру `OpenSharedData()`. Для создания исходного объекта файлового отображения в ней используется функция `CreateFileMapping()`, которая передает созданный объект функции `MapViewOfFile()`. Эта функция отображает образ файла на адресное пространство вызывающего процесса и возвращает адрес начала выделенного адресного пространства. Теперь вспомним, что это адресное пространство принадлежит вызывающему процессу. Для двух различных приложений, использующих данную библиотеку DLL, эти адресные области могут быть различными, несмотря на то, что данные, на которые они ссылаются, одни и те же.

На заметку

Первым параметром, передаваемым функции `CreateFileMapping()`, является дескриптор файла, который требуется отобразить в память. Но если отображение выполняется для адресного пространства системного файла подкачки, передается значение `$FFFFFFFF`, что равносильно значению `DWord(-1)` — как в нашем примере. В качестве второго параметра функции `CreateFileMapping()` нужно передать имя объекта отображения файла в память. Это имя система будет использовать для ссылки на соответствующее файловое отображение. Если сразу несколько процессов создадут отображенный в память файл, используя при этом одно и то же имя, объекты отображения будут ссылаться на одну и ту же системную память.

После обращения к функции MapViewOfFile() переменная GlobalData указывает на адресное пространство файла, отображенного в память. В результате выполнения экспортируемой функции GetDLLData() параметр AGlobalData будет указывать на ту же область памяти, что и указатель GlobalData. Параметр AGlobalData передается из вызывающего приложения, следовательно, вызывающее приложение получит возможность считывать и записывать общие данные.

Процедура CloseSharedData() закрывает отображение общего файла для данного вызывающего процесса и освобождает его объект файлового отображения. Это никак не влияет на другие объекты файлового отображения и на отображение глобального файла со стороны других приложений.

Использование библиотек DLL с общей памятью

Для иллюстрации использования библиотеки DLL с общей памятью мы создали два работающих с ней приложения. Первое приложение (проект App1.dpr) позволяет модифицировать глобальные данные библиотеки DLL. Второе приложение (проект App2.dpr) обращается к этим данным и периодически обновляет два компонента TLabel, используя для этой цели компонент TTimer. При запуске обоих приложений вы сможете увидеть разделяемый доступ к данным библиотеки DLL, т.е. окно приложения App2 будет отображать изменения, внесенные в окне приложения App1.

В листинге 9.13 содержится исходный текст проекта APP1.

Листинг 9.13. Главная форма приложения App1.dpr

```
unit MainFrmA1;

interface

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

{$I DLLDATA.INC}

type

  TMainForm = class(TForm)
    edtGlobDataStr: TEdit;
    btnGetDllData: TButton;
    meGlobDataInt: TMaskEdit;
    procedure btnGetDllDataClick(Sender: TObject);
    procedure edtGlobDataStrChange(Sender: TObject);
    procedure meGlobDataIntChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  public
    GlobalData: PGlobalDLLData;
  end;
```

```
var
  MainForm: TMainForm;

{ Определение экспортируемой процедуры библиотеки DLL }
procedure GetDLLData(var AGlobalData: PGlobalDLLData);
  StdCall External 'SHARELIB.DLL';

implementation

{$R *.DFM}

procedure TMainForm.btnGetDllDataClick(Sender: TObject);
begin
  { Получение указателя на данные библиотеки DLL }
  GetDLLData(GlobalData);
  { Обновление элементов управления для отражения значений полей GlobalData }
  edtGlobDataStr.Text := GlobalData^.S;
  meGlobDataInt.Text  := IntToStr(GlobalData^.I);
end;

procedure TMainForm.edtGlobDataStrChange(Sender: TObject);
begin
  { Обновление данных библиотеки DLL в соответствии с внесенными изменениями }
  GlobalData^.S := edtGlobDataStr.Text;
end;

procedure TMainForm.meGlobDataIntChange(Sender: TObject);
begin
  { Обновление данных библиотеки DLL в соответствии с внесенными изменениями }
  if meGlobDataInt.Text = EmptyStr then
    meGlobDataInt.Text := '0';
  GlobalData^.I := StrToInt(meGlobDataInt.Text);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  btnGetDllDataClick(nil);
end;

end.
```

В это приложение также входит включаемый файл `DllData.inc`, в котором определяется тип данных `TGlobalDLLData` и указатель на эти данные. Обработчик события `btnGetDllDataClick()` с помощью вызова функции `GetDLLData()` получает указатель на глобальные данные библиотеки DLL, доступ к которым возможен с помощью отображенного в память файла, принадлежащего библиотеке. Затем, используя значение этого указателя `GlobalData`, выполняется обновление элементов управления формы. Обработчики события `OnChange` для полей ввода изменяют значение области памяти, на которую указывает переменная `GlobalData`. Поскольку указатель `GlobalData` ссылается на глобальные данные библиотеки DLL, модификации подвергаются данные, на которые ссылается отображенный в память файл, созданный библиотекой DLL.

В листинге 9.14 представлен исходный текст главной формы приложения App2.dpr.

Листинг 9.14. Исходный текст главной формы приложения App2.dpr

```
unit MainFrmA2;

interface

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

{$I DLLDATA.INC}

type

  TMainForm = class(TForm)
    lblGlobDataStr: TLabel;
    tmTimer: TTimer;
    lblGlobDataInt: TLabel;
    procedure tmTimerTimer(Sender: TObject);
  public
    GlobalData: PGlobalDLLData;
  end;

{ Определяем экспортируемую процедуру DLL }
procedure GetDLLData(var AGlobalData: PGlobalDLLData);
  StdCall External 'SHARELIB.DLL';

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.tmTimerTimer(Sender: TObject);
begin
  GetDllData(GlobalData);  // Получение доступа к данным
  { Отображение содержимого полей GlobalData.}
  lblGlobDataStr.Caption := GlobalData^.S;
  lblGlobDataInt.Caption := IntToStr(GlobalData^.I);
end;

end.
```

Эта форма содержит два компонента TLabel, которые обновляются во время обработки события OnTimer объекта tmTimer. Если пользователь изменит значения глобальных данных библиотеки DLL в приложения App1, приложение App2 отобразит эти изменения.

Вы можете запустить оба приложения и поработать с ними. Эти проекты содержаться на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Экспорт объектов из библиотек DLL

К объекту и его методам можно получить доступ даже в том случае, если этот объект содержится внутри библиотеки DLL. Однако к объявлению такого объекта внутри библиотеки DLL предъявляются определенные требования, а на его использование накладываются некоторые ограничения. Представленный здесь подход может быть полезен в весьма специфических ситуациях. Как правило, такого же эффекта можно достичь за счет применения пакетов или интерфейсов.

На экспорт объектов из библиотек DLL накладываются следующие ограничения.

- Вызывающее приложение может использовать лишь те методы объекта, которые были объявлены как виртуальные.
- Экземпляры объектов должны создаваться только внутри библиотеки DLL.
- Экспортируемый объект должен быть определен как в библиотеке DLL, так и в вызывающем приложении, причем объявление методов должно выполняться в одном и том же порядке.
- Нельзя создать объект-потомок из объекта, содержащегося внутри библиотеки DLL.

Здесь перечислены лишь основные ограничения, но возможны и некоторые другие.

Для иллюстрации этой технологии подготовлен простой, но достаточно наглядный пример экспортируемого объекта. Этот объект содержит функцию, возвращающую заданную строку, написанную прописными или строчными буквами — в зависимости от значения другого параметра, определяющего желаемый регистр. Описание этого объекта приведено в листинге 9.15.

Листинг 9.15. Объект, предназначенный для экспорта из библиотеки DLL

```
type

  TConvertType = (ctUpper, ctLower);

  TStringConvert = class(TObject)
{$IFDEF STRINGCONVERTLIB}
  private
    FPrepend: String;
    FAppend : String;
{$ENDIF}
  public
    function ConvertString(AConvertType: TConvertType; AString: String):
    String; virtual; stdcall; {$IFNDEF STRINGCONVERTLIB} abstract; {$ENDIF}
{$IFDEF STRINGCONVERTLIB}
    constructor Create(APrepend, AAppend: String);
    destructor Destroy; override;
{$ENDIF}
  end;

{ Для любого приложения, использующего этот класс, библиотека STRINGCONVERTLIB не
определена, а значит, определение этого класса будет эквивалентно следующему:

  TStringConvert = class(TObject)
```

```
  public
    function ConvertString(AConvertType: TConvertType; AString: String):
      String; virtual; stdcall; abstract;
  end;
}
```

В листинге 9.15 фактически содержится исходный код включаемого файла `StrConvert.inc`. Размещение этого объекта во включаемом файле вызвано необходимостью соблюдения третьего пункта приведенного выше списка требований, согласно которому объект должен быть одинаково определен как в библиотеке DLL, так и в вызывающем приложении. При размещении описания объекта во включаемом файле, как вызывающее приложение, так и библиотека DLL смогут подключить этот файл. Если же в объект будут внесены изменения, то потребуется лишь повторно скомпилировать оба проекта, а не вводить эти изменения дважды — сначала в вызывающее приложение, а затем в библиотеку DLL, — что создает благоприятную почву для появления ошибок.

Рассмотрим следующее определение метода `ConvertSring()`:

```
function ConvertString(AConvertType: TConvertType; AString: String):
  ↳String; virtual; stdcall;
```

Причина объявления этого метода виртуальным (с помощью ключевого слова `virtual`) заключается не в необходимости создавать объекты-потомки, в которых можно было бы впоследствии переопределять метод `ConvertString()`. Он объявлен виртуальным с целью создания точки входа в таблицу виртуальных методов (Virtual Method Table — VMT). Здесь мы не будем вдаваться в подробности работы этой таблицы (ей уделено внимание в главе 13, “Дополнительный инструментарий разработчика”). Пока рассматривайте таблицу VMT как некоторый блок памяти, содержащий указатели на виртуальные методы объекта. С помощью таблицы VMT вызывающее приложение может получить указатель на определенный метод конкретного объекта. Если не объявить метод виртуальным, таблица VMT не будет содержать строки для этого метода, и вызывающее приложение будет лишено возможности получить на него указатель. Как видите, все это делается ради получения в вызывающем приложении указателя на функцию. Но поскольку этот указатель зависит от типа метода, определяемого в объекте, Delphi автоматически обрабатывает любые адресные привязки, неявным образом передавая методу дополнительный параметр `self`.

Обратите внимание на условное определение библиотеки `STRINGCONVERTLIB`. При экспорте объекта переопределение в вызывающем приложении необходимо лишь тем методам, к которым нужно обеспечить внешний доступ из DLL. Кроме того, эти методы можно определить как абстрактные, чтобы избежать генерации ошибки времени компиляции. Это вполне допустимо, поскольку во время выполнения программы эти методы будут реализованы в коде DLL. Рассмотрев приведенный в листинге комментарий, вы поймете, как выглядит объект `TStringConvert` со стороны приложения.

В листинге 9.16 демонстрируется реализация объекта `TStringConvert`.

Листинг 9.16. Реализация объекта `TStringConvert`

```
unit StringConvertImp;
{$DEFINE STRINGCONVERTLIB}

interface
uses SysUtils;
```

```
{$I StrConvert.inc}

function InitStrConvert(APrepend, AAppend: String): TStringConvert; stdcall;

implementation

constructor TStringConvert.Create(APrepend, AAppend: String);
begin
  inherited Create;
  FPrepend := APrepend;
  FAppend  := AAppend;
end;

destructor TStringConvert.Destroy;
begin
  inherited Destroy;
end;

function TStringConvert.ConvertString(AConvertType:
  TConvertType; AString: String): String;
begin
  case AConvertType of
    ctUpper: Result := Format('%s%s%s', [FPrepend, UpperCase(AString), FAppend]);
    ctLower: Result := Format('%s%s%s', [FPrepend, LowerCase(AString), FAppend]);
  end;
end;

function InitStrConvert(APrepend, AAppend: String): TStringConvert;
begin
  Result := TStringConvert.Create(APrepend, AAppend);
end;

end.
```

Согласно предъявляемым к экспортируемым объектам требованиям, такой объект должен создаваться в библиотеке DLL. Это условие реализуется в стандартной экспортируемой функции `InitStrConvert()`, которой передаются два параметра, предназначенных для конструктора. Мы добавили это для иллюстрации способа передачи информации конструктору объекта через функцию интерфейса.

Обратите внимание: в этом модуле определяется условная директива `STRINGCONVERTLIB`. Остальная часть модуля не нуждается в дополнительных разъяснениях. Файл проекта библиотеки DLL представлен в листинге 9.17.

Листинг 9.17. Файл проекта для библиотеки `StringConvertLib.dll`

```
library StringConvertLib;
uses
  ShareMem,
  SysUtils,
  Classes,
```

```
  StringConvertImp in 'StringConvertImp.pas';

exports
  InitStrConvert;
end.
```

В тексте этой библиотеки не должно быть ничего нового для вас. Однако следует обратить внимание на использование модуля ShareMem. Его имя должно быть объявлено первым в файле проекта библиотеки, а также в файле проекта вызывающего приложения. Этот момент очень важен и о нем не следует забывать.

В листинге 9.18 показан пример использования экспортируемого объекта для преобразования произвольной строки в строку из прописных или строчных букв. Проект этого примера называется StrConvertTest.dpr. Его можно найти на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Листинг 9.18. Проект, демонстрирующий использование объекта преобразования строк

```
unit MainFrm;

interface

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

{$I strconvert.inc}

type


  TMainForm = class(TForm)
    btnUpper: TButton;
    edtConvertStr: TEdit;
    btnLower: TButton;
    procedure btnUpperClick(Sender: TObject);
    procedure btnLowerClick(Sender: TObject);
  private
  public
  end;

var
  MainForm: TMainForm;

function InitStrConvert(APrepend, AAppend: String): TStringConvert; stdcall;
  external 'STRINGCONVERTLIB.DLL';

implementation

{$R *.DFM}
```

```
procedure TMainForm.btnUpperClick(Sender: TObject);
var
  ConvStr: String;
  FStrConvert: TStringConvert;
begin
  FStrConvert := InitStrConvert('Upper ', ' end');
  try
    ConvStr := edtConvertStr.Text;
    if ConvStr <> EmptyStr then
      edtConvertStr.Text := FStrConvert.ConvertString(ctUpper, ConvStr);
  finally
    FStrConvert.Free;
  end;
end;

procedure TMainForm.btnLowerClick(Sender: TObject);
var
  ConvStr: String;
  FStrConvert: TStringConvert;
begin
  FStrConvert := InitStrConvert('Lower ', ' end');
  try
    ConvStr := edtConvertStr.Text;
    if ConvStr <> EmptyStr then
      edtConvertStr.Text := FStrConvert.ConvertString(ctLower, ConvStr);
  finally
    FStrConvert.Free;
  end;
end;

end.
```

Резюме

Разработка библиотек DLL — неотъемлемая часть создания приложений Windows. Их роль особенно велика, когда требуется многократное использование некоторого программного кода. В этой главе рассказывалось о создании и использовании библиотек DLL в приложениях Delphi, а также о различных методах их загрузки. Кроме того, были отмечены некоторые специальные соглашения, которые необходимо учитывать при использовании библиотек DLL в среде Delphi, а также показано, как сделать данные в библиотеках DLL доступными различным приложениям.

Вооружившись этими знаниями, вы сможете самостоятельно создавать библиотеки DLL в среде Delphi и без труда использовать их в своих Delphi-приложениях. Дополнительные сведения о работе с библиотеками DLL будут приведены и в других главах этой книги.

Печать в Delphi 5

Глава

10

Печать в Windows — головная боль многих программистов. Но унывать не стоит — Delphi существенно упрощает все, что связано с печатью. Приложив минимум усилий, можно написать простые процедуры для вывода на принтер как текста, так и растровых изображений. Конечно, для выполнения более сложных задач печати придется познакомиться с некоторыми концепциями и освоить несколько специализированных приемов программирования, однако ваши усилия будут вознаграждены, и былые трудности с печатью забудутся как дурной сон.

На заметку

Во вкладке QReport палитры компонентов вы найдете ряд компонентов фирмы QuSoft, предназначенных для составления отчета. Документация к этому набору инструментов находится в файле интерактивной справки `QuickRpt.hlp`.
Инструменты QuSoft удобны для приложений, в которых генерируются сложные отчеты. Однако они ограничивают возможности управления процессом печати в исходном тексте программ, а значит, и степень вашего контроля над тем, что и как будет печататься. Обсуждение использования инструментов QuickReport выходит за рамки этой главы, которая целиком посвящена рассмотрению методов создания в Delphi отчетов произвольной формы.

В среде Delphi подпрограммы класса `TPrinter`, инкапсулирующего механизмы печати в Windows, выполняют б*о*льшую часть рутинной работы, которую в противном случае вам пришлось бы проделать самостоятельно.

В этой главе мы обсудим, как с помощью объекта `TPrinter` выполнить любую из операций печати. Сначала мы рассмотрим самые простые средства Delphi, способные существенно упростить вывод данных на печать, а затем обсудим методы создания более сложных подпрограмм печати, которые помогут вам добиться в этой области любых желаемых результатов.

Объект `TPrinter`

Объект `TPrinter` инкапсулирует интерфейс печати Windows, делая невидимой б*о*льшую часть управления процессом печати. С помощью методов и свойств класса `TPrinter` можно выполнять печать на его канву так, как будто выходные данные просто рисуются на поверхности формы. При первом обращении к функции `Printer()` она возвращает глобальный экземпляр класса `TPrinter`, свойства и методы которого перечислены в табл. 10.1 и 10.2.

Таблица 10.1. Свойства класса TPrinter

Свойство	Назначение
`Aborted`	Переменная типа `Boolean`, которая определяет, не прекратил ли пользователь выполнение задания печати
`Canvas`	Поверхность печати для текущей страницы
`Fonts`	Список, содержащий имена шрифтов, поддерживаемых принтером
`Handle`	Уникальный номер, представляющий дескриптор устройства для принтера. См. врезку “Дескрипторы”в главе 20 второго тома, “Ключевые элементы VCL и информация о типах времени выполнения”
`Orientation`	Переменная, значение которой определяет горизонтальную (`poLandScape`) или вертикальную (`poPortrait`) ориентацию листа печати
`PageHeight`	Высота поверхности печатаемой страницы в пикселях

Окончание табл. 10.1

Свойство	Назначение
PageNumber	Номер печатаемой страницы. Это значение увеличивается с каждым последовательным обращением к методу TPrinter.NewPage()
PageWidth	Ширина поверхности печатаемой страницы в пикселях
PrinterIndex	Индекс, определяющий выбранный принтер из списка доступных в системе пользователя
Printers	Список доступных принтеров в системе
Printing	Переменная типа Boolean, которая определяет, продолжается ли печать задания
Title	Определяемый пользователем текст, который идентифицирует задание печати в окне Print Manager и сетевых страницах

Таблица 10.2. Методы класса TPrinter

Метод	Назначение
Abort()	Прерывает процесс печати задания
BeginDoc()	Начинает процесс печати задания
EndDoc()	Завершает процесс печати задания. (Метод EndDoc завершает печать задания по окончании печати, а метод Abort может прервать выполнение задания до окончания печати)
GetPrinter()	Считывает индекс текущего принтера
NewPage()	Заставляет принтер начать печать на новой странице и увеличивает значение свойства PageCount
SetPrinter()	Определяет указанный принтер в качестве текущего

Свойство TPrinter.Canvas

Свойство TPrinter.Canvas во многом напоминает канву объекта формы. Оно представляет поверхность рисования, на которую выводится текст и графика. Разница лишь в том, что поверхность рисования объекта TPrinter предназначена для вывода на принтер, а не на экран. Большинство процедур, применяемых для рисования фигур, вывода текста и графических изображений, может аналогичным образом использоваться и для вывода на принтер. Однако при печати следует учитывать и некоторые отличия, приведенные ниже.

- Вывод на экран является *динамическим* процессом — всегда можно затереть то, что уже отображено на экране. Вывод на принтер не обладает такой гибкостью: все, что было нанесено на канву класса TPrinter (т.е. присвоено свойству TPrinter.Canvas), распечатается на принтере.
- Вывод текста или графики на экран происходит практически мгновенно, в то время как принтер работает гораздо медленнее, даже если используются самые высокоскоростные лазерные принтеры. Поэтому необходимо дать пользователю возможность прервать задание печати либо с помощью диалогового окна Abort, либо каким-то другим способом.

- Поскольку пользователи работают под управлением Windows, резонно предположить, что их дисплеи поддерживают вывод графических изображений. Но для принтеров такого допущения сделать нельзя. Различные принтеры обладают различными характеристиками. Одни из них могут иметь высокое разрешение, другие — очень низкое, а некоторые могут вообще не поддерживать печать графических изображений. Это нужно обязательно учитывать в своих программах печати.
- Вы никогда не увидите сообщение о том, что у монитора закончилось "дисплейное пространство" и необходимо добавить новую его порцию. Но вы обязательно столкнетесь с сообщением об ошибке, вызванной отсутствием бумаги в принтере. В Windows NT/2000 и Windows 95/98 предусмотрена обработка такой ошибки (на случай ее возникновения). Но вы должны предоставить пользователю возможность отменить печать в подобной ситуации.
- Текст и графика на дисплеях и в напечатанном виде выглядят по-разному. У принтеров и дисплеев различные разрешения. Растровое изображение 300×300 пикселей может выглядеть эффектно на дисплее с разрешением 640×480, но на лазерном принтере с разрешением 300 dpi (точек на дюйм) это будет просто клякса размером 1×1 дюйм. Все программы печати должны быть построены так, чтобы пользователи при чтении напечатанных данных могли обойтись без увеличительного стекла.

Простая печать

Во многих случаях требуется посылать на принтер поток текстовой информации, не отягощенной никаким специальным форматированием или условиями определенного размещения текста. Delphi без труда справляется с такими простыми задачами — именно это и обсуждается в следующих разделах.

Печать содержимого компонента TMemo

Напечатать строки текста с помощью процедуры `AssignPrn()`и в самом деле не представляет труда. Эта процедура позволяет назначить текущему принтеру переменную с текстовым файлом. Эта переменная используется также в процедурах `Rewrite()` и `CloseFile()`, как показано ниже.

```
var
  f: TextFile;
begin
  AssignPrn(f);
  try
    Rewrite(f);
    writeln(f, 'Print the output');
  finally
    CloseFile(f);
  end;
end;
```

Печать строки текста на принтере выполняется так же, как и вывод строки текста в файл. При этом используется следующий синтаксис:

```
writeln(f, 'Это моя строка текста');
```

В главе 16, "MDI-приложения" показано, как добавить команду меню для печати содержимого формы TMdiEditForm. В листинге 10.1 демонстрируется, как напечатать произвольное содержимое из объекта TMdiEditForm. Аналогичный подход можно использовать для печати текста практически из любого источника.

Листинг 10.1. Подпрограмма печати для формы TMdiEditForm

```
procedure TMdiEditForm.mmiPrintClick(Sender: TObject);
var
  i: integer;
  PText: TextFile;
begin
  inherited;
  if PrintDialog.Execute then
  begin
    AssignPrn(PText);
    Rewrite(PText);
    try
      Printer.Canvas.Font := memMainMemo.Font;
      for i := 0 to memMainMemo.Lines.Count -1 do
        writeln(PText, memMainMemo.Lines[i]);
    finally
      CloseFile(PText);
    end;
  end;
end;
```

Обратите внимание, что шрифт, использованный в поле редактирования, также назначается свойству Font объекта Printer. Это обеспечивает использование при печати того шрифта, который был применен в объекте memMainMemo.

Необходимо иметь в виду, что при печати принтер будет использовать заданный свойством Printer.Font шрифт только в том случае, если данный тип принтера его поддерживает. В противном случае будет применен шрифт, который приближается по своим характеристикам к заданному.

Печать растрового изображения

Печать растрового изображения также не вызывает трудностей. В проекте MdiApp, представленном в главе 16, "MDI-приложения", демонстрируется, как распечатать растровое изображение в форме TMdiBmpForm. Текст используемого в проекте обработчика событий приведен в листинге 10.2.

Для печати растрового изображения с помощью метода TCanvas.StretchDraw() требуется всего три строки кода. Такое существенное упрощение печати растровых изображений стало возможным благодаря тому, что начиная с Delphi 3 растровые изображения по умолчанию имеют формат DIB, а это как раз тот формат, который требуется драйверу принтера. Если окажется, что у вас есть дескриптор растра, который хранится не в формате DIB, его можно скопировать (с помощью операции Assign) во временный объект TBitmap, принудительно преобразовав это промежуточное изображение в формат DIB посредством присвоения значения bmDIB свойству TBitmap.HandleType, и лишь затем выполнить печать растра в новом формате.

Листинг 10.2. Текст процедуры печати объекта `TMdiBmpForm`

```
procedure TMdiBMPForm.mmiPrintClick(Sender: TObject);
begin
  inherited;

  with ImgMain.Picture.Bitmap do
  begin
    Printer.BeginDoc;
    Printer.Canvas.StretchDraw(Canvas.ClipRect, imgMain.Picture.Bitmap);
    Printer.EndDoc;
  end;   { конец блока with }
end;
```

На заметку

Один из ключевых моментов вывода на принтер — возможность печатать изображения так, как они выглядят на экране, т.е. приблизительно с теми же размерами. Например, для графического изображения размером 3×3 дюйма на экране с разрешением 640×480 используется меньше пикселей, чем потребовалось бы для печати на принтере с разрешением 300 dpi. Поэтому нужно растянуть изображение на канве объекта `TPrinter`, как это было сделано в предыдущем примере — с помощью обращения к функции `TCanvas.StretchDIBits()`. Еще один способ заключается в выводе изображений с помощью установки специального режима отображения (эти режимы описаны в главе 8, "GDI, шрифты и графика"). При этом следует иметь в виду, что некоторые старые принтеры могут не поддерживать растягивания изображений. Нужную информацию о возможностях конкретного принтера можно получить с помощью функций Win32 API `GetDeviceCaps()`.

Печать данных в формате RTF (Rich Text Format)

Для печати содержимого компонента `TRichEdit` необходимо вызвать всего лишь один метод. В следующем фрагменте показано, как это сделать. Этот код также используется для печати содержимого объекта `TMdiRtfForm` в модуле `MdiApp` (глава 16, "MDI-приложения").

```
procedure TMdiRtfForm.mmiPrintClick(Sender: TObject);
begin
  inherited;
  reMain.Print(Caption);
end;
```

Печать формы

Концептуально печать формы могла бы стать одной из самых трудных задач для реализации, но благодаря методу `Print()` компонента `TForm` библиотеки VCL эта задача значительно упрощается. Следующая процедура (объемом в одну строку) распечатывает область клиента формы вместе со всеми расположенными в этой области компонентами:

```
procedure TForm1.PrintMyForm(Sender: TObject);
begin
  Print;
end;
```

На заметку Печать формы можно считать черновым вариантом печати графических изображений. Однако из-за процесса отсечки Windows на принтер выводится только то, что является видимым на экране дисплея. При этом сначала создается растр с плотностью задействованных экранных пикселей, а затем выполняется растягивание до разрешения принтера. Текст, расположенный на форме, также выводится с разрешением экрана, а не принтера и только потом растягивается, поэтому напечатанное изображение формы имеет заметно зазубренный вид. Это значит, что для печати сложных графических изображений нужно применять более продуманные методы, которые рассматриваются далее в этой главе.

Печать сложных документов

Довольно часто возникает необходимость в печати специфических документов, при этом обычные инструменты разработки или средства создания отчетов, предлагаемые сторонними производителями, оказываются бессильными. В этом случае приходится рассчитывать только на собственные силы и решать задачи печати на низком уровне. В следующих нескольких разделах показано, как пишутся такие процедуры вывода на принтер, а также представлена методология, которую можно применить ко всем задачам печати.

На заметку Несмотря на то что этот раздел посвящен процессу печати, вам следует знать, что за время написания этой книги массовому пользователю стали доступны несколько компонентов печати сторонних производителей, способные удовлетворить большинство ваших потребностей в этой области. Демонстрационные версии некоторых из этих инструментов находятся на компакт-диске, прилагаемом ко второму тому (см. также `www.williamspublishing.com`.).

Печать отчета с несколькими столбцами

Во многих приложениях, особенно в тех, где используются базы данных, необходим вывод на печать отчетов разных типов. Одним из самых распространенных стилей создания отчетов является табличный, когда информация выводится в виде столбцов.

В приведенном ниже проекте отчет с несколькими столбцами распечатывается на основе информации в одной из таблиц, содержащихся в демонстрационных каталогах Delphi. Каждая страница отчета содержит верхний заголовок, названия столбцов и несколько строк записей.

На рис. 10.1 показана главная форма этого проекта. С помощью спаренных компонентов `TEdit`/`TUpDown` пользователь может указать ширину каждого столбца в десятых долях дюйма. Используя компоненты `TUpDown`, можно задать минимальные или максимальные значения. Элемент управления класса `TEdit1` под именем `edtHeaderFont` содержит текст заголовка, который можно распечатать с помощью шрифта, отличного от того, что используется в остальной части отчета.

В листинге 10.3 представлен исходный текст этого проекта. Обработчик событий `mmiPrintClick()` предназначен для выполнения следующих действий.

1. Инициализация задания печати.
2. Печать заголовка.

3. Печать названий столбцов.
4. Печать страницы.
5. Повторение пп. 2, 3 и 4 до окончания печати.
6. Завершение задания печати.

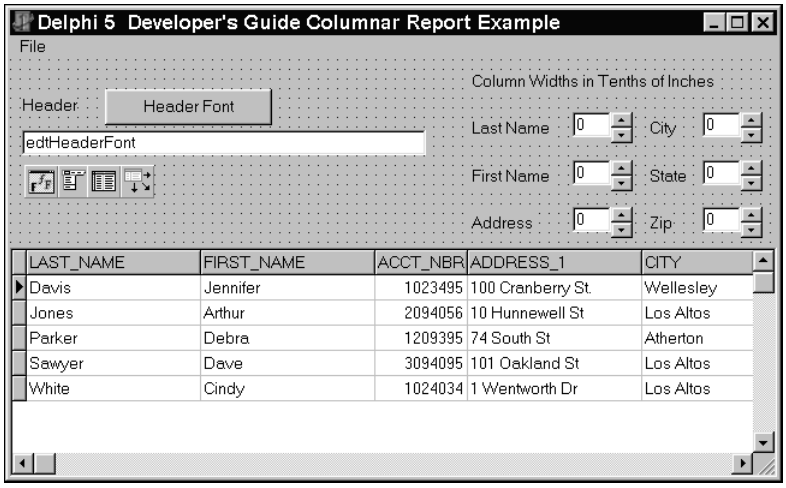


Рис. 10.1. Главная форма отчета, оформленного в виде столбцов

Листинг 10.3. Пример приложения для создания отчета в виде столбцов

```
unit MainFrm;
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, DBGrids, DB, DBTables, Menus, StdCtrls, Spin,
  Gauges, ExtCtrls, ComCtrls;

type
  TMainForm = class(TForm)
  { Исходный текст компонентов, не включенных в этот листинг, можно
    найти в исходных текстах приложения на прилагаемом компакт-диске }
  procedure mmiPrintClick(Sender: TObject);
    procedure btnHeaderFontClick(Sender: TObject);
  private
    PixelsInInchx: integer;
    LineHeight: Integer;
   { Отслеживание вертикального размера текста, уже отпечатанного
     на странице, в пикселях. }
    AmountPrinted: integer;
   { Количество пикселей в 1/10 дюйма. Это значение используется
     для межстрочного интервала. }
    TenthsOfInchPixelsY: integer;
```

```
    procedure PrintLine(Items: TStringList);
    procedure PrintHeader;
    procedure PrintColumnNames;
  end;

var
  MainForm: TMainForm;

implementation
uses printers, AbortFrm;

{$R *.DFM}

procedure TMainForm.PrintLine(Items: TStringList);
var
  OutRect: TRect;
  Inches: double;
  i: integer;
begin
  // Сначала устанавливаем прямоугольник печати на канве принтера
  OutRect.Left := 0;
  OutRect.Top := AmountPrinted;
  OutRect.Bottom := OutRect.Top + LineHeight;
  With Printer.Canvas do
    for i := 0 to Items.Count - 1 do
    begin
      Inches := longint(Items.Objects[i]) * 0.1;
      // Определяем правый край
      OutRect.Right := OutRect.Left + round(PixelsInInchx*Inches);
      if not Printer.Aborted then
       // Печатаем строку
        TextRect(OutRect, OutRect.Left, OutRect.Top, Items[i]);
      // Выравниваем правый край
      OutRect.Left := OutRect.Right;
    end;
  { По мере печати каждой строки значение переменной AmountPrinted
    должно увеличиваться для отражения того, какая часть страницы уже
    напечатана, причем расчет ведется на основе высоты строки. }
  AmountPrinted := AmountPrinted + TenthsOfInchPixelsY*2;
end;

procedure TMainForm.PrintHeader;
var
  SaveFont: TFont;
begin
{ Сохраняем текущий шрифт принтера, затем устанавливаем новый шрифт на основе
выбора, сделанного с помощью элемента управления Edit1. }
  SaveFont := TFont.Create;
  try
    Savefont.Assign(Printer.Canvas.Font);
    Printer.Canvas.Font.Assign(edtHeaderFont.Font);
```

```
    // Сначала печатаем заголовок
    with Printer do
    begin
      if not Printer.Aborted then
        Canvas.TextOut((PageWidth div 2)-(Canvas.TextWidth(edtHeaderFont.Text)
                       div 2),0, edtHeaderFont.Text);
     // Увеличиваем AmountPrinted на значение LineHeight
      AmountPrinted := AmountPrinted + LineHeight+TenthsOfInchPixelsY;
    end;
    // Восстанавливаем прежний шрифт канвы объекта Printer
    Printer.Canvas.Font.Assign(SaveFont);
  finally
    SaveFont.Free;
  end;
end;

procedure TMainForm.PrintColumnNames;
var
  ColNames: TStringList;
begin
  { Создаем экземпляр класса TStringList для хранения названий столбцов
    и их размещения, причем ширина каждого столбца определяется на основе
    значений элементов управления TEdit. }
  ColNames := TStringList.Create;
  try
    // Печатаем названия столбцов, полужирным шрифтом с подчеркиванием
    Printer.Canvas.Font.Style := [fsBold, fsUnderline];

    with ColNames do
    begin
      // Сохраняем названия столбцов и значения ширины в объекте TStringList
      AddObject('LAST NAME',  pointer(StrToInt(edtLastName.Text)));
      AddObject('FIRST NAME', pointer(StrToInt(edtFirstName.Text)));
      AddObject('ADDRESS',    pointer(StrToInt(edtAddress.Text)));
      AddObject('CITY',       pointer(StrToInt(edtCity.Text)));
      AddObject('STATE',      pointer(StrToInt(edtState.Text)));
      AddObject('ZIP',        pointer(StrToInt(edtZip.Text)));
    end;

    PrintLine(ColNames);
    Printer.Canvas.Font.Style := [];
  finally
    ColNames.Free;  // Освобождаем экземпляр TStringList с именами столбцов
  end;
end;

procedure TMainForm.mmiPrintClick(Sender: TObject);
var
  Items: TStringList;
begin
  { Создаем экземпляр класса TStringList для хранения полей и значений
```

```
  ширины столбцов, в которых они будут выведены, на основе содержимого
  строк ввода. }
Items := TStringList.Create;
try
  // Определяем количество пикселей на дюйм по горизонтали
  PixelsInInchx := GetDeviceCaps(Printer.Handle, LOGPIXELSX);
  TenthsOfInchPixelsY := GetDeviceCaps(Printer.Handle, LOGPIXELSY) div 10;
  AmountPrinted := 0;
  MainForm.Enabled := False; // Отключаем родительскую форму
  try
  Printer.BeginDoc;
    AbortForm.Show;
    Application.ProcessMessages;
    { Вычисляем высоту строки на базе высоты текста с учетом текущего шрифта. }
    LineHeight := Printer.Canvas.TextHeight('X')+TenthsOfInchPixelsY;
    if edtHeaderFont.Text <> '' then
      PrintHeader;
    PrintColumnNames;
    tblClients.First;
    { Сохраняем каждое значение поля в объекте TStringList, а также
      соответствующую ширину столбца. }
    while (not tblClients.Eof) or Printer.Aborted do
    begin
      Application.ProcessMessages;
      with Items do
      begin
        AddObject(tblClients.FieldByName('LAST_NAME').AsString,
                  pointer(StrToInt(edtLastName.Text)));
        AddObject(tblClients.FieldByName('FIRST_NAME').AsString,
                  pointer(StrToInt(edtFirstName.Text)));
        AddObject(tblClients.FieldByName('ADDRESS_1').AsString,
                  pointer(StrToInt(edtAddress.Text)));
        AddObject(tblClients.FieldByName('CITY').AsString,
                  pointer(StrToInt(edtCity.Text)));
        AddObject(tblClients.FieldByName('STATE').AsString,
                  pointer(StrToInt(edtState.Text)));
        AddObject(tblClients.FieldByName('ZIP').AsString,
                  pointer(StrToInt(edtZip.Text)));
      end;
      PrintLine(Items);
      { Принудительный переход на новую страницу для данного
        задания печати, если превышена высота страницы. }
      if AmountPrinted + LineHeight > Printer.PageHeight then
      begin
        AmountPrinted := 0;
        if not Printer.Aborted then
          Printer.NewPage;
        PrintHeader;
        PrintColumnNames;
      end;
      Items.Clear;
```

```
        tblClients.Next;
      end;
      AbortForm.Hide;
      if not Printer.Aborted then
        Printer.EndDoc;
    finally
      MainForm.Enabled := True;
    end;
  finally
    Items.Free;
  end;
end;

procedure TMainForm.btnHeaderFontClick(Sender: TObject);
begin
  { Назначение шрифта, выбранного с помощью объекта FontDialog1,
    элементу управления Edit1. }
  FontDialog.Font.Assign(edtHeaderFont.Font);
  if FontDialog.Execute then
    edtHeaderFont.Font.Assign(FontDialog.Font);
end;

end.
```

В обработчике событий `mmiPrintClick()` сначала создается экземпляр класса `TStringList` для хранения строковых значений, составляющих строку, предназначенную для печати. Затем в переменной `PixelsPerInchX` определяется количество пикселей на дюйм вдоль вертикальной оси, и эта величина используется для вычисления значений ширины столбцов. Значение переменной `TenthsOfInchPixelsY` используется для образования междустрочного интервала размером в 0,1 дюйма. Переменная `AmountPrinted` служит для накопления общего числа пикселей вдоль вертикальной оси поверхности печати после каждой отпечатанной строки. Когда значение переменной `AmountPrinted` превысит значение свойства `Printer.PageHeight`, это послужит сигналом к началу новой страницы.

Если в свойстве `edtHeaderFont.Text` указан текст заголовка, то он будет распечатан с помощью процедуры `PrintHeader()`. А процедура `PrintColumnNames()` предназначена для печати названий столбцов для каждого поля (эти две процедуры рассматриваются далее в данном разделе). И, наконец, распечатываются записи таблицы.

В следующем цикле выполняется проход по всем записям таблицы `tblClients`, и внутри каждой записи печатаются выбранные поля:

```
while (not tblClients.Eof) or Printer.Aborted do begin
```

Внутри этого цикла с помощью метода `AddObject()` значения полей добавляются в экземпляр класса `TStringList`. При этом запоминается как строковое значение поля, так и ширина столбца. Ширина столбца добавляется в свойство-массив `Items.Objects`. Затем объект `Items` передается процедуре `PrintLine()`, которая распечатывает эти строки в заданном формате (в виде столбцов).

В приведенном выше листинге не раз встречалась ссылка на свойство `Printer.Aborted`. С помощью этого свойства определяется, не прервал ли пользователь задание печати (читайте следующий раздел).

Свойства-массивы объектов класса Object с именами TStrings и TStringList удобны для хранения целочисленных значений. С помощью методов AddObject() или InsertObject() можно поместить на хранение в этих массивах любое число, не превышающее значения MaxLongInt. Поскольку методу AddObject() в качестве второго параметра передается ссылка на объект типа TObject, необходимо подвергнуть этот параметр операции приведения типа, чтобы он был передан как указатель:

```
MyList.AddObject('SomeString', pointer(SomeInteger));
```

Для считывания значения используется операция приведения к типу Longint:

```
MyInteger := Longint(MyList.Objects[Index]);
```

Затем в обработчике событий определяется, не будет ли превышена высота страницы при печати новой строки:

```
if AmountPrinted + LineHeight > Printer.PageHeight then
```

Если результат сравнения окажется равным True, переменной AmountPrinted вновь присваивается нулевое значение и вызывается метод Printer.NewPage для перехода на новую страницу, на которой снова печатаются верхний заголовок и названия столбцов. После того как будут отпечатаны все записи таблицы tblClients, для завершения задания печати вызывается метод Printer.EndDoc.

Процедура PrintHeader(), используя значения свойств edtHeaderFont.Text и edtHeaderFont.Font, печатает верхний заголовок, центрированный в верхней строке каждой страницы. После этого переменная AmountPrinted увеличивается и восстанавливается прежний стиль шрифта объекта Printer.

Процедура PrintColumnNames() предназначена для вывода на печать названий столбцов отчета. В этом методе имена столбцов добавляются в объект ColNames (экземпляр класса TStringList), который затем передается методу PrintLine(). Обратите внимание на то, что названия столбцов печатаются полужирным шрифтом и подчеркиваются, для чего предварительно выполняется соответствующая установка свойства Printer.Canvas.Font.

Процедура PrintLine(), которой передается аргумент Items типа TStringList, печатает каждое строковое значение в массиве Items на одной строке в виде столбцов. Переменная OutRect хранит параметры ограничительного прямоугольника на канве объекта Printer, в который выводится текст. Переменная OutRect передается методу TextRect() вместе с текстом, предназначенным для вывода на печать. Поскольку значения элементов массива Items.Object[i] измеряются в десятых долях дюйма, то после умножения каждого элемента на число 0,1 получается значение свойства OutRect.Right. Внутри цикла for объект OutRect пересчитывается вдоль той же оси X для перехода к начальной позиции следующего столбца и вывода следующего строкового значения. Наконец, переменная AmountPrinted увеличивается на значение, равное сумме LineHeight + TenthsOfInchPixelsY.

И хотя этот отчет полностью функционален, вы могли бы расширить его возможности за счет включения нижнего колонтитула, нумерации страниц и даже установки полей.

Прерывание процесса печати

Как уже отмечалось в этой главе, пользователи должны иметь возможность прервать процесс печати уже после его запуска. Для этого следует использовать процедуру TPrinter.Abort() и свойство Aborted. Текст, приведенный в листинге 10.3, содержит соответствующие действия. Чтобы добавить в программу печати логику, обеспечивающую прерывание процесса печати, ваш код должен отвечать трем условиям.

- Необходимо обеспечить событие, при обработке которого вызывалась бы процедура `Printer.Abort`, прерывающая процесс печати.
- Перед вызовом любой из функций объекта `TPrinter` (например, `TextOut()`, `NewPage()` и `EndDoc()`) нужно обязательно проверять результат сравнения (`TPrinter.Aborted = True`).
- Процесс печати должен завершаться, как только проверка значения свойства `TPrinter.Aborted` покажет, что оно равно `True`.

Первому условию может удовлетворить простое диалоговое окно типа Abort, присутствующее в предыдущем примере. Это окно должно содержать кнопку, вызывающую прекращение печати.

Обработчик событий этой кнопки должен просто вызвать процедуру `TPrinter.Abort`, которая завершит задание печати и отменит любые запросы, уже сделанные к объекту `TPrinter`.

Рассмотрим код, содержащийся в модуле `MainForm.pas` и предназначенный для отображения формы `AbortForm` сразу после вызова процедуры `TPrinter.Begindoc()`.

```
Printer.BeginDoc;
AbortForm.Show;
Application.ProcessMessages;
```

Поскольку форма `AbortForm` отображается как немодальное диалоговое окно, то обращение к методу `Application.ProcessMessages` гарантирует, что это окно появится до продолжения обработки каких бы то ни было действий, связанных с печатью.

Для удовлетворения второго требования проверка `TPrinter.Aborted = True` выполняется перед вызовом любого метода класса `TPrinter`. Свойство `Aborted` устанавливается равным `True`, когда из формы `AbortForm` вызывается метод `Abort()`. В качестве примера перед вызовом метода `Printer.TextRect` проверим результат сравнения `Aborted = True`:

```
if not Printer.Aborted then
  TextRect(OutRect, OutRect.Left, OutRect.Top, Items[i]);
```

Кроме того, после вызова метода `Abort()` не следует вызывать функцию `EndDoc()` или любой другой метод вывода объекта `TPrinter.Canvas`, поскольку принтер будет заблокирован.

Для удовлетворения третьему требованию в этом примере используется циклическая конструкция `while not Table.Eof`, которая также проверяет, не равно ли свойство `TPrinter.Aborted` значению `True`. При подтверждении равенства выполняется принудительный выход из цикла, реализующего логику печати.

Печать конвертов

В предыдущем примере был показан метод печати отчета в виде столбцов. И хотя описанный способ несколько сложнее, чем простой вызов функций вывода на принтер `writeln()`, тем не менее это был пример построчной печати. При печати конвертов возникают новые факторы, которые несколько усложняют задачу. Во-первых, объекты, подлежащие выводу на принтер, необходимо расположить в определенных позициях на поверхности печати. Во-вторых, единицы измерения этих элементов могут отличаться от единиц измерения канвы принтера. Если учитывать оба фактора, процесс печати превращается в нечто большее, чем простой вывод на принтер строки и отслеживание объема использованного пространства.

На примере печати конверта демонстрируется пошаговый процесс, который можно использовать для печати всего чего угодно. Имейте в виду, что все нарисованное на канве принтера выводится внутри некоторого ограничительного прямоугольника на канве или в определенные точки этой канвы.

Теоретическое изучение задачи

Попробуем подойти к поставленной задаче более абстрактно. В любом случае для ее решения необходима поверхность, на которой будет выполняться печать, и один или несколько элементов, предназначенных для нанесения на эту поверхность. Взгляните на рис. 10.2.

На рис. 10.2 прямоугольник A представляет собой целевую поверхность, а прямоугольники B и C — элементы, которые вы собираетесь наложить (т.е. напечатать) на прямоугольник A. Предположим, для каждого прямоугольника действует такая система координат, в которой значения координат возрастают при перемещении вправо вдоль оси X и вниз вдоль оси Y. Описанная система координат изображена на рис. 10.3, а результат объединения прямоугольников — на рис. 10.4.

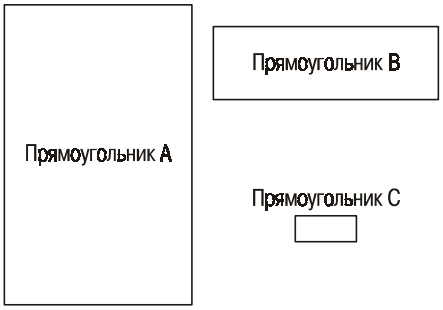


Рис. 10.2. Три прямоугольника

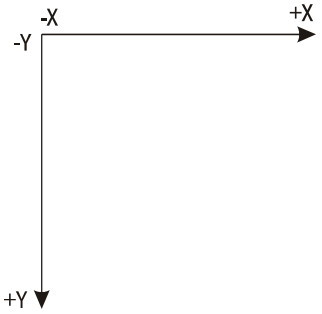


Рис. 10.3. Система координат для прямоугольников A, B и C

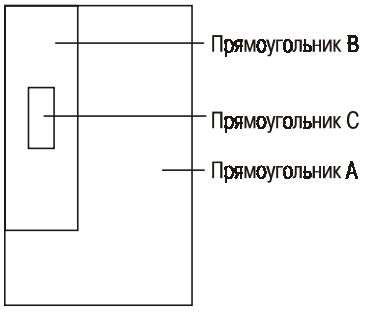


Рис. 10.4. Прямоугольники B и C, наложенные на прямоугольник A

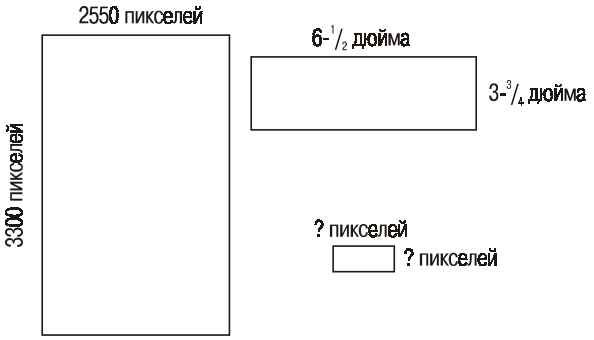


Рис. 10.5. Размеры прямоугольников

Обратите внимание, что для достижения конечного результата прямоугольники B и C повернуты на 90°. Но это еще полбеды. Если бы единицы измерения всех этих прямоугольников были одинаковы, то, применяя простейшую геометрию, можно было легко напечатать эти прямоугольники в требуемом виде. Но что делать, если единицы измерения у них различны?

Предположим, что прямоугольник A представляет поверхность, для которой единицы измерения даны в пикселях; ее размеры — 2550×3300 пикселей. Прямоугольник B измеряется в дюймах: $6^1/_2 \times 3^3/_4$ дюймов. Предположим также, что вам не известны размеры прямоугольника C (вы узнаете их позже), но известно, что они измеряются в пикселях. Размеры прямоугольников показаны на рис. 10.5.

С помощью этой абстракции иллюстрируется одна из проблем, связанных с печатью на конвертах. Прямоугольник A представляет страницу принтера с разрешением 300 dpi (при 300 dpi $8^1/_2 \times 11$ дюймов равны 2550×3300 пикселям). Прямоугольник B представляет конверт, размеры которого указаны в дюймах, а C — ограничительный прямоугольник для печати текста адреса. Но вы, конечно, понимаете, что эта абстракция не привязана только к конвертам. Прямоугольники B и C с тем же успехом могут представлять, например, компоненты `TImage`, размеры которых измеряются в миллиметрах.

Итак, рассмотрев эту задачу совершенно абстрактно, мы уже наметили три первых шага, которые необходимо выполнить для печати в Windows: идентификация всех элементов, предназначенных для печати, идентификация единицы измерения целевой поверхности и идентификация единиц измерения каждого отдельного элемента, предназначенного для наложения на целевую поверхность.

Если до сих пор мы следовали прямой дорогой, т.е. все было просто и понятно, то теперь перед нами поворот — кстати, в буквальном смысле. При печати конверта в вертикальной ориентации текст тоже нужно повернуть вертикально.

Пошаговый процесс печати

При программировании процесса печати следует выполнить следующие действия.

1. Идентифицировать все элементы, предназначенные для печати на целевой поверхности.
2. Идентифицировать единицу измерения для целевой поверхности, или канвы принтера.
3. Идентифицировать единицы измерения для каждого отдельного элемента, который должен быть нанесен на целевую поверхность.
4. Выбрать общую единицу измерения, с которой будут работать все подпрограммы вывода на печать. Как правило, отдается предпочтение единице измерения канвы принтера — пикселям.
5. Написать процедуры для преобразования всех используемых единиц измерения в ту, которая принята в качестве единой.
6. Написать процедуры для вычисления размеров каждого элемента, предназначенного для печати, в выбранной единице измерения. В языке Object Pascal для этого можно использовать структуру `TPoint`. При этом необходимо учесть зависимости от других значений. Например, ограничительный прямоугольник адреса зависит от позиции размещения конверта. Поэтому данные конверта должны быть вычислены первыми.
7. Написать процедуры для вычисления позиции размещения каждого элемента на канве принтера на основе системы координат и размеров, полученных в результате выполнения п. 6. В языке Object Pascal для этого можно использовать структуру `TRect`.
8. Написать функции вывода на принтер для размещения элементов на поверхности печати, используя данные, полученные при выполнении предыдущих пунктов.

На заметку Пункты 5 и 6 можно выполнить, используя метод реализации вывода на печать в специальном режиме отображения. Режимы отображения рассматриваются в главе 8, “GDI, шрифты и графика”.

Переходим к делу

После того как мы разобрались (теоретически) в процессе печати, задача печати конверта стала намного яснее. Все выделенные нами действия будут отражены в проекте печати конверта. Первый шаг заключается в идентификации элементов, предназначенных для вывода на принтер. Такими элементами в примере с конвертом являются сам конверт и адрес.

Из приведенного ниже примера видно, как можно описать конверты двух стандартных размеров: 10 и $6^3/_4$.

В следующей записи содержатся размеры конвертов:

```
type

  TEnvelope = record
    Kind: string;   // Название типа конверта
    Width: double;  // Ширина конверта
    Height: double; // Высота конверта
  end;

const
  // В этом массиве констант хранятся типы конвертов
  EnvArray: array[1..2] of TEnvelope =
  ((Kind:'Size 10';Width:9.5;Height:4.125),     // 9-1/2 x 4-1/8
   (Kind:'Size 6-3/4';Width:6.5;Height:3.625)); // 6-1/2 x 3-3/4
```

Пункты 2 и 3 выполнены. Мы знаем, что в роли целевой поверхности выступает канва принтера `TPrinter.Canvas`, измеряемая в пикселях. Конверты представлены в дюймах, а адрес — в пикселях. Согласно п. 4, нужно выбрать единую единицу измерения. Для этого проекта будем использовать пиксели.

Чтобы перейти к п. 5, достаточно выполнить перевод дюймов в пиксели. Вспомним, что функция Win32 API `GetDeviceCaps()` возвращает количество пикселей на один дюйм вдоль горизонтальной и вертикальной осей для объекта `TPrinter.Canvas`:

```
PixPerInX := GetDeviceCaps(Printer.Handle, LOGPIXELSX);
PixPerInY := GetDeviceCaps(Printer.Handle, LOGPIXELSY);
```

Для перевода размеров конверта в пиксели достаточно умножить число дюймов на значения `PixPerInX` или `PixPerInY`, в результате чего будут получены соответственно горизонтальные и вертикальные размеры конверта в пикселях:

```
EnvelopeWidthInPixels := trunc(EnvelopeWidthValue * PixPerInX);
EnvelopeHeightInPixels := trunc(EnvelopeHeightValue * PixPerInY);
```

Поскольку в результате операции умножения высота или ширина конверта может иметь дробное значение, необходимо использовать функцию `Trunc()` для получения целой части вещественного числа, представленного в формате с плавающей точкой. Эта функция просто отсекает дробную часть числа.

В приведенном ниже проекте демонстрируется возможная реализация пп. 6 и 7. Главная форма этого проекта показана на рис. 10.6. Исходный текст проекта печати конверта представлен в листинге 10.4.

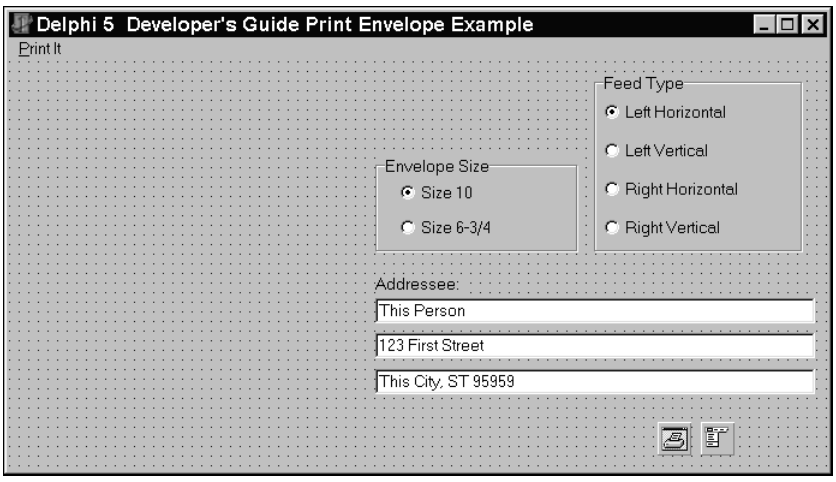


Рис. 10.6. Главная форма демонстрационного проекта печати конверта

Листинг 10.4. Демонстрационный проект печати конверта

```
unit MainFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, printers, StdCtrls, ExtCtrls, Menus, ComCtrls;

type

  TEnvelope = record
    Kind: string;   // Хранит название типа конверта
    Width: double;  // Содержит ширину конверта
    Height: double; // Содержит высоту конверта
  end;

const
  // В этом массиве констант хранятся типы конвертов
  EnvArray: array[1..2] of TEnvelope =
  ((Kind:'Size 10';Width:9.5;Height:4.125),     // 9-1/2 x 4-1/8
   (Kind:'Size 6-3/4';Width:6.5;Height:3.625)); // 6-1/2 x 3-3/4

type

  // Этот перечислимый тип представляет позиции печати
  TFeedType = (epLHorz, epLVert, epRHorz, epRVert);

  TPrintPrevPanel = class(TPanel)
  public
```

```
    property Canvas; // Открытое свойство Canvas
  end;

  TMainForm = class(TForm)
    gbEnvelopeSize: TGroupBox;
    rbSize10: TRadioButton;
    rbSize6: TRadioButton;
    mmMain: TMainMenu;
    mmiPrintIt: TMenuItem;
    lblAdressee: TLabel;
    edtName: TEdit;
    edtStreet: TEdit;
    edtCityState: TEdit;
    rgFeedType: TRadioGroup;
    PrintDialog: TPrintDialog;
    procedure FormCreate(Sender: TObject);
    procedure rgFeedTypeClick(Sender: TObject);
    procedure mmiPrintItClick(Sender: TObject);
  private
    PrintPrev: TPrintPrevPanel;  //Панель предварительного просмотра
    EnvSize: TPoint;     // Хранит размер конверта
    EnvPos: TRect;       // Хранит позицию размещения конверта
    ToAddrPos: TRect;    // Хранит позицию размещения адреса
    FeedType: TFeedType; // Хранит тип подачи из TEnvPosition
    function GetEnvelopeSize: TPoint;
    function GetEnvelopePos: TRect;
    function GetToAddrSize: TPoint;
    function GetToAddrPos: TRect;
    procedure DrawIt;
    procedure RotatePrintFont;
    procedure SetCopies(Copies: Integer);
  end;

var
  MainForm: TMainForm;

implementation
{$R *.DFM}

function TMainForm.GetEnvelopeSize: TPoint;
// Возвращает размер конверта, представленный записью типа TPoint
var
  EnvW, EnvH: integer;
  PixPerInX,
  PixPerInY: integer;
begin
  // Пиксели на дюйм вдоль горизонтальной оси
  PixPerInX := GetDeviceCaps(Printer.Handle, LOGPIXELSX);
  // Пиксели на дюйм вдоль вертикальной оси
  PixPerInY := GetDeviceCaps(Printer.Handle, LOGPIXELSY);
```

```
  // Размер конверта зависит от выбора пользователя
  if RBSize10.Checked then
  begin
    EnvW := trunc(EnvArray[1].Width * PixPerInX);
    EnvH := trunc(EnvArray[1].Height * PixPerInY);
  end
  else begin
    EnvW := trunc(EnvArray[2].Width * PixPerInX);
    EnvH := trunc(EnvArray[2].Height * PixPerInY);
  end;

  // Возвращаемое значение Result представляется в виде записи TPoint
  Result := Point(EnvW, EnvH)
end;

function TMainForm.GetEnvelopePos: TRect;
{ Возвращает позицию конверта относительно типа подачи. Эта функция
  требует, чтобы переменная EnvSize была инициализирована. }
begin
  // Определяем тип подачи на основе выбора пользователя.
  FeedType := TFeedType(rgFeedType.ItemIndex);

  { Формируем структуру TRect, определяющую позицию конверта
    при его выходе из принтера. }
  case FeedType of
   epLHorz:
     Result := Rect(0, 0, EnvSize.X, EnvSize.Y);
   epLVert:
     Result := Rect(0, 0, EnvSize.Y, EnvSize.X);
   epRHorz:
     Result := Rect(Printer.PageWidth - EnvSize.X, 0,
                 Printer.PageWidth, EnvSize.Y);
   epRVert:
     Result := Rect(Printer.PageWidth - EnvSize.Y, 0,
       Printer.PageWidth, EnvSize.X);
  end; // Конец инструкции Case
end;

function MaxLn(V1, V2: Integer): Integer;
// Возвращает большее из двух чисел. При равенстве возвращает первое
begin
  Result := V1;    // По умолчанию результат равен V1 }
  if V1 < V2 then
    Result := V2
end;

function TMainForm.GetToAddrSize: TPoint;
var
  TempPoint: TPoint;
begin
```

```
  // С помощью функции MaxLn() вычисляем размер самой длинной строки
  TempPoint.x := Printer.Canvas.TextWidth(edtName.Text);
  TempPoint.x := MaxLn(TempPoint.x, Printer.Canvas.TextWidth(edtStreet.Text));
  TempPoint.x := MaxLn(TempPoint.x,
Printer.Canvas.TextWidth(edtCityState.Text))+10;
  // Вычисляем общую высоту всех строк адреса
  TempPoint.y := Printer.Canvas.TextHeight(edtName.Text)+
    Printer.Canvas.TextHeight(edtStreet.Text)+
    Printer.Canvas.TextHeight(edtCityState.Text)+10;
  Result := TempPoint;
end;

function TMainForm.GetToAddrPos: TRect;
// Эта функция требует, чтобы записи EnvSize и EnvPos были инициализированы
Var
  TempSize: TPoint;
  LT, RB: TPoint;
begin
  // Определяем размер ограничительного прямоугольника адреса
  TempSize := GetToAddrSize;
  { Вычисляем две точки, одна из которых представляет левую верхнюю
    (Left Top – LT), а вторая – правую нижнюю (Right Bottom – RB) позицию
    ограничительного прямоугольника адреса. Их координаты зависят от
    значения FeedType. }
  case FeedType of
    epLHorz:
      begin
        LT := Point((EnvSize.x div 2) - (TempSize.x div 2),
          ((EnvSize.y div 2) - (TempSize.y div 2)));
        RB := Point(LT.x + TempSize.x, LT.y + TempSize.Y);
      end;
    epLVert:
      begin
        LT := Point((EnvSize.y div 2) - (TempSize.y div 2),
          ((EnvSize.x div 2) - (TempSize.x div 2)));
        RB := Point(LT.x + TempSize.y, LT.y + TempSize.x);
      end;
    epRHorz:
    begin
      LT := Point((EnvSize.x div 2) - (TempSize.x div 2) + EnvPos.Left,
        ((EnvSize.y div 2) - (TempSize.y div 2)));
      RB := Point(LT.x + TempSize.x, LT.y + TempSize.Y);
    end;
    epRVert:
    begin
      LT := Point((EnvSize.y div 2) - (TempSize.y div 2) + EnvPos.Left,
        ((EnvSize.x div 2) - (TempSize.x div 2)));
      RB := Point(LT.x + TempSize.y, LT.y + TempSize.x);
    end;
  end; // Конец инструкции Case
```

```
  Result := Rect(LT.x, LT.y, RB.x, RB.y);
end;

procedure TMainForm.DrawIt;
// Процедура требует, чтобы записи EnvPos и EnvSize были инициализированы
begin
  PrintPrev.Invalidate;   // Стираем содержимое панели
  PrintPrev.Update;
  // Устанавливаем режим отображения для панели равным MM_ISOTROPIC
  SetMapMode(PrintPrev.Canvas.Handle, MM_ISOTROPIC);
  // Устанавливаем размеры объекта TPanel в соответствии с границами принтера
  SetWindowExtEx(PrintPrev.Canvas.Handle,
    Printer.PageWidth, Printer.PageHeight, nil);
  { Устанавливаем размеры окна проекции равными размерам панели
    (объекта TPanel) предварительного просмотра PrintPrev.}
  SetViewPortExtEx(PrintPrev.Canvas.Handle,
    PrintPrev.Width, PrintPrev.Height, nil);
  // Устанавливаем начало координат в точке 0, 0
  SetViewportOrgEx(PrintPrev.Canvas.Handle, 0, 0, nil);
  PrintPrev.Brush.Style := bsSolid;

  with EnvPos do
   // Рисуем прямоугольник, представляющий конверт
    PrintPrev.Canvas.Rectangle(Left, Top, Right, Bottom);

  with ToAddrPos, PrintPrev.Canvas do
    case FeedType of
    epLHorz, epRHorz:
      begin
        Rectangle(Left, Top, Right, Top+2);
        Rectangle(Left, Top+(Bottom-Top) div 2, Right,
          Top+(Bottom-Top) div 2+2);
        Rectangle(Left, Bottom, Right, Bottom+2);
      end;
    epLVert, epRVert:
      begin
        Rectangle(Left, Top, Left+2, Bottom);
        Rectangle(Left + (Right-Left)div 2, Top, Left +
          (Right-Left)div 2+2, Bottom);
        Rectangle(Right, Top, Right+2, Bottom);
      end;
    end; // Конец инструкции Case
end;

procedure TMainForm.FormCreate(Sender: TObject);
var
  Ratio: double;
begin
  // Вычисляем отношение значения PageWidth к PageHeight
  Ratio := Printer.PageHeight / Printer.PageWidth;
```

```
  // Создаем новый экземпляр компонента TPanel
  with TPanel.Create(self) do
  begin
    SetBounds(15, 15, 203, trunc(203*Ratio));
    Color := clBlack;
    BevelInner := bvNone;
    BevelOuter := bvNone;
    Parent := self;
  end;

  // Создаем панель предварительного просмотра печати
  PrintPrev := TPrintPrevPanel.Create(self);

  with PrintPrev do
  begin
    SetBounds(10, 10, 200, trunc(200*Ratio));
    Color := clWhite;
    BevelInner := bvNone;
    BevelOuter := bvNone;
    BorderStyle := bsSingle;
    Parent := self;
  end;

end;

procedure TMainForm.rgFeedTypeClick(Sender: TObject);
begin
  EnvSize := GetEnvelopeSize;
  EnvPos := GetEnvelopePos;
  ToAddrPos := GetToAddrPos;
  DrawIt;
end;

procedure TMainForm.SetCopies(Copies: Integer);
var
  ADevice, ADriver, APort: String;
  ADeviceMode: THandle;
  DevMode: PDeviceMode;
begin
  SetLength(ADevice, 255);
  SetLength(ADriver, 255);
  SetLength(APort, 255);

  { Если значение ADeviceMode равно нулю, значит, драйвер принтера не загружен.
    Установка свойства PrinterIndex вызовет загрузку драйвера. }
  if ADeviceMode = 0 then
  begin
    Printer.PrinterIndex := Printer.PrinterIndex;
    Printer.GetPrinter(PChar(ADevice), PChar(ADriver), PChar(APort),
ADeviceMode);
```

```
  end;

  if ADeviceMode <> 0 then
  begin
    DevMode := GlobalLock(ADeviceMode);
    try
      DevMode^.dmFields := DevMode^.dmFields or DM_Copies;
      DevMode^.dmCopies := Copies;
    finally
      GlobalUnlock(ADeviceMode);
    end;
  end
  else
    raise Exception.Create('Could not set printer copies');
end;

procedure TMainForm.mmiPrintItClick(Sender: TObject);
var
  TempHeight: integer;
  SaveFont: TFont;
begin
  if PrintDialog.Execute then
  begin
    // Устанавливаем количество печатаемых копий
    SetCopies(PrintDialog.Copies);
    Printer.BeginDoc;
    try
      // Вычисляем временную высоту строки
      TempHeight := Printer.Canvas.TextHeight(edtName.Text);
      with ToAddrPos do
      begin
        { При вертикальной печати выполняем поворот шрифта на угол 90 °. }
        if (FeedType = eplVert) or (FeedType = epRVert) then
        begin
          SaveFont := TFont.Create;
          try
            // Сохраняем исходный шрифт
            SaveFont.Assign(Printer.Canvas.Font);
            RotatePrintFont;
            // Выводим строки адреса на канву принтера
            Printer.Canvas.TextOut(Left, Bottom, edtName.Text);
            Printer.Canvas.TextOut(Left+TempHeight+2, Bottom, edtStreet.Text);
            Printer.Canvas.TextOut(Left+TempHeight*2+2, Bottom,
              edtCityState.Text);
            // Восстанавливаем исходный шрифт
            Printer.Canvas.Font.Assign(SaveFont);
          finally
            SaveFont.Free;
          end;
        end
```

```
        else begin
          { Если конверт не печатается вертикально, строки
            адреса выводятся нормальным образом. }
          Printer.Canvas.TextOut(Left, Top, edtName.Text);
          Printer.Canvas.TextOut(Left, Top+TempHeight+2, edtStreet.Text);
          Printer.Canvas.TextOut(Left, Top+TempHeight*2+2,
            edtCityState.Text);
        end;
      end;
    finally
      Printer.EndDoc;
    end;
  end;
end;

procedure TMainForm.RotatePrintFont;
var
  LogFont: TLogFont;
begin
  with Printer.Canvas do
  begin
    with LogFont do
    begin
      lfHeight := Font.Height; // Устанавливаем равным Printer.Canvas.font.height
      lfWidth := 0;            // Разрешаем выбор ширины
      lfEscapement := 900; // Десятые доли градуса, поэтому 900 = 90°
      lfOrientation := lfEscapement; // Всегда уст. значение lfEscapement
      lfWeight := FW_NORMAL;   // По умолчанию
      lfItalic := 0;           // Не курсив
      lfUnderline := 0;        // Подчеркивания нет
      lfStrikeOut := 0;        // Перечеркивания нет
      lfCharSet := ANSI_CHARSET;  //По умолчанию
      StrPCopy(lfFaceName, Font.Name); //Имя шрифта Printer.Canvas
      lfQuality := PROOF_QUALITY;
      lfOutPrecision := OUT_TT_ONLY_PRECIS;   // Только шрифты TrueType
      lfClipPrecision := CLIP_DEFAULT_PRECIS; // По умолчанию
      lfPitchAndFamily := Variable_Pitch;     // По умолчанию
    end;
  end;
  Printer.Canvas.Font.Handle := CreateFontIndirect(LogFont);
end;

end.
```

Когда пользователь щелкает на кнопке переключателя `gbEnvelopeSize` или `gbFeedType`, вызывается обработчик событий `FeedTypeClick()`, который, в свою очередь, вызывает процедуры для вычисления размера конверта и координат размещения на основе выбранных кнопок переключателей.

В этом обработчике событий также вычисляется размер и координаты расположения ограничительного прямоугольника для адреса. Ширина этого прямоугольника определяется на

основе самой длинной строки текста, выбираемой среди трех строк, содержащихся в компонентах TEdit. Высота прямоугольника образуется путем суммирования высот упомянутых строк ввода.

Все вычисления выполняются в пикселях, т.е. в единицах измерения канвы принтера (объекта Printer.Canvas). Процедура mmiPrintItClick() содержит логику для печати конверта в соответствии с заданными параметрами. Если конверт располагается вертикально, используется дополнительная логика для выполнения поворота шрифта. Кроме того, в обработчике событий FormCreate() создается окно предварительного просмотра, содержимое которого обновляется при выборе пользователем другой кнопки переключателя.

С помощью перечислимого типа TFeedType представляются все возможные способы расположения конверта на странице принтера:

```
TFeedType = (epLHorz, epLVert, epRHorz, epRVert);
```

В описание формы TMainForm включены переменные, предназначенные для хранения размеров (переменная EnvSize типа TPoint) и положения (переменные EnvPos и ToAddrPos типа TRect) конверта и адресного прямоугольника, а также текущего значения типа подачи (переменная FeedType типа TFeedType).

В классе TMainForm объявляются методы GetEnvelopeSize(), GetEnvelopePos(), GetToAddrSize() и GetToAddrPos(), в которых определяются размеры и позиции расположения элементов, предназначенных для печати, в соответствии с пп. 6 и 7 приведенной выше модели.

Функции GetEnvelopeSize() и GetDeviceCaps() используются для преобразования "дюймовых" размеров конверта в "пиксельные" на основе выбора, сделанного пользователем в переключателе gbEnvelopeSize. Функция GetEnvelopPos() определяет позицию конверта на канве принтера на основе системы координат объекта Printer.Canvas.

Функция GetToAddrSize() вычисляет размер ограничительного прямоугольника адреса, используя результаты анализа текста, содержащегося в трех компонентах TEdit. Непосредственными исполнителями операции по определению размеров области, занимаемой текстом, являются в данном случае методы объекта Printer.Canvas TextHeight() и TextWidth(). Функция MaxLn() является вспомогательной и предназначена для определения самой длинной строки среди трех строк компонента TEdit. Именно длина этой строки и была использована в качестве ширины ограничительного прямоугольника. Для определения самой длинной строки текста можно было бы использовать и функцию Max() из модуля Math.pas.

Функция GetToAddrPos() вызывает функцию GetToAddrSize()и использует возвращаемое ею значение для определения позиции адресного прямоугольника на канве принтера. Обратите внимание, что для этого используются размеры и координаты расположения конверта.

Обработчик событий mmiPrintItClick() реализует логику вывода на печать. Сначала процесс печати инициируется с помощью метода BeginDoc(). Затем вычисляется временная высота строки, используемая для позиционирования текста. Далее анализируется переменная FeedType, и если ее значение указывает на использование одного из вертикальных типов подачи, выполняется сохранение текущего шрифта принтера и вызывается метод RotatePrintFont(), который поворачивает шрифт на 90°. После вывода на печать строк повернутым шрифтом исходный шрифт объекта Printer.Canvas восстанавливается. Если значение переменной FeedType указывает на использование одного из горизонтальных типов подачи, то с помощью метода TextOut() строки адреса печатаются в нормальном режиме. Завершается работа обработчика событий mmiPrintItClick() путем вызова метода EndDoc().

В процедуре RotatePrintFont() создается структура типа TLogFont, и ее многочисленные элементы устанавливаются на основе свойств объекта Printer.Canvas и других стандартных значений. Обратите внимание на член структуры lfEscapement. Как упоминалось в главе 8, "GDI, шрифты и графика", элемент lfEscapement определяет угол (задаваемый в десятых долях градуса), на который требуется повернуть шрифт. Поскольку нам нужно задать поворот на 90°, то элементу lfEscapement нужно присвоить число 900. Стоит также отметить, что поворачивать можно только шрифты TrueType.

Простейшее окно предварительного просмотра печати

Пользователи ваших приложений будут чрезвычайно благодарны, если вы избавите их от ошибок неправильного выбора параметров, предоставив возможность взглянуть на потенциальный результат печати до начала реального процесса печати. С этой целью в данный проект включена панель предварительного просмотра печати. В объектах программирования она реализуется путем создания потомка класса TPanel и объявления его свойства Canvas публикуемым:

```
TPrintPrevPanel = class(TPanel)
public
  property Canvas; // Делаем это свойство открытым
end;
```

Обработчик событий FormCreate() создает экземпляр компонента TPrintPrevPanel. При выполнении следующей строки определяется отношение ширины страницы принтера к ее высоте:

```
Ratio := Printer.PageHeight / Printer.PageWidth;
```

Это значение отношения используется для вычисления ширины и высоты панели, реализуемой экземпляром компонента TPrintPrevPanel.

Но перед созданием экземпляра компонента TPrintPrevPanel отображается окрашенная в черный цвет обычная панель, являющаяся другим экземпляром класса TPanel, которая будет служить "тенью" для панели предварительного просмотра. Границы "теневой" панели немного сдвигаются вправо и вниз относительно границ панели PrintPrev экземпляра компонента TPrintPrevPanel. Благодаря тени создается эффект объемного изображения панели PrintPrev. Панель PrintPrev используется главным образом для того, чтобы показать, как конверт будет напечатан на принтере. Вывод изображения на панель предварительного просмотра осуществляется процедурой DrawIt().

Прежде всего процедура TEnvPrintForm.DrawIt() вызывает метод PrintPrev.Invalidate для стирания предыдущего содержимого этой панели. Затем вызывается метод PrintPrev.Update(), гарантирующий обработку сообщения Windows до выполнения остального кода. После этого для панели PrintPrev устанавливается режим отображения MM_ISOTROPIC, позволяющий принимать произвольные величины по осям X и Y. Метод SetWindowExt() устанавливает размеры окна PrintPrev в соответствии с размерами канвы (страницы) принтера (объекта Printer.Canvas), а метод SetViewPortExt() — размеры окна проекции PrintPrev в соответствии с собственными размерами высоты и ширины панели (читайте в главе 8, "GDI, шрифты и графика", раздел, посвященный режимам отображения).

Это позволяет процедуре `DrawIt()` использовать для панели `PrintPrev` те же метрические значения, которые используются для объекта `Printer.Canvas`, конверта и прямоугольника адреса. В этой процедуре для представления текстовых строк также используются прямоугольники. Результат работы панели предварительного просмотра печати показан на рис. 10.7.

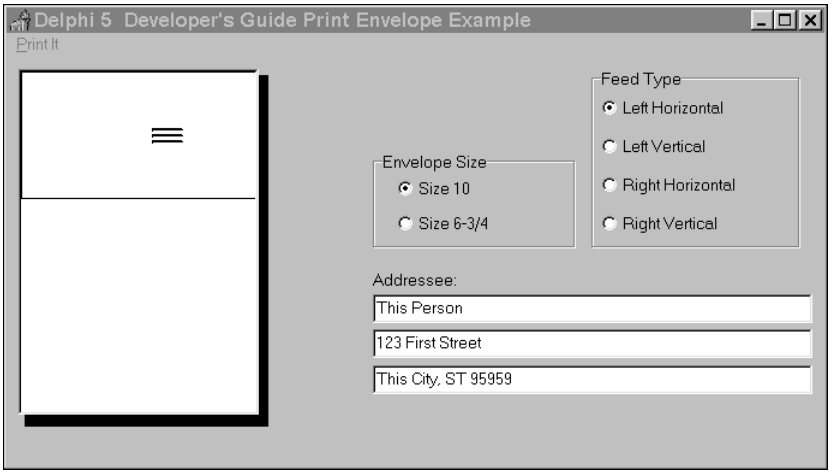


Рис. 10.7. Форма печати конверта с окном предварительного просмотра печати

На заметку

Альтернативный и лучший вариант предварительного просмотра печати можно организовать с помощью метафайлов. Создайте метафайл, используя дескриптор принтера в качестве устройства, а затем выведите то, что предназначено для печати, на канву этого метафайла, как на канву принтера. После этого выведите содержимое метафайла на экран. При этом вам не нужно делать никакого масштабирования или подстройки размеров окна проекции.

Прочие задачи печати

Время от времени вам придется решать такие задачи печати, которые не под силу объекту `TPrinter`, — например, устанавливать количество печатных копий или задавать качество печати. Для решения подобных задач потребуется прибегать к вызову функций Win32 API — и в этом нет ничего сложного. Прежде всего нужно разобраться в структуре `TDeviceMode` (этому посвящен следующий раздел), а затем научиться использовать эту структуру для выполнения разнообразных задач печати.

Структура TDeviceMode

Структура `TDeviceMode` содержит информацию об инициализации драйвера принтера и данные среды. Программисты используют эту структуру для считывания информации или для установки различных атрибутов текущего принтера. Определение этой структуры приведено в тексте модуля Windows.pas.

Описание каждого поля можно найти в интерактивной справочной системе Delphi. Наиболее употребительные поля этой структуры рассматриваются в следующих разделах, но все-таки вам стоит обратиться к справке Delphi и поинтересоваться назначением остальных ее полей. В некоторых случаях они могут сослужить хорошую службу, а кроме того, важно знать, что некоторые из них по-разному используются в Windows NT/2000 и Windows 95/98.

Чтобы получить указатель на структуру TDeviceMode текущего принтера, нужно предварительно вызвать метод TPrinter.GetPrinter() для получения дескриптора блока памяти, занимаемого этой структурой. Затем следует вызвать функцию GlobalLock(), которая возвратит указатель на интересующую нас структуру. В листинге 10.5 показано, как можно получить указатель на структуру TDeviceMode.

Листинг 10.5. Получение указателя на структуру TDeviceMode

```
var
  ADevice, ADriver, APort: array [0..255] of Char;
  DeviceHandle: THandle;
  DevMode: PDeviceMode; // Указатель на структуру TDeviceMode
begin
  { Сначала получаем дескриптор структуры DeviceMode объекта TPrinter. }
  Printer.GetPrinter(ADevice, ADriver, APort, DeviceHandle);
  { Если дескриптор DeviceHandle все еще равен 0, значит, драйвер не был
    загружен. Для принудительного выполнения загрузки установите индекс
    принтера, что сделает дескриптор доступным. }
  if DeviceHandle = 0 then
  begin
    Printer.PrinterIndex := Printer.PrinterIndex;
    Printer.GetPrinter(ADevice, ADriver, APort, DeviceHandle);
  end;
  { Если дескриптор DeviceHandle все еще равен 0, генерируется исключение.
    В противном случае используйте функцию GlobalLock() для получения
    указателя на структуру TDeviceMode. }
  if DeviceHandle = 0 then
    Raise Exception.Create('Could Not Initialize TDeviceMode structure')
  else
    DevMode := GlobalLock(DeviceHandle);
  { Использование структуры DevMode. }
  {  !!!! }
  if not DeviceHandle = 0 then
    GlobalUnlock(DeviceHandle);
end;
```

Приведенные в листинге 10.5 комментарии разъясняют действия, выполняемые для получения указателя на структуру TDeviceMode. С этим указателем могут работать самые разные подпрограммы печати, что демонстрируется в следующих разделах. Но сначала стоит обратить внимание на один комментарий:

```
{ Использование структуры DevMode }
{  !!!! }
```

Это как раз то самое место, где следует разместить текст, различные варианты которого приводятся ниже.

Прежде чем инициализировать какой-либо член структуры TDeviceMode, необходимо конкретно указать, какой член вы собираетесь инициализировать. Это делается путем установки соответствующего разряда в поле признаков dmFields. В табл. 10.3 перечислены различные битовые признаки поля dmFields с указанием члена структуры TDeviceMode, к которому они относятся.

Таблица 10.3. Битовые признаки свойства TDeviceMode.dmFields

Значение dmField	Соответствующее поле структуры
DM_ORIENTATION	dmOrientation
DM_PAPERSIZE	dmPaperSize
DM_PAPERLENGTH	dmPaperLength
DM_PAPERWIDTH	dmPaperWidth
DM_SCALE	dmScale
DM_COPIES	dmCopies
DM_DEFAULTSOURCE	dmDefaultSource
DM_PRINTQUALITY	dmPrintQuality
DM_COLOR	dmColor
DM_DUPLEX	dmDuplex
DM_YRESOLUTION	dmYResolution
DM_TTOPTION	dmTTOption
DM_COLLATE	dmCollate
DM_FORMNAME	dmFormName
DM_LOGPIXELS	dmLogPixels
DM_BITSPERPEL	dmBitsPerPel
DM_PELSWIDTH	dmPelsWidth
DM_PELSHEIGHT	dmPelsHeight
DM_DISPLAYFLAGS	dmDisplayFlags
DM_DISPLAYFREQUENCY	dmDisplayFrequency
DM_ICMMETHOD	dmICMMethod (только для Windows 95)
DM_ICMINTENT	dmICMIntent (только для Windows 95)
DM_MEDIATYPE	dmMediaType (только для Windows 95)
DM_DITHERTYPE	dmDitherType (только для Windows 95)

В приведенных ниже примерах вы узнаете, как установить нужный битовый признак и соответствующий ему член структуры TDeviceMode.

Задание количества печатных копий

Указав число копий в поле dmCopies структуры TDeviceMode, можно сообщить заданию печати о том, сколько копий документа необходимо получить. В следующем фрагменте кода показано, как это сделать:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_COPIES;
   dmCopies := Copies;
end;
```

Сначала необходимо установить соответствующий битовый признак поля dmFields, чтобы указать, какой член структуры TDeviceMode будет инициализирован. Этот программный фрагмент — как раз то, что нужно вставить в код, представленный в листинге 10.5 (вместо комментария {!!!!}). Теперь, при каждом запуске данного задания печати, на принтер будут посылаться сведения о требуемом количестве копий.

Задание ориентации листа печати принтера

Задание ориентации листа печати принтера аналогично заданию числа копий, за исключением того, что в первом случае инициализируется другой член структуры TDeviceMode:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_ORIENTATION;
   dmOrientation := DMORIENT_LANDSCAPE;
end;
```

Для инициализации члена dmOrientation возможны только две опции: DMORIENT_LANDSCAPE (альбомная ориентация) и DMORIENT_PORTRAIT (книжная ориентация). Для этой же цели можно использовать свойство Orientation объекта TPrinter.

Задание размера бумаги

Для указания размера бумаги инициализируется член dmPaperSize структуры TDeviceMode:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PAPERSIZE;
   dmPaperSize := DMPAPER_LETTER;  // Форматы Letter, 8-1/2 × 11 дюймов
end;
```

Для члена dmPaperSize существует несколько предопределенных значений, которые можно найти в интерактивной справочной системе Delphi (в разделе описания структуры TDeviceMode). Если размер бумаги определяется членами dmPaperWidth и dmPaperHeight, то dmPaperSize устанавливается равным нулю.

Задание длины бумаги

Длину бумаги в десятых долях миллиметра можно указать, поместив соответствующее значение в поле dmPaperLength. Это значение переопределит любые установки, примененные к полю dmPaperSize. В следующем фрагменте кода демонстрируется установка длины бумаги:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PAPERLENGTH;
   dmPaperLength := SomeLength;
end;
```

Задание ширины бумаги

Ширина бумаги также указывается в десятых долях миллиметра. Для установки ширины бумаги необходимо поместить требуемое значение в поле dmPaperWidth структуры TDeviceMode. Этот процесс иллюстрируется следующим фрагментом кода:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PAPERWIDTH;
   dmPaperWidth := SomeWidth;
end;
```

Данное значение также переопределяет любые установки поля dmPaperSize.

Задание масштаба печати

Масштаб печати — это коэффициент, который учитывается при масштабировании печатаемых объектов, т.е. реальный размер страницы получается путем умножения физического размера страницы на коэффициент TDeviceMode.dmScale, разделенный на число 100. Следовательно, чтобы сжать печатаемые объекты (графику или текст) вдвое (от их исходного размера), полю dmScale нужно присвоить значение 50. В следующем фрагменте кода показано, как установить масштаб печати:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_SCALE;
   dmScale := 50;
end;
```

Задание цвета печати

Для принтеров, которые поддерживают цветную печать, можно задать работу в цветном или монохромном режиме, установив требуемое значение в поле dmColor, как показано в этом примере:

```
with DevMode^ do
begin
  dmFields := dmFields or DM_COLOR;
  dmColor := DMCOLOR_COLOR;
end;
```

Кроме значения DMCOLOR_COLOR, полю dmColor можно присвоить значение DMCOLOR_MONOCHROME.

Задание качества печати

Под *качеством печати* понимается разрешение, с которым работает принтер. Для установки качества печати существует четыре предопределенных значения.

- DMRES_HIGH — печать с высоким разрешением.
- DMRES_MEDIUM — печать со средним разрешением

- DMRES_LOW — печать с низким разрешением.
- DMRES_DRAFT — режим черновой (скоростной) печати.

Для изменения качества печати поместите требуемое значение в поле dmPrintQuality структуры TDeviceMode:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PRINTQUALITY;
   dmPrintQuality := DMRES_DRAFT;
end;
```

Задание двухсторонней печати

В некоторых принтерах предусмотрена возможность печати на обеих сторонах бумаги. В этом случае можно установить для принтера нужный вам режим путем помещения в поле dmDuplex структуры TDeviceMode одного из следующих значений:

- DMDUP_SIMPLEX;
- DMDUP_HORIZONTAL;
- DMDUP_VERTICAL.

Приведем пример такой установки:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_DUPLEX;
   dmDuplex := DMDUP_HORIZONTAL;
end;
```

Изменение принтера, назначаемого по умолчанию

Несмотря на то что изменить назначаемый по умолчанию принтер можно путем активизации окна папки принтеров, необходимость такого изменения может возникнуть и во время работы приложения. Реализация этой возможности иллюстрируется в проекте, представленном в листинге 10.6.

Листинг 10.6. Изменение принтера, назначаемого по умолчанию

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
```

```
    cbPrinters: TComboBox;
    lblPrinter: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure cbPrintersChange(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation
uses IniFiles, Printers;

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
begin
  { Копируем имена принтеров в комбинированный список и настраиваем
    его на выделение принтера, назначенного по умолчанию в данный момент }
  cbPrinters.Items.Assign(Printer.Printers);
  cbPrinters.Text := Printer.Printers[Printer.PrinterIndex];
  //Обновляем надпись для отображения принтера, назначенного по умолчанию
  lblPrinter.Caption := Printer.Printers[Printer.PrinterIndex];
end;

procedure TMainForm.cbPrintersChange(Sender: TObject);
var
  IniFile: TIniFile;
  TempStr1, TempStr2: String;
begin
  with Printer do
  begin
    // Устанавливаем новый принтер на основе выбранного элемента ComboBox
    PrinterIndex := cbPrinters.ItemIndex;
    // Сохраняем имя принтера во временной строке
    TempStr1 := Printers[PrinterIndex];
    // Удаляем ненужную часть имени принтера
    System.Delete(TempStr1, Pos(' on ', TempStr1), Length(TempStr1));
    // Создаем класс TIniFile
    IniFile := TIniFile.Create('WIN.INI');
    try
      // Считываем имя устройства выделенного принтера
      TempStr2 := IniFile.ReadString('Devices', TempStr1, '');
      // Заменяем принтер по умолчанию тем, что выбрал пользователь
      IniFile.WriteString('windows', 'device', TempStr1 + ',' + TempStr2);
    finally
      IniFile.Free;
```

```
    end;
  end;
  // Обновляем надпись для отображения нового выделенного принтера
  lblPrinter.Caption := Printer.Printers[Printer.PrinterIndex];
end;

end.
```

Этот проект состоит из главной формы с компонентами `TComboBox` и `TLabel`. При создании формы компонент `TComboBox` инициализируется списком строк, содержащих имена принтеров, полученные из свойства `Printer.Printers`. Затем обновляется компонент `TLabel`, чтобы отобразить принтер, выделенный в данный момент. Текст, выполняющий модификацию назначаемого по умолчанию принтера, помещается в обработчик событий `cbPrintersChange()`. Внесенные изменения модифицируют элемент `[device]` в разделе `[windows]` файла `WIN.INI`, расположенного в каталоге Windows. Комментарии, приведенные в листинге 10.6, помогут понять процесс внесения этих модификаций.

Получение информации о принтере

В этом разделе демонстрируется программа получения информации о принтере, а также о его возможностях по выводу текста и графики.

Получить информацию об определенном принтере может потребоваться по самым разным причинам. Например, чтобы уточнить, поддерживается ли принтером та или иная функция. Типичный пример состоит в необходимости выяснить, поддерживает ли текущий принтер печать полосами. Печать полосами представляет собой процесс, позволяющий повысить скорость печати и уменьшить требования к размерам буферов печати на диске для принтеров, имеющих ограниченный объем памяти. Для активизации печати полосами необходимо обратиться к специальной функции Win32 API. Для принтеров, которые не поддерживают данной возможности, вызов этой функции выполнять нельзя. Следовательно, необходимо предварительно уточнить, поддерживает ли данный принтер режим печати полосами (и если да, то как), прежде чем предпринимать попытку его активизации.

Функции `GetDeviceCaps()` и `DeviceCapabilities()`

С помощью функции Win32 API `GetDeviceCaps()` можно получить информацию об устройствах, для которых существует контекст устройства (принтеры, плоттеры, мониторы и т.п.). При обращении к этой функции последней необходимо передать дескриптор контекста устройства и индекс, определяющий информацию, которую вы хотите получить.

Функция `DeviceCapabilities()` относится исключительно к принтерам, причем информация, получаемая с ее помощью, предоставляется драйвером указанного принтера. При обращении к этой функции ей необходимо передать строки, идентифицирующие конкретное устройство печати, и индекс, определяющий нужные данные. Иногда для получения определенных данных требуется выполнить два обращения к функции `DeviceCapabilities()`. При первом вызове определяется объем памяти, необходимый для размещения получаемых данных, а при втором эти данные сохраняются в выделенном блоке памяти. Как это сделать, показано ниже.

Следует иметь в виду, что большинство функций, не поддерживаемых тем или иным принтером, (если судить по информации об их возможностях) на поверку оказывается реализуемыми. Например, если результаты выполнения функций `GetDeviceCaps()` и `DeviceCapabilities()` указывают на то, что функции `BitBlt()`, `StretchBlt()` или шрифты TrueType не поддерживаются, это не означает, что ими совсем нельзя пользоваться. Эти функции может имитировать интерфейс GDI. Но если устройство не является растровым, GDI не сможет имитировать функцию `BitBlt()`. Например, на перьевом плоттере функция `BitBlt()` никогда не будет работать.

Пример программы получения информации о принтере

На рис. 10.8 показана главная форма такой программы. Эта форма содержит восемь вкладок, предназначенных для отображения различных возможностей принтера, выбранного в комбинированном списке.

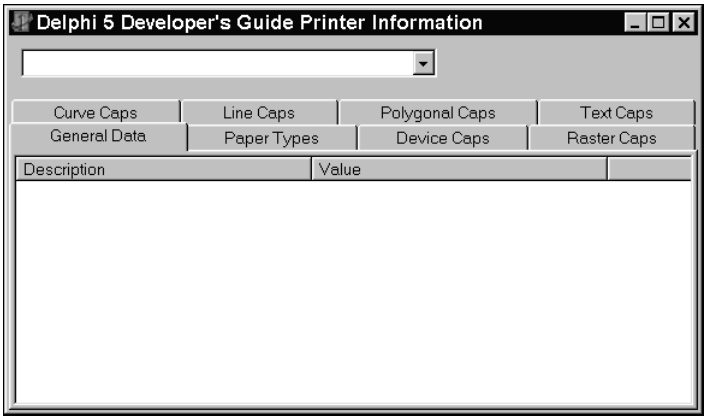


Рис. 10.8. Главная форма программы получения информации о принтере

Объявление функции `DeviceCapabilitiesA`

При попытке использовать функцию `DeviceCapabilities()`, определенную в модуле `WINDOWS.PAS`, вы не сможете выполнить свою программу, поскольку эта функция определена в библиотеке `GDI32.DLL` не так, как обозначено в модуле `WINDOWS.PAS`. Оказывается, в библиотеке `GDI32.DLL` эта функция имеет имя `DeviceCapabilitiesEx()`. Но она все равно не будет работать ожидаемым образом и вернет ошибочные результаты даже в том случае, если определить ее прототип так, как показано в следующем примере:

```
function DeviceCapabilitiesEx(pDevice, pPort: Pchar; fwCapability: Word;
         pOutput: Pchar; DevMode: PdeviceMode):
         Integer; stdcall; external 'Gdi32.dll';
```

Оказывается, что в интерфейсе спулера печати Win32 `WINSPOOL.DRV` определены две функции: `DeviceCapabilitiesA()` — для строк ANSI; `DeviceCapabilitiesW()` — для строк Unicode. Функция `DeviceCapabilitiesA()` будет корректно работать, если она использует-

ся так, как показано в Microsoft Developer's Network CD (MSDN). Вот корректное определение прототипа функции, которая используется в примере программы, приведенном в листинге 10.7:

```
function DeviceCapabilitiesA(pDevice, pPort: Pchar; fwCapability: Word;
         pOutput: Pchar; DevMode: PdeviceMode):
         Integer; stdcall; external 'winspool.drv';
```

Это объявление функции взято из модуля WINSPOOL.PAS в Delphi 5.

Описание работы программы

В листинге 10.7 содержится исходный текст программы получения информации о принтере. Обработчик событий OnCreate главной формы просто заполняет комбинированный список именами принтеров, доступных в системе. Обработчик событий OnChange этого комбинированного списка является центральным звеном приложения, откуда вызываются методы для получения информации о принтере.

Первая вкладка формы General Data содержит общую информацию об устройстве печати. Вы увидите, что при вызове метода TPrinter.GetPrinter() считываются такие данные, как имя принтера, драйвер и порт, а также дескриптор структуры TDeviceMode для выделенного в данный момент принтера. Все эти данные помещаются во вкладку General Data. Для получения версии драйвера принтера нужно использовать функцию DeviceCapabilitiesA() и передать ей индекс DC_DRIVER. В остальной части обработчика событий PrinterComboChange вызываются различные методы для заполнения элементов списков, которые отображаются на других вкладках главной формы.

С помощью метода GetBinNames() иллюстрируется использование функции DeviceCapabilitiesA() для получения имен для выбранного принтера. При первом вызове функции DeviceCapabilitiesA() с передачей ей индекса DC_BINNAMES и значения nil в качестве параметров pOutput и DevMode считывается количество этих имен, т.е. по результату этой функции можно судить об объеме памяти, который необходимо выделить для хранения считываемых имен. В соответствии с документацией к функции DeviceCapabilitiesA(), каждое такое имя определяется как массив из 24 символов. Мы определили тип данных TBinName следующим образом:

```
TBinName = array[0..23] of char;
```

Кроме того, мы определили массив типа TBinName:

```
TBinNames = array[0..0] of TBinName;
```

Этот тип используется для операции приведения типа, а именно — приведения указателя к типу массива TBinNames. При получении доступа к некоторому элементу массива по заданному индексу нужно запретить проверку диапазона, поскольку этот массив определен с диапазоном 0..0, как показано в методе GetBinNames(). После соблюдения всех предосторожностей имена записываются в соответствующий список для отображения.

Тот же способ определения объема требуемой памяти и динамического выделения этой памяти используется в методах GetDevCapsPaperNames() и GetResolutions().

В методах GetDuplexSupport(), GetCopies() и GetEMFStatus() запрашиваемая информация предоставляется непосредственно в виде значения, возвращаемого функцией DeviceCapabilitiesA(). Например, для определения того, поддерживает ли выбранный

принтер двухстороннюю печать, используется следующий код (если возвращаемое значение равно 1, двухсторонняя печать поддерживается, если 0 — нет):

```
DeviceCapabilitiesA(Device, Port, DC_DUPLEX, nil, nil);
```

А следующая инструкция возвращает максимальное количество копий, которое может напечатать данное устройство:

```
DeviceCapabilitiesA(Device, Port, DC_COPIES, nil, nil);
```

В остальных методах используется функция `GetDeviceCaps()` для определения различных возможностей выбранного устройства. В некоторых случаях функция `GetDeviceCaps()` возвращает запрашиваемое значение. Например, следующая инструкция возвращает ширину поля печати принтера в миллиметрах:

```
GetDeviceCaps(Printer.Handle, HORZSIZE);
```

В других случаях функция `GetDeviceCaps()` возвращает целое число, двоичные разряды которого маскируются для определения конкретной возможности. Например, сначала метод `GetDeviceCaps()` считывает целое значение, содержащее поля, на которые наложена маска:

```
RCaps := GetDeviceCaps(Printer.Handle, RASTERCAPS);
```

Затем, чтобы определить, поддерживает ли данный принтер возможность печати полосами (banding), нужно выделить поле `RC_BANDING` путем выполнения операции `AND`:

```
(RCaps and RC_BANDING) = RC_BANDING
```

Эта оценочная формула передается одной из вспомогательных функций `BoolToYesNoStr()`, которая на основании результата вычисления этой формулы возвращает строку `Yes` или `No`. Аналогичным образом происходит выделение других полей. Тот же способ оценки битовых полей, возвращаемых функциями `GetDeviceCaps()` и `DeviceCapabilitiesA()`, используется и в других методах, например, в таких случаях `GetTrueTypeInfo()`.

Обе функции, как `GetDeviceCaps()`, так и `DeviceCapabilitiesA()`, достаточно хорошо описаны в интерактивной справочной системе Win32 API.

Листинг 10.7. Пример программы получения информации о принтере

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    pgcPrinterInfo: TPageControl;
    tbsPaperTypes: TTabSheet;
    tbsGeneralData: TTabSheet;
```

```
    lbPaperTypes: TListBox;
    tbsDeviceCaps: TTabSheet;
    tbsRasterCaps: TTabSheet;
    tbsCurveCaps: TTabSheet;
    tbsLineCaps: TTabSheet;
    tbsPolygonalCaps: TTabSheet;
    tbsTextCaps: TTabSheet;
    lvGeneralData: TListView;
    lvCurveCaps: TListView;
    Splitter1: TSplitter;
    lvDeviceCaps: TListView;
    lvRasterCaps: TListView;
    pnlTop: TPanel;
    cbPrinters: TComboBox;
    lvLineCaps: TListView;
    lvPolyCaps: TListView;
    lvTextCaps: TListView;
    procedure FormCreate(Sender: TObject);
    procedure cbPrintersChange(Sender: TObject);
  private
    Device, Driver, Port: array[0..255] of char;
    ADevMode: THandle;
   public
    procedure GetBinNames;
    procedure GetDuplexSupport;
    procedure GetCopies;
    procedure GetEMFStatus;
    procedure GetResolutions;
    procedure GetTrueTypeInfo;
    procedure GetDevCapsPaperNames;
    procedure GetDevCaps;
    procedure GetRasterCaps;
    procedure GetCurveCaps;
    procedure GetLineCaps;
    procedure GetPolyCaps;
    procedure GetTextCaps;
  end;

var
  MainForm: TMainForm;

implementation
uses
  Printers, WinSpool;

const
  NoYesArray: array[Boolean] of String = ('No', 'Yes');
type

  // Тип для хранения названий
```

```
  TBinName = array[0..23] of char;
  // Тип, использующий для предотвращения ошибки установку $R-
  TBinNames = array[0..0] of TBinName;

  // Типы для хранения названий бумаги
  TPName = array[0..63] of char;

  // Тип, использующий для предотвращения ошибки установку $R-
  TPNames = array[0..0] of TPName;

  // Типы для хранения возможных разрешений
  TResolution = array[0..1] of integer;
  // Тип, использующий для предотвращения ошибки установку $R-
  TResolutions = array[0..0] of TResolution;

  // Тип для хранения массива размеров страниц
  TPageSizeArray = Array[0..0] of word;

var
 Rslt: Integer;

{$R *.DFM}
(*
function BoolToYesNoStr(aVal: Boolean): String;
// Возвращает строку "YES" или "NO" на основе булевого значения
begin
  if aVal then
    Result := 'Yes'
  else
    Result := 'No';
end;
*)
procedure AddListViewItem(const aCaption, aValue: String; aLV: TListView);
// Этот метод добавляет элемент TListItem в список aLV типа TListView
var
  NewItem: TListItem;
begin
  NewItem := aLV.Items.Add;
  NewItem.Caption := aCaption;
  NewItem.SubItems.Add(aValue);
end;

procedure TMainForm.GetBinNames;
var
  BinNames: Pointer;
  i: integer;
begin
{$R-}  // Отключение проверки диапазона
  // Сначала определяем количество доступных имен.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_BINNAMES, nil, nil);
```

```
  if Rslt > 0 then
  begin
    { Каждое имя занимает 24 байта. Значит, для их хранения нужно
      выделить Rslt*24 байтов. }
    GetMem(BinNames, Rslt*24);
    try
      // Теперь считываем имена в выделенный блок памяти.
      if DeviceCapabilitiesA(Device, Port, DC_BINNAMES, BinNames, nil) = -1 then
        raise Exception.Create('DevCap Error');
      // Добавляем информацию в соответствующий список.
      AddListViewItem('BIN NAMES', EmptyStr, lvGeneralData);
      for i := 0 to Rslt - 1 do
      begin
        AddListViewItem(Format('   Bin Name %d', [i]),
          StrPas(TBinNames(BinNames^)[i]), lvGeneralData);
      end;
    finally
      FreeMem(BinNames, Rslt*24);
    end;
  end;
{$R+}  // Восстанавливаем проверку диапазона.
end;

procedure TMainForm.GetDuplexSupport;
begin
  { Эта функция использует функцию DeviceCapabilitiesA для определения того,
    поддерживает принтер двустороннюю печать или нет. }
  Rslt := DeviceCapabilitiesA(Device, Port, DC_DUPLEX, nil, nil);
  AddListViewItem('Duplex Printing', NoYesArray[Rslt = 1], lvGeneralData);
end;

procedure TMainForm.GetCopies;
begin
  { Эта функция определяет максимально возможное количество копий. }
  Rslt := DeviceCapabilitiesA(Device, Port, DC_COPIES, nil, nil);
  AddListViewItem('Copies that printer can print',
    InttoStr(Rslt), lvGeneralData);
end;

procedure TMainForm.GetEMFStatus;
begin
  // Эта функция определяет, поддерживает ли уст-во расширенные метафайлы.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_EMF_COMPLIANT, nil, nil);
  AddListViewItem('EMF Compliant', NoYesArray[Rslt=1], lvGeneralData);
end;

procedure TMainForm.GetResolutions;
var
  Resolutions: Pointer;
  i: integer;
```

```
begin
{$R-}   // Проверку диапазона нужно отключить.
  // Определяем количество доступных разрешений.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_ENUMRESOLUTIONS, nil, nil);
  if Rslt > 0 then begin
    { Выделяем память для хранения различных разрешений, которые
      представляются парами целых: 300, 300. }
    GetMem(Resolutions, (SizeOf(Integer)*2)*Rslt);
    try
      // Считываем различные разрешения.
      if DeviceCapabilitiesA(Device, Port, DC_ENUMRESOLUTIONS,
      Resolutions, nil) = -1 then
        Raise Exception.Create('DevCaps Error'); // Ошибка DevCaps
      // Добавляем разрешения в соответствующий список.
      AddListViewItem('RESOLUTION CONFIGURATIONS', EmptyStr, lvGeneralData);
      for i := 0 to Rslt - 1 do
      begin                        // Конфигурация разрешения:
        AddListViewItem('   Resolution Configuration',
          IntToStr(TResolutions(Resolutions^)[i][0])+
          ' '+IntToStr(TResolutions(Resolutions^)[i][1]), lvGeneralData);
      end;
    finally
      FreeMem(Resolutions, SizeOf(Integer)*Rslt*2);
    end;
  end;
{$R+}   // Восстанавливаем проверку диапазона.
end;

procedure TMainForm.GetTrueTypeInfo;
begin
  { Получаем информацию о возможности применения шрифтов
    TrueType в виде битовых масок. }
  Rslt := DeviceCapabilitiesA(Device, Port, DC_TRUETYPE, nil, nil);
  if Rslt <> 0 then
    { Теперь выделяем отдельные возможности применения шрифтов
      TrueType и результаты помещаем в соответствующий список. }
    AddListViewItem('TRUE TYPE FONTS', EmptyStr, lvGeneralData);
    with lvGeneralData.Items do
    begin
      // Печать шрифтов TrueType как графики:
      AddListViewItem('   Prints TrueType fonts as graphics',
        NoYesArray[(Rslt and DCTT_BITMAP) = DCTT_BITMAP], lvGeneralData);
      // Загрузка шрифтов TrueType:
      AddListViewItem('   Downloads TrueType fonts',
        NoYesArray[(Rslt and DCTT_DOWNLOAD) = DCTT_DOWNLOAD],lvGeneralData);
      // Загрузка контурных шрифтов TrueType:
      AddListViewItem('   Downloads outline TrueType fonts',
        NoYesArray[(Rslt and DCTT_DOWNLOAD_OUTLINE) = DCTT_DOWNLOAD_OUTLINE],
        lvGeneralData);
     // Замена устройства для шрифтов TrueType:
```

```
      AddListViewItem('   Substitutes device for TrueType fonts',
        NoYesArray[(Rslt and DCTT_SUBDEV) = DCTT_SUBDEV], lvGeneralData);
    end;
end;

procedure TMainForm.GetDevCapsPaperNames;
{ Этот метод получает типы бумаги, доступные на выбранном принтере,
  с помощью функции DeviceCapabilitiesA. }
var
  PaperNames: Pointer;
  i: integer;
begin
{$R-}   // Отключение проверки диапазона.
  lbPaperTypes.Items.Clear;
  // Сначала получаем количество доступных названий бумаги.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_PAPERNAMES, nil, nil);
  if Rslt > 0 then begin
    { Теперь выделяем память под массив названий бумаги. Каждое название
      занимает 64 байта. Следовательно, выделяем Rslt*64 байт памяти. }
    GetMem(PaperNames, Rslt*64);
    try
      // Считываем список названий в выделенный блок памяти.
      if DeviceCapabilitiesA(Device, Port, DC_PAPERNAMES,
      PaperNames, nil) = - 1 then
        raise Exception.Create('DevCap Error'); // Ошибка DevCap
      // Добавляем названия бумаги в соответствующий список.
      for i := 0 to Rslt - 1 do
    lbPaperTypes.Items.Add(StrPas(TPNames(PaperNames^)[i]));
    finally
      FreeMem(PaperNames, Rslt*64);
    end;
  end;
{$R+}  // Восстановление проверки диапазона.
end;

procedure TMainForm.GetDevCaps;
{ Этот метод считывает различные характеристики выбранного принтера
  с помощью функции GetDeviceCaps. За описанием значения каждого
  элемента обратитесь к интерактивной справочной системе Win32 API. }
begin
  with lvDeviceCaps.Items do
  begin
    Clear;
    // Ширина в миллиметрах:
    AddListViewItem('Width in millimeters',
      IntToStr(GetDeviceCaps(Printer.Handle, HORZSIZE)), lvDeviceCaps);
    // Высота в миллиметрах:
    AddListViewItem('Height in millimeter',
      IntToStr(GetDeviceCaps(Printer.Handle, VERTSIZE)), lvDeviceCaps);
    // Ширина в пикселях:
```

```
    AddListViewItem('Width in pixels',
      IntToStr(GetDeviceCaps(Printer.Handle, HORZRES)), lvDeviceCaps);
    // Высота в пикселях:
    AddListViewItem('Height in pixels',
      IntToStr(GetDeviceCaps(Printer.Handle, VERTRES)), lvDeviceCaps);
    // Пиксели на дюйм по горизонтали:
    AddListViewItem('Pixels per horizontal inch',
      IntToStr(GetDeviceCaps(Printer.Handle, LOGPIXELSX)), lvDeviceCaps);
    // Пиксели на дюйм по вертикали:
    AddListViewItem('Pixels per vertical inch',
      IntToStr(GetDeviceCaps(Printer.Handle, LOGPIXELSY)), lvDeviceCaps);
    // Цветовые биты на пиксель:
    AddListViewItem('Color bits per pixel',
      IntToStr(GetDeviceCaps(Printer.Handle, BITSPIXEL)), lvDeviceCaps);
    // Количество цветовых плоскостей:
    AddListViewItem('Number of color planes',
      IntToStr(GetDeviceCaps(Printer.Handle, PLANES)), lvDeviceCaps);
    // Количество кистей:
    AddListViewItem('Number of brushes',
      IntToStr(GetDeviceCaps(Printer.Handle, NUMBRUSHES)), lvDeviceCaps);
    // Количество перьев:
    AddListViewItem('Number of pens',
      IntToStr(GetDeviceCaps(Printer.Handle, NUMPENS)), lvDeviceCaps);
    // Количество шрифтов:
    AddListViewItem('Number of fonts',
      IntToStr(GetDeviceCaps(Printer.Handle, NUMFONTS)), lvDeviceCaps);
    Rslt := GetDeviceCaps(Printer.Handle, NUMCOLORS);
    if Rslt = -1 then
      // Количество составляющих цветовой таблицы:
      AddListViewItem('Number of entries in color table', ' > 8', lvDeviceCaps)
    else AddListViewItem('Number of entries in color table',
        IntToStr(Rslt), lvDeviceCaps);
    // Относительный пиксель ширины:
    AddListViewItem('Relative pixel drawing width',
      IntToStr(GetDeviceCaps(Printer.Handle, ASPECTX)), lvDeviceCaps);
    // Относительный пиксель высоты:
    AddListViewItem('Relative pixel drawing height',
      IntToStr(GetDeviceCaps(Printer.Handle, ASPECTY)), lvDeviceCaps);
    // Диагональный пиксель ширины:
    AddListViewItem('Diagonal pixel drawing width',
      IntToStr(GetDeviceCaps(Printer.Handle, ASPECTXY)), lvDeviceCaps);
    if GetDeviceCaps(Printer.Handle, CLIPCAPS) = 1 then
      // Обрезка до прямоугольника:
      AddListViewItem('Clip to rectangle', 'Yes', lvDeviceCaps)
    else AddListViewItem('Clip to rectangle', 'No', lvDeviceCaps);
  end;
end;

procedure TMainForm.GetRasterCaps;
{ Этот метод получает различные растровые характеристики выбранного
```

```
принтера с помощью функции GetDeviceCaps и передачи ей индекса RASTERCAPS.
Описание каждой характеристики имеется в интерактивной справочной системе. }
var
  RCaps: Integer;
begin
  with lvRasterCaps.Items do
  begin
    Clear;
    RCaps := GetDeviceCaps(Printer.Handle, RASTERCAPS);
    AddListViewItem('Banding',
      NoYesArray[(RCaps and RC_BANDING) = RC_BANDING], lvRasterCaps);
    AddListViewItem('BitBlt Capable',
      NoYesArray[(RCaps and RC_BITBLT) = RC_BITBLT], lvRasterCaps);
    AddListViewItem('Supports bitmaps > 64K',
      NoYesArray[(RCaps and RC_BITMAP64) = RC_BITMAP64], lvRasterCaps);
    AddListViewItem('DIB support',
      NoYesArray[(RCaps and RC_DI_BITMAP) = RC_DI_BITMAP], lvRasterCaps);
    AddListViewItem('Floodfill support',
      NoYesArray[(RCaps and RC_FLOODFILL) = RC_FLOODFILL], lvRasterCaps);
    AddListViewItem('Windows 2.0 support',
      NoYesArray[(RCaps and RC_GDI20_OUTPUT) = RC_GDI20_OUTPUT], lvRasterCaps);
    AddListViewItem('Palette based device',
      NoYesArray[(RCaps and RC_PALETTE) = RC_PALETTE], lvRasterCaps);
    AddListViewItem('Scaling support',
      NoYesArray[(RCaps and RC_SCALING) = RC_SCALING], lvRasterCaps);
    AddListViewItem('StretchBlt support',
      NoYesArray[(RCaps and RC_STRETCHBLT) = RC_STRETCHBLT], lvRasterCaps);
    AddListViewItem('StretchDIBits support',
      NoYesArray[(RCaps and RC_STRETCHDIB) = RC_STRETCHDIB], lvRasterCaps);
  end;
end;

procedure TMainForm.GetCurveCaps;
{ Этот метод получает для выбранного принтера различные характеристики
  по выводу на печать кривых с помощью функции GetDeviceCaps и передачи
  ей индекса CURVECAPS. За описанием каждой характеристики обратитесь
  к интерактивной справочной системе. }
var
  CCaps: Integer;
begin
  with lvCurveCaps.Items do
  begin
    Clear;
    CCaps := GetDeviceCaps(Printer.Handle, CURVECAPS);
    // Поддержка кривых:
    AddListViewItem('Curve support',
      NoYesArray[(CCaps and CC_NONE) = CC_NONE], lvCurveCaps);
    // Поддержка окружностей:
    AddListViewItem('Circle support',
      NoYesArray[(CCaps and CC_CIRCLES) = CC_CIRCLES], lvCurveCaps);
```

```
    // Поддержка секторов:
    AddListViewItem('Pie support',
      NoYesArray[(CCaps and CC_PIE) = CC_PIE], lvCurveCaps);
    // Поддержка хорды дуги:
    AddListViewItem('Chord arc support',
      NoYesArray[(CCaps and CC_CHORD) = CC_CHORD], lvCurveCaps);
    // Поддержка эллипсов:
    AddListViewItem('Ellipse support',
      NoYesArray[(CCaps and CC_ELLIPSES) = CC_ELLIPSES], lvCurveCaps);
    // Поддержка утолщенных границ:
    AddListViewItem('Wide border support',
      NoYesArray[(CCaps and CC_WIDE) = CC_WIDE], lvCurveCaps);
    // Поддержка стилизованных границ:
    AddListViewItem('Styled border support',
      NoYesArray[(CCaps and CC_STYLED) = CC_STYLED], lvCurveCaps);
    // Поддержка закругленных прямоугольников:
    AddListViewItem('Round rectangle support',
      NoYesArray[(CCaps and CC_ROUNDRECT) = CC_ROUNDRECT], lvCurveCaps);

  end;
end;

procedure TMainForm.GetLineCaps;
{ Этот метод получает для выбранного принтера различные характеристики
  по выводу прямых с помощью функции GetDeviceCaps и передачи ей индекса
  LINECAPS. За описанием каждой характеристики обратитесь к
  интерактивной справочной системе. }
var
  LCaps: Integer;
begin
  with lvLineCaps.Items do
  begin
    Clear;
    LCaps := GetDeviceCaps(Printer.Handle, LINECAPS);
    // Поддержка прямой:
    AddListViewItem('Line support',
      NoYesArray[(LCaps and LC_NONE) = LC_NONE], lvLineCaps);
    // Поддержка ломаной:
    AddListViewItem('Polyline support',
      NoYesArray[(LCaps and LC_POLYLINE) = LC_POLYLINE], lvLineCaps);
    // Поддержка маркера:
    AddListViewItem('Marker support',
      NoYesArray[(LCaps and LC_MARKER) = LC_MARKER], lvLineCaps);
    // Поддержка нескольких маркеров:
    AddListViewItem('Multiple marker support',
      NoYesArray[(LCaps and LC_POLYMARKER) = LC_POLYMARKER], lvLineCaps);
    // Поддержка широких линий:
    AddListViewItem('Wide line support',
      NoYesArray[(LCaps and LC_WIDE) = LC_WIDE], lvLineCaps);
    // Поддержка стилизованных линий:
```

```
    AddListViewItem('Styled line support',
      NoYesArray[(LCaps and LC_STYLED) = LC_STYLED], lvLineCaps);
    // Поддержка широких и стилизованных линий:
    AddListViewItem('Wide and styled line support',
      NoYesArray[(LCaps and LC_WIDESTYLED) = LC_WIDESTYLED], lvLineCaps);
    // Поддержка интерьера:
    AddListViewItem('Interior support',
      NoYesArray[(LCaps and LC_INTERIORS) = LC_INTERIORS], lvLineCaps);
  end;
end;

procedure TMainForm.GetPolyCaps;
{ Этот метод получает для выбранного принтера различные характеристики
  по выводу многоугольников с помощью функции GetDeviceCaps и передачи
  ей индекса POLYGONALCAPS. За описанием каждой характеристики
  обратитесь к интерактивной справочной системе. }
var
  PCaps: Integer;
begin
  with lvPolyCaps.Items do
  begin
    Clear;
    PCaps := GetDeviceCaps(Printer.Handle, POLYGONALCAPS);
    // Поддержка вывода многоугольников:
    AddListViewItem('Polygon support',
      NoYesArray[(PCaps and PC_NONE) = PC_NONE], lvPolyCaps);
    // Поддержка альтернативной заливки многоугольников:
    AddListViewItem('Alternate fill polygon support',
      NoYesArray[(PCaps and PC_POLYGON) = PC_POLYGON], lvPolyCaps);
    // Поддержка прямоугольников:
    AddListViewItem('Rectangle support',
      NoYesArray[(PCaps and PC_RECTANGLE) = PC_RECTANGLE], lvPolyCaps);
    // Поддержка Winding-заливки многоугольников:
    AddListViewItem('Winding-fill polygon support',
      NoYesArray[(PCaps and PC_WINDPOLYGON) = PC_WINDPOLYGON], lvPolyCaps);
    // Поддержка одного прохода:
    AddListViewItem('Single scanline support',
      NoYesArray[(PCaps and PC_SCANLINE) = PC_SCANLINE], lvPolyCaps);
    // Поддержка широких границ:
    AddListViewItem('Wide border support',
      NoYesArray[(PCaps and PC_WIDE) = PC_WIDE], lvPolyCaps);
    // Поддержка стилизованных границ:
    AddListViewItem('Styled border support',
      NoYesArray[(PCaps and PC_STYLED) = PC_STYLED], lvPolyCaps);
    // Поддержка широких и стилизованных границ:
    AddListViewItem('Wide and styled border support',
      NoYesArray[(PCaps and PC_WIDESTYLED) = PC_WIDESTYLED], lvPolyCaps);
    // Поддержка интерьера:
    AddListViewItem('Interior support',
      NoYesArray[(PCaps and PC_INTERIORS) = PC_INTERIORS], lvPolyCaps);
```

```
  end;
end;

procedure TMainForm.GetTextCaps;
{ Этот метод получает для выбранного принтера различные характеристики
  по выводу текста с помощью функции GetDeviceCaps и передачи ей индекса
  TEXTCAPS. За описанием каждой характеристики обратитесь к
  интерактивной справочной системе. }
var
  TCaps: Integer;
begin
  with lvTextCaps.Items do
  begin
    Clear;
    TCaps := GetDeviceCaps(Printer.Handle, TEXTCAPS);
    // Точность вывода символа:
    AddListViewItem('Character output precision',
      NoYesArray[(TCaps and TC_OP_CHARACTER) = TC_OP_CHARACTER], lvTextCaps);
    // Точность вывода штриха:
    AddListViewItem('Stroke output precision',
      NoYesArray[(TCaps and TC_OP_STROKE) = TC_OP_STROKE], lvTextCaps);
    // Точность обрезки штриха:
    AddListViewItem('Stroke clip precision',
      NoYesArray[(TCaps and TC_CP_STROKE) = TC_CP_STROKE], lvTextCaps);
    // Поворот символа на 90°:
    AddListViewItem('90 degree character rotation',
      NoYesArray[(TCaps and TC_CR_90) = TC_CR_90], lvTextCaps);
    // Поворот символа на любой угол:
    AddListViewItem('Any degree character rotation',
      NoYesArray[(TCaps and TC_CR_ANY) = TC_CR_ANY], lvTextCaps);
    // Независимое масштабирование по осям X и Y:
    AddListViewItem('Independent scale in X and Y direction',
      NoYesArray[(TCaps and TC_SF_X_YINDEP) = TC_SF_X_YINDEP], lvTextCaps);
    // Удвоение символа для масштабирования:
    AddListViewItem('Doubled character for scaling',
      NoYesArray[(TCaps and TC_SA_DOUBLE) = TC_SA_DOUBLE], lvTextCaps);
    // Целые кратные только для масштабирования символов:
    AddListViewItem('Integer multiples only for character scaling',
      NoYesArray[(TCaps and TC_SA_INTEGER) = TC_SA_INTEGER], lvTextCaps);
    // Произвольные кратные для точного масштабирования символов:
    AddListViewItem('Any multiples for exact character scaling',
      NoYesArray[(TCaps and TC_SA_CONTIN) = TC_SA_CONTIN], lvTextCaps);
    // Удвоение насыщенности символов:
    AddListViewItem('Double weight characters',
      NoYesArray[(TCaps and TC_EA_DOUBLE) = TC_EA_DOUBLE], lvTextCaps);
    // Курсив символов:
    AddListViewItem('Italicized characters',
      NoYesArray[(TCaps and TC_IA_ABLE) = TC_IA_ABLE], lvTextCaps);
    // Подчеркивание символов:
    AddListViewItem('Underlined characters',
```

```
      NoYesArray[(TCaps and TC_UA_ABLE) = TC_UA_ABLE], lvTextCaps);
    // Перечеркивание символов:
    AddListViewItem('Strikeout characters',
      NoYesArray[(TCaps and TC_SO_ABLE) = TC_SO_ABLE], lvTextCaps);
    // Растровые шрифты:
    AddListViewItem('Raster fonts',
      NoYesArray[(TCaps and TC_RA_ABLE) = TC_RA_ABLE], lvTextCaps);
    // Векторные шрифты:
    AddListViewItem('Vector fonts',
      NoYesArray[(TCaps and TC_VA_ABLE) = TC_VA_ABLE], lvTextCaps);
    // Прокрутка с помощью бит-блока:
    AddListViewItem('Scrolling using bit-block transfer',
      NoYesArray[(TCaps and TC_SCROLLBLT) = TC_SCROLLBLT], lvTextCaps);
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Сохраняем имена принтеров в комбинированном списке.
  cbPrinters.Items.Assign(Printer.Printers);
  // Отображаем принтер, выбранный по умолчанию.
  cbPrinters.ItemIndex := Printer.PrinterIndex;
  // Генерируем событие OnChange для комбинированного списка.
  cbPrintersChange(nil);
end;

procedure TMainForm.cbPrintersChange(Sender: TObject);
begin
Screen.Cursor := crHourGlass;
  try
    // Заполняем комбинированный список доступными принтерами.
    Printer.PrinterIndex := cbPrinters.ItemIndex;
    with Printer do
      GetPrinter(Device, Driver, Port, ADevMode);
    // Заполняем вкладку информацией о принтере.
    with lvGeneralData.Items do
    begin
      Clear;
      AddListViewItem('Port', Port, lvGeneralData); // Порт
      AddListViewItem('Device', Device, lvGeneralData); // Устройство
      //Версия драйвера:
      Rslt := DeviceCapabilitiesA(Device, Port, DC_DRIVER, nil, nil);
      AddListViewItem('Driver Version', IntToStr(Rslt), lvGeneralData);
    end;

    // Следующие функции используют функцию GetDeviceCapabilitiesA.
    GetBinNames;
    GetDuplexSupport;
    GetCopies;
    GetEMFStatus;
```

```
    GetResolutions;
    GetTrueTypeInfo;

    // Следующие функции используют функцию GetDeviceCaps.
    GetDevCapsPaperNames;
    GetDevCaps;    // Заполнение вкладки Device Caps
    GetRasterCaps; // Заполнение вкладки Raster Caps
    GetCurveCaps;  // Заполнение вкладки Curve Caps
    GetLineCaps;   // Заполнение вкладки Line Caps
    GetPolyCaps;   // Заполнение вкладки Polygonal Caps
    GetTextCaps;   // Заполнение вкладки Text Caps
  finally
    Screen.Cursor := crDefault;
  end;
end;

end.
```

Резюме

В этой главе описаны приемы и методы, которые необходимо знать для программирования процесса вывода информации на печатающее устройство. Вы ознакомились с методологией, применяемой к любым задачам печати. Особое внимание уделялось структуре `TDeviceMode`, позволяющей получить исчерпывающую информацию о характеристиках и возможностях заданного принтера. Полученные знания пригодятся при чтении последующих глав, в которых рассматриваются еще более мощные методы программирования процесса печати.

Глава

11

Создание многопоточных приложений

В операционной системе Win32 предусмотрена возможность выполнения в приложении нескольких потоков. Именно в этом и состоит наиболее весомое преимущество Win32 перед 16-разрядной Windows. Многопоточность может стать одной из главных причин перехода к 32-разрядной версии Delphi. В этой главе вы найдете информацию, которая поможет вам достичь максимальной эффективности приложений за счет распределения выполняемой в нем работы между несколькими потоками.

Понятие о потоках

Как упоминалось в главе 3, "Win32 API", *поток* (thread) — это объект операционной системы, который представляет отдельный путь выполнения программы внутри определенного процесса. Каждое приложение Win32 имеет, по крайней мере, один поток, обычно называемый *первичным*, или *главным*, но приложения имеют право создавать дополнительные потоки, предназначенные для выполнения других задач.

С помощью потоков реализуются средства одновременного выполнения нескольких различных подпрограмм. Конечно, если компьютер оснащен только одним процессором, то о настоящей одновременности работы двух потоков говорить не приходится. Но когда для обработки каждого потока операционная система поочередно выделяет определенное время (измеряемое в мельчайших долях секунды), создается впечатление одновременной работы нескольких приложений.

Потоки никогда не поддерживались в среде 16-разрядной Windows. Это значит, что ни одна 32-разрядная программа Delphi, написанная с использованием потоков, никогда не будет совместимой с Delphi 1. Этот момент нужно обязательно учитывать при разработке приложений, предназначенных для работы на обеих платформах.

Новый тип многозадачности

Понятие потока во многом отличается от многозадачности, поддерживаемой в среде 16-разрядной Windows. Возможно, вам приходилось слышать, что Win32 называют операционной системой с *вытесняющей многозадачностью*, а Windows 3.1 — средой с *кооперативной многозадачностью*.

В чем же разница? В среде вытесняющей многозадачности управление возложено на операционную систему — именно она отвечает за то, какой поток должен выполняться в данный момент времени. Когда первый поток останавливается системой ради передачи второму потоку нескольких циклов центрального процессора, то о первом потоке говорят, что он *вытеснен*. И если при этом окажется, что первый поток выполняет бесконечный цикл, то, как правило, это не приводит к трагической ситуации, поскольку операционная система будет продолжать выделение процессорного времени для всех других потоков.

В среде Windows 3.1 ответственность за передачу управления операционной системе во время выполнения приложения лежит целиком на разработчике этого приложения. И когда приложению не удается с этим справиться, кажется, что операционная система "зависает" — все мы хорошо знакомы с этой печальной ситуацией. Если вдуматься, то это может показаться даже забавным — устойчивость работы всей системы 16-разрядной Windows полностью зависит от поведения *всех* ее приложений, которые должны самостоятельно защитить себя от попадания в бесконечные циклы, замкнутую цепь рекурсии и другие неприятные ситуации. А поскольку все приложения для достижения корректной работы системы должны работать согласованно, этот тип многозадачности называется *кооперативным*.

Использование многопоточности в приложениях Delphi

Не секрет, что потоки — серьезное подспорье для программистов Windows. Вторичные потоки в приложениях можно создавать везде, где требуется выполнение некоторых фоновых работ. Типичным примерами таких действий является вычисление значений отдельных ячеек электронных таблиц или помещение в спул документа текстового процессора при выводе его на печать. И тогда задача разработчика — организовать необходимую обработку фоновых процессов, сохранив при этом приемлемое время реакции пользовательского интерфейса приложения.

Бо́льшая часть компонентов библиотеки VCL построена в предположении, что доступ к ним в каждый конкретный момент времени будет выполняться только из одного потока. Хотя это замечание особенно справедливо в отношении компонентов библиотеки VCL, используемых для создания интерфейса пользователя, тем не менее важно понимать, что оно справедливо и для большинства других типов компонентов.

Невизуальные компоненты библиотеки VCL

Существует всего несколько областей, в которых подпрограммы библиотеки VCL гарантированно поддерживают многопоточность. Возможно, самым заметным в этом отношении является механизм поддержки свойств поточного ввода/вывода компонентов VCL. Особенности его реализации дают полную уверенность, что потоки данных в компонентах могут эффективно считываться и записываться сразу несколькими процессами. Помните, что даже важнейшие базовые классы в библиотеке VCL (например, класс `TList`) спроектированы без поддержки доступа к ним одновременно из нескольких потоков. В некоторых случаях подпрограммы библиотеки VCL предоставляют "потокобезопасное" альтернативное решение, к которому можно обратиться в случае необходимости. Например, если манипулировать некоторым списком должны одновременно несколько потоков, то для его реализации следует использовать класс `TThreadList`, а не обычный класс `TList`.

Визуальные компоненты библиотеки VCL

Подпрограммы библиотеки VCL требуют, чтобы все управление пользовательским интерфейсом происходило в контексте основного потока приложения (исключение составляет "потокобезопасный" класс `TCanvas`, о котором речь пойдет ниже в этой главе). Конечно, существуют решения, позволяющие выполнять обновление пользовательского интерфейса со стороны вторичных потоков (подробнее это будет описано далее). Однако данное ограничение заставляет разработчиков приложений использовать потоки более осторожно. В примерах, приведенных в этой главе, демонстрируются некоторые идеальные варианты использования многопоточности в приложениях Delphi.

Неправильное использование потоков

В любом деле важно не переборщить. Это утверждение справедливо и в отношении потоков. Несмотря на то что потоки оказывают неоценимую помощь в решении одних проблем, они могут принести с собой целый букет новых проблем. Предположим, например, что вы создаете интегрированную среду разработки и хотите, чтобы компилятор работал в своем собственном потоке, а программист мог продолжать работу с приложением во время компиляции программы. Здесь могут возникнуть проблемы вот какого плана. Что будет, если внести изменения в файл,

который компилируется в данный момент? Можно предложить несколько вариантов решения этой проблемы. Например, можно создать временную копию файла в момент его компиляции или сделать недоступными для редактирования файлы, компиляция которых еще не завершена. Все дело в том, что потоки не являются панацеей. Решая те или иные проблемы разработки, они неминуемо порождают новые, чреватые появлением ошибок, которые очень трудно устранить при отладке. Разработка и реализация кода, обеспечивающего устойчивую работу потоков, — задача не из легких, поскольку в этом случае приходится учитывать гораздо больше факторов, чем при разработке однопоточного приложения.

Объект TThread

Delphi инкапсулирует объект потока Win32 API в объекте Object Pascal, именуемом `TThread`. Хотя класс `TThread` инкапсулирует практически все важнейшие потоковые функции API в одном единственном объекте, возможны ситуации, когда вам придется непосредственно обращаться к функциям API. Чаще всего это будет связано с необходимостью синхронизации потоков. В этом разделе мы обсудим, как работает объект `TThread` и как его использовать в создаваемых приложениях.

Принципы работы объекта `TThread`

Определение объекта `TThread` находится в модуле `Classes` и имеет следующий вид:

```
type
  TThread = class
  private
    FHandle: THandle;
    FThreadID: THandle;
    FTerminated: Boolean;
    FSuspended: Boolean;
    FFreeOnTerminate: Boolean;
    FFinished: Boolean;
    FReturnValue: Integer;
    FOnTerminate: TNotifyEvent;
    FMethod: TThreadMethod;
    FSynchronizeException: TObject;
    procedure CallOnTerminate;
    function GetPriority: TThreadPriority;
    procedure SetPriority(Value: TThreadPriority);
    procedure SetSuspended(Value: Boolean);
  protected
    procedure DoTerminate; virtual;
    procedure Execute; virtual; abstract;
    procedure Synchronize(Method: TThreadMethod);
    property ReturnValue: Integer read FReturnValue write FReturnValue;
    property Terminated: Boolean read FTerminated;
  public
    constructor Create(CreateSuspended: Boolean);
    destructor Destroy; override;
    procedure Resume;
```

```
    procedure Suspend;
    procedure Terminate;
    function WaitFor: Integer;
    property FreeOnTerminate: Boolean read FFreeOnTerminate
      write FFreeOnTerminate;
    property Handle: THandle read FHandle;
    property Priority: TThreadPriority read GetPriority write SetPriority;
    property Suspended: Boolean read FSuspended write SetSuspended;
    property ThreadID: THandle read FThreadID;
    property OnTerminate: TNotifyEvent read FOnTerminate write FOnTerminate;
  end;
```

Как видно из объявления, объект `Tthread` — это прямой потомок объекта `Tobject`, следовательно, он не является компонентом. Нетрудно заметить, что метод `TThread.Execute()` абстрактный. Это значит, что класс `TThread` сам является абстрактным, и вы никогда не сможете создать экземпляр самого класса `TThread`. Вы можете создавать только экземпляры потомков класса `TThread`. Самый простой способ создания потомка класса `TThread` состоит в выборе элемента Thread Object в диалоговом окне New Items (рис. 11.1), которое открывается по команде File⇨New.

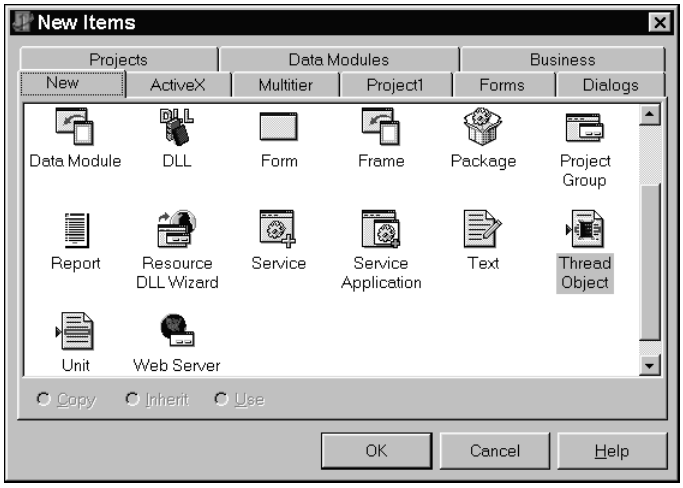


Рис. 11.1. Элемент Thread Object диалогового окна New Items

После выбора элемента Thread Object в диалоговом окне New Items откроется диалоговое окно New Thread Object, в котором вам будет предложено ввести имя для нового объекта. Например, можно ввести имя `TTestThread`. Затем Delphi создаст новый модуль, содержащий ваш объект. Вот как будет выглядеть его объявление:

```
type
  TTestThread = class(TThread)
  private
    { Закрытые объявления }
  protected
    procedure Execute; override;
  end;
```

Как видите, для создания функционального потомка класса TThread вы *должны* переопределить единственный метод — Execute(). Предположим теперь, что внутри класса TTestThread требуется выполнить сложные вычисления. В этом случае метод Execute() можно было бы определить следующим образом:

```
procedure TTestThread.Execute;
var
  i: integer;
begin
  for i := 1 to 2000000 do
    inc(Answer, Round(Abs(Sin(Sqrt(i)))));
end;
```

Единственная цель этих вычислений — затратить как можно больше времени на их выполнение.

Теперь можно выполнить этот пример потока, вызвав конструктор Create(). В данной ситуации это реализуется с помощью щелчка на кнопке главной формы, как показано в следующем фрагменте программы (во избежание ошибок компилятора не забудьте включить модуль, содержащий объект TTestThread, в инструкцию uses модуля TForm1):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewThread: TTestThread;
begin
  NewThread := TTestThread.Create(False);
end;
```

Если запустить приложение и щелкнуть на вышеуказанной кнопке, то можно убедиться, что по-прежнему существует возможность работать с формой, т.е. перемещать ее или изменять ее размеры, на фоне выполнения упомянутых вычислений.

На заметку

Единственный логический параметр, который передается в конструктор Create() класса TThread, называется CreateSuspended. Он показывает, следует ли начинать работу потока с перевода его в приостановленное состояние. Если этот параметр равен False, метод Execute() создаваемого объекта будет вызван автоматически и без промедления. Если этот параметр равен True, то для действительного запуска потока в работу, в определенной точке приложения потребуется вызвать его метод Resume(). Это приведет к выполнению метода Execute() потока и активизирует его. Обычно параметр CreateSuspended устанавливается равным True, если перед запуском потока требуется установить дополнительные свойства его объекта. Установка свойств объекта после запуска потока может привести к возникновению проблем.

При более внимательном изучении работы конструктора Create() оказывается, что он вызывает функцию библиотеки RTL Delphi BeginThread(), которая, в свою очередь, вызывает функцию Win32 API CreateThread() для создания нового потока. Значение параметра CreateSuspended показывает, нужно ли передавать функции CreateThread() признак CREATE_SUSPENDED.

Экземпляры объекта потока

Теперь вернемся к методу Execute() объекта TTestThread. Обратите внимание: он содержит локальную переменную i. Давайте рассмотрим, что произойдет с переменной i, если создать два экземпляра объекта TTestThread. Может ли значение этой переменной для одно-

го потока перезаписать значение одноименной переменной для другого? Имеет ли первый поток преимущество перед вторым? Происходит ли в этом случае разрушение потока? Ответы на все три вопроса одинаковы: нет, нет и нет. Система Win32 поддерживает для каждого работающего в системе потока отдельный стек. Это значит, что при создании нескольких экземпляров объекта `TTestThread` каждый из них будет иметь собственную копию переменной `i` в своем собственном стеке. Следовательно, в этом отношении все потоки будут действовать независимо друг от друга.

Однако необходимо заметить, что понятие "одной и той же" переменной (которая в своем потоке действует независимо от "коллег") отнюдь не переносится на глобальные переменные. Подробнее эта тема рассматривается в разделах "Хранение локальных данных потоков" и "Синхронизация потоков", ниже в этой главе.

Завершение работы потока

Работа объекта потока `TThread` считается законченной, когда завершается выполнение его метода `Execute()`. В этот момент вызывается стандартная процедура Delphi `EndThread()`, которая, в свою очередь, вызывает функцию Win32 API `ExitThread()`. Эта функция должным образом освобождает стек потока и сам потоковый объект API. По завершении ее работы поток перестает существовать и все использованные им ресурсы будут освобождены.

По окончании использования объекта `TThread` нужно гарантированно уничтожить и соответствующий объект Object Pascal. Только в этом случае можно быть уверенным в корректном освобождении всей памяти, занимаемой этим объектом. И хотя это происходит автоматически по завершении процесса, вам, возможно, стоит заняться освобождением объекта несколько раньше, чтобы исключить возникновение утечки памяти в приложении. Простейший способ гарантированно освободить объект `TThread` состоит в установке его свойства `FreeOnTerminate` равным значению `True`. Причем это можно сделать в любое время до завершения выполнения метода `Execute()`. Например, для объекта `TTestThread` эта установка может быть выполнена следующим образом:

```
procedure TTestThread.Execute;
var
  i: integer;
begin
  FreeOnTerminate := True;
  for i := 1 to 2000000 do
    inc(Answer, Round(Abs(Sin(Sqrt(i)))));
end;
```

Объект `TThread` также имеет событие `OnTerminate`, которое происходит при завершении работы потока. Допускается освобождения объекта `TThread` внутри обработчика этого события.

Событие `OnTerminate` объекта `TThread` вызывается в контексте основного потока вашего приложения. Это значит, что внутри обработчика этого события доступ к свойствам и методам VCL разрешается выполнять свободно, не прибегая к услугам метода `Synchronize()`, о котором пойдет речь в следующем разделе.

Нужно иметь в виду, что метод `Execute()` объекта потока самолично несет ответственность за проверку состояния свойства `Terminated` с целью определения необходимости в досрочном завершении. Хотя это означает еще одно усложнение, о котором следует помнить

при работе с потоками, обратной стороной этой детали архитектуры класса является полная гарантия того, что никакая "нечистая сила" не выдернет коврик из-под ваших ног в самый неподходящий момент. А значит, всегда будет существовать возможность выполнить все необходимые операции очистки по окончании работы потока. Подобную проверку состояния свойства `Terminated` довольно легко добавить в метод `Execute()` объекта `TTestThread` — она будет выглядеть следующим образом:

```
procedure TTestThread.Execute;
var
  i: integer;
begin
  FreeOnTerminate := True;
  for i := 1 to 2000000 do begin
    if Terminated then Break;
    inc(Answer, Round(Abs(Sin(Sqrt(i)))));
  end;
end;
```

В случае аварийной ситуации для завершения выполнения потока можно также использовать функцию Win32 API `TerminateThread()`. К этой функции следует обращаться только в том случае, когда другие варианты отсутствуют — например, когда поток попадает в бесконечный цикл и перестает реагировать на сообщения. Функция определяется следующим образом:

```
function TerminateThread(hThread: THandle; dwExitCode: DWORD);
```

Свойство `THandle` объекта `TThread` содержит дескриптор потока, присвоенный ему функциями API, поэтому к данной функции можно обращаться и с помощью следующего синтаксиса:

```
TerminateThread(MyHosedThread.Handle, 0);
```

При использовании этой функции следует помнить о крайне неприятных побочных эффектах, которые она способна вызывать. Во-первых, ее поведение различно в операционных системах Windows NT/2000 и Windows 95/98. Под управлением Windows 95/98 функция `TerminateThread()` освобождает стек, связанный с потоком, а в Windows NT стек не освобождается до тех пор, пока не завершится процесс. Во-вторых, во всех операционных системах Win32 функция `TerminateThread()`, невзирая ни на что, просто останавливает выполнение в случайном месте и не позволяет блоку `try..finally` освободить ресурсы. Это означает, что файлы, открытые потоком, могут остаться незакрытыми, память, выделенная потоком, может не быть освобождена и т.д. Кроме того, динамически компонуемые библиотеки (DLL), загруженные этим процессом, не получают соответствующего уведомления при разрушении потока с помощью функции `TerminateThread()`, что может вызвать проблемы при закрытии DLL. Для получения более подробной информации о возможности уведомления DLL о работе потоков обращайтесь к главе 9, "Динамически компонуемые библиотеки".

Синхронизация с подпрограммами библиотеки VCL

Как уже неоднократно упоминалось выше в этой главе, прямой доступ к свойствам или методам компонентов библиотеки VCL следует выполнять только из основного потока приложения. Это значит, что любой код, который получает доступ или обновляет данные поль-

зовательского интерфейса в приложении, должен выполняться только в контексте основного потока. Недостатки такой архитектуры вполне очевидны, и может показаться, что это требование слишком уж связывает вам руки, однако уже одно то, что вы знаете об этом ограничении, дает определенное преимущество.

Преимущества однопоточного интерфейса пользователя

Прежде всего, наличие только одного потока, получающего доступ к интерфейсу пользователя, значительно уменьшает сложность приложения. В соответствии с требованиями системы Win32, каждый поток, создающий диалоговое окно, должен иметь собственный цикл сообщений на основе использования функции `GetMessage()`. Нетрудно представить, что наличие сообщений, приходящих от различных источников, существенно затруднит отладку приложения. Поскольку очередь сообщений приложения предусматривает их последовательную обработку, т.е. выполнение полной обработки одного сообщения до перехода к следующему, то в большинстве случаев приложение попадает в зависимость от последовательности поступления определенных сообщений, приходящих до или после других. Добавление еще одного цикла сообщений совершенно нарушает правила игры, существующие при последовательном вводе, уступая дорогу потенциальным проблемам синхронизации и вызывая необходимость добавления более сложной программной реализации, предусматривающей решение этих проблем.

Кроме того, поскольку компоненты VCL могут работать только при условии доступа к ним лишь одного потока в каждый момент времени, становится очевидной необходимость создания кода для синхронизации нескольких потоков внутри подпрограмм VCL. Общий итог всего сказанного состоит в том, что общая эффективность приложения прямо зависит от простоты архитектуры, выбранной при его построении.

Метод `Synchronize()`

В классе `TThread` определен метод `Synchronize()`, который позволяет вызывать некоторые из методов этого класса прямо из основного потока приложения. Определение метода `Synchronize()` имеет следующий вид:

```
procedure Synchronize(Method: TThreadMethod);
```

Параметр `Method` имеет тип `TThreadMethod` (означающий процедурный метод, не использующий никаких параметров), который определяется следующим образом:

```
type
  TThreadMethod = procedure of object;
```

Метод, передаваемый в качестве параметра `Method`, и является как раз тем методом, который затем выполняется из основного потока приложения. Возвращаясь к примеру с классом `TTestThread`, предположим, что мы хотим отобразить результат вычислений в строке редактирования на главной форме. Это можно сделать путем ввода в класс `TTestThread` метода, который вносит необходимые изменения в свойство `Text` строки редактирования, и последующего вызова этого метода с помощью процедуры `Synchronize()`.

Предположим, этот метод называется `GiveAnswer()`. В листинге 11.1 представлен полный исходный 0од модуля `ThrdU`, который включает программную реализацию процесса обновления упомянутой выше строки редактирования на главной форме.

Листинг 11.1. Модуль ThrdU.PAS

```
unit ThrdU;

interface

uses
  Classes;

type
  TTestThread = class(TThread)
  private
    Answer: integer;
  protected
    procedure GiveAnswer;
    procedure Execute; override;
  end;

implementation

uses SysUtils, Main;

{ TTestThread }

procedure TTestThread.GiveAnswer;
begin
  MainForm.Edit1.Text := InttoStr(Answer);
end;

procedure TTestThread.Execute;
var
  I: Integer;
begin
  FreeOnTerminate := True;
  for I := 1 to 2000000 do
  begin
    if Terminated then Break;
    Inc(Answer, Round(Abs(Sin(Sqrt(I)))));
    Synchronize(GiveAnswer);
  end;
end;

end.
```

Мы уже знаем, что с помощью метода `Synchronize()` можно выполнять методы в контексте основного потока, но до сих пор мы обращались с ним как с таинственным черным ящиком. Мы знали, *что* он делает, но не знали *как*. Настало время приоткрыть завесу этой тайны.

При первом создании вторичного (или дополнительного) потока в приложении библиотека VCL создает и далее поддерживает скрытое *окно потока* в контексте своего основного потока. Единственная цель этого окна состоит в организации последовательности вызовов процедур, выполненных посредством метода `Synchronize()`.

Метод Synchronize() сохраняет метод, заданный параметром Method, в закрытом поле FMethod и посылает определенное библиотекой VCL сообщение CM_EXECPROC в окно потока, передавая в качестве параметра lParam этого сообщения значение Self (в этом случае Self указывает на объект TThread). Когда процедура окна потока получает сообщение CM_EXECPROC, она вызывает метод, заданный в поле FMethod с помощью экземпляра объекта TThread, переданного параметром lParam. Напомним, что поскольку это окно было создано в контексте основного потока, процедура окна для него также выполняется основным потоком. Следовательно, метод, указанный в поле FMethod, также вызывается основным потоком.

Графическая иллюстрация того, что происходит внутри метода Synchronize(), представлена на рис. 11.2.

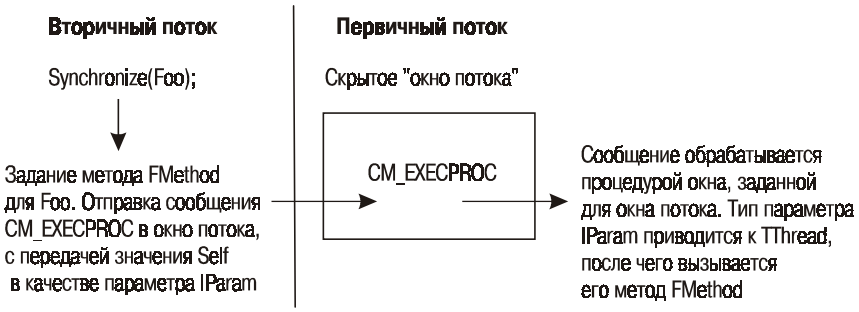


Рис. 11.2. Схема работы метода Synchronize()

Использование сообщений для синхронизации

В качестве альтернативы метода TThread.Synchronize() существует другой способ синхронизации потоков, который заключается в использовании сообщений, передаваемых между потоками. В этом случае для отправки сообщений в окна, действующие в контексте другого потока, используется функция SendMessage() или PostMessage(). Например, приведенный ниже код можно было бы применить для вывода текста в строке редактирования, расположенной в другом потоке:

```
var
  S: string;
begin
  S := 'hello from threadland';
  SendMessage(SomeEdit.Handle, WM_SETTEXT, 0, Integer(PChar(S)));
end;
```

Демонстрационное приложение

Чтобы полностью проиллюстрировать работу нескольких потоков в среде Delphi, создадим проект EZThrd. Поместим на главную форму окно примечания, как показано на рис. 11.3.

Исходный текст главного модуля приложения EZThrd приведен в листинге 11.2.

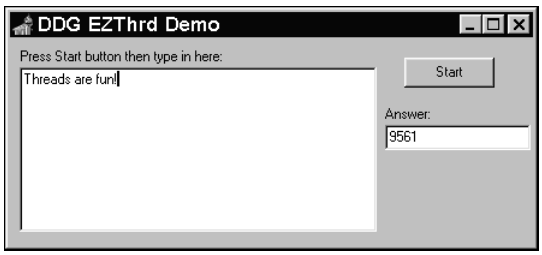


Рис. 11.3. Главная форма демонстрационного приложения EZThrd

Листинг 11.2. Модуль MAIN.PAS демонстрационного приложения EZThrd

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Memo1: TMemo;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.Button1Click(Sender: TObject);
var
  NewThread: TTestThread;
begin
  NewThread := TTestThread.Create(False);
end;

end.
```

Обратите внимание на то, что после щелчка на кнопке и активизации вторичного потока попрежнему можно вводить информацию в окно примечания, как если бы вторичного потока не было вовсе. По завершении вычислений результат будет отображен в строке редактирования.

Приоритеты и составление расписания

Как упоминалось ранее, операционная система отвечает за выделение каждому потоку нескольких циклов процессора, в течение которых они могут работать. Количество времени, выделяемое отдельному потоку, зависит от назначаемого ему приоритета. Общий приоритет отдельного потока определяется путем объединения приоритета процесса, породившего поток (он называется *классом приоритета*), и приоритета самого потока, который называется *относительным приоритетом*.

Класс приоритета процесса

Класс приоритета процесса описывает приоритет определенного процесса, выполняющегося в системе. Система Win32 поддерживает четыре различных класса приоритета: `Idle`, `Normal`, `High` и `Realtime`. По умолчанию любому процессу присваивается класс `Normal`. Каждый из перечисленных классов имеет соответствующий признак, определенный в модуле `Windows`. С помощью операции `OR` (или) любой из этих признаков можно логически сложить с параметром `dwCreationFlags` функции `CreateProcess()`, что позволит при порождении процесса установить ему необходимый приоритет. Кроме того, эти признаки можно использовать и для динамической настройки класса приоритета данного процесса, как это будет показано ниже. Все классы можно представить уровнями приоритета, которые выражаются числовыми значениями, лежащими в диапазоне от 4 до 24 включительно.

На заметку В среде Windows NT модификация класса приоритета процесса требует наличия у этого процесса специальных привилегий. С помощью стандартных установок процессам можно присвоить определенные классы, но они могут быть отключены системными администраторами — в частности, на интенсивно загруженных серверах Windows NT/2000/

В табл. 11.1 приведены все существующие классы приоритета, соответствующие им признаки и числовые значения.

Таблица 11.1. Классы приоритетов процессов

Класс	Признак	Значение
`Idle`	`IDLE_PRIORITY_CLASS`	`$40`
`Below normal*`	`BELOW_NORMAL_PRIORITY_CLASS`	`$4000`
`Normal`	`NORMAL_PRIORITY_CLASS`	`$20`
`Above normal*`	`ABOVE_NORMAL_PRIORITY_CLASS`	`$8000`
`High`	`HIGH_PRIORITY_CLASS`	`$80`
`Realtime`	`REALTIME_PRIORITY_CLASS`	`$100`

* Эти значения доступны только в середе Windows 2000 и соответствующие константы флажков отсутствуют в модуле `Windpws.pas` Delphi 5.

Для динамического считывания и установки класса приоритета данного процесса в Win32 предусмотрены функции `GetPriorityClass()` и `SetPriorityClass()` соответственно. Эти функции определены следующим образом:

```
function GetPriorityClass(hProcess: THandle): DWORD; stdcall;
function SetPriorityClass(hProcess: THandle; dwPriorityClass: DWORD): BOOL;
  stdcall;
```

В обеих функциях параметр `hProcess` представляет дескриптор процесса. Чаще всего эти функции используются для получения доступа к значению класса приоритета собственного процесса. В подобном случае можно обратиться и к функции Win32 API `GetCurrentProcess()`, которая определяется следующим образом:

```
function GetCurrentProcess: THandle; stdcall;
```

Значение, возвращаемое этими функциями, представляет собой псевдодескриптор текущего процесса. Приставка *псевдо* означает, что ни одна из этих функций не создает новый дескриптор, а возвращаемое значение не должно закрываться с помощью функции `CloseHandle()`. Это значение является просто дескриптором, который можно использовать для ссылки на существующий дескриптор.

Чтобы установить класс приоритета приложения равным `High`, используйте следующий код:

```
if not SetPriorityClass(GetCurrentProcess, HIGH_PRIORITY_CLASS) then
  ShowMessage('Error setting priority class.');
```

В большинстве случаев следует избегать установки класса приоритета любого процесса равным `Realtime`. Поскольку большинство потоков операционных систем работает с классом приоритета, который ниже, чем `Realtime`, то ваш поток будет получать больше процессорного времени, чем сама операционная система, а это может вызвать некоторые неожиданные проблемы.

Даже неоправданная установка класса приоритета равным `High` может вызвать проблемы, если потоки процесса не тратят б**о**льшую часть времени на простои или на ожидание внешних событий (например, ввода-вывода). Один высокоприоритетный поток будет отбирать все процессорное время у низкоприоритетных потоков и процессов до тех пор, пока он или не заблокирует какое-нибудь событие, или не перейдет в состояние ожидания, или не займется обработкой сообщений. Неправильное назначение приоритетов легко может перечеркнуть все преимущества от использования многозадачности.

Относительный приоритет

Второй составляющей, которая входит в определение общего приоритета потока, является *относительный приоритет* отдельного потока. Следует подчеркнуть, что класс приоритета связан с процессом, а относительный приоритет — с отдельными потоками внутри процесса. Потоку можно назначить один из семи возможных относительных приоритетов: *Idle* (ожидания), *Lowest* (низший), *Below Normal* (ниже нормального), *Normal* (нормальный), *Above Normal* (выше нормального), *Highest* (высший) или *Time Critical* (критический по времени).

Класс TThread имеет открытое свойство Priority перечислимого типа TThreadPriority. Для каждого относительного приоритета в этом типе существует свой элемент перечисления:

```
type
  TThreadPriority = (tpIdle, tpLowest, tpLower, tpNormal, tpHigher,
    tpHighest, tpTimeCritical);
```

Приоритет любого объекта TThread можно получить или установить путем простого чтения или записи его свойства Priority. С помощью следующей строки кода присваивается высший приоритет экземпляру потомка класса TThread, который носит имя MyThread:

```
MyThread.Priority := tpHighest;
```

Подобно классам приоритетов, каждому относительному приоритету соответствует определенное числовое значение. Разница лишь в том, что значение относительного приоритета имеет знак, который при определении общего приоритета потока внутри системы влияет на результат суммирования класса приоритета процесса и относительного приоритета. Поэтому относительный приоритет иногда называется *дельта-приоритетом*. Общий приоритет потока может выражаться значением, лежащим в диапазоне от 1 до 31 (где число 1 означает самый низкий приоритет). В модуле Windows определены константы, представляющие знаковые значения для каждого приоритета. В табл. 11.2 показано соответствие констант API и элементов перечисления в типе TThreadPriority.

Таблица 11.2. Относительные приоритеты для потоков

TThreadPriority	Константа	Значение
tpIdle	THREAD_PRIORITY_IDLE	-15*
tpLowest	THREAD_PRIORITY_LOWEST	-2
tpBelowNormal	THREAD_PRIORITY_BELOW_NORMAL	-1
tpNormal	THREAD_PRIORITY_NORMAL	0
tpAboveNormal	THREAD_PRIORITY_ABOVE_NORMAL	1
tpHighest	THREAD_PRIORITY_HIGHEST	2
tpTimeCritical	THREAD_PRIORITY_TIME_CRITICAL	15*

Помеченные звездочкой (*) значения относительных приоритетов tpIdle и tpTimeCritical, в отличие от других, не добавляются к классу приоритета для определения общего приоритета потока. Общий приоритет потока, относительный приоритет которого равен tpIdle, независимо от класса приоритета устанавливается равным 1. Исключением из этого правила является класс Realtime, который, объединяясь с относительным приоритетом tpIdle, имеет значение общего приоритета, равное 16. Общий приоритет любого потока с относительным приоритетом tpTimeCritical, независимо от его класса, равен 15. Исключением из этого правила является класс приоритета Realtime, который при объединении с относительным приоритетом tpTimeCritical имеет значение общего приоритета, равное 31.

Приостановка и восстановление работы потоков

Давайте вспомним, что говорилось ранее в этой главе о конструкторе Create() класса TThread. Там указывалось, что поток может быть создан в состоянии приостановки, и для того чтобы он начал выполняться, нужно вызвать метод Resume(). Логично предположить, что поток может быть приостановлен и вновь запущен в работу динамически. Эта задача решается с помощью методов Suspend() и Resume().

Хронометраж потока

При программировании под управлением Windows 3.x нетрудно написать подпрограмму определения количества времени, которое требуется на выполнение определенных вычислений. Используя функцию GetTickCount() или timeGetTime(), можно предложить следующий фрагмент кода:

```
var
  StartTime, Total: Longint;
begin
  StartTime := GetTickCount;
  {  Выполнение вычислений }
  Total := GetTickCount - StartTime;
```

Но в многопоточной среде это сделать намного труднее, поскольку выполнение вашего приложения может быть приостановлено в середине вычислений ради предоставления циклов процессора другим процессам. Следовательно, любой хронометраж, если он опирается на системное время, не в состоянии дать истинную картину затрат времени, требуемых для выполнения вычислений в отдельном потоке.

Во избежание подобных проблем в системе Win32 (вариант Windows NT/2000) предусмотрена функция GetThreadTimes(), которая предоставляет довольно подробную информацию о времени работы потока. Объявление этой функции выглядит следующим образом:

```
function GetThreadTimes(hThread: THandle; var lpCreationTime, lpExitTime,
    lpKernelTime, lpUserTime: TFileTime): BOOL; stdcall;
```

Параметр hThread представляет собой дескриптор потока, для которого требуется получить информацию о времени работы. Другие параметры передаются по ссылке и заполняются функцией.

- lpCreationTime. Время создания потока.
- lpExitTime. Время окончания работы потока. Если поток еще работает, это значение не определено.
- lpKernelTime. Время, затраченное потоком на выполнение кода операционной системы.
- lpUserTime. Время, затраченное потоком на выполнение кода приложения.

Этих четыре параметра имеют тип TFileTime, который определен в модуле Windows следующим образом:

```
type
  TFileTime = record
    dwLowDateTime: DWORD;
    dwHighDateTime: DWORD;
  end;
```

Определение этого типа несколько необычно, но оно действует в Win32 API, и с этим придется мириться. Элементы записи dwLowDateTime и dwHighDateTime объединяются в учетверенное слово и образуют 64-разрядное значение, представляющее количество 100-нанасекундных интервалов, прошедших с 1 января 1601 года. Это значит, что если бы мы захотели смоделировать движение английского флота, который нанес поражение испанской армаде в 1588 году, то тип TFileTime был бы совершенно неприемлем для отслеживания значений времени… Впрочем, мы несколько отвлеклись.

Поскольку тип TFileTime является 64-разрядным, то для выполнения арифметических действий над значениями этого типа можно использовать операцию приведения типа TFileTime к типу Int64. Следующий пример демонстрирует возможный способ быстрого сравнения значений типа TFileTime:

```
If Int64(UserTime) > Int64(KernelTime) then Beep;
```

Для упрощения работы со значениями TFileTime в среде Delphi имеются следующие функции, позволяющие выполнить преобразования между типами TFileTime и TDateTime в прямом и обратном направлениях:

```
function FileTimeToDateTime(FileTime: TFileTime): TDateTime;
var
  SysTime: TSystemTime;
begin
  if not FileTimeToSystemTime(FileTime, SysTime) then
    raise EConvertError.CreateFmt('FileTimeToSystemTime failed. ' +
      'Error code %d', [GetLastError]);
  with SysTime do
    Result := EncodeDate(wYear, wMonth, wDay) +
      EncodeTime(wHour, wMinute, wSecond, wMilliseconds)
end;

function DateTimeToFileTime(DateTime: TDateTime): TFileTime;
var
  SysTime: TSystemTime;
begin
  with SysTime do
  begin
    DecodeDate(DateTime, wYear, wMonth, wDay);
    DecodeTime(DateTime, wHour, wMinute, wSecond, wMilliseconds);
    wDayOfWeek := DayOfWeek(DateTime);
  end;
  if not SystemTimeToFileTime(SysTime, Result) then
    raise EConvertError.CreateFmt('SystemTimeToFileTime failed. ' +
      'Error code %d', [GetLastError]);
end;
```

Не забывайте, что функция GetThreadTimes() реализована только в Windows NT/2000. При ее вызове в среде Windows 95/98 всегда возвращается значение False. К сожалению, в Windows 95/98 не предусмотрен никакой механизм для получения информации о времени работы потоков.

Управление несколькими потоками

Как уже отмечалось, несмотря на то что с помощью потоков можно решить множество проблем программирования, они приносят с собой новые типы проблем, с которыми приходится сталкиваться в создаваемых приложениях. Чаще всего эти новые проблемы связаны с доступом со стороны нескольких потоков к таким глобальным ресурсам, как глобальные переменные или дескрипторы. Кроме того, проблемы могут возникнуть в случае, когда, например, нужно обес-

печить постоянное возникновение некоторого события в одном потоке перед или после некоторого другого события во втором потоке. В этом разделе мы обсудим, как разрешить эти проблемы, используя средства, предоставленные в Delphi для хранения локальных переменных потоков, а также средства API, предназначенные для синхронизации потоков.

Хранение локальных данных потоков

Поскольку каждый поток представляет собой отдельный и независимый путь выполнения программного кода внутри процесса, было бы логично предположить, что на определенном этапе потребуется какое-либо средство хранения данных, связанных с каждым потоком. Существует три метода хранения данных, уникальных для каждого потока. Первый, и самый простой, состоит в использовании локальных переменных (в стеке). Поскольку каждый поток получает собственный стек, при выполнении одной процедуры или функции он будет иметь и собственную копию локальных переменных. Второй метод заключается в сохранении локальной информации в объекте, производном от класса `TThread`. И, наконец, можно также использовать зарезервированное слово Object Pascal `threadvar`, чтобы воспользоваться преимуществами хранения локальной информации потока на уровне операционной системы.

Использование объекта `TThread` для хранения данных

Если сравнивать последних два варианта хранения данных потоков, то, бесспорно, следует остановить выбор на способе хранения данных в объекте потомка класса `TThread`, поскольку он проще и эффективнее метода с использованием зарезервированного слова `threadvar` (речь о нем пойдет позже). Для объявления локальных данных потока этим способом достаточно добавить их в определение потомка класса `TThread`:

```
type
  TMyThread = class(TThread)
  private
    FLocalInt: Integer;
    FLocalStr: String;
    .
    .
    .
  end;
```

Доступ к полю любого объекта осуществляется почти в 10 раз быстрее, чем доступ к переменной `threadvar`, поэтому данные потоков следует сохранять в потомке класса `TThread`, если, конечно, это возможно. Данные, которые существуют только в течение продолжительности жизни отдельной процедуры или функции, лучше сохранять в локальных переменных, поскольку доступ к ним еще быстрее, чем к полям объекта `TThread`.

Объявление `threadvar`: хранение локальных данных потоков с помощью интерфейса API

Выше упоминалось о том, что каждому потоку для хранения локальных переменных предоставляется его собственный стек, в то время как глобальные данные должны совместно использоваться всеми потоками внутри приложения. Допустим, что у вас есть процедура, которая ус-

танавливает или отображает значение глобальной переменной, причем она построена так, что при передаче ей текстовой строки происходит установка, а при передаче пустой строки — отображение этой глобальной переменной. Такая процедура может иметь следующий вид:

```
var
  GlobalStr: String;
procedure SetShowStr(const S: String);
begin
  if S = '' then
    MessageBox(0, PChar(GlobalStr), 'The string is...', MB_OK)
  else
    GlobalStr := S;
end;
```

Если эта процедура вызывается в контексте только одного потока, никаких проблем не возникнет. Первый раз ее можно будет вызвать для установки значения переменной `GlobalStr`, а второй — для отображения этого значения. Давайте разберемся, что же произойдет, если два или больше потоков вызовут эту процедуру в произвольный момент времени. В этом случае возможна ситуация, когда один поток вызовет эту процедуру для установки строки, после чего процессорное время будет отдано другому потоку, который также вызовет эту процедуру для установки своей строки. И к тому времени, когда операционная система передаст обратно процессорное время первому потоку, значение переменной `GlobalStr` будет безнадежно утрачено.

Для ситуаций, подобных описанной выше, в Win32 предусмотрено средство, известное под названием *хранение локальных данных потоков* (thread-local storage), с помощью которого можно создавать отдельные копии глобальных переменных для каждого действующего потока. Delphi великолепно инкапсулирует это средство с помощью зарезервированного слова `threadvar`. От вас требуется лишь объявить любые глобальные переменные, которые вы собираетесь использовать отдельно для каждого потока, внутри блока `threadvar` (вместо `var`) — и делу конец. Переопределение переменной `GlobalStr` проще простого:

```
threadvar
  GlobalStr: String;
```

Модуль, текст которого представлен в листинге 11.3, иллюстрирует именно эту проблему. Вам предлагается разобраться в главном модуле приложения Delphi, в форме которого содержится всего одна кнопка. По щелчку на этой кнопке вызывается процедура установки, а затем и отображения значения переменной `GlobalStr`. После этого создается другой поток, в котором устанавливается и отображается значение его собственной переменной `GlobalStr`. После создания вторичного потока первичный вновь вызывает процедуру `SetShowStr` для отображения переменной `GlobalStr`.

Листинг 11.3. Модуль `MAIN.PAS` — демонстрация хранения локальных переменных потоков

```
unit Main;

interface

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

```
type
  TMainForm = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

{ ЗАМЕЧАНИЕ: Перенесите переменную GlobalStr из блока var в блок threadvar
  и обратите внимание на различия в результатах работы приложения. }
var
//threadvar
  GlobalStr: string;

type
  TTLSThread = class(TThread)
  private
    FNewStr: String;
  protected
    procedure Execute; override;
public
    constructor Create(const ANewStr: String);
  end;

procedure SetShowStr(const S: String);
begin
  if S = '' then
    MessageBox(0, PChar(GlobalStr), 'The string is...', MB_OK)
  else
    GlobalStr := S;
end;

constructor TTLSThread.Create(const ANewStr: String);
begin
  FNewStr := ANewStr;
  inherited Create(False);
end;

procedure TTLSThread.Execute;
begin
  FreeOnTerminate := True;
```

```
  SetShowStr(FNewStr);
  SetShowStr('');
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  SetShowStr('Hello world');
  SetShowStr('');
  TTLSThread.Create('Dilbert');
  Sleep(100);
  SetShowStr('');
end;

end.
```

Попробуйте запустить это приложение, сначала объявив переменную `GlobalStr` с помощью зарезервированного слова `var`, а затем — `threadvar`, и проанализируйте различие в результатах работы.

На заметку

В этой демонстрационной программе после создания вторичного потока вызывается процедура Win32 API `Sleep()`, которая объявляется следующим образом:

```
procedure Sleep(dwMilliseconds: DWORD); stdcall;
```

Процедура `Sleep()` сообщает операционной системе о том, что текущий поток не нуждается в дополнительных циклах процессора в течение миллисекунд, заданных параметром `dwMilliseconds`. Вставка вызова этой процедуры в данный код обусловлена необходимостью создания эффекта имитации условий, когда система работает в режиме большей многозадачности, а также необходимостью внесения в приложение "случайностей", связанных с тем, когда и какой именно поток будет выполняться.

Часто приемлемым вариантом является передача в качестве параметра `dwMilliseconds` нулевого значения. И хотя в этом случае текущий поток не застрахован от работы в течение определенного времени, тем не менее такой ход заставляет операционную систему передать процессорные циклы одному из ожидающих потоков с равным или высшим приоритетом.

Процедуру `Sleep()` следует осторожно использовать при работах, связанных с решением загадочных проблем хронометража. Эта процедура может сносно справиться с частной проблемой на вашей машине, но проблемы хронометража, которые не решены в общем случае, могут проявиться на чужом компьютере, особенно если эта другая машина работает значительно быстрее или медленнее, чем ваша, или если количество процессоров на чужой машине не совпадает с их количеством на вашей машине.

Синхронизация потоков

При работе с несколькими потоками вам часто придется синхронизировать доступ потоков к определенным данным или ресурсам. Предположим, у вас есть приложение, в котором один поток используется для считывания файла в память, а другой — для подсчета количества символов в файле. Само собой разумеется, что, до тех пор пока файл полностью не загрузится в память, вы не сможете сосчитать в нем все символы. Но поскольку каждая операция выполняется в своем собственном потоке, операционная система вольна обращаться с ними как с двумя совершенно несвязанными задачами. Чтобы справиться с этой проблемой, необходимо синхронизировать этих два потока таким образом, чтобы поток, подсчитывающий символы, не начинал работать, пока не завершится поток, загружающий файл.

Существует два типа проблем, связанных с синхронизацией потоков. В Win32 предусмотрены различные пути их решения. В этом разделе вы найдете примеры методов синхронизации потоков с помощью критических секций (critical sections), мьютексов (mutexes), семафоров (semaphores) и событий.

В чем же состоит суть проблемы, связанной с синхронизацией потоков? Для иллюстрации предположим, что имеется массив целых чисел, который нужно инициализировать возрастающими значениями. Сначала требуется "пройти" по массиву и установить значения от 1 до 128, а затем переинициализировать этот массив значениями от 128 до 255. После этого полученный поток отображается в окне списка. Эта задача может быть решена путем выполнения инициализаций в двух отдельных потоках. Попытка реализовать этот подход показана в коде модуля, представленного в листинге 11.4.

Листинг 11.4. Модуль, в котором реализована попытка инициализировать один массив в двух потоках

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    procedure ThreadsDone(Sender: TObject);
  end;

  TFooThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
  MaxSize = 128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;
```

```
function GetNextNumber: Integer;
begin
  Result := NextNumber;  //Функция возвращает глобальную переменную
  Inc(NextNumber);       // Инкремент глобальной переменной
end;

procedure TFooThread.Execute;
var
  i: Integer;
begin
  OnTerminate := MainForm.ThreadsDone;
  for i := 1 to MaxSize do
  begin
    GlobalArray[i] := GetNextNumber;  // Устанавливаем элемент массива
    Sleep(5);                         // Позволяем потокам "перемешаться"
  end;
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  Inc(DoneFlags);
  if DoneFlags = 2 then       // Оба потока завершены
    for i := 1 to MaxSize do
      { Заполняем список элементами массива }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  TFooThread.Create(False);  // Создаем потоки
  TFooThread.Create(False);
end;

end.
```

Поскольку оба потока выполняются одновременно, при инициализации массива происходит искажение равномерности возрастания значений его элементов. Чтобы убедиться в этом, взгляните на результат работы данной программы, показанный на рис. 11.4.

Решение этой проблемы лежит в синхронизации двух потоков во время их доступа к глобальному массиву, чтобы они не выполняли свою работу в одно и то же время. К реализации решения этой проблемы можно подойти по-разному.

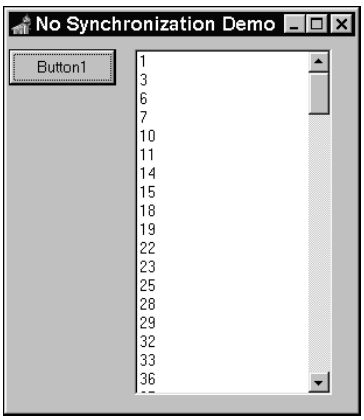


Рис. 11.4. Результат несинхронизированной инициализации массива

Критические секции

Критические секции предоставляют собой один из самых простых способов синхронизации потоков. Критическая секция — это некоторый участок кода, который в каждый момент времени может выполняться только одним из потоков. Если код, используемый для инициализации массива, поместить в критическую секцию, то другие потоки не смогут войти в этот участок кода до тех пор, пока первый поток не завершит его выполнение.

До использования критической секции необходимо инициализировать ее с помощью процедуры Win32 API `InitializeCriticalSection()`, которая определяется следующим образом:

```
procedure InitializeCriticalSection(var lpCriticalSection:
  TRTLCriticalSection); stdcall;
```

Параметр `lpCriticalSection` представляет собой запись типа `TRTLCriticalSection`, которая передается по ссылке. Точное определение записи `TRTLCriticalSection` не имеет большого значения, поскольку вам вряд ли понадобится когда-либо заглядывать в ее содержимое. От вас требуется лишь передать неинициализированную запись в параметр `lpCriticalSection`, и эта запись будет тут же заполнена процедурой.

На заметку

Фирма Microsoft преднамеренно не выставляет напоказ структуру записи `TRTLCriticalSection`, поскольку ее содержимое варьируется при переходе от одной аппаратной платформы к другой, а также потому, что некорректное обращение с содержимым этой структуры потенциально может внести хаос в работу процесса. В системах, построенных на платформе Intel, структура критического раздела содержит счетчик, поле с дескриптором текущего потока и (возможно) дескриптор системного события. В системах, построенных на платформе Alpha, счетчик заменен структурой данных Alpha-CPU, называемой взаимоблокировкой (spinlock), которая более эффективна, чем вариант Intel.

После заполнения записи в программе можно создать критическую секцию, поместив некоторый участок ее текста между вызовами функций `EnterCriticalSection()` и `LeaveCriticalSection()`. Эти процедуры определяются следующим образом:

```
procedure EnterCriticalSection(var lpCriticalSection:
  TRTLCriticalSection); stdcall;
procedure LeaveCriticalSection(var lpCriticalSection:
  TRTLCriticalSection); stdcall;
```

Нетрудно догадаться, что параметр `lpCriticalSection`, который передается этим процедурам, является не чем иным, как записью, созданной процедурой `InitializeCriticalSection()`.

По окончании работы с записью `TRTLCriticalSection` необходимо освободить ее, вызвав процедуру `DeleteCriticalSection()`, которая определяется следующим образом:

```
procedure DeleteCriticalSection(var lpCriticalSection:
  TRTLCriticalSection); stdcall;
```

В листинге 11.5 демонстрируется способ синхронизации инициализирующих массив потоков с помощью критических секций.

Листинг 11.5. Использование критических секций

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    procedure ThreadsDone(Sender: TObject);
  end;

  TFooThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
  MaxSize = 128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;
  CS: TRTLCriticalSection;

function GetNextNumber: Integer;
begin
  Result := NextNumber;  //Функция возвращает глобальную переменную
  inc(NextNumber);       // Увеличение глобальной переменной
end;

procedure TFooThread.Execute;
var
  i: Integer;
```

```
begin
  OnTerminate := MainForm.ThreadsDone;
  EnterCriticalSection(CS);       // Начало критической секции
  for i := 1 to MaxSize do
  begin
    GlobalArray[i] := GetNextNumber;  // Устанавливаем элемент массива
    Sleep(5);                         // Позволяем потокам "перемешаться"
  end;
  LeaveCriticalSection(CS);        // Конец критической секции
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  inc(DoneFlags);
  if DoneFlags = 2 then
  begin // Оба потока завершены
    for i := 1 to MaxSize do
      { Заполняем список данными массива }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
    DeleteCriticalSection(CS);
  end;
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  InitializeCriticalSection(CS);
  TFooThread.Create(False);  // Создание потоков
  TFooThread.Create(False);
end;

end.
```

После того как первый поток вызовет процедуру `EnterCriticalSection()`, всем другим потокам вход в этот блок кода будет запрещен. А следующий поток, который дойдет до этой строки кода, будет остановлен, т.е. перейдет в состояние ожидания до тех пор, пока первый поток не вызовет процедуру `LeaveCriticalSection()`. В этот момент система вновь активизирует второй поток и разрешит ему пройти через критическую секцию. На рис. 11.5 показан результат работы этого приложения, когда потоки уже синхронизированы.

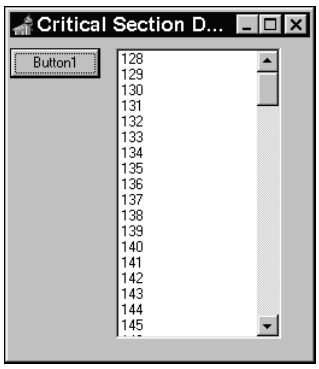


Рис. 11.5. Результат синхронизированной инициализации массива

Мьютексы

По принципу своего действия мьютексы (**MUT**ual **EX**clusions — взаимоисключения) очень похожи на критические секции, за исключением двух моментов. Во-первых, мьютексы можно использовать для синхронизации потоков, переступая через границы процессов. Во-вторых, мьютексу можно присвоить имя и путем ссылки на это имя создать дополнительные дескрипторы существующих объектов мьютексов.

Помимо семантических особенностей, самое большое различие между критическими секциями и такими объектами событий, как мьютексы, заключается в производительности. Критические секции очень эффективны: если нет потоковых коллизий, то на вход или выход из критической секции уходит всего 10–15 системных тактов процессора. Но если для данной критической секции возникает потоковая коллизия, система создает объект события (возможно, мьютекс). В "стоимость" использования таких объектов событий, как мьютексы, входит обязательное вхождение в подпрограммы ядра, что требует переключения контекста процесса и изменения кольцевых уровней, на что уходит от 400 до 600 системных тактов процессора. Причем без этих накладных расходов нельзя обойтись даже в том случае, если в приложении в данный момент вообще нет вторичных потоков или если нет других потоков, которые оспаривают право на защищаемые ресурсы.

Функция, используемая для создания мьютекса, называется `CreateMutex()`, а ее объявление выглядит следующим образом:

```
function CreateMutex(lpMutexAttributes: PSecurityAttributes;
  bInitialOwner: BOOL; lpName: PChar): THandle; stdcall;
```

Параметр `lpMutexAttributes` — это указатель на запись типа `TSecurityAttributes`. Обычно в качестве данного параметра передается значение `nil`, и в этом случае используются атрибуты защиты, действующие по умолчанию.

Параметр `bInitialOwner` определяет, следует ли считать поток, создающий мьютекс, его владельцем. Если этот параметр равен `False`, значит, мьютекс не имеет владельца.

Параметр `lpName` представляет имя мьютекса. Если вы не собираетесь присваивать мьютексу имя, установите этот параметр равным `nil`. Если же значение этого параметра отлично от `nil`, функция выполнит в системе поиск мьютекса с таким же именем. При успешном завершении поиска функция вернет дескриптор найденного мьютекса, в противном случае возвращается дескриптор нового мьютекса.

По завершении использования мьютекса необходимо закрыть его с помощью функции Win32 API `CloseHandle()`.

В листинге 11.6 снова демонстрируется способ синхронизации потоков, инициализирующих массив, но на этот раз с использованием мьютексов.

Листинг 11.6. Использование мьютексов для синхронизации потоков

```
unit Main;

interface

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

```
type
  TMainForm = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    procedure ThreadsDone(Sender: TObject);
  end;

  TFooThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
  MaxSize = 128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;
  hMutex: THandle = 0;

function GetNextNumber: Integer;
begin
  Result := NextNumber;  //Функция возвращает глобальную переменную
  Inc(NextNumber);       // Увеличение значения глобальной переменной
end;

procedure TFooThread.Execute;
var
  i: Integer;
begin
  FreeOnTerminate := True;
  OnTerminate := MainForm.ThreadsDone;
  if WaitForSingleObject(hMutex, INFINITE) = WAIT_OBJECT_0 then
  begin
    for i := 1 to MaxSize do
    begin
      GlobalArray[i] := GetNextNumber;  // Устанавливаем элемент массива
      Sleep(5);                         // Позволяем потокам "перемешаться"
    end;
  end;
```

```
  ReleaseMutex(hMutex);
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  Inc(DoneFlags);
  if DoneFlags = 2 then    // Оба потока завершены
  begin
    for i := 1 to MaxSize do
      { Заполняем список элементами массива }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
    CloseHandle(hMutex);
  end;
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  hMutex := CreateMutex(nil, False, nil);
  TFooThread.Create(False);  // Создаем потоки
  TFooThread.Create(False);
end;

end.
```

Нетрудно заметить, что в этом случае для управления входом потоков в синхронизуемый блок используется функция `WaitForSingleObject()`, объявленная следующим образом:

```
function WaitForSingleObject(hHandle: THandle; dwMilliseconds: DWORD): DWORD;
stdcall;
```

Эта функция предназначена для перевода текущего потока в состояние ожидания, пока объект API, заданный параметром `hHandle`, не станет доступным. При этом состояние ожидания может продлиться вплоть до истечения интервала времени, заданного в миллисекундах параметром `dwMilliseconds`. Термин “доступность” для различных объектов понимается по-разному. Мьютекс становится доступным, если он больше не принадлежит потоку, в то время как, например, процесс становится доступным после его завершения. Помимо реального периода времени, параметр `dwMilliseconds` может также иметь нулевое значение, которое указывает на необходимость проверки состояния объекта и немедленный возврат. Возможно и значение `INFINITE`, указывающее, что ожидать следует до тех пор, пока объект не станет доступным. Значения, которые может возвращать эта функция, перечислены в табл. 11.3.

Повторим еще раз: если мьютекс не принадлежит какому-либо потоку, он находится в доступном состоянии. Первый же поток, вызвавший функцию `WaitForSingleObject()` с запросом на данный мьютекс, получит право собственности на него, а состояние самого объекта мьютекса станет недоступным. Когда поток вызывает функцию `ReleaseMutex()`, передавая в качестве параметра дескриптор принадлежащего ему мьютекса, право собственности на этот мьютекс у данного потока отбирается, а мьютекс вновь становится доступным.

Таблица 11.3. Константы типа ожидания, используемые функцией Win32 API `WaitForSingleObject()`

Значение	Описание
`WAIT_ABANDONED`	Заданный объект является объектом мьютекса, но поток, владевший этим мьютексом, был завершен до его освобождения. Такой мьютекс считается покинутым (abandoned), и в этом случае право собственности на данный объект мьютекса передается вызывающему потоку, а сам мьютекс определяется как недоступный
`WAIT_OBJECT_0`	Состояние заданного объекта определяется как доступное
`WAIT_TIMEOUT`	Установленный интервал времени истек, но состояние объекта определяется как недоступное

На заметку Помимо функции `WaitForSingleObject()`, в Win32 API есть также функции `WaitForMultipleObjects()` и `MsgWaitForMultipleObjects()`, которые позволяют ожидать, пока состояние одного или нескольких объектов не станет доступным. Эти функции описаны в интерактивной справочной системе Win32 API.

Семафоры

Существует еще один метод синхронизации потоков, в котором используются семафорные объекты API. В *семафорах* применен принцип действия мьютексов, но с добавлением одной существенной детали. В них заложена возможность подсчета ресурсов, что позволяет заранее определенному числу потоков одновременно войти в синхронизуемый участок кода. Для создания семафора используется функция `CreateSemaphore()`, которая объявляется следующим образом:

```
function CreateSemaphore(lpSemaphoreAttributes: PSecurityAttributes;
  lInitialCount, lMaximumCount: Longint; lpName: PChar): THandle; stdcall;
```

Как и в случае функции `CreateMutex()`, первым параметром, передаваемым функции `CreateSemaphore()`, является указатель на запись `TSecurityAttributes`, причем значение `Nil` соответствует согласию на использование стандартных атрибутов защиты.

Параметр `lInitialCount` представляет собой начальное значение счетчика семафорного объекта. Это число может находиться в диапазоне от `0` до значения `lMaximumCount`. Семафор доступен, если значение этого параметра больше нуля. Когда поток вызывает функцию `WaitForSingleObject()` или любую другую, ей подобную, значение счетчика семафора уменьшается на единицу. И наоборот, при вызове потоком функции `ReleaseSemaphore()` значение счетчика семафора увеличивается на единицу.

С помощью параметра `lMaximumCount` задается максимальное значение счетчика семафорного объекта. Если семафор используется для подсчета некоторых ресурсов, это число должно представлять общее количество доступных ресурсов.

Параметр `lpName` содержит имя семафора. Поведение этого параметра аналогично поведению одноименного параметра функции `CreateMutex()`.

В листинге 11.7 демонстрируется использование семафоров для выполнения синхронизации потоков при решении проблемы инициализации массива.

Листинг 11.7. Использование семафоров для синхронизации потоков

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    procedure ThreadsDone(Sender: TObject);
  end;

  TFooThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
  MaxSize = 128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;
  hSem: THandle = 0;

function GetNextNumber: Integer;
begin
  Result := NextNumber;  //Функция возвращает глобальную переменную
  Inc(NextNumber);       // Увеличение значения глобальной переменной
end;

procedure TFooThread.Execute;
var
  i: Integer;
  WaitReturn: DWORD;
```

```
begin
  OnTerminate := MainForm.ThreadsDone;
  WaitReturn := WaitForSingleObject(hSem, INFINITE);
  if WaitReturn = WAIT_OBJECT_0 then
  begin
    for i := 1 to MaxSize do
    begin
      GlobalArray[i] := GetNextNumber;  // Устанавливаем элемент массива
      Sleep(5);                         // Позволяем потокам "перемешаться"
    end;
  end;
  ReleaseSemaphore(hSem, 1, nil);
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  Inc(DoneFlags);
  if DoneFlags = 2 then      // Оба потока завершены
  begin
    for i := 1 to MaxSize do
      { Заполняем список элементами массива }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
    CloseHandle(hSem);
  end;
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  hSem := CreateSemaphore(nil, 1, 1, nil);
  TFooThread.Create(False);  // Создаем потоки
  TFooThread.Create(False);
end;

end.
```

Поскольку мы разрешаем только одному потоку проходить через участок кода синхронизации, максимальное значение счетчика для семафора в этом случае равно 1.

Функция `ReleaseSemaphore()` используется для увеличения значения счетчика семафора. Обратите внимание на то, что эта функция имеет больше параметров, чем ее "коллега" `ReleaseMutex()`. Объявление функции `ReleaseSemaphore()` выглядит следующим образом:

```
function ReleaseSemaphore(hSemaphore: THandle; lReleaseCount: Longint;
  lpPreviousCount: Pointer): BOOL; stdcall;
```

С помощью параметра `lReleaseCount` можно задать число, на которое будет уменьшено значение счетчика семафора. При этом старое значение счетчика будет сохранено в переменной типа `Longint`, на которую указывает параметр `lpPreviousCount`, если его значение не равно `Nil`. Скрытый смысл этого средства состоит в том, что семафор никогда не принадлежит ни одному

отдельному потоку. Предположим, что максимальное значение счетчика семафора было равно 10 и десять потоков вызвали функцию `WaitForSingleObject()`. В результате счетчик потоков сбрасывается до нуля и тем самым семафор переводится в недоступное состояние. После этого достаточно одному из потоков вызвать функцию `ReleaseSemaphore()` и в качестве параметра `lReleaseCount` передать число 10, как семафор не просто будет снова пропускать потоки, т.е. станет доступным, но и увеличит значение своего счетчика до прежнего числа — до 10. Это мощное средство может привести к возникновению в вашем приложении трудно отслеживаемых ошибок, поэтому следует использовать его с большой осторожностью.

Для освобождения дескриптора семафора, выделенного ему с помощью функции `CreateSemaphore()`, не забудьте вызвать функцию `CloseHandle()`.

Пример многопоточного приложения

Для демонстрации использования объектов `TThread` в этом разделе приводится пример создания реального приложения, предназначенного для поиска файлов в специализированном потоке. Имя проекта — `DelSrch`, оно образовано от слов *Delphi Search*, означающих поиск файлов Delphi. Главная форма этой утилиты показана на рис. 11.6.

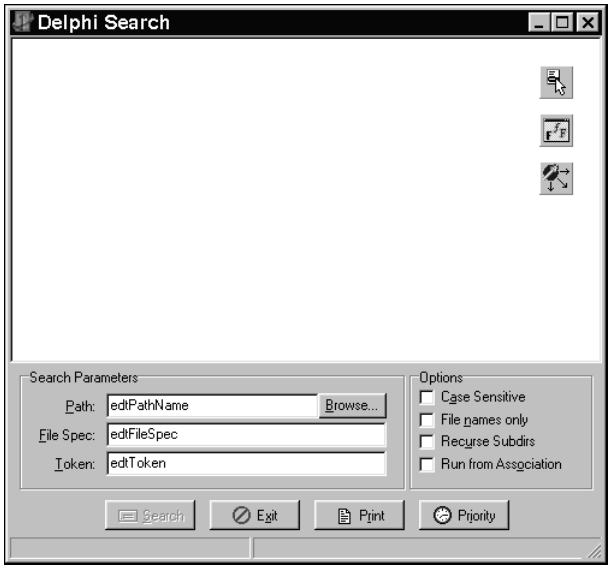


Рис. 11.6. Главная форма проекта `DelSrch`

Это приложение работает следующим образом. Пользователь выбирает путь, где следует проводить поиск, и указывает маску файла, чтобы уточнить тип искомых файлов. Кроме того, в соответствующую строку редактирования пользователь вводит лексему поиска. В форме также имеются флажки опций, с помощью которых можно указать специальные условия поиска. По щелчку на кнопке Search создается поток поиска и в объект потомка класса `TThread` передается информация, необходимая для выполнения поиска: лексема, путь и маска файла. Когда поток обнаруживает в определенных файлах искомую лексему, в окно списка добавляется соответствующая информация. Наконец, если пользователь дважды щелкнет на файле в окне списка, ему предоставляется возможность просмотреть этот файл с помощью текстового редактора или другой связанной с ним программы.

Хотя вашему вниманию предлагается полнофункциональное приложение, мы сосредоточимся в основном на реализации в нем ключевых функций поиска и их связи с многопоточностью.

Пользовательский интерфейс

Главный модуль этого приложения называется `Main.pas`, его исходный текст представлен в листинге 11.8. Основная задача этого модуля — обеспечить функционирование главной формы и ее пользовательского интерфейса. В частности, в модуле выполняются действия, необходимые для заполнения окна списка, вызова соответствующей программы для просмотра указанного пользователем файла, создания потока поиска, печати содержимого списка, а также чтения и записи установок пользовательского интерфейса в файл инициализации (`.INI`).

Листинг 11.8. Модуль `Main.pas` проекта `DelSrch`

```
unit Main;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, ExtCtrls, Menus, SrchIni, SrchU, ComCtrls;
AppEvnts;

type
  TMainForm = class(TForm)
    lbFiles: TListBox;
    StatusBar: TStatusBar;
    pnlControls: TPanel;
    PopupMenu: TPopupMenu;
    FontDialog: TFontDialog;
    pnlOptions: TPanel;
    gbParams: TGroupBox;
    LFileSpec: TLabel;
    LToken: TLabel;
    lPathName: TLabel;
    edtFileSpec: TEdit;
    edtToken: TEdit;
    btnPath: TButton;
    edtPathName: TEdit;
    gbOptions: TGroupBox;
    cbCaseSensitive: TCheckBox;
    cbFileNamesOnly: TCheckBox;
    cbRecurse: TCheckBox;
    cbRunFromAss: TCheckBox;
    pnlButtons: TPanel;
    btnSearch: TBitBtn;
    btnClose: TBitBtn;
    btnPrint: TBitBtn;
    btnPriority: TBitBtn;
    Font1: TMenuItem;
```

```
    Clear1: TMenuItem;
    Print1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    procedure btnSearchClick(Sender: TObject);
    procedure btnPathClick(Sender: TObject);
    procedure lbFilesDrawItem(Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure Font1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnPrintClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure lbFilesDblClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure btnPriorityClick(Sender: TObject);
    procedure edtTokenChange(Sender: TObject);
    procedure Clear1Click(Sender: TObject);
    procedure ApplicationEventsHint(Sender: TObject);
  private
    procedure ReadIni;
    procedure WriteIni;
  public
    Running: Boolean;
    SearchPri: Integer;
    SearchThread: TSearchThread;
    procedure EnableSearchControls(Enable: Boolean);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses Printers, ShellAPI, StrUtils, FileCtrl, PriU;

procedure PrintStrings(Strings: TStrings);
{ Эта процедура выводит на печать все строки, переданные впараметре Strings. }
var
  Prn: TextFile;
  I: Integer;
begin
  if Strings.Count = 0 then // Есть ли строки для печати?
    raise Exception.Create('No text to print!');
  AssignPrn(Prn);           // Назначаем Prn принтеру
  try
    Rewrite(Prn);           // Открываем принтер
    try
      for I := 0 to Strings.Count - 1 do   // Выполняем цикл по всем строкам
```

```
        WriteLn(Prn, Strings.Strings[I]);    // Выводим на принтер
    finally
      CloseFile(Prn);                       // Закрываем принтер
    end;
  except
    on EInOutError do
      MessageDlg('Error Printing text.', mtError, [mbOk], 0); // Ошибка печати
  end;
end;

procedure TMainForm.EnableSearchControls(Enable: Boolean);
{ Делаем доступными или недоступными определенные элементы управления,
  чтобы соответствующие опции нельзя было модифицировать во время
  выполнения поиска. }
begin
  btnSearch.Enabled := Enable;    // Разрешаем/запрещаем соответствующие
                                  // элементы управления
  cbRecurse.Enabled := Enable;
  cbFileNamesOnly.Enabled := Enable;
  cbCaseSensitive.Enabled := Enable;
  btnPath.Enabled := Enable;
  edtPathName.Enabled := Enable;
  edtFileSpec.Enabled := Enable;
  edtToken.Enabled := Enable;
  Running := not Enable;         // Устанавливаем признак Running
  edtTokenChange(nil);
  with btnClose do
  begin
    if Enable then
    begin                 // Устанавливаем реквизиты кнопки Close/Stop
      Caption := '&Close';  // Закрыть
      Hint := 'Close Application'; // Закрыть приложение
    end
    else begin
      Caption := '&Stop';  // Остановить
      Hint := 'Stop Searching';  // Остановить поиск
    end;
  end;
end;

procedure TMainForm.btnSearchClick(Sender: TObject);
{ Вызывается по щелчку на кнопке Search.  Создает поток поиска. }
begin
  EnableSearchControls(False);    // Элементы управления недоступны
  lbFiles.Clear;                   // Очищаем список
  { Начало выполнения потока }
  SearchThread := TSearchThread.Create(cbCaseSensitive.Checked,
    cbFileNamesOnly.Checked, cbRecurse.Checked, edtToken.Text,
    edtPathName.Text, edtFileSpec.Text);
end;
```

```
procedure TMainForm.edtTokenChange(Sender: TObject);
begin
  btnSearch.Enabled := not Running and (edtToken.Text <> '');
end;

procedure TMainForm.btnPathClick(Sender: TObject);
{ Вызывается по щелчку на кнопке Path; разрешает пользователю
  выбрать новый путь. }
var
  ShowDir: string;
begin
  ShowDir := edtPathName.Text;
  if SelectDirectory('Choose a search path...', '', ShowDir) then  // Выберите
путь поиска
    edtPathName.Text := ShowDir;
end;

procedure TMainForm.lbFilesDrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
{ Вызывается для заполнения окна списка. }
var
  CurStr: string;
begin
  with lbFiles do
  begin
    CurStr := Items.Strings[Index];
    Canvas.FillRect(Rect);              // Очистка прямоугольника
    if not cbFileNamesOnly.Checked then // Если не только имя файла...
      { Если текущая строка является именем файла... }
      if (Pos('File ', CurStr) = 1) and
        (CurStr[Length(CurStr)] = ':') then
        with Canvas.Font do
      begin
        Style := [fsUnderline]; // Шрифт с подчеркиванием
        Color := clRed;      // Красный цвет шрифта
      end
      else
        Rect.Left := Rect.Left + 15;      // В противном случае – отступ
    DrawText(Canvas.Handle, PChar(CurStr), Length(CurStr), Rect,
      dt_SingleLine);
  end;
end;

procedure TMainForm.Font1Click(Sender: TObject);
{ Позволяет пользователям подобрать для списка новый шрифт. }
begin
  { Выбор нового шрифта для списка. }
  if FontDialog.Execute then
    lbFiles.Font := FontDialog.Font;
end;
```

```
procedure TMainForm.FormDestroy(Sender: TObject);
{ Обработчик события OnDestroy для формы. }
begin
  WriteIni;
end;

procedure TMainForm.FormCreate(Sender: TObject);
{ Обработчик события OnCreate для формы. }
begin
  ReadIni;                          // Читаем INI-файл
end;

procedure TMainForm.btnPrintClick(Sender: TObject);
{ Вызывается по щелчку на кнопке Print. }
begin
  // Послать результаты поиска на принтер?
  if MessageDlg('Send search results to printer?', mtConfirmation,
    [mbYes, mbNo], 0) = mrYes then
    PrintStrings(lbFiles.Items);
end;

procedure TMainForm.btnCloseClick(Sender: TObject);
{ Вызывается для остановки потока или закрытия приложения. }
begin
  // Если поток выполняется, его нужно остановить
  if Running then SearchThread.Terminate
  // В противном случае закрываем приложение
  else Close;
end;

procedure TMainForm.lbFilesDblClick(Sender: TObject);
{ Вызывается по двойному щелчку пользователя в окне списка.
  Запускает программу просмотра для выделенного файла. }
var
  ProgramStr, FileStr: string;
  RetVal: THandle;
begin
  { Если пользователь щелкнул на файле.. }
  if (Pos('File ', lbFiles.Items[lbFiles.ItemIndex]) = 1) then
  begin
    { Загружаем текстовый редактор, указанный в INI-файле.
      По умолчанию загружается Notepad. }
    ProgramStr := SrchIniFile.ReadString('Defaults', 'Editor', 'notepad');
    FileStr := lbFiles.Items[lbFiles.ItemIndex]; // Получаем выбранный файл
    FileStr := Copy(FileStr, 6, Length(FileStr) - 5);  // Удаляем префикс
    if FileStr[Length(FileStr)] = ':' then        // Удаляем ":"
      DecStrLen(FileStr, 1);
    if cbRunFromAss.Checked then
      { Запускаем приложение, связанное с выбранным в списке файлом. }
      RetVal := ShellExecute(Handle, 'open', PChar(FileStr), nil, nil,
        SW_SHOWNORMAL)
```

```
    else
      { Просмотр файла с помощью текстового редактора. }
      RetVal := ShellExecute(Handle, 'open', PChar(ProgramStr),
        PChar(FileStr), nil, SW_SHOWNORMAL);
    { Проверка на  наличие ошибок. }
    if RetVal < 32 then RaiseLastWin32Error;
  end;
end;

procedure TMainForm.FormResize(Sender: TObject);
{ Обработчик события OnResize. Центрирует элементы управления в форме. }
begin
 {  Делим строку состояния на две панели в отношении 1/3 - 2/3. }
  with StatusBar do
  begin
    Panels[0].Width := Width div 3;
    Panels[1].Width := Width * 2 div 3;
  end;
end;

procedure TMainForm.btnPriorityClick(Sender: TObject);
{ Отображаем форму приоритета потока. }
begin
  ThreadPriWin.Show;
end;

procedure TMainForm.ReadIni;
{ Считывает значения из реестра, установленные по умолчанию. }
begin
  with SrchIniFile do
  begin
    edtPathName.Text := ReadString('Defaults', 'LastPath', 'C:\ ');
    edtFileSpec.Text := ReadString('Defaults', 'LastFileSpec', '*.*');
    edtToken.Text := ReadString('Defaults', 'LastToken', '');
    cbFileNamesOnly.Checked := ReadBool('Defaults', 'FNamesOnly', False);
    cbCaseSensitive.Checked := ReadBool('Defaults', 'CaseSens', False);
    cbRecurse.Checked := ReadBool('Defaults', 'Recurse', False);
    cbRunFromAss.Checked := ReadBool('Defaults', 'RunFromAss', False);
    Left := ReadInteger('Position', 'Left', 100);
    Top := ReadInteger('Position', 'Top', 50);
    Width := ReadInteger('Position', 'Width', 510);
    Height := ReadInteger('Position', 'Height', 370);
  end;
end;

procedure TMainForm.WriteIni;
{ Записывает текущие установки назад в реестр. }
begin
  with SrchIniFile do
  begin
    WriteString('Defaults', 'LastPath', edtPathName.Text);
```

```
    WriteString('Defaults', 'LastFileSpec', edtFileSpec.Text);
    WriteString('Defaults', 'LastToken', edtToken.Text);
    WriteBool('Defaults', 'CaseSens', cbCaseSensitive.Checked);
    WriteBool('Defaults', 'FNamesOnly', cbFileNamesOnly.Checked);
    WriteBool('Defaults', 'Recurse', cbRecurse.Checked);
    WriteBool('Defaults', 'RunFromAss', cbRunFromAss.Checked);
    WriteInteger('Position', 'Left', Left);
    WriteInteger('Position', 'Top', Top);
    WriteInteger('Position', 'Width', Width);
    WriteInteger('Position', 'Height', Height);
  end;
end;

procedure TMainForm.Clear1Click(Sender: TObject);
begin
  lbFiles.Items.Clear;
end;

procedure TMainForm.ApplicationEventsHint(Sender: TObject);
{ Обработчик события OnHint объекта Application }
begin
  { Отображение подсказки на панели состояния окна приложения }
  StatusBar.Panels[0].Text := Application.Hint;
end;

end.
```

Некоторые фрагменты этого модуля заслуживают особого внимания. Прежде всего стоит остановиться на маленькой процедуре `PrintStrings()`, которая используется для отправки содержимого объекта `TStrings` на принтер. Для этого процедура использует преимущества стандартной процедуры Delphi `AssignPrn()`, в которой принтеру назначается переменная типа `TextFile`, благодаря чему любой текст, записываемый в эту переменную, автоматически выводится на принтер. Завершив вывод на печать, нужно обязательно разорвать подключение к принтеру с помощью процедуры `CloseFile()`.

Интересно также использование процедуры Win32 API `ShellExecute()` для запуска программы просмотра файла, отображаемого в окне списка. Эта процедура позволяет вызывать не только выполняемые программы, но и приложения, связанные с определенными расширениями файлов. Например, если с помощью процедуры `ShellExecute()` вы попытаетесь вызвать файл с расширением `.pas`, то для просмотра этого файла будет автоматически загружена программа Delphi.

Если процедура `ShellExecute()` возвращает значение, указывающее на наличие ошибки, приложение вызывает процедуру `RaiseLastWin32Error()`, которая расположена в модуле `SysUtils`. Эта процедура, в свою очередь, для получения более подробной информации об ошибке и форматирования этой информации в строку вызывает функцию API `GetLastError()` и функцию Delphi `SysErrorMessage()`. Точно так же и вы можете использовать процедуру `RaiseLastWin32Error()` в своих приложениях, если хотите, чтобы пользователи получали подробные сообщения об ошибках, связанных со сбоями в работе интерфейса API.

Поток поиска

Реализация механизма поиска содержится в модуле SrchU.pas, который представлен в листинге 11.9. Этот модуль выполняет массу интересных вещей, включая копирование целого файла в строку, рекурсивную обработку каталогов и передачу информации в главную форму.

Листинг 11.9. Модуль SrchU.pas

```
unit SrchU;

interface

uses Classes, StdCtrls;

type
  TSearchThread = class(TThread)
  private
    LB: TListbox;
    CaseSens: Boolean;
    FileNames: Boolean;
    Recurse: Boolean;
    SearchStr: string;
    SearchPath: string;
    FileSpec: string;
    AddStr: string;
    FSearchFile: string;
    procedure AddToList;
    procedure DoSearch(const Path: string);
    procedure FindAllFiles(const Path: string);
    procedure FixControls;
    procedure ScanForStr(const FName: string; var FileStr: string);
    procedure SearchFile(const FName: string);
    procedure SetSearchFile;
  protected
    procedure Execute; override;
  public
    constructor Create(CaseS, FName, Rec: Boolean; const Str, SPath,
      FSpec: string);
    destructor Destroy; override;
  end;

implementation

uses SysUtils, StrUtils, Windows, Forms, Main;

constructor TSearchThread.Create(CaseS, FName, Rec: Boolean; const Str,
  SPath, FSpec: string);
begin
  CaseSens := CaseS;
  FileNames := FName;
```

```
  Recurse := Rec;
  SearchStr := Str;
  SearchPath := AddBackSlash(SPath);
  FileSpec := FSpec;
  inherited Create(False);
end;

destructor TSearchThread.Destroy;
begin
  FSearchFile := '';
  Synchronize(SetSearchFile);
  Synchronize(FixControls);
  inherited Destroy;
end;

procedure TSearchThread.Execute;
begin
  FreeOnTerminate := True;     // Устанавливаем все поля
  LB := MainForm.lbFiles;
  Priority := TThreadPriority(MainForm.SearchPri);
  if not CaseSens then SearchStr := UpperCase(SearchStr);
  FindAllFiles(SearchPath);     // Обрабатываем текущий каталог
  if Recurse then               // Если подкаталоги, то...
    DoSearch(SearchPath);       // Рекурсия - в противном случае...
end;

procedure TSearchThread.FixControls;
{ Делает доступными элементы управления в главной форме.
  Эту процедуру нужно вызывать с помощью метода Synchronize. }
begin
  MainForm.EnableSearchControls(True);
end;

procedure TSearchThread.SetSearchFile;
{ Обновляет имя файла в строке состояния. Эту процедуру нужно
  вызывать с помощью метода Synchronize. }
begin
  MainForm.StatusBar.Panels[1].Text := FSearchFile;
end;

procedure TSearchThread.AddToList;
{ Добавляет строку в основной список. Эту процедуру нужно
  вызывать с помощью метода Synchronize.}
begin
  LB.Items.Add(AddStr);
end;

procedure TSearchThread.ScanForStr(const FName: string;
  var FileStr: string);
{ Ищет строку SearchStr в строке FileStr файла FName. }
```

```
var
  Marker: string[1];
  FoundOnce: Boolean;
  FindPos: integer;
begin
  FindPos := Pos(SearchStr, FileStr);
  FoundOnce := False;
  while (FindPos <> 0) and not Terminated do
  begin
    if not FoundOnce then
    begin
      { Используем ":" только в случае, если пользователь не выберет
        опцию "File Names Only" (Только имена файлов). }
      if FileNames then
        Marker := ''
      else
        Marker := ':';
      { Добавляем файл в список }
      AddStr := Format('File %s%s', [FName, Marker]);
      Synchronize(AddToList);
      FoundOnce := True;
    end;
    { При поиске исключительно имен файлов (т.е. при выборе опции
      "File Names Only") ту же самую строку не ищем в том же файле. }
    if FileNames then Exit;

    { Добавляем строку, если не выбрана опция "File Names Only". }
    AddStr := GetCurLine(FileStr, FindPos);
    Synchronize(AddToList);
    FileStr := Copy(FileStr, FindPos + Length(SearchStr), Length(FileStr));
    FindPos := Pos(SearchStr, FileStr);
  end;
end;

procedure TSearchThread.SearchFile(const FName: string);
{ Ищет строку SearchStr в файле FName. }
var
  DataFile: THandle;
  FileSize: Integer;
  SearchString: string;
begin
  FSearchFile := FName;
  Synchronize(SetSearchFile);
  try
    DataFile := FileOpen(FName, fmOpenRead or fmShareDenyWrite);
    if DataFile = 0 then raise Exception.Create('');
    try
      { Устанавливаем длину искомой строки. }
      FileSize := GetFileSize(DataFile, nil);
      SetLength(SearchString, FileSize);
```

```
      { Копируем содержимое файла в строку }
      FileRead(DataFile, Pointer(SearchString)^, FileSize);
    finally
      CloseHandle(DataFile);
    end;
    if not CaseSens then SearchString := UpperCase(SearchString);
    ScanForStr(FName, SearchString);
  except
    on Exception do
    begin
      AddStr := Format('Error reading file: %s', [FName]);
      Synchronize(AddToList);
    end;
  end;
end;

procedure TSearchThread.FindAllFiles(const Path: string);
{ Процедура ищет файлы, отвечающие заданной спецификации в
  подкаталогах заданного пути. }
var
  SR: TSearchRec;
begin
  { Находим первый файл с заданной спецификацией. }
  if FindFirst(Path + FileSpec, faArchive, SR) = 0 then
    try
      repeat
        SearchFile(Path + SR.Name);             // Обрабатываем файл
      until (FindNext(SR) <> 0) or Terminated; // Находим следующий файл
    finally
      SysUtils.FindClose(SR);                   // Освобождаем память
    end;
end;

procedure TSearchThread.DoSearch(const Path: string);
{ Процедура рекурсивно обрабатывает дерево подкаталогов,
  начиная с пути Path. }
var
  SR: TSearchRec;
begin
  { Ищем каталоги. }
  if FindFirst(Path + '*.*', faDirectory, SR) = 0 then
    try
      repeat
        { Если это каталог, а не '.' или '..', то... }
        if ((SR.Attr and faDirectory) <> 0) and (SR.Name[1] <> '.') and
          not Terminated then
        begin
          FindAllFiles(Path + SR.Name + '\ ');  // Обрабатываем каталог
          DoSearch(Path + SR.Name + '\ ');      // Рекурсия
        end;
```

```
      until (FindNext(SR) <> 0) or Terminated;  // Находим следующий
                                                // Каталог
    finally
      SysUtils.FindClose(SR);                  // Освобождаем память
    end;
end;

end.
```

При создании этого потока сначала вызывается метод `FindAllFiles()`, который использует методы `FindFirst()` и `FindNext()` для поиска в текущем каталоге всех файлов, соответствующих заданной пользователем спецификации. Если пользователь установил флажок опции Recurse subdirs (Заход в подкаталоги), то для исследования дерева подкаталогов вызывается метод `DoSearch()`. Этот метод вновь пользуется услугами методов `FindFirst()` и `FindNext()`, но на этот раз — для обнаружения каталогов, причем весь фокус заключается в том, что для операций внутри дерева подкаталогов метод `DoSearch()` выполняет рекурсию, т.е. вызывает сам себя. При обнаружении каждого каталога вызывается процедура `FindAllFiles()`, которая обрабатывает все файлы, отвечающие условиям поиска.

Алгоритм рекурсии, используемый процедурой `DoSearch()`, является стандартным методом прохода дерева каталогов. Как известно, рекурсивные алгоритмы представляют определенную трудность для отладки, и поэтому опытные программисты стараются использовать проверенные, заведомо работающие алгоритмы. Учитывая вышесказанное, рекомендуем сохранять этот метод, чтобы впоследствии можно было применить его в других приложениях.

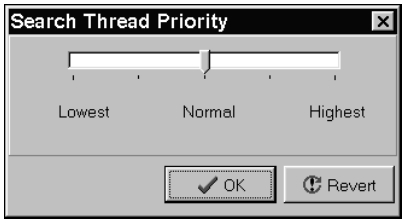
Обратите внимание, что при обработке каждого файла, включающей поиск внутри него заданной лексемы, используется объект `TMemMapFile`, который в системе Win32 инкапсулирует файл, отображенный в память. Этот объект подробно рассматривается в главе 12, “Работа с файлами”, а пока вам достаточно знать, что он предоставляет собой простой способ отображения содержимого любого файла в память. Полный алгоритм работает следующим образом.

1. При обнаружении методом `FindAllFiles()` файла, отвечающего заданной спецификации, вызывается метод `SearchFile()` и его содержимое копируется в строку.
2. Для каждой строки файла вызывается метод `ScanForStr()`, в котором внутри каждой строки ищется вхождение искомой лексемы.
3. Если обнаружено вхождение лексемы в список, отображаемый на форме, то добавляется имя файла и/или строка текста. Строка текста добавляется только в том случае, если пользователь не устанавливал флажок опции File Names Only (Только имена файлов).

Обратите внимание на то, что все методы объекта `TSearchThread` периодически проверяют состояние признаков `StopIt` (который устанавливается при остановке потока) и `Terminated` (который устанавливается при завершении работы с объектом `TThread`).

Помните, что любые методы внутри объекта `TThread`, которые используют пользовательский интерфейс приложения, должны вызываться с помощью метода `Synchronize()`. Альтернативный способ модификации пользовательского интерфейса состоит в посылке сообщений.

Настройка приоритета



Проект DelSrch позволяет пользователям динамически настраивать приоритет потока поиска. Используемая для этого форма, показана на рис. 11.7, а реализующий ее модуль PriU.pas представлен в листинге 11.10.

Рис. 11.7. Форма установки приоритета для проекта DelSrch

Листинг 11.10. Модуль PriU.pas

```
unit PriU;

interface

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

type
  TThreadPriWin = class(TForm)
    tbrPriorityTrackBar: TTrackBar;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    btnOK: TBitBtn;
    btnRevert: TBitBtn;
    Panel1: TPanel;
    procedure tbrPriorityTrackBarChange(Sender: TObject);
    procedure btnRevertClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormShow(Sender: TObject);
    procedure btnOKClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Закрытые объявления }
    OldPriVal: Integer;
  public
    { Открытые объявления }
  end;

var
  ThreadPriWin: TThreadPriWin;

implementation

{$R *.DFM}

uses Main, SrchU;
```

```
procedure TThreadPriWin.tbrPriorityTrackBarChange(Sender: TObject);
begin
  with MainForm do
  begin
    SearchPri := tbrPriorityTrackBar.Position;
    if Running then
      SearchThread.Priority := TThreadPriority(tbrPriorityTrackBar.Position);
  end;
end;

procedure TThreadPriWin.btnRevertClick(Sender: TObject);
begin
  tbrPriorityTrackBar.Position := OldPriVal;
end;

procedure TThreadPriWin.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caHide;
end;

procedure TThreadPriWin.FormShow(Sender: TObject);
begin
  OldPriVal := tbrPriorityTrackBar.Position;
end;

procedure TThreadPriWin.btnOKClick(Sender: TObject);
begin
  Close;
end;

procedure TThreadPriWin.FormCreate(Sender: TObject);
begin
  tbrPriorityTrackBarChange(Sender);  // Инициализация приоритета потока
end;

end.
```

Этот модуль довольно прост. Он служит для установки значения переменной `SearchPri`, которое бы соответствовало позиции на линейке с ползунком, находящейся на главной форме проекта. Если поток уже в работе, то его приоритет устанавливается тем же способом. Но поскольку объект `TThreadPriority` имеет перечислимый тип, то простая операция приведения типа преобразует значения от 1 до 5 (генерируемые линейкой с ползунком) в элементы перечисления объекта `TThreadPriority`.

Многопоточный доступ к базе данных

Несмотря на то что созданию приложений с доступом к базам данных уделяется немалое внимание в главе 28 второго тома, “Создание локальных приложений баз данных”, в этом разделе мы дадим вам некоторые советы, касающиеся использования многопоточности в

контексте разработки приложений, получающих информацию из баз данных. Если вы совсем не знакомы с основами программирования подобных приложений в среде Delphi, то, прежде чем читать этот раздел, вам стоит просмотреть указанную главу.

Пожалуй, самым важным требованием к разработчикам приложений с доступом к базам данных в системе Win32 является умение обеспечивать выполнение сложных запросов или хранимых процедур в фоновых потоках. К счастью, эта функция поддерживается системой доступа к базам данных фирмы Borland (Borland Database Engine, или BDE) и довольно легко реализуется в Delphi.

К запуску в фоновом потоке запроса, выполняемого, например, с помощью компонента `TQuery`, предъявляются только два требования.

- Каждый выделенный в отдельный поток запрос должен располагаться внутри собственного сеанса. Обеспечение объекта `TQuery` собственным сеансом достигается путем размещения в форме компонента `TSession` и присвоения его имени свойству `SessionName` компонента `TQuery`. Это также предполагает, что если компонентом `TQuery` используется компонент `TDatabase`, то для каждого сеанса также должен быть использован уникальный экземпляр компонента `TDatabase`.
- Компонент `TQuery` не должен быть связан с какими бы то ни было компонентами `TDataSource` в тот момент, когда запрос открывается из вторичного потока. Если же запрос присоединен к компоненту `TDataSource`, то это должно быть реализовано в контексте основного (первичного) потока. Компонент `TDataSource` используется только для подключения наборов данных к элементам управления пользовательского интерфейса, а управление пользовательским интерфейсом должно осуществляться только в основном потоке.

Для иллюстрации методов реализации фоновых запросов в собственных потоках подготовлен демонстрационный проект `BDEThrd`, главная форма которого показана на рис. 11.8. С помощью этой формы можно задать псевдоним базы данных, имя пользователя, пароль для входа в конкретную базу данных, а также ввести сам запрос. По щелчку на кнопке Go! для обработки запроса порождается вторичный поток и результаты его работы отображаются в дочерней форме проекта.

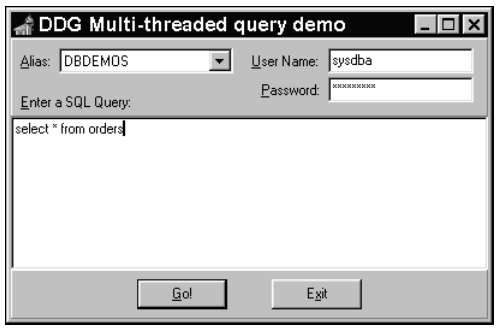


Рис. 11.8. Главная форма демонстрационного проекта `BDEThrd`

Дочерняя форма `TQueryForm` показана на рис. 11.9. Обратите внимание на то, что эта форма содержит по одному экземпляру каждого из компонентов — `TQuery`, `TDatabase`, `TSession`, `TDataSource` и `TDBGrid`. Следовательно, каждый экземпляр компонента `TQueryForm` имеет собственные экземпляры этих компонентов.

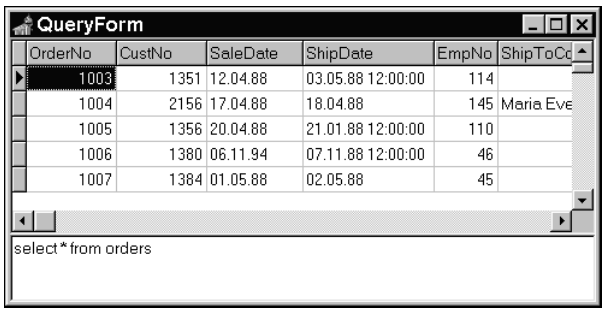


Рис. 11.9. Дочерняя форма запроса демонстрационного проекта BDEThrd

В листинге 11.11 содержится код главного модуля приложения `Main.pas`.

Листинг 11.11. Модуль `Main.pas` демонстрационного проекта `BDEThrd`

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    pnlBottom: TPanel;
    pnlButtons: TPanel;
    GoButton: TButton;
    Button1: TButton;
    memQuery: TMemo;
    pnlTop: TPanel;
    Label1: TLabel;
    AliasCombo: TComboBox;
    Label3: TLabel;
    UserNameEd: TEdit;
    Label4: TLabel;
    PasswordEd: TEdit;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure GoButtonClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
```

```
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses QryU, DB, DBTables;

var
  FQueryNum: Integer = 0;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.GoButtonClick(Sender: TObject);
begin
  Inc(FQueryNum);   // Сохранение уникального номера запроса
  { Формирование нового запроса }
  NewQuery(FQueryNum, memQuery.Lines, AliasCombo.Text, UserNameEd.Text,
    PasswordEd.Text);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  { Заполнение раскрывающегося списка с использованием псевдонимов BDE }
  Session.GetAliasNames(AliasCombo.Items);
end;

end.
```

Как видите, этот модуль невелик по размеру. В обработчике события главной формы `OnCreate` комбинированный список (компонент `TComboBox`) `AliasCombo` заполняется псевдонимами BDE с помощью метода `GetAliasNames()` класса `TSession`. Обратите внимание: когда обработчик щелчка на кнопке Go! выполняет новый запрос путем вызова процедуры `NewQuery()`, расположенной во втором модуле `QryU.pas`, этой процедуре при каждом щелчке на кнопке передается новый уникальный номер запроса `FQueryNum`. Этот номер используется для создания уникального сеанса и имени базы данных для каждого потока запроса.

В листинге 11.12 содержится код модуля `QryU.pas`.

Листинг 11.12. Модуль `QryU.pas`

```
unit QryU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Grids, DBGrids, DB, DBTables, StdCtrls;
```

```
type
  TQueryForm = class(TForm)
    Query: TQuery;
    DataSource: TDataSource;
    Session: TSession;
    Database: TDatabase;
    dbgQueryGrid: TDBGrid;
    memSQL: TMemo;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

procedure NewQuery(QryNum: integer; Qry: TStrings; const Alias, UserName,
  Password: string);

implementation

{$R *.DFM}

type
  TDBQueryThread = class(TThread)
  private
    FQuery: TQuery;
    FDataSource: TDataSource;
    FQueryException: Exception;
    procedure HookUpUI;
    procedure QueryError;
  protected
    procedure Execute; override;
  public
    constructor Create(Q: TQuery; D: TDataSource); virtual;
  end;

constructor TDBQueryThread.Create(Q: TQuery; D: TDataSource);
begin
  inherited Create(True);    // Создаем приостановленный поток
  FQuery := Q;               // Устанавливаем параметры
  FDataSource := D;
  FreeOnTerminate := True;
  Resume;                    // Да здравствует поток!
end;

procedure TDBQueryThread.Execute;
begin
  try
    FQuery.Open;            // Открываем запрос
    Synchronize(HookUpUI);  // Обновляем UI из основного потока
```

```
  except
    FQueryException := ExceptObject as Exception;
    Synchronize(QueryError);    // Отображаем исключительную
                                // ситуацию из основного потока
  end;
end;

procedure TDBQueryThread.HookUpUI;
begin
  FDataSource.DataSet := FQuery;
end;

procedure TDBQueryThread.QueryError;
begin
  Application.ShowException(FQueryException);
end;

procedure NewQuery(QryNum: integer; Qry: TStrings; const Alias, UserName,
  Password: string);
begin
  { Создаем новую форму запроса для отображения результатов запроса }
  with TQueryForm.Create(Application) do
  begin
    { Устанавливаем уникальное имя сеанса }
    Session.SessionName := Format('Sess%d', [QryNum]);
    with Database do
    begin
      { Устанавливаем уникальное имя базы данных }
      DatabaseName := Format('DB%d', [QryNum]);
      { Устанавливаем псевдоним }
      AliasName := Alias;
      { Подключаем базу данных к сеансу }
      SessionName := Session.SessionName;
      { Определяемые пользователем имя пользователя и пароль }
      Params.Values['USER NAME'] := UserName;
      Params.Values['PASSWORD'] := Password;
    end;
    with Query do
    begin
      { Подключаем запрос к базе данных и сеансу }
      DatabaseName := Database.DatabaseName;
      SessionName := Session.SessionName;
      { Устанавливаем строки запроса }
      SQL.Assign(Qry);
    end;
    { Отображаем строки запроса в окне SQL Memo }
    memSQL.Lines.Assign(Qry);
    { Отображаем форму запроса }
    Show;
    { Открываем запрос в его собственном потоке }
```

```
    TDBQueryThread.Create(Query, DataSource);
  end;
end;

procedure TQueryForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;

end.
```

В процедуре `NewQuery()` создается новый экземпляр дочерней формы `TQueryForm`, устанавливаются свойства каждого из ее компонентов доступа к данным и присваиваются уникальные имена ее компонентам `TDatabase` и `TSession`. Свойство запроса `SQL` заполняется значениями строк, переданных в качестве параметра `Qry`, после чего порождается поток запроса.

Как видите, для описания класса `TDBQueryThread` не требуется большого объема программного текста. Конструктор просто устанавливает несколько переменных экземпляра, а сам запрос открывается в методе `Execute()`, из которого вызывается процедура `HookupUI()` с помощью метода `Synchronize()` для привязки запроса к источнику данных. Следует также обратить внимание на блок `try..except` внутри процедуры `Execute()`, которая использует метод `Synchronize()` отображения сообщений, связанных с возникновением исключительных ситуаций в контексте основного потока.

Многопоточная графика

Ранее упоминалось, что компоненты библиотеки VCL не предназначены для одновременной работы с несколькими потоками, но это утверждение не совсем точное. Подпрограммы библиотеки VCL обладают особой способностью, которая позволяет многим потокам поддерживать отдельные графические объекты. Благодаря новым методам `Lock()` и `Unlock()`, введенным в класс `TCanvas`, был создан целый модуль `Graphics`, который обеспечивает поддержку многопоточности. Он включает такие классы, как `TCanvas`, `TPen`, `TBrush`, `TFont`, `TBitmap`, `TMetafile`, `TPicture` и `TIcon`.

Программная реализация всех этих методов `Lock()` имеет похожие элементы, связанные с использованием критической секции и функции Win32 API `EnterCriticalSection()` (описанной ранее в этой главе), предназначенных для защиты доступа к канве или графическим объектам. После того как конкретный поток вызовет метод `Lock()`, ему предоставляется право единолично управлять канвой или графическими объектами. Другие потоки, стремящиеся получить разрешение на вход в ту часть кода, которая следует за обращением к методу `Lock()`, будут переведены в состояние ожидания. Оно продлится до тех пор, пока поток, владеющий критическим разделом, не вызовет метод `Unlock()`. Чтобы освободить критический раздел и разрешить следующему ожидающему потоку (если таковой имеется) войти в защищенную часть кода, метод `Unlock()` вызывает, в свою очередь, функцию `LeaveCriticalSection()`. На примере следующих строк показано, как использовать эти методы для управления доступом к канве:

```
Form.Canvas.Lock;
// Здесь должен быть код, управляющий канвой
Form.Canvas.Unlock;
```

В листинге 11.13 представлен модуль Main проекта MTGraph — приложения, в котором демонстрируется доступ нескольких потоков к канве формы.

Листинг 11.13. Модуль Main.pas проекта MTGraph

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Menus;

type
  TMainForm = class(TForm)
    MainMenu1: TMainMenu;
    Options1: TMenuItem;
    AddThread: TMenuItem;
    RemoveThread: TMenuItem;
    ColorDialog1: TColorDialog;
    Add10: TMenuItem;
    RemoveAll: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure AddThreadClick(Sender: TObject);
    procedure RemoveThreadClick(Sender: TObject);
    procedure Add10Click(Sender: TObject);
    procedure RemoveAllClick(Sender: TObject);
  private
    ThreadList: TList;
  public
    { Открытые объявления }
  end;
TDrawThread = class(TThread)
  private
    FColor: TColor;
    FForm: TForm;
  public
    constructor Create(AForm: TForm; AColor: TColor);
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

{ TDrawThread }

constructor TDrawThread.Create(AForm: TForm; AColor: TColor);
```

```
begin
  FColor := AColor;
  FForm := AForm;
  inherited Create(False);
end;

procedure TDrawThread.Execute;
var
  P1, P2: TPoint;

  procedure GetRandCoords;
  var
    MaxX, MaxY: Integer;
  begin
    // Инициализируем точки P1 и P2 случайными координатами
    // в пределах границ формы
    MaxX := FForm.ClientWidth;
    MaxY := FForm.ClientHeight;
    P1.x := Random(MaxX);
    P2.x := Random(MaxX);
    P1.y := Random(MaxY);
    P2.y := Random(MaxY);
  end;

begin
  FreeOnTerminate := True;
  // Поток выполняется до тех пор, пока он или приложение не будут завершены
  while not (Terminated or Application.Terminated) do
  begin
    GetRandCoords;           // Инициализируем точки P1 и P2
    with FForm.Canvas do
    begin
      Lock;                  // Блокируем канву
      // Одновременно только один поток может выполнять следующий код:
      Pen.Color := FColor;   // Устанавливаем цвет пера
      MoveTo(P1.X, P1.Y);    // Переходим к точке канвы P1
      LineTo(P2.X, P2.Y);    // Рисуем прямую до точки P2
      // После выполнения следующей строки кода другому потоку будет
      // разрешено войти в вышестоящий блок кода
      Unlock;                // Снятие блокировки канвы
    end;
  end;
end;

{ TMainForm }

procedure TMainForm.FormCreate(Sender: TObject);
begin
  ThreadList := TList.Create;
end;
```

```
procedure TMainForm.FormDestroy(Sender: TObject);
begin
  RemoveAllClick(nil);
  ThreadList.Free;
end;

procedure TMainForm.AddThreadClick(Sender: TObject);
begin
  // Добавляем новый поток в список и разрешаем пользователю выбрать цвет
  if ColorDialog1.Execute then
    ThreadList.Add(TDrawThread.Create(Self, ColorDialog1.Color));
end;

procedure TMainForm.RemoveThreadClick(Sender: TObject);
begin
  // Завершаем последний поток в списке и удаляем его из списка
  TDrawThread(ThreadList[ThreadList.Count - 1]).Terminate;
  ThreadList.Delete(ThreadList.Count - 1);
end;

procedure TMainForm.Add10Click(Sender: TObject);
var
  i: Integer;
begin
  // Создаем 10 потоков, каждый со случайным цветом
  for i := 1 to 10 do
    ThreadList.Add(TDrawThread.Create(Self, Random(MaxInt)));
end;

procedure TMainForm.RemoveAllClick(Sender: TObject);
var
  i: Integer;
begin
  Cursor := crHourGlass;
  try
    for i := ThreadList.Count - 1 downto 0 do
    begin
      TDrawThread(ThreadList[i]).Terminate;  // Завершаем поток
      TDrawThread(ThreadList[i]).WaitFor;    // Убеждаемся, что поток завершен
    end;
    ThreadList.Clear;
  finally
    Cursor:= crDefault;
  end;
end;

initialization
  Randomize;  // Установка начального числа для генератора случайных чисел

end.
```

В этом приложении предусмотрено главное меню, содержащее четыре элемента (рис. 11.10). По команде Add Thread (Добавить поток) создается новый экземпляр компонента `TDrawThread`, который рисует прямые, случайным образом расположенные на главной форме. Эту команду можно выбирать снова и снова, вливая тем самым свежие струи в коктейль потоков, уже получивших доступ к канве главной формы. При выборе следующей команды, Remove Thread (Удалить поток), удаляется поток, который был добавлен последним. С помощью третьей команды, Add 10 (Добавить 10), создается десять новых экземпляров компонента `TDrawThread`. И, наконец, четвертая команда, Remove All (Удалить все), завершает и удаляет все экземпляры компонента `TDrawThread`. На рис. 11.10 также показан результат работы десяти потоков, которые одновременно рисуют на канве формы.

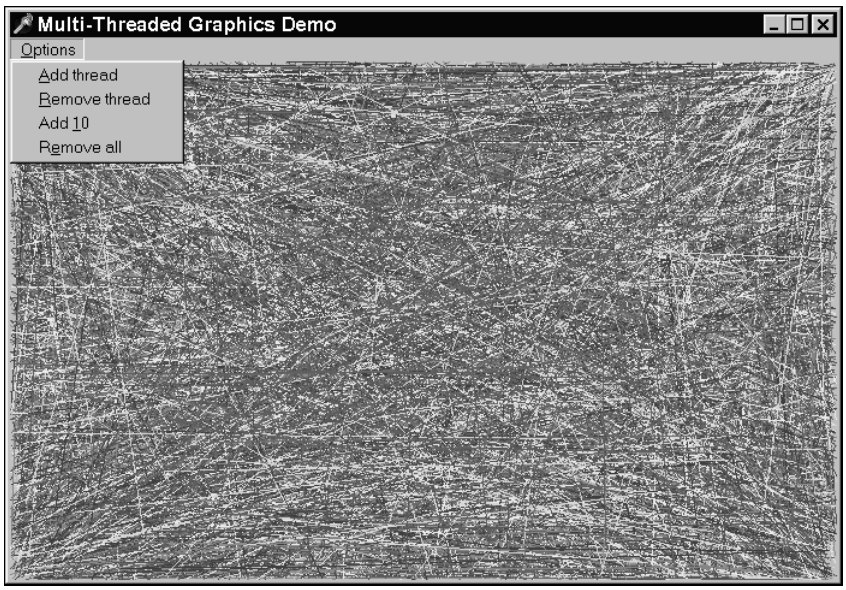


Рис. 11.10. Главная форма проекта `MTGraph`

Из правил блокировки канвы следует, что, пока каждый пользователь канвы будет блокировать ее перед рисованием и отменять блокировку по окончании рисования, потоки не будут мешать друг другу. Обратите внимание на то, что все события `OnPaint` и вызовы метода `Paint()`, инициированные VCL, автоматически блокируют и отменяют блокировку канвы вместо вас, поэтому обычный код Delphi может сосуществовать с графическими операциями новых фоновых потоков.

Используя это приложение в качестве примера, рассмотрим последовательность симптомов возможной коллизии потоков, возникающей при невыполнении блокировки канвы. Если поток 1 устанавливает красный цвет пера канвы и рисует прямую, а поток 2 устанавливает синий цвет пера канвы и рисует окружность, причем эти потоки не блокируют канву перед началом выполнения своих операций, то возможен следующий сценарий коллизии потоков. Поток 1 установит красный цвет пера. Планировщик операционной системы передаст процессорный ресурс потоку 2. Поток 2 установит синий цвет пера и нарисует окружность. Эстафета выполнения кода снова переходит к потоку 1. Поток 1 рисует прямую. Однако эта прямая оказывается не красной, а синей, поскольку потоку 2 удалось “втиснуться” со своей установкой синего цвета между операциями потока 1.

Необходимо также заметить, что для возникновения подобных проблем достаточно иметь хотя бы один поток-нарушитель. Если поток 1 будет исправно блокировать канву, а поток 2 — нет, только что описанный сценарий не изменится. Для предотвращения сценария коллизии потоков необходимо, чтобы оба потока блокировали канву на время выполнения над ней своих операций.

Резюме

В этой главе рассматривались потоки и правила работы с ними в среде Delphi. Вы познакомились с методами синхронизации нескольких потоков и научились поддерживать связь между вторичными потоками и главным потоком приложения Delphi. Кроме того, здесь были продемонстрированы примеры использования потоков в контексте реальных приложений поиска файлов и работы с базой данных. И в заключение вы узнали о возможности рисования на канве формы с использованием сразу нескольких потоков.

В следующей главе рассматриваются методы работы в Delphi с различными типами файлов.

Работа с файлами

Глава

12

Работа с файлами, каталогами и устройствами является обычной задачей программирования, с которой рано или поздно вам обязательно придется столкнуться. В этой главе рассматриваются способы работы с различными типами файлов: текстовыми, типизированными и нетипизированными. Кроме того, мы рассмотрим, как применяется для инкапсуляции процессов ввода-вывода файлов класс `TFileStream` и как воспользоваться преимуществами *файлов, отображенных в память*, — одного из самых мощных средств Win32. Вы также узнаете, как создать класс `TMemoryMappedFile`, который инкапсулирует некоторые возможности отображения в память, и научитесь использовать этот класс для выполнения поиска заданного текста в текстовых файлах. Кроме того, в этой главе демонстрируются эффективные методы определения доступных устройств, исследования деревьев каталогов для обнаружения нужных файлов и получения информации о версиях файлов. Прочитав главу, вы получите знания, которых будет вполне достаточно для успешной работы с файлами, каталогами и устройствами внешней памяти.

Файловые операции ввода-вывода

Как программисту, вам придется иметь дело с тремя типами файлов: текстовыми, типизированными и двоичными. Процедуры ввода-вывода файлов этих типов рассматриваются в нескольких последующих разделах. *Текстовые файлы*, как следует из их названия, содержат текст ASCII, который может прочитать любой текстовый редактор. *Типизированные файлы* включают данные, тип которых определяется программистом. Наконец, *двоичные файлы* охватывают все множество остальных файлов. Под этим обтекаемым названием может скрываться любой файл, содержащий данные, представленные в любом формате или вовсе неформатированные.

Работа с текстовыми файлами

В этом разделе показаны примеры манипуляции текстовыми файлами с помощью процедур и функций, встроенных в библиотеку времени исполнения (RunTime Library — RTL) Object Pascal. Прежде чем что-либо делать с текстовым файлом, его необходимо открыть, но сначала нужно объявить переменную типа `TextFile`:

```
var MyTextFile: TextFile;
```

Теперь эту переменную можно использовать для ссылки на любой текстовый файл.

Чтобы открыть текстовый файл, нужно знать о существовании двух процедур. Первая называется `AssignFile()` и связывает имя файла с файловой переменной:

```
AssignFile(MyTextFile, 'MyTextFile.txt');
```

После того как вы свяжете файловую переменную с именем файла, можно открыть сам файл. В случае текстового файла это можно сделать тремя способами. Во-первых, создать и открыть файл можно с помощью процедуры `Rewrite()`. Если применить эту процедуру к уже существующему файлу, он будет перезаписан, т.е. будет создан новый файл с тем же именем. Во-вторых, можно открыть файл с доступом “только для чтения” — для этого потребуется вызвать процедуру `Reset()`. И, в-третьих, для добавления информации в конец существующего файла вам не обойтись без процедуры `Append()`.

На заметку Процедура `Reset()` открывает с доступом "только для чтения" как типизированные, так и нетипизированные файлы.

Чтобы закрыть файл после его открытия используйте процедуру `CloseFile()`. Рассмотрим пример, в котором иллюстрируется каждая процедура.

Для открытия файла с доступом "только для чтения" используйте следующую последовательность инструкций:

```
var
  MyTextFile: TextFile;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Reset(MyTextFile);
  try
    { Работа с файлом }
  finally
    CloseFile(MyTextFile);
  end;
end;
```

Для создания нового файла выполните следующее:

```
var
  MyTextFile: TextFile;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Rewrite(MyTextFile);
  try
    { Работа с файлом }
  finally
    CloseFile(MyTextFile);
  end;
end;
```

Для добавления данных в конец существующего файла используйте такую последовательность инструкций:

```
var
  MyTextFile: TextFile;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Append(MyTextFile);
  try
    { Работа с файлом }
  finally
    CloseFile(MyTextFile);
  end;
end;
```

В листинге 12.1 показан пример использования процедуры `Rewrite()` для создания текстового файла и добавления в него пяти строк текста.

Листинг 12.1. Создание текстового файла

```
var
  MyTextFile: TextFile;
  S: String;
  i: integer;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Rewrite(MyTextFile);
  try
    for i := 1 to 5 do
    begin
      S := 'Это строка # ';
      Writeln(MyTextFile, S, i);
    end;
  finally
    CloseFile(MyTextFile);
  end;
end;
```

После выполнения этой программы файл `MyTextFile.txt` будет содержать следующий текст:

```
Это строка # 1
Это строка # 2
Это строка # 3
Это строка # 4
Это строка # 5
```

Листинг 12.2 демонстрирует, как добавить в этот же файл еще пять строк.

Листинг 12.2. Добавление строк в конец текстового файла

```
var
  MyTextFile: TextFile;
  S: String;
  i: integer;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Append(MyTextFile);
  try
    for i := 6 to 10 do
    begin
      S := 'Это строка # ';
      Writeln(MyTextFile, S, i);
    end;
  finally
    CloseFile(MyTextFile);
  end;
end;
```

Теперь содержимое этого файла выглядит следующим образом:

```
Это строка # 1
Это строка # 2
Это строка # 3
Это строка # 4
Это строка # 5
Это строка # 6
Это строка # 7
Это строка # 8
Это строка # 9
Это строка # 10
```

Обратите внимание на то, что в обоих листингах реализована возможность записи в файл как строки, так и целого. То же справедливо и для всех числовых типов, определенных в Object Pascal. Чтение информации, хранящейся в текстовом файле, можно выполнить так, как показано в листинге 12.3.

Листинг 12.3. Чтение из текстового файла

```
var
  MyTextFile: TextFile;
  S: String[12];
  i: integer;
  j: integer;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Reset(MyTextFile);
  try
    while not Eof(MyTextFile) do
    begin
      Readln(MyTextFile, S, j);
      Memo1.Lines.Add(S+IntToStr(j));
    end;
  finally
    CloseFile(MyTextFile);
  end;
end;
```

Обратите внимание, что в листинге 12.3 строковая переменная `S` объявлена как `String[12]`. Такое объявление предотвращает чтение целой строки из файла в переменную `S` — в противном случае возникла бы ошибка при попытке считать значение в целочисленную переменную `j`. Кстати говоря, этот момент иллюстрирует еще одну важную особенность ввода-вывода из текстовых файлов — возможность записи в текстовые файлы данных, разбитых по столбцам с последующим считыванием этих столбцов в строки заданной длины.

Важно иметь в виду, что ширина каждого столбца устанавливается равной определенному значению, даже если реальные строки имеют различную длину. Обратите также внимание на использование функции `Eof()`, которая выполняет проверку того, находится ли указатель в конце файла. Если да, то необходимо прервать выполнение цикла, поскольку больше не осталось текста, который можно считывать.

Чтобы проиллюстрировать чтение текстового файла, отформатированного в виде столбцов, был создан файл USCaps.txt, включающий названия столиц штатов США. Часть этого файла выглядит следующим образом:

```
Alabama             Montgomery
Alaska              Juneau
Arizona             Phoenix
Arkansas            Little Rock
California          Sacramento
Colorado            Denver
Connecticut         Hartford
Delaware            Dover
```

Столбец названия штата имеет ширину, равную 20 символам, поэтому названия столиц при распечатке выравниваются вертикально. Мы создали проект, в котором выполняется чтение этого файла и хранение списка штатов в таблице СУБД Paradox. Его исходный код представлен в листинге 12.4. Полный текст этого проекта имеется на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

На заметку

Прежде чем этот демонстрационный проект можно будет запустить, необходимо создать алиас BDE с именем DDGData. В противном случае программа работать не сможет. Если была выполнена программа установки всего комплекта примеров, приведенных в этой книге, которая также присутствует на прилагаемом компакт-диске, то требуемый алиас был ею создан автоматически.

Листинг 12.4. Исходный код проекта Capitals.dpr

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Grids, DBGrids, DB, DBTables;

type

  TMainForm = class(TForm)
    btnReadCapitals: TButton;
    tblCapitals: TTable;
    dsCapitals: TDataSource;
    dbgCapitals: TDBGrid;
    procedure btnReadCapitalsClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
```

```
procedure TMainForm.btnReadCapitalsClick(Sender: TObject);
var
  F: TextFile;
  StateName: String[20];
  CapitalName: String[20];
begin
  tblCapitals.Open;
  // Назначаем файловую переменную текстовому файлу,
  // отформатированному в виде столбцов
  AssignFile(F, 'USCAPS.TXT');
  // Открываем файл с доступом "только для чтения".
  Reset(F);
  try
    while not Eof(F) do
    begin
      { Читаем строку файла в две строки, причем длина каждой из этих строк
        совпадает с числом символов, составляющих ширину требуемого столбца. }
      Readln(F, StateName, CapitalName);
      //Теперь сохраняем обе строки в отдельных столбцах Paradox-таблицы
      tblCapitals.Insert;
      tblCapitals['State_Name']    := StateName;
      tblCapitals['State_Capital'] := CapitalName;
      tblCapitals.Post;
    end;
  finally
    CloseFile(F); // По окончании чтения закрываем файл
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Очистка таблицы при запуске проекта
  tblCapitals.EmptyTable;
end;

end.
```

Хотя работа с базами данных нами еще не рассматривалось, текст приведенного выше проекта понять несложно, поскольку он чрезвычайно прост. На этот пример стоит обратить внимание, так как подобная обработка текстовых файлов может эффективно использоваться для различных целей. Этот текстовый файл с таким же успехом мог бы содержать информацию, например, о банковских счетах, загруженную через компьютерную сеть.

Работа с типизированными файлами (файлами записей)

Структуры данных Object Pascal можно хранить в дисковых файлах, а затем считывать файловую информацию прямо в структуры данных. Это позволяет применять типизированные файлы для хранения и чтения информации, подобно записям в таблице.

Файлы, в которых хранятся структуры данных Object Pascal, называют *файлами записей*. Проиллюстрируем использование таких файлов с помощью записи, имеющей следующую структуру:

```
TPersonRec = packed record
  FirstName: String[20];
  LastName: String[20];
  MI: String[1];
  BirthDay: TDateTime;
  Age: Integer;
end;
```

На заметку Записи, содержащие такие типы данных, как `AnsiString` и `variant`, а также экземпляры классов, интерфейсы или динамические массивы, не могут быть записаны в файл.

Теперь предположим, что вы хотели бы сохранить одну или несколько таких записей в файле. В предыдущих разделах было показано, как это делается в случае текстового файла. Рассмотрим, как это выполнить для файла записей, определенного следующим образом:

```
DataFile: File of TPersonRec;
```

Чтобы прочитать одну запись типа `TPersonRec`, необходима следующая последовательность инструкций:

```
var
  PersonRec: TPersonRec;
  DataFile: File of TPersonRec;
begin
  AssignFile(DataFile, 'PersonFile.dat');
  Reset(DataFile);
  try
    if not Eof(DataFile) then
      read(DataFile, PersonRec);
  finally
    CloseFile(DataFile);
  end;
end;
```

Следующий фрагмент иллюстрирует, как добавить в этот файл одну запись:

```
var
  PersonRec: TPersonRec;
  DataFile: File of TPersonRec;
begin
  AssignFile(DataFile, 'PersonFile.dat');
  Reset(DataFile);
  Seek(DataFile, FileSize(DataFile));
  try
    write(DataFile, PersonRec);
```

```
  finally
    CloseFile(DataFile);
  end;
end;
```

Обратите внимание на использование процедуры `Seek()` для перемещения в конец файла перед внесением записи в файл. Эта функция достаточно хорошо описана в интерактивной справочной системе Delphi, поэтому здесь мы не будем углубляться в подробности ее применения.

Для иллюстрации использования типизированных файлов мы создали маленькое приложение, которое хранит информацию о персонале в формате Object Pascal. Это приложение позволяет просматривать, добавлять и редактировать записи. Мы также проиллюстрируем использование потомка класса `TFileStream` для инкапсуляции файловых операций ввода-вывода таких записей.

Определение потомка класса `TFileStream` для выполнения операций ввода-вывода с типизированными файлами

Класс `TFileStream` представляет собой класс потоков данных, который можно использовать для хранения элементов, не являющихся объектами. Структуры записей не обладают методами, позволяющими им сохраняться на диске или в памяти. Одно из решений состоит в том, чтобы сделать запись объектом, а затем к этом объекту присоединить функции хранения. Другое решение заключается в использовании для запоминания записей функций класса `TFileStream`. В листинге 12.5 представлен текст модуля, в котором определяется запись `TPersonRec`, и класс `TRecordStream`, который является производным от класса `TFileStream` и выполняет файловые операции ввода-вывода для запоминания и чтения записей.

На заметку Более подробно работа с потоками данных (stream) рассматривается в главе 22 второго тома, "Сложные методики работы с компонентами".

Листинг 12.5. Исходный код модуля `PersRec.PAS`

```
unit persrec;

interface
uses Classes, dialogs, sysutils;

type

  // Определяем запись для хранения информации о персонале
  TPersonRec = packed record
    FirstName: String[20];
    LastName: String[20];
    MI: String[1];
    BirthDay: TDateTime;
    Age: Integer;
  end;
```

```
  // Создаем потомок класса TFileStream для работы с записью TPersonRec

  TRecordStream = class(TFileStream)
  private
    function GetNumRecs: Longint;
    function GetCurRec: Longint;
    procedure SetCurRec(RecNo: Longint);
  protected
    function GetRecSize: Longint; virtual;
  public
    function SeekRec(RecNo: Longint; Origin: Word): Longint;
    function WriteRec(const Rec): Longint;
    function AppendRec(const Rec): Longint;
    function ReadRec(var Rec): Longint;
    procedure First;
    procedure Last;
    procedure NextRec;
    procedure PreviousRec;
    // Свойство NumRecs отображает число записей в потоке данных
    property NumRecs: Longint read GetNumRecs;
    // Свойство CurRec отражает текущую запись в потоке данных
    property CurRec: Longint read GetCurRec write SetCurRec;
  end;

implementation

function TRecordStream.GetRecSize:Longint;
begin
  { Эта функция возвращает размер записи, с которой работает
    данный поток данных, т.е. записи TPersonRec. }
  Result := SizeOf(TPersonRec);
end;

function TRecordStream.GetNumRecs: Longint;
begin
  // Эта функция возвращает число записей в потоке данных
  Result := Size div GetRecSize;
end;

function TRecordStream.GetCurRec: Longint;
begin
  { Эта функция возвращает позицию текущей записи. К этому значению
    нужно добавлять единицу, поскольку указатель файла всегда находится
    в начале записи, не отраженной в выражении Position div GetRecSize }
  Result := (Position div GetRecSize) + 1;
end;

procedure TRecordStream.SetCurRec(RecNo: Longint);
begin
  { Эта процедура устанавливает позицию для записи в потоке,
    заданную значением переменной RecNo. }
```

```
  if RecNo > 0 then
    Position := (RecNo - 1) * GetRecSize
  else
    // Нельзя выйти за начало файла
    Raise Exception.Create('Cannot go beyond beginning of file.');
end;

function TRecordStream.SeekRec(RecNo: Longint; Origin: Word): Longint;
begin
  { Эта функция перемещает указатель файла в позицию,
    заданную переменной RecNo. }

  { ЗАМЕЧАНИЕ: Этот метод не содержит обработки ошибки, возникающей
    при попытке выхода за пределы начала/конца файла потоков данных. }
  Result := Seek(RecNo * GetRecSize, Origin);
end;

function TRecordStream.WriteRec(Const Rec): Longint;
begin
  // Эта функция записывает запись Rec в поток
  Result := Write(Rec, GetRecSize);
end;

function TRecordStream.AppendRec(Const Rec): Longint;
begin
  // Эта функция записывает запись Rec в поток
  Seek(0, 2);
  Result := Write(Rec, GetRecSize);
end;

function TRecordStream.ReadRec(var Rec): Longint;
begin
  { Эта функция считывает запись Rec из потока и перемещает
    указатель назад в начало этой записи. }
  Result := Read(Rec, GetRecSize);
  Seek(-GetRecSize, 1);
end;

procedure TRecordStream.First;
begin
  { Эта функция перемещает указатель файла в начало потока. }
  Seek(0, 0);
end;

procedure TRecordStream.Last;
begin
  // Эта функция перемещает указатель файла в конец потока
  Seek(0, 2);
  Seek(-GetRecSize, 1);
end;
```

```
procedure TRecordStream.NextRec;
begin
  { Эта процедура перемещает указатель файла на следующую запись. }

  { Переходим к следующей записи, если при этом
    не происходит выход за конец файла. }
  if ((Position + GetRecSize) div GetRecSize) = GetNumRecs then
    // Нельзя читать за концом файла
    raise Exception.Create('Cannot read beyond end of file')
  else
    Seek(GetRecSize, 1);
end;

procedure TRecordStream.PreviousRec;
begin
{ Эта процедура перемещает указатель файла к предыдущей записи в потоке данных. }

  { Вызываем эту функцию, если не происходит выхода за начало файла. }
  if (Position - GetRecSize >= 0) then
    Seek(-GetRecSize, 1)
  else
    // Нельзя читать за началом файла
    Raise Exception.Create('Cannot read beyond beginning of the  file.');
end;

end.
```

В этом модуле сначала объявляется запись TPersonRec, которая подлежит сохранению. Класс TRecordStream является потомком класса TFileStream и используется для выполнения файловых операций ввода-вывода для записи TPersonRec. Класс TRecordStream имеет два свойства: NumRecs, которое сохраняет число записей в потоке данных, и CurRec, которое указывает на текущую запись в этом потоке.

С помощью метода GetNumRecs(), который является методом доступа к свойству NumRecs, определяется количество записей, существующих в потоке данных. Значение, возвращаемое функцией GetNumRecs(), образуется путем деления общего размера потока в байтах, полученного из свойства TStream.Size, на размер записи TPersonRec. Следовательно, если исходить из того, что длина записи TPersonRec составляет 56 байт, то в потоке длиной 162 байта должно быть четыре записи. Заметьте, однако, что точный размер записи в 56 байт достигается только в случае, если она упакована. Оказывается, что элементы таких структурированных типов, как записи и массивы, в целях более быстрого доступа к ним выравниваются по границам слов или двойных слов. Это значит, что запись занимает больше памяти, чем ей нужно на самом деле. При использовании перед объявлением записи зарезервированного слова packed применяется точный способ хранения данных без выравнивания. Если ключевое слово packed отсутствует, результаты функции GetNumRecs() могут быть неточными.

С помощью метода GetCurRec() определяется, какая запись является текущей. Это делается путем деления свойства TStream.Position на размер свойства TPersonRec и прибавления к частному значения 1. Метод SetCurRec() размещает указатель файла в такой позиции потока данных, которая соответствует началу записи, заданной свойством RecNo.

Метод SeekRec() позволяет источнику вызова разместить указатель файла в позиции, определяемой параметрами RecNo и Origin. В зависимости от значения свойства Origin указатель файла можно перемещать в потоке вперед или назад, отталкиваясь от начальной, конечной или текущей позиции указателя. Эти перемещения совершаются благодаря использованию метода Seek() объекта TStream. Использование метода TStream.Seek() описано в интерактивном справочном файле "Component Writers Guide".

Метод WriteRec() записывает содержимое параметра TPersonRec в файл, начиная с текущей позиции, которая будет принадлежать существующей записи, поэтому последняя будет перезаписана содержимым параметра TPersonRec.

Метод AppendRec() добавляет новую запись в конец файла.

Метод ReadRec() считывает данные из потока в параметр TPersonRec, а затем перемещает указатель файла в начало этой записи с помощью метода Seek(). Это делается с целью использования класса TRecordStream в характерном для баз данных стиле, в соответствии с которым указатель файла должен всегда находиться в начале текущей записи, и тогда эта запись будет в поле зрения системы.

Методы First() и Last() размещают указатель в начале и в конце файла соответственно.

Метод NextRec() помещает указатель файла в начало следующей записи при условии, если он не находится в позиции, принадлежащей последней записи файла.

Метод PreviousRec() помещает указатель файла в начало предыдущей записи при условии, если он не находится в позиции, принадлежащей первой записи файла.

Использование потомка класса TFileStream для файловых операций ввода-вывода

В листинге 12.6 содержится исходный текст, обеспечивающий функционирование главной формы приложения. Именно здесь используется объект TRecordStream.

Листинг 12.6. Исходный код главной формы проекта FileOfRec.dpr

```
unit MainFrm;

interface

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

const
  // Объявляем имя файла как константу
  FName = 'PERSONS.DAT';

type

  TMainForm = class(TForm)
    edtFirstName: TEdit;
    edtLastName: TEdit;
    edtMI: TEdit;
    meAge: TMaskEdit;
    lblFirstName: TLabel;
```

```
    lblLastName: TLabel;
    lblMI: TLabel;
    lblBirthDate: TLabel;
    lblAge: TLabel;
    btnFirst: TButton;
    btnNext: TButton;
    btnPrev: TButton;
    btnLast: TButton;
    btnAppend: TButton;
    btnUpdate: TButton;
    btnClear: TButton;
    lblRecNoCap: TLabel;
    lblRecNo: TLabel;
    lblNumRecsCap: TLabel;
    lblNoRecs: TLabel;
    dtpBirthDay: TDateTimePicker;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure btnAppendClick(Sender: TObject);
    procedure btnUpdateClick(Sender: TObject);
    procedure btnFirstClick(Sender: TObject);
    procedure btnNextClick(Sender: TObject);
    procedure btnLastClick(Sender: TObject);
    procedure btnPrevClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
  public
    PersonRec: TPersonRec;
    RecordStream: TRecordStream;
    procedure ShowCurrentRecord;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
begin
  { Если файла не существует, создаем его. В противном случае
    открываем его для чтения и записи. Это реализуется путем
    создания экземпляра класса TRecordStream. }
  if  FileExists(FName) then
    RecordStream := TRecordStream.Create(FName, fmOpenReadWrite)
  else
    RecordStream := TRecordStream.Create(FName, fmCreate);
end;
```

```
procedure TMainForm.FormDestroy(Sender: TObject);
begin
  RecordStream.Free; // Освобождаем экземпляр класса TRecordStream
end;

procedure TMainForm.ShowCurrentRecord;
begin
  // Читаем текущую запись
  RecordStream.ReadRec(PersonRec);
  // Копируем данные из параметра PersonRec в элементы управления формы
  with PersonRec do
  begin
    edtFirstName.Text := FirstName;
    edtLastName.Text  := LastName;
    edtMI.Text        := MI;
    dtpBirthDay.Date  := BirthDay;
    meAge.Text        := IntToStr(Age);
  end;
  // Отображаем на главной форме номер записи и общее число записей
  lblRecNo.Caption  := IntToStr(RecordStream.CurRec);
  lblNoRecs.Caption := IntToStr(RecordStream.NumRecs);
end;

procedure TMainForm.FormShow(Sender: TObject);
begin
  // Отображаем текущую запись, если она существует
  if RecordStream.NumRecs <> 0 then
     ShowCurrentRecord;
end;

procedure TMainForm.btnAppendClick (Sender: TObject);
begin
  // Копируем содержимое элементов управления формы в запись PersonRec
  with PersonRec do
  begin
    FirstName := edtFirstName.Text;
    LastName  := edtLastName.Text;
    MI        := edtMI.Text;
    BirthDay  := dtpBirthDay.Date;
    Age       := StrToInt(meAge.Text);
  end;
  // Записываем новую запись в поток
  RecordStream.AppendRec(PersonRec);
  // Отображаем текущую запись
  ShowCurrentRecord;
end;

procedure TMainForm.btnUpdateClick(Sender: TObject);
begin
  { Копируем содержимое элементов управления формы в запись
```

```
    PersonRec и записываем ее в поток данных. }
  with PersonRec do
  begin
    FirstName := edtFirstName.Text;
    LastName  := edtLastName.Text;
    MI        := edtMI.Text;
    BirthDay  := dtpBirthDay.Date;
    Age       := StrToInt(meAge.Text);
  end;
  RecordStream.WriteRec(PersonRec);
end;

procedure TMainForm.btnFirstClick(Sender: TObject);
begin
  { Переходим на первую запись в потоке и отображаем ее,
    если в потоке существуют какие-либо записи }
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.First;
    ShowCurrentRecord;
  end;
end;

procedure TMainForm.btnNextClick(Sender: TObject);
begin
  // Переходим на следующую запись в потоке, если в нем вообще есть записи
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.NextRec;
    ShowCurrentRecord;
  end;
end;

procedure TMainForm.btnLastClick(Sender: TObject);
begin
  { Переходим на последнюю запись в потоке, если в нем вообще есть записи }
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.Last;
    ShowCurrentRecord;
  end;
end;

procedure TMainForm.btnPrevClick(Sender: TObject);
begin
  { Переходим на предыдущую запись в потоке, если в нем вообще есть записи }
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.PreviousRec;
    ShowCurrentRecord;
```

```
  end;
end;

procedure TMainForm.btnClearClick(Sender: TObject);
begin
  // Очищаем все элементы управления на форме
  edtFirstName.Text := '';
  edtLastName.Text  := '';
  edtMI.Text        := '';
  meAge.Text        := '';
end;

end.
```

На рис. 12.1 показана главная форма для данного проекта.

Главная форма содержит как поле `TPersonRec`, так и класс `TRecordStream`. В поле `TPersonRec` хранится содержимое текущей записи. Экземпляр класса `TRecordStream` создается в обработчике события формы `OnCreate`. Если заданного файла еще не существует, то выполняется его создание. В противном случае он просто открывается.

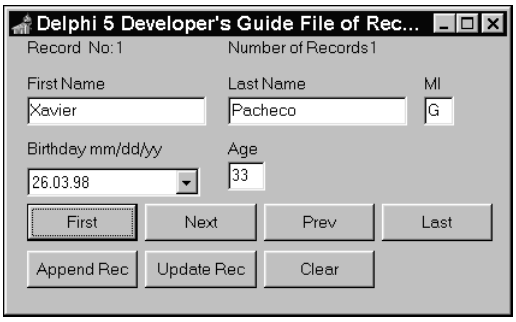


Рис. 12.1. Главная форма, демонстрирующая работу экземпляра класса `TRecordStream`

Метод `ShowCurrentRecord()` используется для извлечения текущей записи из потока данных путем вызова метода `RecordStream.ReadRec()`, который сначала считывает запись, а затем перемещает указатель файла в начало этой записи.

Работа большинства функций этого приложения описана в комментариях исходного кода. Здесь же обратим ваше внимание лишь на самые важные моменты.

Процедура `btnAppendClick()` добавляет в файл новую запись.

Метод `btnUpdateClick()` записывает содержимое элементов управления формы в файл, начиная с позиции текущей записи; таким образом модифицируется содержимое этого участка файла.

Остальные методы перемещают указатель файла на следующую, предыдущую, первую или последнюю запись в файле, что позволяет легко просматривать любые записи.

Данный пример демонстрирует, как с помощью файловых операций ввода-вывода использовать типизированные файлы для выполнения простых манипуляций, необходимых для работы с базами данных. Этот пример также иллюстрирует возможность использования объекта `TFileStream` в качестве упаковки функций ввода-вывода записей применительно к файлу.

Работа с нетипизированными файлами

До сих пор рассматривались способы манипуляции текстовыми и типизированными файлами. Текстовые файлы используются для хранения последовательностей ASCII-символов, а типизированные — для хранения данных, в которых каждый элемент отвечает определенному формату структуры записи Object Pascal. В обоих случаях каждый файл хранит некоторое количество байтов, которые могут быть соответствующим образом интерпретированы приложениями.

Однако многие файлы не связаны с использованием строгого формата. Например, RTF-файлы не только содержат текст, но и включают информацию о различных текстовых атрибутах. Для просмотра содержимого этих файлов нельзя использовать обычные текстовые редакторы, для этого требуется специальная программа, которая в состоянии интерпретировать данные, представленные в формате RTF.

Ниже иллюстрируются возможности работы с нетипизированными файлами.

Следующая строка объявляет нетипизированный файл:

```
var UntypedFile: File;
```

Этой инструкцией объявляется файл, состоящий из последовательности блоков размером в 128 байт.

Для чтения данных из нетипизированного файла используется процедура `BlockRead()`, а для записи — процедура `BlockWrite()`. Эти процедуры объявляются следующим образом:

```
procedure BlockRead(var F: File; var Buf;
  Count: Integer [; var Result: Integer]);
procedure BlockWrite(var f: File; var Buf;
  Count: Integer [; var Result: Integer]);
```

В каждую из процедур передается по три обязательных параметра. Параметр `F` представляет собой переменную нетипизированного файла. Параметр `Buf` является переменной буфера, который предназначен для хранения данных, считанных или записанных в файл. Параметр `Count` содержит количество записей, которые нужно считать из файла. Необязательный параметр `Result` содержит количество записей, действительно считанных из файла во время операции чтения. Аналогично, параметр `Result` в операции записи содержит количество записей, полностью записанных в файл. Если это значение не равно значению `Count`, возможно, на диске не хватило места.

Постараемся объяснить смысл утверждения, что эти процедуры считывают или записывают число записей, равное `Count`. Если объявить нетипизированный файл, как показано ниже, то по умолчанию этой строкой определяется файл, каждая запись которого состоит из 128 байт данных:

```
UntypedFile: File;
```

Это объявление не имеет ничего общего с любой отдельно взятой структурой записи. Оно просто задает размер блока данных, в который считывается одна запись. В листинге 12.7 показано, как считать из файла одну запись (размером 128 байт).

Листинг 12.7. Считывание из нетипизированного файла

```
var
  UnTypedFile: File;
  Buffer: array[0..128] of byte;
  NumRecsRead: Integer;
begin
```

```
  AssignFile(UnTypedFile, 'SOMEFILE.DAT');
  Reset(UnTypedFile);
  try
    BlockRead(UnTypedFile, Buffer, 1, NumRecsRead);
  finally
    CloseFile(UnTypedFile);
  end;
end;
```

В этой небольшой программе открывается файл под именем `SOMEFILE.DAT` и 128 байт данных (что соответствует одной записи или блоку) считывается в буфер под именем `Buffer`. Чтобы записать в файл 128 байт данных, воспользуйтесь вариантом программы, предложенным в листинге 12.8.

Листинг 12.8. Запись данных в нетипизированный файл

```
var
  UnTypedFile: File;
  Buffer: array[0..128] of byte;
  NumRecsWritten: Integer;
begin
  AssignFile(UnTypedFile, 'SOMEFILE.DAT');
  // Если файл не существует, создаем его. В противном случае
  // просто открываем его для чтения и записи
  if FileExists('SOMEFILE.DAT') then
    Reset(UnTypedFile)
  else
    Rewrite(UnTypedFile);
  try
    // Перемещаем указатель файла в конец файла
    Seek(UnTypedFile, FileSize(UnTypedFile));
    FillChar(Buffer, SizeOf(Buffer), 'Y');
    BlockWrite(UnTypedFile, Buffer, 1, NumRecsWritten);
  finally
    CloseFile(UnTypedFile);
  end;
end;
```

Если используется стандартный размер блока (128 байт), то при считывании из нетипизированного файла может возникнуть проблема, связанная с тем, что размер этого файла должен быть кратным 128. В противном случае будет предпринята попытка чтения за пределами конца файла. Эту проблему можно легко обойти, задав с помощью процедуры `Reset()` размер записи равным одному байту. Если передать размер записи, равный одному байту, то при чтении блоков любого размера общий размер файла будет всегда кратным одному байту. В листинге 12.9 представлена простая программа копирования файлов с помощью процедур `BlockRead()` и `BlockWrite()`.

Листинг 12.9. Демонстрационная программа копирования файлов

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    prbCopy: TProgressBar;
    btnCopy: TButton;
    procedure btnCopyClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnCopyClick(Sender: TObject);
var
  SrcFile, DestFile: File;
  BytesRead, BytesWritten, TotalRead: Integer;
  Buffer: array[1..500] of byte;
  FSize: Integer;
begin
  { Назначим файлам источника и приемника
    соответствующие файловые переменные. }
  AssignFile(SrcFile, 'srcfile.tst');
  AssignFile(DestFile, 'destfile.tst');
  // Откроем файл источника с доступом "для чтения"
  Reset(SrcFile, 1);
  try
    // Откроем файл приемника с доступом "для записи"
    Rewrite(DestFile, 1);
    try
      { Поместим процесс копирования в блок try..except,
        чтобы в случае ошибки можно было удалить этот файл. }
      try
        // Сначала установим общее число прочитанных байтов равным нулю
        TotalRead := 0;
        // Получаем размер файла источника
        FSize := FileSize(SrcFile);
        { Считываем SizeOf(Buffer) байтов из файла источника и
          добавляем эти байты в файл приемника. Повторяем этот
          процесс до тех пор, пока все байты не будут считаны
          из файла источника. В окне процесса отображается
```

```
        динамика выполнения операции копирования. }
      repeat
        BlockRead(SrcFile, Buffer, SizeOf(Buffer), BytesRead);
        if BytesRead > 0 then
        begin
          BlockWrite(DestFile, Buffer, BytesRead, BytesWritten);
          if BytesRead <> BytesWritten then
            raise Exception.Create('Error copying file')
          else begin
            TotalRead := TotalRead + BytesRead;
            prbCopy.Position := Trunc(TotalRead / Fsize) * 100;
            prbCopy.Update;
          end;
        end
      until BytesRead = 0;
    except
      { В случае возникновения исключительной ситуации удаляем
        файл приемника, поскольку он может быть с искажениями.
        Затем повторно генерируем исключительную ситуацию. }
      Erase(DestFile);
      raise;
    end;
  finally
    CloseFile(DestFile); // Закрываем файл приемника
  end;
finally
  CloseFile(SrcFile);   // Закрываем файл источника
end;
end;

end.
```

На заметку

Одна из демонстрационных программ, поставляемых вместе с Delphi 5, содержит несколько полезных функций, среди которых есть и функция копирования файла. Эта демонстрационная программа находится в каталоге `\DEMOS\DOS\FILMANEX\`. Ниже перечислены функции, содержащиеся в файле `FmxUtils.PAS`:

```
procedure CopyFile(const FileName, DestName: string);
procedure MoveFile(const FileName, DestName: string);
function GetFileSize(const FileName: string): longint;
function FileDateTime(const FileName: string): TDateTime;
function HasAttr(const FileName: string; Attr: Word): Boolean;
function ExecuteFile(const FileName, Params,
  DefaultDir: string; howCmd: Integer): THandle;
```

Далее в этой главе мы покажем, как можно копировать файлы и целые каталоги с помощью функции `ShFileOperation()`.

Прежде всего в демонстрационной программе открывается файл источника и создается файл приемника, в который будет скопировано содержимое файла источника. Переменные `TotalRead` и `FSize` используются для обновления окна динамики процесса `TProgressBar`, чтобы периодически отображать состояние операции копирования. Сама операция копирова-

ния выполняется внутри цикла repeat. Сначала из файла источника считывается SizeOf(Buffer) байтов, а переменная BytesRead определяет реальное количество считанных байтов. Затем делается попытка скопировать BytesRead байтов в файл приемника, в то время как количество реально записанных байтов сохраняется в переменной BytesWritten. Если во время операции копирования не возникло никакой ошибки, то в этот момент переменные BytesRead и BytesWritten должны иметь одинаковые значения. Описанный процесс продолжается до тех пор, пока не будут скопированы все байты файла. При возникновении ошибки возбуждается исключительная ситуация и файл приемника удаляется с диска.

Приложение с примером, иллюстрирующим изложенный выше материал, представляет собой проект FileCopy.dpr. Полный текст этого приложения имеется на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.) этой книги.

Структуры записей формата TTextRec и TFileRec

Большинство функций управления файлами на самом деле является функциями или прерываниями операционной системы, инкапсулированными в процедуры языка Object Pascal. Например, функция Reset() представляет собой Pascal-оболочку для функции Win32 API CreateFileA() из библиотеки DLL KERNEL32. Инкапсуляция функций Win32 API в функции и процедуры Object Pascal позволяет не учитывать детали реализации соответствующих файловых операций в системе. Однако остается неясным, как в случае необходимости получить доступ к определенным файловым атрибутам (например, к дескриптору файла), которые скрыты для пользователей Object Pascal.

При использовании несобственных функций Object Pascal, требующих указания дескриптора файла (например, функция LZCopy()), последний можно получить с помощью операций приведения типа переменных текстового или двоичного файлов в тип данных TTextRec или TFileRec соответственно. Записи указанных типов содержат дескриптор файла и другие файловые атрибуты. Но, кроме дескриптора файла, вам вряд ли придется когда-либо получать доступ к другим полям данных этих записей. Процедура получения дескриптора файла выглядит следующим образом:

```
TFileRec(MyFileVar).Handle
```

Определение записи TTextRec имеет вид:

```
PTextBuf = ^TTextBuf;
TTextBuf = array[0..127] of Char; // Определение буфера для первых
                                  // 127 символов в файле
TTextRec = record
    Handle: Integer;              // Дескриптор файла
    Mode: Integer;                // Режим файла
    BufSize: Cardinal;            // Следующих 4 параметра
    BufPos: Cardinal;             // используются для буферизации памяти
    BufEnd: Cardinal;
    BufPtr: PChar;
    OpenFunc: Pointer;             // Поля XXXXFunc являются указателями на
    InOutFunc: Pointer;            // функции доступа к файлу. Их можно
```

```
    FlushFunc: Pointer;            // модифицировать при написании
    CloseFunc: Pointer;            // файловых драйверов устройств
    UserData: array[1..32] of Byte;// Не используется
    Name: array[0..259] of Char;   // Полное имя файла
    Buffer: TTextBuf;              // Буфер, содержащий
                                   // первых 127 символов файла
  end;
```

Определение записи TFileRec выглядит следующим образом:

```
TFileRec = record
  Handle: Integer;                 // Дескриптор файла
  Mode: Integer;                   // Режим файла
  RecSize: Cardinal;               // Размер каждой записи файла
  Private: array[1..28] of Byte;   // Используется для внутренних целей
                                   // Object Pascal
  UserData: array[1..32] of Byte;  // Не используется
  Name: array[0..259] of Char;     // Полное имя файла
end;
```

Работа с файлами, отображенными в память

Очевидно, что одной из наиболее удобных особенностей Win32 является возможность получать доступ к файлам на диске так, как если бы их содержимое размещалось в памяти. Эта возможность реализуется с помощью функций отображения файлов в память.

С помощью отображения файла в память можно избежать необходимости выполнения для него всех файловых операций ввода-вывода. Вместо этого резервируется некоторая область виртуального адресного пространства, а затем физическая память файла на диске связывается с адресом этого зарезервированного пространства памяти. После этого можно ссылаться на содержимое файла с помощью указателя, действующего в данной зарезервированной области. Немного позже мы покажем, как использовать эту возможность для создания эффективной утилиты поиска текста в текстовых файлах, которая своей простотой обязана именно применению файлов, отображенных в память.

Использование средств отображения файлов в память

Функции отображения файлов в память имеют три области применения.

- Система Win32 загружает и выполняет EXE- и DLL-файлы с помощью механизма отображения файлов в память. Это позволяет уменьшить объем файла страничного обмена и, следовательно, сократить время загрузки данных файлов.
- Имеется возможность организовать доступ к содержимому отображенных в память файлов с помощью обычного указателя на адреса памяти в области отображения. Это не только упрощает доступ к данным файла, но и освобождает разработчика от необходимости применять различные схемы буферизации файлов.
- Отображенные в память файлы позволяют организовать разделение их данных между различными процессами, выполняющимися на одном и том же компьютере.

В этой главе мы не будем касаться первой области применения отображенных в память файлов, поскольку это затрагивает поведение системы в целом. Детально рассмотрена будет лишь вторая область применения подобных файлов, так как именно она, вероятнее всего, будет интересовать вас как разработчика. Что касается третьей области применения, то способы разделения данных между различными процессами с помощью отображенных в память файлов обсуждались нами в главе 9, "Динамически компонуемые библиотеки". Прочитав этот раздел, вы можете вернуться к указанной главе, чтобы лучше разобраться в приведенных примерах.

Доступ к содержимому отображенных в память файлов

При создании отображенного в память файла последний, в сущности, просто связывается с некоторой областью адресного пространства виртуальной памяти процесса. Для организации этой связи необходимо создать *объект отображения файла* (file-mapping object). Для просмотра или редактирования содержимого файла необходимо организовать специальное окно просмотра (file view), позволяющее получить доступ к содержимому файла с помощью указателя, аналогичного обычным указателям, используемым для доступа к некоторой области памяти.

При записи в это окно просмотра система выполняет кэширование, буферизацию и загрузку данных файла, а также управляет выделением и освобождением памяти, и вам остается только отредактировать данные, расположенные в некоторой области памяти. Выполнение всех файловых операции ввода-вывода система полностью берет на себя. В этом и заключается вся прелесть использования отображенных в память файлов — задача управления файлом решается значительно проще, чем стандартные задачи ввода-вывода, рассмотренными ранее. Кроме того, определенный выигрыш достигается и в скорости выполнения этих операций.

В следующих разделах описана последовательность действий, необходимая для создания и открытия файла, отображенного в память.

Создание или открытие файла

Первый шаг на пути создания или открытия отображенного в память файла состоит в получении дескриптора для файла, подлежащего отображению. Для этого можно воспользоваться функцией `FileCreate()` или `FileOpen()`. Определение функции `FileCreate()` содержится в модуле `SysUtils.pas` и выглядит следующим образом:

```
function FileCreate(const FileName: string): Integer;
```

Эта функция создает новый файл под именем, заданным строковым параметром `FileName`. При успешном выполнении этой функции возвращается действительное значение дескриптора файла. В противном случае возвращается значение, определенное константой `INVALID_HANDLE_VALUE`.

Функция `FileOpen()` открывает существующий файл, используя заданный режим доступа. При успешном завершении эта функция вернет действительный дескриптор файла. В противном случае будет возвращено значение, определенное константой `INVALID_HANDLE_VALUE`. Определение функции `FileOpen()` находится в модуле `SysUtils.pas` и выглядит следующим образом:

```
function FileOpen(const FileName: string; Mode: Word): Integer;
```

Первый параметр представляет собой полный путь к файлу, к которому применяется операция отображения в память. В качестве второго параметра можно передать один из возможных режимов доступа, описанных в табл. 12.1.

Таблица 12.1. Режимы доступа к файлу (fmOpenXXXX)

Режим доступа	Значение
fmOpenRead	Позволяет выполнять только чтение из файла
fmOpenWrite	Позволяет выполнять только запись в файл
fmOpenReadWrite	Позволяет выполнять и чтение, и запись в файл

Если в качестве параметра Mode передать значение 0, то будет невозможно ни чтение, ни запись данных в файл. Нулевое значение можно использовать только для получения различных атрибутов файла. Характер совместного использования заданного файла различными приложениями можно определить путем применения поразрядной операции OR (ИЛИ), используя режимы доступа, перечисленные в табл. 12.1, вместе с одним из режимов fmShareXXXX, описанных в табл. 12.2.

Таблица 12.2. Режимы совместного доступа fmShareXXXX

Режим разделения	Значение
fmShareCompat	Механизм разделения файла совместим с блоками управления DOS 1.x и 2.x. Это значение используется, вместе с другими режимами fmShareXXXX
fmShareExclusive	Разделение не разрешается
fmShareDenyWrite	Попытки других программ открыть файл с доступом fmOpenWrite будут неуспешными
fmShareDenyRead	Попытки других программ открыть файл с доступом fmOpenRead будут неуспешными
fmShareDenyNone	Попытки других программ открыть файл с любым доступом будут успешными

Получив действительный дескриптор файла, можно создать для него объект отображения в память.

Создание объекта отображения в память

Для создания *именованных* или *неименованных* объектов отображения файла используйте функцию CreateFileMapping(), которая определяется следующим образом:

```
function CreateFileMapping(
    hFile: THandle;
    lpFileMappingAttributes: PSecurityAttributes;
    flProtect, dwMaximumSizeHigh, dwMaximumSizeLow: DWORD;
    lpName: PChar) : THandle;
```

С помощью передаваемых функции CreateFileMapping() параметров системе предоставляется информация, необходимая для создания объекта отображения файла. Первый параметр hFile является дескриптором файла, полученным от предыдущего обращения к функции FileOpen() или

FileCreate(). Важно отметить, что этот файл открывается с признаками защиты, совместимыми с параметром flProtect, который будет рассмотрен чуть ниже. В главе 9, "Динамически компонуемые библиотеки", описан метод создания условий для разделения данных между отдельными процессами: в этом случае также используется функция CreateFileMapping() для создания объекта отображения файла, поддерживаемого системным файлом страничного обмена.

Параметр lpFileMappingAttributes является указателем типа PSecurityAttributes, который ссылается на атрибуты защиты для объекта отображения файла. Этот параметр практически всегда равен нулю.

Параметр flProtect задает тип защиты, применяемый к окну просмотра файла. Как упоминалось ранее, это значение должно быть совместимо с атрибутами, с которыми файл был открыт для получения его дескриптора. В табл. 12.3 перечислены различные атрибуты, которые могут быть присвоены параметру flProtect.

Таблица 12.3. Атрибуты защиты

Атрибут защиты	Значение
PAGE_READONLY	Содержимое этого файла можно читать. Файл должен быть создан с помощью функции FileCreate() или открыт путем вызова функции FileOpen() с режимом доступа fmOpenRead
PAGE_READWRITE	К этому файлу разрешен доступ "для чтения и записи". Файл должен быть открыт с режимом доступа fmOpenReadWrite
PAGE_WRITECOPY	К этому файлу разрешен доступ "для чтения и записи", однако при записи в файл создается закрытая копия модифицированной страницы. Смысл этой копии состоит в том, что файлы, отображенные в память, которые разделяются между процессами, не потребляют двойной объем ресурсов в системной памяти или в памяти файла подкачки. Дублированию подлежит только память, необходимая для хранения модифицированных страниц. Файл должен быть открыт с режимом доступа fmOpenWrite или fmOpenReadWrite

К параметру flProtect можно также применить атрибуты раздела с использованием поразрядного логического оператора or. Значения атрибутов описаны в табл. 12.4.

Таблица 12.4. Атрибуты раздела

Атрибут раздела	Значение
SEC_COMMIT	Выделяет для всех страниц раздела физическую память в области системной памяти или в файле подкачки. Это значение устанавливается по умолчанию
SEC_IMAGE	Информация об отображении файла и атрибутах берется из образа файла. Этот атрибут применяется только к выполняемым файлам изображений. (Примечание: данный атрибут игнорируется в среде Windows 95/98)
SEC_NOCACHE	Страницы, отображенные в память, не кэшируются, следовательно, все, что записывается в этот файл, применяется системой непосредственно к данным файла на диске. (Примечание: данный атрибут игнорируется в среде Windows 95/98)
SEC_RESERVE	Резервирует страницы раздела без выделения физической памяти

Параметр dwMaximumSizeHigh задает старших 32 разряда максимального размера объекта отображения файла. Если требуется получить доступ к файлу, размер которого не превышает 4 Гбайта, это значение следует установить равным 0.

Параметр dwMinimumSizeLow задает младших 32 разряда максимального размера объекта отображения файла. Нулевое значение этого параметра будет означать, что максимальный размер объекта отображения файла равен размеру отображаемого файла.

Параметр lpName определяет имя объекта отображения файла. Это значение может содержать любой символ, за исключением обратной косой черты (\). Если значение этого параметра совпадает с именем существующего объекта отображения файла, значит, эта функция требует доступа к тому же объекту отображения с помощью атрибутов, заданных параметром flProtect. В качестве параметра lpName допускается передача значения nil, что приводит к созданию неименованного объекта отображения файла.

При успешном выполнении функции CreateFileMapping() последняя возвращает дескриптор, связанный с объектом отображения файла. Если функции CreateFileMapping() не удалось создать заказанный объект, она вернет значение nil. Чтобы установить причину неудачи, следует вызвать функцию GetLastError(). Если окажется, что создаваемый объект отображения файла указывает на уже существующий объект подобного типа, то функция GetLastError() вернет значение ERROR_ALREADY_EXISTS.

В Windows 95/98 не применяются функции, выполняющие операции ввода-вывода на основе дескрипторов файлов, использовавшихся для создания отображений. В результате подобных действий данные в таких файлах могут оказаться несогласованными. Поэтому рекомендуется открывать файл с исключительными правами доступа (подробнее это описано ниже, в разделе “Согласованность файлов, отображенных в память”).

Получив объект отображения файла, можно выполнить отображение данных файла в адресное пространство процесса.

Отображение данных файла в адресное пространство процесса

Для отображения данных файла в адресное пространство процесса предназначена функция MapViewOfFile(), которая определяется следующим образом:

```
function MapViewOfFile(
  hFileMappingObject: THandle;
  dwDesiredAccess: DWORD;
  dwFileOffsetHigh,
  dwFileOffsetLow,
  dwNumberOfBytesToMap: DWORD): Pointer;
```

Параметр hFileMappingObject представляет собой дескриптор, связанный с открытым объектом отображения, который был создан с помощью обращения к функции CreateFileMapping() либо к функции OpenFileMapping().

Параметр dwDesiredAccess определяет способ доступа к данным файла и может иметь одно из значений, перечисленных в табл. 12.5.

Параметр dwFileOffsetHigh задает старших 32 разряда смещения файла, откуда начинается его отображение.

Параметр dwFileOffsetLow задает младших 32 разряда смещения файла, откуда начинается его отображение.

Таблица 12.5. Доступ к данным файла

Значение параметра dwDesiredAccess	Описание
FILE_MAP_WRITE	Позволяет выполнять чтение и запись данных файла. При этом в функции CreateFileMapping() должен использоваться атрибут PAGE_READ_WRITE
FILE_MAP_READ	Позволяет выполнять чтение и запись данных файла. При этом в функции CreateFileMapping() должны использоваться атрибуты PAGE_READ_WRITE или PAGE_READ
FILE_MAP_ALL_ACCESS	Предоставляет тот же доступ, который действует при использовании атрибута FILE_MAP_WRITE
FILE_MAP_COPY	Предоставляет доступ к записи с возможностью копирования. При записи в файл создается закрытая (private) копия записываемой страницы. В этом случае функция CreateFileMapping() должна использоваться с атрибутами PAGE_READ_ONLY, PAGE_READ_WRITE или PAGE_WRITE_COPY

Параметр dwNumberOfBytesToMap определяет количество байтов файла, предназначенных для отображения. Нулевое значение указывает на отображение целого файла.

При успешном выполнении функция MapViewOfFile() возвращает начальный адрес отображения, а в случае неудачи — значение nil, и тогда для определения причины ошибки необходимо вызвать функцию GetLastError().

Освобождение окна просмотра файла

С помощью функции UnmapViewOfFile() отображение файла в адресном пространстве вызывающего процесса удаляется. Эта функция определяется следующим образом:

```
function UnmapViewOfFile(lpBaseAddress: Pointer): BOOL;
```

Единственный параметр lpBaseAddress этой функции должен указывать на базовый адрес отображаемой области, который совпадает со значением, возвращаемым функцией MapViewOfFile().

По окончании работы с файлом следует непременно вызвать функцию UnmapViewOfFile(); в противном случае отображаемая область памяти не будет освобождена системой до тех пор, пока не закончится процесс.

Закрытие объекта отображения

Обращения к функциям FileOpen() и CreateFileMapping() связаны с открытием объектов ядра операционной системы, и поэтому вы сами несете ответственность за их закрытие. Это можно сделать с помощью функции CloseHandle(), которая определяется следующим образом:

```
function CloseHandle(hObject: THandle): BOOL;
```

При успешном выполнении функция CloseHandle() возвращает значение True, в противном случае — значение False, и тогда для определения причины ошибки нужно проверить результат выполнения функции GetLastError().

Пример использования файла, отображенного в память

В листинге 12.10 представлен пример использования функций работы с файлом, отображенным в память.

Листинг 12.10. Простой пример использования файла, отображенного в память

```
unit MainFrm;

interface

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

const
  FName = 'test.txt';

type

  TMainForm = class(TForm)
    btnUpperCase: TButton;
    memTextContents: TMemo;
    lblContents: TLabel;
    btnLowerCase: TButton;
    procedure btnUpperCaseClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnLowerCaseClick(Sender: TObject);
  public
    UCase: Boolean;
    procedure ChangeFileCase;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnUpperCaseClick(Sender: TObject);
begin
  UCase := True;
  ChangeFileCase;
end;

procedure TMainForm.btnLowerCaseClick(Sender: TObject);
begin
  UCase := False;
  ChangeFileCase;
end;
procedure TMainForm.FormCreate(Sender: TObject);
```

```
begin
  memTextContents.Lines.LoadFromFile(FName);
  // По умолчанию преобразуем в прописные буквы
  UCase := True;
end;

procedure TMainForm.ChangeFileCase;
var
  FFileHandle: THandle; // Дескриптор открытого файла
  FMapHandle: THandle;  // Дескриптор объекта отображения файла
  FFileSize: Integer;   // Переменная для хранения размера файла
  FData: PByte;         // Указатель на данные файла при отображении
  PData: PChar;         // Указатель для ссылки на данные файла
begin

  { Сначала получаем дескриптор файла, который подлежит отображению.
    Этот код предполагает существование такого файла. В противном случае
    для создания нового файла можно использовать функцию FileCreate(). }

  if not FileExists(FName) then
    raise Exception.Create('File does not exist.') // Файл не существует
  else
    FFileHandle := FileOpen(FName, fmOpenReadWrite);

  { В случае неудачного выполнения функции CreateFile()
    генерируется исключительная ситуация }
  if FFileHandle = INVALID_HANDLE_VALUE then
    raise Exception.Create('Failed to open or create file');
        // Файл не был открыт или создан
    try
    { Теперь получим размер файла, который будет передан другим функциям
      отображения файла. Сделаем этот размер на один байт больше, чтобы
      добавить завершающий нуль-символ в конец файла, отображенного в память.}
    FFileSize := GetFileSize(FFileHandle, Nil);

    { Получаем дескриптор объекта отображения файла. Если работа этой
      функции не увенчалась успехом, генерируем исключительную ситуацию. }
    FMapHandle := CreateFileMapping(FFileHandle, nil,
      PAGE_READWRITE, 0, FFileSize, nil);

    if FMapHandle = 0 then
      raise Exception.Create('Failed to create file mapping');
        // Не удалось создать отображение файла
  finally
    CloseHandle(FFileHandle);   // Освобождаем дескриптор файла
  end;

  try
    { Представляем объект отображения файла в окне просмотра.
      Эта функция вернет указатель на данные файла. В случае
      неуспешной работы генерируется исключительная ситуация. }
    FData := MapViewOfFile(FMapHandle, FILE_MAP_ALL_ACCESS, 0, 0, FFileSize);
```

```
    if FData = Nil then
      raise Exception.Create('Failed to map view of file');
        // Не удалось отобразить окно просмотра файла
  finally
        // Освобождаем дескриптор объекта отображения файла
    CloseHandle(FMapHandle);
  end;

  try
    { !!! Сюда следует поместить функции для работы с данными
      файла,отображенного в память. Например, следующая строка
      преобразует все символы файла в прописные буквы. }
    PData := PChar(FData);
    // Устанавливаем указатель в конец данных файла
    inc(PData, FFileSize);

    // Добавляем завершающий нуль-символ в конец данных файла
    PData^ := #0;

    // Теперь переводим все символы файла в прописные буквы
    if UCase then
      StrUpper(PChar(FData))
    else
      StrLower(PChar(FData));

  finally
    // Освобождаем отображение файла
    UnmapViewOfFile(FData);
  end;
  memTextContents.Lines.Clear;
  memTextContents.Lines.LoadFromFile(FName);
end;

end.
```

Как видно из листинга 12.10, первый этап состоит в получении дескриптора файла, подлежащего отображению в область памяти процесса. Он реализуется посредством вызова функции `FileOpen()`. Этой функции передается режим доступа к файлу, равный значению `fmOpenReadWrite`, что позволяет получить возможность чтения содержимого файла и записи в него.

Затем определяется размер файла и его последний символ устанавливается равным значению нуль-терминатора. На самом деле это будет маркер конца файла, байтовое значение которого совпадает с ограничивающим нулем. Все это делается ради ясности и простоты. Поскольку обращение с данными файла осуществляется как со строками, ограниченными завершающими нулями, наличие такого завершающего нуль-терминатора просто необходимо.

Следующий этап — получение объекта файла отображения в память путем вызова функции `CreateFileMapping()`. При неуспешном выполнении этой функции генерируется исключительная ситуация. В противном случае программа переходит к очередному этапу — представлению объекта отображения файла в окне просмотра. И теперь в случае сбоя в работе этой функции генерируется исключительная ситуация.

Далее выполняется смена регистра представления данных (перевод символов в прописные буквы). Если после выполнения этой процедуры вы просмотрели бы файл в текстовом редак-

торе, то увидели бы, что все символы этого файла преобразованы в прописные буквы. И, наконец, с помощью вызова функции `UnmapViewOfFile()` выполняется удаление объекта окна просмотра отображенного в память файла.

Согласованность файлов, отображенных в память

Система Win32 гарантирует, что несколько окон просмотра файла остаются согласованными, если они отображены с помощью одного и того же объекта отображения файла. Это значит, что если содержимое файла модифицируется в одном окне, то второе окно будет "в курсе" этих модификаций. Но имейте в виду, что это справедливо только при использовании одних и тех же объектов отображения файлов. Использование же различных объектов отображения не гарантирует согласованности нескольких окон просмотра. Следует отметить, что эта проблема существует только для файлов, которые отображаются с доступом "для записи". Окна просмотра для файлов, открываемых только для чтения, всегда согласованны. Следует также отметить, что файлы, разделяемые в сети, не сохраняют согласованности в отображениях файлов с доступом "для записи" на различных машинах.

Утилита поиска в текстовых файлах

Для иллюстрации эффективности использования файлов, отображенных в память, мы создали проект, предназначенный для поиска текста в текстовых файлах текущего каталога. Главная форма, разработанная для этого проекта, показана на рис. 12.2.

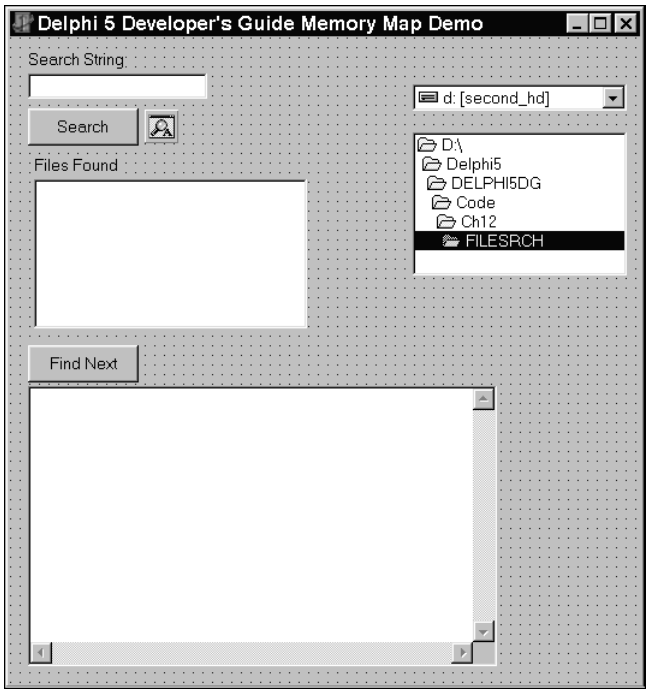


Рис. 12.2. Главная форма для проекта поиска текста

В окно списка, расположенного на главной форме, добавляются имена файлов вместе с количеством вхождений искомой строки, обнаруженных в соответствующем файле. Этот проект, имеющий название FileSrch.dpr, можно найти на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Проект служит также иллюстрацией того, как в объекте инкапсулируются функции файлов, отображенных в память. Для демонстрации был создан класс TMemMapFile.

Класс TMemMapFile

Исходный текст модуля, содержащего класс TMemMapFile, представлен в листинге 12.11.

Листинг 12.11. Исходный код модуля MemMap.pas, в котором определяется класс TMemMapFile

```
unit MemMap;

interface

uses Windows, SysUtils, Classes;

type
  EMMFError = class(Exception);

  TMemMapFile = class
  private
    FFileName: String;    // Имя файла, отображенного в память
    FSize: Longint;       // Размер окна отображения
    FFileSize: Longint;   // Действительный размер файла
    FFileMode: Integer;   // Режим доступа к файлу
    FFileHandle: Integer; // Дескриптор файла
    FMapHandle: Integer;  // Дескриптор объекта отображения файла
    FData: PByte;         // Указатель на данные файла
    FMapNow: Boolean;     // Определяет, можно ли отображать окно просмотра
    procedure AllocFileHandle;
    { Считывает дескриптор дискового файла }
    procedure AllocFileMapping;
    { Считывает дескриптор объекта отображения файла }
    procedure AllocFileView;
    { Отображает окно просмотра в файл }
    function GetSize: Longint;
    { Возвращает размер отображенного окна }
  public
    constructor Create(FileName: String; FileMode: integer;
                       Size: integer; MapNow: Boolean); virtual;
    destructor Destroy; override;
    procedure FreeMapping;
    property Data: PByte read FData;
    property Size: Longint read GetSize;
    property FileName: String read FFileName;
    property FileHandle: Integer read FFileHandle;
```

```
    property MapHandle: Integer read FMapHandle;
  end;

implementation

constructor TMemMapFile.Create(FileName: String; FileMode: integer;
                               Size: integer; MapNow: Boolean);
{ Создает окно просмотра файла FileName.
  FileName: Полный путь файла.
  FileMode: Используются константы fmXXX.
  Size: размер отображения памяти. При использовании собственного
        размера файла передайте нулевое значение. }
begin

  { Инициализируем закрытые поля }
  FMapNow := MapNow;
  FFileName := FileName;
  FFileMode := FileMode;

  AllocFileHandle;  // Получаем дескриптор дискового файла
  { Предполагаем, что файл < 2 Гбайт }

  FFileSize := GetFileSize(FFileHandle, Nil);
  FSize := Size;

  try
    AllocFileMapping; // Получаем дескриптор объекта отображения файла
  except
    on EMMFError do
    begin
      CloseHandle(FFileHandle); // Закрываем дескриптор файла в случае ошибки
      FFileHandle := 0;         // Снова устанавливаем дескриптор равным 0
      raise;                    // Повторно генерируем исключительную ситуацию
    end;
  end;
  if FMapNow then
    AllocFileView;  // Отображаем окно просмотра файла
end;

destructor TMemMapFile.Destroy;
begin

  if FFileHandle <> 0 then
    CloseHandle(FFileHandle); // Освобождаем дескриптор файла

  { Освобождаем дескриптор объекта отображения файла }
  if FMapHandle <> 0 then
    CloseHandle(FMapHandle);

  FreeMapping;   { Отменяем отображение окна просмотра файла }
```

```
  inherited Destroy;
end;

procedure TMemMapFile.FreeMapping;
{ В этом методе отменяется отображение окна просмотра файла
  в адресное пространство данного процесса }
begin
  if FData <> Nil then
  begin
    UnmapViewOfFile(FData);
    FData := Nil;
  end;
end;

function TMemMapFile.GetSize: Longint;
begin
  if FSize <> 0 then
    Result := FSize
  else
    Result := FFileSize;
end;

procedure TMemMapFile.AllocFileHandle;
{ Создает или открывает дисковый файл перед созданием файла,
  отображенного в память }
begin
  if FFileMode = fmCreate then
    FFileHandle := FileCreate(FFileName)
  else
    FFileHandle := FileOpen(FFileName, FFileMode);

  if FFileHandle < 0 then
    raise EMMFError.Create('Failed to open or create file');
        // Не удалось открыть или создать файл
end;

procedure TMemMapFile.AllocFileMapping;
var
  ProtAttr: DWORD;
begin
  if FFileMode = fmOpenRead then
    // Получаем атрибут корректной защиты
    ProtAttr := Page_ReadOnly
  else
    ProtAttr := Page_ReadWrite;
  { Пытаемся создать отображение дискового файла.
    В случае ошибки возбуждаем исключительную ситуацию. }
  FMapHandle := CreateFileMapping(FFileHandle, Nil, ProtAttr,
      0, FSize, Nil);
  if FMapHandle = 0 then
```

```
    raise EMMFError.Create('Failed to create file mapping');
        // Не удалось создать отображение файла
end;

procedure TMemMapFile.AllocFileView;
var
  Access: Longint;
begin
  if FFileMode = fmOpenRead then
    // Получаем корректный режим доступа к файлу
    Access := File_Map_Read
  else
    Access := File_Map_All_Access;
  FData := MapViewOfFile(FMapHandle, Access, 0, 0, FSize);
  if FData = Nil then
    raise EMMFError.Create('Failed to map view of file');
        // Не удалось отобразить окно просмотра файла
end;

end.
```

В комментариях описано назначение различных полей и методов класса `TMemMapFile`.

Этот класс содержит методы `AllocFileHandle()`, `AllocFileMapping()` и `AllocFileView()`, которые используются для считывания дескрипторов дискового файла, объекта отображения и окна просмотра заданного файла соответственно.

Именно в конструкторе `Create()` выполняется инициализация полей и вызываются методы для назначения различных дескрипторов. При неудачном выполнении любого из этих методов возбуждается исключительная ситуация. Деструктор `Destroy()` гарантирует, что при вызове метода `UnMapViewOfFile()` отменяется окно просмотра файла.

Использование класса `TMemMapFile`

Функционирование главной формы, предназначенной для проекта поиска текста, реализовано с помощью кода, представленного в листинге 12.12.

Листинг 12.12. Исходный код главной формы проекта поиска текста

```
unit MainFrm;

interface

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

type

  TMainForm = class(TForm)
    btnSearch: TButton;
    lbFilesFound: TListBox;
```

```
    edtSearchString: TEdit;
    lblSearchString: TLabel;
    lblFilesFound: TLabel;
    memFileText: TMemo;
    btnFindNext: TButton;
    FindDialog: TFindDialog;
    dcbDrives: TDriveComboBox;
    dlbDirectories: TDirectoryListBox;
    procedure btnSearchClick(Sender: TObject);
    procedure lbFilesFoundClick(Sender: TObject);
    procedure btnFindNextClick(Sender: TObject);
    procedure FindDialogFind(Sender: TObject);
    procedure edtSearchStringChange(Sender: TObject);
    procedure memFileTextChange(Sender: TObject);
  public
  end;

var
  MainForm: TMainForm;

implementation
uses MemMap, Search;

{$R *.DFM}

procedure TMainForm.btnSearchClick(Sender: TObject);
var
  MemMapFile: TMemMapFile;
  SearchRec: TSearchRec;
  RetVal: Integer;
  FoundStr: PChar;
  FName: String;
  FindString: String;
  WordCount: Integer;
begin
  memFileText.Lines.Clear;
  btnFindNext.Enabled := False;
  lbFilesFound.Items.Clear;

  { Считываем каждый текстовый файл, в котором предполагается
    выполнить поиск текста. Используем при поиске последовательность
    вызовов функций FindFirst/FindeNext. }
  RetVal := FindFirst(dlbDirectories.Directory+'\ *.txt', faAnyFile, SearchRec);
  try
    while RetVal = 0 do
    begin
      FName := SearchRec.Name;

      { Открываем файл, отображенный в память, с доступом "только для чтения".}
      MemMapFile := TMemMapFile.Create(FName, fmOpenRead, 0, True);
```

```
      try

        { Используем для искомой строки временную область памяти }
        FindString := edtSearchString.Text;

        WordCount := 0;
            // Начальная установка счетчика WordCount
        {  Получаем первое вхождение искомой строки }
        FoundStr := StrPos(PChar(MemMapFile.Data), PChar(FindString));

        if FoundStr <> nil then
        begin
          { Продолжаем поиск вхождений заданной строки в остальной
            части текста файла. При каждом обнаружении искомой
            строки увеличиваем переменную WordCount. }
          repeat
            inc(WordCount);
            inc(FoundStr, Length(FoundStr));

           { Считываем следующее вхождение искомой строки. }
            FoundStr := StrPos(PChar(FoundStr), PChar(FindString));
          until FoundStr = nil;
          { Добавляем имя файла в окно списка. }
          lbFilesFound.Items.Add(SearchRec.Name +
             ' - '+IntToStr(WordCount));
        end;
        { Считываем следующий файл для выполнения поиска. }
        RetVal := FindNext(SearchRec);
      finally
        MemMapFile.Free;
        { Освобождаем экземпляр файла, отображенного в память. }
      end;
    end;
  finally
    FindClose(SearchRec);
  end;
end;

procedure TMainForm.lbFilesFoundClick(Sender: TObject);
var
  FName: String;
  B: Byte;
begin
  with lbFilesFound do
    if ItemIndex <> -1 then
    begin
      B := Pos(' ', Items[ItemIndex]);
      FName := Copy(Items[ItemIndex], 1, B);
      memFileText.Clear;
      memFileText.Lines.LoadFromFile(FName);
```

```
    end;
end;

procedure TMainForm.btnFindNextClick(Sender: TObject);
begin
  FindDialog.FindText := edtSearchString.Text;
  FindDialog.Execute;
  FindDialog.Top := Top+Height;
  FindDialogFind(FindDialog);
end;

procedure TMainForm.FindDialogFind(Sender: TObject);
begin
  with Sender as TFindDialog do
    if not SearchMemo(memFileText, FindText, Options) then
      ShowMessage('Cannot find "' + FindText + '".');
end;

procedure TMainForm.edtSearchStringChange(Sender: TObject);
begin
  btnSearch.Enabled := edtSearchString.Text <> EmptyStr;
end;

procedure TMainForm.memFileTextChange(Sender: TObject);
begin
  btnFindNext.Enabled := memFileText.Lines.Count > 0;
end;

end.
```

В этом проекте выполняется поиск (с учетом регистра) заданной строки в текстовых файлах текущего каталога.

Процедура `btnSearchClick()` содержит код, предназначенный для выполнения реального поиска, определения числа вхождений заданной строки в каждом файле и добавления имен файлов, в которых была обнаружена искомая строка, в список `lbFilesFound`.

Сначала в этой процедуре с помощью последовательности вызовов функций `FindFirst()`/`FindNext()` в текущем каталоге выполняется поиск файлов с расширением `.txt`. Обе функции рассматриваются ниже в этой главе. Затем для получения доступа к данным очередного текстового файла используется класс `TMemMapFile`, причем файл открывается с доступом "только для чтения", поскольку в этом проекте никакой модификации данных не предусмотрено. В следующих строках кода реализована логика, требуемая для получения числа вхождений искомой строки в рассматриваемом файле:

```
if FoundStr <> nil then
begin
  repeat
    inc(WordCount);
    inc(FoundStr, length(FoundStr));
    FoundStr := StrPos(PChar(FoundStr), PChar(FindString))
  until FoundStr = nil;
```

Как имя файла, так и число вхождений искомой строки в файле добавляются в список lbFilesFound для отображения в главной форме проекта.

Когда пользователь дважды щелкает на элементе TListBox, в окно примечания (элемент управления TMemo) загружается соответствующий файл, в котором можно найти очередное вхождение искомой строки, щелкнув на кнопке Find Next (Найти следующее).

Обработчик событий btnFindNext() инициализирует свойство FindDialog.FindText, устанавливая его равным тексту, содержащемуся в строке редактирования edtSearchString. Затем активизируется диалоговое окно FindDialog.

Когда пользователь щелкает на кнопке Find Next в диалоговом окне FindDialog, вызывается обработчик события OnFind, реализованный в виде процедуры FindDialogFind(). В ней используется функция SearchMemo(), которая определена в модуле Search.pas.

На заметку

Модуль Search.pas представляет собой файл, который поставлялся с Borland Delphi 1.0 в качестве одного из демонстрационных проектов. В этом модуле не используются различные средства обработки строк, поскольку он предназначен для Delphi 1.0, но мы внесли минимальные изменения, благодаря которым курсор сможет оказаться в поле зрения элемента управления TMemo, что автоматически делалось в Windows 3.1. А в Win32 в элемент управления TMemo (после установки его свойства SelStart) необходимо передавать сообщение Windows EM_SCROLLCARET. Для получения дополнительной информации читайте комментарии, приведенные в тексте модуля Search.pas.

Функция SearchMemo() просматривает текст любого потомка класса TCustomEdit и выбирает нужный фрагмент текста для отображения в поле примечания.

Каталоги и устройства

В приложениях можно эффективно выполнять разнообразные задачи, связанные с устройствами, инсталлированными в системе, и каталогами данных устройств. Некоторые из этих задач рассматриваются в следующих разделах.

Получение списка доступных устройств и их типов

Для получения списка доступных устройств в системе используйте функцию Win32 API GetDriveType(). Ей передается параметр Pchar, а она возвращает целое значение, представляющее один из типов устройств, перечисленных в табл. 12.6.

Таблица 12.6. Значения, возвращаемые функцией GetDriveType()

Возвращаемое значение	Описание
0	Тип устройства определить невозможно
1	Корневого каталога не существует
DRIVE_REMOVABLE	Съемное устройство памяти
DRIVE_FIXED	Несъемное устройство памяти

Окончание табл. 12.6

Возвращаемое значение	Описание
DRIVE_REMOTE	Удаленное (сетевое) устройство памяти
DRIVE_CDROM	Компакт-диск
DRIVE_RAMDISK	Псевдодиск в ОЗУ

В листинге 12.13 демонстрируется вариант использования функции `GetDriveType()`.

Листинг 12.13. Использование функции `GetDriveType()`

```
procedure TMainForm.btnGetDriveTypesClick(Sender: TObject);
var
  i: Integer;
  C: String;
  DType: Integer;
  DriveString: String;
begin
  { Цикл по всем буквам A..Z для определения доступных устройств }
  for i := 65 to 90 do
  begin
    C := chr(i)+':\ ';
    // Форматируем строку для представления корневого каталога
    { Вызываем функцию GetDriveType(), которая возвращает целое значение,
      представляющее один из типов, приведенных в инструкции case. }
    DType := GetDriveType(PChar(C));
    { На основе полученного типа устройства форматируем строку
      для добавления в список различных типов устройств. }
    case DType of
      0: DriveString := C+' The drive type cannot be determined.';
        // Нельзя определить тип устройства
      1: DriveString := C+' The root directory does not exist.';
        // Корневого каталога не существует
      DRIVE_REMOVABLE: DriveString :=
         C+' The drive can be removed from the drive.';
        // Съемное устройство
      DRIVE_FIXED: DriveString :=
         C+' The disk cannot be removed from the drive.';
        // Несъемное устройство
      DRIVE_REMOTE: DriveString :=
         C+' The drive is a remote (network) drive.';
        // Удаленное (сетевое) устройство
      DRIVE_CDROM: DriveString := C+' The drive is a CD-ROM drive.';
        // Компакт-диск
      DRIVE_RAMDISK: DriveString := C+' The drive is a RAM disk.';
        // Псевдодиск в ОЗУ
    end;
```

```
    { Добавляем только те типы, которые поддаются определению. }
    if not ((DType = 0) or (DType = 1)) then
      lbDrives.Items.AddObject(DriveString, Pointer(i));
  end;

end;
```

В листинге 12.13 представлен текст простой процедуры, в которой выполняется цикл по всем символам букв алфавита. Эти символы (в виде выражения для корневого каталога) по очереди передаются функции `GetDriveType()`, чтобы определить, относятся ли они к допустимым типам устройств. Если тип поддается определению, функция `GetDriveType()` возвращает соответствующий тип устройства, распознаваемый с помощью инструкции `case`. При этом создается описывающая строка, которая добавляется в окно списка вместе с символом, представляющим букву устройства в массиве списка `Objects`. Следует отметить, что в список добавляются только те устройства, которые существуют в системе и тип которых поддается определению. Кстати, Delphi 5 поставляется с компонентом `TDriveComboBox`, позволяющим выбирать дисковое устройство. Его можно найти во вкладке Win3.1 палитры компонентов.

Получение информации об устройстве

Помимо определения доступных устройств и их типов, можно также получить полезную информацию о конкретном устройстве, которая включает следующие элементы:

- число секторов на кластер;
- число байтов в секторе;
- количество свободных кластеров;
- общее число кластеров;
- общее число байтов свободного пространства на диске;
- общее число байтов на диске (размер диска).

Четыре первых элемента можно получить путем вызова функции Win32 API `GetDiskFreeSpace()`; на их основе вычисляются два последних. Как использовать функцию `GetDiskFreeSpace()`, показано на примере листинга 12.14.

Листинг 12.14. Использование функции `GetDiskFreeSpace()`

```
procedure TMainForm.lbDrivesClick(Sender: TObject);
var
  RootPath: String;          // Хранит путь корневого каталога устройства
  SectorsPerCluster: DWord;  // Число секторов в кластере
  BytesPerSector: DWord;     // Число байтов в секторе
  NumFreeClusters: DWord;    // Число свободных кластеров
  TotalClusters: DWord;      // Общее число кластеров
  DriveByte: Byte;           // Значение байта устройства
  FreeSpace: Int64;          // Свободное пространство на диске
  TotalSpace: Int64;         // Размер диска
  DriveNum:  Integer;        // Номер диска: 1 = A, 2 = B и т.д.
```

```
begin
  with lbDrives do
  begin
    { Преобразуем значение ASCII, используемое для буквы
      устройства, в допустимый номер устройства (1 = A, 2 = B
      и т.д.) путем вычитания числа 64 из значения кода ASCII. }
    DriveByte := Integer(Items.Objects[ItemIndex])-64;
    { Сначала создаем строку пути корневого каталога. }
    RootPath := chr(Integer(Items.Objects[ItemIndex]))+':\ ';
    { Вызов функции GetDiskFreeSpace для получения информации об устройстве }
    if GetDiskFreeSpace(PChar(RootPath), SectorsPerCluster,
      BytesPerSector, NumFreeClusters, TotalClusters) then
    begin
      { При успешном выполнении этой функции обновляем метки
        для отображения информации о диске. }
      lblSectPerCluster.Caption := Format('%.0n', [SectorsPerCluster*1.0]);
      lblBytesPerSector.Caption := Format('%.0n', [BytesPerSector*1.0]);
      lblNumFreeClust.Caption := Format('%.0n', [NumFreeClusters*1.0]);
      lblTotalClusters.Caption := Format('%.0n', [TotalClusters*1.0]);
      // Получаем информацию о свободном и максимальном дисковом пространстве
      FreeSpace  := DiskFree(DriveByte);
      TotalSpace := DiskSize(DriveByte);
      lblFreeSpace.Caption := Format('%.0n', [FreeSpace*1.0]);
      { Вычисляем общий объем дискового пространства. }
      lblTotalDiskSpace.Caption := Format('%.0n', [TotalSpace*1.0]);
    end
    else begin
      { Заполняем метки для "пустого" отображения. }
      lblSectPerCluster.Caption := 'X';
      lblBytesPerSector.Caption := 'X';
      lblNumFreeClust.Caption := 'X';
      lblTotalClusters.Caption := 'X';
      lblFreeSpace.Caption := 'X';
      lblTotalDiskSpace.Caption := 'X';
      ShowMessage('Cannot get disk info');
        // Информацию о диске получить не удалось
    end;
  end;

end;
```

В листинге 12.14 представлен код обработчика события `OnClick` для элемента управления списком.

Как видно из листинга 12.14, когда пользователь щелкает на одном из доступных элементов окна списка `lbDrives`, для выбранного диска создается строковое представление корневого каталога, которое передается функции `GetDiskFreeSpace()`. Если функция успешно справится с задачей получения информации об устройстве, для ее отображения будут обновлены различные метки главной формы. Пример формы, разработанной для данного проекта, показан на рис. 12.3.

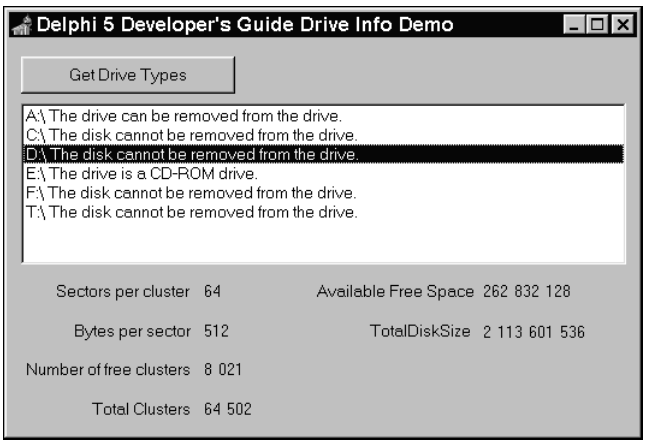


Рис. 12.3. Главная форма, отображающая информацию о диске, выбранном из списка доступных устройств

Обратите внимание, что в этом обработчике событий для определения размера диска и объема свободного пространства не используются значения, возвращаемые функцией `GetDiskFreeSpace()`. Вместо этого применяются функции `DiskFree()` и `DiskSize()`, которые определены в модуле `SysUtils.pas`. Дело в том, что функция `GetDiskFreeSpace()` в Windows 95 не доработана — она не в состоянии правильно оценить объем дисков, размер которых превышает 2 Гбайт, а для дисков размером, большим чем 1 Гбайт, сообщаются усеченные размеры секторов. Функции `DiskSize()` и `DiskFree()` используют для получения информации новые возможности Win32 API, если они доступны в операционной системе.

Получение информации о размещении каталога Windows

Для получения информации о расположении каталога Windows необходимо использовать функцию Win32 API `GetWindowsDirectory()`, которая определена следующим образом:

```
function GetWindowsDirectory(lpBuffer: PChar; uSize: UINT): UINT;
```

Первый параметр представляет собой буфер, в который помещается каталог Windows в виде строки с ограничивающим нуль-символом. Второй параметр определяет размер буфера. Ниже показано, как следует использовать эту функцию.

```
var
  WDir: String;
begin
  SetLength(WDir, 144);
  if GetWindowsDirectory(PChar(WDir), 144) <> 0 then
  begin
    SetLength(WDir, StrLen(PChar(WDir)));
    ShowMessage(WDir);
  end
  else
    RaiseLastWin32Error;
end;
```

На заметку

Нетрудно заметить, что в приведенном фрагменте кода после обращения к функции `GetWindowsDirectory()` добавлена следующая строка:

```
SetLength(WDir, StrLen(PChar(WDir)));
```

При передаче длинной строки функции с помощью такой операции приведения типа, как `PChar`, Delphi не в состоянии узнать, была ли эта строка обработана, а следовательно, не может обновить информацию о ее длине. Вы должны сделать это в явном виде: для поиска ограничивающего нуль-символа, который позволит определить длину строки, используйте функцию `StrLen()` и измените размер строки с помощью функции `SetLength()`.

Следует отметить, что использование переменной длинной строки позволило выполнить операцию приведения к типу `PChar`. Функция `GetWindowsDirectory()` возвращает целое значение, представляющее длину пути искомого каталога. В случае возникновения ошибки возвращается нулевое значение, и тогда для определения причины ошибочной ситуации нужно вызвать функцию `RaiseLastWin32Error`.

Получение информации о размещении системного каталога

Вызвав функцию Win32 API `GetSystemDirectory()`, можно также получить информацию о расположении системного каталога. Эта функция работает аналогично функции `GetWindowsDirectory()`, за исключением того, что она возвращает полный путь в системный каталог Windows, а не в некоторый другой каталог. Следующий фрагмент кода показывает, как использовать эту функцию:

```
var
  SDir: String;
begin
  SetLength(SDir, 144);
  if GetSystemDirectory(PChar(SDir), 144) <> 0 then
  begin
    SetLength(SDir, StrLen(PChar(SDir)));
    ShowMessage(SDir);
  end
  else
    RaiseLastWin32Error;
end;
```

Значения, возвращаемые этой функцией, совпадают со значениями, возвращаемыми функцией `GetWindowsDirectory()`.

Получение имени текущего каталога

Часто возникает необходимость в получении имени текущего каталога, из которого было запущено данное приложение. Для этого достаточно вызвать функцию Win32 API `GetCurrentDirectory()`. Если вы думаете, что эта функция должна действовать подобно

двум предыдущим, то совершенно правы (или почти правы). Разница лишь в том, что у нее другой порядок следования параметров. Приведенный ниже фрагмент кода иллюстрирует использование этой функции.

```
var
  CDir: String;
begin
  SetLength(CDir, 144);
  if GetCurrentDirectory(144, PChar(CDir)) <> 0 then
  begin
    SetLength(CDir, StrLen(PChar(CDir)));
    ShowMessage(CDir);
  end
  else
    RaiseLastWin32Error;
end;
```

На заметку Для аналогичных целей в Delphi предусмотрены функции `CurDir()` и `ChDir()`, определенные в модуле `System`, а также функции `GetCurrentDir()` и `SetCurrentDir()`, определенные в модуле `SysUtils.pas`.

На заметку Delphi поставляется с собственным набором процедур, предназначенных для получения сведений о каталоге указанного файла. Например, свойство `TApplication.ExeName` содержит полный путь и имя файла выполняемого процесса. Если, например, этот параметр будет иметь значение `C:\Delphi\Bin\Project.exe`, то некоторые функции при передаче им свойства `TApplication.ExeName` вернут значения, перечисленные в табл. 12.7.

Таблица 12.7. Функции, возвращающие данные о файле или каталоге

Функция	Результат при передаче параметра C:\Delphi\Bin\Project.exe
`ExtractFileDir()`	`C:\Delphi\Bin`
`ExtractFileDrive()`	`C:`
`ExtractFileExt()`	`.exe`
`ExtractFileName()`	`Project1.exe`
`ExtractFilePath()`	`C:\Delphi\Bin\`

Поиск файла

Возможно, в один прекрасный день у вас возникнет необходимость найти в каком-то каталоге и его подкаталогах некоторые файлы (заданные маской) или выполнить с ними некоторые операции. В листинге 12.15 показано, как это можно сделать с помощью процедуры, которая использует рекурсию для того, чтобы вместе с текущим каталогом просматривались и все его подкаталоги.

На заметку Для поиска заданного каталога, системных каталогов, каталогов, заданных с помощью переменной окружения `PATH`, или списка каталогов, разделенных точкой с запятой, можно использовать функцию Win32 API `SearchPath()`. К сожалению, эта функция не выполняет поиск файла среди подкаталогов данного каталога.

Листинг 12.15. Пример просмотра каталогов в поисках файла

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, FileCtrl, Grids, Outline, DirOutln;

type
  TMainForm = class(TForm)
    dcbDrives: TDriveComboBox;
    edtFileMask: TEdit;
    lblFileMask: TLabel;
    btnSearchForFiles: TButton;
    lbFiles: TListBox;
    dolDirectories: TDirectoryOutline;
    procedure btnSearchForFilesClick(Sender: TObject);
    procedure dcbDrivesChange(Sender: TObject);
  private
    FFileName: String;
    function GetDirectoryName(Dir: String): String;
    procedure FindFiles(APath: String);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

function TMainForm.GetDirectoryName(Dir: String): String;
{ Эта функция форматирует имя каталога таким образом, чтобы оно в
  качестве последнего символа содержало обратную косую черту (\). }
begin
  if Dir[Length(Dir)]<> '\ ' then
    Result := Dir+'\ '
  else
    Result := Dir;
end;

procedure TMainForm.FindFiles(APath: String);
{ Эта процедура вызывается рекурсивно для выполнения поиска
  заданного маской файла в текущем каталоге и его подкаталогах.}
var
  FSearchRec,
  DSearchRec: TSearchRec;
  FindResult: integer;
```

```
  function IsDirNotation(ADirName: String): Boolean;
  begin
    Result := (ADirName = '.') or (ADirName = '..');
  end;

begin
  APath := GetDirectoryName(APath);   // Получаем имя каталога
  { Находим первое вхождение заданного имени файла. }
  FindResult := FindFirst(APath+FFileName,faAnyFile+faHidden+
    faSysFile+faReadOnly,FSearchRec);
  try
    { Продолжаем искать файлы в соответствии с заданной маской.
      При обнаружении искомого файла добавляем в список его имя и путь.}
    while FindResult = 0 do
    begin
      lbFiles.Items.Add(LowerCase(APath+FSearchRec.Name));
      FindResult := FindNext(FSearchRec);
    end;

    { Теперь просмотрим подкаталоги текущего каталога. Для этого
      воспользуемся функцией FindFirst для циклического просмотра
      каждого каталога, а затем снова вызовем функцию FindFiles
      (т.е. себя же). Этот рекурсивный процесс будет продолжаться
      до тех пор, пока не будут просмотрены все подкаталоги. }
    FindResult := FindFirst(APath+'*.*', faDirectory, DSearchRec);

    while FindResult = 0 do
    begin
      if ((DSearchRec.Attr and faDirectory) = faDirectory) and not
        IsDirNotation(DSearchRec.Name) then
        FindFiles(APath+DSearchRec.Name); // Рекурсия
      FindResult := FindNext(DSearchRec);
    end;
  finally
    FindClose(FSearchRec);
  end;
end;

procedure TMainForm.btnSearchForFilesClick(Sender: TObject);
{ Этот метод запускает процесс поиска. Сначала курсор принимает
  вид песочных часов, означающих, что для выполнения поиска
  потребуется некоторое время. Затем очищается список и
  рекурсивно вызывается функция FindFiles() для просмотра подкаталогов. }
begin
  Screen.Cursor := crHourGlass;
  try
    lbFiles.Items.Clear;
    FFileName := edtFileMask.Text;
    FindFiles(dolDirectories.Directory);
  finally
    Screen.Cursor := crDefault;
```

```
  end;
end;

procedure TMainForm.dcbDrivesChange(Sender: TObject);
begin
  dolDirectories.Drive := dcbDrives.Drive;
end;

end.
```

В методе `FindFiles()` первая конструкция `while..do` выполняет поиск файлов в текущем каталоге, заданном параметром `APath`, а затем добавляет найденные файлы и их пути в список `lbFiles`. Вторая конструкция `while..do` предназначена для отыскания подкаталогов в текущем каталоге и добавления их в конец переменной `APath`. Затем метод `FindFiles()` передает самому себе параметр `APath`, содержащий уже и имя подкаталога, образуя таким образом рекурсивный вызов. Этот процесс продолжается до тех пор, пока не будут просмотрены все подкаталоги.

На рис. 12.4 показаны результаты поиска `.PAS`-файлов в каталоге Delphi 5.

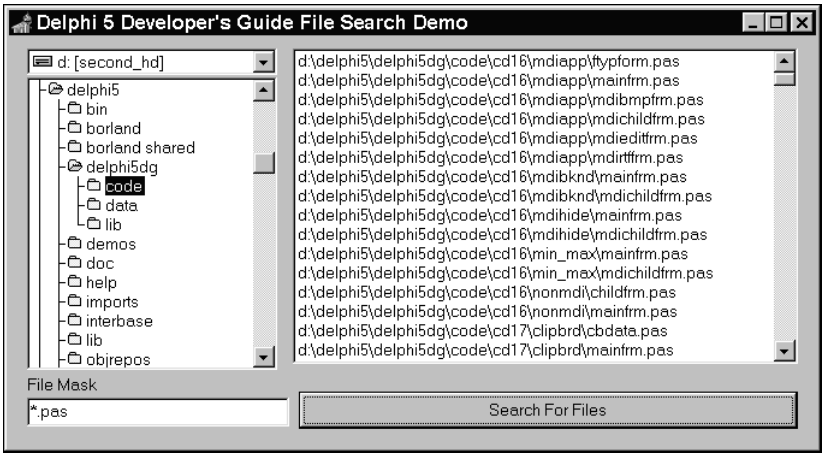


Рис. 12.4. Отображение результатов поиска файлов в каталогах

В этом проекте особого внимания заслуживают две структуры Object Pascal. Прежде всего рассмотрим структуру `TSearchRec` и функции `FindFirst()` и `FindNext()`, а затем — структуру `TWin32FindData`.

Копирование и удаление дерева каталогов

До существования Win32 для копирования каталога в другое место диска нам приходилось анализировать дерево каталогов и использовать пары функций `FindFirst()`/`FindNext()`. Теперь можно прибегнуть к помощи функции Win32 `ShFileOperation()`, которая значительно упрощает этот процесс. Ниже приводится код, иллюстрирующий работу процедуры, которая использует функцию Win32 API `ShFileOperation()` для выполнения операции копирования каталога. Эта функция хорошо описана в интерактивной справочной системе Win32, и

мы не будем дублировать здесь эту информацию. Наоборот, будем полагать, что вы уже тщательно ознакомились с этим разделом справочных данных. Обратите также внимание на включение модуля ShellAPI в предложение uses.

```
Uses
  ShellAPI;

procedure CopyDirectoryTree(AHandle: THandle; AFromDir, AToDir: String);
var
  SHFileOpStruct: TSHFileOpStruct;
Begin
  with SHFileOpStruct do
  begin
    Wnd     := AHandle;
    wFunc   := FO_COPY;
    pFrom   := PChar(AFromDir);
    pTo     := PChar(AToDir);
    fFlags  := FOF_NOCONFIRMATION or FOF_RENAMEONCOLLISION;
    fAnyOperationsAborted   := False;
    hNameMappings           := nil;
    lpszProgressTitle       := nil;
  end;
  ShFileOperation(SHFileOpStruct);
end;
```

Функцию ShFileOperation() также можно использовать для перемещения каталога в корзину:

```
uses ShellAPI;

procedure ToRecycle(AHandle: THandle; AFileName: STring);
var
  SHFileOpStruct: TSHFileOpStruct;
begin
  with SHFileOpStruct do
  begin
    Wnd    := AHandle;
    wFunc  := FO_DELETE;
    pFrom  := PChar(AFileName);
    fFlags := FOF_ALLOWUNDO;
  end;
  SHFileOperation(SHFileOpStruct);
end;
```

Запись TSearchRec

Запись TSearchRec определяет данные, необходимые для передачи функциям FindFirst() и FindNext(). Object Pascal определяет эту запись следующим образом:

```
TSearchRec = record
    Time: Integer;
    Size: Integer;
```

```
    Attr: Integer;
    Name: TFileName;
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;
end;
```

При обнаружении искомого файла поля записи `TSearchRec` модифицируются вышеупомянутыми функциями.

Поле `Time` содержит время создания или модификации файла, а поле `Size` — размер файла в байтах. В поле `Name` хранится имя файла, а поле `Attr` содержит один или несколько атрибутов файла, перечисленных в табл. 12.8.

Таблица 12.8. Атрибуты файла

Атрибут	Значение	Описание
`faReadOnly`	`$01`	Файл, предназначенный только для чтения
`faHidden`	`$02`	Скрытый файл
`faSysFile`	`$04`	Системный файл
`faVolumeID`	`$08`	Метка тома
`faDirectory`	`$10`	Каталог
`faArchive`	`$20`	Архивный файл
`faAnyFile`	`$3F`	Любой файл

Поля `FindHandle` и `ExcludeAttr` используются функциями `FindFirst()` и `FindNext()` для внутренних нужд, поэтому нет необходимости вникать в их назначение.

Функции `FindFirst()` и `FindNext()` принимают путь в качестве параметра, который содержит символы шаблона (например, выражение `C:\DELPHI 5\BIN\*.EXE` означает все файлы с расширением `EXE` в каталоге `C:\DELPHI 5\BIN\`). Параметр `Attr` определяет атрибуты файла, по которым следует проводить поиск. Если, например, вы хотите найти только системные файлы, следует вызывать функции `FindFirst()` и/или `FindNext()` следующим образом:

```
FindFirst(Path, faSysFile, SearchRec);
```

Запись `TWin32FindData`

Запись `TWin32FindData` содержит информацию о найденном файле или подкаталоге и определяется следующим образом:

```
TWin32FindData = record
    dwFileAttributes: DWORD;
    ftCreationTime: TFileTime;
    ftLastAccessTime: TFileTime;
    ftLastWriteTime: TFileTime;
    nFileSizeHigh: DWORD;
    nFileSizeLow: DWORD;
    dwReserved0: DWORD;
```

```
    dwReserved1: DWORD;
    cFileName: array[0..MAX_PATH - 1] of AnsiChar;
    cAlternateFileName: array[0..13] of AnsiChar;
  end;
```

В табл. 12.9 описаны значения полей записи TWin32FindData.

Таблица 12.9. Значения полей записи TWin32FindData

Поле	Значение
dwFileAttributes	Атрибуты найденного файла. За дополнительной информацией обращайтесь к электронной справочной системе (параграф WIN32_FIND_DATA)
FtCreationTime	Время создания файла
FtLastAccessTime	Время последнего доступа к файлу
FtLastWriteTime	Время последней модификации файла
NFileSizeHigh	Старшие разряды (старшее двойное слово DWORD) размера файла в байтах. Если размер файла не превышает MAXDWORD, то это значение равно нулю
NFileSizeLow	Младшие разряды (младшее двойное слово DWORD) размера файла в байтах
DwReserved0	В данный момент не используется — зарезервировано
DwReserved1	В данный момент не используется — зарезервировано
CFileName	Имя файла в виде строки с ограничивающим нуль-символом
CAlternateFileName	Имя файла в формате 8.3, усечение длинного имени файла

Получение информации о версии файла

Из файлов EXE или DLL можно выделить информацию об их версиях. В следующих разделах описано создание класса, который инкапсулирует функции, предназначенные для выделения информации о версии, и приведен пример проекта с использованием этого класса.

Определение класса TVerInfoRes

Класс TVerInfoRes инкапсулирует три функции Win32 API для выделения информации из файлов, которые ее содержат. Речь идет о функциях GetFileVersionInfoSize(), GetFileVersionInfo() и VerQueryValue(). Информация о версии может включать такие данные, как имя компании, описание файла, номер версии и некоторые комментарии. Поскольку программирование не терпит приблизительности, ниже приводится полный список разделов, из которых состоит информация о версии файла.

- *Имя компании*. Имя компании, создавшей файл.
- *Комментарии*. Любые дополнительные комментарии, которые характеризуют файл.
- *Описание файла*. Описание файла.
- *Версия файла*. Номер версии.
- *Внутреннее имя*. Внутреннее имя, присвоенное компанией — создателем файла.

- *Допустимые авторские права*. Все замечания об авторских правах, применимые к данному файлу.
- *Допустимые торговые марки*. Допустимые торговые марки, применимые к данному файлу.
- *Оригинальное имя файла*. Оригинальное имя файла, если таковое существует.

Модуль VERINFO.PAS, в котором определяется класс TVerInfoRes, представлен в листинге 12.16.

Листинг 12.16. Исходный код модуля VERINFO.PAS, содержащего определение класса TVerInfoRes

```
unit VerInfo;

interface

uses SysUtils, WinTypes, Dialogs, Classes;

type
  { Определяем общие классы для исключительных ситуаций,
    возникающих при получении информации о версии, а также при
    недоступности этой информации. }
  EVerInfoError   = class(Exception);
  ENoVerInfoError = class(Exception);
  eNoFixeVerInfo  = class(Exception);

  { Определяем перечислимый тип, представляющий различные
    виды информации о версии.}
  TVerInfoType =
    (viCompanyName,
     viFileDescription,
     viFileVersion,
     viInternalName,
     viLegalCopyright,
     viLegalTrademarks,
     viOriginalFilename,
     viProductName,
     viProductVersion,
     viComments);

const

  { Определяем массив строк-констант, представляющих заранее
    установленные ключевые элементы информации о версии файла.}
  VerNameArray: array[viCompanyName..viComments] of String[20] =
  ('CompanyName',
   'FileDescription',
   'FileVersion',
   'InternalName',
   'LegalCopyright',
   'LegalTrademarks',
```

```
   'OriginalFilename',
   'ProductName',
   'ProductVersion',
   'Comments');

type

  // Определям класс информации о версии
  TVerInfoRes = class
  private
    Handle            : DWord;
    Size              : Integer;
    RezBuffer         : String;
    TransTable        : PLongint;
    FixedFileInfoBuf  : PVSFixedFileInfo;
    FFileFlags        : TStringList;
    FFileName         : String;
    procedure FillFixedFileInfoBuf;
    procedure FillFileVersionInfo;
    procedure FillFileMaskInfo;
  protected
    function GetFileVersion   : String;
    function GetProductVersion: String;
    function GetFileOS        : String;
  public
    constructor Create(AFileName: String);
    destructor Destroy; override;
    function GetPreDefKeyString(AVerKind: TVerInfoType): String;
    function GetUserDefKeyString(AKey: String): String;
    property FileVersion    : String read GetFileVersion;
    property ProductVersion : String read GetProductVersion;
    property FileFlags      : TStringList read FFileFlags;
    property FileOS         : String read GetFileOS;
  end;

implementation

uses Windows;

const
  // Строки, которые должны быть переданы функции VerQueryValue()
  SFInfo               = '\ StringFileInfo\ ';
  VerTranslation: PChar = '\ VarFileInfo\ Translation';
  FormatStr            = '%s%.4x%.4x\ %s%s';


constructor TVerInfoRes.Create(AFileName: String);
begin
  FFileName := aFileName;
  FFileFlags := TStringList.Create;
```

```
  // Получаем информацию о версии файла
  FillFileVersionInfo;
  // Получаем фиксированную информацию о версии файла
  FillFixedFileInfoBuf;
  // Получаем значения масок файла
  FillFileMaskInfo;
end;

destructor TVerInfoRes.Destroy;
begin
  FFileFlags.Free;
end;

procedure TVerInfoRes.FillFileVersionInfo;
var
  SBSize: UInt;
begin
        // Определяем размер информации о версии
  Size := GetFileVersionInfoSize(PChar(FFileName), Handle);
  if Size <= 0 then
        // Если размер <= 0, генерируем исключительную ситуацию
    raise ENoVerInfoError.Create('No Version Info Available.');
        // Информация о версии недоступна
        // Устанавливаем соответствующую длину
  SetLength(RezBuffer, Size);
        // Заполняем буфер информацией о версии, а в случае
        // ошибки генерируем исключительную ситуацию
  if not GetFileVersionInfo(PChar(FFileName), Handle, Size,
      PChar(RezBuffer)) then
    raise EVerInfoError.Create('Cannot obtain version info.');
        // Невозможно получить информацию о версии
  if not VerQueryValue(PChar(RezBuffer), VerTranslation,
      pointer(TransTable), SBSize) then
    raise EVerInfoError.Create('No language info.');
end; // Нет информации о языке

procedure TVerInfoRes.FillFixedFileInfoBuf;
var
  Size: Longint;
begin
  if VerQueryValue(PChar(RezBuffer), '\ ',
      pointer(FixedFileInfoBuf), Size) then begin
    if Size < SizeOf(TVSFixedFileInfo) then
        raise eNoFixeVerInfo.Create('No fixed file info');
  end
  else
    raise eNoFixeVerInfo.Create('No fixed file info')
end;

procedure TVerInfoRes.FillFileMaskInfo;
```

```
begin
  with FixedFileInfoBuf^ do begin
    if (dwFileFlagsMask and dwFileFlags and VS_FF_PRERELEASE) <> 0 then
      FFileFlags.Add('Pre-release');
    if (dwFileFlagsMask and dwFileFlags and VS_FF_PRIVATEBUILD) <> 0 then
      FFileFlags.Add('Private build');
    if (dwFileFlagsMask and dwFileFlags and VS_FF_SPECIALBUILD) <> 0 then
      FFileFlags.Add('Special build');
    if (dwFileFlagsMask and dwFileFlags and VS_FF_DEBUG) <> 0 then
      FFileFlags.Add('Debug');
  end;
end;

function TVerInfoRes.GetPreDefKeyString(AVerKind: TVerInfoType): String;
var
  P: PChar;
  S: UInt;
begin
  Result := Format(FormatStr, [SfInfo, LoWord(TransTable^),
    HiWord(TransTable^), VerNameArray[aVerKind], #0]);
  // Получаем и возвращаем информацию о версии,
  // в случае ошибки возвращаем пустую строку
  if VerQueryValue(PChar(RezBuffer), @Result[1], Pointer(P), S) then
    Result := StrPas(P)
  else
    Result := '';
end;

function TVerInfoRes.GetUserDefKeyString(AKey: String): String;
var
  P: Pchar;
  S: UInt;
begin
  Result := Format(FormatStr, [SfInfo, LoWord(TransTable^),
    HiWord(TransTable^), aKey, #0]);
  // Получаем и возвращаем информацию о версии,
  // в случае ошибки возвращаем пустую строку
  if VerQueryValue(PChar(RezBuffer), @Result[1], Pointer(P), S) then
    Result := StrPas(P)
  else
    Result := '';
end;

function VersionString(Ms, Ls: Longint): String;
begin
  Result := Format('%d.%d.%d.%d', [HIWORD(Ms), LOWORD(Ms),
     HIWORD(Ls), LOWORD(Ls)]);
end;

function TVerInfoRes.GetFileVersion: String;
```

```
begin
  with FixedFileInfoBuf^ do
    Result := VersionString(dwFileVersionMS, dwFileVersionLS);
end;

function TVerInfoRes.GetProductVersion: String;
begin
  with FixedFileInfoBuf^ do
    Result := VersionString(dwProductVersionMS, dwProductVersionLS);
end;

function TVerInfoRes.GetFileOS: String;
begin
  with FixedFileInfoBuf^ do
    case dwFileOS of
      VOS_UNKNOWN:  // То же самое, что и VOS__BASE
        Result := 'Unknown';
      VOS_DOS:
        Result := 'Designed for MS-DOS';
      VOS_OS216:
        Result := 'Designed for 16-bit OS/2';
      VOS_OS232:
        Result := 'Designed for 32-bit OS/2';
      VOS_NT:
        Result := 'Designed for Windows NT';
      VOS__WINDOWS16:
        Result := 'Designed for 16-bit Windows';
      VOS__PM16:
        Result := 'Designed for 16-bit PM';
      VOS__PM32:
        Result := 'Designed for 32-bit PM';
      VOS__WINDOWS32:
        Result := 'Designed for 32-bit Windows';
      VOS_DOS_WINDOWS16:
        Result := 'Designed for 16-bit Windows, running on MS-DOS';
      VOS_DOS_WINDOWS32:
        Result := 'Designed for Win32 API, running on MS-DOS';
      VOS_OS216_PM16:
        Result := 'Designed for 16-bit PM, running on 16-bit OS/2';
      VOS_OS232_PM32:
        Result := 'Designed for 32-bit PM, running on 32-bit OS/2';
      VOS_NT_WINDOWS32:
        Result := 'Designed for Win32 API, running on Windows/NT';
    else
      Result := 'Unknown';
    end;
end;

end.
```

Класс TVerInfoRes содержит необходимые поля и инкапсулирует соответствующие методы Win32 API, требующиеся для получения информации о версии из любого файла. Файл, из которого добывается информация о версии, указывается путем передачи его имени конструктору TVerInfoRes.Create() в качестве параметра AFileName. Это имя файла присваивается полю FFileName, которое используется в другом методе для реального выделения информации о версии. Затем в этом конструкторе вызываются по очереди три метода: FillFileVersionInfo(), FillFixedFileInfoBuf() и FillFileMaskInfo().

Метод FillFileVersionInfo()

Этот метод предназначен для выполнения подготовительного этапа работы по загрузке информации о версии, после которого можно приступать к рассмотрению отдельных элементов этой информации. Прежде всего, в этом методе определяется факт наличия информации о версии, а в случае положительного результата — и ее размер. Сведения о размере позволяют узнать, какой объем памяти нужно выделить для хранения этой информации. Непосредственным определением размера информации о версии, содержащейся в исследуемом файле, занимается функция Win32 API GetFileVersionInfoSize(), которая объявляется следующим образом:

```
function GetFileVersionInfoSize(lptstrFilename: PChar; var lpdwHandle: DWORD):
DWORD; stdcall;
```

Параметр lptstrFilename указывает на файл, из которого должна быть получена информация о версии. Параметр lpdwHandle представляет собой переменную типа DWORD, которая при вызове этой функции устанавливается равной нулю. Насколько мы могли понять, другого назначения эта переменная не имеет.

Метод FillFileVersionInfo() передает параметр FFileName функции GetFileVersionInfoSize(), и если возвращаемое ею значение, присвоенное переменной Size, оказывается больше нуля, то для хранения указанного в переменной Size количества байтов выделяется буфер RezBuffer.

После выделения памяти для буфера RezBuffer последний передается функции GetFileVersionInfo(), которая заполняет его информацией о версии. Функция GetFileVersionInfo() определяется следующим образом:

```
function GetFileVersionInfo(lptstrFilename: PChar; dwHandle,
  dwLen: DWORD; lpData: Pointer): BOOL; stdcall;
```

В качестве параметра lptstrFilename передается имя файла FFileName. Параметр dwHandle игнорируется. Параметр dwLen — это не что иное, как значение, возвращаемое функцией GetFileVersionInfoSize(), которое хранится в переменной Size. Параметр lpData является указателем на буфер, выделенный для хранения информации о версии. Если считывание информации о версии не увенчается успехом, функция GetFileVersionInfo() вернет значение False; в противном случае — значение True.

И, наконец, метод FillFileVersionInfo() вызывает функцию API VerQueryValue(), которая используется для возврата информации о версии, выбранной из информационного ресурса. В данном примере функция VerQueryValue() вызывается с целью считывания указателя на массив языков и идентификаторов символьного набора. Этот массив используется в последующих обращениях к функции VerQueryValue() для получения доступа к определенной части информационного ресурса (с помощью поля TransTable), которая соответствует конкретному языку.

Функция VerQueryValue() определяется следующим образом:

```
function VerQueryValue(pBlock: Pointer; lpSubBlock: PChar; var
  lplpBuffer: Pointer; var puLen: UINT): BOOL; stdcall;
```

Параметр pBlock указывает на параметр lpData, который передается функции GetFileVersionInfo(). Параметр LpSubBlock представляет собой строку с завершающим нулем, которая определяет, какое значение информации о версии требуется считать. Весьма полезно просмотреть электронную справку о функции VerQueryValue(), где описаны различные строки, допустимые для передачи функции VerQueryValue(). Для предыдущего примера, чтобы считать информацию о языке и соответствующем наборе символов, в качестве параметра lpSubBlock передается строка "\VarFileInfo\Translation". Параметр lplpBuffer указывает на буфер, в котором хранится значение информации о версии, а параметр puLen содержит длину считываемых данных.

Метод FillFixedFileInfoBuf()

В методе FillFixedFileInfoBuf() иллюстрируется использование функции VerQueryValue() для получения указателя на структуру VS_FIXEDFILEINFO, которая содержит информацию о файле. Это делается путем передачи функции VerQueryValue() в качестве параметра lpSubBlock строки "\". Этот указатель сохраняется в поле TVerInfoRes.FixedFileInfoBuf.

Метод FillFileMaskInfo()

Этот метод служит иллюстрацией получения атрибутов модуля. Поставленная цель в данном случае достигается путем выполнения логических операций над соответствующими битовыми масками и полями dwFileFlagsMask и dwFileFlags поля FixedFileInfoBuf с последующей оценкой выделяемого признака. Мы не будем останавливаться на значениях этих признаков, но, если вас это интересует, можете обратиться к соответствующей странице интерактивной справочной системы Delphi, доступ к которой осуществляется по щелчку на кнопке Help вкладки Version Info диалогового окна Project Options.

Методы GetPreDefKeyString() и GetUserDefKeyString()

В этих методах иллюстрируется использование функции VerQueryValue() для считывания строк информации о версии, которые являются элементами таблицы Key, расположенной во вкладке Version Info диалогового окна Project Options. По умолчанию Win32 API предоставляет 10 заранее определенных строк, которые мы поместили в массив констант VerNameArray. Чтобы прочитать определенную строку, нужно передать в качестве параметра lpSubBlock функции VerQueryValue() строку "\StringFileInfo*lang-charset**string-name*". Значение *lang-charset* указывает на идентификатор языка и символьного набора, который был ранее считан в методе FillFileVersionInfo() и к которому можно получить доступ с помощью поля TransTable. Строка *string-name* определяет одну из встроенных строковых констант в массиве VerNameArray. Функция GetPreDefKeyString() предназначена для считывания встроенных строк информации о версии.

Функция GetUserDefKeyString() действует аналогично функции GetPreDefKeyString(), за исключением того, что в качестве параметра должна быть передана строка, содержащая ключ. В этом методе выполняется сборка значения строки lpSubBlock с использованием в качестве ключа параметра AKey.

Получение номеров версий

Методы GetFileVersion() и GetProductVersion() иллюстрируют способ получения номеров версий файла и продукта.

Структура FixedFileInfoBuf содержит поля, отведенные для номера версии как самого файла, так и номера версии продукта, с которым данный файл может распространяться. Эти номера версий хранятся в виде 64-разрядных чисел. Старшие и младшие 32-разрядные части этих чисел считываются по отдельности, и при этом используются различные поля.

Двоичный номер версии файла хранится в полях dwFileVersionMS и dwFileVersionLS, а номер версии продукта, с которым распространяется данный файл, — в полях dwProductVersionMS и dwProductVersionLS.

Строковое представление номера версии для данного файла возвращают методы GetFileVersion() и GetProductVersion(), использующие для соответствующего форматирования строки вспомогательную функцию VersionString().

Получение информации об операционной системе

На примере метода GetFileOS() иллюстрируется способ определения того, для какой операционной системы разработан данный файл. Это выполняется путем анализа поля dwFileOS структуры FixedFileInfoBuf. Дополнительную информацию о различных значениях поля dwFileOS можно получить, обратившись к интерактивной справочной системе Win32 API для структуры VS_FIXEDFILEINFO.

Использование класса TVerInfoRes

Для иллюстрации использования класса TVerInfoRes был создан проект VerInfo.dpr. Исходный код главной формы этого проекта представлен в листинге 12.17.

Листинг 12.17. Исходный код главной формы проекта, демонстрирующего получение информации о версии

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, FileCtrl, StdCtrls, verinfo, Grids, Outline,
  DirOutln, ComCtrls;

type
  TMainForm = class(TForm)
    lvVersionInfo: TListView;
    btnClose: TButton;
    procedure FormDestroy(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure FormCreate(Sender: TObject);
```

```
    procedure btnCloseClick(Sender: TObject);
  private
    VerInfoRes: TVerInfoRes;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure AddListViewItem(const aCaption, aValue: String;
  aData: Pointer; aLV: TListView);
{ Этот метод используется для добавления элемента TListItem
  в список aLV типа TListView. }
var
  NewItem: TListItem;
begin
  NewItem := aLV.Items.Add;
  NewItem.Caption := aCaption;
  NewItem.Data := aData;
  NewItem.SubItems.Add(aValue);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  VerInfoRes := TVerInfoRes.Create(Application.ExeName);
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  VerInfoRes.Free;
end;

procedure TMainForm.FormShow(Sender: TObject);
var
  VerString: String;
  i: integer;
  sFFlags: String;

begin
  for i := ord(viCompanyName) to ord(viComments) do begin
    VerString := VerInfoRes.GetPreDefKeyString(TVerInfoType(i));
    if VerString <> '' then
      AddListViewItem(VerNameArray[TVerInfoType(i)], VerString, nil,
        lvVersionInfo);
  end;
  VerString := VerInfoRes.GetUserDefKeyString('Author');
  if VerString <> EmptyStr then
```

```
      AddListViewItem('Author', VerString, nil, lvVersionInfo); // Автор
  AddListViewItem('File Version', VerInfoRes.FileVersion, nil,
    lvVersionInfo); // Версия файла
  AddListViewItem('Product Version', VerInfoRes.ProductVersion, nil,
    lvVersionInfo); // Версия продукта
  for i := 0 to VerInfoRes.FileFlags.Count - 1 do begin
    if i <> 0 then
      sFFlags := SFFlags+', ';
    sFFlags := SFFlags+VerInfoRes.FileFlags[i];
  end;
  AddListViewItem('File Flags',SFFlags, nil, lvVersionInfo); // Признаки файла
  AddListViewItem('Operating System', VerINfoRes.FileOS, nil, lvVersionInfo);
    // Операционная система
end;

procedure TMainForm.btnCloseClick(Sender: TObject);
begin
  Close;
end;

end.
```

Текст этой программы не должен вызывать затруднений. Она просто выполняет отображение информации о версии для самой себя. На рис. 12.5 показан результат запуска этого проекта — отображение соответствующей информации.

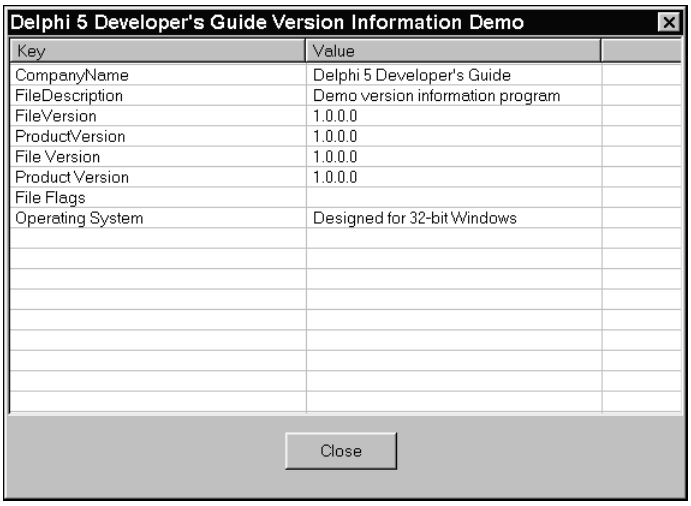


Рис. 12.5. Информация о версии демонстрационного приложения

Использование функции SHFileOperation()

Весьма полезной функцией Win32 API является `SHFileOperation()`. Для выполнения операций копирования, перемещения, переименования или удаления любых объектов файловых систем (как файлов, так и каталогов) она использует структуру `SHFILEOPSTRUCT`. Подробное описание этой

структуры имеется в интерактивной справочной системе Win32 API, поэтому мы не будет дублировать приведенную в ней информацию. Полезнее будет привести примеры некоторых нужных и широко используемых методов применения этой структуры для копирования всего каталога в иное местоположение, а также для удаления файла с перемещением его в корзину Windows 95/98.

Копирование каталога

В листинге 12.18 приведен текст процедуры копирования дерева каталогов из одного местоположения в другое.

Листинг 12.18. Процедура копирования каталога `CopyDirectoryTree()`

```
procedure CopyDirectoryTree(AHandle: THandle;
  const AFromDirectory, AToDirectory: String);
var
  SHFileOpStruct: TSHFileOpStruct;
  FromDir: PChar;
  ToDir: PChar;
begin

  GetMem(FromDir, Length(AFromDirectory)+2);
  try
    GetMem(ToDir, Length(AToDirectory)+2);
    try

      FillChar(FromDir^, Length(AFromDirectory)+2, 0);
      FillChar(ToDir^, Length(AToDirectory)+2, 0);

      StrCopy(FromDir, PChar(AFromDirectory));
      StrCopy(ToDir, PChar(AToDirectory));

      with SHFileOpStruct do
      begin
        Wnd    := AHandle;   // Назначаем дескриптор окна
        wFunc  := FO_COPY;  // Указываем копируемый файл
        pFrom  := FromDir;
        pTo    := ToDir;
        fFlags := FOF_NOCONFIRMATION or FOF_RENAMEONCOLLISION;
        fAnyOperationsAborted := False;
        hNameMappings := nil;
        lpszProgressTitle := nil;
        if SHFileOperation(SHFileOpStruct) <> 0 then
          RaiseLastWin32Error;
      end;
    finally
      FreeMem(ToDir, Length(AToDirectory)+2);
    end;
  finally
    FreeMem(FromDir, Length(AFromDirectory)+2);
  end;
end;
```

В процедуру CopyDirectoryTree() передается три параметра. Первый, AHandle, является дескриптором владельца диалогового окна, которое будет отображать различную информацию о ходе выполнения операции копирования. Оставшихся два параметра задают исходное и конечное расположение копируемого дерева каталогов. Поскольку все функции Windows API работают со строками типа PChar, достаточно просто скопировать сведения о местоположении каталогов в две переменные типа PChar, предварительно выделив им память. Полученные значения назначаются в качестве членов структуры SHFileOpStruct. Обратите внимание на то, что члену wFunc структуры присваивается значение FO_COPY, указывающее, что требуется выполнить операцию копирования. Назначение всех остальных членов структуры можно узнать из интерактивной справочной системы. По завершении выполнения функции SHFileOperation() исходное дерево каталогов будет скопировано в местоположение, определяемое параметром AToDirectory.

Перемещение файлов и каталогов в корзину Windows

В листинге 12.19 приведен пример аналогичной подпрограммы, но на этот раз выполняется перемещение файлов в корзину Windows.

Листинг 12.19. Процедура ToRecycle(), выполняющая копирование дерева каталогов в корзину Windows

```
procedure ToRecycle(AHandle: THandle; const ADirName: String);
var
  SHFileOpStruct: TSHFileOpStruct;
  DirName: PChar;
  BufferSize: Cardinal;
begin
  BufferSize := Length(ADirName) +1 +1;
  GetMem(DirName, BufferSize);
  try
    FillChar(DirName^, BufferSize, 0);
    StrCopy(DirName, PChar(ADirName));

    with SHFileOpStruct do
    begin
      Wnd := AHandle;
      wFunc := FO_DELETE;
      pFrom := DirName;
      pTo := nil;
      fFlags := FOF_ALLOWUNDO;
      fAnyOperationsAborted := False;
      hNameMappings := nil;
      lpszProgressTitle := nil;
    end;

    if SHFileOperation(SHFileOpStruct) <> 0 then
      RaiseLastWin32Error;
  finally
    FreeMem(DirName, BufferSize);
  end;
end;
```

Легко заметить, что отличия этой процедуры от предыдущей совсем невелики — члену `wFunc` присваивается значение `FO_DELETE`, а члену `pTo` — значение `nil`. При выполнении операций удаления функция `SHFileOperation()` игнорирует значение члена `pTo`. Кроме того, поскольку к значению члена `fFlags` добавлен флажок `FOF_ALLOWUNDO`, функция реально выполнит не обычное удаление файлов, а перемещение их в корзину Windows, что позволит восстановить их в случае необходимости.

Примеры использования обеих этих операций можно найти в проекте `SHFileOp.dpr`, присутствующем на компакт-диске, прилагаемом ко второму тому (см. также `www.williamspublishing.com`.).

Резюме

В этой главе содержится информация, которой вполне достаточно для работы с файлами, каталогами и устройствами внешней памяти (дисками). Во-первых, мы рассмотрели способы обработки различных типов файлов, что подготовило вас к самостоятельному созданию потомка класса Delphi `TFileStream`, в котором инкапсулированы операции ввода-вывода для работы с файлами, состоящими из записей. Во-вторых, мы рассказали об использовании отображенных в память файлов, которые являются мощным средством Win32. Для инкапсуляции функций отображения в память был создан класс `TMemMapFile`. И, наконец, продемонстрировали возможность извлечения информации о версии из файлов, которые содержат подобные сведения.

Глава

13

Дополнительный инструментарий разработчика

Рано или поздно наступает время, когда для решения некоторых неординарных задач приходится сворачивать с проторенного пути. В этой главе рассматриваются методы повышенной сложности, которые можно использовать в разнообразных приложениях Delphi. Эти методы потребуют от вас более тесного сотрудничества с интерфейсом Win32 API, что поможет разобраться в вопросах не совсем очевидных либо выходящих за рамки средств библиотеки VCL. Вы познакомитесь с оконными процедурами, экземплярами программ, ловушками Windows и совместной работой кода Delphi и C++.

Дополнительная обработка сообщений, поступающих в приложение

Как упоминалось ранее (см. главу 5, "Сообщения Windows"), процедура окна (window procedure) представляет собой функцию, которую Windows вызывает в том случае, когда окно получает сообщение. Поскольку объект `Application` содержит окно, он обладает и собственной процедурой окна, которая вызывается для получения всех сообщений, посланных данному приложению. Класс `TApplication` имеет даже собственное событие `OnMessage`, уведомляющее о приходе одного из этих сообщений.

Все это прекрасно, но только на первый взгляд…

Событие `TApplication.OnMessage` происходит только тогда, когда сообщение выбирается из очереди сообщений данного приложения (согласно терминологии, используемой в главе 5, "Сообщения Windows"). Сообщения, стоящие в очереди приложения, обычно либо имеют отношение к управлению окном (например, `WM_PAINT` и `WM_SIZE`), либо посланы окну с помощью таких функций API, как `PostMessage()`, `PostAppMessage()` или `BroadcastSystemMessage()`. Однако процедуре окна могут быть посланы и другие типы сообщений от системы Windows или от приложений с помощью функции `SendMessage()`. Вот тут-то и возникают проблемы. В этом случае событие `TApplication.OnMessage` не возникает, а следовательно, оно не может использоваться для того, чтобы узнать, пришло подобное сообщение или нет.

Подмена окна

Чтобы узнать, когда сообщение поступило в ваше приложение, нужно заменить процедуру окна объекта `Application` своей собственной. Именно здесь следует выполнить всю необходимую обработку полученного сообщения, причем до отправки этого сообщения исходной процедуре окна. Этот процесс называется подменой окна (subclassing).

Чтобы установить для окна новую функцию, можно воспользоваться функцией Win32 API `SetWindowLong()` с константой `GWL_WNDPROC`. Сама по себе функция для процедуры окна может иметь один из двух форматов. В первом случае она должна соответствовать определению процедуры окна Win32 API. Во втором случае можно воспользоваться преимуществами некоторых вспомогательных функций Delphi и оформить процедуру окна в виде специального метода, называемого методом окна (window method).

При подмене процедуры окна библиотеки VCL может возникнуть проблема. Есть вероятность того, что будет переопределен дескриптор этого окна, что приведет к сбою в работе приложения. Поэтому следует избегать применения этого метода, если существует вероятность переопределения дескриптора окна, для которого выполняется подмена. Более безопасным является использование метода `Application.HookMainWindow()`, описанного ниже в этой главе.

Процедура окна Win32 API

Процедура окна Win32 API должна иметь следующее объявление:

```
function AWndProc(Handle: hWnd; Msg, wParam, lParam: Longint):
  Longint; stdcall;
```

Параметр `Handle` идентифицирует окно приемника, параметр `Msg` представляет собой сообщение окна, а параметры `wParam` и `lParam` содержат дополнительную информацию, связанную с сообщением. Эта функция возвращает значение, которое зависит от полученного сообщения. Особое внимание обратите на то, что эта функция должна использовать спецификатор `stdcall`, обозначающий тип соглашения о вызове, определяющего способ передачи параметров.

Функцию `SetWindowLong()` можно использовать для установки процедуры окна объекта `Application`, как показано в следующем примере:

```
var
  WProc: Pointer;
begin
  WProc := Pointer(SetWindowLong(Application.Handle, GWL_WNDPROC,
    Integer(@NewWndProc)));
```

После этого вызова переменная `WProc` будет содержать указатель на старую процедуру окна. Это значение обязательно нужно сохранить, поскольку те сообщения, которые не нуждаются в обработке, должны передаваться старой процедуре окна с помощью функции Win32 API `CallWindowProc()`. Следующий код предлагает общую идею реализации процедуры окна:

```
function NewWndProc(Handle: hWnd; Msg, wParam, lParam: Longint):
  Longint; stdcall;
begin
  { Проверяем значение параметра сообщения Msg и в зависимости от
    результатов этой проверки выполняем соответствующие действия.
    Для сообщений, не требующих обработки, нужно передать их
    данные оригинальной процедуре окна, как показано ниже. }
  Result := CallWindowProc(WProc, Application.Handle, Msg, wParam, lParam);
end;
```

В листинге 13.1 представлено содержимое модуля `ScWndPrc.pas`, в котором выполняется подмена процедуры окна объекта `Application` с целью обработки определенного пользователем сообщения с именем `DDGM_FOOMSG`.

Листинг 13.1. Модуль `ScWndPrc.pas`

```
unit ScWndPrc;

interface

uses Forms, Messages;

const
  DDGM_FOOMSG = WM_USER;

implementation
```

```
uses Windows, SysUtils, Dialogs;
var
  WProc: Pointer;

function NewWndProc(Handle: hWnd; Msg, wParam, lParam: Longint): Longint;
stdcall;
  { Это процедура окна уровня Win32 API. Она обрабатывает
    сообщения, полученные окном объекта Application. }
begin
  if Msg = DDGM_FOOMSG then
    { Если это – определенное пользователем сообщение,
      то предупреждаем пользователя }
    ShowMessage(Format('Message seen by WndProc! Value is: $%x', [Msg]));
  { Передаем сообщение старой процедуре окна. }
  Result := CallWindowProc(WProc, Handle, Msg, wParam, lParam);
end;

initialization
  { Устанавливаем процедуру окна для окна объекта Application. }
  WProc := Pointer(SetWindowLong(Application.Handle, gwl_WndProc,
    Integer(@NewWndProc)));
end.
```

Не забудьте сохранить старую процедуру окна, возвращаемую функцией `GetWindowLong()`. Если не организовать вызов прежней процедуры окна из процедуры подмены для **всех** сообщений, не подлежащих собственной обработке, это, по всей вероятности, приведет к аварийному отказу приложения, а возможно, и самой операционной системы.

Метод окна в варианте Delphi

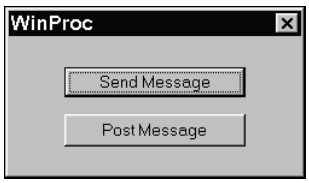
В Delphi предусмотрена функция `MakeObjectInstance()`, которая "наводит мосты" между процедурой окна Win32 API и методом Delphi. С помощью этой функции можно создать метод с типом `TWndMethod`, который будет выполнять функции процедуры окна. Функция `MakeObjectInstance()` объявляется в модуле `Forms` следующим образом:

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

Определение типа `TWndMethod` также содержится в модуле `Forms` и имеет вид

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

Значение, возвращаемое функцией `MakeObjectInstance()`, является указателем (с типом `Pointer`) на адрес вновь созданной процедуры окна. Именно это значение и передается в качестве последнего параметра функции `SetWindowLong()`. Для освобождения любых методов окон, создаваемых с помощью функции `MakeObjectInstance()`, следует использовать функцию `FreeObjectInstance()`.



В проекте WinProc.dpr иллюстрируются способы подмены процедуры окна объекта Application и их преимущества перед использованием события Application.OnMessage. Главная форма этого проекта показана на рис. 13.1.

Рис. 13.1. Главная форма проекта WinProc.dpr

В листинге 13.2 представлен исходный текст модуля Main.pas — главного модуля проекта WinProc.dpr.

Листинг 13.2. Исходный текст модуля Main.pas

```
unit Main;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    SendBtn: TButton;
    PostBtn: TButton;
    procedure SendBtnClick(Sender: TObject);
    procedure PostBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    OldWndProc: Pointer;
    WndProcPtr: Pointer;
    procedure WndMethod(var Msg: TMessage);
    procedure HandleAppMessage(var Msg: TMsg; var Handled: Boolean);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses ScWndPrc;

procedure TMainForm.HandleAppMessage(var Msg: TMsg; var Handled: Boolean);
{ Обработчик события OnMessage для объекта Application. }
begin
  if Msg.Message = DDGM_FOOMSG then
    { Если это сообщение определено пользователем,
      ставим его в известность. }
```

```
    ShowMessage(Format('Message seen by OnMessage! Value is: $%x',
      [Msg.Message]));
    { Сообщение воспринято событием OnMessage. }
end;

procedure TMainForm.WndMethod(var Msg: TMessage);
begin
  if Msg.Msg = DDGM_FOOMSG then
    { Если это сообщение определено пользователем,
      ставим его в известность. }
    ShowMessage(Format('Message seen by WndMethod! Value is: $%x',
      [Msg.Msg]));
    { Сообщение воспринято методом окна. }
  with Msg do
    { Передаем сообщение старой процедуре окна. }
    Result := CallWindowProc(OldWndProc, Application.Handle, Msg,
      wParam, lParam);
end;

procedure TMainForm.SendBtnClick(Sender: TObject);
begin
  SendMessage(Application.Handle, DDGM_FOOMSG, 0, 0);
end;

procedure TMainForm.PostBtnClick(Sender: TObject);
begin
  PostMessage(Application.Handle, DDGM_FOOMSG, 0, 0);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnMessage := HandleAppMessage;
    { Установка обработчика события OnMessage. }
  WndProcPtr := MakeObjectInstance(WndMethod);
    { Создание процедуры окна. Устанавливаем процедуру окна
      для окна объекта Application. }
  OldWndProc := Pointer(SetWindowLong(Application.Handle, GWL_WNDPROC,
    Integer(WndProcPtr)));
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  { Восстанавливаем старую процедуру окна для окна
    объекта Application. }
  SetWindowLong(Application.Handle, GWL_WNDPROC, Longint(OldWndProc));
  { Освобождаем созданную пользователем процедуру окна. }
  FreeObjectInstance(WndProcPtr);
end;

end.
```

По щелчку на кнопке SendBtn дескриптору окна объекта Application отправляется сообщение DDGM_FOOMSG с использованием функции API SendMessage(), а по щелчку на кнопке PostBtn то же самое сообщение посылается объекту Application с использованием функции API PostMessage().

Для обработки события Application.OnMessage используется процедура HandleAppMessage(), в которой с помощью функции ShowMessage() открывается диалоговое окно, уведомляющее о получении сообщения. Обработчик события OnMessage назначается в обработчике события OnCreate главной формы.

Обратите внимание на то, что обработчик события OnDestroy главной формы возвращает оригинальное значение (OldWndProc) процедуры окна объекта Application, и это делается перед вызовом функции FreeObjectInstance(), предназначенной для освобождения процедуры, созданной с помощью функции MakeProcInstance(). Если предварительно не вернуть первоначальную установку старой процедуры окна, то освобождение процедуры подмены полностью лишит окно способности обрабатывать сообщения, что, в свою очередь, может послужить причиной разрушения приложения и всей операционной системы.

В модуль Main также включен модуль ScWndPrc, приведенный выше в этой главе. Это значит, что подмена окна объекта Application будет выполнена дважды: один раз — модулем ScWndPrc с помощью API, а второй — модулем Main с помощью метода окна. И в этом нет ничего страшного, если, конечно, вы не забудете для передачи сообщений старым процедурам окна использовать функцию CallWindowProc() как в процедуре окна, так и в методе окна.

Запустив это приложение, вы увидите, что диалоговое окно, обязанное своим появлением на экране функции ShowMessage(), открывается при работе как процедуры окна, так и метода окна, т.е. при нажатии любой кнопки. Но при этом событие Application.OnMessage может заметить только те сообщения, которые отправлены окну с помощью кнопки Post Message.

Метод HookMainWindow()

Существует другой, пожалуй, более дружественный по отношению к подпрограммам библиотеки VCL способ перехвата сообщений, предназначенных для окна объекта Application. Речь идет о методе HookMainWindow(), принадлежащем классу TApplication. С его помощью можно вставить собственный обработчик сообщений в начало метода WndProc(). Это позволит выполнять специальную обработку сообщений или организовать защиту класса TApplication от обработки определенных сообщений. Метод HookMainWindow() определяется следующим образом:

```
procedure HookMainWindow(Hook: TWindowHook);
```

Единственный параметр этого метода имеет тип TWindowHook, который определяется следующим образом:

```
type
  TWindowHook = function (var Message: TMessage): Boolean of object;
```

Этот метод не нуждается в пространных пояснениях: просто вызовите его, передав в качестве параметра Hook собственный метод, — и ваш метод будет добавлен в список методов перехвата окна, которые будут вызваны до нормальной обработки сообщений, выполняющейся в методе TApplication.WndProc(). Если метод перехвата окна вернет значение, равное True, то сообщение считается обработанным и метод WndProc() будет немедленно завершен.

Завершив обработку сообщений, вызовите метод `UnhookMainWindow()` для удаления своего метода из списка методов перехвата окна. Он определяется уже известным вам способом:

```
procedure UnhookMainWindow(Hook: TWindowHook);
```

В листинге 13.3 представлен текст главной (и единственной) формы простого проекта VCL, в котором реализуется описанный выше способ обработки сообщений, а на рис. 13.2 это приложение показано в процессе работы.

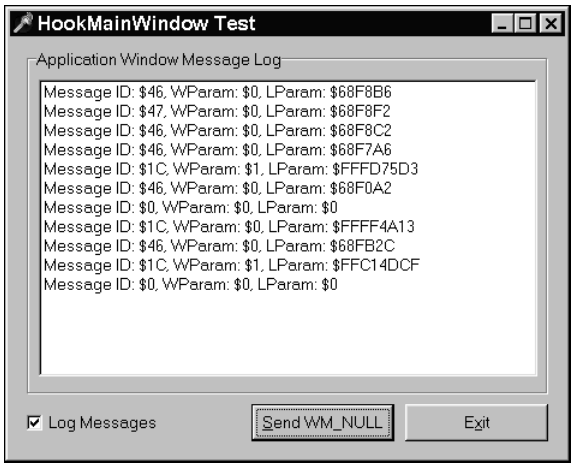


Рис. 13.2. Контроль за работой объекта `Application` в окне проекта `HookWnd`

Листинг 13.3. Модуль `Main.pas` проекта `HookWnd`

```
unit HookMain;

interface

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

type
  THookForm = class(TForm)
    SendBtn: TButton;
    GroupBox1: TGroupBox;
    LogList: TListBox;
    DoLog: TCheckBox;
    ExitBtn: TButton;
    procedure SendBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
  private
```

```
    function AppWindowHook(var Message: TMessage): Boolean;
  end;

var
  HookForm: THookForm;

implementation

{$R *.DFM}

procedure THookForm.FormCreate(Sender: TObject);
begin
  Application.HookMainWindow(AppWindowHook);
end;

procedure THookForm.FormDestroy(Sender: TObject);
begin
  Application.UnhookMainWindow(AppWindowHook);
end;

function THookForm.AppWindowHook(var Message: TMessage): Boolean;
const
  LogStr = 'Message ID: $%x, WParam: $%x, LParam: $%x';
begin
  Result := True;
  if DoLog.Checked then
    with Message do
      LogList.Items.Add(Format(LogStr, [Msg, WParam, LParam]));
end;

procedure THookForm.SendBtnClick(Sender: TObject);
begin
  SendMessage(Application.Handle, WM_NULL, 0, 0);
end;

procedure THookForm.ExitBtnClick(Sender: TObject);
begin
  Close;
end;

end.
```

Предотвращение запуска нескольких экземпляров приложения

Несколько экземпляров означает одновременный запуск более одной копии вашей программы. Возможность запуска нескольких экземпляров приложения независимо друг от друга обеспечивается операционной системой Win32. Хотя такая возможность может быть полез-

ной, существуют ситуации, в которых пользователю следует разрешить запуск только одной копии определенного приложения. В частности, это приложение может использовать уникальный ресурс компьютера, например модем или параллельный порт. В подобных ситуациях необходимо поместить в приложение некоторые подпрограммы, которые обеспечат решение указанных проблем, исключив возможность запуска более одной копии этого приложения в любой конкретный момент времени.

Для 16-разрядной Windows это совсем простая задача, поскольку для определения факта одновременной работы нескольких копий приложения может быть использована системная переменная `hPrevInst`. Если значение переменной `hPrevInst` не равно нулю, значит, активен еще один экземпляр данного приложения. Однако, как объяснялось в главе 3, "Win32 API", Win32 обеспечивает надежную изоляцию процессов друг от друга. Поэтому для приложений Win32 значение переменной `hPrevInst` всегда будет равно нулю.

Другой метод, который подходит как для 16-, так и для 32-разрядной Windows, заключается в использовании функции API `FindWindow()` для поиска уже активного окна объекта `Application`. Но это решение имеет два недостатка. Во-первых, функция `FindWindow()` позволяет найти окно только по имени класса или заголовка. Зависимость от имени класса не может служить надежным решением, поскольку нет гарантии, что имя класса вашей формы является уникальным в системе. А недостаток поиска, основанного на заголовке формы, заключается в том, что достаточно часто заголовок формы изменяется в процессе работы приложения (например, в Delphi или в Word). Во-вторых, функция `FindWindow()` имеет невысокое быстродействие, поскольку ей приходится перебирать все окна верхнего уровня.

Таким образом, оптимальное решение для Win32 состоит в использовании некоторых типов объектов API, которые могут быть доступны со стороны нескольких процессов. Как объяснялось в главе 11, "Создание многопоточных приложений", существует несколько типов объектов синхронизации потоков, которые являются доступными сразу нескольким процессам. Благодаря простоте использования, идеальным решением этой проблемы будет применение мьютекса.

При первом запуске приложения с помощью функции Win32 API `CreateMutex()` создается некоторый мьютекс. Параметр `lpName` этой функции содержит уникальный строковый идентификатор. Последующие экземпляры этого же приложения должны попытаться открыть мьютекс по имени, используя функцию `OpenMutex()`, которая успешно выполнится только в том случае, если мьютекс уже был создан с помощью функции `CreateMutex()`.

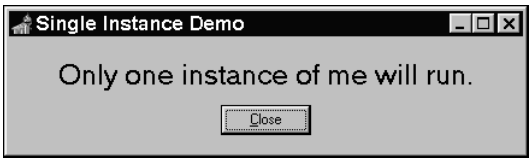


Рис. 13.3. Главная форма проекта `OneInst`

Кроме того, при попытке запустить второй экземпляр приложения, следует передать фокус ввода первому экземпляру этого приложения. Для этого проще всего использовать зарегистрированное с помощью функции `RegisterWindowMessage()` сообщение окна, используемое для данного приложения как уникальный идентификатор сообщения. Затем можно заставить первоначальный экземпляр приложения ответить на это сообщение путем возврата идентификатора его главного окна, который и будет использован вторым экземпляром приложения для передачи фокуса. Описанный подход иллюстрируется в листинге 13.4, содержащем исходный код модуля `MultInst.pas`; в листинге 13.5 представлен код модуля `OIMain.pas`, который является главным модулем проекта `OneInst`. Вид формы этого приложения показан на рис. 13.3.

Листинг 13.4. Модуль `MultInst.pas`

```
unit MultInst;

interface

const
  MI_QUERYWINDOWHANDLE   = 1;
  MI_RESPONDWINDOWHANDLE = 2;

  MI_ERROR_NONE          = 0;
  MI_ERROR_FAILSUBCLASS  = 1;
  MI_ERROR_CREATINGMUTEX = 2;

// Вызов этой функции для определения наличия ошибки при запуске.
// Значением будет одно или более флагов ошибки MI_ERROR_*
function GetMIError: Integer

implementation

uses Forms, Windows, SysUtils;

const
  UniqueAppStr = 'DDG.I_am_the_Eggman!';

var
  MessageId: Integer;
  WProc: TFNWndProc;
  MutHandle: THandle;
  MIError: Integer;

function GetMIError: Integer;
begin
  Result := MIError;
end;

function NewWndProc(Handle: HWND; Msg: Integer; wParam, lParam: Longint):
  Longint; stdcall;
begin
  Result :=0;
  // Если это - зарегистрированное сообщение...
  if Msg = MessageID then
  begin
  case wParam of
    MI_QUERYWINDOWHANDLE:
      { Новый экземпляр приложения запрашивает дескриптор главного
        окна с целью передачи ему фокуса, поэтому требуется привести
        окно приложения в нормальное состояние и послать ответное
        сообщение с дескриптором главного окна }
        begin
          if IsIconic(Application.Handle) then
          begin
```

```
            Application.MainForm.WindowState := wsNormal;
            Application.Restore;
          end;
          PostMessage(HWND(lParam), MessageID, MI_RESPONDWINDOWHANDLE,
            Application.MainForm.Handle);
        end
      MI_RESPONDWINDOWHANDLE:
        { Выполняемый экземпляр приложения возвратил дескриптор своего
          главного окна, поэтому требуется передать ему фокус и
          завершить работу данного экземпляра. }
        begin
          SetForegroundWindow(HWND(lParam));
          Application.Terminate;
        end;
    end;
  end

  { В противном случае передаем сообщение старой процедуре окна. }
  else
    Result := CallWindowProc(WProc, Handle, Msg, wParam, lParam);
end;

procedure SubClassApplication;
begin
  { Подменяем процедуру окна объекта Application, чтобы событие
    Application.OnMessage оставалось доступным пользователю. }
  WProc := TFNWndProc(SetWindowLong(Application.Handle, GWL_WNDPROC,
    Longint(@NewWndProc)));
  { Устанавливаем признак ошибки, если возникло условие ошибки. }
  if WProc = Nil then
    MIError := MIError or MI_FAIL_SUBCLASS;
end;

procedure DoFirstInstance;
{ Эта процедура вызывается только для первого экземпляра приложения }
begin
  // Создание мьютекса с помощью уникальной (предположительно) строки
  MutHandle := CreateMutex(nil, False, UniqueAppStr);
  if MutHandle = 0 then
    MIError := MIError or MI_ERROR_CREATINGMUTEX;
end;

procedure BroadcastFocusMessage;
{ Эта процедура вызывается, когда уже есть работающий экземпляр приложения }
var
  BSMRecipients: DWORD;
begin
  // Предотвращение мерцания главной формы
  Application.ShowMainForm := False;
  // Посылка сообщения с целью установки диалога с предыдущим
  // экземпляром приложения
```

```
  BSMRecipients := BSM_APPLICATIONS;
  BroadCastSystemMessage(BSF_IGNORECURRENTTASK or BSF_POSTMESSAGE,
    @BSMRecipients, MessageID, MI_QUERYWINDOWHANDLE, Application.Handle);
end;

procedure InitInstance;
begin
  SubClassApplication;  // Замена процедуры окна приложения
  MutHandle := CreateMutex(Nil, False, UniqueAppStr);
  if MutHandle = 0 then
    // Объект мьютекса еще не был создан, т.е. предыдущего
    // экземпляра данного приложения не существует
    DoFirstInstance
  else
    BroadcastFocusMessage;
end;

 initialization
  MessageID := RegisterWindowMessage(UniqueAppStr);
  InitInstance;
finalization
  if WProc <> Nil then
    { Восстанавливаем старую процедуру окна. }
    SetWindowLong(Application.Handle, GWL_WNDPROC, LongInt(WProc));
  if MutHandle <> 0 then CloseHandle(MutHandle);  // Освобождение мьютекса
end.
```

Листинг 13.5. Модуль `OIMain.pas`

```
unit OIMain;

interface

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

type
  TMainForm = class(TForm)
    Label1: TLabel;
    CloseBtn: TButton;
    procedure CloseBtnClick(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;
```

```
implementation

uses MultInst;

{$R *.DFM}

procedure TMainForm.CloseBtnClick(Sender: TObject);
begin
  Close;
end;

end.
```

Использование BASM с Delphi

Поскольку Delphi представляет собой настоящий компилятор, то одним из положительных следствий этого факта является возможность вставлять ассемблерный код прямо в процедуры и функции Object Pascal. Реализация этой возможности облегчается благодаря наличию встроенного в среду Delphi транслятора с языка ассемблера — BASM. Прежде чем познакомиться с BASM, целесообразно выяснить, в каких случаях следует использовать язык ассемблера в программах Delphi. Безусловно, иметь в своем распоряжении столь мощный инструмент очень удобно, но, как и в любом другом случае, использование BSAM имеет свою оборотную сторону. Если вы будете постоянно следовать приведенным ниже рекомендациям, то это сделает ваши программы лучше, яснее и компактнее.

- Никогда не используйте ассемблер для реализации того, что можно выполнить с помощью Object Pascal. Например, не стоит писать ассемблерные процедуры для связи с внешними устройствами через последовательные порты, поскольку в интерфейсе Win32 API предусмотрены встроенные функции соответствующего назначения.
- Не следует чрезмерно увлекаться оптимизацией программ за счет добавления в них ассемблерных вставок. Оптимизированный вручную ассемблерный код может работать быстрее, чем код Object Pascal, но при этом в жертву приносится читабельность и удобство сопровождения программ. Кроме того, используемые Delphi алгоритмы весьма эффективны, и после всех ассемблерных усилий вы можете вдруг с удивлением обнаружить, что оптимизирующий компилятор Delphi создает код, который работает быстрее, чем ассемблерный код, написанный вами вручную.
- Всегда сопровождайте ассемблерный код подробными комментариями. В будущем вам (или другому программисту), скорее всего, придется вновь обратится к этому коду, и отсутствие комментариев может затруднить его понимание.
- Не используйте BASM для доступа к аппаратным средствам компьютера. Несмотря на то что Windows 95/98 в большинстве случаев такие вольности прощает, Windows NT/2000 — никогда.
- По возможности оформляйте ассемблерный код в виде процедур или функций, вызываемых из Object Pascal. Это облегчит не только будущую поддержку и развитие вашего продукта, но во многом упростит и переход к другим платформам в случае необходимости.

На заметку

В этом разделе нет учебного материала по программированию на языке ассемблера. Однако в нем демонстрируются приемы связи ассемблера с Delphi, что может оказаться полезным для вас, если вы уже знакомы с этим языком.

Кроме того, если вам приходилось программировать на BASM в среде Delphi 1, имейте в виду, что BASM в 32-разрядной Delphi — это совсем другой язык, т.е. совершенно новое поле деятельности. Скорее всего, чтобы 16-разрядный код BASM соответствовал требованиям новой платформы, вам придется переписать его практически заново. А поскольку поддержка кода BASM может потребовать от вас слишком большого внимания, то этот факт — еще одна веская причина свести к минимуму использование BASM в создаваемых приложениях.

Как работает BASM

Вставить ассемблерный код в приложения Delphi гораздо легче, чем можно себе представить. От вас требуется только поместить его между ключевыми словами `asm` и `end`. Подключение ассемблерного кода демонстрируется на примере следующего программного фрагмента:

```
var
  i: integer;
begin
  i := 0;
  asm
    mov eax, i
    inc eax
    mov i, eax
  end;
  { Значение переменной i увеличено на единицу}
```

В этом фрагменте кода объявляется переменная `i`, которая инициализируется нулем. Затем значение переменной `i` пересылается в регистр `eax`, содержимое которого увеличивается на единицу, и его новое содержимое опять передается переменной `i`. Этот пример демонстрирует не только простоту использования BASM, но также и простоту доступа к переменным Object Pascal из BASM (в данном случае имеется в виду переменная `i`).

Доступ к параметрам

На примере следующего кода вы убедитесь, что простота доступа к переменным, объявленным в процедуре глобально и локально, свойственна также переменным, передаваемым процедуре в качестве параметров:

```
procedure Foo(I: integer);
begin
  { некоторые действия }
  asm
    mov eax, I
    inc eax
    mov I, eax
  end;
  { I увеличено на 1 }
  { дальнейшие действия }
end;
```

Возможность доступа к параметру по имени весьма удобна, поскольку в этом случае не приходится ссылаться на передаваемые процедуре переменные через регистр базового указателя стека (ebp), как в обычной ассемблерной программе. В этой программе пришлось бы обращаться к переменной i как к содержимому ячейки [ebp+4], т.е. с учетом ее смещения относительно базового указателя стека.

На заметку

Используя BASM для ссылки на параметры, передаваемые процедуре, помните, что доступ к этим параметрам можно получить по имени, а не только по смещению относительно регистра ebp. Доступ по смещению затруднит сопровождение и последующее развитие ваших программ.

Передача параметров по ссылке

Напомним, что если параметр (в списке параметров процедуры или функции) объявляется с помощью ключевого слова var, то вместо значения переменной передается указатель на эту переменную. Это значит, что при выполнении ссылок на var-параметры внутри блока BASM параметр будет представлять собой 32-разрядный указатель на переменную, а не собственно экземпляр переменной. В следующем примере показано, как можно обратиться к переменной i, если она передается в качестве var-параметра:

```
procedure Foo(var I: integer);
begin
  { Некоторые действия }
  asm
    mov eax, I
    inc dword ptr [eax]
  end;
  { Переменная I была увеличена на единицу }
  { Дальнейшие действия }
end;
```

Регистровое соглашение о вызове

Напомним, что соглашение о вызове, действующее по умолчанию в отношении функций и процедур языка Object Pascal, является регистровым. Использование преимуществ этого метода передачи параметров может помочь в оптимизации кода. Согласно регистровому соглашению о вызове, первых три 32-разрядных параметра передаются в регистры eax, edx и ecx. Следовательно, для объявления функции

```
function BlahBlah(I1, I2, I3: Integer): Integer;
```

можно рассчитывать на то, что значение переменной I1 будет передано в регистр eax, I2 — в edx, а I3 — в ecx. В качестве другого примера рассмотрим следующий метод:

```
procedure TSomeObject.SomeProc(S1, S2: PChar);
```

В этом случае значение S1 будет передано в регистр ecx, S2 — в edx, а неявно заданный параметр Self будет передан в регистр eax.

Процедуры, целиком написанные на языке ассемблера

В Object Pascal предусмотрена возможность создания процедур и функций, полностью написанных на языке ассемблера. Для этого достаточно начать такую процедуру или функцию с ключевого слова `asm`, а не со слова `begin`:

```
function IncAnInt(I: Integer): Integer;
asm
  mov eax, I
  inc eax
end;
```

На заметку Не стоит использовать директиву `assembler` Delphi 1. Эта директива просто игнорируется компилятором Delphi 5.

Предыдущая процедура в качестве параметра принимает переменную `i` и увеличивает ее. А поскольку это значение переменной находится в регистре `eax`, то именно оно и возвращается функцией. В табл. 13.1 показано, как в Delphi выполняется возврат различных типов данных.

Таблица 13.1. Возврат значений из функций Delphi

Возвращаемый тип	Метод возврата
`Char`, `Byte`	`al`
`SmallInt`, `Word`	`ax`
`Integer`, `LongWord`, `AnsiString`, `Pointer`, `Class`	`eax`
`Real48`	Регистр `eax` содержит указатель на данные в стеке
`Int64`	Пара регистров `edx:eax`
`Single`, `Double`, `Extended`, `Comp`	`ST(0)` в стеке сопроцессора

На заметку Значения типа `LongString` возвращаются в виде указателя на временную строку в стеке.

Записи

Язык BASM предоставляет превосходное средство доступа к полям записи, заключающееся в использовании синтаксиса `Register.Type.Field`. Рассмотрим, например, запись, определенную следующим образом:

```
type
  TDumbRec = record
    i: integer;
    c: char;
  end;
```

Функция, которая принимает запись TDumbRec в качестве параметра, передаваемого по ссылке, имеет вид:

```
procedure ManipulateRec(var DR: TDumbRec);
asm
  mov [eax].TDumbRec.i, 24
  mov [eax].TDumbRec.c, 's'
end;
```

Обратите внимание на синтаксис быстрого доступа к полям записи. Альтернативное решение для чтения или записи некоторого значения — вычисление вручную соответствующего смещения в записи. Используйте этот метод для работы с записями в BASM, и тогда ваш код BASM будет не слишком болезненно воспринимать какие-либо изменения типов данных.

Использование ловушек Windows

Ловушки (hook) Windows предоставляют программистам средства управления возникновением и обработкой системных событий. Благодаря ловушкам программисты получают, очевидно, наивысшую степень власти над системой, поскольку ловушка дает возможность просматривать и модифицировать системные события и сообщения, а также предотвращать их распространение по системе. (В русскоязычной литературе термин "hook" иногда переводится как "перехват". — Прим.ред.)

Установка ловушки

Ловушка Windows устанавливается с помощью функции Win32 API SetWindowsHookEx():

```
function SetWindowsHookEx(idHook: Integer; lpfn: TFNHookProc;
  hmod: HINST; dwThreadID: DWORD): HHOOK; stdcall;
```

В своих приложениях используйте только функцию SetWindowsHookEx(), а не SetWindowsHook(), которая существовала в Windows 3.x, но не реализована в Win32 API.

Параметр idHook описывает тип устанавливаемой ловушки. Он может быть определен с помощью одной из встроенных констант, перечисленных в табл. 13.2.

Таблица 13.2. Константы имен ловушек Windows

Константа	Описание
WH_CALLWNDPROC	Фильтр процедуры окна. Процедура ловушки вызывается в случае, если процедуре окна посылается сообщение
WH_CALLWNDPROCRET	Устанавливает процедуру ловушки, которая контролирует сообщения после их обработки процедурой окна приемника
WH_CBT	Тренировочный фильтр с ориентацией на конкретный компьютер. Процедура ловушки вызывается перед обработкой большинства сообщений окон, мыши и клавиатуры

Окончание табл. 13.2

Константа	Описание
WH_DEBUG	Фильтр отладки. Эта функция ловушки вызывается перед любой другой ловушкой Windows
WH_GETMESSAGE	Фильтр сообщений. Функция ловушки вызывается, когда из очереди приложения считывается любое сообщение
WH_HARDWARE	Фильтр сообщений оборудования. Функция ловушки вызывается, когда из очереди приложения считывается сообщение оборудования
WH_JOURNALPLAYBACK	Функция ловушки вызывается, когда из очереди системы считывается любое сообщение. Обычно используется для вставки в очередь системных событий
WH_JOURNALRECORD	Функция ловушки вызывается, когда из очереди системы запрашивается какое-нибудь событие. Обычно используется для "регистрации" системных событий
WH_KEYBOARD	Фильтр клавиатуры. Функция ловушки вызывается, когда из очереди приложения считывается сообщение wm_KeyDown или wm_KeyUp
WH_KEYBOARD_LL	Низкоуровневый фильтр клавиатуры
WH_MOUSE	Фильтр сообщений мыши. Функция ловушки вызывается, когда из очереди приложения считывается сообщение мыши
WH_MOUSE_LL	Низкоуровневый фильтр мыши
WH_MSGFILTER	Фильтр специального сообщения. Функция ловушки вызывается, когда сообщение должно быть обработано диалоговым окном приложения, меню или окном сообщения
WH_SHELL	Фильтр приложения-оболочки. Функция ловушки вызывается, когда создаются и разрушаются окна верхнего уровня, а также когда приложению-оболочке требуется стать активным

* Доступны только в среде Windows NT 4.0 и Windows 2000.

Параметр lpfn представляет собой адрес функции обратного вызова, действующей подобно функции ловушки Windows. Эта функция имеет тип TFNHookProc, который определяется следующим образом:

```
TFNHookProc = function (code: Integer; wparam: WPARAM; lparam: LPARAM):
   LRESULT stdcall;
```

Содержимое каждого из параметров функции ловушки Windows варьируется в зависимости от типа установленной ловушки. Сами параметры описаны в интерактивной справочной системе Win32 API.

Параметр hMod должен иметь значение hInstance в EXE- или DLL-файлах, содержащих функцию обратного вызова ловушки.

Параметр dwThreadID идентифицирует поток, с которым будет связана ловушка. Если он равен нулю, данная ловушка связывается со всеми потоками.

Возвращаемым значением является дескриптор ловушки, который необходимо сохранить в глобальной переменной (для использования в дальнейшем).

Windows может установить несколько ловушек одновременно. Более того, можно даже установить несколько ловушек одного и того же типа.

На действие некоторых ловушек накладывается ограничение, состоящее в том, что они должны выполняться из DLL. Об особенностях каждой отдельной ловушки можно узнать в документации Win32 API.

Одно серьезное ограничение для системных ловушек заключается в том, что новые экземпляры DLL с ловушками загружаются в адресное пространство каждого процесса отдельно, благодаря чему DLL-ловушки не могут непосредственно связываться с главным приложением, установившим ловушку. Поэтому для связи с главным приложением необходимо использовать сообщения или области разделяемой памяти (в виде файлов, отображенных в память, которые описаны в главе 12, "Работа с файлами").

Использование функции ловушки

Содержимое параметров функции ловушки `Code`, `wParam` и `lParam` варьируется в зависимости от типа установленной ловушки и описано в интерактивной справочной системе Win32 API. У всех этих параметров есть одно общее свойство: в зависимости от значения параметра `Code` на вас возлагается ответственность за вызов следующей ловушки в цепочке.

Для вызова следующей ловушки используйте функцию Win32 API `CallNextHookEx()`:

```
Result := CallNextHookEx(HookHandle, Code, wParam, lParam);
```

Вызывая следующую ловушку в цепочке, не обращайтесь к функции `DefHookProc()`. Это еще одна нереализованная функция Windows 3.x.

Всегда помните, что некоторые ловушки действуют только в случае их реализации из DLL. Обязательно уточняйте подобные детали по каждой отдельной ловушке в документации Win32 API.

Использование функции отмены ловушки

Для освобождения ловушки Windows используйте функцию `UnhookWindowsHookEx()`:

```
UnhookWindowsHookEx(HookHandle);
```

В качестве параметра ей следует передать дескриптор освобождаемой ловушки. И вновь предупреждаем: не вызывайте функцию `UnhookWindowsHook()`, поскольку она устарела.

Создание функции `SendKeys`: ловушка `JournalPlayback`

Если вы переходите к Delphi из такой среды, как Visual Basic или Paradox для Windows, то вам, вероятно, знакома функция `SendKeys()`. Она позволяет передавать в Windows строку символов, которые затем воспроизводятся так, как если бы вы вводили их с клавиатуры, причем все "нажатия клавиш" посылаются в активное окно. Поскольку в Delphi нет встроенной функции такого типа, ее создание служит наглядной демонстрацией возможности наращивания мощного потенциала Delphi, а также иллюстрацией реализации ловушки `JournalPlayback` в среде Delphi.

Определение, можно ли использовать ловушку JournalPlayback

Существует несколько причин, по которым ловушку следует считать лучшим способом имитации нажатий клавиш в приложении. Вы можете спросить: "А почему бы просто не отправить сообщения wm_KeyDown и wm_KeyUp?" Основная причина в том, что вы можете не знать дескриптор окна, которому хотите оправить сообщения, или же дескриптор этого окна может периодически изменяться. А без информации о дескрипторе окна об отправке сообщений не может быть и речи. Кроме того, некоторые приложения, помимо просмотра сообщений (для получения информации о нажатии клавиш), вызывают функции API, чтобы проверить состояние клавиатуры.

Как работает функция SendKeys

Объявление функции SendKeys() имеет следующий вид:

```
function SendKeys(S: String): TSendKeyError; export;
```

Возвращаемое значение имеет перечислимый тип TSendKeyError, в котором определяются условия возникновения ошибки. Эта функция возвращает одно из значений, описанных в табл. 13.3.

Таблица 13.3. Коды ошибок функции SendKey()

Значение	Описание
sk_None	Функция выполнена успешно
sk_FailSetHook	Ловушку Windows установить не удалось
sk_InvalidToken	В строке обнаружена неверная лексема
sk_UnknownError	Возникла фатальная ошибка неизвестного происхождения
sk_AlreadyPlaying	Ловушка в данный момент активна, и нажатия клавиш уже воспроизводятся

Параметр S может включать любой алфавитно-цифровой символ или символ @ для обозначения клавиши <Alt>, символ ^ — для обозначения клавиши <Ctrl> и символ ~ — для обозначения клавиши <Shift>. Функция SendKeys() также позволяет задавать специальные клавиши клавиатуры в фигурных скобках, как показано в тексте модуле KeyDefs.pas, представленном в листинге 13.6.

Листинг 13.6. Модуль KeyDefs.pas определения специальных клавиш для функции SendKeys()

```
unit KeyDefs;

interface

uses Windows;

const
  MaxKeys       = 24;
  ControlKey    = '^';
  AltKey        = '@';
  ShiftKey      = '~';
  KeyGroupOpen  = '{ ';
```

```
  KeyGroupClose = '} ';

type
  TKeyString = String[7];

  TKeyDef = record
    Key: TKeyString;
    vkCode: Byte;
  end;

const
  KeyDefArray : array[1..MaxKeys] of TKeyDef = (
    (Key: 'F1';     vkCode: vk_F1),
    (Key: 'F2';     vkCode: vk_F2),
    (Key: 'F3';     vkCode: vk_F3),
    (Key: 'F4';     vkCode: vk_F4),
    (Key: 'F5';     vkCode: vk_F5),
    (Key: 'F6';     vkCode: vk_F6),
    (Key: 'F7';     vkCode: vk_F7),
    (Key: 'F8';     vkCode: vk_F8),
    (Key: 'F9';     vkCode: vk_F9),
    (Key: 'F10';    vkCode: vk_F10),
    (Key: 'F11';    vkCode: vk_F11),
    (Key: 'F12';    vkCode: vk_F12),
    (Key: 'INSERT'; vkCode: vk_Insert),
    (Key: 'DELETE'; vkCode: vk_Delete),
    (Key: 'HOME';   vkCode: vk_Home),
    (Key: 'END';    vkCode: vk_End),
    (Key: 'PGUP';   vkCode: vk_Prior),
    (Key: 'PGDN';   vkCode: vk_Next),
    (Key: 'TAB';    vkCode: vk_Tab),
    (Key: 'ENTER';  vkCode: vk_Return),
    (Key: 'BKSP';   vkCode: vk_Back),
    (Key: 'PRTSC';  vkCode: vk_SnapShot),
    (Key: 'SHIFT';  vkCode: vk_Shift),
    (Key: 'ESCAPE'; vkCode: vk_Escape));

function FindKeyInArray(Key: TKeyString; var Code: Byte): Boolean;

implementation

uses SysUtils;

function FindKeyInArray(Key: TKeyString; var Code: Byte): Boolean;
{ Функция просматривает массив, чтобы найти лексему, заданную в параметре
 Key, и возвращает через параметр Code код виртуальной клавиши. }
var
  i: word;
begin
  Result := False;
  for i := Low(KeyDefArray) to High(KeyDefArray) do
    if UpperCase(Key) = KeyDefArray[i].Key then begin
```

```
      Code := KeyDefArray[i].vkCode;
      Result := True;
      Break;
    end;
end;

end.
```

Приняв строку, функция `SendKeys()` выделяет из нее отдельные нажатия клавиш и добавляет их в список в форме записей сообщений, содержащей сообщения `wm_KeyUp` и `wm_KeyDown`. Последние затем отправляются назад в Windows посредством ловушки `wh_JournalPlayback`.

Создание сообщений о нажатии клавиш

После выделения из строки каждого нажатия клавиши процедуре `MakeMessage()` передается виртуальный код клавиши и сообщение `wm_KeyUp`, `wm_KeyDown`, `wm_SysKeyUp` или `wm_SysKeyDown`. Процедура `MakeMessage()` создает новую запись сообщения, имитирующую нажатие требуемой клавиши, и добавляет ее в список сообщений, называемый `MessageList`. Используемая здесь запись сообщения не является стандартным типом `TMessage`, с которым вы уже знакомы, или даже типом `TMsg`, рассмотренным в главе 5, "Сообщения Windows". Эта запись называется сообщением `TEvent` и представляет сообщение системной очереди. Ее определение выглядит следующим образом:

```
type
{ Структура сообщений, используемых для публикации }
PEventMsg = ^TEventMsg;
  TEventMsg = packed record
    message: UINT;
    paramL: UINT;
    paramH: UINT;
    time: DWORD;
    hwnd: HWND;
  end;
```

В табл. 13.4 описаны значения полей записи `TEventMsg`.

Таблица 13.4. Значения полей записи TEventMsg

Поле	Значение
`message`	Константа сообщения. Для сообщения клавиатуры она может принимать значение `wm_(Sys)KeyUp` или `wm_(Sys)KeyDown`, а для сообщения мыши — значение `wm_XButtonUp`, `wm_XButtonDown` или `wm_MouseMove`
`paramL`	Если `message` является сообщением клавиатуры, то в этом поле хранится виртуальный код клавиши. Если `message` — сообщение мыши, то `wParam` содержит координату `x` курсора мыши (в единицах измерения экрана)
`paramH`	Если `message` является сообщением клавиатуры, то это поле содержит скан-код клавиши. Если `message` — сообщение мыши, то `lParam` содержит координату `y` курсора мыши
`time`	Время прихода сообщения, заданное в системных тиках
`Hwnd`	Идентифицирует окно, которому отправляется сообщение. С ловушками `wh_JournalPlayback` этот параметр не используется

Поскольку таблица преобразований в модуле KeyDefs переводит нажатия только в виртуальный код клавиши, необходимо найти способ для определения скан-кода клавиши, заданной виртуальным кодом. К счастью, в Windows API предусмотрена функция MapVirtualKey(), которая делает именно то, что нам нужно. В следующем фрагменте показан исходный текст процедуры MakeMessage():

```
procedure MakeMessage(vKey: byte; M: Cardinal);
{ Эта процедура создает запись TEventMsg, имитирующую
  нажатие клавиши, и добавляет ее в список сообщений. }
var
  E: PEventMsg;
begin
  New(E);           // Выделяем память для записи сообщения
  with E^ do begin
    message := M;                    // Устанавливаем поле message
    paramL := vKey;                  // Код vKey заносим в ParamL
    paramH := MapVirtualKey(vKey, 0); // Скан-код заносим в ParamH
    time := GetTickCount;            // Устанавливаем время
    hwnd := 0;                       // Игнорируется
  end;
  MessageList.Add(E);
end;
```

После создания списка сообщений ловушку можно настроить для воспроизведения последовательности клавиш. Это делается с помощью процедуры StartPlayback(), которая помещает первое сообщение из списка в глобальный буфер. Она также инициализирует глобальную переменную, которая подсчитывает количество воспроизведенных сообщений, и признаки, обозначающие состояние клавиш <Ctrl>, <Alt> и <Shift>. Затем эта процедура устанавливает ловушку. Текст процедуры StartPlayback() показан в следующем фрагменте:

```
procedure StartPlayback;
{ Инициализирует глобальные переменные и устанавливает ловушку. }
begin
  { Выбираем первое сообщение из списка и размещаем его в буфере,
    если получаем значение hc_GetNext раньше значения hc_Skip. }
MessageBuffer := TEventMsg(MessageList.Items[0]^);
  { Инициализируем счетчик сообщений и индикатор воспроизведения. }
  MsgCount := 0;
  { Инициализируем признаки клавиш <Alt>, <Control> и <Shift>. }
  AltPressed := False;
  ControlPressed := False;
  ShiftPressed := False;
  { Устанавливаем ловушку! }
  HookHandle := SetWindowsHookEx(wh_JournalPlayback, Play, hInstance, 0);
  if HookHandle = 0 then
    raise ESKSetHookError.Create('Couldn''t set hook')
    { Ловушку установить не удалось. }
  else
    Playing := True;
end;
```

Как вы могли заметить, в вызове функции SetWindowsHookEx(), Play — это имя функции ловушки. Объявление этой функции имеет следующий вид:

```
function Play(Code: integer; wParam, lParam: Longint): Longint; stdcall;
```

Параметры функции Play() описаны в табл. 13.5.

Таблица 13.5. Параметры функции ловушки Windows Play()

Значение	Описание
Code	Значение hc_GetNext говорит о том, что требуется подготовить для обработки следующее сообщение в списке. Это делается путем копирования следующего сообщения из списка в глобальный буфер. Значение hc_Skip означает, что в параметр lParam должен быть помещен указатель на следующее сообщение, предназначенное для обработки. Любое другое значение указывает, что нужно вызвать функцию CallNextHookEx() и передать ей параметры для следующей ловушки в цепочке
wParam	Не используется
lParam	Если Code принимает значение hc_Skip, требуется поместить указатель на следующую запись TEventMsg в параметр lParam
Возвращаемое значение	Возвращается нуль, если Code принимает значение hc_GetNext. Если же Code равен hc_Skip, возвращается время (в тиках), до истечения которого это сообщение должно быть обработано. Если возвращается нулевое значение, сообщение обрабатывается. В противном случае возвращаемое значение должно совпадать со значением, возвращенным функцией CallNextHookEx()

В листинге 13.7 содержится полный исходный текст модуля SendKey.pas.

Листинг 13.7. Модуль SendKey.pas

```
unit SendKey;

interface

uses
  SysUtils, WinTypes, Messages, Classes, KeyDefs;

type
  { Коды ошибок. }
  TSendKeyError = (sk_None, sk_FailSetHook, sk_InvalidToken,
    sk_UnknownError, sk_AlreadyPlaying);
  { От первого до последнего кода vk. }
  TvkKeySet = set of vk_LButton..vk_Scroll;

  { Исключительные ситуации. }
  ESendKeyError = class(Exception);
  ESKSetHookError = class(ESendKeyError);
  ESKInvalidToken = class(ESendKeyError);
```

```
  ESKAlreadyPlaying = class(ESendKeyError);

function SendKeys(S: String): TSendKeyError;
procedure WaitForHook;
procedure StopPlayback;

var
  Playing: Boolean = False;

implementation

uses Forms;

type
  { Потомок класса TList, который знает, как освободить свое содержимое. }
  TMessageList = class(TList)
  public
    destructor Destroy; override;
  end;

const
  { Действующие системные клавиши. }
  vkKeySet: TvkKeySet = [Ord('A')..Ord('Z'), vk_Menu, vk_F1..vk_F12];

destructor TMessageList.Destroy;
var
  i: longint;
begin
  { Освобождаем все записи сообщений перед отказом от списка. }
  for i := 0 to Count - 1 do
    Dispose(PEventMsg(Items[i]));
  inherited Destroy;
end;

var
  { Переменные, глобальные для DLL. }
  MsgCount: word = 0;
  MessageBuffer: TEventMsg;
  HookHandle: hHook = 0;
  MessageList: TMessageList = Nil;
  AltPressed, ControlPressed, ShiftPressed: Boolean;

procedure StopPlayback;
{ Отмена ловушки и очистка. }
begin
  { Если ловушка активна в данный момент, отключаем ее. }
  if Playing then
    UnhookWindowsHookEx(HookHandle);
  MessageList.Free;
  Playing := False;
end;
```

```
function Play(Code: integer; wParam, lParam: Longint): Longint; stdcall;
{ Это функция обратного вызова JournalPlayback. Она вызывается из Windows,
  когда Windows опрашивает события оборудования. Дальнейшие действия
  определяются значением параметра Code. }
begin
  case Code of
    HC_SKIP:
      { HC_SKIP означает опрос следующего сообщения из нашего списка.
        Если мы находимся в конце списка, нужно отменить ловушку
        JournalPlayback с этого момента. }
      begin
        inc(MsgCount);      // Увеличение счетчика сообщений
        { Проверяем, все ли сообщения были воспроизведены. }
        if MsgCount >= MessageList.Count then StopPlayback
        { В противном случае копируем следующее сообщение списка в буфер }
        else MessageBuffer := TEventMsg(MessageList.Items[MsgCount]^);
        Result := 0;
      end;

    HC_GETNEXT:
      { HC_GETNEXT означает заполнение параметров wParam и lParam
        соответствующими значениями, чтобы данное сообщение можно
        было воспроизвести. НЕ ОТМЕНЯЙТЕ установку ловушки в этот момент.
        Возвращаемое значение определяет интервал времени, в течение
        которого Windows должна обработать сообщение. Мы возвратим 0,
        чтобы оно было обработано немедленно. Передаем сообщение
        в буфер очереди сообщений. }
      begin
        { Перемещение сообщения в буфер очереди сообщений }
        PEventMsg(lParam)^ := MessageBuffer;
        Result := 0  { немедленная обработка }
      end

    else
      { Если значение параметра Code не равно hc_Skip или hc_GetNext,
        вызываем следующую ловушку в цепочке. }
      Result := CallNextHookEx(HookHandle, Code, wParam, lParam);
  end;
end;
procedure StartPlayback;
{ Инициализируем глобальные переменные и устанавливаем ловушку. }
begin
  { Берем первое сообщение из списка и размещаем в буфере,
    если значение hc_GetNext получено до значения hc_Skip. }
  MessageBuffer := TEventMsg(MessageList.Items[0]^);
  { Инициализируем счетчик сообщений и индикатор воспроизведения. }
  MsgCount := 0;
  { Инициализируем признаки клавиш <Alt>, <Control> и <Shift>. }
  AltPressed := False;
  ControlPressed := False;
  ShiftPressed := False;
```

```
  { Устанавливаем ловушку! }
  HookHandle := SetWindowsHookEx(wh_JournalPlayback, Play, hInstance, 0);
  if HookHandle = 0 then
    raise ESKSetHookError.Create('Couldn''t set hook')
  else
    Playing := True;
end;

procedure MakeMessage(vKey: byte; M: Cardinal);
{ Эта процедура создает запись TEventMsg, имитирующую нажатие
  клавиши, и добавляет ее в список сообщений. }
var
  E: PEventMsg;
begin
  New(E);               // Выделяем память для записи сообщения
  with E^ do begin
    message := M;                    // Устанавливаем поле message
    paramL := vKey;                  // Записываем код клавиши в ParamL
    paramH := MapVirtualKey(vKey, 0); // Скан-код сохраняем в ParamH
    time := GetTickCount;            // Устанавливаем время
    hwnd := 0;                       // Игнорируется
  end;
  MessageList.Add(E);
end;

procedure KeyDown(vKey: byte);
{ Генерирует сообщение KeyDown.}
begin
  { Не генерируем нажатие "sys"-клавиши, если нажата управляющая клавиша. }
  if AltPressed and (not ControlPressed) and (vKey in vkKeySet) then
    MakeMessage(vKey, wm_SysKeyDown)
  else
    MakeMessage(vKey, wm_KeyDown);
end;

procedure KeyUp(vKey: byte);
{ Генерирует сообщение KeyUp. }
begin
  { Не генерируем нажатие "sys"-клавиши, если нажата управляющая клавиша. }
  if AltPressed and (not ControlPressed) and (vKey in vkKeySet) then
    MakeMessage(vKey, wm_SysKeyUp)
  else
    MakeMessage(vKey, wm_KeyUp);
end;

procedure SimKeyPresses(VKeyCode: Word);
{ Эта функция имитирует нажатия клавиш для данной клавиши,
  учитывая текущее состояние клавиш <Alt>, <Ctrl> и <Shift>. }
begin
  { Нажимаем клавишу <Alt>, если был установлен признак AltPressed. }
  if AltPressed then
```

```
    KeyDown(vk_Menu);
  { Нажимаем клавишу <Ctrl>, если был установлен признак ControlPressed. }
  if ControlPressed then
    KeyDown(vk_Control);
  { Если клавиша <Shift> нажата или не нажаты клавиши сдвига и <Ctrl>... }
  if (((Hi(VKeyCode) and 1) <> 0) and (not ControlPressed)) or
    ShiftPressed then
    KeyDown(vk_Shift);    { ...нажимаем клавишу <Shift> }
  KeyDown(Lo(VKeyCode));  { Нажимаем клавишу }
  KeyUp(Lo(VKeyCode));    { Отпускаем клавишу }
  { Если клавиша <Shift> нажата или не нажаты клавиши сдвига и <Control>... }
  if (((Hi(VKeyCode) and 1) <> 0) and (not ControlPressed)) or
    ShiftPressed then
    KeyUp(vk_Shift);      {  ...отпускаем клавишу <Shift> }
  { Если установлен признак клавиши <Shift>, сбрасываем его. }
  if ShiftPressed then begin
    ShiftPressed := False;
  end;
  { Если установлен признак, отпускаем клавишу <Ctrl> и сбрасываем признак. }
  if ControlPressed then begin
    KeyUp(vk_Control);
    ControlPressed := False;
  end;
  { Если установлен признак, отпускаем клавишу <Alt> и сбрасываем признак. }
  if AltPressed then begin
    KeyUp(vk_Menu);
    AltPressed := False;
  end;
end;

procedure ProcessKey(S: String);
{ Эта функция анализирует каждый символ в строке,
  чтобы создать список сообщений. }
var
  KeyCode: word;
  Key: byte;
  index: integer;
  Token: TKeyString;
begin
  index := 1;
  repeat
    case S[index] of
      KeyGroupOpen:
        { Это начало специальной лексемы! }
        begin
          Token := '';
          inc(index);
          while S[index] <> KeyGroupClose do begin
           { Наращивание переменной Token до тех пор, пока не
             встретится символ конца лексемы. }
            Token := Token + S[index];
```

```
          inc(index);
          { Следим за тем, чтобы лексема не была слишком длинной. }
          if (Length(Token) = 7) and (S[index] <> KeyGroupClose) then
            raise ESKInvalidToken.Create('No closing brace');
              { Нет закрывающей скобки }
        end;
        { Ищем лексему в массиве - параметр Key будет содержать
          vk-код в случае успешного поиска. }
        if not FindKeyInArray(Token, Key) then
          raise ESKInvalidToken.Create('Invalid token');
        { Имитируем последовательность нажатий клавиш. }
        SimKeyPresses(MakeWord(Key, 0));
      end;
    AltKey: AltPressed := True; { Устанавливаем признак клавиши <Alt>. }
    ControlKey: ControlPressed := True; { Устанавливаем признак
                                          клавиши <Ctrl>. }
    ShiftKey: ShiftPressed := True; { Устанавливаем признак клавиши <Shift>. }
    else begin
      { Была нажата обычная клавиша. }
      { Преобразуем символ в слово, в котором старший байт содержит
        состояние клавиши <Shift>, а младший байт - vk-код. }
      KeyCode := vkKeyScan(S[index]);
      { Имитируем последовательность нажатий клавиш. }
      SimKeyPresses(KeyCode);
    end;
  end;
  inc(index);
until index > Length(S);
end;

procedure WaitForHook;
begin
  repeat Application.ProcessMessages until not Playing;
end;

function SendKeys(S: String): TSendKeyError;
{ Это единственная точка входа. На основе строки, передаваемой
  параметру S, эта функция создает список сообщений keyup/keydown,
  устанавливает ловушку JournalPlayback и воспроизводит сообщения,
  состоящие из нажатий клавиш. }
begin
  Result := sk_None;    // Предполагаем успешное завершение
  try
    if Playing then raise ESKAlreadyPlaying.Create('');
    MessageList := TMessageList.Create;  // Создаем список сообщений
    ProcessKey(S);                       // Создаем сообщения из строки
    StartPlayback;   // Устанавливаем ловушку и воспроизводим сообщения
  except
    { Если возникает исключительная ситуация, возвращается код
      ошибки и выполняется очистка. }
    on E:ESendKeyError do
```

```
    begin
      MessageList.Free;
      if E is ESKSetHookError then
        Result := sk_FailSetHook
      else if E is ESKInvalidToken then
        Result := sk_InvalidToken
      else if E is ESKAlreadyPlaying then
        Result := sk_AlreadyPlaying;
    end
    else
      { Обработчик всех остальных исключительных ситуаций. }
      Result := sk_UnknownError;
  end;
end;

end.
```

Использование функции SendKeys()

В этом разделе описано создание маленького проекта, в котором демонстрируется работа функции SendKeys(). Начнем с формы, которая содержит две строки редактирования (компоненты TEdit) и несколько кнопок (компоненты TButton), как показано на рис. 13.4. Этот проект называется TestSend.dpr.

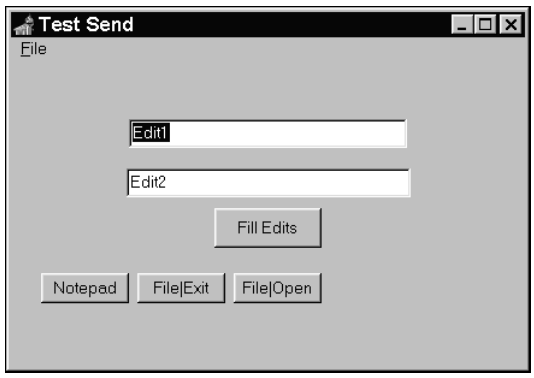


Рис. 13.4. Главная форма проекта TestSend

В листинге 13.8 представлен исходный код главного модуля этого проекта Main.pas, который включает обработчики событий, возникающих в результате щелчков на кнопках.

Листинг 13.8. Исходный код модуля Main.pas

```
unit Main;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
```

```
  Forms, Dialogs, StdCtrls, Menus;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Button1: TButton;
    Button2: TButton;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Open1: TMenuItem;
    Exit1: TMenuItem;
    Button4: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Open1Click(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses SendKey, KeyDefs;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.SetFocus;                    // Передаем фокус Edit1
  SendKeys('^{ DELETE} I love...');  // Отправляем нажатия клавиш в Edit1
  WaitForHook;                       // Воспроизводим клавиши
  Perform(wm_NextDlgCtl, 0, 0);      // Переходим в Edit2
  SendKeys('~delphi 5 ~developer''s ~guide!'); // Посылаем клавиши в Edit2
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  H: hWnd;
  PI: TProcessInformation;
  SI: TStartupInfo;
begin
```

```
  FillChar(SI, SizeOf(SI), 0);
  SI.cb := SizeOf(SI);
  { Запускаем Notepad }
  if CreateProcess(nil, 'notepad', nil, nil, False, 0, nil, nil, SI, PI) then
  begin
    { Ожидаем, пока Notepad не будет готов принять нажатия клавиш. }
    WaitForInputIdle(PI.hProcess, INFINITE);
    H := FindWindow('Notepad', nil);  // Находим окно Notepad
    if SetForegroundWindow(H) then    // Переносим его на передний план
      SendKeys('Hello from the SendKeys example!{ ENTER} ');
      // Посылаем нажатия клавиш!
  end
  else
    MessageDlg(Format('Failed to invoke Notepad.  Error code %d',
      [GetLastError]), mtError, [mbOk], 0);
        // Не удалось вызвать Notepad

end;

procedure TForm1.Open1Click(Sender: TObject);
begin
  ShowMessage('Open');
end;

procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  WaitForInputIdle(GetCurrentProcess, INFINITE);
  SendKeys('@fx');
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  WaitForHook;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  WaitForInputIdle(GetCurrentProcess, INFINITE);
  SendKeys('@fo');
end;

end.
```

По щелчку на кнопке `Button1` вызывается функция `SendKeys()`, которой передается нажатие комбинации клавиш <Shift+Delete>, для удаления содержимого строки редактирования `Edit1`. Затем в строку редактирования `Edit1` вводятся символы "`I love...`", посылается сим-

вол <Tab>, который передает фокус строке редактирования Edit2, и теперь уже в эту строку вводятся символы <Shift+D>, “elphi 5 ”, <Shift+D>, “eveloper’s ”, <Shift+G>, “uide!”.

Обработчик события OnClick кнопки Button2 также заслуживает внимания. В этом методе функция API CreateProcess() используется для вызова экземпляра редактора Notepad. Затем вызывается функция API WaitForInputIdle(), для того чтобы выдержать паузу, дабы процесс Notepad подготовился к приему. И, наконец, в окно Notepad вводится сообщение.

Использование OBJ-файлов C/C++[1]

В Delphi предусмотрена возможность компоновки объектных файлов (с расширением .OBJ), созданных с помощью других компиляторов, непосредственно в программы Delphi. Объектный файл можно скомпоновать в коде Object Pascal, используя директивы $L or $LINK, которые имеют следующий синтаксис:

```
{ $L filename.obj}
```

После компоновки объектного файла необходимо определить каждую функцию, вызываемую из него в коде Object Pascal. Используйте директиву external, обозначающую, что компилятор Object Pascal должен подождать до начала компоновки, прежде чем выполнить необходимые действия, связанные с именем этой функции. Например, в следующей строке кода определяется внешняя функция Foo, которая не принимает и не возвращает никаких параметров:

```
procedure Foo; external;
```

Несмотря на то что эта возможность кажется на первый взгляд довольно мощной, она сопровождается рядом ограничений, которые делают ее во многих случаях трудной для реализации.

- Object Pascal может напрямую получать доступ только к коду, но не к данным, содержащимся в объектных файлах (немного ниже рассматривается хитроумный метод, позволяющий добраться и к данным в OBJ-файле). Однако из объектных файлов доступ к данным Object Pascal открыт.
- Object Pascal не может компоновать код с LIB-файлами (статическими библиотеками).
- Объектные файлы, содержащие классы C++, нельзя скомпоновать из-за неявных ссылок на библиотеки времени выполнения (RTL) C++. Несмотря на то что проблему этих ссылок можно решить путем перенесения C++ RTL в OBJ-файлы, в общем случае игра не стоит свеч.
- Объектные файлы должны быть в формате Intel OMF. Это выходной формат компиляторов Borland C++, но не компиляторов Microsoft C++, которые используют COFF-формат OBJ-файлов.

На заметку В предыдущих версиях Delphi объектные файлы не могли содержать ссылки на код или данные, хранящиеся в других объектных файлах. Проблема ссылок OBJ–OBJ успешно решена в компиляторах Delphi версии 3 и последующих.

[1] *Здесь под C++ авторы книги понимают Borland C++, если не сказано иное. Следует отдавать себе отчет в том, что при использовании другого компилятора C++ не все сказанное далее остается справедливым.* — Прим. ред.

Вызов функции

Предположим, у нас есть объектный файл C++ `ccode.obj`, содержащий функцию со следующим прототипом:

```
int __fastcall SAYHELLO(char * hellostr)
```

Для вызова этой функции из приложения Delphi необходимо сначала с помощью директивы `$L` или `$LINK` указать, что ее объектный файл потребуется скомпоновать в EXE-файл Delphi:

```
{ $L ccode.obj}
```

Затем нужно создать определение Object Pascal для этой функции, как показано ниже.

```
function SayHello(Text: PChar): integer; external;
```

Обратите внимание на применение в C++ директивы `__fastcall`, которая служит для гарантии использования одного и того же соглашения о вызове в коде C++ и Object Pascal. Отсутствие полного соответствия между прототипом C++ и объявлением Object Pascal может привести к фатальным ошибкам. Проблемы с соглашением о вызове являются самыми распространенными у разработчиков, пытающихся состыковать два языка. Чтобы внести ясность в этот вопрос, ниже приведено соответствие между директивами соглашения о вызове Object Pascal и C++.

Object Pascal	C++
`register*`	`__fastcall`
`pascal`	`__pascal`
`cdecl`	`__cdecl*`
`stdcall`	`__stdcall`

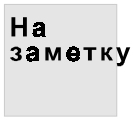
* Означает соглашение о вызове, действующее по умолчанию для данного языка.

Изменение имен

По умолчанию компилятор C++ будет изменять (mangle) имена функций (путем дополнения типами аргументов и возврата), не объявленных явным образом с помощью модификатора `extern "C"`. Компилятор Object Pascal, конечно же, не занимается таким изменением имен функций. Например, утилита Delphi `TDUMP` обнаруживает экспортированное символьное имя функции `SAYHELLO`, представленной выше в объектном файле `ccode.obj` в виде `@SAYHELLO$qqrpc`, в то время как в качестве имени импортированной функции в соответствии с правилами Object Pascal будет использоваться имя `SAYHELLO` (Object Pascal переводит все символы в прописные буквы).

На первый взгляд это кажется неразрешимой проблемой: как компилятор Delphi сможет разрешать внешние ссылки, если даже не совпадают имена функций? Оказывается, компилятор Delphi просто игнорирует дополнительную порцию имени (символ `@` и все, что находится после `$`). (Еще раз напомним: под C++ здесь понимается Borland C++ — у другого компилятора может быть совершенно иная система преобразования имен. — Прим. ред.) Но это, в свою очередь, может дать несколько неприятных побочных эффектов.

Истинная причина дополнения имен компилятором C++ — разрешить перегрузку функций (наличие у разных функций одинаковых имен, но различных списков параметров). Если у вас есть функция, для которой существует несколько перегруженных определений, а Delphi игнорирует дополненную порцию символьного имени, вы никогда не будете знать наверняка, вызывает ли Delphi именно ту перегруженную функцию, которую вы хотели вызвать. Ввиду этих сложностей рекомендуем не вызывать перегруженные функции через объектные файлы.



Имена функций в исходном файле C++ (с расширением `.CPP`) при компиляции всегда дополняются типами аргументов и возврата, если в прототипе функции не использован модификатор `extern "C"` или если для компилятора C++ не был задействован соответствующий ключ командной строки, подавляющий правку имен.

Разделение данных

Как упоминалось выше, к данным Delphi можно получить доступ из объектного файла. Сначала нужно объявить глобальную переменную в исходном модуле Object Pascal аналогично тому, как показано ниже (обратите внимание на символ подчеркивания).

```
var
  _GLOBALVAR: PChar = 'This is a Delphi String';
```

Необходимо отметить, что, хотя в данном случае переменная инициализирована, это не обязательное требование.

В модуле C++ объявите переменную с тем же именем, используя модификатор `extern`:

```
extern char * GLOBALVAR;
```

По умолчанию компилятор Borland C++, генерируя внешние символьные имена, предваряет внешние переменные символом подчеркивания (т.е. переменная `GLOBALVAR` становится `_GLOBALVAR`). Это можно обойти двумя способами.

- Используйте ключ командной строки, запрещающий добавление символа подчеркивания (`-u-` для компиляторов Borland C++).
- Разместите символ подчеркивания перед именем переменной в коде Object Pascal.

Несмотря на то что в коде Object Pascal нельзя непосредственно использовать данные, объявленные в объектном файле, этого можно добиться обходными средствами. Сначала данные, которые требуется экспортировать, нужно объявить в коде C++ с помощью директивы `__export`. Например, пусть некоторый символьный массив (типа `char`) необходимо сделать доступным для экспортирования:

```
char __export C_VAR[128];
```

Затем (внимание: это первая часть нашего трюка) следует объявить эти данные как внешнюю процедуру в коде Object Pascal (еще раз обратите внимание на символ подчеркивания):

```
procedure _C_VAR; external;  // "Хитрый" способ импортирования OBJ-данных
```

Это позволит компоновщику разрешить ссылку на переменную `_C_VAR` в коде Object Pascal. Наконец (это вторая часть трюка), вы можете использовать переменную `_C_VAR` в своем Pascal-коде как указатель на данные. Например, для получения значения массива можно использовать следующий код:

```
type
  PCharArray = ^TCharArray;
  TCharArray = array[0..127] of char;

function GetCArray: string;
var
  A: PCharArray;
begin
  A := PCharArray(@_C_VAR);
  Result := A^;
end;
```

А этот текст можно использовать для установки значения массива:

```
procedure SetCArray(const S: string);
var
  A: PCharArray;
begin
  A := PCharArray(@_C_VAR);
  StrLCopy(A^, PChar(S), SizeOf(TCharArray));
end;
```

Использование библиотеки RTL Delphi

Если объектный файл содержит ссылки на библиотеку RTL C++, то при его компоновке с вашим приложением Delphi возникают определенные трудности, связанные с тем, что библиотеки RTL обычно размещаются в LIB-файлах, а у Delphi нет возможности компоновки с LIB-файлами.

Как же решить эту проблему? Один из способов состоит в вырезке определений используемых внешних функций из исходного кода библиотеки RTL C++ и помещении их в объектный файл. Однако, если вызывается не одна или две внешние функции, а существенно больше, этот тип решения может оказаться слишком сложным — не говоря уже о том, что объектный файл станет просто огромным.

Более элегантное решение этой проблемы состоит в создании одного или нескольких файлов заголовков, в которых переопределяются все функции RTL, вызываемые с помощью модификатора `external`, и в действительной реализации этих функций внутри кода Object Pascal. Предположим, необходимо вызвать из кода C++ функцию API `MessageBox()`. Обычно это потребовало бы включения в текст строки `#include <windows.h>` и компоновки с соответствующими библиотеками Win32. Переопределим функцию `MessageBox()` в коде C++ следующим образом:

```
extern "C" int __stdcall MessageBox(long, char *, char *, long);
```

В этом случае при построении выполняемого файла компоновщик Object Pascal сможет найти собственную функцию с именем `MessageBox`. Конечно же, такая функция существует и определена в модуле `Windows`. Теперь приложение благополучно скомпилируется, и его компоновка пройдет “без сучка и задоринки”.

В листинге 13.9 демонстрируется законченный пример всего того, о чем шла речь до сих пор. Это довольно простой модуль C++, именуемый `ccode.cpp`.

Листинг 13.9. Простой модуль ccode.cpp, написанный на языке C++

```
#include "PasStng.h"

// Глобальные переменные
extern char * GLOBALVAR;

// Экспортируемые данные
char __export C_VAR[128];

#ifdef __cplusplus
extern "C" {
#endif

// Внешние функции
extern int __stdcall MessageBox(long, char *, char *, long);

//Функции
int  __export__cdecl SAYHELLO(char * hellostr)
{
  char a[64];
  memset(a, 64, 0);
  strcat(a, hellostr);
  strcat(a, " from Borland C++Builder");
 MessageBox(0, a, GLOBALVAR, 0);
  return 0;
}

#ifdef __cplusplus
}     // end of extern "C"
#endif
```

Помимо функции MessageBox(), заслуживают внимания обращения к функциям C++ RTL memset() и strcat(). Эти функции обрабатываются аналогичным образом в файле заголовков PasStng.h, который содержит еще несколько самых распространенных функций из заголовочного файла string.h. Файл PasStng.h представлен в листинге 13.10.

Листинг 13.10. Модуль PasStng.h

```
// PasStng.h
// Этот модуль реализует часть заголовочного файла C++ RTL string.h,
// чтобы эти вызовы вместо него мог обработать RTL-файл Object Pascal.

#ifndef PASSTNG_H
#define PASSTNG_H

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned size_t;
```

```
#endif

#ifdef __cplusplus
extern "C" {
#endif

extern char * __cdecl strcat(char *dest, const char *src);
extern int __cdecl stricmp(const char *s1, const char *s2);
extern size_t __cdecl strlen(const char *s);
extern char * __cdecl strlwr(char *s);
extern char * __cdecl strncat(char *dest, const char *src, size_t maxlen);
extern void * __cdecl memcpy(void *dest, const void *src, size_t n);
extern int __cdecl strncmp(const char *s1, const char *s2, size_t maxlen);
extern int __cdecl strncmpi(const char *s1, const char *s2, size_t n);
extern void * __cdecl memmove(void *dest, const void *src, size_t n);
extern char * __cdecl strncpy(char *dest, const char *src, size_t maxlen);
extern void * __cdecl memset(void *s, int c, size_t n);
extern int __cdecl strnicmp(const char *s1, const char *s2, size_t maxlen);
extern void __cdecl movmem(const void *src, void *dest, unsigned length);
extern void __cdecl setmem(void *dest, unsigned length, char value);
extern char * __cdecl stpcpy(char *dest, const char *src);
extern int __cdecl strcmp(const char *s1, const char *s2);
extern char * __cdecl strstr(char *s1, const char *s2);
extern int __cdecl strcmpi(const char *s1, const char *s2);
extern char * __cdecl strupr(char *s);
extern char * __cdecl strcpy(char *dest, const char *src);

#ifdef __cplusplus
}
#endif

#endif  // PASSTNG_H
```

Поскольку этих функций нет в RTL Object Pascal, данную проблему можно обойти путем создания и включения в наш проект модуля Object Pascal, который преобразует эти функции к их аналогам Object Pascal. Текст этого модуля (`PasStrng.pas`) представлен в листинге 13.11.

Листинг 13.11. Модуль `PasStrng.pas`

```
unit PasStrng;

interface

uses Windows;

function _strcat(Dest, Source: PChar): PChar; cdecl;
procedure _memset(P: Pointer; Count: Integer; value: DWORD); cdecl;
function _stricmp(P1, P2: PChar): Integer; cdecl;
function _strlen(P1: PChar): Integer; cdecl;
function _strlwr(P1: PChar): PChar; cdecl;
```

```
function _strncat(Dest, Source: PChar; MaxLen: Integer): PChar; cdecl;
function _memcpy(Dest, Source: Pointer; Len: Integer): Pointer;
function _strncmp(P1, P2: PChar; MaxLen: Integer): Integer; cdecl;
function _strncmpi(P1, P2: PChar; MaxLen: Integer): Integer; cdecl;
function _memmove(Dest, Source: Pointer; Len: Integer): Pointer;
function _strncpy(Dest, Source: PChar; MaxLen: Integer): PChar; cdecl;
function _strnicmp(P1, P2: PChar; MaxLen: Integer): Integer; cdecl;
procedure _movmem(Source, Dest: Pointer; MaxLen: Integer); cdecl;
procedure _setmem(Dest: Pointer; Len: Integer; Value: Char); cdecl;
function _stpcpy(Dest, Source: PChar): PChar; cdecl;
function _strcmp(P1, P2: PChar): Integer; cdecl;
function _strstr(P1, P2: PChar): PChar; cdecl;
function _strcmpi(P1, P2: PChar): Integer; cdecl;
function _strupr(P: PChar): PChar; cdecl;
function _strcpy(Dest, Source: PChar): PChar; cdecl;

implementation

uses SysUtils;

function _strcat(Dest, Source: PChar): PChar;
begin
  Result := SysUtils.StrCat(Dest, Source);
end;

function _stricmp(P1, P2: PChar): Integer;
begin
  Result := StrIComp(P1, P2);
end;

function _strlen(P1: PChar): Integer;
begin
  Result := SysUtils.StrLen(P1);
end;

function _strlwr(P1: PChar): PChar;
begin
  Result := StrLower(P1);
end;

function _strncat(Dest, Source: PChar; MaxLen: Integer): PChar;
begin
  Result := StrLCat(Dest, Source, MaxLen);
end;

function _memcpy(Dest, Source: Pointer; Len: Integer): Pointer;
begin
  Move(Source^, Dest^, Len);
  Result := Dest;
end;
```

```
function _strncmp(P1, P2: PChar; MaxLen: Integer): Integer;
begin
  Result := StrLComp(P1, P2, MaxLen);
end;

function _strncmpi(P1, P2: PChar; MaxLen: Integer): Integer;
begin
  Result := StrLIComp(P1, P2, MaxLen);
end;

function _memmove(Dest, Source: Pointer; Len: Integer): Pointer;
begin
  Move(Source^, Dest^, Len);
  Result := Dest;
end;

function _strncpy(Dest, Source: PChar; MaxLen: Integer): PChar;
begin
  Result := StrLCopy(Dest, Source, MaxLen);
end;

procedure _memset(P: Pointer; Count: Integer; Value: DWORD);
begin
  FillChar(P^, Count, Value);
end;

function _strnicmp(P1, P2: PChar; MaxLen: Integer): Integer;
begin
  Result := StrLIComp(P1, P2, MaxLen);
end;

procedure _movmem(Source, Dest: Pointer; MaxLen: Integer);
begin
  Move(Source^, Dest^, MaxLen);
end;

procedure _setmem(Dest: Pointer; Len: Integer; Value: Char);
begin
  FillChar(Dest^, Len, Value);
end;

function _stpcpy(Dest, Source: PChar): PChar;
begin
  Result := StrCopy(Dest, Source);
end;

function _strcmp(P1, P2: PChar): Integer;
begin
  Result := StrComp(P1, P2);
end;
```

```
function _strstr(P1, P2: PChar): PChar;
begin
  Result := StrPos(P1, P2);
end;

function _strcmpi(P1, P2: PChar): Integer;
begin
  Result := StrIComp(P1, P2);
end;

function _strupr(P: PChar): PChar;
begin
  Result := StrUpper(P);
end;

function _strcpy(Dest, Source: PChar): PChar;
begin
  Result := StrCopy(Dest, Source);
end;

end.
```

Используя показанный здесь метод, вы могли бы реализовать в заголовочных файлах и другие функции C++ RTL и Win32 API (которые затем переводятся в аналоги Object Pascal).

Использование классов C++

Несмотря на невозможность применения классов C++, содержащихся в объектном файле, можно достигнуть некоторого ограниченного использования классов C++, содержащихся в DLL. "Ограниченное использование" означает, что вам разрешается вызывать виртуальные функции, представленные в классе C++, из Delphi. Причем это возможно благодаря тому, что как Object Pascal, так и C++ следуют стандарту COM для виртуальных интерфейсов (см. главу 23 второго тома, "COM-ориентированные технологии").

В листинге 13.12 представлен исходный код модуля C++ `cdll.cpp`, содержащего определение класса. Обратите, в частности, внимание на две специальные функции — одна из них создает и возвращает ссылку на новый объект, а другая освобождает эту ссылку. Эти функции можно сравнить с потайным ходом, с помощью которого объект будет разделен между языками.

Листинг 13.12. Модуль C++ `cdll.cpp`, содержащий определение класса

```
#include <windows.h>

// Объекты
class TFoo
{
  virtual int function1(char *);
```

```
  virtual int function2(int);
} ;

//Функции-члены
int TFoo::function1(char * str1)
{
  MessageBox(NULL, str1, "Hello from C++ DLL", MB_OK);
  return 0;
}

int TFoo::function2(int i)
{
  return i * i;
}

#ifdef __cplusplus
extern "C"  {
#endif

// Прототипы
TFoo * __declspec(dllexport) ClassFactory(void);
void __declspec(dllexport) ClassKill(TFoo *);

TFoo * __declspec(dllexport) CLASSFACTORY(void)
{
  TFoo * Foo;
  Foo = new TFoo;
  return Foo;
}
void __declspec(dllexport) CLASSKILL(TFoo * Foo)
{
  delete Foo;
}

int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
    return 1;
}

#ifdef __cplusplus
}
#endif
```

Чтобы использовать этот объект в приложении Delphi, нужно сделать две вещи. Во-первых, требуется импортировать функции, которые создают и разрушают экземпляры класса. А во-вторых, необходимо построить определение виртуального абстрактного класса Object Pascal, который бы включал в себя класс C++. Вот как это делается:

```
type
  TFoo = class
    function Function1(Str1: PChar): integer; virtual; cdecl; abstract;
```

```
    function Function2(i: integer): integer; virtual; cdecl; abstract;
  end;
function ClassFactory: TFoo; cdecl; external 'cdll.dll' name '_CLASSFACTORY';
procedure ClassKill(Foo: TFoo); cdecl; external 'cdll.dll' name '_CLASSKILL';
```

На заметку

При определении оболочки Object Pascal для класса C++ не нужно беспокоиться об именах функций, поскольку они не имеют значения при внутреннем вызове функций. Так как все вызовы будут осуществляться через таблицу виртуальных методов (Virtual Method Table), решающим фактором здесь является порядок, в котором объявляются эти функции. Поэтому проследите за тем, чтобы порядок как в определении C++, так и в определении Object Pascal был одним и тем же.

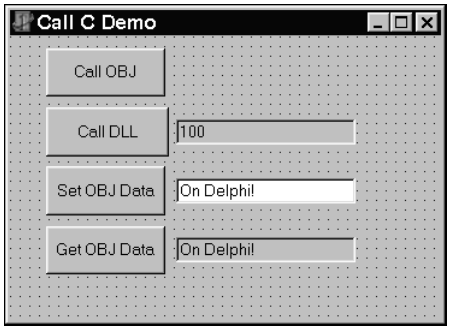


Рис. 13.5. Главная форма проекта CallC

В листинге 13.13 представлен модуль `Main.pas` — главный модуль проекта `CallC.dpr`, в котором демонстрируются все методы работы с объектами C++, обсуждавшиеся в этой главе. Главная форма этого проекта представлена на рис. 13.5.

Листинг 13.13. Модуль `Main.pas`

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    Button1: TButton;
    Button2: TButton;
    FooData: TEdit;
    Button3: TButton;
    Button4: TButton;
    SetCVarData: TEdit;
    GetCVarData: TEdit;
    procedure Button1Click(Sender: TObject);
```

```
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;
  _GlobalVar: PChar = 'This is a Delphi String';

implementation

uses PasStrng;

{$R *.DFM}

{$L ccode.obj}

type
  TFoo = class
    function Function1(Str1: PChar): integer; virtual; cdecl; abstract;
    function Function2(i: integer): integer; virtual; cdecl; abstract;
  end;

  PCharArray = ^TCharArray;
  TCharArray = array[0..127] of char;

// Импорт из OBJ-файла:
function _SAYHELLO(Text: PChar): Integer; cdecl; external;
procedure _C_VAR; external;  // трюк с импортом OBJ-данных

// Импорт из файла DLL:
function ClassFactory: TFoo; cdecl; external 'cdll.dll' name '_CLASSFACTORY';
procedure ClassKill(Foo: TFoo); cdecl; external 'cdll.dll' name '_CLASSKILL';

procedure TMainForm.Button1Click(Sender: TObject);
begin
  _SayHello('hello world');
end;

procedure TMainForm.Button2Click(Sender: TObject);
var
  Foo: TFoo;
begin
  Foo := ClassFactory;
  Foo.Function1('huh huh, cool.');
  FooData.Text := IntToStr(Foo.Function2(10));
```

```
  ClassKill(Foo);
end;

function GetCArray: string;
var
  A: PCharArray;
begin
  A := PCharArray(@_C_VAR);
  Result := A^;
end;

procedure SetCArray(const S: string);
var
  A: PCharArray;
begin
  A := PCharArray(@_C_VAR);
  StrLCopy(A^, PChar(S), SizeOf(TCharArray));
end;

procedure TMainForm.Button3Click(Sender: TObject);
begin
  SetCArray(SetCVarData.Text);
end;

procedure TMainForm.Button4Click(Sender: TObject);
begin
  GetCVarData.Text := GetCArray;
end;

end.
```

Хотя продемонстрированный здесь метод и в самом деле позволяет использовать ограниченные средства связи Object Pascal с классами C++, но, если вы хотите реализовать это в больших масштабах, рекомендуем для связи между языками использовать объекты COM, как описано в главе 23 второго тома, "COM-ориентированные технологии".

Санкинг

На некотором этапе разработки приложений Windows или Win32 вам, возможно, потребуется вызвать 16-разрядный код из 32-разрядного приложения или даже 32-разрядный код из 16-разрядного приложения. Этот процесс называется санкингом (thunking). (В связи с тем, что дословно перевести термин thunk на русский язык невозможно, мы пользуемся фонетически близким термином. — Прим. ред.) Несмотря на то что в Win32 предусмотрены различные пути решения этой задачи, она все равно остается одной из самых трудных при разработке приложений Windows.

Необходимо иметь в виду, что помимо санкинга, существует такой процесс, как автоматизация (см. главу 23 второго тома, "COM-ориентированные технологии"), который предоставляет приемлемую альтернативу для пересечения границ между 16- и 32-разрядными системами. Это средство встроено в интерфейс автоматизации `IDispatch`.

В Win32 предусмотрено три различных типа санкинга: универсальный (universal), общий (generic) и линейный (flat). Все эти разновидности обладают как достоинствами, так и недостатками.

- Универсальный санкинг доступен только на платформе Win32s (Win32s представляет собой подмножество Win32 API, доступное в 16-разрядной Windows). Он позволяет 16-разрядным приложениям загружать библиотеки DLL Win32 и вызывать их функции. Поскольку эта разновидность санкинга применима только для платформы Win32s, которая официально не поддерживается Delphi, данная тема больше рассматриваться не будет.
- Общий санкинг позволяет 16-разрядным приложениям Windows вызывать функции библиотек DLL Win32, в системах под управлением Windows 95, 98, NT и 2000. Это наиболее гибкий тип санкинга, поскольку он доступен на обеих важнейших платформах Win32 и опирается на интерфейс API. Этот вариант санкинга будет рассмотрен во всех деталях.
- Линейный санкинг позволяет приложениям Win32 вызывать функции 16-разрядных библиотек DLL, а 16-разрядным приложениям вызывать функции библиотек DLL Win32. К сожалению, этот тип санкинга доступен только в ОС Windows 95/98. Он также требует использования санк-компилятора для создания объектных файлов, которые должны компоноваться как с 32-, так и 16-разрядной частью. Ввиду недостаточной совместимости и потребности в дополнительных инструментах линейный санкинг рассматриваться здесь не будет.

Кроме того, существует метод разделения данных между 32- и 16-разрядными процессами посредством использования сообщения Windows `WM_COPYDATA`, которое, в частности, предоставляет простое средство доступа к 16-разрядному коду из среды Windows NT (подробности ниже).

Общий санкинг

Реализация общего санкинга облегчается благодаря набору функций API, предусмотренных как на 16-, так и на 32-разрядной сторонах. Эти модули функций API называются соответственно `WOW16` и `WOW32`. Модуль `WOW16` содержит функции, которые позволяют загружать библиотеки DLL Win32, получать адреса функций в загруженных библиотеках и вызывать эти функции. Исходный код модуля `WOW16.pas` представлен в листинге 13.14.

Листинг 13.14. Модуль `WOW16.pas`, включающий функции для загрузки библиотек DLL Win32 из 16-разрядных приложений

```
unit WOW16;
{ Модуль, который предоставляет интерфейс к функциям API 16-разрядной
  Windows в Win32 (WOW) из 16-разрядного приложения, запускаемого в среде
  Win32. Эти функции позволяют использовать 32-разрядные библиотеки DLL
  в 16-разрядных приложениях. }
{ Copyright (c) 1996, 99 Steve Teixeira and Xavier Pacheco}

interface

uses WinTypes;
```

```
type
  THandle32 = Longint;
  DWORD = Longint;

{ Управляющая часть модуля Win32.}

{ Следующие функции принимают параметры, прямо соответствующие
  вызовам функций Win32 API, которые они загружают. За подробностями
  обращайтесь к справочной документации по Win32. }
function LoadLibraryEx32W(LibFileName: PChar; hFile, dwFlags: DWORD):
  THandle32;
function FreeLibrary32W(LibModule: THandle32): BOOL;
function GetProcAddress32W(Module: THandle32; ProcName: PChar): TFarProc;

{ Функция GetVDMPointer32W преобразует 16-разрядный (16:16) указатель
  в 32-разрядный линейный (0:32) указатель. Значение FMode должно быть
  равно 1, если 16-разрядный указатель является адресом в защищенном
  режиме (нормальная ситуация в Windows 3.x), или 0, если 16-разрядный
  указатель является адресом в реальном режиме. }
{ ПРИМЕЧАНИЕ.  При построении пользовательских проектов Windows NT
  проверка диапазонов не выполняется. Она выполняется в отладочном
  проекте WOW32.DLL и приведет к возврату 0, если предлагаемое
  смещение превысит действующие границы.}
function GetVDMPointer32W(Address: Pointer; fProtectedMode: WordBool):
  DWORD;

{ Функция CallProc32W вызывает процедуру, адрес которой был считан
  функцией GetProcAddress32W. Настоящее определение этой функции позволяет,
  на самом деле передавать несколько параметров типа DWORD, предшествующих
  параметру ProcAddress, а параметр nParams должен обозначать число
  параметров, передаваемых до параметра ProcAddress. Параметр
  AddressConvert представляет собой битовую маску, означающую то, какие
  из параметров являются 16-разрядными указателями, нуждающимися в
  преобразовании до вызова 32-разрядной функции. Поскольку эта функция
  не "опускается" до определения в Object Pascal, то вам стоит
  использовать вместо нее упрощенную функцию Call32BitProc. }
function CallProc32W(Params: DWORD; ProcAddress, AddressConvert,
  nParams: DWORD): DWORD;

{ Функцию Call32BitProc передается массив констант типа Longint в качестве
  списка параметров для функции, заданной параметром ProcAddress.
  Эта процедура отвечает за упаковку параметров в корректный формат
  и вызов WOW-функции CallProc32W. }
function Call32BitProc(ProcAddress: DWORD; Params: array of Longint;
  AddressConvert: Longint): DWORD;

{ Преобразует 16-разрядный дескриптор окна в 32-разрядный для использования
  в Windows NT. }
function HWnd16To32(Handle: hWnd): THandle32;
```

```
{ Преобразует 32-разрядный дескриптор окна в 16-разрядный. }
function HWnd32To16(Handle: THandle32): hWnd;

implementation
uses WinProcs;
function HWnd16To32(Handle: hWnd): THandle32;
begin
  Result := Handle or $FFFF0000;
end;
function HWnd32To16(Handle: THandle32): hWnd;
begin
  Result := LoWord(Handle);
end;
function BitIsSet(Value: Longint; Bit: Byte): Boolean;
begin
  Result := Value and (1 shl Bit) <> 0;
end;
procedure FixParams(var Params: array of Longint; AddConv: Longint);
var
  i: integer;
begin
  for i := Low(Params) to High(Params) do
    if BitIsSet(AddConv, i) then
      Params[i] := GetVDMPointer32W(Pointer(Params[i]), True);
end;
function Call32BitProc(ProcAddress: DWORD; Params: array of Longint;
  AddressConvert: Longint): DWORD;
var
  NumParams: word;
begin
  FixParams(Params, AddressConvert);
  NumParams := High(Params) + 1;
  asm
    les di, Params              {  es:di -> Params }
    mov cx, NumParams           {  Счетчик цикла = число параметров }
  @@1:
    push es:word ptr [di + 2]   {  Помещаем в стек старшее слово параметра x }
    push es:word ptr [di]       { Помещаем в стек младшее слово параметра x }
    add di, 4                   {  Переход к следующему параметру }
    loop @@1                    {  Проход по всем параметрам }
    mov cx, ProcAddress.Word[2] {  cx = старшее слово адреса ProcAddress }
    mov dx, ProcAddress.Word[0] {  dx = младшее слово адреса ProcAddress }
    push cx                     {  В стек: старшая часть ProcAddress }
    push dx                     { В стек: младшая часть ProcAddress }
    mov ax, 0
    push ax          { В стек: старшая часть псевдоадреса AddressConvert }
    push ax          { В стек: младшая часть псевдоадреса AddressConvert }
    push ax          { В стек: старшая часть NumParams }
    mov cx, NumParams
    push cx          { В стек: младшая часть NumParams }
```

```
    call CallProc32W            {  Вызов функции }
    mov Result.Word[0], ax
    mov Result.Word[2], dx      {  Запоминаем значение возврата }
  end
end;
{ 16-разрядные WOW-функции }
function LoadLibraryEx32W;            external 'KERNEL' index 513;
function FreeLibrary32W;              external 'KERNEL' index 514;
function GetProcAddress32W;           external 'KERNEL' index 515;
function GetVDMPointer32W;            external 'KERNEL' index 516;
function CallProc32W;                 external 'KERNEL' index 517;
end.
```

Все функции в этом модуле просто экспортируются из 16-разрядного ядра, за исключением функции `Call32BitProc()`, в которой используется некоторый ассемблерный код, чтобы разрешить пользователям передачу переменного числа параметров в массиве типа `array of Longint`.

В листинге 13.15 представлен модуль `WOW32.pas`, который состоит из функций `WOW32`.

Листинг 13.15. Модуль `WOW32.pas` — интерфейс для `WOW32.DLL`, функции которой обеспечивают доступ к 16-разрядному коду из приложений Win32

```
unit WOW32;
{ Импортирование функций библиотеки WOW32.DLL, содержащей утилиты для
  доступа к 16-разрядному коду из среды приложений Win32 }
{ Copyright (c) 1996, 1999 Steve Teixeira and Xavier Pacheco }

interface
uses Windows;

{ Преобразование указателя 16:16 -> 0:32.
  Функция WOWGetVDMPointer преобразует передаваемый ей 16-разрядный
  адрес в эквивалентный 32-разрядный линейный указатель. Если параметр
  fProtectedMode имеет значение TRUE, функция воспринимает старшие
  16 разрядов как селектор в локальной таблице дескрипторов.
  Если значение параметра fProtectedMode равно FALSE, старшие 16 разрядов
  воспринимаются как значение сегмента в реальном режиме. В любом случае
  младшие 16 разрядов используются как смещение.

  Если селектор некорректен, возвращаемое значение равно 0.

  ПРИМЕЧАНИЕ. При построении пользовательских проектов Windows NT проверка
  диапазонов не выполняется. Она выполняется в отладочном проекте WOW32.DLL
  и вернет 0, если предлагаемое смещение превысит действующие границы.}
function WOWGetVDMPointer(vp, dwBytes: DWORD; fProtectedMode: BOOL):
  Pointer; stdcall;

{ Следующие две функции присутствуют здесь для совместимости с  Windows 95.
  В Win95 глобальная куча может быть переупорядочена, что приведет
  к получению неверных линейных указателей, возвращаемых функцией
```

```
  WOWGetVDMPointer при выполнении санкинга. В Windows NT 16-разрядный
  процесс VDM при выполнении санкинга полностью прекращается, поэтому
  возможность реорганизации кучи реализуется только при условии, если
  обратный вызов выполняется для кода Win16.

  Win95-версии этих функций вызывают функцию GlobalFix, чтобы
  заблокировать линейный адрес сегмента, а функцию GlobalUnfix –
  чтобы освободить сегмент.

  Версии этих функций, реализованные для Windows NT, *не* вызывают
  функции GlobalFix/GlobalUnfix, связанные с сегментом, поскольку без
  обратного вызова не будет выполнено никаких действий над кучей. Если ваш
  санкинг выполнит обратный вызов для 16-разрядной стороны, обязательно
  откажитесь от линейных указателей и снова вызовите функцию
  WOWGetVDMPointer, чтобы убедиться в корректности адреса.}
function WOWGetVDMPointerFix(vp, dwBytes: DWORD; fProtectedMode: BOOL):
  Pointer; stdcall;
procedure WOWGetVDMPointerUnfix(vp: DWORD); stdcall;

{ Управление памятью Win16.

  Эти функции можно использовать для управления памятью в куче Win16.
  Следующие четыре функции идентичны их Win16-аналогам, за исключением
  того, что они вызываются из кода Win32.}
function WOWGlobalAlloc16(wFlags: word; cb: DWORD): word; stdcall;
function WOWGlobalFree16(hMem: word): word; stdcall;
function WOWGlobalLock16(hMem: word): DWORD; stdcall;
function WOWGlobalUnlock16(hMem: word): BOOL; stdcall;
{ Следующие три функции объединяют две обычные операции в один
  переключатель для 16-разрядного режима.}

function WOWGlobalAllocLock16(wFlags: word; cb: DWORD; phMem: PWord):
  DWORD; stdcall;
function WOWGlobalLockSize16(hMem: word; pcb: PDWORD): DWORD; stdcall;
function WOWGlobalUnlockFree16(vpMem: DWORD): word; stdcall;

{ Работа невытесняющего диспетчера Win16.

  Следующие две функции предназначены для кода Win32, вызываемого посредством
  общего санкинга (Generic Thunks), который нуждается в работе диспетчера
  Win16, чтобы эти задачи можно было выполнить за то время, пока санкинг
  ожидает завершения некоторой операции. Эти две функции идентичны
  обратному вызову для 16-разрядного кода, который вызывает процедуру
  Yield либо процедуру DirectedYield.}
procedure WOWYield16;
procedure WOWDirectedYield16(htask16: word);

{ Общие обратные вызовы.

  Функцию WOWCallback16 можно использовать в коде Win32, вызываемом
  из 16-разрядного кода (с помощью общего санкинга) для обратного вызова
```

в 16-разрядную часть. Эта функция должна быть объявлена по следующему образцу:

```
function CallbackRoutine(dwParam: Longint): Longint; export;
```

Если вы передаете указатель, объявите параметр следующим образом:

```
function CallbackRoutine(vp: Pointer): Longint; export;
```

ПРИМЕЧАНИЕ. При передаче указателя вам понадобится получить его с помощью функции WOWGlobalAlloc16 или WOWGlobalAllocLock16.

Если вызываемая функция возвращает слово, а не значение типа Longint, значит, старших 16 разрядов возвращаемого значения не определены. Аналогично, если вызываемая функция не имеет значения возврата, значит, полное возвращаемое значение не определено.

Функция WOWCallback16Ex разрешает любую комбинацию аргументов, но общий объем байтов, передаваемых 16-разрядной функции, не должен превышать значение WCB16_MAX_CBARGS. Параметр cbArgs используется для корректной очистки 16-разрядного стека после вызова функции. Независимо от значения cbArgs, значение WCB16_MAX_CBARGS (в байтах) будет всегда скопировано из параметра pArgs в 16-разрядный стек. Если значение pArgs меньше числа байтов WCB16_MAX_CBARGS от конца страницы, а следующая страница недоступна, функция WOWCallback16Ex понесет ответственность за нарушение прав доступа.

Если значение cbArgs больше значения WCB16_MAX_ARGS, поддерживаемого работающей системой, функция возвращает значение FALSE, а функция GetLastError – значение ERROR_INVALID_PARAMETER. В противном случае функция возвращает значение TRUE, а значение типа DWORD, на которое указывает параметр pdwRetCode, содержит код, возвращаемый функцией обратного вызова. Если функция обратного вызова возвращает значение типа WORD, значит, значение функции HIWORD() от кода возврата не определено и должно игнорироваться с помощью функции LOWORD(dwRetCode).

Функция WOWCallback16Ex может вызывать процедуры, используя соглашения о вызове PASCAL и CDECL. По умолчанию используется соглашение PASCAL. Чтобы воспользоваться соглашением CDECL, передайте в качестве параметра dwFlags значение WCB16_CDECL.

Аргументы, на которые указывает параметр pArgs, должны соблюдать порядок следования, соответствующий соглашению для функций обратного вызова. Для вызова функции SetWindowText

```
SetWindowText(Handle: hWnd; lpsz: PChar): Longint;
```

параметр pArgs должен указывать на массив слов:

```
SetWindowTextArgs: array[0..2] of word =
  (LoWord(Longint(lpsz)), HiWord(Longint(lpsz)), Handle);
```

```
  Другими словами, аргументы располагаются в массиве в обратном порядке,
  т.е. для указателей типа DWORD первым будет следовать наименее
  значимое слово,а для указателей типа FAR - смещение. Затем аргументы
  размещаются в массиве в порядке, указанном в прототипе функции, причем для
  указателей типа DWORD первым будет следовать наименее значимое слово,  а для
  указателей типа FAR - смещение.}
  function WOWCallback16(vpfn16, dwParam: DWORD): DWORD; stdcall;

const
  WCB16_MAX_CBARGS = 16;
  WCB16_PASCAL     = $0;
  WCB16_CDECL      = $1;

function WOWCallback16Ex(vpfn16, dwFlags, cbArgs: DWORD; pArgs: Pointer;
  pdwRetCode: PDWORD): BOOL; stdcall;

{ Функции преобразования дескрипторов 16 <--> 32.}

type
  TWOWHandleType = (
    WOW_TYPE_HWND,
    WOW_TYPE_HMENU,
    WOW_TYPE_HDWP,
    WOW_TYPE_HDROP,
    WOW_TYPE_HDC,
    WOW_TYPE_HFONT,
    WOW_TYPE_HMETAFILE,
    WOW_TYPE_HRGN,
    WOW_TYPE_HBITMAP,
    WOW_TYPE_HBRUSH,
    WOW_TYPE_HPALETTE,
    WOW_TYPE_HPEN,
    WOW_TYPE_HACCEL,
    WOW_TYPE_HTASK,
    WOW_TYPE_FULLHWND);

function WOWHandle16(Handle32: THandle; HandType: TWOWHandleType):
  word; stdcall;
function WOWHandle32(Handle16: word; HandleType: TWOWHandleType):
  THandle; stdcall;

implementation

const
  WOW32DLL = 'WOW32.DLL';
function WOWCallback16;
  external WOW32DLL name 'WOWCallback16';
function WOWCallback16Ex;
  external WOW32DLL name 'WOWCallback16Ex';
function WOWGetVDMPointer;
  external WOW32DLL name 'WOWGetVDMPointer';
```

```
function WOWGetVDMPointerFix;
  external WOW32DLL name 'WOWGetVDMPointerFix'
procedure WOWGetVDMPointerUnfix;
  external WOW32DLL name 'WOWGetVDMPointerUnfix'
function WOWGlobalAlloc16;
  external WOW32DLL name 'WOWGlobalAlloc16'
function WOWGlobalAllocLock16;
  external WOW32DLL name 'WOWGlobalAllocLock16';
function WOWGlobalFree16;
  external WOW32DLL name 'WOWGlobalFree16';
function WOWGlobalLock16;
  external WOW32DLL name 'WOWGlobalLock16';
function WOWGlobalLockSize16;
  external WOW32DLL name 'WOWGlobalLockSize16';
function WOWGlobalUnlock16;
  external WOW32DLL name 'WOWGlobalUnlock16';
function WOWGlobalUnlockFree16;
  external WOW32DLL name 'WOWGlobalUnlockFree16';
function WOWHandle16;
  external WOW32DLL name 'WOWHandle16';
function WOWHandle32;
  external WOW32DLL name 'WOWHandle32';
procedure WOWYield16;
  external WOW32DLL name 'WOWYield16';
procedure WOWDirectedYield16;
  external WOW32DLL name 'WOWDirectedYield16';
end.
```

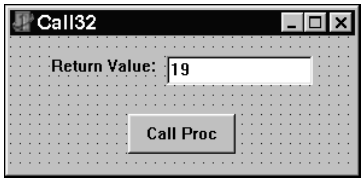
Для иллюстрации общего санкинга создана небольшая 32-разрядная DLL, которая будет вызываться из 16-разрядного приложения (32-разрядный проект DLL `TestDLL.dpr` представлен в листинге 13.16).

Листинг 13.16. Проект `TestDLL.dpr`

```
library TestDLL;
uses
  SysUtils, Dialogs, Windows, WOW32;
const
  DLLStr = 'Я нахожусь в 32-разрядной DLL. Вы послали строку: "%s"';
function DLLFunc32(P: PChar; CallBackFunc: DWORD): Integer; stdcall;
const
  MemSize = 256;
var
  Mem16: DWORD;
  Mem32: PChar;
  Hand16: word;
begin
  { Отображаем строку P. }
  ShowMessage(Format(DLLStr, [P]));
  { Выделяем 16-разрядную память. }
  Hand16 := WOWGlobalAlloc16(GMem_Share or GMem_Fixed or
```

```
    GMem_ZeroInit, MemSize);
  { Блокируем 16-разрядную память. }
  Mem16 := WOWGlobalLock16(Hand16);
  { Преобразуем 16-разрядный указатель в 32-разрядный.
    Теперь они указывают на одно и то же место. }
  Mem32 := PChar(WOWGetVDMPointer(Mem16, MemSize, True));
  { Копируем строку в 32-разрядный указатель. }
  StrPCopy(Mem32, 'I REALLY love DDG!!');
  { Выполняем обратный вызов в 16-разрядном приложении,
    передавая 16-разрядный указатель. }
  Result := WOWCallback16(CallBackFunc, Mem16);
  { Освобождаем выделенную 16-разрядную память. }
  WOWGlobalUnlockFree16(Mem16);
end;
exports
  DLLFunc32 name 'DLLFunc32' resident;
begin
end.
```

Эта DLL экспортирует одну функцию, которая принимает в качестве параметров строку типа `PChar` и функцию обратного вызова. Строка немедленно отображается с помощью функции `ShowMessage()`. Функция обратного вызова позволяет выполнить обратный вызов в 16-разрядном процессе путем передачи некоторого объема специально выделенной 16-разрядной памяти.



Код 16-разрядного приложения `Call32.dpr` представлен в листинге 13.17, а главная форма приложения показана на рис. 13.6.

Рис. 13.6. Главная форма проекта `Call32`

Листинг 13.17. Модуль `Main.pas`

```
unit Main;
{$C FIXED DEMANDLOAD PERMANENT}

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    CallBtn: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    procedure CallBtnClick(Sender: TObject);
  private
    { Закрытые объявления }
```

```
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation
{$R *.DFM}

uses WOW16;

const
  ExeStr = 'The 32-bit DLL has called back into the 16-bit EXE. ' +
           'The string to the EXE is: "%s"';
function CallBackFunc(P: PChar): Longint; export;
begin
  ShowMessage(Format(ExeStr, [StrPas(P)]));
  Result := StrLen(P);
end;

procedure TMainForm.CallBtnClick(Sender: TObject);
var
  H: THandle32;
  R, P: Longint;
  AStr: PChar;
begin
  { Загружаем 32-разрядную DLL }
  H := LoadLibraryEx32W('TestDLL.dll', 0, 0);
  AStr := StrNew('I love DDG.');
  try
    if H > 0 then begin
      { Считываем адрес процедуры из 32-разрядной DLL }
      TFarProc(P) := GetProcAddress32W(H, 'DLLFunc32');
      if P > 0 then begin
        { Вызываем процедуру в 32-разрядной DLL }
        R := Call32BitProc(P, [Longint(AStr),
          Longint(@CallBackFunc)], 1);
        Edit1.Text := IntToStr(R);
      end;
    end;
finally
    StrDispose(AStr);
    if H > 0 then FreeLibrary32W(H);
  end;
end;

end.
```

Это приложение передает 16-разрядную строку типа PChar и адрес функции в 32-разрядную библиотеку DLL. Функция CallBackFunc() в конечном счете вызывается в 32-разрядной библиотеке DLL. И в самом деле, если внимательнее рассмотреть листинг, то вы увидите, что значение, возвращаемое функцией DLLFunc32(), является значением возврата функции CallBackFunc().

Сообщение WM_COPYDATA

Для вызова 16-разрядных DLL из 32-разрядных приложений в Windows 95/98 предусмотрена поддержка линейного санкинга. Windows NT/2000 не предоставляет средств для непосредственного вызова 16-разрядного кода из 32-разрядного приложения. В связи с этим ограничением возникает вопрос: как же тогда лучше всего связывать данные между 32- и 16-разрядными процессами в среде Windows NT? За этим вопросом возникает следующий: существует ли простой способ совместного использования данных, который "работает" под управлением как Windows 95, так и Windows NT?

Ответ на оба вопроса один: сообщение WM_COPYDATA. Сообщение Windows WM_COPYDATA предоставляет средства для передачи двоичных данных между процессами — либо 32-, либо 16-разрядными. При отправке окну сообщения WM_COPYDATA параметр wParam этого сообщения идентифицирует окно, передающее данные, а параметр lParam содержит указатель на запись TCopyDataStruct. Эта запись определяется следующим образом:

```
type
  PCopyDataStruct = ^TCopyDataStruct;
  TCopyDataStruct = packed record
    dwData: DWORD;
    cbData: DWORD;
    lpData: Pointer;
  end;
```

В поле dwData хранятся 32 разряда информации, определенной пользователем. В поле cbData содержится размер буфера, на который указывает содержимое поля lpData. Другими словами, поле lpData — это указатель на буфер с информацией, которую требуется передать между приложениями. Если это сообщение посылается между 32- и 16-разрядными приложениями, Windows автоматически преобразует указатель lpData из его линейной формы (0:32) в указатель 16:16 или наоборот. Кроме того, Windows гарантирует, что данные, на которые указывает поле lpData , отображаются на адресное пространство принимающего процесса.

На заметку Сообщение WM_COPYDATA прекрасно работает в случае относительно малых объемов информации. Но если через границу между 16- и 32-разрядными приложениями нужно переслать большие информационные массивы, целесообразнее использовать автоматизацию, описанную в главе 23 второго тома, "COM-ориентированные технологии".

Хотя в Windows NT не поддерживается непосредственное использование 16-разрядных библиотек DLL из приложений Win32, тем не менее можно создать 16-разрядный выполняемый файл, который будет инкапсулировать библиотеку DLL и связываться с вашим приложением с помощью сообщения WM_COPYDATA.

Для демонстрации использования сообщения WM_COPYDATA созданы два проекта, первый из которых играет роль 32-разрядного приложения. В этом приложении используется элемент управления, в который можно вводить некоторый текст. Кроме того, в нем предусмот-

рены средства для связи со вторым проектом, 16-разрядным приложением. Связь заключается в передаче текста примечания. Для того чтобы средство коммуникации между двумя приложениями заработало, выполните следующие действия.

1. Зарегистрируйте сообщение окна, чтобы получить уникальный идентификатор сообщения для связи между приложениями.
2. Передайте общесистемное сообщение из приложения Win32. В параметре `wParam` этого сообщения сохраните дескриптор для главного окна приложения Win32.
3. Когда 16-разрядное приложение получит переданное сообщение, оно ответит отправкой зарегистрированного сообщения обратно приложению-отправителю и в качестве параметра `wParam` передаст дескриптор окна собственной главной формы.
4. Получив ответ, 32-разрядное приложение будет иметь дескриптор главной формы 16-разрядного приложения. Теперь 32-разрядное приложение сможет отправить сообщение `WM_COPYDATA` 16-разрядному приложению, чтобы активизировать процесс разделения данных.

В листинге 13.18 представлен исходный текст модуля `RegMsg.pas`, который используется двумя проектами.

Листинг 13.18. Модуль `RegMsg.pas`

```
unit RegMsg;

interface

var
  DDGM_HandshakeMessage: Cardinal;

implementation

uses WinProcs;

const
  HandshakeMessageStr: PChar = 'DDG.CopyData.Handshake';

initialization
  DDGM_HandshakeMessage := RegisterWindowMessage(HandshakeMessageStr);
end.
```

Исходный код модуля `CopyMain.pas`, который входит в состав 32-разрядного проекта `CopyData.dpr`, содержится в листинге 13.19. В этом модуле закладывается фундамент обмена сообщениями между приложениями и выполняется пересылка данных.

Листинг 13.19. Модуль `CopyMain.pas`

```
unit CopyMain;

interface

uses
```

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

type
  TMainForm = class(TForm)
    DataMemo: TMemo;
    BottomPnl: TPanel;
    BtnPnl: TPanel;
    CloseBtn: TButton;
    CopyBtn: TButton;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    CopyData1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    procedure CloseBtnClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure About1Click(Sender: TObject);
    procedure CopyBtnClick(Sender: TObject);
  private
    { Закрытые объявления }
  protected
    procedure WndProc(var Message: TMessage); override;
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation
{$R *.DFM}

uses AboutU, RegMsg;

{ Следующее объявление необходимо из-за ошибки в объявлении
  функции BroadcastSystemMessage() в модуле Windows }
function BroadcastSystemMessage(Flags: DWORD; Recipients: PDWORD;
  uiMessage: UINT; wParam: WPARAM; lParam: LPARAM): Longint; stdcall;
  external 'user32.dll';

var
  Recipients: DWORD = BSM_APPLICATIONS;

procedure TMainForm.WndProc(var Message: TMessage);
var
  DataBuffer: TCopyDataStruct;
  Buf: PChar;
  BufSize: Integer;
```

```
begin
  if Message.Msg = DDGM_HandshakeMessage then begin
    { Выделяем буфер }
    BufSize := DataMemo.GetTextLen + (1 * SizeOf(Char));
    Buf := AllocMem(BufSize);
    { Копируем примечание в буфер }
    DataMemo.GetTextBuf(Buf, BufSize);
    try
      with DataBuffer do begin
        { Заполняем поле dwData зарегистрированным сообщением для
          контроля безопасности }
        dwData := DDGM_HandshakeMessage;
        cbData := BufSize;
        lpData := Buf;
      end;
      { ПРИМЕЧАНИЕ: сообщение WM_COPYDATA должно быть отправлено
        с помощью функции SendMessage. }
      SendMessage(Message.wParam, WM_COPYDATA, Handle,
        Longint(@DataBuffer));
    finally
      FreeMem(Buf, BufSize);
    end;
  end
  else
    inherited WndProc(Message);
end;

procedure TMainForm.CloseBtnClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormResize(Sender: TObject);
begin
  BtnPnl.Left := BottomPnl.Width div 2 - BtnPnl.Width div 2;
end;

procedure TMainForm.About1Click(Sender: TObject);
begin
  AboutBox;
end;

procedure TMainForm.CopyBtnClick(Sender: TObject);
begin
  { Вызываем для любых "слушающих" приложений }
  BroadcastSystemMessage(BSF_IGNORECURRENTTASK or BSF_POSTMESSAGE,
    @Recipients, DDGM_HandshakeMessage, Handle, 0);
end;

end.
```

В листинге 13.20 представлен исходный текст главного модуля ReadMain.pas 16-разрядного проекта ReadData.dpr. Этот модуль связывается с проектом CopyData и принимает буфер данных.

Листинг 13.20. Модуль ReadMain.pas

```
unit Readmain;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, Menus, StdCtrls;

{ Сообщение Windows WM_COPYDATA не определяется в 16-разрядном модуле
  Messages, хотя оно и доступно для 16-разрядных приложений, работающих
  под управлением Windows 95 или NT. Это сообщение описано в электронной
  справочной системе Win32 API. }
const
  WM_COPYDATA = $004A;

type
  TMainForm = class(TForm)
    ReadMemo: TMemo;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    procedure Exit1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure About1Click(Sender: TObject);
  private
    procedure OnAppMessage(var M: TMsg; var Handled: Boolean);
    procedure WMCopyData(var M: TMessage); message WM_COPYDATA;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses RegMsg, AboutU;

type
  { Тип записи TCopyDataStruct не определен в модуле WinTypes, хотя
  он доступен в 16-разрядном интерфейсе Windows API, работающем под
  управлением Windows 95 и NT. Параметр lParam сообщения WM_COPYDATA
```

```
  указывает на одну из записей этого типа. }
  PCopyDataStruct = ^TCopyDataStruct;
  TCopyDataStruct = record
    dwData: Longint;
    cbData: Longint;
    lpData: Pointer;
  end;

procedure TMainForm.OnAppMessage(var M: TMsg; var Handled: Boolean);
{ Обработчик события OnMessage для объекта Application. }
begin
  { Сообщение DDGM_HandshakeMessage принимается как общесистемное,
  предназначенное для всех приложений. Параметр wParam этого сообщения
  содержит дескриптор окна - отправителя этого сообщения. Мы отвечаем
  отправкой того же самого сообщения назад отправителю, указывая наш
  дескриптор в параметре wParam. }
  if M.Message = DDGM_HandshakeMessage then begin
    PostMessage(M.wParam, DDGM_HandshakeMessage, Handle, 0);
    Handled := True;
  end;
end;

procedure TMainForm.WMCopyData(var M: TMessage);
{ Обработчик сообщения WM_COPYDATA. }
begin
  { Проверяем параметр wParam для подтверждения того, что мы знаем,
  КТО послал нам сообщение WM_COPYDATA }
  if PCopyDataStruct(M.lParam)^.dwData = DDGM_HandshakeMessage then
    { После получения сообщения WM_COPYDATA
      параметр lParam указывает на запись. }
    ReadMemo.SetTextBuf(PChar(PCopyDataStruct(M.lParam)^.lpData));
end;
procedure TMainForm.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnMessage := OnAppMessage;
end;

procedure TMainForm.About1Click(Sender: TObject);
begin
  AboutBox;
end;
end.
```

На рис. 13.7 показана согласованная работа двух приложений.

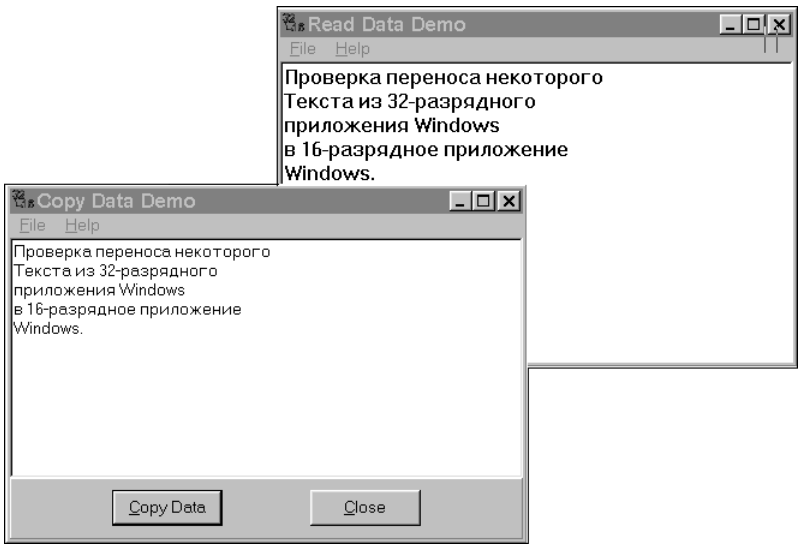


Рис. 13.7. Связь с помощью сообщения `WM_COPYDATA`

Получение информации о пакете

Пакеты представляют собой удобное средство логического и физического разделения приложения на отдельные модули. Они содержат откомпилированные двоичные блоки, состоящие из одного или нескольких исходных модулей, и позволяют ссылаться (из них) на другие модули, содержащиеся в иных пакетах. Безусловно, если вы располагаете исходным текстом модулей, помещенных в определенный пакет, то не составит труда выяснить, какие именно модули входят в его состав и какие другие пакеты потребуются им при выполнении. Но как поступить, если эту информацию требуется получить для модуля, исходный текст которого отсутствует? К счастью, эта задача вполне решима, и все, что потребуется — это написать несколько строк программного текста. Фактически, всю эту информацию можно получить с помощью процедуры `GetPackageInfo()`, которая входит в состав модуля `SysUtils`. Процедура `GetPackageInfo()` определяется в этом модуле следующим образом:

```
procedure GetPackageInfo(Module: HMODULE; Param: Pointer; var Flags: Integer;
  InfoProc: TPackageInfoProc);
```

Параметр `Module` представляет собой дескриптор Win32 API для файла пакета, подобный дескриптору, возвращаемому функцией Win32 API `LoadLibrary()`.

Параметр `Param` определяет состав необходимых пользователю данных, которые будут переданы процедуре, указанной параметром `InfoProc`.

В параметре `Flags` указывается определенная информация о пакете. Значение этого параметра представляет собой комбинацию из флажков, описанных в табл. 13.6.

Таблица 13.6. Флажки функции GetPackageInfo()

Флажок	Значение	Описание
pfNeverBuild	$00000001	"Никогда не создаваемый" пакет
pfDesignOnly	$00000002	Пакет времени проектирования
pfRunOnly	$00000004	Пакет времени выполнения
pfIgnoreDupUnits	$00000008	Наличие в этом пакете нескольких экземпляров одного и того же модуля игнорируется
pfModuleTypeMask	$C0000000	Для определения типа модуля используется маска
pfExeModule	$00000000	Пакет представляет собой EXE-файл (не используется)
pfPackageModule	$40000000	Пакет представляет собой файл пакета
pfProducerMask	$0C000000	Для определения создавшего этот пакет продукта используется маска
pfV3Produced	$00000000	Пакет создан в Delphi 3 или BCB 3.
PfProducerUndefined	$04000000	Среда создания этого пакета не определена
pfBCB4Produced	$08000000	Пакет создан в BCB 4
pfDelphi4Produced	$0C000000	Пакет создан в Delphi 4
pfLibraryModule	$80000000	Пакет представляет собой DLL-файл

Параметр `InfoProc` определяет процедуру обратного вызова, которая будет вызвана по одному разу для каждого пакета, использование которого требует данный пакет, а также для каждого модуля в этом пакете. Этот параметр должен иметь тип `TPackageInfoProc`, определяемый следующим образом:

```
type
  TNameType = (ntContainsUnit, ntRequiresPackage);
  TPackageInfoProc = procedure (const Name: string; NameType: TNameType;
    Flags: Byte; Param: Pointer);
```

В этом процедурном типе параметр `Name` задает имя пакета или модуля; параметр `NameType` определяет, что это — пакет или модуль; параметр `Flags` предоставляет дополнительную информацию о файле; параметр `Param` содержит некоторые предоставленные пользователем данные, предназначенные для передачи методу `GetPackageInfo()`.

Для демонстрации методов использования процедуры `GetPackageInfo()` было разработано приложение, предназначенное для получения информации о любом пакете. Данный проект называется `PackInfo` — текст его модуля проекта представлен в листинге 13.21.

Листинг 13.21. Файл `PackInfo.dpr` — файл проекта демонстрационного приложения

```
program PkgInfo;

uses
  Forms,
  Dialogs,
```

```
  SysUtils,
  PkgMain in 'PkgMain.pas' {PackInfoForm};
{$R *.RES}

var
  OpenDialog: TOpenDialog;
begin
  if (ParamCount > 0) and FileExists(ParamStr(1)) then
    PkgName := ParamStr(1)
  else begin
    OpenDialog := TOpenDialog.Create(Application);
    OpenDialog.DefaultExt := '*.bpl';
    OpenDialog.Filter := 'Packages (*.bpl)|*.bpl|Delphi 3 Packages ' +
      '(*.dpl)|*.dpl';
    if OpenDialog.Execute then PkgName := OpenDialog.FileName;
  end;
  if PkgName <> '' then
  begin
    Application.Initialize;
    Application.CreateForm(TPackInfoForm, PackInfoForm);
    Application.Run;
  end;

end.
```

Если в приложение не передаются никакие параметры командной строки, работа его начинается с предоставления пользователю диалогового окна открытия файла. В этом окне пользователь должен указать требуемый файл пакета. В любом случае — как при получении имени файла пакета в виде параметра командной строки, так и при выборе пользователем требуемого файла в диалоговом окне — указанное имя файла назначается переменной `PkgName` и приложение продолжает свою работу.

Текст главного модуля приложения `PackInfo` представлен в листинге 13.22. Именно в этом модуле осуществляется вызов процедуры `GetPackageInfo()`.

Листинг 13.22. Текст модуля `PkgMain.pas`, выполняющего выборку информации о пакете

```
unit PkgMain;

interface

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

type
  TPackInfoForm = class(TForm)
    GroupBox1: TGroupBox;
    DsgnPkg: TCheckBox;
```

```
    RunPkg: TCheckBox;
    BuildCtl: TRadioGroup;
    GroupBox2: TGroupBox;
    GroupBox3: TGroupBox;
    Button1: TButton;
    Label1: TLabel;
    DescEd: TEdit;
    memContains: TMemo;
    memRequires: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  end;

var
  PackInfoForm: TPackInfoForm;
  PkgName: string;  // Это значение определяется в файле проекта

implementation

{$R *.DFM}

procedure PackageInfoCallback(const Name: string; NameType: TNameType;
  Flags: Byte; Param: Pointer);
var
 AddName: string;
  Memo: TMemo;
begin
  Assert(Param <> nil);
  AddName := Name;
  case NameType of
    ntContainsUnit: Memo := TPackInfoForm(Param).memContains;
    ntRequiresPackage: Memo := TPackInfoForm(Param).memRequires;
  else
    Memo := nil;
  end;
  if Memo <> nil then
  begin
    if Memo.Text <> '' then AddName := ', ' + AddName;
    Memo.Text := Memo.Text + AddName;
  end;
end;

procedure TPackInfoForm.FormCreate(Sender: TObject);
var
  PackMod: HMODULE;
  Flags: Integer;
begin
  { Поскольку необходимо получить доступ только к ресурсам пакета,
  вызов функции API LoadLibraryEx() с флагом LOAD_LIBRARY_AS_DATAFILE
    позволит ускорить процедуру загрузки пакета }
  PackMod := LoadLibraryEx(PChar(PkgName), 0, LOAD_LIBRARY_AS_DATAFILE);
  if PackMod = 0 then Exit;
  try
```

```
    GetPackageInfo(PackMod, Pointer(Self), Flags, PackageInfoCallback);
  finally
    FreeLibrary(PackMod);
  end;
  Caption := 'Package Info: ' + ExtractFileName(PkgName);
  DsgnPkg.Checked := Flags and pfDesignOnly <> 0;
  RunPkg.Checked := Flags and pfRunOnly <> 0;
  if Flags and pfNeverBuild <> 0 then
    BuildCtl.ItemIndex := 1;
  DescEd.Text := GetPackageDescription(PChar(PkgName));
end;

procedure TPackInfoForm.Button1Click(Sender: TObject);
begin
  Close;
end;

end.
```

Создается впечатление, что текст этого модуля непропорционально мал по сравнению с объемом предоставляемой им низкоуровневой информации. После создания формы выполняется загрузка требуемого пакета, после чего вызывается процедура `GetPackageInfo()` и обновляются некоторые элементы интерфейса пользователя. В параметре `InfoProc` процедуре передается адрес входа в метод `PackageInfoCallback()`. В этом методе имена пакетов и модулей помещаются в соответствующие компоненты `TMemo`. На рис. 13.8 показана форма приложения `PackInfo`, содержащая сведения об одном из пакетов Delphi.

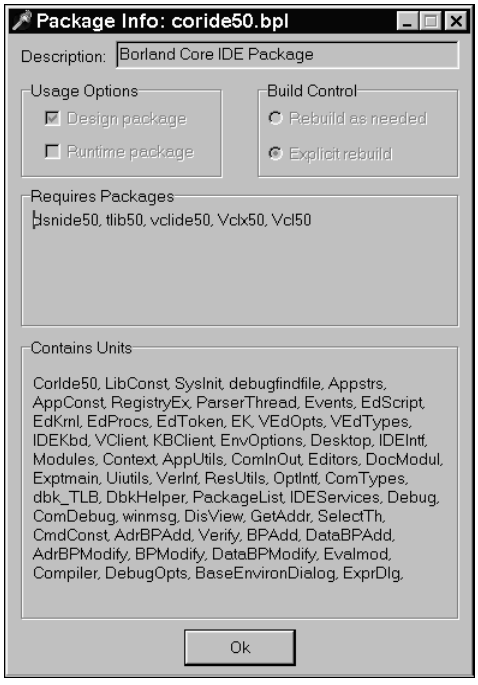


Рис. 13.8. Получение сведений о пакете с помощью приложения `PackInfo`

Резюме

Вот, пожалуй, и все. Попробуем остановиться и окинуть взглядом все, что обсуждалось в этой главе: процедуры подмены окон, запрещение запуска нескольких экземпляров приложений, ловушки Windows, программирование на языке BASM, использование объектных файлов C++, использование классов C++, санкинг, использование сообщения `WM_COPYDATA` и получение информации о пакетах. Поскольку мы уже достаточно углубились в профессиональные секреты программирования на системном уровне, то в следующей главе уместна будет попытка проникнуть в самое сердце операционной системы с целью выборки информации о функционировании процессов, потоков и модулей.

Глава

14

Получение системной информации

В этой главе мы рассмотрим, как создать полнофункциональную утилиту SysInfo, предназначенную для просмотра жизненно важных параметров системы. В процессе разработки этого приложения вы познакомитесь с менее известными функциями API, позволяющими получить доступ к низкоуровневой общесистемной информации о процессах, потоках, модулях, кучах, драйверах и страницах. В этой главе также рассматриваются различные способы получения этой информации в Windows 95/98 и Windows NT. С помощью SysInfo можно будет получить информацию о свободных ресурсах памяти, данные о версии Windows, значения переменных окружения, а также список загруженных модулей. При этом мы не только будем вникать в мельчайшие детали использования функций API, но и обсудим, как интегрировать полученную от системы информацию в функциональный и эстетически привлекательный интерфейс пользователя. Кроме того, вы узнаете, какие функции Win32 были разработаны для замены устаревших функций API Windows 3.x.

Для чего же может понадобиться информация о состоянии Windows? Во-первых, для удовлетворения собственного любопытства (ведь все мы в душе немножечко хакеры, и одно только ощущение больших возможностей делает нас еще сильнее). Во-вторых, может появиться необходимость написать программу, в которой потребуется получить доступ к переменным окружения с целью найти определенные файлы. В-третьих, у вас может возникнуть необходимость в информации обо всех загруженных модулях, чтобы вручную удалить их из памяти. В-четвертых, в-пятых... — словом, всех причин не перечесть.

Получение общей информации о системе

Для начала покажем, как получить системную информацию с помощью функции API, которая остается постоянной при смене версий Win32. Код этого приложения приобретет больше смысла, если сначала познакомиться с его пользовательским интерфейсом. Причем на сей раз сделаем это “с черного хода”, т.е. начнем с дочерней формы приложения. Эта форма показана на рис. 14.1 и называется InfoForm. Она используется для отображения различных параметров системы и процессов, а именно: информации о памяти и аппаратных средствах, версии операционной системы (ОС), некоторых данных о каталогах и переменных окружения.

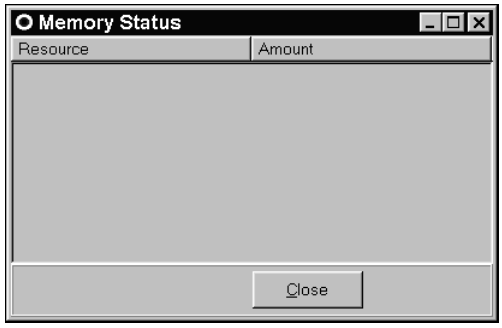


Рис. 14.1. Дочерняя форма InfoForm

Содержимое формы очень простое. В ее состав входит пользовательский компонент THeaderListbox (см. главу 21 второго тома, “Создание пользовательских компонентов в Delphi”) и кнопка (компонент TButton). Напомню, что элемент управления THeaderListbox

представляет собой сочетание элементов управления THeader (заголовок) и TListbox (список). При изменении размера разделов заголовка размер окна списка также меняется соответствующим образом. Элемент управления THeaderListbox (именуемый InfoLB) предназначен для отображения перечисленных выше типов информации.

Форматирование строк

В этом приложении широко используется функция Format() для форматирования встроенных строк, объединяемых с данными, полученными от ОС во время выполнения приложения. Эти строки определяются в разделе const главного модуля следующим образом:

```
const
  { Строки состояния памяти }
  SMemUse   = 'Memory in useq%d%%';
        // Используемая память в байтах
  STotMem   = 'Total physical memoryq$%.8x bytes';
        // Общая физическая память в байтах
  SFreeMem  = 'Free physical memoryq$%.8x bytes';
        // Свободная физическая память в байтах
  STotPage  = 'Total page file memoryq$%.8x bytes';
        // Общая память файлов подкачки в байтах
  SFreePage = 'Free page file memoryq$%.8x bytes';
        // Свободная память файлов подкачки в байтах
  STotVirt  = 'Total virtual memoryq$%.8x bytes';
        // Общая виртуальная память в байтах
  SFreeVirt = 'Free virtual memoryq$%.8x bytes';
        // Свободная виртуальная память в байтах
  { Строки информации о версии ОС }
  SOSVer    = 'OS Versionq%d.%d';  // Версия операционной системы
  SBuildNo  = 'Build Numberq%d';   // Номер версии
  SOSPlat   = 'Platformq%s';       // Платформа
  SOSWin32s = 'Windows 3.1x running Win32s';
  SOSWin95  = 'Windows 95';
  SOSWinNT  = 'Windows NT';
  { Строки системной информации }
  SProc     = 'Processor Arhitectureq%s';
        // Архитектура процессора
  SPIntel   = 'Intel';
  SPageSize = 'Page Sizeq$%.8x bytes';
        // Размер страницы
  SMinAddr  = 'Minimum Application Addressq$%p';
        // Минимальный адрес приложения
  SMaxAddr  = 'Maximum Application Addressq$%p';
        // Максимальный адрес приложения
  SNumProcs = 'Number of Processorsq%d';
        // Число процессоров
  SAllocGra = 'Allocation Granularityq$%.8x bytes';
        // Степень разбиения при выделении ресурсов
  SProcLevl = 'Processor Levelq%s';
        // Уровень процессора
```

```
  SIntel3   = '80386';
  SIntel4   = '80486';
  SIntel5   = 'Pentium';
  SIntel6   = 'Pentium Pro';
  SProcRev  = 'Processor Revisionq%.4x';
        // Модификация процессора
  { Строки информации о каталогах }
  SWinDir   = 'Windows directoryq%s';
        // Каталог Windows
  SSysDir   = 'Windows system directoryq%s';
        // Системный каталог Windows
  SCurDir   = 'Current directoryq%s';
        // Текущий каталог
```

Возможно, вас удивило присутствие буквы `q` в середине каждой строки. При отображении этих строк свойство `DelimChar` элемента управления `InfoLB` установлено равным `q`, а это значит, что символ `q` используется в окне списка в качестве разделителя между столбцами.

Существует три основные причины использования функции `Format()` со встроенными строками вместо отдельного форматирования строковых литералов.

- Лаконичность. Поскольку функции `Format()` передаются в качестве параметров различные типы данных, не требуется "утяжелять" программу вызовами таких функций, как `IntToStr()` или `IntToHex()`, которые форматируют различные типы параметров для отображения на экране.
- Гибкость. Функция `Format()` легко справляется с несколькими типами данных. В этом случае строки формата `%s` и `%d` используются для форматирования строковых и числовых данных, что является более гибким вариантом обработки данных.
- Простота поддержки. Хранение строк в отдельном программном блоке позволяет быстрее и проще найти их, дополнить или отредактировать в случае необходимости.

На заметку Для отображения одиночного символа процента в форматированной строке используйте двойной знак процента (`%%`).

Получение информации о состоянии памяти

Первый набор системной информации, предназначенный для заполнения элемента управления `InfoLB`, относится к определению состояния памяти — его можно получить с помощью вызова функции API `GlobalMemoryStatus()`. Этой процедуре передается один `var`-параметр типа `TMemoryStatus`. Запись `TMemoryStatus` определяется следующим образом:

```
type
  TMemoryStatus = record
    dwLength: DWORD;
    dwMemoryLoad: DWORD;
    dwTotalPhys: DWORD;
    dwAvailPhys: DWORD;
    dwTotalPageFile: DWORD;
```

```
    dwAvailPageFile: DWORD;
    dwTotalVirtual: DWORD;
    dwAvailVirtual: DWORD;
  end;
```

- Первое поле в этой записи, `dwLength`, описывает длину записи `TMemoryStatus`. Необходимо инициализировать это поле значением `SizeOf(TMemoryStatus)` до вызова функции `GlobalMemoryStatus()`. Это позволит Windows изменять размер записи в будущих версиях, поскольку она сможет различать версии на основе значения первого поля.
- В поле `dwMemoryLoad` содержится число от `0` до `100`, на основании которого можно получить общее представление об использовании памяти: `0` означает, что память вообще не используется, а `100` говорит о занятости всей памяти.
- Поле `dwTotalPhys` показывает общее число байтов физической памяти (объем памяти ОЗУ, установленного в компьютере), а поле `dwAvailPhys` — объем свободной в данный момент физической памяти.
- Поле `dwTotalPageFile` показывает общее число байтов, которое может быть выведено на жесткий диск в файл (файлы) подкачки. Это число не обязательно совпадает с размером файла подкачки на диске. Поле `dwAvailPageFile` определяет объем еще доступной памяти на основании этого общего значения.
- Поле `dwTotalVirtual` показывает общее число байтов виртуальной памяти, используемой в вызывающем процессе, а поле `dwAvailVirtual` — объем этой памяти, доступной для вызывающего процесса.

С помощью следующего кода можно получить информацию о состоянии памяти, а также заполнить список этой информацией:

```
procedure TInfoForm.ShowMemStatus;
var
  MS: TMemoryStatus;
begin
  InfoLB.DelimChar := 'q';
  MS.dwLength := SizeOf(MS);
  GlobalMemoryStatus(MS);
  with InfoLB.Items, MS do
  begin
    Clear;
    Add(Format(SMemUse, [dwMemoryLoad]));
    Add(Format(STotMem, [dwTotalPhys]));
    Add(Format(SFreeMem, [dwAvailPhys]));
    Add(Format(STotPage, [dwTotalPageFile]));
    Add(Format(SFreePage, [dwAvailPageFile]));
    Add(Format(STotVirt, [dwTotalVirtual]));
    Add(Format(SFreeVirt, [dwAvailVirtual]));
  end;
  InfoLB.Sections[0].Text := 'Resource'; // Ресурс
  InfoLB.Sections[1].Text := 'Amount'; // Объем
  Caption:= 'Memory Status'; // Состояние памяти
end;
```

Не забудьте перед вызовом функции `GlobalMemoryStatus()` инициализировать поле `dwLength`.

На рис. 14.2 показана форма `InfoForm`, отображающая информацию о состоянии памяти.

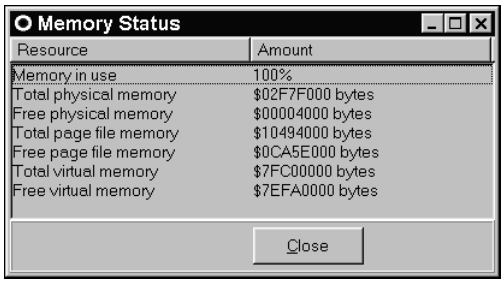


Рис. 14.2. Отображение информации о состоянии памяти

Получение информации о версии операционной системы

Получить информацию об установленной на данном компьютере версии Windows и Win32 можно с помощью функции API `GetVersionEx()`. Этой функции по ссылке передается только один параметр — запись `TOSVersionInfo`, которая определяется следующим образом:

```
type
  TOSVersionInfo = record
    dwOSVersionInfoSize: DWORD;
    dwMajorVersion: DWORD;
    dwMinorVersion: DWORD;
    dwBuildNumber: DWORD;
    dwPlatformId: DWORD;
    szCSDVersion: array[0..126] of AnsiChar; { Строка поддержки для PSS }
  end;
```

- Поле `dwOSVersionInfoSize` должно быть инициализировано значением `SizeOf(TOSVersionInfo)` до вызова функции `GetVersionEx()`.
- Поле `dwMajorVersion` содержит старший номер версии ОС. Другими словами, если версия ОС имеет номер 4.0, то в этом поле будет записано число `4`.
- Поле `dwMinorVersion` содержит младший номер версии ОС. Другими словами, если версия ОС имеет номер 4.0, то в этом поле будет записано число `0`.
- В поле `dwBuildNumber` содержится номер ОС, который хранится в ее самом младшем слове.
- Поле `dwPlatformId` описывает текущую платформу Win32. Этот параметр может иметь любое из перечисленных в табл. 14.1 значений.

Таблица 14.1. Обозначения различных платформ Windows

Значение	Платформа
VER_PLATFORM_WIN32s	Win32s в Windows 3.1
VER_PLATFORM_WIN32_WINDOWS	Win32 в Windows 95
VER_PLATFORM_WIN32_NT	Windows NT

- Поле szCSDVersion содержит дополнительную информацию об ОС. Это поле часто хранит пустую строку.

Следующая процедура заполняет элемент управления InfoLB информацией о версии ОС:

```
procedure TInfoForm.GetOSVerInfo;
var
  VI: TOSVersionInfo;
begin
  VI.dwOSVersionInfoSize := SizeOf(VI);
  GetVersionEx(VI);
  with InfoLB.Items, VI do
  begin
    Clear;
    Add(Format(SOSVer, [dwMajorVersion, dwMinorVersion]));
    Add(Format(SBuildNo, [LoWord(dwBuildNumber)]));
    case dwPlatformID of
      VER_PLATFORM_WIN32S: Add(Format(SOSPlat, [SOSWin32s]));
      VER_PLATFORM_WIN32_WINDOWS: Add(Format(SOSPlat, [SOSWin95]));
      VER_PLATFORM_WIN32_NT: Add(Format(SOSPlat, [SOSWinNT]));
    end;
  end;
end;
```

На заметку В Windows 3.x аналогичную информацию о версии ОС можно было получить с помощью функции GetVersion(). Но поскольку теперь вы работаете под управлением Win32, используйте функцию GetVersionEx(), которая предоставляет более детальную информацию, чем функция GetVersion().

Получение информации о каталогах

Операционная система активно использует каталоги Windows и System для хранения библиотек DLL, драйверов, приложений и INI-файлов. Кроме того, Win32 поддерживает текущий каталог для каждого процесса. Вполне вероятно, что при создании различных приложений Win32 может потребоваться получить точную информацию о расположении этих каталогов. В этом случае следует использовать три функции Win32 API, с помощью которых эти данные легко можно будет получить.

О функциях GetWindowsDirectory(), GetSystemDirectory() и GetCurrentDirectory() можно сказать, что они очень просты в применении. Каждой из них в качестве первого параметра передается указатель на буфер, в который копируется строка результата, а в качестве второго параметра — размер этого буфера. Результат, копируемый в буфер, представляет собой строку, содержащую путь, причем эта строка завершается ограничивающим нуль-символом.

Надеемся, что по имени функции вы сможете легко определить, информацию о каком каталоге возвращает каждая функция. Если это не так, остается надеяться лишь на то, что вы зарабатываете себе на жизнь чем-то, отличным от программирования.

В этом методе используется временный массив элементов типа `char`, в которые заносится информация о соответствующем каталоге. Затем информация из массива добавляется в элемент управления `InfoLB`, как в следующем коде:

```
procedure TInfoForm.GetDirInfo;
var
  S: array[0..MAX_PATH] of char;
begin
  { Получаем каталог Windows }
  GetWindowsDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SWinDir, [S]));
  { Получаем системный каталог Windows }
  GetSystemDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SSysDir, [S]));
  { Получаем текущий каталог процесса}
  GetCurrentDirectory(SizeOf(S), S);
  InfoLB.Items.Add(Format(SCurDir, [S]));
end;
```

На заметку Функции `GetWindowsDir()` и `GetSystemDir()` из Windows 3.x API под управлением Win32 недоступны.

Получение системной информации

В интерфейсе Win32 API предусмотрена процедура `GetSystemInfo()`, которая считывает некоторые специфические подробности о работе операционной системы. Этой процедуре по ссылке передается один параметр типа `TSystemInfo`, после чего она заполняет указанную запись соответствующими значениями. Запись `TSystemInfo` определяется следующим образом:

```
type
  PSystemInfo = ^TSystemInfo;
  TSystemInfo = record
    case Integer of
      0: (
        dwOemId: DWORD);
      1: (
        wProcessorArchitecture: Word;
        wReserved: Word;
        dwPageSize: DWORD;
        lpMinimumApplicationAddress: Pointer;
        lpMaximumApplicationAddress: Pointer;
        dwActiveProcessorMask: DWORD;
        dwNumberOfProcessors: DWORD;
        dwProcessorType: DWORD;
        dwAllocationGranularity: DWORD;
```

```
      wProcessorLevel: Word;
      wProcessorRevision: Word);
end;
```

- Поле `dwOemId` используется для Windows 95. Это поле всегда устанавливается равным `0` или `PROCESSOR_ARCHITECTURE_INTEL`.
- В среде Windows NT используется поле `wProcessorArchitecture`, которое описывает тип архитектуры процессора, установленного на данном компьютере. Но, поскольку Delphi предназначена только для процессоров Intel, только этот тип и может составлять содержимое данного поля. Если же не ориентироваться лишь на Delphi (и ради полноты информации), то в этом поле может храниться одно из следующих значений:

  ```
  PROCESSOR_ARCHITECTURE_INTEL
  PROCESSOR_ARCHITECTURE_MIPS
  PROCESSOR_ARCHITECTURE_ALPHA
  PROCESSOR_ARCHITECTURE_PPC
  ```

- Поле `wReserved` в настоящее время не используется.
- В поле `dwPageSize` указывается размер страницы в килобайтах и определяется степень разбиения при защите и фиксации страниц. Например, на компьютерах Intel x86 это значение равно 4 Кбайт.
- В поле `lpMinimumApplicationAddress` хранится самый младший адрес памяти, доступный приложениям и функциям библиотек DLL. Попытка получить доступ к адресу памяти ниже этого значения приведет, вероятнее всего, к нарушению прав доступа. В поле `lpMaximumApplicationAddress` содержится самый старший адрес памяти, доступный приложениям и функциям библиотек DLL. Попытка получить доступ к адресу памяти выше этого значения приведет, скорее всего, к нарушению прав доступа.
- Поле `dwActiveProcessorMask` возвращает маску, представляющую набор процессоров, установленных в системе. Разряд 0 представляет первый процессор, а разряд 31 — 32-й. Было бы совсем неплохо иметь 32 процессора, правда? Но поскольку Windows 95/98 поддерживает только один процессор, то в данной реализации Win32 будет установлен лишь разряд 0.
- Поле `dwNumberOfProcessors` также возвращает количество процессоров в системе. Трудно сказать, зачем Microsoft понадобилось вносить этих два поля в запись `TSystemInfo`, но, тем не менее, это так.
- Поле `dwProcessorType` больше неактуально. Оно оставлено для обратной совместимости. Это поле может иметь одно из следующих значений:

  ```
  PROCESSOR_INTEL_386
  PROCESSOR_INTEL_486
  PROCESSOR_INTEL_PENTIUM
  PROCESSOR_MIPS_R4000
  PROCESSOR_ALPHA_21064
  ```

- Конечно же, под управлением Windows 95 возможно только значение `PROCESSOR_INTEL_x`, в то время как под управлением Windows NT допустимы все значения.
- Поле `dwAllocationGranularity` возвращает степень разбиения, которая будет учитываться при распределении памяти. В предыдущих реализациях Win32 это значение было жестко закодировано в виде 64 Кбайт. Но вполне возможно, что другие типы архитектуры аппаратных средств могут потребовать других значений.

- Поле `wProcessorLevel` определяет уровень процессора, зависящий от архитектуры системы. Это поле может содержать различные значения для разных процессоров. Для процессоров ряда Intel этот параметр может принимать любое из значений, перечисленных в табл. 14.2.

Таблица 14.2. Допустимые значения поля wProcessorLevel

Значение	Описание
3	Процессор 80386
4	Процессор 80486
5	Процессор Pentium
6	Процессор Pentium Pro и выше

- Поле `wProcessorRevision` определяет модификацию процессора, зависящую от архитектуры системы. Подобно полю `wProcessorLevel`, оно может содержать различные значения для разных процессоров. Для процессоров с архитектурой Intel в этом поле будет записано число в формате `xxyy`. Для процессоров 386 и 486 ряда Intel `xx + $0A` означает уровень модификации, а `yy` — саму модификацию (например, `0300` означает микросхему D0). Для микросхем Intel Pentium или Cyrix/NextGen 486 `xx` означает номер модели, а `yy` — модификацию (например, `0201` означает модель 2, модификацию 1).

Следующая процедура используется для получения и добавления форматированных строк системной информации в элемент управления `InfoLB` (обратите внимание на то, что этот код нацелен на отображение информации только об архитектуре Intel):

```
procedure TInfoForm.GetSysInfo;
var
  SI: TSystemInfo;
begin
  GetSystemInfo(SI);
  with InfoLB.Items, SI do
  begin
    Add(Format(SProc, [SPIntel]));
    Add(Format(SPageSize, [dwPageSize]));
    Add(Format(SMinAddr, [lpMinimumApplicationAddress]));
    Add(Format(SMaxAddr, [lpMaximumApplicationAddress]));
    Add(Format(SNumProcs, [dwNumberOfProcessors]));
    Add(Format(SAllocGra, [dwAllocationGranularity]));
    case wProcessorLevel of
      3: Add(Format(SProcLevl, [SIntel3]));
      4: Add(Format(SProcLevl, [SIntel4]));
      5: Add(Format(SProcLevl, [SIntel5]));
      6: Add(Format(SProcLevl, [SIntel6]));
    else Add(Format(SProcLevl, [IntToStr(wProcessorLevel)]));
    end;
  end;
end;
```

На заметку Функция `GetSystemInfo()` эффективно заменяет функцию `GetWinFlags()` из Windows 3.x API.

На рис. 14.3 показана форма InfoForm, отображающая системную информацию, включая информацию о версии и каталогах.

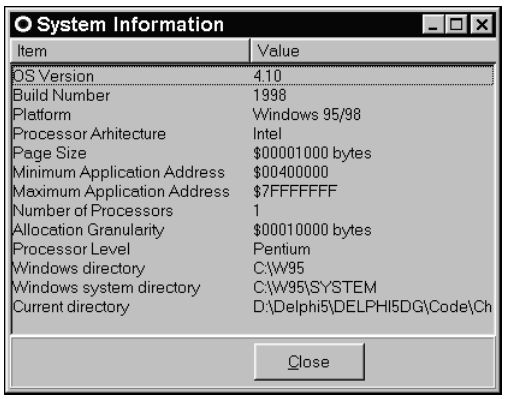


Рис. 14.3. Отображение системной информации

Получение информации о среде

Благодаря функции Win32 API GetEnvironmentStrings() получение списка переменных окружения (различных set-установок, путей и формы приглашения) стало для текущего процесса довольно простой задачей. Эта функция не имеет параметров и возвращает список строк окружения, разделенных нулями. Формат этого списка таков: строка, за которой следует нуль, за ним снова строка, снова нуль и так до тех пор, пока последняя строка не завершится двойным нулем (#0#0). В приложении SysInfo используется следующая функция, которая обрабатывает результат работы функции GetEnvironmentStrings() и размещает выделенные компоненты в элементе управления InfoLB:

```
procedure TInfoForm.ShowEnvironment;
var
  EnvPtr, SavePtr: PChar;
begin
  InfoLB.DelimChar := '=';
  EnvPtr := GetEnvironmentStrings;
  SavePtr := EnvPtr;
  InfoLB.Items.Clear;
  repeat
    InfoLB.Items.Add(StrPas(EnvPtr));
    inc(EnvPtr, StrLen(EnvPtr) + 1);
  until EnvPtr^ = #0;
  FreeEnvironmentStrings(SavePtr);
  InfoLB.Sections[0].Text := 'Environment Variable'; // Переменная окружения
  InfoLB.Sections[1].Text := 'Value'; // Значение
  Caption:= 'Current Environment'; // Текущее окружение
end;
```

На заметку

Метод ShowEnvironment() использует способность Object Pascal выполнять арифметические операции с указателями на строки типа PChar. Обратите внимание, что для обработки целого списка строк окружения требуется всего несколько строк программного кода.

Несколько комментариев к последнему методу. Во-первых, обратите внимание на то, что свойство `DelimChar` элемента управления `InfoLB` вначале устанавливается равным `'='`. А поскольку каждая пара, состоящая из переменной окружения и ее значения, уже разделена этим символом, то очень легко отобразить ее в элементе управления `InfoLB`. Кроме того, завершив обработку строк окружения, вы должны вызвать функцию `FreeEnvironmentStrings()`, чтобы освободить выделенный блок памяти.

С помощью функции `GetEnvironmentStrings()` нельзя получить или установить отдельные переменные окружения. Для этого нужно использовать функции `GetEnvironmentVariable()` и `SetEnvironmentVariable()` (см. справку Win32 API).

На рис. 14.4 показаны строки, определяющие состояние среды в том виде, как они отображаются на форме `InfoForm`.

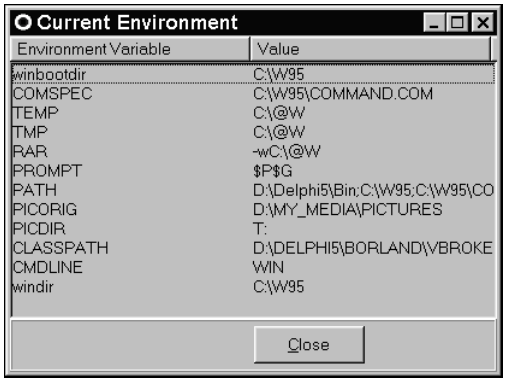


Рис. 14.4. Отображение строк значений переменных системного окружения

В листинге 14.1 исходный текст модуля `InfoU.pas` представлен полностью.

Листинг 14.1. Исходный текст модуля `InfoU.pas`

```
unit InfoU;

interface

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

type
  TInfoVariety = (ivMemory, ivSystem, ivEnvironment);

  TInfoForm = class(TForm)
    InfoLB: THeaderListbox;
    Panel1: TPanel;
    OkBtn: TButton;
  private
```

```
    procedure GetOSVerInfo;
    procedure GetSysInfo;
    procedure GetDirInfo;
  public
    procedure ShowMemStatus;
    procedure ShowSysInfo;
    procedure ShowEnvironment;
  end;

procedure ShowInformation(Variety: TInfoVariety);

implementation

{$R *.DFM}

procedure ShowInformation(Variety: TInfoVariety);
begin
  with TInfoForm.Create(Application) do
    try
      Font := MainForm.Font;
      case Variety of
        ivMemory: ShowMemStatus;
        ivSystem: ShowSysInfo;
        ivEnvironment: ShowEnvironment;
      end;
      ShowModal;
    finally
      Free;
    end;
end;

const
  { Строки состояния памяти }
  SMemUse   = 'Memory in useq%d%%';
  STotMem   = 'Total physical memoryq$%.8x bytes';
  SFreeMem  = 'Free physical memoryq$%.8x bytes';
  STotPage  = 'Total page file memoryq$%.8x bytes';
  SFreePage = 'Free page file memoryq$%.8x bytes';
  STotVirt  = 'Total virtual memoryq$%.8x bytes';
  SFreeVirt = 'Free virtual memoryq$%.8x bytes';

  { Строки информации о версии ОС }
  SOSVer    = 'OS Versionq%d.%d';
  SBuildNo  = 'Build Numberq%d';
  SOSPlat   = 'Platformq%s';
  SOSWin32s = 'Windows 3.1x running Win32s';
  SOSWin95  = 'Windows 95';
  SOSWinNT  = 'Windows NT';

  { Строки информации о системе }
```

```
  SProc     = 'Processor Arhitectureq%s';
  SPIntel   = 'Intel';
  SPageSize = 'Page Sizeq$%.8x bytes';
  SMinAddr  = 'Minimum Application Addressq$%p';
  SMaxAddr  = 'Maximum Application Addressq$%p';
  SNumProcs = 'Number of Processorsq%d';
  SAllocGra = 'Allocation Granularityq$%.8x bytes';
  SProcLevl = 'Processor Levelq%s';
  SIntel3   = '80386';
  SIntel4   = '80486';
  SIntel5   = 'Pentium';
  SIntel6   = 'Pentium Pro';
  SProcRev  = 'Processor Revisionq%.4x';

  { Строки информации о каталогах }
  SWinDir   = 'Windows directoryq%s';
  SSysDir   = 'Windows system directoryq%s';
  SCurDir   = 'Current directoryq%s';

procedure TInfoForm.ShowMemStatus;
var
  MS: TMemoryStatus;
begin
  InfoLB.DelimChar := 'q';
  MS.dwLength := SizeOf(MS);
  GlobalMemoryStatus(MS);
  with InfoLB.Items, MS do
  begin
    Clear;
    Add(Format(SMemUse, [dwMemoryLoad]));
    Add(Format(STotMem, [dwTotalPhys]));
    Add(Format(SFreeMem, [dwAvailPhys]));
    Add(Format(STotPage, [dwTotalPageFile]));
    Add(Format(SFreePage, [dwAvailPageFile]));
    Add(Format(STotVirt, [dwTotalVirtual]));
    Add(Format(SFreeVirt, [dwAvailVirtual]));
  end;
  InfoLB.Sections[0].Text := 'Resource';
  InfoLB.Sections[1].Text := 'Amount';
  Caption:= 'Memory Status';
end;

procedure TInfoForm.GetOSVerInfo;
var
  VI: TOSVersionInfo;
begin
  VI.dwOSVersionInfoSize := SizeOf(VI);
  GetVersionEx(VI);
  with InfoLB.Items, VI do
  begin
```

```
    Clear;
    Add(Format(SOSVer, [dwMajorVersion, dwMinorVersion]));
    Add(Format(SBuildNo, [LoWord(dwBuildNumber)]));
    case dwPlatformID of
      VER_PLATFORM_WIN32S: Add(Format(SOSPlat, [SOSWin32s]));
      VER_PLATFORM_WIN32_WINDOWS: Add(Format(SOSPlat, [SOSWin95]));
      VER_PLATFORM_WIN32_NT: Add(Format(SOSPlat, [SOSWinNT]));
    end;
  end;
end;

procedure TInfoForm.GetSysInfo;
var
  SI: TSystemInfo;
begin
  GetSystemInfo(SI);
  with InfoLB.Items, SI do
  begin
    Add(Format(SProc, [SPIntel]));
    Add(Format(SPageSize, [dwPageSize]));
    Add(Format(SMinAddr, [lpMinimumApplicationAddress]));
    Add(Format(SMaxAddr, [lpMaximumApplicationAddress]));
    Add(Format(SNumProcs, [dwNumberOfProcessors]));
    Add(Format(SAllocGra, [dwAllocationGranularity]));
    case wProcessorLevel of
      3: Add(Format(SProcLevl, [SIntel3]));
      4: Add(Format(SProcLevl, [SIntel4]));
      5: Add(Format(SProcLevl, [SIntel5]));
      6: Add(Format(SProcLevl, [SIntel6]));
    else Add(Format(SProcLevl, [IntToStr(wProcessorLevel)]));
    end;
  end;
end;

procedure TInfoForm.GetDirInfo;
var
  S: array[0..MAX_PATH] of char;
begin
  { Получение каталога Windows }
  GetWindowsDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SWinDir, [S]));
  { Получение системного каталога Windows }
  GetSystemDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SSysDir, [S]));
  { Получение текущего каталога процесса }
  GetCurrentDirectory(SizeOf(S), S);
  InfoLB.Items.Add(Format(SCurDir, [S]));
end;

procedure TInfoForm.ShowSysInfo;
```

```
begin
  InfoLB.DelimChar := 'q';
  GetOSVerInfo;
  GetSysInfo;
  GetDirInfo;
  InfoLB.Sections[0].Text := 'Item';
  InfoLB.Sections[1].Text := 'Value';
  Caption:= 'System Information';
end;

procedure TInfoForm.ShowEnvironment;
var
  EnvPtr, SavePtr: PChar;
begin
  InfoLB.DelimChar := '=';
  EnvPtr := GetEnvironmentStrings;
  SavePtr := EnvPtr;
  InfoLB.Items.Clear;
  repeat
    InfoLB.Items.Add(StrPas(EnvPtr));
    inc(EnvPtr, StrLen(EnvPtr) + 1);
  until EnvPtr^ = #0;
  FreeEnvironmentStrings(SavePtr);
  InfoLB.Sections[0].Text := 'Environment Variable';
  InfoLB.Sections[1].Text := 'Value';
  Caption:= 'Current Environment';
end;

end.
```

Обеспечение независимости от платформы

Приложение SysInfo разработано для функционирования под управлением как Windows 95/98, так и Windows NT, невзирая на то что разные версии Win32 имеют различные способы доступа к низкоуровневой информации (о процессорах и памяти). Подход, который взят нами на вооружение для обеспечения независимости от платформы, состоит в определении интерфейса, содержащего методы получения системной информации. Затем этот интерфейс реализуется для двух различных операционных систем. Он называется IWin32Info и имеет довольно простую организацию, которая выглядит следующим образом:

```
type
  IWin32Info = interface
    procedure FillProcessInfoList(ListView: TListView; ImageList: TImageList);
    procedure ShowProcessProperties(Cookie: Pointer);
  end;
```

- Процедура `FillProcessInfoList()` предназначена для заполнения элементов управления `TListView` и `TImageList` списком выполняющихся процессов и связанных с ними пиктограмм, если таковые имеются.
- Процедура `ShowProcessProperties()` вызывается для получения дополнительной информации об отдельном процессе, выбранном в списке (элементе управления `TListView`).

В проекте `SysInfo` есть модуль `W95Info`, содержащий класс `TWin95Info`, который реализует интерфейс `IWin32Info` для Windows 95 с помощью ToolHelp32 API. И точно так же в этом проекте содержится модуль `WNTInfo` с классом `TWinNTInfo`, который использует преимущества PSAPI для реализации интерфейса `IWin32Info`. В следующем фрагменте кода (`SysMain`), который был взят из главного модуля проекта, показано создание соответствующего класса в зависимости от операционной системы:

```
if Win32Platform = VER_PLATFORM_WIN32_WINDOWS then
    FWinInfo := TWin95Info.Create
else if Win32Platform = VER_PLATFORM_WIN32_NT then
    FWinInfo := TWinNTInfo.Create
else
    raise Exception.Create('This application must be run on Win32');
    { Это приложение должно запускаться под управлением Win32 }
```

Windows 95/98: использование ToolHelp32

ToolHelp32 — это семейство функций и процедур, составляющих подмножество Win32 API, которые позволяют получить сведения о некоторых низкоуровневых аспектах работы ОС. В частности, сюда входят функции, с помощью которых можно получить информацию обо всех процессах, выполняющихся в системе в данный момент, а также потоках, модулях и кучах, принадлежащих каждому процессу. Нетрудно догадаться, что большинство данных, получаемых от функций ToolHelp32, используется главным образом приложениями, которые должны заглядывать "внутрь" ОС (например, отладчики), хотя необходимо признать, что с помощью этих функций даже средний разработчик получит более точное представление о работе Win32.

На заметку

Семейство процедур и функций ToolHelp32 API доступно только в варианте реализации Win32 для Windows 95/98. В среде Windows NT вызов их приведет к нарушению системы защиты и безопасности NT-процессов. Поэтому приложения, которые используют функции ToolHelp32, работоспособны только под управлением Windows 95/98, но не Windows NT.

Мы используем название ToolHelp32, чтобы отличить эту подсистему от 16-разрядной версии ToolHelp, которая была включена в Windows 3.1x. Большинство функций предыдущей версии не применимы к Win32 и, следовательно, больше не поддерживаются. Кроме того, под управлением Windows 3.1x функции ToolHelp были физически расположены в библиотеке DLL, именуемой `TOOLHELP.DLL`, в то время как функции ToolHelp32 расположены в ядре Win32.

Типы и определения функций ToolHelp32 размещаются в модуле `TlHelp32`, поэтому при работе с этими функциями не забудьте включить его имя в список инструкции `uses`. Для закрепления полученных знаний в приложение, создание которого описано в этой главе, включены все функции, определенные в модуле `TlHelp32`.

На рис. 14.5 показана главная форма проекта `SysInfo`. Пользовательский интерфейс состоит главным образом из пользовательского элемента управления `TheaderListbox` (см. главу 11, "Создание многопоточных приложений"). Дважды щелкнув на любом процессе в списке, вы получите о нем более подробную информацию, которая отображается в дочерней форме, похожей на главную.

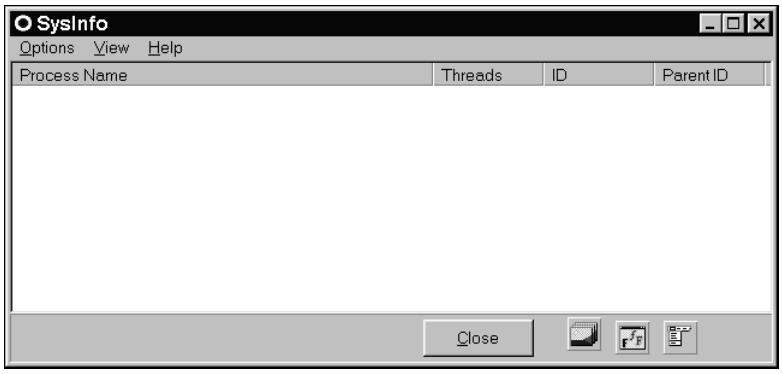


Рис. 14.5. Главная форма `TMainForm` проекта `SysInfo`

Моментальные снимки

Благодаря многозадачной природе среды Win32 такие объекты, как процессы, потоки, модули и т.п., постоянно создаются, разрушаются и модифицируются. И поскольку состояние компьютера непрерывно изменяется, системная информация, которая, возможно, будет иметь значение в данный момент, через секунду уже никого не заинтересует. Например, предположим, что вы хотите написать программу для регистрации всех модулей, загруженных в систему. Поскольку операционная система в любое время может прервать выполнение потока, отрабатывающего вашу программу, чтобы предоставить какие-то кванты времени другому потоку в системе, модули теоретически могут создаваться и разрушаться даже в момент выборки информации о них.

В этой динамической среде имело бы смысл на мгновение заморозить систему, чтобы получить такую системную информацию. В ToolHelp32 не предусмотрено средств замораживания системы, но есть функция, с помощью которой можно сделать "снимок" системы в заданный момент времени. Эта функция называется `CreateToolhelp32Snapshot()`, и ее объявление выглядит следующим образом:

```
function CreateToolhelp32Snapshot(dwFlags, th32ProcessID: DWORD): THandle;
stdcall;
```

- Параметр `dwFlags` означает тип информации, подлежащий включению в моментальный снимок. Этот параметр может иметь одно из перечисленных в табл. 14.3 значений.

Таблица 14.3. Допустимые значения параметра dwFlags

Значение	Описание
TH32CS_INHERIT	Означает, что дескриптор снимка будет наследуемым
TH32CS_SNAPALL	Эквивалентно заданию значений TH32CS_SNAPHEAPLIST, TH32CS_SNAPMODULE, TH32CS_SNAPPROCESS и TH32CS_SNAPTHREAD
TH32CS_SNAPHEAPLIST	Включает в снимок список куч заданного процесса Win32
TH32CS_SNAPMODULE	Включает в снимок список модулей заданного процесса Win32
TH32CS_SNAPPROCESS	Включает в снимок список процессов Win32
TH32CS_SNAPTHREAD	Включает в снимок список потоков Win32

- Функция CreateToolhelp32Snapshot() возвращает дескриптор созданного снимка или -1 в случае ошибки. Возвращаемый дескриптор работает подобно другим дескрипторам Win32 относительно процессов и потоков, для которых он действителен.

Следующий код создает дескриптор снимка, который содержит информацию обо всех процессах, загруженных в настоящий момент (EToolHelpError — это исключительная ситуация, определенная программистом):

```
var
  Snap: THandle;
begin
  Snap := CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if Snap = -1 then
    raise EToolHelpError.Create('CreateToolHelp32Snapshot failed');
end;
```

На заметку По завершении работы с созданным функцией CreateToolhelp32Snapshot() дескриптором, для освобождения связанных с ним ресурсов используйте функцию CloseHandle().

Обработка информации о процессах

Имея дескриптор снимка, содержащий информацию о процессах, можно воспользоваться двумя функциями ToolHelp32, которые позволяют последовательно просмотреть сведения обо всех процессах в системе. Функции Process32First() и Process32Next() определены следующим образом:

```
function Process32First(hSnapshot: THandle;
  var lppe: TProcessEntry32): BOOL; stdcall;
function Process32Next(hSnapshot: THandle;
  var lppe: TProcessEntry32): BOOL; stdcall;
```

Первый параметр у обеих функций является дескриптором снимка, возвращаемым функцией CreateToolhelp32Snapshot().

Второй параметр, lppe, представляет собой запись TProcessEntry32, которая передается по ссылке. По мере прохождения по элементам перечисления функции будут заполнять эту запись информацией о следующем процессе. Запись TProcessEntry32 определяется так:

```
type
  TProcessEntry32 = record
    dwSize: DWORD;
    cntUsage: DWORD;
    th32ProcessID: DWORD;
    th32DefaultHeapID: DWORD;
    th32ModuleID: DWORD;
    cntThreads: DWORD;
    th32ParentProcessID: DWORD;
    pcPriClassBase: Longint;
    dwFlags: DWORD;
    szExeFile: array[0..MAX_PATH - 1] of Char;
  end;
```

- В поле dwSize содержится размер записи TProcessEntry32. До использования этой записи поле dwSize должно быть инициализировано значением SizeOf(TProcessEntry32).
- В поле cntUsage хранится значение счетчика ссылок процесса. Когда это значение станет равным нулю, операционная система выгрузит процесс.
- В поле th32ProcessID содержится идентификационный номер процесса.
- Поле th32DefaultHeapID предназначено для хранения идентификатора (ID) для кучи процесса, действующей по умолчанию. Этот ID имеет значение только для функций ToolHelp32, и его нельзя использовать с другими функциями Win32.
- Поле thModuleID идентифицирует модуль, связанный с процессом. Это поле имеет значение только для функций ToolHelp32.
- По значению поля cntThreads можно судить о том, сколько потоков начало выполняться в данном процессе.
- Поле th32ParentProcessID идентифицирует родительский процесс для данного процесса.
- В поле pcPriClassBase хранится базовый приоритет процесса. Операционная система использует это значение для управления работой потоков.
- Поле dwFlags зарезервировано.
- В поле szExeFile содержится строка с ограничивающим нуль-символом, которая представляет собой путь и имя файла EXE-программы или драйвера, связанного с данным процессом.

После создания снимка, содержащего информацию о процессах, для опроса данных по каждому процессу следует вызвать сначала функцию Process32First(), а затем вызывать функцию Process32Next() до тех пор, пока она не вернет значение False.

Код опроса процессов инкапсулирован в классе TWin95Info, который реализует интерфейс IWin32Info. В следующем фрагменте представлен код метода Refresh() класса TWin95Info, который предназначен для опроса системных процессов и для добавления каждого из них в список:

```
procedure TWin95Info.Refresh;
var
  PE: TProcessEntry32;
  PPE: PProcessEntry32;
begin
  FProcList.Clear;
  if FSnap > 0 then CloseHandle(FSnap);
  FSnap := CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if FSnap = -1 then
    raise Exception.Create('CreateToolHelp32Snapshot failed');
  PE.dwSize := SizeOf(PE);
  if Process32First(FSnap, PE) then      // Получаем процесс
    repeat
      New(PPE);                          // Создаем новый PPE
      PPE^ := PE;                       // Заполняем его
      FProcList.Add(PPE);               // Добавляем его в список
    until not Process32Next(FSnap, PE); // Получаем следующий процесс
end;
```

Метод `Refresh()` вызывается методом `FillProcessInfoList()`. Как пояснялось выше, этот метод заполняет элементы управления `TListView` и `TImageList` информацией обо всех выполняемых процессах. В этом вы можете убедиться сами, рассмотрев внимательно следующий фрагмент текста:

```
procedure TWin95Info.FillProcessInfoList(ListView: TListView;
  ImageList: TImageList);
var
  I: Integer;
  ExeFile: string;
  PE: TProcessEntry32;
  HAppIcon: HIcon;
begin
  Refresh;
  ListView.Columns.Clear;
  ListView.Items.Clear;
  for I := Low(ProcessInfoCaptions) to High(ProcessInfoCaptions) do
    with ListView.Columns.Add do
    begin
      if I = 0 then Width := 285
      else Width := 75;
      Caption := ProcessInfoCaptions[I];
    end;
  for I := 0 to FProcList.Count - 1 do
  begin
    PE := PProcessEntry32(FProcList.Items[I])^;
    HAppIcon := ExtractIcon(HInstance, PE.szExeFile, 0);
    try
      if HAppIcon = 0 then HAppIcon := FWinIcon;
      ExeFile := PE.szExeFile;
      if ListView.ViewStyle = vsList then
        ExeFile := ExtractFileName(ExeFile);
```

```
        { Вставляем новый элемент, устанавливаем значение его заголовка,
          добавляем подэлементы. }
        with ListView.Items.Add, SubItems do
        begin
          Caption := ExeFile;
          Data := FProcList.Items[I];
          Add(IntToStr(PE.cntThreads));
          Add(IntToHex(PE.th32ProcessID, 8));
          Add(IntToHex(PE.th32ParentProcessID, 8));
          if ImageList <> nil then
            ImageIndex := ImageList_AddIcon(ImageList.Handle, HAppIcon);
        end;
      finally
        if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
      end;
    end;
  end;
```

На рис. 14.6 показаны результаты выполнения этого кода, отображающие информацию о процессах, запущенных под управлением Windows 95.

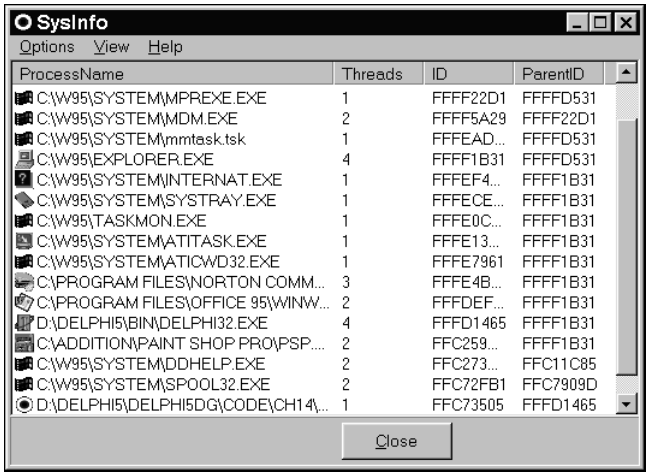


Рис. 14.6. Просмотр процессов под управлением Windows 95

Не оставьте без внимания код, который обеспечивает каждый процесс соответствующим значком (пиктограммой). Отображение значка вместе с именем приложения придает программе более профессиональный вид и создает ощущение "родного" продукта Windows. Функция API `ExtractIcon()` из модуля `ShellAPI` предпринимает попытку выделить значок из файла приложения. Если работа функции `ExtractIcon()` завершится неудачей, для отображения на форме используется стандартная пиктограмма Windows `HWinIcon`, которая заранее загружается в обработчике события `OnCreate` для этой формы с помощью функции API `LoadImage()`:

```
FWinIcon := LoadImage(0, IDI_WINLOGO, IMAGE_ICON, LR_DEFAULTSIZE,
  LR_DEFAULTSIZE, LR_DEFAULTSIZE or LR_DEFAULTCOLOR or LR_SHARED);
```

По двойному щелчку пользователя на одном из процессов в главной форме (см. рис. 14.6) вызывается метод ShowProcessProperties() интерфейса IWin32Info, а код реализации этого метода передает соответствующий параметр методу ShowProcessDetails() из модуля Detail95:

```
procedure TWin95Info.ShowProcessProperties(Cookie: Pointer);
begin
  ShowProcessDetails(PProcessEntry32(Cookie));
end;
```

Чтобы получить снимок информации для выбранного процесса, метод ShowProcessDetails() должен сделать другой снимок с помощью функции CreateToolHelp32Snapshot(). Это реализуется путем передачи параметра Cookie, который идентифицирует процесс (с помощью ID в данном случае), в качестве поля th32ProcessID для функции CreateToolHelp32Snapshot(). Чтобы поместить всю информацию в снимок, в качестве параметра dwFlags передается признак TH32CS_SNAPALL, как показано в следующем фрагменте:

```
{ Создаем снимок для текущего процесса }
FCurSnap := CreateToolhelp32Snapshot(TH32CS_SNAPALL, P^.th32ProcessID);
if FCurSnap = -1 then raise EToolHelpError.Create('CreateToolHelp32Snapshot
failed');
```

Объект TDetailForm отображает одновременно только один список. При этом тип списка определяется с помощью переменной перечислимого типа:

```
type
  TListType = (ltThread, ltModule, ltHeap);
```

Кроме того, объект TDetailForm управляет тремя отдельными списками TStringList для потоков, модулей и куч. Эти списки определяются как часть массива DetailLists:

```
DetailLists: array[TListType] of TStringList;
```

Обработка информации о потоках

Для составления списка потоков некоторого процесса в ToolHelp32 предусмотрены две функции, которые аналогичны функциям, предназначенным для регистрации процессов: Thread32First() и Thread32Next(). Эти функции объявляются следующим образом:

```
function Thread32First(hSnapshot: THandle;
  var lpte: TThreadEntry32): BOOL; stdcall;

function Thread32Next(hSnapshot: THandle;
  var lpte: TThreadENtry32): BOOL; stdcall;
```

Помимо обычного параметра hSnapshot, этим функциям также передается по ссылке параметр типа TThreadEntry32. Как и в случае функций, работающих с процессами, каждая из них заполняет запись TThreadEntry32, объявление которой имеет вид

```
type
  TThreadEntry32 = record
    dwSize: DWORD;
```

```
  cntUsage: DWORD;
  th32ThreadID: DWORD;
  th32OwnerProcessID: DWORD;
  tpBasePri: Longint;
  tpDeltaPri: Longint;
  dwFlags: DWORD;
end;
```

- Поле `dwSize` определяет размер записи, и поэтому оно должно быть инициализировано значением `SizeOf(TThreadEntry32)` до использования этой записи.
- В поле `cntUsage` содержится счетчик ссылок данного потока. При обнулении этого счетчика поток выгружается операционной системой.
- Поле `th32ThreadID` представляет собой идентификационный номер потока, который имеет значение только для функций ToolHelp32.
- В поле `th32OwnerProcessID` содержится идентификатор (ID) процесса, которому принадлежит данный поток. Этот ID можно использовать с другими функциями Win32.
- Поле `tpBasePri` представляет собой базовый класс приоритета потока. Это значение одинаково для всех потоков данного процесса. Возможные значения этого поля обычно лежат в диапазоне от 4 до 24. Описания этих значений приведены в табл. 14.4.

Таблица 14.4. Допустимые значения класса приоритета потоков

Значение	Описание
4	Ожидающий
8	Нормальный
13	Высокий
24	Реальное время

- Поле `tpDeltaPri` представляет собой дельта-приоритет (разницу), определяющий величину отличия реального приоритета от значения `tpBasePri`. Это число со знаком, которое в сочетании с базовым классом приоритета отображает общий приоритет потока. Константы, определенные для всех возможных значений дельта-приоритета, перечислены в табл. 14.5.

Таблица 14.5. Допустимые значения дельта-приоритета потоков

Константа	Значение
`THREAD_PRIORITY_IDLE`	`-15`
`THREAD_PRIORITY_LOWEST`	`-2`
`THREAD_PRIORITY_BELOW_NORMAL`	`-1`
`THREAD_PRIORITY_NORMAL`	`0`
`THREAD_PRIORITY_ABOVE_NORMAL`	`1`
`THREAD_PRIORITY_HIGHEST`	`2`
`THREAD_PRIORITY_TIME_CRITICAL`	`15`

- Поле `dwFlags` в данный момент зарезервировано и не должно использоваться.

Метод `WalkThreads()` объекта `TDetailForm` предназначен для составления списка потоков. Это делается путем добавления значимой информации о каждом потоке в элементы массива `DetailLists`, относящиеся к потокам. Вот как выглядит код этого метода:

```
procedure TWin95DetailForm.WalkThreads;
{ Для составления списка потоков используются функции ToolHelp32 }
var
  T: TThreadEntry32;
begin
  DetailLists[ltThread].Clear;
  T.dwSize := SizeOf(T);
  if Thread32First(FCurSnap, T) then
    repeat
      { Обязательно убеждаемся, что исследуемый поток принадлежит
        текущему процессу }
      if T.th32OwnerProcessID = FCurProc.th32ProcessID then
        DetailLists[ltThread].Add(Format(SThreadStr, [T.th32ThreadID,
        GetClassPriorityString(T.tpBasePri),
        GetThreadPriorityString(T.tpDeltaPri), T.cntUsage]));
    until not Thread32Next(FCurSnap, T);
end;
```

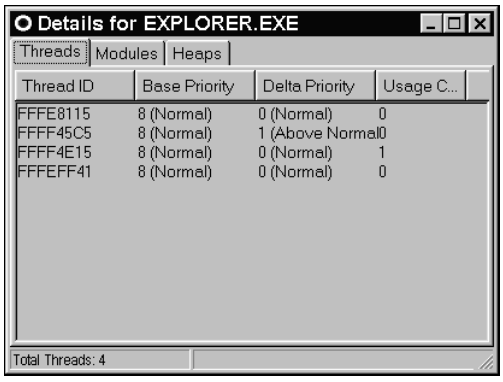


Рис. 14.7. Отображение потоков Windows 95 на вспомогательной форме

На заметку

Следующая строка кода в методе `WalkThreads()` имеет важное значение, поскольку списки потоков, "составляемые" с помощью функций ToolHelp32, не связываются с определенным потоком:

```
if T.th32OwnerProcessID = FCurProc.th32ProcessID then
```

Иными словами, при сканировании потоков нужно обязательно вручную проверять результат приведенного выше сравнения, чтобы выделить потоки, связанные с интересующим вас процессом.

На рис. 14.7 показана вспомогательная форма, на которой отображен список потоков.

Обработка информации о модулях

Опрос модулей выполняется практически так же, как опрос процессов или потоков. Для этого в ToolHelp32 предусмотрены две функции: `Module32First()` и `Module32Next()`, которые определяются следующим образом:

```
function Module32First(hSnapshot: THandle;
  var lpme: TModuleEntry32): BOOL; stdcall;

function Module32Next(hSnapshot: THandle;
  var lpme: TModuleEntry32): BOOL; stdcall;
```

Первым параметром в обеих функциях является дескриптор снимка, а вторым `var`-параметром — запись `TModuleEntry32`. Ее определение имеет следующий вид:

```
type
  TModuleEntry32 = record
    dwSize: DWORD;
    th32ModuleID: DWORD;
    th32ProcessID: DWORD;
    GlblcntUsage: DWORD;
    ProccntUsage: DWORD;
    modBaseAddr: PBYTE;
    modBaseSize: DWORD;
    hModule: HMODULE;
    szModule: array[0..MAX_MODULE_NAME32 + 1] of Char;
    szExePath: array[0..MAX_PATH - 1] of Char;
  end;
```

- Поле `dwSize` определяет размер записи и поэтому должно быть инициализировано значением `SizeOf(TModuleEntry32)` до использования этой записи.
- Поле `th32ModuleID` представляет собой идентификатор модуля, который имеет значение только для функций ToolHelp32.
- Поле `th32ProcessID` содержит идентификатор (ID) опрашиваемого процесса. Этот ID можно использовать с другими функциями Win32.
- Поле `GlblcntUsage` содержит глобальный счетчик ссылок данного модуля.
- Поле `ProccntUsage` содержит счетчик ссылок модуля в контексте процесса-владельца.
- Поле `modBaseAddr` представляет собой базовый адрес модуля в памяти. Это значение действительно только в контексте идентификатора процесса `th32ProcessID`.
- Поле `modBaseSize` определяет размер (в байтах) модуля в памяти.
- В поле `hModule` содержится дескриптор модуля. Это значение действительно только в контексте идентификатора процесса `th32ProcessID`.
- В поле `szModule` содержится строка с именем модуля, завершающаяся нуль-символом.
- Поле `szExepath` предназначено для хранения строки с ограничивающим нуль-символом, содержащей полный путь модуля.

Метод WalkModules() объекта TDetailForm очень похож на метод WalkThreads(). Как показано в следующем коде, этот метод создает список модулей и добавляет его в соответствующую часть списка массива DetailLists:

```
procedure TWin95DetailForm.WalkModules;
{ Использует функции Uses ToolHelp32 для создания списка модулей }
var
  M: TModuleEntry32;
begin
  DetailLists[ltModule].Clear;
  M.dwSize := SizeOf(M);
  if Module32First(FCurSnap, M) then
    repeat
      DetailLists[ltModule].Add(Format(SModuleStr,
        [M.szModule, M.ModBaseAddr, M.ModBaseSize, M.ProcCntUsage]));
    until not Module32Next(FCurSnap, M);
end;
```

На рис. 14.8 показана вспомогательная форма, на которой отображается список модулей.

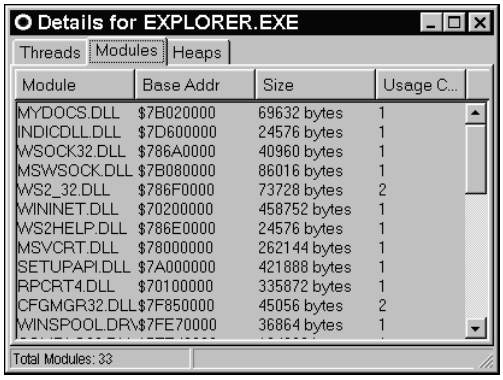


Рис. 14.8. Отображение модулей Windows 95 во вспомогательной форме

Обработка информации о динамической памяти (кучах)

Опрос куч несколько сложнее опроса других типов объектов, с которыми мы познакомились в этой главе. В ToolHelp32 предусмотрены четыре функции, с помощью которых можно получить информацию о кучах. Первые две, Heap32ListFirst() и Heap32ListNext(), позволяют выполнить проход по всем кучам процесса, а две другие, Heap32First() и Heap32Next(), используются для получения более подробной информации обо всех блоках внутри отдельной кучи.

Функции Heap32ListFirst() и Heap32ListNext() определяются следующим образом:

```
function Heap32ListFirst(hSnapshot: THandle;
  var lphl: THeapList32): BOOL; stdcall;
```

```
function Heap32ListNext(hSnapshot: THandle;
  var lphl: THeapList32): BOOL; stdcall;
```

И вновь первый параметр является дескриптором снимка, а второй (lphl) представляет собой запись типа THeapList32, передаваемую по ссылке. Определение этой записи имеет следующий вид:

```
type
  THeapList32 = record
    dwSize: DWORD;
    th32ProcessID: DWORD;
    th32HeapID: DWORD;
    dwFlags: DWORD;
  end;
```

- Поле dwSize определяет размер записи, и поэтому оно должно быть инициализировано значением SizeOf(THeapList32) до использования этой записи.
- В поле th32ProcessID содержится идентификатор (ID) процесса-владельца.
- В поле th32HeapID содержится идентификатор (ID) кучи. Этот ID имеет значение только для заданного процесса, и его можно использовать только с функциями ToolHelp32.
- В поле dwFlags хранится признак, который определяет тип кучи. В качестве значения этого поля может использоваться либо константа HF32_DEFAULT (которая означает, что текущая куча является стандартной кучей процесса), либо константа HF32_SHARED (которая означает, что текущая куча является разделяемой обычным способом).

Функции Heap32First() и Heap32Next() определяются следующим образом:

```
function Heap32First(var lphe: THeapEntry32; th32ProcessID,
  th32HeapID: DWORD): BOOL; stdcall;

function Heap32Next(var lphe: THeapEntry32): BOOL; stdcall;
```

Обратите внимание на то, что списки параметров этих функций немного отличаются от соответствующих списков функций, связанных с перечислением процессов, потоков, модулей и куч, с которыми мы познакомились выше в этой главе. Эти функции предназначены для перечисления блоков данной кучи в данном процессе, а не некоторых свойств одного процесса. При вызове функции Heap32First() параметры, установленные в полях th32ProcessID и th32HeapID, должны быть равны значениям одноименных полей записи THeapList32, заполненной с помощью функций Heap32ListFirst() или Heap32ListNext(). Var-параметр lphe функций Heap32First() и Heap32Next() имеет тип THeapEntry32. Эта запись содержит дескриптивную информацию, относящуюся к блоку кучи, и ее определение имеет следующий вид:

```
type
  THeapEntry32 = record
    dwSize: DWORD;
    hHandle: THandle;     // Дескриптор этого блока кучи
    dwAddress: DWORD;     // Линейный адрес начала блока
    dwBlockSize: DWORD;   // Размер блока в байтах
    dwFlags: DWORD;
```

```
    dwLockCount: DWORD;
    dwResvd: DWORD;
    th32ProcessID: DWORD; // Процесс-владелец
    th32HeapID: DWORD;    // Идентификатор кучи в нем
  end;
```

- Поле dwSize определяет размер записи и поэтому должно быть инициализировано значением SizeOf(THeapEntry32) до использования этой записи.
- В поле hHandle содержится дескриптор блока кучи.
- Поле dwAddress представляет собой линейный адрес начала блока кучи.
- В поле dwBlockSize содержится размер в байтах этого блока кучи.
- В поле dwFlags хранится признак, который определяет тип блока кучи. Это поле может иметь одно из значений, приведенных в табл. 14.6.

Таблица 14.6. Допустимые значения параметра dwFlags

Значение	Описание
LF32_FIXED	Блок памяти имеет фиксированное местонахождение
LF32_FREE	Блок памяти не используется
LF32_MOVEABLE	Блок памяти можно перемещать

- Поле dwLockCount представляет собой счетчик блокировок блока памяти. Это значение увеличивается на единицу при каждом вызове процессом функции GlobalLock() или LocalLock().
- Поле dwResvd зарезервировано в данный момент и не должно использоваться.
- В поле th32ProcessID содержится идентификатор процесса-владельца.
- Поле th32HeapID является идентификатором кучи, которой принадлежит блок.

Поскольку до составления списка блоков кучи вам придется сначала составить список куч, код опроса блоков кучи немного сложнее того, что было продемонстрировано до сих пор. Как видно из приведенного ниже метода TDetailForm.WalkHeaps(), весь фокус состоит во вложении цикла Heap32First()/Heap32Next() внутрь цикла Heap32ListFirst()/Heap32ListNext(). В этом методе вводится дополнительный уровень сложности путем добавления указателя на объекты типа записи PHeapEntry32 в ту часть массива DetailLists, которая относится к списку куч. Причем это выполняется таким образом, чтобы информация о куче была доступна позже, при присмотре содержимого кучи.

```
procedure TWin95DetailForm.WalkHeaps;
{ Использует функции ToolHelp32 для составления списка куч }
var
  HL: THeapList32;
  HE: THeapEntry32;
  PHE: PHeapEntry32;
begin
  DetailLists[ltHeap].Clear;
  HL.dwSize := SizeOf(HL);
```

```
  HE.dwSize := SizeOf(HE);
  if Heap32ListFirst(FCurSnap, HL) then
    repeat
      if Heap32First(HE, HL.th32ProcessID, HL.th32HeapID) then
        repeat
          { Необходимо создать копию записи THeapList32, что позволит
            сохранить информацию для просмотра сведений о куче позднее }
          New(PHE);
          PHE^ := HE;
          DetailLists[ltHeap].AddObject(Format(SHeapStr, [HL.th32HeapID,
          Pointer(HE.dwAddress), HE.dwBlockSize,
          GetHeapFlagString(HE.dwFlags)]), TObject(PHE));
        until not Heap32Next(HE);
    until not Heap32ListNext(FCurSnap, HL);
  HeapListAlloc := True;
end;
```

На рис. 14.9 показана вспомогательная форма, отображающая список блоков кучи.

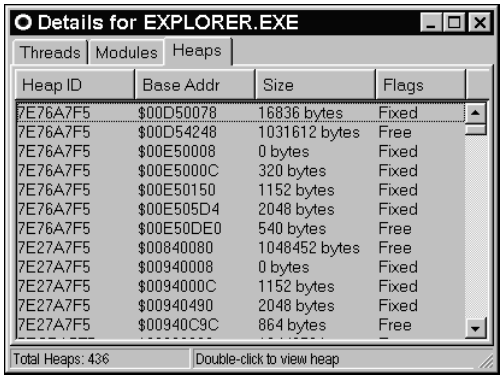


Рис. 14.9. Отображение блоков кучи Windows 95 на вспомогательной форме

Просмотр данных о куче

К этому моменту вы уже узнали обо всех функциях ToolHelp32 API, за исключением одной — `ToolHelp32ReadProcessMemory()`. Давайте познакомимся и с этой функцией.

Функция `ToolHelp32ReadProcessMemory()` объявляется следующим образом:

```
function Toolhelp32ReadProcessMemory(th32ProcessID: DWORD;
  lpBaseAddress: Pointer; var lpBuffer; cbRead: DWORD;
  var lpNumberOfBytesRead: DWORD): BOOL; stdcall;
```

Эта функция, возможно, самая мощная и, бесспорно, самая интригующая в ToolHelp32, поскольку действительно позволяет заглянуть в пространство памяти другого процесса. Рассмотрим ее параметры.

- Параметр th32ProcessID является идентификатором процесса, память которого требуется прочитать. Это значение можно получить с помощью одной из функций перечислений ToolHelp32. Для указания текущего процесса можно передать в качестве этого параметра нулевое значение.
- Параметр lpBaseAddress представляет собой линейный адрес первого байта памяти, которую требуется прочитать в процессе th32ProcessID. При этом нужно указывать правильный процесс и правильный адрес, поскольку любой линейный адрес имеет значение только для конкретного процесса.
- Параметр lpBuffer — это буфер, в который следует скопировать память процесса th32ProcessID. Не забудьте позаботиться о выделении памяти для этого буфера.
- Параметр cbRead определяет количество байтов для считывания из процесса th32ProcessID, начиная с адреса lpBaseAddress.
- Параметр lpNumberOfBytesRead заполняется во время работы функции перед возвратом из нее. Он определяет количество байтов, действительно считанных из процесса th32ProcessID.

Поскольку с помощью этой функции память отдельного процесса копируется в локальный буфер, приложение SysInfo использует еще одну модальную форму — HeapViewForm, которая форматирует дамп памяти для просмотра. Для выполнения этого форматирования форма HeapViewForm использует пользовательский компонент TddgMemView. Поскольку описание работы этого элемента управления выходит за рамки данной главы (и поскольку он не слишком сложный для понимания), вы можете просмотреть исходный код для этого компонента, обратившись к компакт-диску, прилагаемому ко второму тому (см. также www.williamspublishing.com.). А здесь приводится исходный код метода DetailLBDblClick() формы TDetailForm, который вызывается по двойному щелчку пользователя на любом элементе в DetailLB:

```
procedure TWin95DetailForm.DetailLBDblClick(Sender: TObject);
{ Эта процедура вызывается по двойному щелчку пользователя на любом
  элементе в списке DetailLB. Если текущей является вкладка
  куч, отображается форма просмотра куч. }
var
  NumRead: DWORD;
  HE: THeapEntry32;
  MemSize: integer;
begin
  inherited;
  if DetailTabs.TabIndex = 2 then
  begin
    HE := PHeapEntry32(DetailLB.Items.Objects[DetailLB.ItemIndex])^;
    MemSize := HE.dwBlockSize;          // Получаем размер кучи
    { Если куча слишком велика, используем ProcMemMaxSize }
    if MemSize > ProcMemMaxSize then MemSize := ProcMemMaxSize;
    ProcMem := AllocMem(MemSize);      // Выделяем временный буфер
    Screen.Cursor := crHourGlass;
    try
      { Копируем кучу во временный буфер }
      if Toolhelp32ReadProcessMemory(FCurProc.th32ProcessID,
        Pointer(HE.dwAddress), ProcMem^, MemSize, NumRead) then
```

```
        { Указываем на временный буфер для элемента управления HeapView }
        ShowHeapView(ProcMem, MemSize)
      else
        MessageDlg(SHeapReadErr, mtInformation, [mbOk], 0);
    finally
      Screen.Cursor := crDefault;
      FreeMem(ProcMem, MemSize);
    end;
  end;
end;
```

В этом методе сначала проверяется, является ли вкладка списка куч текущей. При положительном результате выделяется временный буфер, который передается для заполнения функции `ToolHelp32ReadProcessMemory()`. После заполнения буфер отображается в элементе управления `TddgMemView`, а форма `HeapViewForm` отображается модально. При возвращении управления от этой формы, вызванной методом `ShowModal()`, буфер освобождается. На рис. 14.10 показан пример просмотра кучи процесса.

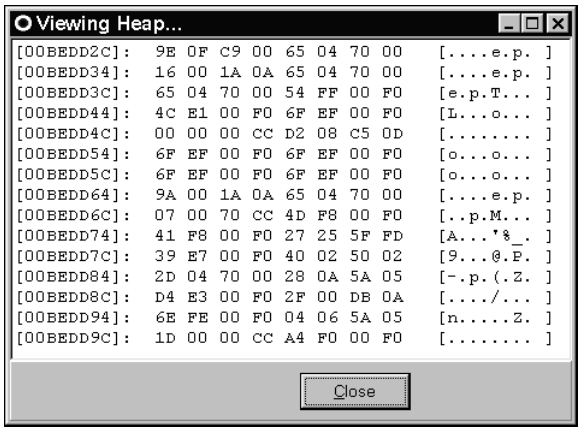


Рис. 14.10. Просмотр куч другого процесса Windows 95

Исходный код

В листингах 14.2 и 14.3 представлен исходный текст модулей `W95Info.pas` и `Detail95.pas` соответственно.

Листинг 14.2. Модуль `W95Info.pas`

```
unit W95Info;

interface

uses Windows, InfoInt, Classes, TlHelp32, Controls, ComCtrls;

type
  TWin95Info = class(TInterfacedObject, IWin32Info)
```

```
  private
    FProcList: TList;
    FWinIcon: HICON;
    FSnap: THandle;
    procedure Refresh;
  public
    constructor Create;
    destructor Destroy; override;
    procedure FillProcessInfoList(ListView: TListView; ImageList: TImageList);
    procedure ShowProcessProperties(Cookie: Pointer);
  end;

implementation

uses ShellAPI, CommCtrl, SysUtils, Detail95;

const
  ProcessInfoCaptions: array[0..3] of string =
    ('ProcessName', 'Threads', 'ID', 'ParentID');

{ TProcList }

type
  TProcList = class(TList)
    procedure Clear; override;
  end;

procedure TProcList.Clear;
var
  I: Integer;
begin
  for I := 0 to Count - 1 do Dispose(PProcessEntry32(Items[I]));
  inherited Clear;
end;

{ TWin95Info }

constructor TWin95Info.Create;
begin
  FProcList := TProcList.Create;
  FWinIcon := LoadImage(0, IDI_WINLOGO, IMAGE_ICON, LR_DEFAULTSIZE,
    LR_DEFAULTSIZE, LR_DEFAULTSIZE or LR_DEFAULTCOLOR or LR_SHARED);
end;

destructor TWin95Info.Destroy;
begin
  DestroyIcon(FWinIcon);
  if FSnap > 0 then CloseHandle(FSnap);
  FProcList.Free;
  inherited Destroy;
end;
```

```
procedure TWin95Info.FillProcessInfoList(ListView: TListView;
  ImageList: TImageList);
var
  I: Integer;
  ExeFile: string;
  PE: TProcessEntry32;
  HAppIcon: HIcon;
begin
  Refresh;
  ListView.Columns.Clear;
  ListView.Items.Clear;
  for I := Low(ProcessInfoCaptions) to High(ProcessInfoCaptions) do
    with ListView.Columns.Add do
    begin
      if I = 0 then Width := 285
      else Width := 75;
      Caption := ProcessInfoCaptions[I];
    end;
  for I := 0 to FProcList.Count - 1 do
  begin
    PE := PProcessEntry32(FProcList.Items[I])^;
    HAppIcon := ExtractIcon(HInstance, PE.szExeFile, 0);
    try
      if HAppIcon = 0 then HAppIcon := FWinIcon;
      ExeFile := PE.szExeFile;
      if ListView.ViewStyle = vsList then
        ExeFile := ExtractFileName(ExeFile);
      { Вставляем новый элемент, устанавливаем его заголовок
        и добавляем подэлементы. }
      with ListView.Items.Add, SubItems do
      begin
        Caption := ExeFile;
        Data := FProcList.Items[I];
        Add(IntToStr(PE.cntThreads));
        Add(IntToHex(PE.th32ProcessID, 8));
        Add(IntToHex(PE.th32ParentProcessID, 8));
        if ImageList <> nil then
          ImageIndex := ImageList_AddIcon(ImageList.Handle, HAppIcon);
      end;
    finally
      if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
    end;
  end;
end;

procedure TWin95Info.Refresh;
var
  PE: TProcessEntry32;
  PPE: PProcessEntry32;
begin
  FProcList.Clear;
```

```
  if FSnap > 0 then CloseHandle(FSnap);
  FSnap := CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if FSnap = -1 then
    raise Exception.Create('CreateToolHelp32Snapshot failed');
  PE.dwSize := SizeOf(PE);
  if Process32First(FSnap, PE) then     // Получаем процесс
    repeat
      New(PPE);                         // Создаем новый PPE
      PPE^ := PE;                       // Заполняем его
      FProcList.Add(PPE);               // Добавляем его в список
    until not Process32Next(FSnap, PE); // Получаем следующий процесс
end;

procedure TWin95Info.ShowProcessProperties(Cookie: Pointer);
begin
  ShowProcessDetails(PProcessEntry32(Cookie));
end;

end.
```

Листинг 14.3. Модуль `Detail95.pas`

```
unit Detail95;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls, HeadList, TlHelp32, Menus, SysMain, DetBase;

type
  TListType = (ltThread, ltModule, ltHeap);

  TWin9xDetailForm = class(TBaseDetailForm)
    procedure DetailTabsChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure DetailLBDblClick(Sender: TObject);
  private
    FCurSnap: THandle;
    FCurProc: TProcessEntry32;
    DetailLists: array[TListType] of TStringList;
    ProcMem: PByte;
    HeapListAlloc: Boolean;
    procedure FreeHeapList;
    procedure ShowList(ListType: TListType);
    procedure WalkThreads;
    procedure WalkHeaps;
    procedure WalkModules;
  public
    procedure NewProcess(P: PProcessEntry32);
```

```
  end;

procedure ShowProcessDetails(P: PProcessEntry32);

implementation

{$R *.DFM}

uses ProcMem;

const
  { Массив строк для заголовка каждого соответствующего списка. }
  HeaderStrs: array[TListType] of TDetailStrings = (
    ('Thread ID', 'Base Priority', 'Delta Priority', 'Usage Count'),
    ('Module', 'Base Addr', 'Size', 'Usage Count'),
    ('Heap ID', 'Base Addr', 'Size', 'Flags'));

  { Массив строк для нижней части страницы каждого списка. }
  ACountStrs: array[TListType] of string[31] = (
    'Total Threads: %d', 'Total Modules: %d', 'Total Heaps: %d');

  TabStrs: array[TListType] of string[7] = ('Threads', 'Modules', 'Heaps');

  SCaptionStr  = 'Details for %s';       // Заголовок формы
  SThreadStr   = '%x'#1'%s'#1'%s'#1'%d';
    { ID, базовый приоритет, дельта-приоритет, использование }
  SModuleStr   = '%s'#1'$%p'#1'%d bytes'#1'%d';
    { Имя, адрес, размер, использование }
  SHeapStr     = '%x'#1'$%p'#1'%d bytes'#1'%s';
    { ID, адрес, размер, признаки }
  SHeapReadErr = 'This heap is not accessible for read access.';
    { Эта куча не доступна для чтения }

  ProcMemMaxSize = $7FFE; { Максимальный размер диапазона просмотра кучи }

procedure ShowProcessDetails(P: PProcessEntry32);
var
  I: TListType;
begin
  with TWin9xDetailForm.Create(Application) do
    try
      for I := Low(TabStrs) to High(TabStrs) do
        DetailTabs.Tabs.Add(TabStrs[I]);
      NewProcess(P);
      Font := MainForm.Font;
      ShowModal;
    finally
      Free;
    end;
end;
```

```
function GetThreadPriorityString(Priority: DWORD): string;
{ Возвращает строки, описывающие приоритет потока }
begin
  case Priority of
    THREAD_PRIORITY_IDLE:          Result := '%d (Idle)';
    THREAD_PRIORITY_LOWEST:        Result := '%d (Lowest)';
    THREAD_PRIORITY_BELOW_NORMAL:  Result := '%d (Below Normal)';
    THREAD_PRIORITY_NORMAL:        Result := '%d (Normal)';
    THREAD_PRIORITY_ABOVE_NORMAL:  Result := '%d (Above Normal)';
    THREAD_PRIORITY_HIGHEST:       Result := '%d (Highest)';
    THREAD_PRIORITY_TIME_CRITICAL: Result := '%d (Time critical)';
  else
    Result := '%d (unknown)';
  end;
  Result := Format(Result, [Priority]);
end;

function GetClassPriorityString(Priority: DWORD): String;
{ Возвращает строки, описывающие класс приоритета процесса }
begin
  case Priority of
    4:   Result := '%d (Idle)';      // Ожидающий
    8:   Result := '%d (Normal)';    // Нормальный
    13:  Result := '%d (High)';      // Высокий
    24:  Result := '%d (Real time)'; // Реальное время
  else
    Result := '%d (non-standard)';   // Нестандартный
  end;
  Result := Format(Result, [Priority]);
end;

function GetHeapFlagString(Flag: DWORD): String;
{ Возвращает строку, описывающую признак кучи }
begin
  case Flag of
    LF32_FIXED:    Result := 'Fixed';    // Фиксированная
    LF32_FREE:     Result := 'Free';     // Свободная
    LF32_MOVEABLE: Result := 'Moveable'; // Перемещаемая
  end;
end;

procedure TWin9xDetailForm.ShowList(ListType: TListType);
{ Отображает соответствующий список потоков, куч или модулей в DetailLB }
var
  i: Integer;
begin
  Screen.Cursor := crHourGlass;
  try
    with DetailLB do
    begin
      for i := 0 to 3 do
```

```
        Sections[i].Text := HeaderStrs[ListType, i];
      Items.Clear;
      Items.Assign(DetailLists[ListType]);
    end;
     DetailSB.Panels[0].Text := Format(ACountStrs[ListType],
       [DetailLists[ListType].Count]);
     if ListType = ltHeap then
       DetailSB.Panels[1].Text := 'Double-click to view heap'
        { Для просмотра кучи щелкните дважды }
     else
       DetailSB.Panels[1].Text := '';
  finally
    Screen.Cursor := crDefault;
  end;
end;

procedure TWin9xDetailForm.WalkThreads;
{ Использует функции ToolHelp32 для опроса списка потоков }
var
  T: TThreadEntry32;
begin
  DetailLists[ltThread].Clear;
  T.dwSize := SizeOf(T);
  if Thread32First(FCurSnap, T) then
    repeat
      { Убедитесь, что поток принадлежит текущему процессу }
      if T.th32OwnerProcessID = FCurProc.th32ProcessID then
        DetailLists[ltThread].Add(Format(SThreadStr, [T.th32ThreadID,
        GetClassPriorityString(T.tpBasePri),
        GetThreadPriorityString(T.tpDeltaPri), T.cntUsage]));
    until not Thread32Next(FCurSnap, T);
end;

procedure TWin9xDetailForm.WalkModules;
{ Использует функции ToolHelp32 для опроса списка модулей }
var
  M: TModuleEntry32;
begin
  DetailLists[ltModule].Clear;
  M.dwSize := SizeOf(M);
  if Module32First(FCurSnap, M) then
    repeat
      DetailLists[ltModule].Add(Format(SModuleStr, [M.szModule,
        M.ModBaseAddr, M.ModBaseSize, M.ProcCntUsage]));
    until not Module32Next(FCurSnap, M);
end;

procedure TWin9xDetailForm.WalkHeaps;
{ Использует функции ToolHelp32 для опроса списка куч }
var
  HL: THeapList32;
```

```
  HE: THeapEntry32;
  PHE: PHeapEntry32;
begin
  DetailLists[ltHeap].Clear;
  HL.dwSize := SizeOf(HL);
  HE.dwSize := SizeOf(HE);
  if Heap32ListFirst(FCurSnap, HL) then
    repeat
      if Heap32First(HE, HL.th32ProcessID, HL.th32HeapID) then
        repeat
          { Необходимо создать копию записи THeapList32 с целью сохранения
            информации, необходимой для просмотра сведений о куче }
          New(PHE);
          PHE^ := HE;
          DetailLists[ltHeap].AddObject(Format(SHeapStr,
            [HL.th32HeapID, Pointer(HE.dwAddress), HE.dwBlockSize,
            GetHeapFlagString(HE.dwFlags)]), TObject(PHE));
        until not Heap32Next(HE);
    until not Heap32ListNext(FCurSnap, HL);
  HeapListAlloc := True;
end;

procedure TWin9xDetailForm.FreeHeapList;
{ Поскольку в список добавляются специальные выделения объектов
  PHeapList32, они должны быть освобождены. }
var
  i: integer;
begin
  for i := 0 to DetailLists[ltHeap].Count - 1 do
    Dispose(PHeapEntry32(DetailLists[ltHeap].Objects[i]));
end;

procedure TWin9xDetailForm.NewProcess(P: PProcessEntry32);
{ Эта процедура вызывается из главной формы, чтобы отобразить
  вспомогательную форму для отдельного процесса. }
begin
  { Создаем снимок для текущего процесса }
  FCurSnap := CreateToolhelp32Snapshot(TH32CS_SNAPALL, P^.th32ProcessID);
  if FCurSnap = INVALID_HANDLE_VALUE then
    raise Exception.Create('CreateToolHelp32Snapshot failed');
  HeapListAlloc := False;
  Screen.Cursor := crHourGlass;
  try
    FCurProc := P^;
    { Включаем имя модуля в заголовок вспомогательной формы }
    Caption := Format(SCaptionStr, [ExtractFileName(FCurProc.szExeFile)]);
    WalkThreads;                // Проходим по спискам ToolHelp32
    WalkModules;
    WalkHeaps;
    DetailTabs.TabIndex := 0;   // 0 = вкладка потоков
    ShowList(ltThread);         // Сначала отображаем вкладку потоков
```

```
  finally
    Screen.Cursor := crDefault;
    if HeapListAlloc then FreeHeapList;
    CloseHandle(FCurSnap);      // Закрываем дескриптор снимка
  end;
end;

procedure TWin9xDetailForm.DetailTabsChange(Sender: TObject);
{ Обработчик события OnChange для установки вкладки.
  Устанавливает список, соответствующий вкладке. }
begin
  inherited;
  ShowList(TListType(DetailTabs.TabIndex));
end;

procedure TWin9xDetailForm.FormCreate(Sender: TObject);
var
  LT: TListType;
begin
  inherited;
  { Создаем списки }
  for LT := Low(TListType) to High(TListType) do
    DetailLists[LT] := TStringList.Create;
end;

procedure TWin9xDetailForm.FormDestroy(Sender: TObject);
var
  LT: TListType;
begin
  inherited;
  { Освобождаем списки }
  for LT := Low(TListType) to High(TListType) do
    DetailLists[LT].Free;
end;

procedure TWin9xDetailForm.DetailLBDblClick(Sender: TObject);
{ Эта процедура вызывается по двойному щелчку на элементе
  в DetailLB. Если текущей является вкладка куч, отображается форма
  для просмотра куч. }
var
  NumRead: DWORD;
  HE: THeapEntry32;
  MemSize: integer;
begin
  inherited;
  if DetailTabs.TabIndex = 2 then
begin
    HE := PHeapEntry32(DetailLB.Items.Objects[DetailLB.ItemIndex])^;
    MemSize := HE.dwBlockSize;         // Получаем размер кучи
    { Если куча слишком велика, используем значение ProcMemMaxSize }
    if MemSize > ProcMemMaxSize then MemSize := ProcMemMaxSize;
```

```
    ProcMem := AllocMem(MemSize);     // Выделяем временный буфер
    Screen.Cursor := crHourGlass;
    try
      { Копируем кучу во временный буфер }
      if Toolhelp32ReadProcessMemory(FCurProc.th32ProcessID,
        Pointer(HE.dwAddress), ProcMem^, MemSize, NumRead) then
        ShowHeapView(ProcMem, MemSize)
      else
        MessageDlg(SHeapReadErr, mtInformation, [mbOk], 0);
    finally
      Screen.Cursor := crDefault;
      FreeMem(ProcMem, MemSize);
    end;
  end;
end;

end.
```

Windows NT: PSAPI

Как упоминалось выше, в среде Windows NT/2000 не существует подмножества функций ToolHelp32. Однако в SDK (Software Development Kit — комплекс инструментальных средств разработки программного обеспечения) Windows NT предусмотрена библиотека DLL PSAPI.DLL, из которой можно получить информацию о тех же объектах, что и с помощью ToolHelp32, включая:

- выполняющиеся процессы;
- модули, загружаемые для каждого процесса;
- загруженные драйверы устройств;
- информацию об использовании памяти процессов;
- файлы, отображенные в память.

Более поздние версии Windows NT и все версии Windows 2000 включают библиотеку PSAPI.DLL, хотя вы можете и сами распространять этот файл со своими приложениями. В Delphi для этой библиотеки DLL предусмотрен модуль интерфейса PSAPI.pas, который динамически загружает все ее функции. Следовательно, приложения, которые используют этот модуль, будут запускаться на компьютерах в любом случае: с или без библиотеки PSAPI.DLL (но, конечно же, если библиотека PSAPI.DLL не установлена, то ее функции работать не будут, хотя само приложение будет запускаться).

Первый этап получения информации о процессах с помощью функций PSAPI состоит в вызове функции EnumProcesses(), которая определяется следующим образом:

```
function EnumProcesses(lpidProcess: LPDWORD; cb: DWORD;
  var cbNeeded: DWORD): BOOL;
```

- Параметр lpidProcess — это указатель на массив элементов типа DWORD, который в результате работы этой функции будет заполнен значениями ID процессов.
- Параметр cb содержит количество элементов типа DWORD, передаваемых в массиве с помощью параметра lpidProcess.

- По окончании работы функции параметр cbNeeded будет содержать количество байтов, скопированных в массив, на который указывает параметр lpidProcess. Выражение cbNeeded div SizeOf(DWORD) будет определять число элементов, скопированных в массив, т.е. число запущенных процессов.

После вызова этой функции массив, передаваемый с помощью параметра lpidProcess, будет содержать несколько значений ID процессов. Нужно отметить, что эти значения сами по себе не представляют особой ценности, но с их помощью (путем передачи ID процесса функции API OpenProcess()) можно получить дескриптор процесса, вооружившись которым вы сможете вызывать другие функции PSAPI или даже другие функции Win32 API, требующие дескриптора процесса.

В PSAPI для получения информации о загруженных драйверах устройств имеется функция EnumDeviceDrivers(); ее определение выглядит следующим образом:

```
function EnumDeviceDrivers(lpImageBase: PPointer; cb: DWORD; var lpcbNeeded:
DWORD): BOOL;
```

- Параметр lpImageBase — это указатель на массив указателей, т.е. элементов типа Pointer, который в результате работы этой функции будет заполнен базовыми адресами всех драйверов устройств.
- Параметр cb содержит количество элементов типа Pointer, передаваемых в массиве с помощью параметра lpImageBase.
- По окончании работы функции параметр lpcbNeeded будет содержать количество байтов, скопированных в массив, на который указывает параметр lpImageBase.

В проекте SysInfo модуль WNTInfo.pas содержит класс TWinNTInfo, который реализует интерфейс IWin32Info. Этот класс содержит закрытый метод Refresh(), который получает информацию о процессах и драйверах устройств:

```
procedure TWinNTInfo.Refresh;
var
  Count: DWORD;
  BigArray: array[0..$3FFF - 1] of DWORD;
begin
  // Получаем массив значений ID процессов
  if not EnumProcesses(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
  SetLength(FProcList, Count div SizeOf(DWORD));
  Move(BigArray, FProcList[0], Count);
  // Получаем массив адресов драйверов устройств
  if not EnumDeviceDrivers(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
  SetLength(FDrvList, Count div SizeOf(DWORD));
  Move(BigArray, FDrvList[0], Count);
end;
```

Этот метод сначала передает локальный массив BigArray функциям EnumProcesses() и EnumDeviceDrivers(), а затем перемещает данные из массива BigArray в динамические массивы FProcList и FDrvList. Такая неуклюжая реализация этих функций простительна, поскольку ни EnumProcesses(), ни EnumDeviceDrivers() не предоставляют средств для опреде-

ления числа элементов, которые будут возвращены перед выделением памяти для массива. Поэтому мы передаем методам заведомо большой массив (полагаем, что достаточно большой) и копируем результаты в динамический массив соответствующего размера.

Метод `FillProcessInfoList()` класса `TWinNTInfo` вызывает два вспомогательных метода — `FillProcesses()` и `FillDrivers()`, — которые предназначены для заполнения содержимого компонента `TListView` на главной форме. Метод `FillProcesses()` представлен в следующем листинге:

```
procedure TWinNTInfo.FillProcesses(ListView: TListView; ImageList: TImageList);
var
  I, Count: Integer;
  ProcHand: THandle;
  ModHand: HMODULE;
  HAppIcon: HICON;
  ModName: array[0..MAX_PATH] of char;
begin
  for I := Low(FProcList) to High(FProcList) do
  begin
    ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or PROCESS_VM_READ,
      False, FProcList[I]);
    if ProcHand > 0 then
      try
        EnumProcessModules(Prochand, @ModHand, 1, Count);
        if GetModuleFileNameEx(Prochand, ModHand, ModName,
          SizeOf(ModName)) > 0 then
        begin
          HAppIcon := ExtractIcon(HInstance, ModName, 0);
          try
            if HAppIcon = 0 then HAppIcon := FWinIcon;
            with ListView.Items.Add, SubItems do
            begin
              Caption := ModName;               // Имя файла
              Data := Pointer(FProcList[I]);    // Сохраняем ID
              Add(SProcName);                   // "Процесс"
              Add(IntToStr(FProcList[I]));      // ID процесса
              Add('$' + IntToHex(ProcHand, 8)); // Дескриптор процесса
              // Класс приоритета
              Add(GetPriorityClassString(GetPriorityClass(ProcHand)));
              // Значок
              if ImageList <> nil then
                ImageIndex := ImageList_AddIcon(ImageList.Handle, HAppIcon);
            end;
          finally
            if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
          end;
        end;
      finally
        CloseHandle(ProcHand);
      end;
  end;
end;
```

В этом методе функция OpenProcess() используется для преобразования идентификатора каждого процесса в дескриптор процесса. В качестве первого параметра этому методу можно передать несколько признаков, но для запроса информации с помощью функций PSAPI лучше всего использовать тандем PROCESS_QUERY_INFORMATION и PROCESS_VM_READ. Имея на руках дескриптор процесса, метод FillProcesses() вызывает функцию EnumProcessModules() для получения имени файла, связанного с процессом. Эта функция определяется следующим образом:

```
function EnumProcessModules(hProcess: THandle; lphModule: LPDWORD;
  cb: DWORD; var lpcbNeeded: DWORD): BOOL;
```

Этот метод работает аналогично другим функциям PSAPI: hProcess представляет собой дескриптор процесса, lphModule — это указатель на массив дескрипторов модулей, cb определяет число элементов в массиве, а последний параметр возвращает количество байтов, скопированных в массив, на который указывает параметр lphModule.

Поскольку в данный момент нас интересует только главный модуль этого процесса, мы передаем массив, состоящий только из одного элемента. Первый модуль, возвращаемый функцией EnumProcessModules(), и является главным модулем процесса. А затем вся информация о процессах добавляется в элемент управления TListView аналогично тому, как показано в методах класса TWin95Info.

Подобным образом действует и метод FillDrivers(), за исключением того, что он вызывает функцию GetDeviceDriverFileName(), которая определяется так:

```
function GetDeviceDriverFileName(ImageBase: Pointer; lpFileName: PChar;
  nSize: DWORD): DWORD;
```

Этот метод в качестве первого параметра принимает базовый адрес драйвера устройства, в качестве второго — указатель на строковый буфер, а в качестве последнего — размер буфера. При успешном выполнении функции параметр lpFileName будет содержать имя файла драйвера устройства. В следующем листинге показан наш вариант использования метода:

```
procedure TWinNTInfo.FillDrivers(ListView: TListView; ImageList: TImageList);
var
  I: Integer;
  DrvName: array[0..MAX_PATH] of char;
begin
  for I := Low(FDrvList) to High(FDrvList) do
    if GetDeviceDriverFileName(FDrvList[I], DrvName, SizeOf(DrvName)) > 0
      then
      with ListView.Items.Add do
      begin
        Caption := DrvName;
        SubItems.Add(SDrvName);
        SubItems.Add('$' + IntToHex(Integer(FDrvList[I]), 8));
      end;
end;
```

На рис. 14.11 показан результат работы приложения SysInfo, запущенного на компьютере с установленной системой Windows NT 4.0.

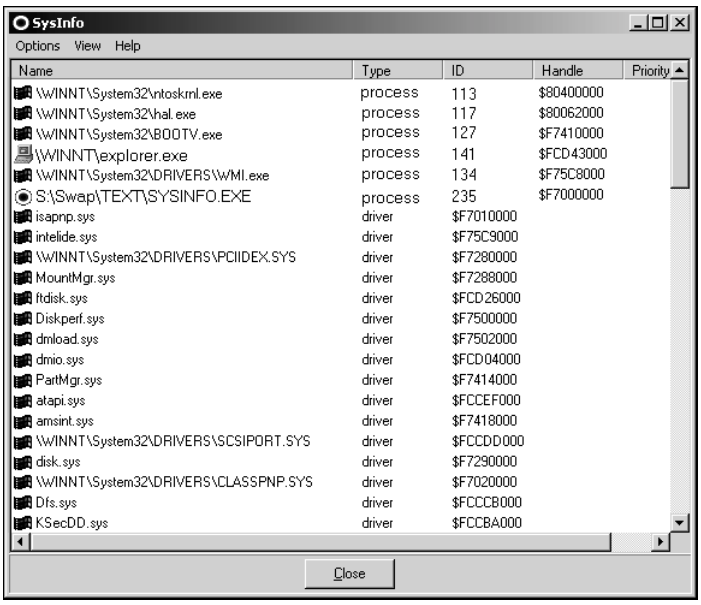


Рис. 14.11. Просмотр процессов и драйверов устройств Windows NT

Подобно тому как в классе `TWin95Info` реализован метод `ShowProcessProperties()`, в классе `TWinNTInfo` выполняется обращение к другому модулю с целью отображения формы, содержащей более подробную информацию о процессах. В частности, дополнительная информация связана с модулями и памятью, используемой процессами. Метод, предназначенный для получения этой информации, принадлежит к классу `TWinNTDetailForm`, определенному в модуле `DetailNT`, и включает следующий код:

```
procedure TWinNTDetailForm.NewProcess(ProcessID: DWORD);
const
  AddrMask = DWORD($FFFFF000);
var
  I, Count: Integer;
  ProcHand: THandle;
  WSPtr: Pointer;
  ModHandles: array[0..$3FFF - 1] of DWORD;
  WorkingSet: array[0..$3FFF - 1] of DWORD;
  ModInfo: TModuleInfo;
  ModName, MapFileName: array[0..MAX_PATH] of char;
begin
  ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or PROCESS_VM_READ,
    False, ProcessID);
  if ProcHand = 0 then
    raise Exception.Create('No information available for this process/driver');
  try
    EnumProcessModules(ProcHand, @ModHandles, SizeOf(ModHandles), Count);
    for I := 0 to (Count div SizeOf(DWORD)) - 1 do
      if (GetModuleFileNameEx(ProcHand, ModHandles[I],
        ModName, SizeOf(ModName)) > 0) and
```

```
        GetModuleInformation(ProcHand,
        ModHandles[I], @ModInfo, SizeOf(ModInfo)) then
        with ModInfo do
          DetailLists[ltModules].Add(Format(SModuleStr,
            [ModName, lpBaseOfDll, SizeOfImage, EntryPoint]));
    if QueryWorkingSet(ProcHand, @WorkingSet, SizeOf(WorkingSet)) then
      for I := 1 to WorkingSet[0] do
      begin
        WSPtr := Pointer(WorkingSet[I] and AddrMask);
        GetMappedFileName(ProcHand, WSPtr, MapFileName, SizeOf(MapFileName));
        DetailLists[ltMemory].Add(Format(SMemoryStr,
          [WSPtr, MemoryTypeToString(WorkingSet[I]),
          MapFileName]));
      end;
  finally
    CloseHandle(ProcHand);
  end;
end;
```

Как видите, в этом методе выполняются обращения к функциям `OpenProcess()` и `EnumProcessModules()`, с которыми вы уже знакомы. Этот метод также вызывает функцию PSAPI `QueryWorkingSet()`, но в данном случае это делается для получения информации о процессе. Упомянутая функция определяется следующим образом:

```
function QueryWorkingSet(hProcess: THandle; pv: Pointer; cb: DWORD): BOOL;
```

Параметр `hProcess` является дескриптором процесса, параметр `pv` — это указатель на массив элементов типа `DWORD`, а параметр `cb` содержит число элементов в массиве. По окончании работы этой функции параметр `pv` будет указывать на массив элементов типа `DWORD`, причем старших 20 разрядов каждого элемента будут содержать базовый адрес страницы памяти, а младших 12 разрядов — признаки, по которым можно определить, данная страница читается ли, записывается ли, выполняется ли и т.д.

На рис. 14.12 и 14.13 показаны результаты отображения информации о модулях и используемой памяти в среде Windows NT. В листингах 14.4 и 14.5 представлен исходный код модулей `WNTInfo.pas` и `DetailNT.pas` соответственно.

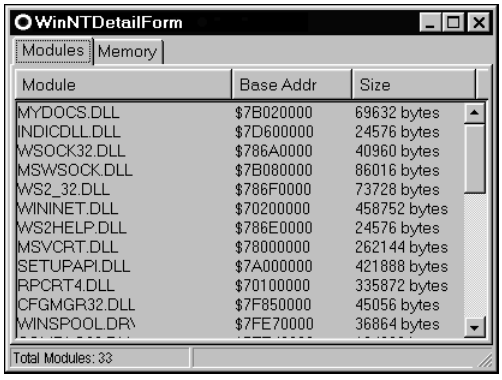


Рис. 14.12. Просмотр модулей процессов в Windows NT

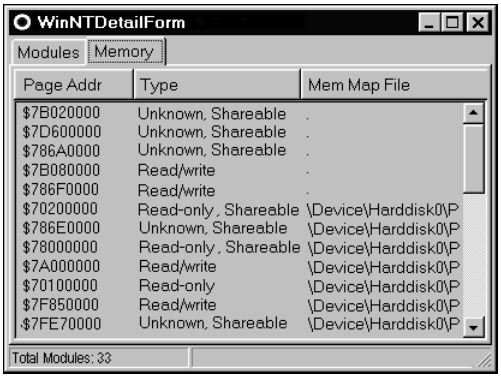


Рис. 14.13. Просмотр данных об использовании памяти процессами в Windows NT

Листинг 14.4. Модуль WNTInfo.pas

```
unit WNTInfo;

interface

uses InfoInt, Windows, Classes, ComCtrls, Controls;

type
  TWinNTInfo = class(TInterfacedObject, IWin32Info)
  private
    FProcList: array of DWORD;
    FDrvlist: array of Pointer;
    FWinIcon: HICON;
    procedure FillProcesses(ListView: TListView; ImageList: TImageList);
    procedure FillDrivers(ListView: TListView; ImageList: TImageList);
    procedure Refresh;
  public
    constructor Create;
    destructor Destroy; override;
    procedure FillProcessInfoList(ListView: TListView; ImageList: TImageList);
    procedure ShowProcessProperties(Cookie: Pointer);
  end;

implementation

uses SysUtils, PSAPI, ShellAPI, CommCtrl, DetailNT;

const
  SFailMessage = 'Failed to enumerate processes or drivers.  Make sure ' +
    'PSAPI.DLL is installed on your system.';
    { Не удалось опросить процессы или драйверы. Убедитесь в
      установке библиотеки PSAPI.DLL в вашей системе. }
  SDrvName = 'driver';  // Драйвер
  SProcname = 'process';  // Процесс
  ProcessInfoCaptions: array[0..4] of string = (
    'Name', 'Type', 'ID', 'Handle', 'Priority');
    // Имя, тип, идентификатор, дескриптор, приоритет
function GetPriorityClassString(PriorityClass: Integer): string;
begin
  case PriorityClass of
    HIGH_PRIORITY_CLASS: Result := 'High';         // Высокий
    IDLE_PRIORITY_CLASS: Result := 'Idle';         // Ждущий
    NORMAL_PRIORITY_CLASS: Result := 'Normal';     // Нормальный
    REALTIME_PRIORITY_CLASS: Result := 'Realtime'; // Реального времени
  else
    Result := Format('Unknown ($%x)', [PriorityClass]); // Неизвестный
  end;
end;

{ TWinNTInfo }
```

```
constructor TWinNTInfo.Create;
begin
  FWinIcon := LoadImage(0, IDI_WINLOGO, IMAGE_ICON, LR_DEFAULTSIZE,
  LR_DEFAULTSIZE, LR_DEFAULTSIZE or LR_DEFAULTCOLOR or LR_SHARED);
end;

destructor TWinNTInfo.Destroy;
begin
  DestroyIcon(FWinIcon);
  inherited Destroy;
end;

procedure TWinNTInfo.FillDrivers(ListView: TListView; ImageList: TImageList);
var
  I: Integer;
  DrvName: array[0..MAX_PATH] of char;
begin
  for I := Low(FDrvList) to High(FDrvList) do
    if GetDeviceDriverFileName(FDrvList[I], DrvName, SizeOf(DrvName)) > 0 then
      with ListView.Items.Add do
      begin
        Caption := DrvName;
        SubItems.Add(SDrvName);
        SubItems.Add('$' + IntToHex(Integer(FDrvList[I]), 8));
      end;
end;

procedure TWinNTInfo.FillProcesses(ListView: TListView;
  ImageList: TImageList);
var
  I, Count: Integer;
  Count: DWORD
  ProcHand: THandle;
  ModHand: HMODULE;
  HAppIcon: HICON;
  ModName: array[0..MAX_PATH] of char;
begin
  for I := Low(FProcList) to High(FProcList) do
  begin
    ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or PROCESS_VM_READ,
      False, FProcList[I]);
    if ProcHand > 0 then
      try
        EnumProcessModules(Prochand, @ModHand, 1, Count);
        if GetModuleFileNameEx(Prochand, ModHand, ModName,
          SizeOf(ModName)) > 0 then
        begin
          HAppIcon := ExtractIcon(HInstance, ModName, 0);
          try
            if HAppIcon = 0 then HAppIcon := FWinIcon;
            with ListView.Items.Add, SubItems do
```

```
          begin
            Caption := ModName;                 // Имя файла
            Data := Pointer(FProcList[I]);      // Сохраняем ID
            Add(SProcName);                     // "Процесс"
            Add(IntToStr(FProcList[I]));        // ID процесса
            Add('$' + IntToHex(ProcHand, 8));   // Дескриптор процесса
            // Класс приоритета
            Add(GetPriorityClassString(GetPriorityClass(ProcHand)));
            // Значок
            if ImageList <> nil then
              ImageIndex := ImageList_AddIcon(ImageList.Handle, HAppIcon);
          end;
        finally
          if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
        end;
      end;
    finally
      CloseHandle(ProcHand);
    end;
  end;
end;

procedure TWinNTInfo.FillProcessInfoList(ListView: TListView;
  ImageList: TImageList);
var
  I: Integer;
begin
  Refresh;
  ListView.Columns.Clear;
  ListView.Items.Clear;
  for I := Low(ProcessInfoCaptions) to High(ProcessInfoCaptions) do
    with ListView.Columns.Add do
    begin
      if I = 0 then Width := 285
      else Width := 75;
      Caption := ProcessInfoCaptions[I];
    end;
  FillProcesses(ListView, ImageList);
  { Добавляем процессы в список просмотра }
  FillDrivers(ListView, ImageList);
  { Добавляем драйверы устройств в список просмотра }
end;

procedure TWinNTInfo.Refresh;
var
  Count: DWORD;
  BigArray: array[0..$3FFF - 1] of DWORD;
begin
  // Получаем массив идентификаторов процессов
  if not EnumProcesses(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
```

```
  SetLength(FProcList, Count div SizeOf(DWORD));
  Move(BigArray, FProcList[0], Count);
  // Получаем массив адресов драйверов
  if not EnumDeviceDrivers(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
  SetLength(FDrvList, Count div SizeOf(DWORD));
  Move(BigArray, FDrvList[0], Count);
end;

procedure TWinNTInfo.ShowProcessProperties(Cookie: Pointer);
begin
  ShowProcessDetails(DWORD(Cookie));
end;

end.
```

Листинг 14.5. Модуль `DetailNT.pas`

```
unit DetailNT;

interface

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

type
  TListType = (ltModules, ltMemory);

  TWinNTDetailForm = class(TBaseDetailForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure DetailTabsChange(Sender: TObject);
  private
    FProcHand: THandle;
    DetailLists: array[TListType] of TStringList;
    procedure ShowList(ListType: TListType);
  public
    procedure NewProcess(ProcessID: DWORD);
  end;

procedure ShowProcessDetails(ProcessID: DWORD);

implementation

uses PSAPI;
{$R *.DFM}

const
  TabStrs: array[0..1] of string[7] = ('Modules', 'Memory'); // Модули, память
```

```
  { Массив строк для нижней части каждого списка. }
  ACountStrs: array[TListType] of string[31] =
    ( 'Total Modules: %d', 'Total Pages: %d');
    // Всего модулей, всего страниц

  { Массив строк для заголовка каждого соответствующего списка. }
  HeaderStrs: array[TListType] of TDetailStrings =
    (('Module', 'Base Addr', 'Size', 'Entry Point'),
    ('Page Addr', 'Type', 'Mem Map File', ''));
    { Модуль, базовый адрес, размер, точка входа, адрес страницы,
      тип, файл, отображенный в память. }

  SCaptionStr  = 'Details for %s';    // Заголовок формы
  SModuleStr   = '%s'#1'$%p'#1'%d bytes'#1'$%p';
    { Имя, адрес, size, точка входа }
  SMemoryStr   = '$%p'#1'%s'#1'%s';
    { Адрес, тип, файл, отображенный в память }

procedure ShowProcessDetails(ProcessID: DWORD);
var
  I: Integer;
begin
  with TWinNTDetailForm.Create(Application) do
    try
      for I := Low(TabStrs) to High(TabStrs) do
        DetailTabs.Tabs.Add(TabStrs[I]);
      NewProcess(ProcessID);
      ShowList(ltModules);
      ShowModal;
    finally
      Free;
    end;
end;

function MemoryTypeToString(Value: DWORD): string;
const
  TypeMask = DWORD($0000000F);
begin
  Result := '';
  case Value and TypeMask of
    1: Result := 'Read-only';
    2: Result := 'Executable';
    4: Result := 'Read/write';
    5: Result := 'Copy on write';
  else
    Result := 'Unknown';
  end;
  if Value and $100 <> 0 then
    Result := Result + ', Shareable';
end;
```

```
procedure TWinNTDetailForm.FormCreate(Sender: TObject);
var
  LT: TListType;
begin
  inherited;
  { Создание списков }
for LT := Low(TListType) to High(TListType) do
    DetailLists[LT] := TStringList.Create;
end;

procedure TWinNTDetailForm.FormDestroy(Sender: TObject);
var
  LT: TListType;
begin
  inherited;
  { Освобождение списков }
  for LT := Low(TListType) to High(TListType) do
    DetailLists[LT].Free;
end;

procedure TWinNTDetailForm.NewProcess(ProcessID: DWORD);
const
  AddrMask = DWORD($FFFFF000);
var
  I, Count: Integer;
  ProcHand: THandle;
  WSPtr: Pointer;
  ModHandles: array[0..$3FFF - 1] of DWORD;
  WorkingSet: array[0..$3FFF - 1] of DWORD;
  ModInfo: TModuleInfo;
  ModName, MapFileName: array[0..MAX_PATH] of char;
begin
  ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or
    PROCESS_VM_READ, False, ProcessID);
  if ProcHand = 0 then
    raise Exception.Create('No information available for this process/driver');
    { Для данного процесса/драйвера нет доступной информации }
  try
    EnumProcessModules(ProcHand, @ModHandles, SizeOf(ModHandles), Count);
    for I := 0 to (Count div SizeOf(DWORD)) - 1 do
      if (GetModuleFileNameEx(ProcHand, ModHandles[I], ModName,
        SizeOf(ModName)) > 0) and GetModuleInformation(ProcHand,
        ModHandles[I], @ModInfo, SizeOf(ModInfo)) then
        with ModInfo do
          DetailLists[ltModules].Add(Format(SModuleStr, [ModName,
            lpBaseOfDll, SizeOfImage, EntryPoint]));
    if QueryWorkingSet(ProcHand, @WorkingSet, SizeOf(WorkingSet)) then
      for I := 1 to WorkingSet[0] do
      begin
        WSPtr := Pointer(WorkingSet[I] and AddrMask);
        GetMappedFileName(ProcHand, WSPtr, MapFileName, SizeOf(MapFileName));
```

```
        DetailLists[ltMemory].Add(Format(SMemoryStr, [WSPtr,
          MemoryTypeToString(WorkingSet[I]), MapFileName]));
      end;
  finally
    CloseHandle(ProcHand);
  end;
end;

procedure TWinNTDetailForm.ShowList(ListType: TListType);
var
  I: Integer;
begin
  Screen.Cursor := crHourGlass;
  try
    with DetailLB do
    begin
      for I := 0 to 3 do
        Sections[I].Text := HeaderStrs[ListType, i];
      Items.Clear;
      Items.Assign(DetailLists[ListType]);
    end;
     DetailSB.Panels[0].Text := Format(ACountStrs[ListType],
       [DetailLists[ListType].Count]);
  finally
    Screen.Cursor := crDefault;
  end;
end;

procedure TWinNTDetailForm.DetailTabsChange(Sender: TObject);
begin
  inherited;
  ShowList(TListType(DetailTabs.TabIndex));
end;

end.
```

Резюме

В этой главе продемонстрированы методы доступа к системной информации из программ Delphi. Особое внимание уделялось корректному использованию функций ToolHelp32 в среде Windows 95/98 и функций PSAPI в среде Windows NT. Вы узнали, как использовать некоторые функции Win32 API для получения других типов системной информации, включая данные об использовании памяти, переменных окружения и версии операционной системы. Кроме того, вы научились включать в свои приложения такие пользовательские компоненты, как `TListView`, `TImageList`, `THeaderListbox` и `TddgMemView`. Следующая глава посвящена переходу к Delphi 5 от предыдущих версий.

Переход на Delphi 5

Глава

15

Если вы переходите к Delphi 5 с одной из предыдущих версий, эта глава — специально для вас. Первый ее раздел посвящен вопросам, связанным с переходом на Delphi 5 из предыдущих версий. Во втором, третьем и четвертом разделах речь идет о незначительных различиях между имеющимися 32-разрядными версиями Delphi и о том, как эти различия учитывать при переносе приложений в Delphi 5. Пятый раздел написан в помощь тем, кто переносит свои приложения с 16-разрядной Delphi 1 в 32-разрядную среду Delphi 5. Несмотря на то что Borland сделала все возможное для совместимости версий, вполне понятно, что для обеспечения прогресса некоторые изменения должны иметь место. Поэтому в любое переносимое из прежних версий Delphi приложение все же должны быть внесены определенные изменения с целью его успешной компиляции и правильной работы в среде Delphi 5.

Новое в Delphi 5

В целом, чем более новой является версия Delphi, с которой необходим переход, тем проще осуществляется процесс переноса приложений в среду Delphi 5. Однако, с какой бы из версий вы не переходили к Delphi 5 — с Delphi 1, 2, 3 или 4, — в данном разделе вы найдете всю необходимую для этого информацию.

Определение версии

Несмотря на то что большинство программ Delphi будет компилироваться во всех версиях компилятора, в некоторых случаях отличия языка или подпрограмм библиотеки VCL вынуждают вносить незначительные изменения в текст, предназначенный для выполнения определенных задач, с целью его соответствия конкретной версии этого программного продукта. Время от времени вам придется компилировать один и тот же код для различных версий Delphi. Для этого компилятор каждой версии содержит ее условное определение `VERxxx`, которое может протестировать ваш код. Поскольку Borland C++ Builder поставляется с новыми версиями компилятора Delphi, то эта среда разработки также содержит упомянутое условное определение. Условные определения для различных версий компилятора Delphi представлены в табл. 15.1.

Таблица 15.1. Условные определения для версий компиляторов Borland

Продукт	Условное определение	Продукт	Условное определение
Delphi 1	`VER80`	C++ Builder 3	`VER110`
Delphi 2	`VER90`	Delphi 4	`VER120`
C++ Builder 1	`VER95`	C++ Builder 4	`VER120`
Delphi 3	`VER100`	Delphi 5	`VER130`

Исходя из указанных значений, исходный код, предназначенный для обработки компилятором различных версий, должен оформляться приблизительно так:

```
{$IFDEF VER80}
  Код для компилятора Delphi 1
{$ENDIF}
{$IFDEF VER90}
  Код для компилятора Delphi 2
{$ENDIF}
```

```
{$IFDEF VER95}
  Код для компилятора C++ Builder 1
{$ENDIF}
{$IFDEF VER100}
  Код для компилятора Delphi 3
{$ENDIF}
{$IFDEF VER110}
  Код для компилятора C++ Builder 3
{$ENDIF}
{$IFDEF VER120}
 Код для компиляторов Delphi 4 и C++ Builder 4
{$ENDIF}
{$IFDEF VER130}
  Код для компилятора Delphi 5
{$ENDIF}
```

На заметку

Вас может удивить тот факт, что компилятор Delphi 1 считается версией 8, Delphi 2 — версией 9 и т.д. Это связано с тем, что в Delphi 1 вошла 8-я версия компилятора Borland Pascal. Предыдущая версия компилятора Turbo Pascal была 7-й, и в Delphi 1 эта линия продуктов была продолжена.

Помимо различий, связанных с версиями языка и библиотеки VCL, вы можете столкнуться с изменениями и в Windows API. Для того чтобы преодолеть сложности переноса программ, связанные с различиями между 16- и 32-разрядными API, следует воспользоваться дополнительными определениями, специально созданными для этой цели. Компилятор Delphi для 16-разрядной среды Windows определяет значение `Windows`, тогда как компилятор Delphi для среды Win32 — `WIN32`. Таким образом, определение разрядности компилятора может выглядеть так:

```
{$IFDEF WINDOWS}
  программный текст для среды 16-разрядной Windows
{$ENDIF}
{$IFDEF WIN32}
  программный текст для среды Win32
{$ENDIF}
```

Модули, компоненты и пакеты

Скомпилированные модули Delphi 5 (DCU-файлы) отличаются от подобных им модулей всех предыдущих версий Delphi и C++ Builder. Для того чтобы построить приложение для любой наперед заданной версии Delphi, необходимо иметь исходный код каждого модуля, используемого вашим приложением. Это означает, что в подобном случае вы не сможете воспользоваться компонентами (неважно, каким производителем они изготовлены — вашим или сторонним), если у вас не будет их исходного кода. При отсутствии исходного кода какого-либо компонента, разработанного независимым производителем, обратитесь к распространителю за версией компонента, соответствующей вашей версии Delphi.

На заметку

Вопрос соответствия версий компилятора и файлов модулей ненов. Если вы распространяете или покупаете компоненты без исходного кода, то должны понимать, что эти файлы предназначены для определенной версии компилятора и, вероятно, потребуют пересмотра при появлении последующих версий.
Более того, вопрос версий модулей DCU не обязательно связан только с компилятором. Даже если в новой версии компилятор останется прежним, изменения и усовершенствования ядра библиотеки VCL, вполне вероятно, могут потребовать перекомпиляции исходного кода этих модулей.

В Delphi 3 впервые появились пакеты — реализация идеи хранения нескольких модулей в одном бинарном файле. Начиная с Delphi 3 библиотека компонентов стала больше походить на коллекцию пакетов, чем на единый монолитный файл динамически компонуемой библиотеки (DLL). Как и модули, пакеты не совместимы с различными версиями продукта, а следовательно, потребуется выполнить их повторное построение для использования в Delphi 5. За обновленными версиями используемых пакетов нужно обратиться к их распространителям.

Переход с Delphi 4

При перемещении приложений Delphi 4 в среду Delphi 5 проблем будет совсем немного. В большинстве случаев вам понадобится просто загрузить проект в Delphi 5 и перекомпилировать его. Если при этом возникнут какие-либо осложнения, не расстраивайтесь — в этом разделе вам будет предоставлена вся информация, необходимая для исправления ситуации и успешного переноса приложения в новую среду.

Интегрированная среда разработки (IDE)

Различия в работе IDE, вероятно, будут первыми особенностями, которые вам придется учитывать при переносе приложения. Основные из них перечислены ниже.

- Файл символических имен отладчика Delphi 4 не совместим с форматом этого файла в Delphi 5. Следствием этого факта может стать появление сообщения “`Error reading symbol file`“ (“`Ошибка при чтении файла символических имен`”). Чтобы выйти из данной ситуации, следует заново построить это приложение.
- В Delphi 5 файлы форм по умолчанию сохраняются в символьном формате. Если требуется обеспечить совместимость с DFM-файлами предыдущих версий, необходимо установить режим сохранения файлов форм в двоичном формате. Это осуществляется путем сброса флажка опции New Form As Text во вкладке Preference диалогового окна Environment Options.
- Генератор кода, используемый для генерации библиотек типов при импортировании компонентов, был изменен. Помимо прочих незначительных улучшений, в новый генератор кода была внедрена поддержка средств отображения символических имен. Появилась возможность настройки библиотек типов на использование символических имен, принятых в языке Pascal, посредством редактирования файла `tlibimp.sym`. Подробное описание этой функции можно найти в статье “Mapping Symbol Names in the Type Library” интерактивной справочной системы Delphi.

Библиотека времени выполнения (RTL)

Единственная проблема, имеющая отношение к библиотеке времени выполнения, связана с установкой управляющего слова сопроцессора обработки вещественных чисел (FPU) в библиотеках DLL. В предыдущих версиях Delphi подпрограммы библиотек DLL имели право устанавливать управляющее слово FPU, а следовательно, изменять значение этого слова, установленного при запуске основного приложения. В новой версии в коде инициализации библиотек DLL установить значение управляющего слова FPU больше нельзя. Если установка этого управляющего слова необходима для достижения некоторого особого поведения FPU, следует выполнить ее вручную, посредством вызова функции `Set8087()`, определенной в модуле `System`.

Библиотека VCL

При работе с библиотекой VCL могут возникнуть некоторые осложнения, но в большинстве случаев для переноса приложения в среду Delphi 5 потребуется лишь слегка отредактировать исходный текст проекта. Перечень возможных проблем приведен ниже.

- Тип свойства, представляющего индекс в списке графических изображений, был изменен с `Integer` на тип `TImageIndex`. Тип `TImageIndex` — это строго типизированное целое, определяемое в модуле `ImgList` следующим образом:

  ```
  TImageIndex = type Integer;
  ```

  Проблемы такого рода могут возникнуть только в тех случаях, если необходимо точное соответствие типов (например, при передаче `var`-параметров).

- В методе `TCustomTreeview.CustomDrawItem()` появился новый параметр типа `Boolean`, носящий имя `PaintImages`. Если в вашем приложении этот метод переопределяется, потребуется добавить в него указанный параметр, иначе успешная компиляция в Delphi 5 будет невозможна.

- Переменная `CoInitFlags` в объекте `ComObj`, предназначенная для хранения флажков, передаваемых методу `CoInitializeEx()` в модуле `ComServ`, была изменена с целью корректной поддержки инициализации многопоточных COM-серверов. Теперь, в соответствии с ситуацией, в нее будет добавлен флаг `COINIT_MULTITHREADED` или `COINIT_APARTMENTTHREADED`.

- Если приложение Delphi 4 выводит контекстные меню в ответ на сообщение `WM_RBUTTONUP` или при обработке события `OnMouseUp`, то после его компиляции в Delphi 5 оно будет выводить либо "удвоенные" контекстные меню, либо вовсе прекратит их вывод. В Delphi 5 для организации вывода контекстного меню используется сообщение `WM_CONTEXTMENU`.

Разработка приложений для Internet

Для тех, кто занят разработкой приложений для Internet, у нас есть как плохие новости, так и хорошие.

- Компонент `TWebBrowser`, инкапсулирующий элемент управления ActiveX Microsoft Internet Explorer, заменил в новой версии Delphi компонент `THTML` компании Netmasters. Компонент `TWebBrowser` имеет существенно больший диапазон возможностей, однако при замене им компонента `THTML` потребуется значительная переработка приложений, так как интерфейсы этих компонентов различны. Если нет желания вносить изменения в

уже созданные приложения, можно обеспечить использование в них прежнего компонента, для чего необходимо импортировать файл HTML.OCX, расположенный в каталоге \Info\Extras\NetManage на компакт-диске с дистрибуцией Delphi 5.

- Теперь при создании библиотек DLL ISAPI и NSAPI обеспечивается поддержка пакетов. Этим преимуществом воспользоваться очень просто — достаточно заменить в объявлении uses модуль HTTPApp модулем WebBroker.

Работа с базами данных

При переводе в среду Delphi 5 приложений, работающих с базами данных, перечень возможных осложнений совсем невелик и включает переименование нескольких существовавших ранее имен и новую архитектуру приложений MIDAS.

- Тип события TDatabase.OnLogin был изменен с TLoginEvent на TDatabaseLoginEvent. Это изменение едва ли способно вызвать проблемы, поскольку они возможны только в том случае, когда в приложении назначается обработчик события OnLogin.
- Глобальные подпрограммы FMTBCDToCurr() и CurrToFMTBCD() были заменены новыми подпрограммами BCDToCurr() и CurrToBCD(). (Соответствующие им закрытые методы класса TDataSet были заменены закрытым и недокументированным методом DataConvert.)
- При переходе от Delphi 4 к Delphi 5 архитектура компонентов MIDAS подверглась серьезной переработке. Обо всех внесенных изменениях подробно рассказывается в главе 32 второго тома, “Разработка приложений MIDAS”, помещенной во второй том данной книги. Там же детально описаны вновь появившиеся возможности этой технологии, а также процедура переноса MIDAS-приложений Delphi 4 в среду Delphi 5.

Переход с Delphi 3

Несмотря на то что областей, в которых Delphi 3 и последующие версии этого продукта несовместимы, не так уж много, в некоторых случаях такая несовместимость служит потенциальным источником более серьезных проблем, нежели те, которые возникают при переходе с любых предыдущих версий к последующим. Большинство проблем несовместимости касается использования новых типов данных и изменения поведения некоторых уже существовавших.

32-разрядные беззнаковые целые

В Delphi 4 введен тип LongWord — 32-разрядный беззнаковый целый тип. В предыдущих версиях наибольшим целым типом был 32-разрядный знаковый целый тип. По этой причине многие типы, которые должны принимать только беззнаковые (положительные) значения, такие как DWORD, UINT, HResult, HWND, HINSTANCE и другие типы дескрипторов, ранее определявшиеся просто как Integer, в Delphi 4 и последующих версиях переопределены как LongWord. Кроме того, тип Cardinal, ранее определявшийся как поддиапазон 0..MaxInt, сейчас также представляет собой тип LongWord. За исключением некоторых описанных ниже случаев, все эти изменения не оказывают никакого влияния на поведение программы.

- При передаче var-параметров типы Integer и LongWord не совместимы. Нельзя передавать значение типа LongWord в var-параметр типа Integer и наоборот. При обнаружении подобных действий компилятор сообщит об ошибке, поэтому для решения этой проблемы следует либо изменить тип параметра или переменной, либо выполнить явное приведение типов.

- Литеральные константы со значениями от $80000000 до $FFFFFFFF считаются имеющими тип LongWord. Следует выполнить приведение типа такой константы к Integer, если требуется присвоить ее целому типу:

```
var
  I: Integer;
begin
  I := Integer($FFFFFFFF);
```

- Любой литерал, имеющий отрицательное значение, выходит за пределы диапазона LongWord, и поэтому перед присвоением литерала с отрицательным значением переменной типа LongWord требуется выполнить соответствующее приведение типов:

```
var
  L: LongWord;
begin
  L := LongWord(-1);
```

- Если знаковые и беззнаковые целые смешиваются в одном арифметическом выражении или в одной операции сравнения, для вычисления этого выражения или выполнения сравнения компилятор автоматически приводит каждый операнд к типу Int64. Это неявное действие может послужить источником трудно выявляемых ошибок. Рассмотрим следующий код:

```
var
  I: Integer;
  D: DWORD;
begin
  I := -1;
  D := $FFFFFFFF;
  if I = D then DoSomething;
```

В Delphi 3 произойдет вызов DoSomething, так как -1 и $FFFFFFFF имеют одинаковое значение при переводе в Integer. В Delphi 4 каждый операнд переводится в Int64 для выполнения более точного сравнения, поэтому сгенерированный код будет сравнивать $FFFFFFFFFFFFFFFF и $00000000FFFFFFFF, которые определенно не равны между собой, и операторы *DoSomething* выполняться не будут.

Компилятор Delphi 4 и последующих версий генерирует большое количество новых подсказок, предупреждений и сообщений об ошибках, в том числе и относящихся к описанным проблемам совместимости и неявного преобразования типов. Надеемся, что вы будете использовать режим выдачи подсказок и предупреждений компилятора, что поможет вам в написании программ, свободных от подобных ошибок.

64-разрядный целый тип

В Delphi 4 также был введен новый тип, именуемый Int64, который представляет собой 64-разрядный знаковый целый тип. Он широко используется в библиотеках RTL и VCL. Так, например, стандартные функции Trunc() и Round() возвращают значение типа Int64. Кроме того, появились новые версии функций IntToStr(), IntToHex() и других связанных с ними функций, работающих с int64.

Действительный тип

В Delphi версии 4 и последующих тип `Real` является псевдонимом типа `Double`. В предыдущих версиях Delphi и Turbo Pascal тип `Real` представлял собой 6-байтовый тип чисел с плавающей точкой. Это не составляет проблемы, если только значения типа `Real` не сохранялись записанными в двоичных файлах (например, в `file of record`), созданных приложениями предыдущих версий, и текст программ не зависит от конкретной организации значений типа `Real` в памяти. Можно принудительно сделать тип `Real` прежним, 6-байтовым типом, включив директиву `{$REALCOMPATIBILITY ON}` в модули, в которых необходимо использовать старое представление действительных типов данных. Если же все, что вам нужно, — это "заставить" некоторое ограниченное количество экземпляров типа `Real` иметь прежний размер, следует вместо этой директивы воспользоваться типом `Real48`.

Переход с Delphi 2

Переходя с Delphi 2, вы обнаружите, что высокая степень совместимости Delphi 2 с последующими версиями означает возможность довольно гладкого перехода к последующим версиям Delphi. Тем не менее с момента выпуска Delphi 2 произошли некоторые изменения и в языке, и в подпрограммах библиотеки VCL, и вы должны знать о том, как перейти от Delphi 2 к последней версии Delphi и как воспользоваться всеми преимуществами ее новых возможностей.

Изменения в булевых типах

Реализация булевых типов в Delphi 2 (`Boolean`, `ByteBool`, `WordBool`, `LongBool`) предопределяет, что значение `True` представляется целым значением `1`, а значение `False` — целым значением `0`. Для обеспечения лучшей совместимости с Win32 API реализация `ByteBool`, `WordBool` и `LongBool` немного изменилась: значение `True` теперь представляется целым — `-1` (`$FF`, `$FFFF` и `$FFFFFFFF` соответственно). Тип же `Boolean` остался прежним. Эти изменения могут сказаться на поведении программ, но только если в них используются числовые представления значений этих типов. Взгляните на следующее объявление:

```
var
  A: array[LongBool] of Integer;
```

В Delphi 2 этот код вполне безобиден — он объявляет массив целых `array[False..True]` или `[0..1]` с двумя элементами. Однако в Delphi 3 эта декларация может привести к некоторым весьма неожиданным результатам: поскольку `True` определяется как `$FFFFFFFF` для `LongBool`, подобная декларация приводит к созданию массива `array[0..$FFFFFFFF]` целых или массива из четырех миллиардов (!) целых типа `Integer`. Для решения этой проблемы в качестве индекса массива используйте тип `Boolean`.

Это изменение вызвано тем, что большинство элементов управления ActiveX и контейнеров элементов управления (типа Visual Basic) "опрашивает" значение булевых переменных посредством сравнения с целым значением `-1`, а не посредством проверки на нулевое или ненулевое значение.

Для гарантии переносимости кода и отсутствия в нем ошибок никогда не используйте подобные выражения:

```
if BoolVar = True then ...
```

Вместо этого всегда проверяйте булевы типы следующим образом:

```
if BoolVar then ...
```

Строковые ресурсы

Если в приложении используются строковые ресурсы, рекомендуем воспользоваться преимуществом описания `ResourceString` (см. главу 2, "Язык программирования Object Pascal"). Это не увеличит размер приложения и скорость его выполнения, но зато упростит процесс трансляции. Объявление `ResourceString` и другие близкие к этому методы использования ресурсных DLL важны при написании приложений, отображающих информацию на различных языках, но работающих при этом с одним и тем же пакетом ядра библиотеки VCL.

Изменения в RTL

Некоторые изменения, внесенные в библиотеку времени выполнения (RTL) после выхода версии Delphi 2, могут привести к проблемам при переносе приложений в новую среду. Во-первых, изменилось значение глобальной переменной `HInstance` — теперь она содержит дескриптор экземпляра текущей DLL, `.exe`-файла или пакета. Для получения дескриптора экземпляра основного приложения используйте новую глобальную переменную `MainInstance`.

Второе важное изменение имеет отношение к глобальной переменной `IsLibrary`. В Delphi 2 значение переменной `isLibrary` проверялось для выяснения того, как выполняется данный код — из контекста библиотеки DLL или из `EXE`-файла. Переменная `isLibrary` не знает о существовании пакетов, поэтому не следует полагаться на нее при определении способа вызова программы (из `EXE`-файла, библиотеки DLL или модуля пакета). Для этого лучше использовать глобальную переменную `ModuleIsLib`, которая возвращает `True` при вызове программы из библиотеки DLL или пакета. Кроме того, переменную `ModuleIsLib` можно использовать в комбинации с глобальной переменной `ModuleIsPackage`, что позволит отличить библиотеку DLL от пакета.

Класс `TCustomForm`

В библиотеке VCL Delphi 3 появился новый промежуточный класс между классами `TScrollingWinControl` и `TForm`, называемый `TCustomForm`. Сам по себе этот факт не усложняет перенос приложений Delphi 2 в новую среду, но, если программа работает с экземплярами класса `TForm`, ее текст следует обновить так, чтобы вместо класса `TForm` использовался класс `TCustomForm`. В частности, подобные действия необходимы, при вызове в программе функций `GetParentForm()`, `ValidParentForm()`, а также при любом использовании класса `TDesigner`.

Со времен Delphi 2 несколько изменилась семантика методов `GetParentForm()`, `ValidParentForm()` и других методов компонентов библиотеки VCL, возвращающих указатели `Parent`. Эти подпрограммы теперь могут возвращать значение `nil`, даже если компонент имеет контекст родительского окна, в котором выполняется прорисовка. Например, если компонент инкапсулирован в элемент управления ActiveX, он может иметь родительское окно, но не элемент управления `Parent`. Это значит, что необходимо проследить, чтобы в коде Delphi 2 не встречались подобные вещи:

```
with GetParentForm(xx) do ...
```

Метод `GetParent()` теперь может вернуть `nil`, в зависимости от того, каким образом инкапсулирован компонент.

Метод GetChildren()

Разработчики компонентов! Имейте в виду, что объявление TComponent.GetChildren() изменилось и выглядит сейчас так:

```
procedure GetChildren(Proc: TGetChildProc; Root: TComponent); dynamic;
```

В новом параметре Root содержится указатель на корневого владельца компонента, т.е. на тот компонент, у которого при перемещении по цепи компонентов-владельцев свойство Owner окажется равным nil.

Серверы автоматизации

Код, требуемый для использования функций автоматизации, значительно изменился — по сравнению с тем, каким он был в Delphi 2. (Создание серверов автоматизации в Delphi 5 описывается в главе 23 второго тома, "COM-ориентированные технологии".) Мы считаем, что нет смысла подробно описывать эти различия, достаточно сказать, что не следует стили создания серверов автоматизации, применяемые в Delphi 2, использовать в последующих версиях Delphi.

В Delphi 2 автоматизация осуществляется с использованием возможностей модулей OleAuto и Ole2. Эти модули присутствуют в последующих версиях Delphi лишь в целях обратной совместимости, и их не рекомендуется использовать в новых проектах. В настоящее время та же функциональность обеспечивается модулями ComObj, ComServ и ActiveX. Не следует смешивать старые модули с новыми в одном проекте.

Переход с Delphi 1

Большинство изменений при переносе приложений Delphi 1 в последующие версии связано с программированием в новой операционной системе, но есть и такие, которые связаны с изменениями в программах библиотеки VCL и языке Object Pascal. Некоторые программисты предпочитают запускать немодифицированные приложения Delphi 1 под управлением Delphi 5. Если вы с ними солидарны, имейте в виду, что иногда стоит пожертвовать малым для значительного продвижения вперед, а каждая очередная версия Delphi предоставляет все условия для такого продвижения.

Помимо обсуждения переноса приложений Delphi 1 в среду Delphi 5, в этом разделе речь пойдет и об оптимизации проектов для 32-разрядных версий Delphi и создании текстов, в равной степени совместимых и с 16-разрядной, и с 32-разрядной версиями Delphi.

Строки и символы

Для повышения гибкости использования строк Borland включила в Delphi 2 новый строковый тип, известный как AnsiString. Помимо прочих преимуществ, тип AnsiString поддерживает создание виртуальных строк неограниченной длины. В Delphi 2 также были введены новые символьные и завершающиеся нулевым символом (null-terminated) строковые типы, необходимые для полной поддержки средств интернационализации приложений с использованием двухбайтового формата Unicode. В Delphi 3 поддержка двухбайтового формата была расширена введением типа WideString. По этим причинам большинство проблем перехода с Delphi 1 связано с использованием и манипуляциями строками.

Новые символьные типы

Строки, как известно, состоят из символов, поэтому, прежде чем познакомиться с новыми строковыми типами, нужно разобраться с устройством символьных типов в Delphi 5. Наиболее существенное изменение в этой части языка — новый символьный тип `WideChar`, введенный в Delphi 2 для поддержки двухбайтовых символов кода `Unicode` ("широких" символов) в символьных и строковых типах данных. В дополнение к типу `WideChar`, в Delphi 3 было введено новое имя типа `AnsiChar`, определяющего обычный однобайтовый символ.

Тип `AnsiChar` — это то же, что и тип `Char` в Delphi 1. Его один байт может представлять 256 различных значений. Используйте `AnsiChar` только в том случае, если вы уверены, что используемое для него значение всегда будет иметь размер, равный одному байту.

Тип `WideChar` использует для представления значения два байта и предназначен для совместимости с кодировкой `Unicode`, используемой Win32 API для поддержки языковой локализации. Двухбайтовый `WideChar` может представлять 65 536 различных возможных значений, что вполне достаточно даже для самого большого алфавита.

Цель введения стандарта Unicode — поддержка строк на местном языке во всей системе (и всей глобальной сети) без потерь информации. Для большинства языков (кроме языков, использующих иероглифы) имеются соответствующие наборы однобайтовых символов, но преобразование между наборами символов различных языков не всегда обратимо, т.е. происходит с потерей информации. Кодировка Unicode решает эту проблему способом полного исключения необходимости преобразования кодировок наборов символов. Более того, благодаря большому объему возможных кодов символов, кодировка Unicode позволяет работать и с дальневосточными языками, в которых для каждого возможного слога имеется отдельный иероглиф, как, например, в китайском.

Тем не менее, тип `Char` по-прежнему является в Delphi допустимым. В настоящее время типы `char` и `AnsiChar` эквивалентны, однако Borland оставляет за собой право изменять в будущих версиях Delphi определение `char` таким образом, чтобы этот тип соответствовал `WideChar`. Вам не следует полагаться в своем коде на значение длины `char`, поэтому всегда используйте функцию `SizeOf()` для получения действительного размера этого типа.

Новые строковые типы

Ниже перечислены строковые типы, поддерживаемые в Delphi 5.

- `AnsiString` (называемый также "длинной строкой") — это новый стандартный строковый тип Object Pascal, используемый по умолчанию. Он состоит из символов `AnsiChar` и может иметь размер вплоть до гигабайта. Этот строковый тип совместим со строками с завершающим нулевым символом. Такие строки всегда размещаются динамически и представляют собой тип с управляемым временем жизни.
- `ShortString` — синоним стандартного строкового типа Delphi 1. Размер строки типа `ShortString` ограничен 255 символами.
- `WideString` — содержит символы `WideChar` и, подобно строкам `AnsiString`, динамически размещается в памяти и представляет собой тип с управляемым временем жизни. В главе 2, "Язык программирования Object Pascal", представлен полный обзор этих и остальных строковых типов.
- `PAnsiChar` — указатель на строку типа `AnsiChar` с завершающим нулевым символом.
- `PWideChar` — указатель на строку типа `WideChar` для двухбайтовых символов `Unicode` с завершающим нулевым символом.

- PChar — указатель на строку Char с завершающим нулевым символом, полностью совместимую со строками языка C, используемыми Windows API. Этот тип не изменился с первой версии Delphi и в настоящее время соответствует PAnsiChar.

По умолчанию строки, определенные в Delphi 2 и последующих версиях как строки типа String, являются строками типа AnsiString. Поэтому, если вы объявляете строку типа string так, как показано ниже, компилятор относит ее к типу AnsiString:

```
var
  S: String;  // S имеет тип AnsiString
```

Можно изменить эту установку с помощью директивы компилятора $H. Тип String будет считаться типом ShortString, если значение этой директивы компилятора отрицательное. Если значение этой директивы положительное (оно принимается по умолчанию), переменные типа String реально создаются с типом AnsiString. Ниже приведен код, демонстрирующий этот факт:

```
var
  {$H-}
  S1: String;  // S1 имеет тип ShortString
  {$H+}
  S2: String;  // S2 имеет тип AnsiString
```

Исключением из этого правила является случай, когда переменная типа String объявляется с явным указанием размера, равным менее чем 255 символам. Такая строка всегда имеет тип ShortString:

```
var
  S: String[63];  // ShortString размером не более 63 символов
```

Будьте осторожны при передаче строк, объявленных в модулях с директивой $H+, в функции и процедуры, определенные в модулях с директивой $H-, и наоборот. Это часто приводит к возникновению ошибок, которые крайне трудно найти при отладке.

Установка длины строки

В Delphi 1 можно было установить реальную длину строки типа string путем присвоения определенного значения ее нулевому байту, как показано ниже.

```
S[0] := 23;   { Установка байта длины короткой строки }
```

Это было возможно потому, что максимальная длина короткой строки (255 байт) могла быть размещена в единственном начальном байте. Учитывая, что максимальная длина длинной строки составляет один гигабайт, значение длины, очевидно, не поместится в один байт, и, таким образом, оно должно храниться иначе. По этой причине в Delphi 2 была введена новая стандартная процедура SetLength(), предназначенная для установки длины строки. Процедура SetLength() объявляется так:

```
procedure SetLength(var S: String; NewLength: Integer};
```

Процедура SetLength() может быть использована как для коротких, так и для длинных строк. Если желательно иметь общий исходный текст проекта и для 16-разрядной Delphi 1, и для 32-разрядной версии Delphi, функция SetLength() для варианта 16-разрядной Delphi 1 может быть определена следующим образом:

```
{$IFDEF WINDOWS}
{ Для варианта 16-разрядной Delphi 1}
procedure SetLength(var S: String; NewLength: Integer);
begin
  S[0] := Char(NewLength);
end;
{$ENDIF}
```

За дополнительной информацией о физическом устройстве типа данных AnsiString обратитесь к главе 2, "Язык программирования Object Pascal".

Динамически размещаемые строки

В Delphi 1 допускается использование переменных типа PString для реализации динамически размещенных строк с помощью стандартных процедур работы с памятью NewStr(), GetMem() и AllocMem(). В 32-разрядной Delphi, ввиду того что длинные строки автоматически размещаются в куче, нет нужды обращаться к подобной технологии — измените указатели PString на String, и программа динамически разместит их в памяти и при необходимости освободит ее. Потребуется только избавиться от разыменования переменных PString в тексте программы. Посмотрите на следующий фрагмент кода Delphi 1:

```
var
  S1, S2: PString;
begin
  S1 := AllocMem(SizeOf(S1^));
  S1^ := 'Give up the rock.';
  S2 := NewStr(S1^);
  FreeMem(S1, SizeOf(S1^));
  Edit1.Text := S2^;
  DisposeStr(S2);
end;
```

Этот код значительно упрощается и оптимизируется при использовании длинных строк, как показано ниже.

```
var
  S1, S2: string;
begin
  S1 := 'Give up the rock.';
  S2 := S1;
  Edit1.Text := S2;
end;
```

Индексирование строк как массивов

Иногда, чтобы получить доступ к определенному символу строки, необходимо проиндексировать строку как массив. Например, данный код назначает пятому символу строки значение 'A':

```
S[5] := 'A';
```

Этот тип операции поддерживается длинными строками, но есть один нюанс: поскольку длинные строки размещаются в памяти динамически, вы должны убедиться, что длина строки больше или равна индексу нужного символа. Например, следующий код неправильный:

```
var
  S: string;
begin
  S[5] := 'A';  // Место для S еще не выделено!
end;
```

А вот этот код правильный:

```
var
  S: string;
begin
  S:= 'Hello'    // Выделение памяти для строки S
  S[5] := 'A';
end;
```

Следующий вариант кода также является допустимым:

```
var
  S: string;
begin
  SetLength(S, 5); // Выделить 5 символов для S
  S[5] := 'A';
end;
```

Не следует смешивать понятия индекса символа в строке с понятием смещения байта в строке. Например, `WideStringVar[5]` определяет пятый символ (но с реальным смещением в байтах, равным десяти).

Строки с завершающим нулевым символом

При вызове функций Windows 3.1 API в Delphi 1 программист обязан знать различия между типом `String` языка Pascal и типом `PChar` языка C, представляющим строки с завершающим нулевым символом, используемые в Windows. Длинные строки упрощают вызов функций Win32 API. Они размещаются в динамической области памяти и являются строками с завершающим нулевым символом. Поэтому можно выполнить приведение типов переменной длинной строки для использования ее в качестве нуль-завершенной строки типа `PChar` в вызовах Win32 API. Предположим, имеется процедура `Foo()` с таким определением:

```
procedure Foo(P: PChar);
```

В Delphi 1 эту функцию следовало бы вызывать таким образом:

```
var
  S: string;            { Короткая строка Pascal }
  P: PChar;             { Нуль-завершенная строка }
begin
  S := 'Hello world';   { Инициализация S }
  P := AllocMem(255);   { Размещение P }
  StrPCopy(P, S);       { Копирование S в P }
  Foo(P);               { Вызов Foo с P }
  FreeMem(P, 255);      { Освобождение P }
end;
```

В 32-разрядной версии Delphi можно непосредственно вызвать функцию `Foo()` с переменной длинной строкой в качестве параметра:

```
var
  S: string;       // Длинная строка является нуль-завершенной
begin
  S := 'Hello World';
  Foo(PChar(S));   // Полная совместимость с типом PChar
end;
```

Это означает, что можно оптимизировать 32-разрядный код, удалив ненужные временные буферы, предназначенные для хранения нуль-завершенных строк.

Использование строки с завершающим нулевым символом как буфера

Зачастую переменные типа `PChar` используют как буфер, передаваемый в функцию API, которая помещает в него какую-то информацию. Классическим примером является обращение к функции `GetWindowsDirectory()`, определенной в Win32 API так:

```
function GetWindowsDirectory(lpBuffer: PChar; uSize: UINT): UINT;
```

Если нужно сохранить путь к каталогу Windows в некоторой строковой переменной, в Delphi1 обычно передают в функцию адрес первого элемента строки, как показано ниже.

```
var
  S: string;
begin
  GetWindowsDirectory(@S[1], 254); { 254 - нужно оставить место для нуля }
  S[0] := Chr(StrLen(@S[1]));      { Настройка длины }
end;
```

Эта технология не работает с длинными строками по двум причинам. Во-первых, как уже было указано, необходимо инициализировать строку для размещения ее в памяти. Во-вторых, поскольку длинная строка сама является указателем динамической области памяти, использование оператора `@` приведет к тому, что в функцию будет передан указатель на указатель, а это совсем не то, что требовалось! Для длинных строк используется технология приведения типов `string` и `PChar`:

```
var
  S: string;
begin
  SetLength(S, MAX_PATH + 1);      // Размещение памяти
  GetWindowsDirectory(PChar(S), MAX_PATH);
  SetLength(S, StrLen(PChar(S))); // Настройка длины
end;
```

Использование типа PChar как String

Поскольку длинные строки могут использоваться как тип PChar, логично будет предположить, что и обратное утверждение также справедливо. Нуль-завершенные строки совместимы с длинными строками по присвоению. В Delphi 1 следующий код требует вызова функции StrPCopy():

```
var
  S: string;
  P: PChar;
begin
  P := StrNew('Object Pascal');
  S := StrPas(P);
  StrDispose(P);
end;
```

Теперь те же операции можно свести к простому присвоению длинных строк:

```
var
  S: string;
  P: PChar;
begin
  P := StrNew('Object Pascal');
  S := P;
  StrDispose(P);
end;
```

Точно так же можно передавать нуль-завершенные строки в функции и процедуры, которым, в свою очередь, следует передавать параметры типа String. Предположим, что процедура Bar() определена следующим образом:

```
procedure Bar(S: string);
```

Можно вызвать эту процедуру, передав ей значение типа PChar:

```
var
  P: PChar;
begin
  P := StrNew('Hello');
  Bar(P);
  StrDispose(P);
end;
```

Однако эта технология не сработает с процедурами и функциями, которым следует передавать параметры типа String по ссылке. Пусть теперь процедура Bar() определяется так:

```
procedure Bar(var S: string);
```

Приведенный выше код не может быть использован для вызова процедуры Bar(). Его нужно изменить так, чтобы вместо данных типа PChar в процедуру передавалась временная переменная типа string:

```
var
  P: PChar;
  TempStr: string;
begin
  P := StrNew('Hello');
  TempStr := P;
  Bar(TempStr);
  StrDispose(P);
  P := PChar(TempStr);
end;
```

В Delphi 2 была введена стандартная процедура SetString(), которая позволяет копировать только часть строки PChar в переменную типа string. Процедура SetString() работает и с длинными, и с короткими строками. Вот ее определение:

```
procedure SetString(var S: string: Buffer: PChar; Len: Integer);
```

При присвоении переменной типа Pchar в качестве значения переменной типа string следует иметь в виду, что время жизни первой превосходит время жизни string. При выходе в процессе выполнения программы переменной типа string за пределы области видимости эта строка тут же освобождается, и переменная PChar будет указывать на мусор в памяти. Вот код, иллюстрирующий данную проблему:

```
var
  P: PChar;

procedure Bar(var P: PChar);
var
  S: String;
begin
  S := 'Hola Mundo';
  P := PChar(S);   // Переменная P здесь действительна
end;               // Здесь S освобождается

procedure Foo;
begin
  Bar(P);
  ShowMessage(P);  // ОШИБКА! Переменная P уже недействительна
end;
```

Размер и диапазон переменных

Другой важный вопрос при переносе программ возникает в связи с тем, что размер некоторых типов при переходе из 16- в 32-разрядную среду изменился (а, значит, изменился и диапазон их значений). В табл. 15.2 и 15.3 приведена информация об этих изменениях.

Таблица 15.2. Изменения размера переменных

Тип	Размер в байтах в 16-разрядной Delphi	Размер в байтах в 32-разрядной Delphi
`Integer`	2	4
`Cardinal`	2	4
`String`	256	4

Таблица 15.3. Изменения диапазона переменных

Тип	Диапазон в 16-разрядной Delphi	Диапазон в 32-разрядной Delphi
`Integer`	-32768..32767	-2147483648..2147483647
`Cardinal`	0..65536	0..2147483647
`String`	255 символов	1 Гбайт символов

В большинстве случаев эти изменения никак не скажутся на поведении создаваемых приложений. Там, где важны размеры типов, используйте функцию `SizeOf()`. Если приведенные типы данных записываются в двоичные файлы или BLOB-файлы 16-разрядной версии Delphi, то при чтении данных в 32-разрядной версии Delphi следует учесть изменение их размеров. В табл. 15.4 приведены типы 32-разрядной версии Delphi, сохраняющие двоичную совместимость с типами Delphi 1.

Таблица 15.4. Совместимость типов переменных

Тип 16-разрядной Delphi	Совместимый тип 32-разрядной Delphi
`Integer`	`SmallInt`
`Cardinal`	`Word`
`String`	`ShortString`

Выравнивание записей

В 32-разрядной Delphi принято размещать записи с выравниванием. Так, 32-разрядные данные, такие как `Integer`, выравниваются по 32-битовой границе (`DWORD`), а 16-разрядные, такие как `Word`, — по 16-битовой границе.

```
type
  TX = record
    B: Byte;
    L: Longint;
  end;
```

На заметку Выравнивание данных выполняется для повышения производительности процессора при операциях доступа к памяти.

С использованием стандартных установок компилятора функция Delphi 1 `SizeOf(TX)` вернет значение 5, тогда как в Delphi 2 и последующих версиях функция `SizeOf(TX)` вернет значение 8. Обычно это не важно, но может иметь значение, если для определения размера записи в коде функция `SizeOf()` не используется или если записи хранятся в двоичном файле.

Тридцатидвухразрядный компилятор Delphi выравнивает границы элементов записи по границам `DWORD`, потому что это позволяет ему генерировать оптимизированный код. Если существуют основания заблокировать такое поведение компилятора, при объявлении типа следует использовать модификатор `packed`:

```
type
  TX = packed record
    B: Byte;
    L: Longint;
  end;
```

Если запись `TX` определена как `packed record`, функция `SizeOf(TX)` вернет значение 5 и в любой 32-разрядной версии Delphi. Можно установить режим сжатия записей, принимаемый по умолчанию, с помощью директивы компилятора `$A-`.

Совет Поля в записи можно перетасовать так, чтобы они начинались со своих естественных границ (например, переместить поля `Byte`, чтобы они шли перед полем `Integer`, а не с двух сторон от него). Тем самым будет достигнута оптимальная упаковка записи без снижения производительности из-за невыровненности данных.

32-разрядная математика

Существенно более значительные изменения, относящиеся к размеру переменных, связаны с тем, что 32-разрядный компилятор Delphi автоматически выполняет оптимизированные 32-разрядные математические операции для всех операндов выражения (в Delphi 1 используется 16-разрядная математика). Посмотрите на следующий код Object Pascal:

```
var
  L: longint;
  w1, w2: word;
begin
  w1 := $FFFE;
  w2 := 5;
  L := w1 + w2;
end;
```

В Delphi 1 значение переменной `L` на выходе из подпрограммы равно 3, так как сумма переменных `w1+w2` вычисляется как 16-разрядное значение и происходит переполнение разрядной сетки. В Delphi 3 значение переменной `L` на выходе из подпрограммы будет равно `$10003`, поскольку сложение переменных выполняется с использованием 32-разрядной математики. Следствием внесенных в Delphi изменений является то, что если в приложении

Delphi 1 для обнаружения ошибок переполнения использовались функции компилятора по контролю за выходом значений из допустимого диапазона, то в последующих версиях Delphi для этих целей следует использовать другие средства.

Тип TDateTime

Для обеспечения совместимости с OLE и Win32 API изменено нулевое значение переменной TDateTime. Значения даты начинаются с 00.00.0000 в Delphi 1 и с 30.12.1899 в 32-разрядной Delphi. Несмотря на то что это изменение не влияет на хранение даты в полях баз данных, оно может сказаться на хранении даты в двоичном виде в бинарном файле или поле BLOB базы данных.

Раздел Finalization модуля

В Delphi 1 имеются процедура AddExitProc() и указатель ExitProc, позволяющие определить для модуля процедуру, содержащую код, выполняемый при выходе из программы. В 32-разрядной Delphi процесс добавления процедуры выхода заметно упростился благодаря появлению раздела модуля finalization. Эта процедура задумана, как аналог раздела initialization, и ее код гарантированно вызывается при закрытии приложения. Хотя эта секция не обязательно должна присутствовать в приложении для успешной компиляции его в 32-разрядной Delphi, тем не менее она позволяет создать более ясный код.

На заметку Преобразование процедур ExitProc в блоки finalization обязательно для пакетов. Пакеты могут многократно динамически загружаться и выгружаться во время разработки, а процедура ExitProc не вызывается при динамической выгрузке пакета из IDE. Поэтому "очищающий" код для пакета обязательно следует поместить в секцию finalization.

Обратите внимание на приведенный ниже код инициализации и выхода из программы Delphi 1.

```
procedure MyExitProc;
begin
  MyGlobalObject.Free;
end;
initialization
  AddExitProc(MyExitProc);
  MyGlobalObject := TGlobalObject.Create;
end.
```

В 32-разрядной Delphi этот код может быть значительно упрощен за счет использования секции finalization:

```
initialization
  MyGlobalObject := TGlobalObject.Create;
finalization
  MyGlobalObject.Free;
end.
```

Использование языка ассемблера

Поскольку язык ассемблера очень сильно зависит от той вычислительной платформы, для которой он создан, встроенный в приложение Delphi 1 16-разрядный компилятор языка ассемблера не будет работать в 32-разрядной Delphi. Необходимо переписать все подпрограммы на этом языке с использованием языка 32-разрядного ассемблера.

Следует добавить, что в среде Win32 поддерживаются не все прерывания. Примером прерывания, которое уже не поддерживается системой Win32, является прерывание `$31` с набором функций DPMI. В некоторых случаях функции этого набора могут быть заменены функциями и процедурами Win32 API (например, новыми функциями ввода-вывода файлов Win32). Если в приложении выполняется обработка прерываний, обратитесь к документации к системе Win32, для того чтобы найти подходящий способ их замены.

Кроме того, встроенный шестнадцатеричный код больше не поддерживается компилятором 32-разрядной Delphi. Замените подпрограммы, использующие такой код, подпрограммами на языке 32-разрядного ассемблера.

Соглашения о вызовах

Для передачи параметров и очистки стека после вызовов функций и процедур Delphi 1 может использовать соглашение о вызовах `cdecl` или `pascal` (по умолчанию используется `pascal`).

В Delphi 2 появились новые директивы, представляющие два новых соглашения о вызовах — `register` и `stdcall`. В Delphi 2 и 3 по умолчанию действует соглашение `register`, обеспечивающее более высокую производительность. При его использовании первых три 32-разрядных параметра передаются в регистрах `eax`, `edx` и `ecx` соответственно. Для оставшихся параметров используется соглашение `pascal`. Соглашение о вызовах `stdcall` — это своеобразный гибрид соглашений `pascal` и `cdecl`, т.е. параметры в нем передаются так же, как и в `cdecl`, но стек очищается как в `pascal`.

В Delphi 3 введена новая процедурная директива `safecall`, которая следует соглашению `stdcall` в части передачи параметров, но позволяет обрабатывать ошибки функций COM способом, более близким к принятому в Delphi. Большинство функций COM возвращает сведения об ошибке в виде значения `HRESULT`, тогда как стиль обработки ошибок в Delphi заключается в использовании структурированной обработки исключительных ситуаций. При вызове в Delphi функций по соглашению `safecall`, возвращаемое ими значение `HRESULT` преобразуется в исключительную ситуацию, которая может быть обработана в программе обычным способом. При реализации функций типа `safecall` средствами Delphi любая возникшая исключительная ситуация будет преобразована в значение `HRESULT`, возвращаемое вызывающей программе.

На заметку

В то время как функции и процедуры 16-разрядного Windows API используют соглашение о вызовах `pascal`, функции и процедуры Win32 API применяют соглашение `stdcall`. Таким образом, если в вашем коде имеются *функции обратного вызова* (callback function), все они должны использовать соглашение `stdcall`. Вот как описана функция обратного вызова, используемая в функции `EnumWindows()` 16-разрядной Windows:

```
function EnumWindowsProc(Handle: hwnd; lParam: Longint): BOOL; export;
```

В 32-разрядной Windows эта функция определяется так:

```
function EnumWindowsProc(Handle: hwnd; lParam: Longint): BOOL; stdcall;
```

Библиотеки DLL

Создание и использование библиотек DLL — практически одинаковый процесс во всех версиях Delphi, однако разные версии имеют ряд небольших отличий. Некоторые из них перечислены ниже.

- Благодаря линейной модели памяти Win32 директива `export`, необходимая в Delphi 1 для функций обратного вызова и функций библиотек DLL, в более поздних версиях не нужна. Она просто игнорируется компилятором.
- При разработке библиотеки DLL, предназначенной для вызова из выполняемых файлов, созданных с помощью других средств разработки, для максимальной совместимости рекомендуется использовать соглашение о вызовах `stdcall`.
- Для экспортирования функций из библиотек DLL Win32 предпочтительнее использовать их имена, а не порядковый номер. Ниже приведен пример экспортирования функций по их порядковым номерам в Delphi 1.

```
function SomeFunction: integer; export;
begin
.
.
end;

procedure SomeProcedure; export;
begin
.
.
end;

exports
  SomeFunction index 1,
  SomeProcedure index 2;
```

А вот те же функции, но экспортируемые по имени, как принято в 32-разрядной Delphi:

```
function SomeFunction: integer; stdcall;
begin
.
.
end;

procedure SomeProcedure; stdcall;
begin
.
.
end;

exports
  SomeFunction name 'SomeFunction';
  SomeProcedure name 'SomeProcedure';
```

- Экспортируемые имена чувствительны к регистру. Следует использовать правильный регистр как при импортировании функций, так и в вызове функции `GetProcAddress()`.
- При импортировании функции или процедуры и определении имени библиотеки после директивы `external`, в определении должно присутствовать расширение файла библиотеки. По умолчанию принято расширение `.DLL`.
- В Windows 3.x библиотека DLL имела в памяти только один сегмент данных, разделяемый всеми экземплярами этой библиотеки. Таким образом, если два приложения A и B загружали одну и ту же библиотеку DLL C, то изменения глобальных переменных, внесенные в приложении A, были видны в приложении B (и наоборот). В Win32 каждая библиотека DLL получает свой сегмент данных, а это значит, что теперь изменения глобальных данных библиотеки DLL, внесенные в одной программе, не будут видны в другой.

За дополнительной информацией о поведении DLL в среде Win32 обратитесь к главе 9, "Динамически компонуемые библиотеки".

Изменения в операционной системе Windows

Некоторые изменения в 32-разрядной архитектуре Windows оказывают влияние на код, написанный в Delphi. Это касается изменений, связанных с 32-разрядной моделью памяти, изменений форматов ресурсов, неподдерживаемых возможностей и изменений в Windows API.

32-разрядное адресное пространство

Система Win32 предоставляет приложениям 4-гигабайтовое плоское адресное пространство. Термин "плоское" означает, что все регистры сегментов содержат одинаковое значение и указатель определяется через смещение в этом 4-гигабайтовом пространстве. Поэтому код любых приложений Delphi 1, основанный на концепции указателя, состоящего из селектора и смещения, должен быть изменен для соответствия новой архитектуре.

Следующие элементы библиотеки времени выполнения Delphi 1 основаны на 16-разрядных указателях и не используются в RTL 32-разрядных версий Delphi: `DSeg`, `SSeg`, `CSEG`, `Seg`, `Ofs` и `SPtr`.

Поскольку в системе Win32 применяется иной метод организации файла страничного обмена на жестком диске, предназначенного для симуляции оперативной памяти, функции Delphi 1 `MemAvail()` и `MaxAvail()` больше не подходят для определения объема доступной памяти. Для получения этой информации в 32-разрядной версии Delphi используется функция библиотеки RTL `GetHeapStatus()`, определяемая следующим образом:

```
function GetHeapStatus: THeapStatus;
```

Запись типа `THeapStatus` разработана для предоставления информации о динамической области памяти данного процесса (в байтах). Вот ее определение:

```
type
  THeapStatus = record
    TotalAddrSpace: Cardinal;
    TotalUncommitted: Cardinal;
    TotalCommitted: Cardinal;
    TotalAllocated: Cardinal;
```

```
    TotalFree: Cardinal;
    FreeSmall: Cardinal;
    FreeBig: Cardinal;
    Unused: Cardinal;
    Overhead: Cardinal;
    HeapErrorCode: Cardinal;
  end;
```

И опять-таки, природа Win32 такова, что значение объема свободной памяти имеет мало смысла. Большинство пользователей обнаружат, что поле `TotalAllocated`, отображающее общее количество памяти, распределенной для текущего процесса, содержит более полезную для отладки информацию.

На заметку В главе 3, "Win32 API", вы найдете дополнительную информацию о внутреннем устройстве операционной системы Win32.

32-разрядные ресурсы

Перед использованием в 32-разрядной Delphi ресурсов Delphi 1, представленных в виде файлов `.RES` или `.DCR` и связанных с приложением или компонентом, следует создать 32-разрядные версии этих файлов. Обычно это довольно просто — нужно лишь открыть ресурсы во встроенном редакторе Image Editor или в отдельном редакторе ресурсов (например, в таком как Resource Workshop), после чего сохранить их в 32-разрядном совместимом формате.

Элементы управления VBX

Поскольку Microsoft больше не поддерживает 16-разрядных элементов управления VBX в 32-разрядных приложениях для Windows 95 и Windows NT, они не поддерживаются и в 32-разрядных версиях Delphi. Элементы управления ActiveX (OCX) успешно вытеснили элементы VBX на 32-разрядной платформе. Если необходимо перенести в новую среду приложение, использующее элементы управления VBX, следует заменить компоненты VBX эквивалентными им 32-разрядными элементами управления ActiveX.

Изменения в функциях Windows API

Часть функций Windows API изменилась при переходе с Windows 3.1 на Win32. Одних функций 16-разрядного API в Win32 больше не существует, другие функции вышли из употребления и используются лишь для обратной совместимости, третьим же следует передавать другие параметры, или же они возвращают иные значения. Все эти функции приведены в табл. 15.5 и 15.6. Более полную информацию можно найти в разделе интерактивной справочной системы Delphi 5, посвященном Win32 API.

Таблица 15.5. Устаревшие функции API Windows 3.x

Функция Windows 3.x	Аналог в Win32
`OpenComm()`	`CreateFile()`
`CloseComm()`	`CloseHandle()`
`FlushComm()`	`PurgeComm()`

Продолжение табл. 15.5.

Функция Windows 3.x	Аналог в Win32
GetCommError()	ClearCommError()
ReadComm()	ReadFile()
WriteComm()	WriteFile()
UngetCommChar()	Не определен
DlgDirSelect()	DlgDirSelectEx()
DlgDirSelectComboBox()	DlgDirSelectComboBoxEx()
GetBitmapDimension()	GetBitmapDimensionEx()
SetBitmapDimension()	SetBitmapDimensionEx()
GetBrushOrg()	GetBrushOrgEx()
GetAspectRatioFilter()	GetAspectRatioFilterEx()
GetTextExtent()	GetTextExtentPoint()
GetViewportExt()	GetViewportExtEx()
GetViewportOrg()	GetViewportOrgEx()
GetWindowExt()	GetWindowExtEx()
GetWindowOrg()	GetWindowOrgEx()
OffsetViewportOrg()	OffsetViewportOrgEx()
OffsetWindowOrg()	OffsetWindowOrgEx()
ScaleViewportExt()	ScaleViewportExtEx()
ScaleWindowExt()	ScaleWindowExtEx()
SetViewportExt()	SetViewportExtEx()
SetViewportOrg()	SetViewportOrgEx()
SetWindowExt()	SetWindowExtEx()
SetWindowOrg()	SetWindowOrgEx()
GetMetafileBits()	GetMetafileBitsEx()
SetMetafileBits()	SetMetafileBitsEx()
GetCurrentPosition()	GetCurrentPositionEx()
MoveTo()	MoveToEx()
DeviceCapabilities()	DeviceCapabilitiesEx()
DeviceMode()	DeviceModeEx()
ExtDeviceMode()	ExtDeviceModeEx()
FreeSelector()	Не определен
AllocSelector()	Не определен
ChangeSelector()	Не определен
GetCodeInfo()	Не определен
GetCurrentPDB()	GetCommandLine() и/или GetEnvironmentStrings()

Окончание табл. 15.5.

Функция Windows 3.x	Аналог в Win32
GlobalDOSAlloc()	Не определен
GlobalDOSFree()	Не определен
SwitchStackBack()	Не определен
SwitchStackTo()	Не определен
GetEnvironment()	(Функции файлового ввода-вывода Win32)
SetEnvironment()	(Функции файлового ввода-вывода Win32)
ValidateCodeSegments()	Не определен
ValidateFreeSpaces()	Не определен
GetInstanceData()	Не определен
GetKBCodePage()	Не определен
GetModuleUsage()	Не определен
Yield()	WaitMessage() и/или Sleep()
AccessResource()	Не определен
AllocResource()	Не определен
SetResourceHandler()	Не определен
AllocDSToCSAlias()	Не определен
GetCodeHandle()	Не определен
LockData()	Не определен
UnlockData()	Не определен
GlobalNotify()	Не определен
GlobalPageLock()	VirtualLock()

Таблица 15.6. Совместимость функций Win32 API

Функция Windows 3.x	Замена в Win32
DefineHandleTable()	Не определена
MakeProcInstance()	Не определена
FreeProcInstance()	Не определена
GetFreeSpace()	GlobalMemoryStatus()
GlobalCompact()	Не определена
GlobalFix()	Не определена
GlobalUnfix()	Не определена
GlobalWire()	Не определена
GlobalUnwire()	Не определена
LocalCompact()	Не определена
LocalShrink()	Не определена

Окончание табл. 15.6.

Функция Windows 3.x	Замена в Win32
`LockSegment()`	Не определена
`UnlockSegment()`	Не определена
`SetSwapAreaSize()`	Не определена

Параллельные 16- и 32-разрядные проекты

В этом разделе мы поговорим о разработке проектов, компилируемых как в 16-разрядной Delphi 1, так и в 32-разрядных версиях Delphi. Ниже приведено несколько полезных советов.

- Для Delphi 1 действует определение компилятора `Windows`, а для Delphi 5 — `Win32`. Вы можете использовать их для условной компиляции с помощью директив `{$IFDEF WINDOWS}` и `{$IFDEF WIN32}`.
- Не используйте компоненты или другие возможности 32-разрядных версий Delphi, не поддерживаемые Windows 3.1 или Delphi 1, если требуется компилировать приложение для 16-разрядной среды. В частности, избегайте использования компонентов Windows 95 и возможностей, не поддерживаемых в Windows 3.1 (например, таких как многопоточность). Самый простой способ обеспечить совместимость проектов Delphi 1 и Delphi 5 — разработать проект в Delphi 1 и затем скомпилировать его в Delphi 5.
- Не забывайте о различиях в функциях API. Если вам нужно применить процедуру или функцию API, по-разному реализованную на различных платформах, используйте условные определения `WINDOWS` и `WIN32`.
- В каждую последующую версию Delphi добавляются новые свойства или изменяются их характеристики по сравнению с предыдущими версиями. Это означает, что, когда компоненты Delphi 5 сохраняются в `.DFM`-файле, эти новые и измененные свойства также записываются. Несмотря на то что при загрузке проектов с измененными свойствами в Delphi 1 возникающие ошибки, как правило, можно игнорировать, лучше иметь два отдельных набора `.DFM`-файлов для каждой платформы.

Win32s

Еще одна возможность использования одного и того же кода в 16- и 32-разрядных версиях Windows заключается в попытке запуска 32-разрядного приложения Delphi под управлением системы Win32s, представляющей собой расширение Windows 3.x. Это расширение позволяет использовать некоторые функции Win32 API в 16-разрядной среде. Одним из наиболее серьезных недостатков данного метода является то, что многие элементы Win32, например потоки, не поддерживаются в Win32s (это, в частности, не позволяет использовать ядро Borland Database Engine, работающее с потоками). Если вы выбрали этот путь, имейте в виду, что Win32s не является официально поддерживаемой Delphi платформой, поэтому при возникновении каких бы то ни было сбоев вы сможете рассчитывать только на самих себя.

Резюме

Информация, приведенная в этой главе, поможет вам быстро и без осложнений перенести проекты из предыдущих версий Delphi в Delphi 5. Приложив некоторые дополнительные усилия, можно создавать проекты, прекрасно работающие в различных версиях Delphi.

Глава

16

MDI-приложения

Многодокументный интерфейс (Multiple Document Interface — MDI), впервые появился в Windows 2.0 в программе электронных таблиц Microsoft Excel. Он позволил пользователям Excel одновременно работать сразу с несколькими таблицами. Позже MDI-интерфейс стал использоваться в диспетчере программ и диспетчере файлов Windows 3.1. Одним из первых MDI-приложений стала программа Borland Pascal for Windows.

Перед появлением Windows 95 многим разработчикам казалось, что Microsoft собирается исключить возможности MDI. Но, к их удивлению, Microsoft сделала MDI частью Windows 95, и никаких намеков на то, что от этого интерфейса хотят избавиться, больше не было.

Фирма Microsoft объявила, что поддержка MDI-интерфейса в среде Windows является неэффективной и рекомендовала разработчикам избегать создания приложений с использованием этого типа интерфейса. Однако позднее Microsoft вновь обратилась к разработке собственных приложений на основе модели MDI, но уже без использования функций поддержки MDI Windows. Вы можете использовать многодокументный интерфейс в своих разработках, однако не следует забывать, что его реализация в Windows все еще неэффективна, и фирма Microsoft не планирует его модернизации. То, что предлагается в этой главе, представляет собой вполне безопасный вариант реализации многодокументного интерфейса.

Обработка событий нескольких форм одновременно может показаться довольно сложной проблемой. В традиционном Windows-программировании для этого потребуется знакомство с классом `MDICLIENT` Windows и знание структур данных MDI, а также использование ряда функций и сообщений, специфических для MDI-интерфейса. В Delphi 5 создание MDI-приложений предельно упрощено. Освоение материала этой главы даст вам прочный фундамент для построения собственных MDI-приложений, которые впоследствии можно будет легко модернизировать с целью включения других современных технологий.

Создание MDI-приложений

Создание MDI-приложений требует хотя бы поверхностного знакомства с методологией программирования MDI-интерфейса и со стилями форм `fsMDIForm` и `fsMDIChild`. В следующем разделе излагаются основные концепции MDI и демонстрируется их применение на примере специализированных дочерних MDI-форм.

Основы MDI-технологии

Для того чтобы понять принципы работы MDI-приложения, вначале следует разобраться с их устройством. На рис. 16.1 изображено MDI-приложение, подобное тому, создание которого будет описано в этой главе.

Ниже перечислены окна, характерные для интерфейса MDI-приложения.

- *Главное окно*. Это главное окно приложения, которое имеет панель заголовка, панель меню и кнопку системного меню. Кнопки минимизации, максимизации и закрытия окна находятся в верхнем правом углу. Свободное пространство внутри главного окна называется *клиентской областью* (client area) и обычно представляет собой клиентское окно.
- *Клиентское окно*. Выступает в качестве диспетчера MDI-приложений. Клиентское окно обрабатывает все специфические команды MDI-интерфейса и управляет дочерними окнами, расположенными на его поверхности — включая и функции их прорисовки. Клиентское окно автоматически создается подпрограммами библиотеки VCL при создании главного окна.

- *Дочерние окна*. Дочерние окна MDI-интерфейса представляют отдельные документы — текстовые файлы, электронные таблицы, изображения и др. Подобно главным окнам, дочерние окна имеют панель заголовка, кнопку системного меню, кнопки минимизации, максимизации, закрытия и (иногда) собственное меню. В дочернее окно можно поместить и кнопку вызова справки. Меню дочернего окна дополняет меню главного окна. Дочерние окна никогда не выходят за пределы клиентской области.

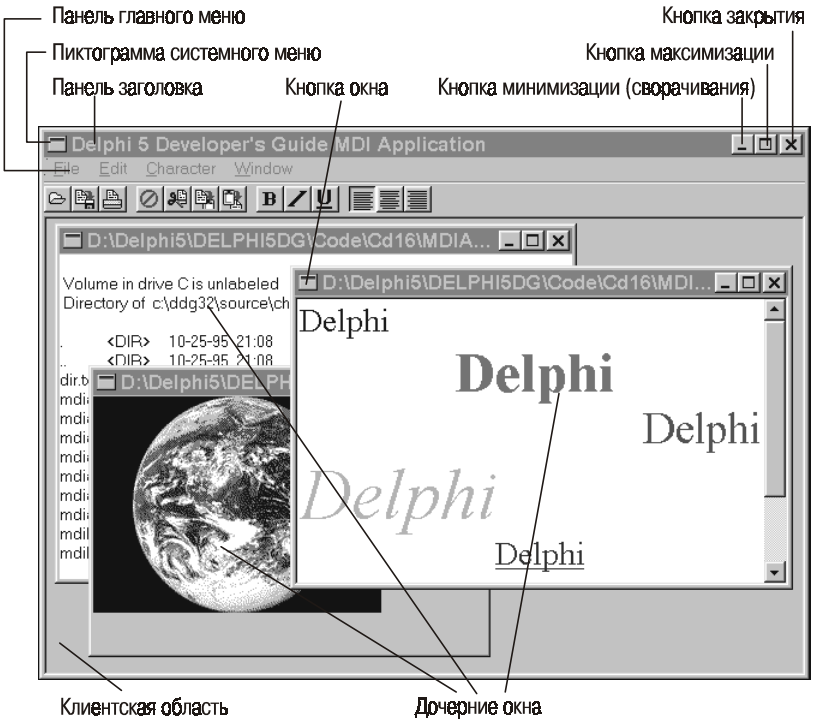


Рис. 16.1. Структура интерфейса MDI-приложения

Delphi 5 не требует от вас знания специальных сообщений окон MDI. За управление всеми функциями MDI-интерфейса отвечает клиентское окно — например, за вывод дочерних окон каскадом или мозаикой. В частности, для упорядочения дочерних окон каскадом традиционным методом нужно, чтобы функция Win32 API `SendMessage()` послала в клиентское окно сообщение `WM_MDICASCADE`:

```
procedure TFrameForm.Cascade1Click(Sender: TObject);
begin
  SendMessage(ClientHandle, WM_MDICASCADE, 0, 0);
end;
```

В Delphi 5 для получения того же результата достаточно вызвать метод `Cascade()`:

```
procedure TFrameForm.Cascade1Click(Sender: TObject);
begin
  cascade;
end;
```

Следующие разделы посвящены созданию законченного MDI-приложения, чьи дочерние MDI-окна будут поддерживать функции текстового редактора, просмотра файлов растровых изображений и редактора RTF-файлов. Цель создания этого приложения — показать, как строится MDI-приложение, дочерние окна которого должны обеспечивать отображение и обработку информации различных типов. В нашем примере текстовый редактор позволяет редактировать любые текстовые файлы, а редактор RTF-файлов — данные, представленные в формате RTF. Наконец, окно просмотра растровых изображений позволяет просматривать любые изображения в растровом формате Windows.

Мы также рассмотрим некоторые более сложные MDI-технологии, использующие Win32 API. Эти технологии в основном относятся к управлению дочерними формами MDI-приложений. Вначале мы проанализируем методы построения дочерних форм и обеспечения их функциональных возможностей, а потом обратимся к обсуждению главной формы.

Дочерняя форма

Как упоминалось выше, создаваемое MDI-приложение содержит три типа дочерних форм — `TMdiEditForm`, `TMdiRTFForm` и `TMdiBMPForm`. Каждый из этих трех типов является производным от класса `TMDIChildForm`, используемого в качестве базового. В следующем разделе этот класс описывается подробно, после чего мы рассмотрим три дочерние формы данного MDI-приложения.

Базовый класс `TMDIChildForm`

Дочерние формы, используемые в MDI-приложении, обладают некоторой общей функциональностью. Все они имеют одно и то же меню File, их свойствам `FormStyle` присвоено значение `fsMDIChild` и все они используют один и тот же компонент `TToolBar`. Избежать повторного определения всех этих установок для каждой из форм можно, сделав их производными от общего базового класса. Базовая форма `TMDIChildForm` определена в модуле `MdiChildFrm.pas`, текст которого показан в листинге 16.1.

Листинг 16.1. Модуль `MdiChildFrm.pas` — определение формы `TMDIChildForm`

```
unit MdiChildFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms,
  Dialogs, Menus, ComCtrls, ToolWin, ImgList;

type

  TMDIChildForm = class(TForm)
    { Список компонентов формы удален - см. текст модуля }
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiOpenClick(Sender: TObject);
    procedure mmiNewClick(Sender: TObject);
    procedure FormActivate(Sender: TObject);
```

```
    procedure FormDeactivate(Sender: TObject);
  end;

var
  MDIChildForm: TMDIChildForm;

implementation
uses MainFrm, Printers;

{$R *.DFM}

procedure TMDIChildForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
  { Переопределение объекта Parent панели инструментов }
  tlbMain.Parent := self;
  { Если это была последняя дочерняя форма, то делаем видимой панель
    инструментов главной формы }
  if (MainForm.MDIChildCount = 1) then
    MainForm.tlbMain.Visible := True
end;

procedure TMDIChildForm.mmiExitClick(Sender: TObject);
begin
  MainForm.Close;
end;

procedure TMDIChildForm.mmiCloseClick(Sender: TObject);
begin
  Close;
end;

procedure TMDIChildForm.mmiOpenClick(Sender: TObject);
begin
  MainForm.mmiOpenClick(nil);
end;

procedure TMDIChildForm.mmiNewClick(Sender: TObject);
begin
  MainForm.mmiNewClick(nil);
end;

procedure TMDIChildForm.FormActivate(Sender: TObject);
begin
  { Когда форма становится активной, панель инструментов главной формы
    следует убрать, после чего назначить и вывести в родительской форме
    панель инструментов данного дочернего окна. }
  MainForm.tlbMain.Visible := False;
  tibMain.Parent := MainForm;
  tlbMain.Visible := True;
end;
```

```
procedure TMDIChildForm.FormDeactivate(Sender: TObject);
begin
  { Дочерняя форма становится неактивной либо при ее уничтожении,
    либо в том случае, когда активной становится другая дочерняя форма.
    Убираем панель инструментов этой формы, чтобы можно было сделать
    видимой панель инструментов другой формы. }
  tlbMain.Visible := False;
end;

end.
```

На заметку Отметим, что удаление из приведенного выше листинга объявлений компонентов базового класса `TMDIChildForm` выполнено из соображений экономии места.

Класс `TMDIChildForm` содержит обработчики событий для пунктов главного меню и некоторых общих кнопок панели инструментов. В действительности кнопки панели инструментов просто связаны с обработчиками событий соответствующих пунктов меню. Некоторые из этих обработчиков событий вызывают методы главной формы. Например, обработчик события `mmiNewClick()` вызывает обработчик события `MainForm.mmiNewClick()` главной формы, осуществляющий создание новой дочерней MDI-формы. Имеются также другие обработчики событий, такие как `mmiOpenClick()` и `mmiExitClick()`, вызывающие соответствующие обработчики событий главной формы. Реализация функции класса `TMainForm` будет описана в разделе “Главная форма”.

Поскольку все дочерние MDI-окна имеют некоторую общую функциональность, разумно поместить ее в базовый класс, от которого они будут произведены. В этом случае в дочерних формах не потребуется определять одни и те же методы. Все они просто унаследуют главное меню и компоненты панели инструментов главной формы.

Обратите внимание на то, что в обработчике события `TMDIChildForm.FormClose()` параметру `Action` назначается значение `caFree` с целью гарантированного уничтожения экземпляра формы `TMDIChildForm` при ее закрытии. Основанием для этого является тот факт, что дочерние MDI-формы не закрываются автоматически при вызове метода `Close()`. Вы должны самостоятельно определить в обработчике события `OnClose`, что именно должно произойти с дочерней формой при вызове ее метода `Close()`. Обработчик события `OnClose` дочерней формы передает переменную `Action` перечислимого типа `TCloseAction` с одним из четырех присвоенных ей допустимых значений:

- `caNone`. Ничего не выполняется.
- `caHide`. Форма удаляется с экрана, но не уничтожается.
- `caFree`. Форма освобождается.
- `caMinimize`. Форма минимизируется (выполняется по умолчанию).

Когда дочерняя форма становится активной, вызывается ее обработчик события `OnActivate`. Всякий раз, когда дочерняя форма становится активной, выполняются определенные действия. Прежде всего, в обработчике `TMdiChildForm.FormActivate` делается невидимой панель инструментов главной формы, что позволяет сделать видимой панель инструментов дочерней формы. Кроме того, главная форма назначается родительским объектом па-

нели инструментов дочернего окна — поэтому данная панель инструментов будет отображаться в окне главной формы, а не дочерней. Это один из способов, который позволяет отображать в главной форме различные панели инструментов, когда становятся активными дочерние формы различных типов. В обработчике события `OnDeactivate` панель инструментов дочерней формы делается просто невидимой. Наконец, при обработке события `OnClose` дочерняя форма вновь назначается в качестве родительского объекта ее панели инструментов, и если текущая дочерняя форма являлась единственной в приложении, то делается видимой панель инструментов главного окна. В результате при работе приложения создается впечатление, что главная форма содержит единственную панель инструментов, состав кнопок которой изменяется в соответствии с типом активного дочернего окна.

Форма текстового редактора

Форма текстового редактора разрешает пользователю загружать и редактировать любой текстовый файл. Эта форма представлена классом `TMdiEditForm`, производным от класса `TMDIChildForm`. Она содержит компонент `TMemo`, выровненный по размерам клиентской области.

Класс `TMdiEditForm` также содержит компоненты `TPrintDialog`, `TSaveDialog` и `TFontDialog`. Форма `TMdiEditForm` не является автоматически создаваемой, и поэтому она удалена из списка автоматически создаваемых форм проекта в диалоговом окне Project Options.

На заметку Ни одна из форм MDI-проекта, кроме `TMainForm`, не создается автоматически; поэтому все они удалены из списка автоматически создаваемых форм. Эти формы создаются динамически при выполнении кода проекта.

Исходный текст формы `TMdiEditForm` приведен в листинге 16.2.

Листинг 16.2. Модуль `MdiEditFrm.pas` — определение формы `TMdiEditForm`

```
unit MdiEditFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Menus, ExtCtrls, Buttons, ComCtrls,
  ToolWin, MdiChildFrm, ImgList;

type

  TMdiEditForm = class(TMDIChildForm)
    memMainMemo: TMemo;
    SaveDialog: TSaveDialog;
    FontDialog: TFontDialog;
    mmiEdit: TMenuItem;
    mmiSelectAll: TMenuItem;
    N7: TMenuItem;
    mmiDelete: TMenuItem;
    mmiPaste: TMenuItem;
    mmiCopy: TMenuItem;
```

```
    mmiCut: TMenuItem;
    mmiCharacter: TMenuItem;
    mmiFont: TMenuItem;
    N8: TMenuItem;
    mmiWordWrap: TMenuItem;
    N9: TMenuItem;
    mmiCenter: TMenuItem;
    mmiRight: TMenuItem;
    mmiLeft: TMenuItem;
    mmiUndo: TMenuItem;
    N4: TMenuItem;
    mmiBold: TMenuItem;
    mmiItalic: TMenuItem;
    mmiUnderline: TMenuItem;
    PrintDialog: TPrintDialog;

    { Обработчики событий меню File }
    procedure mmiSaveClick(Sender: TObject);
    procedure mmiSaveAsClick(Sender: TObject);

    { Обработчики событий меню Edit }
    procedure mmiCutClick(Sender: TObject);
    procedure mmiCopyClick(Sender: TObject);
    procedure mmiPasteClick(Sender: TObject);
    procedure mmiDeleteClick(Sender: TObject);
    procedure mmiUndoClick(Sender: TObject);
    procedure mmiSelectAllClick(Sender: TObject);

    { Обработчики событий меню Character }
    procedure CharAlignClick(Sender: TObject);
    procedure mmiBoldClick(Sender: TObject);
    procedure mmiItalicClick(Sender: TObject);
    procedure mmiUnderlineClick(Sender: TObject);
    procedure mmiWordWrapClick(Sender: TObject);
    procedure mmiFontClick(Sender: TObject);

    { Обработчики событий формы }
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure mmiPrintClick(Sender: TObject);
  public
    { Методы, определенные пользователем }
    procedure OpenFile(FileName: String);
    procedure SetButtons;
  end;

var
  MdiEditForm: TMdiEditForm;

implementation
```

```
uses Printers;

{$R *.DFM}
{ Обработчики событий меню File }

procedure TMdiEditForm.mmiSaveClick(Sender: TObject);
begin
  inherited;
  { Если нет заголовка, то нет и имени файла. Необходимо вызвать
    mmiSaveAsClick для получения имени файла. }
  if Caption = '' then
    mmiSaveAsClick(nil)
  else begin
    { Сохранение в файл, определяемый заголовком формы. }
    memMainMemo.Lines.SaveToFile(Caption);
    memMainMemo.Modified := false; // Установка false, т.к. текст сохранен.}
  end;
end;

procedure TMdiEditForm.mmiSaveAsClick(Sender: TObject);
begin
  inherited;
  SaveDialog.FileName := Caption;
  if SaveDialog.Execute then
  begin
    { Помещение в заголовок имени файла, выбранного в SaveDialog1. }
    Caption := SaveDialog.FileName;
    mmiSaveClick(nil); // Сохранение файла
  end;
end;

{ Обработчики событий меню Edit }

procedure TMdiEditForm.mmiCutClick(Sender: TObject);
begin
  inherited;
  memMainMemo.CutToClipBoard;
end;

procedure TMdiEditForm.mmiCopyClick(Sender: TObject);
begin
  inherited;
  memMainMemo.CopyToClipBoard;
end;

procedure TMdiEditForm.mmiPasteClick(Sender: TObject);
begin
  inherited;
  memMainMemo.PasteFromClipBoard;
end;
```

```
procedure TMdiEditForm.mmiDeleteClick(Sender: TObject);
begin
  inherited;
  memMainMemo.ClearSelection;
end;

procedure TMdiEditForm.mmiUndoClick(Sender: TObject);
begin
  inherited;
  memMainMemo.Perform(EM_UNDO, 0, 0);
end;

procedure TMdiEditForm.mmiSelectAllClick(Sender: TObject);
begin
  inherited;
  memMainMemo.SelectAll;
end;

{ Обработчики событий меню Character }
procedure TMdiEditForm.CharAlignClick(Sender: TObject);
begin
  inherited;
  mmiLeft.Checked := false;
  mmiRight.Checked := false;
  mmiCenter.Checked := false;

{ TAlignment определяется библиотекой VCL следующим образом:

  TAlignment = (taLeftJustify, taRightJustify, taCenter);

  Поэтому каждый из элементов меню содержит свойство Tag, значения
  которого представляют одно из значений TAlignment: 0, 1, 2. }

  { Если для пункта меню вызывается обработчик события, то пункт становится
    выбранным и для текста Memo устанавливается
    соответствующее выравнивание. }
  if Sender is TMenuItem then
  begin
    TMenuItem(Sender).Checked := true;
    memMainMemo.Alignment := TAlignment(TMenuItem(Sender).Tag);
  end
  { Если обработчик события вызван кнопкой TToolButton главной формы,
    устанавливается требуемое выравнивание текста и соответствующий
    элемент TMenuItem помечается как выбранный. }
  else if Sender is TToolButton then
  begin
    memMainMemo.Alignment := TAlignment(TToolButton(Sender).Tag);
    case memMainMemo.Alignment of
      taLeftJustify:   mmiLeft.Checked := True;
      taRightJustify:  mmiRight.Checked := True;
```

```
      taCenter:        mmiCenter.Checked := True;
    end;
  end;
  SetButtons;
end;

procedure TMdiEditForm.mmiBoldClick(Sender: TObject);
begin
  inherited;
  if not mmiBold.Checked then
    memMainMemo.Font.Style := memMainMemo.Font.Style + [fsBold]
  else
    memMainMemo.Font.Style := memMainMemo.Font.Style - [fsBold];
  SetButtons;
end;

procedure TMdiEditForm.mmiItalicClick(Sender: TObject);
begin
  inherited;
  if not mmiItalic.Checked then
    memMainMemo.Font.Style := memMainMemo.Font.Style + [fsItalic]
  else
    memMainMemo.Font.Style := memMainMemo.Font.Style - [fsItalic];
  SetButtons;
end;

procedure TMdiEditForm.mmiUnderlineClick(Sender: TObject);
begin
  inherited;
  if not mmiUnderline.Checked then
    memMainMemo.Font.Style := memMainMemo.Font.Style + [fsUnderline]
  else
    memMainMemo.Font.Style := memMainMemo.Font.Style - [fsUnderline];
  SetButtons;
end;

procedure TMdiEditForm.mmiWordWrapClick(Sender: TObject);
begin
  inherited;
  with memMainMemo do
  begin
    WordWrap := not WordWrap;
    { Удаление полос прокрутки Memo1, если включено обтекание текста.
      В противном случае полосы прокрутки должны присутствовать. }
  if WordWrap then
      ScrollBars := ssVertical
   else
      ScrollBars := ssBoth;
    mmiWordWrap.Checked := WordWrap;
  end;
end;
```

```
procedure TMdiEditForm.mmiFontClick(Sender: TObject);
begin
  inherited;
  FontDialog.Font := memMainMemo.Font;
  if FontDialog.Execute then
    memMainMemo.Font := FontDialog.Font;
end;

{ Обработчики событий формы }
procedure TMdiEditForm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
{ Эта процедура напоминает пользователю о необходимости сохранить содержимое
  компонента TMemo, если оно изменилось со времени последнего сохранения. }
const
  CloseMsg = '''%s'' has been modified, Save?';
var
  MsgVal: integer;
  FileName: string;
begin
  inherited;
  FileName := Caption;
  if memMainMemo.Modified then
  begin
    MsgVal := MessageDlg(Format(CloseMsg, [FileName]),
      mtConfirmation, mbYesNoCancel, 0);
    case MsgVal of
      mrYes:    mmiSaveClick(Self);
      mrCancel: CanClose := false;
    end;
  end;
end;

procedure TMdiEditForm.OpenFile(FileName: string);
begin
  memMainMemo.Lines.LoadFromFile(FileName);
  Caption := FileName;
end;

procedure TMdiEditForm.SetButtons;
{ Эта процедура обеспечивает корректное отображение различных установок
  текста memo на соответствующих кнопках и пунктах меню. }
begin
  mmiBold.Checked := fsBold in memMainMemo.Font.Style;
  mmiItalic.Checked := fsItalic in memMainMemo.Font.Style;
  mmiUnderLine.Checked := fsUnderline in memMainMemo.Font.Style;

  tbBold.Down      := mmiBold.Checked;
  tbItalic.Down    := mmiItalic.Checked;
  tbUnderline.Down := mmiUnderLine.Checked;
  tbLAlign.Down    := mmiLeft.Checked;
```

```
  tbRAlign.Down    := mmiRight.Checked;
  tbCAlign.Down    := mmiCenter.Checked;
end;

procedure TMdiEditForm.mmiPrintClick(Sender: TObject);
var
  i: integer;
  PText: TextFile;
begin
  inherited;
  if PrintDialog.Execute then
  begin
    AssignPrn(PText);
    Rewrite(PText);
    try
      Printer.Canvas.Font := memMainMemo.Font;
      for i := 0 to memMainMemo.Lines.Count -1 do
        writeln(PText, memMainMemo.Lines[i]);
    finally
      CloseFile(PText);
    end;
  end;
  end;

end.
```

Многие методы формы `TMdiEditForm` являются обработчиками событий различных пунктов меню этой формы, б*о*льшая часть которых унаследована от формы `TMDIChildForm`. Обратите внимание на добавленные в меню дополнительные пункты, касающиеся специфических функций формы `TMdiEditForm`.

Обратите также внимание на то, что пункты меню File⇨New, File⇨Open, File⇨Close и File⇨Exit не имеют обработчиков событий, поскольку они связаны с обработчиками событий базовой формы `TMDIChildForm`.

Обработчики событий пунктов меню Edit представляют собой методы, взаимодействующие с объектом `TMemo` и состоящие из одной строки. Например, обработчики событий `mmiCutClick()`, `mmiCopyClick()` и `mmiPasteClick()` взаимодействуют с буфером обмена Windows для выполнения операций вырезания, копирования и вставки. Остальные обработчики событий меню Edit выполняют различные редактирующие функции, реализуемые в компоненте `Memo`, и имеют дело с удалением, очисткой и выделением текста.

Меню Character содержит различные атрибуты форматирования текста.

Данные установки выравнивания текста хранятся в свойстве `Tag` компонента `TToolButton`. Это свойство имеет перечислимый тип `TAlignment`. Для установки соответствующего выравнивания текста в компоненте `Memo` извлекается значение свойства `Tag` компонента `TToolButton`, для которого был вызван обработчик события.

Все пункты меню и кнопки панели инструментов, устанавливающие выравнивание текста, связаны с обработчиком события `CharAlignClick()`. Вот почему нужно проверить, какой именно компонент — `TMenuItem` или `TToolButton` — вызвал событие, и выполнить соответствующие действия в обработчике события.

Обработчик события CharAlignClick() вызывает метод SetButton(), устанавливающий состояние различных пунктов меню и компонентов кнопок исходя из значений атрибутов компонента TMemo.

Обработчик события mmiWordWrapClick() просто переключает состояние атрибута сворачивания текста в TMemo и состояние свойства Checked для данного пункта меню. Основываясь на этих установках, этот метод также определяет, сколько полос прокрутки будет у компонента TMemo — две или одна.

Обработчик события mmiFontClick() вызывает диалоговое окно TFontDialog и применяет выбранный шрифт к тексту TMemo. Перед запуском компонента FontDialog на выполнение свойству Font назначается текущий шрифт компонента TMemo, так что в раскрывшемся диалоговом окне отображается реально используемый шрифт.

Обработчик события mmiSaveAsClick() вызывает диалоговое окно TSaveDialog, чтобы получить от пользователя имя файла, используемое для сохранения содержимого текстового редактора. После сохранения файла свойству MdiEditForm.Caption назначается значение нового имени файла.

Обработчик события mmiSaveClick() вызывает обработчик события mmiSaveAsClick(), если требуемое имя файла не было указано. Это происходит, если пользователь создает новый файл, вместо того чтобы открыть существующий. В противном случае содержимое объекта редактора сохраняется в существующем файле под именем, которое определяется значением свойства MdiEditForm.Caption. Обратите внимание на то, что обработчик mmiSaveClick() также присваивает значение False свойству memMainMemo.Modified. Свойство Modified автоматически получает значение True, как только пользователь изменяет содержимое компонента TMemo. Однако значение False не устанавливается автоматически при сохранении содержимого текстового редактора.

Метод FormCloseQuery() — это обработчик события OnCloseQuery. Этот обработчик вычисляет значение свойства memMainMemo.Modified, когда пользователь пытается закрыть форму. Если компонент TMemo был модифицирован, пользователь уведомляется об этом и ему предлагается сохранить содержимое объекта текстового редактора.

Открытый метод TMdiEditForm.OpenFile() загружает файл, определенный параметром FileName, и помещает его содержимое в свойство memMainMemo.Lines, а также устанавливает значение свойства заголовка равным имени файла.

Все вышеизложенное исчерпывает функциональные возможности формы TMdiEditForm. Остальные формы имеют похожую функциональность.

Форма RTF-редактора

Редактор файлов формата Rich Text File (RTF) позволяет пользователю загружать и редактировать файлы, созданные в этом формате. Его форма TMdiRtfForm является производной от формы TMDIChildForm. Она содержит выровненный по клиентской области компонент TRichEdit.

Формы TMdiRtfForm и TMdiEditForm практически идентичны, за исключением того, что форма TMdiRtfForm включает в качестве элемента редактирования компонент TRichEdit, а форма TMdiEditForm для этого использует компонент TMemo. Форма TMdiRtfForm отличается от текстового редактора тем, что текстовые атрибуты, установленные для компонента TRichEdit, относятся к абзацам или выделенному тексту, тогда как в компоненте TMemo они применяются ко всему тексту.

Исходный текст формы TMdiRtfForm приведен в листинге 16.3.

Листинг 16.3. Модуль `MdiRtfFrm.pas` — определение формы `TMdiRtfForm`

```
unit MdiRtfFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs,  MdiChildFrm, StdCtrls, ComCtrls,
  ExtCtrls, Buttons, Menus, ToolWin, ImgList;

type

  TMdiRtfForm = class(TMDIChildForm)
    reMain: TRichEdit;
    FontDialog: TFontDialog;
    SaveDialog: TSaveDialog;
    mmiEdit: TMenuItem;
    mmiSelectAll: TMenuItem;
    N7: TMenuItem;
    mmiPaste: TMenuItem;
    mmiCopy: TMenuItem;
    mmiCut: TMenuItem;
    mmiCharacter: TMenuItem;
    mmiFont: TMenuItem;
    N8: TMenuItem;
    mmiWordWrap: TMenuItem;
    N9: TMenuItem;
    mmiCenter: TMenuItem;
    mmiRight: TMenuItem;
    mmiLeft: TMenuItem;
    mmiUndo: TMenuItem;
    mmiDelete: TMenuItem;
    N4: TMenuItem;
    mmiBold: TMenuItem;
    mmiItalic: TMenuItem;
    mmiUnderline: TMenuItem;

    { Обработчики событий меню File }
    procedure mmiSaveClick(Sender: TObject);
    procedure mmiSaveAsClick(Sender: TObject);

    { Обработчики событий меню Edit }
    procedure mmiCutClick(Sender: TObject);
    procedure mmiCopyClick(Sender: TObject);
    procedure mmiPasteClick(Sender: TObject);
    procedure mmiDeleteClick(Sender: TObject);
    procedure mmiUndoClick(Sender: TObject);
    procedure mmiSelectAllClick(Sender: TObject);
```

```
    { Обработчики событий меню Character }
    procedure CharAlignClick(Sender: TObject);
    procedure mmiBoldClick(Sender: TObject);
    procedure mmiItalicClick(Sender: TObject);
    procedure mmiUnderlineClick(Sender: TObject);
    procedure mmiWordWrapClick(Sender: TObject);
    procedure mmiFontClick(Sender: TObject);

    { Обработчики событий формы }
    procedure FormShow(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure reMainSelectionChange(Sender: TObject);
    procedure mmiPrintClick(Sender: TObject);
  public
    { Функции, определенные пользователем }
    procedure OpenFile(FileName: String);
    function GetCurrentText: TTextAttributes;
    procedure SetButtons;
  end;

var
  MdiRtfForm: TMdiRtfForm;

implementation
{$R *.DFM}

{ Обработчики событий меню File }

procedure TMdiRtfForm.mmiSaveClick(Sender: TObject);
begin
  inherited;
  reMain.Lines.SaveToFile(Caption);
end;

procedure TMdiRtfForm.mmiSaveAsClick(Sender: TObject);
begin
  inherited;
  SaveDialog.FileName := Caption;
  if SaveDialog.Execute then
  begin
    Caption := SaveDialog.FileName;
    mmiSaveClick(Sender);
  end;
end;

{ Обработчики событий меню Edit }

procedure TMdiRtfForm.mmiCutClick(Sender: TObject);
begin
  inherited;
```

```
  reMain.CutToClipBoard;
end;

procedure TMdiRtfForm.mmiCopyClick(Sender: TObject);
begin
  inherited;
  reMain.CopyToClipBoard;
end;

procedure TMdiRtfForm.mmiPasteClick(Sender: TObject);
begin
  inherited;
  reMain.PasteFromClipBoard;
end;

procedure TMdiRtfForm.mmiDeleteClick(Sender: TObject);
begin
  inherited;
  reMain.ClearSelection;
end;

procedure TMdiRtfForm.mmiUndoClick(Sender: TObject);
begin
  inherited;
  reMain.Perform(EM_UNDO, 0, 0);
end;

procedure TMdiRtfForm.mmiSelectAllClick(Sender: TObject);
begin
  inherited;
  reMain.SelectAll;
end;

{ Обработчики событий меню Character }

procedure TMdiRtfForm.CharAlignClick(Sender: TObject);
begin
  inherited;
  mmiLeft.Checked := false;
  mmiRight.Checked := false;
  mmiCenter.Checked := false;

  { Если TMenuItem вызывает этот обработчик события, его свойству checked
    присваивается значение true, а текущему абзацу RichEdit1
    устанавливается соответствующий атрибут. }

  if Sender is TMenuItem then
  begin
    TMenuItem(Sender).Checked := true;
    with reMain.Paragraph do
```

```
      if mmiLeft.Checked then
        Alignment := taLeftJustify
      else if mmiRight.Checked then
        Alignment := taRightJustify
      else if mmiCenter.Checked then
        Alignment := taCenter;
  end
  { Если обработчик события вызван кнопкой панели инструментов главной
    формы, необходимо установить требуемый атрибут текущему абзацу
    reMain и пунктам меню выравнивания. }
  else if Sender is TSpeedButton then
  begin
    reMain.Paragraph.Alignment :=
       TAlignment(TSpeedButton(Sender).Tag);
    case reMain.Paragraph.Alignment of
      taLeftJustify: mmiLeft.Checked := True;
      taRightJustify: mmiRight.Checked := True;
      taCenter: mmiCenter.Checked := True;
    end;
  end;
  SetButtons;
end;

procedure TMdiRtfForm.mmiBoldClick(Sender: TObject);
begin
  inherited;
  if not mmiBold.Checked then
    GetCurrentText.Style := GetCurrentText.Style + [fsBold]
  else
    GetCurrentText.Style := GetCurrentText.Style - [fsBold];
end;

procedure TMdiRtfForm.mmiItalicClick(Sender: TObject);
begin
  inherited;
  if not mmiItalic.Checked then
    GetCurrentText.Style := GetCurrentText.Style + [fsItalic]
  else
    GetCurrentText.Style := GetCurrentText.Style - [fsItalic];
end;

procedure TMdiRtfForm.mmiUnderlineClick(Sender: TObject);
begin
  inherited;
  if not mmiUnderline.Checked then
    GetCurrentText.Style := GetCurrentText.Style + [fsUnderline]
  else
    GetCurrentText.Style := GetCurrentText.Style - [fsUnderline];
end;
```

```
procedure TMdiRtfForm.mmiWordWrapClick(Sender: TObject);
begin
  inherited;
  with reMain do
  begin
    { Удаление полос прокрутки в Memo1, если включено сворачивание
      текста. В противном случае они должны присутствовать. }
    WordWrap := not WordWrap;
    if WordWrap then
      ScrollBars := ssVertical
    else
      ScrollBars := ssNone;
    mmiWordWrap.Checked := WordWrap;
  end;
end;

procedure TMdiRtfForm.mmiFontClick(Sender: TObject);
begin
  inherited;
  FontDialog.Font.Assign(reMain.SelAttributes);
  if FontDialog.Execute then
    GetCurrentText.Assign(FontDialog.Font);
  reMain.SetFocus;
end;

{ Обработчики событий формы }

procedure TMdiRtfForm.FormShow(Sender: TObject);
begin
  inherited;
  reMainSelectionChange(nil);
end;

procedure TMdiRtfForm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
{ Эта процедура напоминает пользователю о необходимости сохранить содержимое
  reMain, если оно изменилось со времени последнего сохранения. }
const
  CloseMsg = '''%s'' has been modified, Save?';
var
  MsgVal: integer;
  FileName: string;
begin
  inherited;
  FileName := Caption;
  if reMain.Modified then
  begin
    MsgVal := MessageDlg(Format(CloseMsg, [FileName]),
      mtConfirmation, mbYesNoCancel, 0);
case MsgVal of
      mrYes: mmiSaveClick(Self);
```

```
      mrCancel: CanClose := false;
    end;
  end;
end;

procedure TMdiRtfForm.reMainSelectionChange(Sender: TObject);
begin
  inherited;
  SetButtons;
end;

procedure TMdiRtfForm.OpenFile(FileName: String);
begin
  reMain.Lines.LoadFromFile(FileName);
  Caption := FileName;
end;

function TMdiRtfForm.GetCurrentText: TTextAttributes;
{ Эта процедура возвращает текстовые атрибуты текущего абзаца
  или выделенного текста reMain.}
begin
  if reMain.SelLength > 0 then
    Result := reMain.SelAttributes
else
    Result := reMain.DefAttributes;
end;

procedure TMdiRtfForm.SetButtons;
{ Эта процедура обеспечивает корректное отображение различных атрибутов
  абзаца в элементах управления формы, для чего просматриваются текущие
  атрибуты абзаца и настраиваются соответствующие элементы управления. }
begin

  with reMain.Paragraph do
  begin
    mmiLeft.Checked := Alignment = taLeftJustify;
    mmiRight.Checked := Alignment = taRightJustify;
    mmiCenter.Checked := Alignment = taCenter;
  end;

  with reMain.SelAttributes do
  begin
    mmiBold.Checked := fsBold in Style;
    mmiItalic.Checked := fsItalic in Style;
    mmiUnderline.Checked := fsUnderline in Style;
  end;
  mmiWordWrap.Checked := reMain.WordWrap;

  tbBold.Down      := mmiBold.Checked;
  tbItalic.Down    := mmiItalic.Checked;
  tbUnderline.Down := mmiUnderline.Checked;
```

```
  tbLAlign.Down    := mmiLeft.Checked;
  tbRAlign.Down    := mmiRight.Checked;
  tbCAlign.Down    := mmiCenter.Checked;
end;

procedure TMdiRtfForm.mmiPrintClick(Sender: TObject);
begin
  inherited;
  reMain.Print(Caption)
end;

end.
```

Как и в случае формы `TMdiEditForm`, большинство методов формы `TMdiRtfForm` представляют собой обработчики событий для различных кнопок панели инструментов и пунктов меню. Эти обработчики событий подобны своим аналогам в форме `TMdiEditForm`.

Пункты меню File формы `TMdiRtfForm` вызывают пункты меню File базового класса `TMdiChildForm`. Как вы помните, последний является предком `TMdiRtfForm`. Обработчики событий `mmiSaveClick()` и `mmiSaveAsClick()` для сохранения содержимого объекта `reMain` вызывают метод `reMain.Lines.SaveToFile()`.

Обработчики событий меню Edit формы `TMdiRtfForm` содержат по одной строке кода и мало чем отличаются от обработчиков событий меню Edit формы `TMdiEditForm`, за исключением того, что они вызывают методы, относящиеся к объекту `reMain`. Имена этих методов такие же, как и у методов `TMemo`, выполняющих аналогичные операции.

Пункты меню Character формы `TMdiRtfForm` позволяют изменить выравнивание абзацев или выделенного текста компонента `TRichEdit` (в отличие от компонента `TMemo`, в котором подобные изменения применяются ко всему тексту). От возвращаемого значения функции `GetCurrentText()` зависит, к чему будут применены атрибуты — к абзацу или к выделенному тексту. Эта функция проверяет, выделен ли текст путем просмотра значения свойства `TRichEdit.SelLength`. Значение `0` говорит о том, что никакой текст не выделен. Свойство `TRichEdit.SelAttributes` относится к выделенному тексту компонента `TRichEdit`, а свойство `TRichEdit.DefAttributes` — к текущему абзацу компонента `TRichEdit`.

Обработчик события `mmiFontClick()` позволяет пользователю определить атрибуты шрифта абзаца. Это же относится и к выделенному тексту.

Управление сворачиванием текста в компоненте `TRichEdit` осуществляется так же, как и в компоненте текстового редактора `TMemo`.

Обработчик события `TRichEdit.OnSelectionChange` определен для того, чтобы позволить программисту добавить функциональность, которая должна быть задействована при изменении выделения текста в компоненте. Когда пользователь перемещает позицию вставки в компоненте `TRichEdit`, значение свойства `SelStart` изменяется. Это приводит к вызову обработчика события `OnSelectionChange` и выполнению кода для изменения статуса различных компонентов `TMenuItem` и `TSpeedButton` главной формы, для того чтобы те отражали атрибуты текста во время его прокрутки в компоненте `TRichEdit`. Это необходимо, поскольку, в отличие от атрибутов текста компонента `TMemo`, атрибуты различных частей текста в компоненте `TRichEdit` могут быть разными.

Таким образом, функциональные возможности форм RTF-редактора и простого текстового редактора практически совпадают. Главное отличие между ними заключается в том, что RTF-редактор позволяет пользователю изменять атрибуты отдельных абзацев или выделенного текста, а текстовый редактор такой возможности не предоставляет.

Программа просмотра растровых изображений — третья дочерняя MDI-форма

Программа просмотра растровых изображений позволяет загружать и просматривать растровые изображения Windows. Подобно двум другим дочерним MDI-формам, форма просмотра изображений `TMdiBmpForm` также произведена от базового класса `TMDIChildForm`. Форма содержит выровненный по размерам клиентской области компонент `TImage`.

Форма `TMdiBmpForm` содержит лишь унаследованный компонент `TMainMenu`. В листинге 16.4 приведен исходный текст формы `TMdiBmpForm`.

Листинг 16.4. Модуль `MdiBmpFrm.pas` — определение формы `TMdiBmpForm`

```
unit MdiBmpFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  MdiChildFrm, ExtCtrls, Menus, Buttons, ComCtrls, ToolWin, ImgList;

type
  TMdiBMPForm = class(TMDIChildForm)
    mmiEdit: TMenuItem;
    mmiCopy: TMenuItem;
    mmiPaste: TMenuItem;
    imgMain: TImage;
    procedure mmiCopyClick(Sender: TObject);
    procedure mmiPasteClick(Sender: TObject);
    procedure mmiPrintClick(Sender: TObject);
  public
    procedure OpenFile(FileName: string);
  end;

var
  MdiBMPForm: TMdiBMPForm;

implementation

uses ClipBrd, Printers;

{$R *.DFM}

procedure TMdiBMPForm.OpenFile(FileName: String);
begin
  imgMain.Picture.LoadFromFile(FileName);
  Caption := FileName;
end;

procedure TMdiBMPForm.mmiCopyClick(Sender: TObject);
```

```
begin
  inherited;
  ClipBoard.Assign(imgMain.Picture);
end;

procedure TMdiBMPForm.mmiPasteClick(Sender: TObject);
{ Этот метод вставляет содержимое буфера обмена в imgMain }
begin
  inherited;
  { Копирование содержимого буфера обмена в imgMain }
  imgMain.Picture.Assign(ClipBoard);
  ClientWidth := imgMain.Picture.Width;
  { Установка ширины клиентской области для настройки полос прокрутки }
  VertScrollBar.Range := imgMain.Picture.Height;
  HorzScrollBar.Range := imgMain.Picture.Width;
end;

procedure TMdiBMPForm.mmiPrintClick(Sender: TObject);
begin
  inherited;

  with ImgMain.Picture.Bitmap do
  begin
    Printer.BeginDoc;
    Printer.Canvas.StretchDraw(Canvas.ClipRect, imgMain.Picture.Bitmap);
    Printer.EndDoc;
  end; { with }
end;

end.
```

В исходном коде формы `TMdiBmpForm` не так много текста, как в двух предыдущих формах. Команды меню File формы `TMdiBmpForm` вызывают обработчики событий `TMDIChildForm` точно так же, как это делается в формах `TMdiEditForm` и `TMdiRtfForm`. Команды меню Edit копируют изображение в буфер обмена Windows и вставляют его из буфера. Перед вызовом метода `TImage.Picture.Assign()` для присвоения данных буфера обмена компоненту `TImage` последний распознает форматы `CF_BITMAP` и `CF_PICTURE` как форматы растровых изображений.

Буфер обмена Windows

Буфер обмена предоставляет собой наиболее простой способ обмена информацией между двумя приложениями. Это не что иное, как глобальный блок памяти, доступ к которому предоставляется системой Windows любому приложению с помощью специального набора функций.

Буфер обмена поддерживает несколько стандартных форматов, таких как текст, OEM-текст, растровые изображения и метафайлы, а также некоторые специализированные форматы. Кроме того, можно расширить возможности буфера обмена с целью использования специальных форматов конкретного приложения.

Delphi 5 инкапсулирует буфер обмена Windows в глобальной переменной `Clipboard` типа `TClipBoard`, что упрощает его применение. Использование класса `TClipBoard` подробно освещено в главе 17, "Перенос информации с помощью буфера обмена".

Главная форма

Пользователь начинает работу в главной форме, создавая в ней дочерние MDI-формы и переключаясь между ними. Самое подходящее имя для этой формы — `MainForm`. Она служит родительской формой для дочерних форм текстового редактора, просмотра изображений и редактора RTF.

В отличие от остальных форм, описанных выше в этой главе, форма `TMainForm` не является производной от класса `TMDIChildForm`. Значение свойства `FormStyle` формы `TMainForm` равно `fsMDIForm`, в то время как остальные три формы унаследовали от формы `TMDIChild` в этом свойстве значение `fsMDIChild`. Форма `TMainForm` содержит компоненты `TMainMenu` и `TOpenDialog`, а также панель инструментов с одной кнопкой. Исходный текст формы `TMainForm` приведен в листинге 16.5.

Листинг 16.5. Модуль `MdiMainForm.pas` — определение формы `TMainForm`

```
unit MainFrm;

interface

uses
  WinTypes, WinProcs, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Messages, Dialogs, SysUtils, ComCtrls,
  ToolWin, ExtCtrls, Buttons, ImgList;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    OpenDialog: TOpenDialog;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    N3: TMenuItem;
    mmiOpen: TMenuItem;
    mmiNew: TMenuItem;
    mmiWindow: TMenuItem;
    mmiArrangeIcons: TMenuItem;
    mmiCascade: TMenuItem;
    mmiTile: TMenuItem;
    mmiCloseAll: TMenuItem;
    tlbMain: TToolBar;
    ilMain: TImageList;
    tbFileOpen: TToolButton;

    { Обработчики событий меню File }
    procedure mmiNewClick(Sender: TObject);
    procedure mmiOpenClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);

    { Обработчики событий меню Window }
    procedure mmiTileClick(Sender: TObject);
    procedure mmiArrangeIconsClick(Sender: TObject);
```

```
    procedure mmiCascadeClick(Sender: TObject);
    procedure mmiCloseAllClick(Sender: TObject);
  public
    { Методы, определенные пользователем }
    procedure OpenTextFile(EditForm: TForm; Filename: string);
    procedure OpenBMPFile(FileName: String);
    procedure OpenRTFFile(RTFForm: TForm; FileName: string);
  end;

var
  MainForm: TMainForm;

implementation
uses MDIBmpFrm, MdiEditFrm, MdiRtfFrm, FTypForm;

const
  { Определение констант для представления расширений имен файлов }
  BMPExt      = '.BMP';  // Файл растрового изображения Windows
  TextExt     = '.TXT';  // Текстовый файл
  RTFExt      = '.RTF';  // Файл в формате Rich Text

{$R *.DFM}

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  { Определяем тип файла, который хочет открыть пользователь, с
    помощью вызова функции GetFileType. Выбор подходящего метода
    основан на полученном типе файла. }
  case GetFileType of
    mrTXT: OpenTextFile(nil, ''); // Открывает текстовый файл
    mrRTF: OpenRTFFile(nil, '');  // Открывает RTF-файл
    mrBMP:
     begin
       { Установка (по умолчанию) фильтра BMP-файлов для диалогового
окна OpenDialog1. }
        OpenDialog.FilterIndex := 2;
        mmiOpenClick(nil);
     end;
  end;
end;

procedure TMainForm.mmiOpenClick(Sender: TObject);
var
  Ext: string[4];
begin
  { Вызов подходящего метода на основе типа файла,
    выбранного в окне OpenDialog1. }
  if OpenDialog.Execute then
  begin
    { Получает расширение файла и определяет тип открываемого файла.
```

```
      Далее вызывается подходящий метод с передачей ему имени файла. }
    Ext := ExtractFileExt(OpenDialog.FileName);
    if CompareStr(UpperCase(Ext), TextExt) = 0 then
      OpenTextFile(ActiveMDIChild, OpenDialog.FileName)
    else if CompareStr(UpperCase(Ext), BMPExt) = 0 then
      OpenBMPFile(OpenDialog.FileName)
    else if CompareStr(UpperCase(Ext), RTFExt) = 0 then
      OpenRTFFile(ActiveMDIChild, OpenDialog.FileName);
  end;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

{ Обработчики событий меню Window }

procedure TMainForm.mmiTileClick(Sender: TObject);
begin
  Tile;
end;

procedure TMainForm.mmiArrangeIconsClick(Sender: TObject);
begin
   ArrangeIcons;
end;

procedure TMainForm.mmiCascadeClick(Sender: TObject);
begin
  Cascade;
end;

procedure TMainForm.mmiCloseAllClick(Sender: TObject);
var
  i: integer;
begin
  { Закрывает все формы в порядке, обратном порядку их
    перечисления в свойстве MDIChildren. }
  for i := MdiChildCount - 1 downto 0  do
    MDIChildren[i].Close;
end;

{ Методы, определенные пользователем }
procedure TMainForm.OpenTextFile(EditForm: TForm; FileName: string);
begin
  { Если EditForm имеет тип TEditForm, пользователь может загрузить
    содержимое файла в эту форму. В противном случае создается новый
    экземпляр TEditForm,и файл загружается в него. }
  if (EditForm <> nil) and (EditForm is TMdiEditForm) then
```

```
    if MessageDlg('Load file into current form?', mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then
    begin
      TMdiEditForm(EditForm).OpenFile(FileName);
       Exit;
    end;
  { Создает новый экземпляр TEditForm и вызывает его метод OpenFile().}
  with TMdiEditForm.Create(self) do
    if FileName <> '' then
      OpenFile(FileName)
end;

procedure TMainForm.OpenRTFFile(RTFForm: TForm; FileName: string);
begin
  { Если RTFForm имеет тип TRTFForm, пользователь может загрузить содержимое
    файла в эту форму. В противном случае создается новый экземпляр
    TRTFForm,и файл загружается в него. }
  if (RTFForm <> nil) and (RTFForm is TMdiRTFForm) then
    if MessageDlg('Load file into current form?', mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then begin
        (RTFForm as TMdiRTFForm).OpenFile(FileName);
      Exit;
    end;
    { Создает новый экземпляр TRTFForm и вызывает его метод OpenFile().}
    with TMdiRTFForm.Create(self) do
      if FileName <> '' then
        OpenFile(FileName);
end;

procedure TMainForm.OpenBMPFile(FileName: String);
begin
  { Создает новый экземпляр TBMPForm и загружает в него BMP-файл. }
  with TMdiBmpForm.Create(self) do
    OpenFile(FileName);
end;

 end.
```

Форма `TMainForm` использует другую форму — `FileTypeForm` типа `TFileTypeForm`. В листинге 16.6 приведен исходный код этой формы.

Листинг 16.6. Модуль `FTYPFORM.PAS` — определение формы `TFileTypeForm`

```
unit FTypForm;

interface

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

```
const
  mrTXT = mrAll+1;
  mrBMP = mrAll+2;
  mrRTF = mrAll+3;

type

  TFileTypeForm = class(TForm)
    rgFormType: TRadioGroup;
    btnOK: TButton;
    procedure btnOkClick(Sender: TObject);
  end;

var
  FileTypeForm: TFileTypeForm;

function GetFileType: Integer;

implementation

function GetFileType: Integer;
{ Эта функция возвращает тип файла, выбранного пользователем
  в виде одной из ранее определенных констант. }
begin
  FileTypeForm := TFileTypeForm.Create(Application);
  try
    Result := FileTypeForm.ShowModal;
  finally
    FileTypeForm.Free;
  end;
end;

{$R *.DFM}

procedure TFileTypeForm.btnOkClick(Sender: TObject);
begin
  { Возвращает модальный результат, исходя из выбранного типа файла. }
  case rgFormType.ItemIndex of
   0: ModalResult := mrTXT;
   1: ModalResult := mrRTF;
   2: ModalResult := mrBMP;
  end;
end;

end.
```

Форма `TFileTypeForm` используется для запроса типа создаваемого файла у пользователя. Эта форма возвращает значение `ModalResult` исходя из состояния переключателя `TRadioButton`, определяющего выбранный пользователем тип файла. Функция `GetFileType()`

заботится о создании, отображении и освобождении экземпляра TFileTypeForm. Функция возвращает значение свойства TFileForm.ModalResult. Эта форма не создается автоматически и исключена из списка автоматически создаваемых форм проекта.

Панель инструментов компонента TMainForm содержит только одну кнопку, используемую для открытия начальной дочерней формы. Когда дочерняя форма становится активной, ее панель инструментов заменяет панель инструментов главной формы. Эта операция осуществляется в обработчике события OnActivate дочерней формы. Открытые методы формы TMainForm, такие как OpenTextFile(), OpenRTFFile() и OpenBMPFile(), вызываются из обработчика события TMainForm.mmiOpenClick(), который выполняется при выборе пользователем пункта меню File⇨Open.

Методу OpenTextFile() передается два параметра: экземпляр объекта TForm и имя файла. Экземпляр объекта TForm представляет текущую активную форму приложения. Он передается в метод OpenTextFile() для того, чтобы тот определил, является ли переданный TForm классом TMdiEditForm. Если ответ положительный, вы сможете открыть текстовый файл в существующей форме TMdiEditForm, а не создавать новый экземпляр TMdiEditForm. Если в OpenTextFile() передается экземпляр TMdiEditForm, пользователю предлагается выбор — открывать или нет текстовый файл в существующей форме. Если пользователь не соглашается или значение переданного параметра TMdiEditForm равно nil, создается новый экземпляр TMdiEditForm.

Метод OpenRTFFile() действует так же, как и OpenTextFile(), за исключением того, что он проверяет принадлежность текущей активной формы классу TRTFForm. Остальная функциональность остается той же.

Метод OpenBmpFile() всегда "подразумевает", что пользователь открывает новый файл. Дело в том, что TMdiBmpForm предназначен только для просмотра, а не для редактирования растровых изображений. Если бы форма позволяла пользователю редактировать изображение, метод OpenBMPFile() действовал бы подобно двум предыдущим методам.

Обработчик события mmiNewClick() вызывает функцию GetFileType() для получения от пользователя типа файла. Затем, основываясь на полученном значении, он вызывает подходящий метод OpenXXXFile(). Если файл имеет формат BMP, свойство OpenDialog.Filter получает значение по умолчанию, соответствующее формату BMP, и затем вызывается метод mmiOpenClick(), так как пользователь не создает новый .bmp-файл, а просматривает существующий.

Обработчик события mmiOpenClick() открывает диалоговое окно OpenDialog и вызывает подходящий метод OpenXXXFile(). Обратите внимание на то, что первым параметром, передаваемым в методы OpenTextFile() и OpenRTFFile(), является значение свойства TMainForm.ActiveMDIChild. Свойство ActiveMDIChild представляет дочернюю форму, получившую фокус в данный момент. Напомним, что оба метода определяют, не желает ли пользователь открыть файл в уже существующей дочерней MDI-форме. Если ни одна из форм не активна, ActiveMDIChild имеет значение nil. Если же при вызове OpenTextFile() свойство ActiveMDIChild указывает на TMdiRTFForm, метод это распознает и действует по-прежнему корректно благодаря следующему выражению:

```
if (RTFForm <> nil) and (RTFForm is TMdiRTFForm) then
```

Это выражение определяет, указывает ли ActiveMDIChild на TMdiRTFForm. Если нет, то создается новая форма.

Обработчик события mmiExitClick() вызывает метод TMainForm.Close, который не только закрывает главную форму, но и прекращает выполнение приложения. Если во время вызова этого обработчика события открыты дочерние формы, то они также закрываются и уничтожаются.

Обработчики событий меню Window — это методы “в одну строку”, управляющие расположением дочерних MDI-окон в клиентской области главной формы. На рис. 16.2 и 16.3 показаны формы, упорядоченные мозаикой и каскадом соответственно.

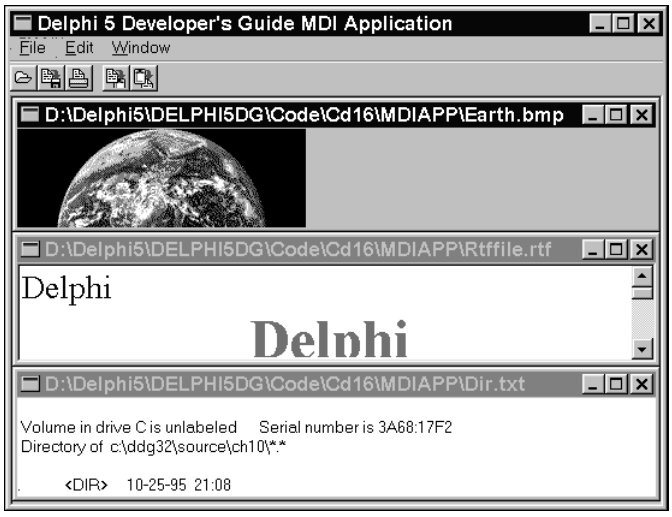


Рис. 16.2. Дочерние формы расположены мозаикой

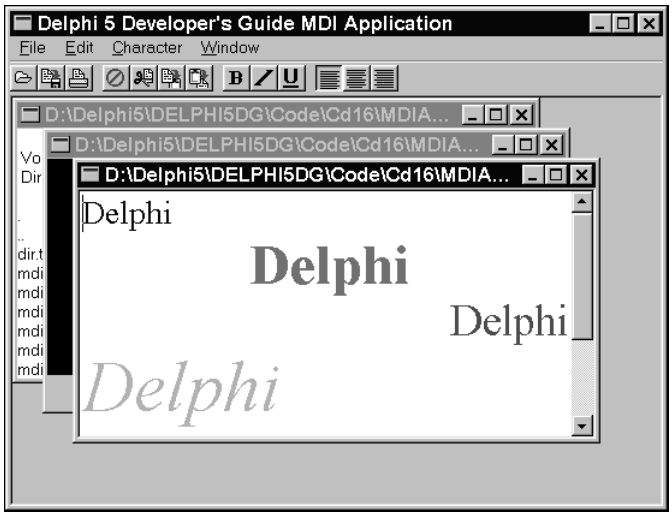


Рис. 16.3. Дочерние формы расположены каскадом

Метод `mmiArrangeIconsClick()` упорядочивает пиктограммы клиентской области главной формы, чтобы те были равномерно распределены и не “налезали” друг на друга.

Обработчик события `mmiCloseAllClick()` закрывает все открытые дочерние MDI-формы. Цикл закрытия дочерних окон перебирает их в порядке, обратном тому, в котором они перечислены в свойстве-массиве `MDIChildren`. Напомним, что `MDIChildren` представляет собой массив, начинающийся с элемента с нулевым индексом и представляющий активные дочерние MDI-формы приложения. Свойство `MDIChildCount` содержит количество открытых дочерних окон.

На этом обсуждение функциональных возможностей примера MDI-приложения можно завершить. Следующие разделы посвящены использованию разнообразных технологий и компонентов в различных формах приложения.

Работа с меню

Использование меню в MDI-приложениях не сложнее, чем в любых других типах приложений. Тем не менее существуют определенные особенности работы меню в MDI-приложениях. В следующих разделах описывается, каким образом MDI-приложение позволяет своим дочерним формам совместно использовать одну и ту же строку меню с помощью технологии *слияния меню* (merging menu). Кроме того, мы обсудим, как заставить обычные (не MDI) приложения использовать общую строку меню.

Слияние меню в MDI-приложениях

Давайте посмотрим на компонент `TMainMenu` формы `TMdiEditForm`. Двойной щелчок на его пиктограмме вызовет раскрытие окна редактора меню.

В строке меню формы `TMdiEditForm` находятся три отдельных меню — File, Edit и Character. Каждое из них обладает свойством `GroupIndex`, отображаемым в окне Object Inspector по щелчку на объекте меню в редакторе меню. Заметьте, что свойство `GroupIndex` меню File имеет значение `0`, а то же свойство меню Edit и Character — значения `1`.

Строка меню формы `TMainForm` содержит два элемента — меню File и Window. Как и в случае формы `TMdiEditForm`, свойство `GroupIndex` меню File формы `TMainForm` имеет значение `0`. Однако значение свойства `GroupIndex` меню Window — `9`.

Обратите внимание на то, что меню File формы `TMainForm` и меню File формы `TMdiEditForm` содержат различные команды. Меню File `TMdiEditForm` содержит больше команд, чем его аналог в форме `TMainForm`.

Свойство `GroupIndex` играет большую роль, так как именно оно управляет процессом слияния меню. Это означает, что, когда главная форма запускает дочернюю форму, меню последней сливается с меню главной формы. Свойство `GroupIndex` определяет, в каком порядке следуют отдельные меню и какие меню главной формы заменяются меню дочерней формы. Запомните, что слияние применяется лишь к элементам строки меню компонента `TMainMenu`, а не к их командам.

Если значение свойства `GroupIndex` элемента меню дочерней формы совпадает со значением свойства `GroupIndex` элемента меню главной формы, элемент меню дочерней формы заменяет собой элемент меню главной формы. Оставшиеся меню располагаются в строке в порядке, определенном значениями свойств `GroupIndex` элементов объединенного меню. Как только в окне приложения становится активной форма `MdiEditForm`, в строке меню главной формы появляются элементы меню в следующем порядке: File, Edit, Character и Window. В качестве меню File используется меню File формы `TMdiEditForm`, потому что оба этих элемента меню имеют одинаковое значение свойства `GroupIndex` — `0`. Порядок следования элементов меню отражает порядок значений их свойств `GroupIndex`: `0`, `1`, `1`, `9`.

На заметку Существуют определенные правила нумерации пунктов меню, которым необходимо следовать для обеспечения лучшей интеграции приложения с объектами OLE-контейнеров (в смысле слияния их меню). Эти правила изложены в документе “Borland Delphi Library Reference Guide”. В данном случае они не используются.

Все вышеизложенное справедливо и для других форм MDI-приложения. Как только форма становится активной, меню главной строки изменяются в результате слияния меню главной и дочерней форм. В процессе работы с приложением строка меню изменяется в зависимости от того, какая из дочерних форм активна в данный момент.

Слияние меню в MDI-приложениях выполняется автоматически. Если только значения свойств `GroupIndex` элементов меню установлены в требуемом порядке, всегда будет происходить корректное слияние элементов меню при вызове дочерних MDI-форм.

В обычных (не MDI) приложениях слияние происходит аналогичным образом, необходимо лишь выполнить некоторые дополнительные операции. В главе 4, "Строение приложения и концепции конструирования", при обсуждении класса `TNavStatForm` приводился пример слияния меню в обычных приложениях. Но в том приложении слияние было организовано для дочерних форм, в действительности являющихся дочерними окнами элемента управления, отличного от главной формы, в результате чего в них требовалось явно вызывать функции `Merge()` и `Unmerge()`. В общем случае слияние меню в обычных приложениях не является автоматическим процессом. Свойству `AutoMerge` компонента `TMainMenu` формы, меню которого должно сливаться с меню главной формы, следует установить значение `True`. Пример приложения, в котором демонстрируется слияние меню для не MDI-форм, имеется на компакт-диске, прилагаемом ко второму тому (см. также `www.williamspublishing.com`.), и представлен проектом `NonMDI.dpr`.

Добавление в меню списка открытых документов

Для добавления в меню Window списка открытых документов необходимо присвоить свойству `WindowMenu` главной формы экземпляр элемента меню, содержащего требуемый список открытых документов. Например, свойству `TMainForm.WindowMenu` нашего MDI-приложения присваивается объект `mmiWindow`, ссылающийся на меню Window в строке меню приложения. Этому свойству может присваиваться готовый элемент строки меню, а не отдельные его команды. Приложение отображает список открытых документов в меню Window.

Разнообразные MDI-технологии

В следующих разделах мы познакомимся с различными полезными технологиями, которые могут применяться в MDI-приложениях.

Отображение растрового изображения в клиентском окне MDI-формы

При разработке MDI-приложений иногда может потребоваться поместить в клиентскую область главной формы приложения некоторый фоновый рисунок, например логотип компании. В случае обычных (не MDI) форм это сделать очень просто. Достаточно поместить в

форму компонент TImage, присвоить его свойству Align значение alClient — и все (сравните с формой просмотра растровых изображений в MDI-приложении, рассмотренном выше в этой главе). Однако помещение изображения в главную форму MDI-приложения — совсем другая история.

Напомним, что клиентское окно MDI-приложения является окном, отдельным от окна главной формы. Оно отвечает за выполнение определенных действий, связанных с функционированием MDI-приложения, включая отображение дочерних MDI-окон.

Представьте себе, что главная форма — это прозрачное окно, расположенное над клиентским окном. При помещении в клиентскую область главной формы любые компоненты — например, TButton, TEdit или Timage — попадают на это прозрачное окно. Когда клиентское окно выполняет прорисовку дочерних окон (или, скорее, дочерних форм), эти дочерние формы рисуются *под* компонентами главной формы. Внешне эти компоненты будут выглядеть подобно рисункам, наклеенным на стекло, под которым расположены дочерние формы, как показано на рис. 16.4.

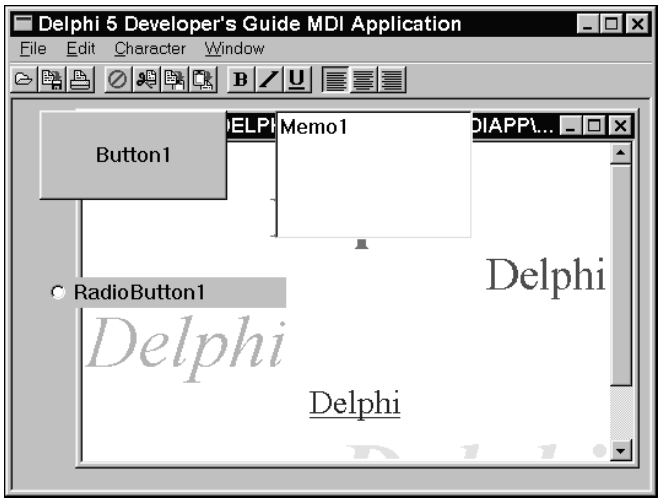


Рис. 16.4. Дочерние формы отображаются под компонентами главной формы

Итак, как же нарисовать изображение в клиентском окне? Поскольку Delphi 5 не обеспечивает инкапсуляцию клиентского окна в подпрограммах библиотеки VCL, придется обратиться к функциям Win32 API. Предлагаемый метод заключается в создании подкласса клиентского окна и перехвате сообщения, отвечающего за прорисовку его фона, — WM_ERASEBACKGND. Затем следует переопределить поведение по умолчанию и выполнить операции по его отображению.

Приведенный ниже текст относится к проекту MdiBknd.dpr, присутствующему на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.). Этот проект представляет собой MDI-приложение с компонентом TImage, содержащим растровое изображение. В меню можно определить, как именно отображать это изображение в клиентском MDI-окне: по центру, мозаикой или в растянутом виде. Этих три варианта показаны на рис. 16.5, 16.6 и 16.7 соответственно.

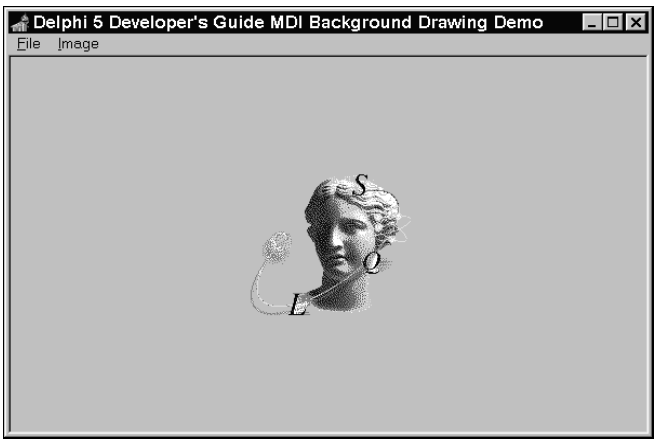


Рис. 16.5. Клиентское MDI-окно с изображением, расположенным по центру

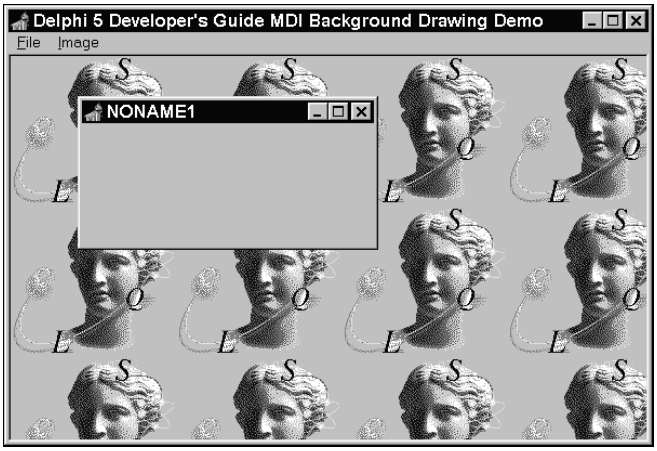


Рис. 16.6. Клиентское MDI-окно с изображением, расположенным мозаикой

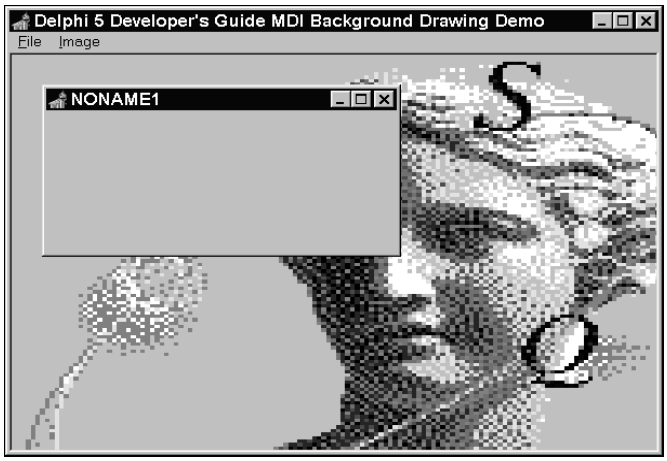


Рис. 16.7. Клиентское MDI-окно с растянутым изображением

В листинге 16.7 приведен текст модуля, в котором выполняются упомянутые операции отображения фонового изображения.

Листинг 16.7. Вывод изображений в клиентское MDI-окно

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiNew: TMenuItem;
    mmiClose: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    mmiImage: TMenuItem;
    mmiTile: TMenuItem;
    mmiCenter: TMenuItem;
    mmiStretch: TMenuItem;
    imgMain: TImage;
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiTileClick(Sender: TObject);
  private
    FOldClientProc,
    FNewClientProc: TFarProc;
    FDrawDC: hDC;
    procedure CreateMDIChild(const Name: string);
    procedure ClientWndProc(var Message: TMessage);
    procedure DrawStretched;
    procedure DrawCentered;
    procedure DrawTiled;
  protected
    procedure CreateWnd; override;
  end;

var
  MainForm: TMainForm;

implementation
```

```
uses MdiChildFrm;
{$R *.DFM}
procedure TMainForm.CreateWnd;
begin
  inherited CreateWnd;
  // Превращение метода ClientWndProc в действительную процедуру окна
  FNewClientProc := MakeObjectInstance(ClientWndProc);
  // Получение указателя начальной процедуры окна
  FOldClientProc := Pointer(GetWindowLong(ClientHandle, GWL_WNDPROC));
  // Установка в качестве новой процедуры окна процедуры ClientWndProc
  SetWindowLong(ClientHandle, GWL_WNDPROC, LongInt(FNewClientProc));
end;

procedure TMainForm.DrawCentered;
{ Эта процедура центрирует изображение в клиентской области формы. }
var
  CR: TRect;
begin
  GetWindowRect(ClientHandle, CR);
   with imgMain do
    BitBlt(FDrawDC, ((CR.Right - CR.Left) - Picture.Width) div 2,
        ((CR.Bottom - CR.Top) - Picture.Height) div 2,
        Picture.Graphic.Width, Picture.Graphic.Height,
        Picture.Bitmap.Canvas.Handle, 0, 0, SRCCOPY);
end;

procedure TMainForm.DrawStretched;
{ Эта процедура растягивает изображение так, чтобы оно занимало
  всю клиентскую область формы. }
var
  CR: TRect;
begin
  GetWindowRect(ClientHandle, CR);
  StretchBlt(FDrawDC, 0, 0, CR.Right, CR.Bottom,
    imgMain.Picture.Bitmap.Canvas.Handle, 0, 0,
    imgMain.Picture.Width, imgMain.Picture.Height, SRCCOPY);
end;

procedure TMainForm.DrawTiled;
{ Эта процедура располагает изображение мозаикой в клиентской области. }
var
  Row, Col: Integer;
  CR, IR: TRect;
  NumRows, NumCols: Integer;
begin
  GetWindowRect(ClientHandle, CR);
  IR := imgMain.ClientRect;
```

```
  NumRows := CR.Bottom div IR.Bottom;
  NumCols := CR.Right div IR.Right;
  with imgMain do
    for Row := 0 to NumRows+1 do
      for Col := 0 to NumCols+1  do
        BitBlt(FDrawDC, Col * Picture.Width, Row * Picture.Height,
           Picture.Width, Picture.Height, Picture.Bitmap.Canvas.Handle,
           0, 0, SRCCOPY);
end;

procedure TMainForm.ClientWndProc(var Message: TMessage);
begin
  case Message.Msg of
    { Прехватываем сообщения WM_ERASEBKGND и выполняем
      прорисовку клиентской области.}
    WM_ERASEBKGND:
      begin
        CallWindowProc(FOldClientProc, ClientHandle,
            Message.Msg, Message.wParam, Message.lParam);
        FDrawDC := TWMEraseBkGnd(Message).DC;
        if mmiStretch.Checked then
          DrawStretched
        else if mmiCenter.Checked then
          DrawCentered
        else DrawTiled;
        Message.Result := 1;
      end;
    { Перехватываем сообщения прокрутки и перерисовываем клиентскую
      область с помощью вызова метода InvalidateRect. }
    WM_VSCROLL, WM_HSCROLL:
      begin
        Message.Result := CallWindowProc(FOldClientProc, ClientHandle,
          Message.Msg, Message.wParam, Message.lParam);
        InvalidateRect(ClientHandle, nil, True);
      end;
    else
    { По умолчанию вызывается начальная процедура окна.}
      Message.Result := CallWindowProc(FOldClientProc, ClientHandle,
        Message.Msg, Message.wParam, Message.lParam);
  end; { case }
end;

procedure TMainForm.CreateMDIChild(const Name: string);
var
  MdiChild: TMDIChildForm;
begin
  MdiChild := TMDIChildForm.Create(Application);
```

```
  MdiChild.Caption := Name;
end;

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  CreateMDIChild('NONAME' + IntToStr(MDIChildCount + 1));
end;

procedure TMainForm.mmiCloseClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.mmiTileClick(Sender: TObject);
begin
  mmiTile.Checked := false;
  mmiCenter.Checked := False;
  mmiStretch.Checked := False;
  { Установка свойства Checked элемента меню,
    вызвавшего этот обработчик события. }
  if Sender is TMenuItem then
    TMenuItem(Sender).Checked := not TMenuItem(Sender).Checked;
  { Перерисовка клиентской области формы. }
  InvalidateRect(ClientHandle, nil, True);
end;

end.
```

Чтобы вывести изображение в клиентском окне MDI-приложения, следует использовать технологию *создания подклассов* (subclassing). Использование подклассов обсуждалось в главе 5, "Сообщения Windows". Подкласс клиентского окна должен сохранять адрес исходной процедуры этого окна (чтобы можно было ее вызвать при необходимости), а также подготовить указатель на новую процедуру окна. Принадлежащая форме переменная `FOldClienProc` хранит адрес исходной процедуры окна, а переменная `FNewClientProc` ссылается на новую процедуру окна.

Процедура `ClientWndProc()` является той процедурой, на которую указывает переменная `FNewClientProc`. В связи с тем, что процедура `ClientWndProc()` представляет собой метод формы `TMainForm`, необходимо использовать функцию `MakeObjectInstance()` для передачи системе указателя на процедуру окна, создаваемого этой функцией, — как описано в главе 13, "Дополнительный инструментарий разработчика".

Метод TMainForm.CreateWnd() был переопределен в подклассе клиентского окна главной формы с помощью функций Win32 API GetWindowLong() и SetWindowLong(). Новой процедурой окна является процедура ClientWndProc().

В форме TMainForm объявлены три закрытых метода: DrawCentered(), DrawTiled() и DrawStretched(). Каждый из них использует функции Win32 API для выполнения подпрограмм GDI, обеспечивающих прорисовку изображения. Функции Win32 API применяется потому, что контекст устройства клиентского окна не инкапсулирован объектом TCanvas, и воспользоваться встроенными средствами Delphi 5 в этом случае нельзя. В действительности можно присвоить свойству TCanvas.Handle данный контекст устройства. Конечно, для этого потребуется создать экземпляр объекта TCanvas, но, в принципе, это возможно.

Необходимо перехватывать три сообщения, которые предназначены для управления прорисовкой фона: WM_ERASEBKGND, WM_VSCROLL и WM_HSCROLL. Сообщение WM_ERASEBKGND посылается в окно, когда оно должно быть стерто. Это самый подходящий момент для выполнения специальных операций с целью отображения фоновых изображений. Какую именно процедуру отображения следует вызвать, программа определяет исходя из выбранного элемента меню. Сообщения WM_VSCROLL и WM_HSCROLL перехватываются для корректного отображения фонового изображения при прокрутке содержимого окна главной формы. Все остальные сообщения посылаются в исходную процедуру окна с помощью следующего выражения:

```
Message.Result := CallWindowProc(FOldClientProc, ClientHandle,
        Message.Msg, Message.wParam, Message.lParam);
```

Приведенный пример не только демонстрирует то, как можно улучшить внешний вид создаваемого приложения, но и методы разработки на уровне функций Win32 API, не поддерживаемых библиотекой VCL.

Создание скрытых дочерних MDI-форм

Delphi 5 возвратит сообщение об ошибке, если сделать попытку скрыть дочернюю MDI-форму с помощью следующего выражения:

```
ChildForm.Hide;
```

Эта ошибка означает, что сокрытие дочерних MDI-форм не разрешено. Причина в том, что разработчики Delphi обнаружили следующий факт: реализованный в Windows механизм сокрытия дочерних MDI-форм нарушает *порядок следования* (z-order) дочерних окон. Если, используя эту технологию, не проявить предельную осторожность, попытка скрыть дочернюю MDI-форму, скорее всего, вызовет беспорядочное изменение внешнего вида вашего приложения. Но что делать, если все же потребуется скрыть дочернюю MDI-форму? Существует два способа это сделать. Но не забывайте об упомянутой аномалии Win32 и используйте обе технологии с предельной осторожностью.

Один из способов сокрытия дочерней MDI-формы состоит в предотвращении отображения этой формы клиентским окном. Для этого следует использовать функцию Win32 API LockWindowUpdate(), отключающую отображение клиентского окна MDI-приложения. Эта технология особенно полезна, если требуется создать дочернюю MDI-форму, но не желательно показывать ее пользователю до успешного завершения определенного процесса. Например, таким процессом может быть запрос к базе данных — если он будет безрезультатным, форму можно освободить. Если не использовать сокрытие формы, на экране появится мерцание, связанное с созданием формы, которую пока нельзя уничтожить. Функция LockWindowUpdate() отключает отображение канвы данного окна. В одно и то же время мо-

жет быть заблокировано только одно окно. Вызов функции LockWindowUpdate() с параметром 0 позволяет восстановить отображение ранее заблокированной формы.

Другой способ сокрытия дочерней MDI-формы заключается в использовании функции Win32 API ShowWindow(). Для сокрытия формы необходимо передать в эту функцию флаг SW_HIDE. Затем следует воспользоваться функцией SetWindowPos() для восстановления дочернего окна. Этой технологией можно пользоваться, если дочерняя форма уже создана и отображается.

В листинге 16.8 содержится исходный текст главной формы проекта MdiHide.dpr, иллюстрирующего описанные технологии. Полный текст этого проекта имеется на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Листинг 16.8. Модуль, демонстрирующий использование технологий сокрытия дочерних MDI-форм

```
unit MainFrm;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, ComCtrls, MdiChildFrm;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiNew: TMenuItem;
    mmiClose: TMenuItem;
    mmiWindow: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    mmiHide: TMenuItem;
    mmiShow: TMenuItem;
    mmiHideForm: TMenuItem;
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiHideClick(Sender: TObject);
    procedure mmiShowClick(Sender: TObject);
    procedure mmiHideFormClick(Sender: TObject);
  private
    procedure CreateMDIChild(const Name: string);
  public
    HideForm: TMDIChildForm;
    Hidden: Boolean;
  end;

var
```

```
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.CreateMDIChild(const Name: string);
var
  MdiChild: TMDIChildForm;
begin
  MdiChild := TMDIChildForm.Create(Application);
  MdiChild.Caption := Name;
end;

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  CreateMDIChild('NONAME' + IntToStr(MDIChildCount + 1));
end;

procedure TMainForm.mmiCloseClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.mmiHideClick(Sender: TObject);
begin
  if Assigned(HideForm) then
    ShowWindow(HideForm.Handle, SW_HIDE);
  Hidden := True;
end;

procedure TMainForm.mmiShowClick(Sender: TObject);
begin
  if Assigned(HideForm) then
    SetWindowPos(HideForm.handle, HWND_TOP, 0, 0, 0, 0, SWP_NOSIZE
       or SWP_NOMOVE or SWP_SHOWWINDOW);
  Hidden := False;
end;

procedure TMainForm.mmiHideFormClick(Sender: TObject);
begin
```

```
  if not Assigned(HideForm) then
  begin
    if MessageDlg('Создать скрытую форму?', mtConfirmation,
         [mbYes, mbNo], 0) = mrYes then
    begin
       LockWindowUpdate(Handle);
      try
        HideForm := TMDIChildForm.Create(Application);
        HideForm.Caption := 'Скрытая форма';
        ShowMessage('Форма создана и скрыта. Нажмите OK, чтобы она появилась');
     finally
        LockWindowUpdate(0);
      end;
    end
    else begin
      HideForm := TMDIChildForm.Create(Application);
      HideForm.Caption := 'Скрытая форма';
    end;
  end
  else if not Hidden then
      HideForm.SetFocus;
end;

end.
```

Этот проект — простое MDI-приложение. Обработчик события `mmiHideFormClick()` создает дочернюю форму, которая может быть либо сразу создана скрытой, либо вначале показана, а затем скрыта пользователем.

Вначале метод `mmiHideFormClick()` проверяет, создан ли уже экземпляр формы `THideForm`. Если да, метод отображает этот экземпляр, отслеживая, чтобы он не был скрыт пользователем. Если такого экземпляра не существует, метод запрашивает у пользователя, нужно ли создать уже скрытую форму. При подтверждении пользователем запроса в сообщении `WM_SETREDRAW` запрещается отображение клиентского окна перед созданием формы. Если отображение клиентского окна не запрещено, созданная форма отображается. На экран выводится окно сообщения, уведомляющее о создании формы. Когда пользователь закрывает это окно, отображение клиентского окна возобновляется и дочерняя форма отображается в результате принудительной перерисовки окна. Вы можете заменить окно сообщения о созданной форме некоторым продолжительным процессом, для которого нужно, чтобы форма была создана, но не отображалась. Если пользователь выбрал вариант, при котором не нужно, чтобы созданная форма была скрытой, то форма создается обычным образом.

Второй метод используется для сокрытия созданной формы путем вызова функции Win32 API `ShowWindow()` с передачей ей в качестве параметров указателя дочерней формы и флага `SW_HIDE`. В результате изображение дочерней формы удаляется с экрана. Для повторного отображения формы вызывается функция Wn32 API `SetWindowPos()`, которой в качестве параметров передаются указатель дочерней формы и различные флаги, которые можно просмотреть в листинге. Функция `SetWindowPos()` используется для изменения

размеров, позиции и порядка следования окон. В нашем примере SetWindowPos() используется для повторного отображения скрытого окна путем установки его порядка следования. В данном случае в функцию передается флаг HWND_TOP, и она назначает скрытую форму в качестве верхнего окна.

Минимизация, максимизация и восстановление всех дочерних MDI-окон

Достаточно часто возникает необходимость выполнить определенное действие над всеми активными MDI-формами проекта сразу. Изменение значения свойства WindowState формы — типичный пример действия, выполняемого с каждым экземпляром дочерней MDI-формы. Эта задача довольно проста и требует всего лишь перебора форм в свойстве-массиве MDIChildren главной формы. Это свойство содержит активные дочерние формы. В листинге 16.9 приведен исходный текст обработчиков событий, минимизирующих, максимизирующих и восстанавливающих одновременно все дочерние MDI-окна приложения. Этот проект называется Min_Max.dpr — его можно найти на компакт-диске, прилагаемом ко второму тому (см. также www.williamspublishing.com.).

Листинг 16.9. Минимизация, максимизация и восстановление всех дочерних MDI-форм

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    MainMenu1: TMainMenu;
    mmiFile: TMenuItem;
    mmiNew: TMenuItem;
    mmiClose: TMenuItem;
    mmiWindow: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    mmiMinimizeAll: TMenuItem;
    mmiMaximizeAll: TMenuItem;
    mmiRestoreAll: TMenuItem;
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiMinimizeAllClick(Sender: TObject);
    procedure mmiMaximizeAllClick(Sender: TObject);
    procedure mmiRestoreAllClick(Sender: TObject);
  private
```

```
  { Закрытые объявления }
    procedure CreateMDIChild(const Name: string);
  public
  { Открытые объявления }
end;

var
  MainForm: TMainForm;

implementation
uses MdiChildFrm;

{$R *.DFM}

procedure TMainForm.CreateMDIChild(const Name: string);
var
  Child: TMDIChildForm;
begin
  Child := TMDIChildForm.Create(Application);
  Child.Caption := Name;
end;

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  CreateMDIChild('NONAME' + IntToStr(MDIChildCount + 1));
end;

procedure TMainForm.mmiCloseClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.mmiMinimizeAllClick(Sender: TObject);
var
  i: integer;
begin
  for i := MDIChildCount - 1 downto 0 do
    MDIChildren[i].WindowState := wsMinimized;
end;

procedure TMainForm.mmiMaximizeAllClick(Sender: TObject);
```

```
var
  i: integer;
begin
  for i := 0 to MDIChildCount - 1 do
    MDIChildren[i].WindowState := wsMaximized;
end;

procedure TMainForm.mmiRestoreAllClick(Sender: TObject);
var
  i: integer;
begin
  for i := 0 to MDIChildCount - 1 do
    MDIChildren[i].WindowState := wsNormal;
end;

end.
```

Резюме

В этой главе описан процесс создания MDI-приложений в Delphi 5. Вы также познакомились с некоторыми дополнительными технологиями разработки MDI-приложений, которых должно быть вполне достаточно для создания профессиональных MDI-приложений.

Перенос информации с помощью буфера обмена

Глава

17

Когда-то человечество боролось исключительно за выживание. Люди жили в пещерах, охотились с помощью дротиков и камней и общались между собой с помощью невнятных звуков и жестов. Они поклонялись огню, поскольку он давал им свет, при котором они работали на своих чрезвычайно медленных компьютерах. В ту пору компьютеры могли одновременно работать лишь с одним приложением из-за ограниченных возможностей аппаратных средств и программного обеспечения. Единственным способом переноса информации было сохранение файлов на дискете для использования на других машинах.

В настоящее время аппаратные средства и программное обеспечение существенно изменились к лучшему. Под управлением современных операционных систем, таких как Windows 95/98 или Windows NT/2000, можно одновременно запускать сразу несколько программ, что значительно упрощает жизнь пользователям и экономит их время. Одним из преимуществ Windows является возможность обмена информацией между приложениями, запущенными на одной машине. Буфер обмена Win32 и стандарт Dynamic Data Exchange (DDE) — наиболее ранние технологии такого обмена информацией. Использование любого из этих механизмов в приложениях позволит их пользователям с легкостью переносить информацию из одного приложения в другое.

В этой главе мы расскажем, как Delphi инкапсулирует функциональность буфера обмена Win32. В предыдущих изданиях мы освещали также технологию DDE. Однако после появления новых технологий взаимодействия процессов, например, таких как COM (которая подробно обсуждается в главе 23 второго тома, "COM-ориентированные технологии"), было бы бессмысленно отсылать вас к уже мертвым технологиям. Ну а для простой реализации переноса информации между приложениями буфер обмена по-прежнему является достаточно надежным средством.

Вначале был буфер обмена...

Если вы — опытный Windows-программист, вероятно, вам уже знаком буфер обмена Win32 — хотя бы основы его функционирования. Если же вы — новичок в программировании, но опытный пользователь Windows, то, очевидно, постоянно используя буфер обмена, вы не особенно задумываясь над тем, как он реализован на программном уровне.

Почти любое приложение, имеющее подменю Edit в своем главном меню, использует буфер обмена. Так что же все-таки представляет собой пресловутый буфер обмена? Буфер обмена — это всего лишь область памяти и соответствующий набор функций Win32 API, позволяющих приложениям хранить и получать информацию из данной области памяти. Вы можете скопировать исходный код из редактора Delphi и вставить его в любой другой редактор.

Почему же Win32 нуждается в специальном наборе функций и сообщений для использования буфера обмена? Копирование данных в буфер — нечто большее, чем просто резервирование области памяти и помещение в нее данных. Другим приложениям должно быть известно, как извлечь эти данные и в каком формате они хранятся. Win32 берет на себя заботу об управлении буфером обмена, позволяя копировать, вставлять данные и запрашивать информацию о них.

Форматы буфера обмена

Win32 поддерживает 25 предопределенных форматов данных, используемых для копирования и вставки информации из буфера обмена. Наиболее распространенными среди них являются следующие.

`CF_BITMAP`	Данные растровой графики
`CF_DIB`	Данные растровой графики с использованием палитры
`CF_PALETTE`	Палитра
`CF_TEXT`	Символьный массив, в котором каждая строка заканчивается символом возврата каретки. Это чаще всего используемый формат данных

Если вас интересуют другие, менее распространенные форматы, обратитесь к разделу "SetClipboardData" интерактивной справки Win32 API. Кроме того, Win32 допускает определение собственных пользовательских форматов, что будет проиллюстрировано ниже в этой главе.

До появления Delphi программисту необходимо было вызывать разнообразные функции буфера обмена самостоятельно, стараясь не испортить при этом содержимое буфера обмена. В Delphi для работы с буфером обмена вы можете использовать глобальную переменную `Clipboard`. Переменная `Clipboard` — это экземпляр класса Delphi, инкапсулирующий буфер обмена Win32.

Использование буфера обмена для работы с текстом

В главе 16, "MDI-приложения", уже было описано, как использовать буфера обмена для работы с текстом. Для текстового редактора MDI-приложения в меню были созданы пункты для вырезания, копирования, вставки, удаления и выделения текста.

В MDI-приложении редактор (компонент `TMemo`) занимает клиентскую область формы. Компонент `TMemo` имеет функции, взаимодействующие с глобальным объектом `Clipboard`, — `CutToClipboard()`, `CopyToClipboard()` и `PasteFromClipboard()`. Методы `ClearSelection()` и `SelectAll()` не являются необходимыми процедурами для работы с буфером обмена, но позволяют выделять текст для копирования в него. Листинг 17.1 демонстрирует обработку команд подменю Edit.

Листинг 17.1. Работа с текстовыми данными в буфере обмена

```
procedure TMdiEditForm.mmiCutClick(Sender: TObject);
begin
  inherited;
  memMainMemo.CutToClipboard;
  end;

procedure TMdiEditForm.mmiCopyClick(Sender: TObject);
  begin
  inherited;
  memMainMemo.CopyToClipboard;
  end;

procedure TMdiEditForm.mmiPasteClick(Sender: TObject);
  begin
  inherited;
  memMainMemo.PasteFromClipboard;
end;
```

Как следует из листинга 17.1, для выполнения функций работы с буфером обмена необходимо просто вызывать методы класса `TMemo`. Кроме того, поместить текст в буфер обмена можно и вручную — с помощью свойства `Clipboard.AsText`:

```
Clipboard.AsText := 'Delphi Rules';
```

В 16-разрядной среде для свойства AsText существовало ограничение на максимальный размер строки в 255 символов, и необходимо было использовать методы SetTextBuf() и GetTextBuf() для того, чтобы поместить в буфер обмена строку большего размера. В 32-разрядном Delphi такого ограничения нет, и используемый тип свойства AsText — длинная строка. Тем не менее SetTextBuf() и GetTextBuf() все еще поддерживаются в Delphi.

На заметку

Методы Clipboard SetTextBuf() и GetTextBuf() используют типы PChar языка Pascal в качестве буфера для передачи и получения данных из Clipboard. Используя эти методы, можно привести длинную строку к типу PChar, чтобы в дальнейшем не выполнять никаких преобразований String в PChar.

Использование буфера обмена для работы с изображениями

В буфер обмена можно скопировать не только текст, но и графические данные, а затем вставить их куда-либо, как было показано на примере нашего MDI-приложения. Программы обработки событий, обеспечивающих эти операции, приведены в листинге 17.2.

Листинг 17.2. Работа с графическими данными в буфере обмена

```
procedure TMdiBMPForm.mmiCopyClick(Sender: TObject);
begin
  inherited;
  Clipboard.Assign(imgMain.Picture);
end;

procedure TMdiBMPForm.mmiPasteClick(Sender: TObject);
{ Этот метод копирует содержимое буфера обмена в imgMain }
begin
  inherited;
  // Копирует содержимое буфера обмена в imgMain
  imgMain.Picture.Assign(Clipboard);
  ClientWidth := imgMain.Picture.Width;
  { Настройка полос прокрутки }
  VertScrollBar.Range := imgMain.Picture.Height;
  HorzScrollBar.Range := imgMain.Picture.Width;
end;
```

Для доступа к глобальной переменной Clipboard следует включить модуль CLIPBRD в секцию uses модуля, использующего эту переменную.

В листинге 17.2 приведена программа обработки события mmiCopyClick(), которая для копирования изображения в буфер обмена использует метод Clipboard.Assign(). После этого можно вставить изображение в другое приложение Win32 — нужно лишь, чтобы оно поддерживало формат CF_BITMAP (таким приложением может быть, например, Windows Paint).

Процедура mmiPasteClick() определяет, поддерживается ли формат данных в буфере обмена (CF_BITMAP или CF_PICTURE). Если да, то используется метод Image.Assign() для копирования изображения из буфера обмена и настройки полос прокрутки.

На заметку

Формат CF_PICTURE в Win32 не является стандартным. Это собственный формат, который используется Delphi-приложениями для определения того, имеют ли данные буфера обмена TPicture-совместимый формат (например, растровое изображение или метафайл). Если вы выполните регистрацию собственного формата, класс TPicture также будет его поддерживать. Для получения более подробной информации о совместимых с классом TPicture форматах обратитесь к разделу "TPicture" интерактивной справочной системы Delphi.

Создание собственного формата для буфера обмена

Представьте себе, что вы вводите информацию об адресах клиентов в программу базы данных и вам встретилась запись, лишь незначительно отличающаяся от предыдущей. Было бы очень удобно иметь возможность копировать содержимое предыдущей записи и вставлять в текущую, вместо того чтобы повторно заполнять каждое поле. Вам может понадобиться вставить ту же информацию и в другое приложение, например в качестве адреса письма. Ниже представлен пример создания объекта, который "знает" о буфере обмена Win32 и способен сохранять в нем свои специфически отформатированные данные. Рассматривая этот пример, вы также узнаете, как сохранять информацию в буфере обмена в формате CF_TEXT, для того чтобы и другие приложения, поддерживающие этот формат, могли ею воспользоваться.

Создание объектов, способных работать с буфером обмена

Возможно, наиболее подходящим способом определения нового формата буфера обмена вам кажется создание класса-потомка TClipboard. Да, конечно, такой класс может оказаться полезным в каждом конкретном случае, но весьма утомительно постоянно изменять его с целью поддержки дополнительных форматов или при необходимости переопределить данные. К тому же, если каждый из семидесяти различных производителей выйдет на рынок со своим классом-потомком TClipboard для своих форматов, возникнут серьезные проблемы при попытке одновременного использования хотя бы нескольких из них, поскольку классы-потомки TClipboard будут конфликтовать между собой.

Лучшее решение — организовать данные в объект и затем сделать так, чтобы именно *он* знал, как работать с классом TClipboard, а не наоборот. Этот подход и был использован Borland в компонентах Delphi. Компонент TMemo знает, как поместить свои данные в буфер обмена, а компонент TImage — как поместить свои. Все компоненты используют один и тот же объект TClipboard, что позволяет избежать конфликтов между ними. В этом разделе показано, как реализовать данный подход для определения пользовательского формата для буфера обмена. Новый формат представляет собой не что иное, как запись с информацией об имени, возрасте и дате рождения клиета. Исходный код модуля, определяющего методы копирования и вставки данных в буфер обмена, приведен в листинге 17.3.

Листинг 17.3. Модуль, определяющий формат данных для буфера обмена

```
unit cbdata;
interface
uses SysUtils, Windows, clipbrd;

const

  DDGData = 'CF_DDG'; // Константа для регистрации нового формата
type

  // Данные записи для хранения в буфере обмена
  TDataRec = packed record
    LName: string[10];
    FName: string[10];
    MI: string[2];
    Age: Integer;
    BirthDate: TDateTime;
  end;

  { Определяет объект типа TDataRec, содержащий методы для
    копирования и вставки данных в буфер обмена и из него }
  TData = class
  public
    Rec: TDataRec;
    procedure CopyToClipboard;
    procedure GetFromClipboard;
  end;

var
  CF_DDGDATA: word; // Получает возвращаемое значение RegisterClipboardFormat()

implementation

procedure TData.CopyToClipboard;
{ Эта функция копирует содержимое поля типа TDataRec под именем
  Rec в буфер обмена и как двоичные данные, и как текст.
  Оба формата будут доступны при вставке из буфера обмена.  }
const
  CRLF = #13#10;
var
  Data: THandle;
  DataPtr: Pointer;
  TempStr: String[50];
begin
  // Размещает SizeOf(TDataRec) байт в динамической области
  Data := GlobalAlloc(GMEM_MOVEABLE, SizeOf(TDataRec));
  try
    // Получает указатель на первый байт размещенной памяти
    DataPtr := GlobalLock(Data);
```

```
    try
      // Перемещает данные из Rec в блок памяти
      Move(Rec, DataPtr^, SizeOf(TDataRec));
      { Если в буфер обмена копируется сразу несколько форматов,
        должен быть вызван метод Clipboard.Open. Если копируются данные
        лишь одного формата, в вызове этого метода нет необходимости. }
      Clipboard.Open;
      // Сначала копируем данные в их "родном" формате
      Clipboard.SetAsHandle(CF_DDGDATA, Data);
      // Теперь копируем данные в текстовом формате
      with Rec do
        TempStr := FName+CRLF+LName+CRLF+MI+CRLF+IntToStr(Age)+CRLF+
          DateTimeToStr(BirthDate);
      Clipboard.AsText := TempStr;
      { Если был сделан вызов Clipboard.Open, необходим
        вызов Clipboard.Close. }
      Clipboard.Close
    finally
      // Освобождает глобально выделенную память
      GlobalUnlock(Data);
    end;
  except
    { Вызов GlobalFree требуется лишь при обработке исключений. В
 противном
      случае буфер обмена управляет выделенной памятью сам.}
    GlobalFree(Data);
    raise;
  end;
end;

procedure TData.GetFromClipboard;
{ Этот метод вставляет содержимое буфера обмена в формате CF_DDGDATA.
  Данные хранятся в поле TDataRec этого объекта. }
var
  Data: THandle;
  DataPtr: Pointer;
  Size: Integer;
begin
  // Получает указатель буфера обмена
  Data := Clipboard.GetAsHandle(CF_DDGDATA);
  if Data = 0 then Exit;
  // Получает указатель на блок памяти, относящийся к Data
  DataPtr := GlobalLock(Data);
  try
    // Получаем размер данных для вставки
    if SizeOf(TDataRec) > GlobalSize(Data) then
      Size := GlobalSize(Data)
    else
      Size := SizeOf(TDataRec);
    // Копируем данные в поля TDataRec
```

```
    Move(DataPtr^, Rec, Size)
  finally
    // Освобождаем указатель на блок памяти
    GlobalUnlock(Data);
  end;
end;

initialization
  // Регистрируем новый формат буфера обмена
  CF_DDGDATA := RegisterClipboardFormat(DDGData);
end.
```

Этот модуль решает сразу несколько задач. Прежде всего, он регистрирует новый формат для работы с Win32 посредством вызова функции `RegisterClipboardFormat()`. Эта функция возвращает значение, идентифицирующее новый формат. Каждое приложение, зарегистрировавшее тот же формат (задаваемый строковой переменной), получит одно и то же значение при вызове этой функции. Новый формат становится доступен в списке форматов буфера обмена. Этот список вызывается свойством `Clipboard.Formats`.

Модуль также определяет запись, содержащую данные для вставки через буфер и объект, инкапсулирующий эту запись. Запись `TDataRec` содержит строковые поля с именем клиента (целочисленный тип) и поле `TDataTime` с датой рождения клиента.

Объект инкапсулирует `TDataRec` и `TData`, определяя методы `CopyToClipboard()` и `GetFromClipboard()`.

Метод `CopyToClipboard()` помещает содержимое поля `TData.Rec` в буфер обмена в двух форматах, `CF_DDGDATA` и `CF_TEXT`. Как вы уже знаете, `CF_TEXT` — предопределенный формат буфера обмена. Текстовая версия содержимого `TData.Rec` со строковыми значениями полей, разделенными символами перевода строки, помещается в буфер обмена. Перед тем как готовая строка попадает в буфер обмена, все поля, отличные от строковых, преобразуются в строковые. Метод `Clipboard.SetHandle()` помещает дескриптор в буфер обмена в формате, задаваемом его параметром. В нашем случае параметром будет новый формат для буфера обмена `CF_DDGDATA`.

Перед вызовом `Clipboard.SetHandle()` метод готовит корректный экземпляр объекта `THandle`, который должен стать параметром `SetAsHandle()`. Дескриптор представляет собой блок памяти с данными, посылаемыми в буфер обмена (обратите внимание на врезку "Работа с объектами THandle" ниже в этой главе). Следующая строка заставляет Win32 разместить `Sizeof(TDataRec)` байт памяти, которую можно перемещать, и возвратить дескриптор памяти в переменную `Data`:

```
Data := GlobalAlloc(GMEM_MOVEABLE, SizeOf(TDataRec));
```

Указатель на эту память можно получить с помощью следующего оператора:

```
DataPtr := GlobalLock(Data);
```

Затем данные помещаются в блок памяти с помощью функции `Move()`. В оставшихся строках кода метод `Clipboard.Open()` открывает буфер обмена, чтобы не допустить его возможное использование другими приложениями во время помещения в него данных :

```
Clipboard.Open;
try
  Clipboard.SetAsHandle(CF_DDGDATA, Data);
```

```
  with Rec do
    TempStr := FName+CRLF+LName+CRLF+MI+CRLF+IntToStr(Age)+CRLF+
      DateTimeToStr(BirthDate);
    Clipboard.AsText := TempStr;
finally
  Clipboard.Close
End;
```

Как правило, нет необходимости вызывать метод `Open()`, если только в буфер обмена не помещается сразу несколько форматов, как в предыдущем примере. Причина в том, что каждое помещение данных в буфер связано с использованием одного из его методов (например, `Clipboard.SetTextBuf()`) или свойств (например, `Clipboard.AsText`), что вынуждает буфер обмена очистить свое содержимое, применяя внутренние вызовы `Open()` и `Close()`. Этого можно избежать, если в самом начале вызвать метод `Clipboard.Open()`, что позволит записывать в буфер обмена данные в нескольких форматах одновременно. Если не вызвать метод `Open()`, то в буфере обмена будет доступен лишь последний формат — `CF_TEXT`. Строки после вызова метода `Open()` просто помещают данные в буфер обмена, после чего вызывается метод `ClipBoard.Close()`.

После этого за управление памятью, выделенной для буфера обмена функцией `GlobalAlloc()`, отвечает Win32. Вызов `GlobalFree()` необходим только для обработки исключения, если оно возникнет во время процесса копирования. Не вызывайте `GlobalFree()` в других случаях, поскольку система Win32 уже взяла на себя управление памятью, выделенной для буфера обмена.

Данные в форматах `CF_DDGDATA` и `CF_TEXT` можно вставить из буфера обмена в ту же программу или в другие приложения, что мы сейчас и продемонстрируем.

Метод `TData.GetFromClipboard()` выполняет обратную операцию: берет данные из буфера обмена в формате `CF_DDGDATA` и помещает их в поле `TData.Rec`. Комментарии в листинге 17.4 поясняют, как этот метод действует. Приложение-пример, которое мы приведем ниже, иллюстрирует использование этого модуля. Обратите внимание: наш объект может быть легко модифицирован для хранения записи любого нового типа.

Не освобождайте дескриптор, возвращаемый функцией `GetAsHandle()`, — он принадлежит не вашему приложению, а буферу обмена. Благодаря этому данные, на которые ссылается указатель, могут быть скопированы.

Работа с объектами `THandle`

Тип `THandle` — не более чем 32-битовое целое, представляющее индекс таблицы, в которой Win32 хранит информацию о блоке памяти. Существует много типов `THandle`, и Delphi инкапсулирует большинство из них в соответствующих классах (`TIcon`, `TBitmap`, `TCanvas` и др.).

Определенные функции Win32 (в частности, различные функции буфера обмена) используют динамическую область памяти для работы с данными буфера. Чтобы работать с этой областью памяти, вам придется воспользоваться функциями размещения памяти из приведенного ниже списка.

`GlobalAlloc()`	Размещает указанное количество байтов в динамической памяти и возвращает `THandle` этого объекта.
`GlobalFree()`	Освобождает память, размещенную `GlobalAlloc()`.
`GlobalLock()`	Возвращает указатель и блокирует глобальный объект памяти, полученный `GlobalAlloc()`.
`GlobalUnlock()`	Открывает память, закрытую `GlobalLock()`.

Использование пользовательского формата данных для буфера обмена

Главная форма проекта, иллюстрирующего использование определенного пользователем формата данных для буфера обмена, показана на рис. 17.1.

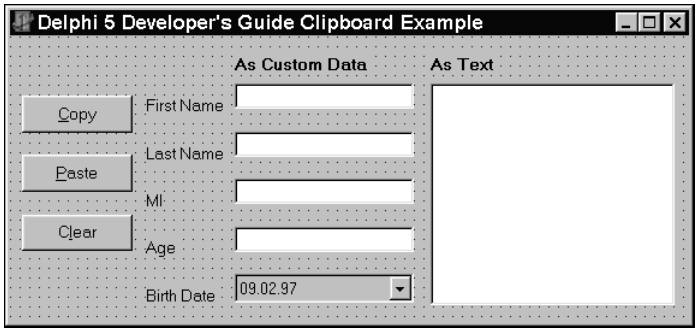


Рис. 17.1. Главная форма приложения, использующего для буфера обмена формат, определенный пользователем

Как видите, эта форма содержит все элементы управления, необходимые для заполнения данными полей записи `TDataRec` объекта `TData`. Текст этого приложения приведен в листинге 17.4), а соответствующий проект `Ddgcbp.dpr` находится на компакт-диске, прилагаемом ко второму тому (см. также `www.williamspublishing.com`.).

Листинг 17.4. Исходный текст приложения для работы с пользовательским форматом данных буфера обмена

```
unit MainFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, clipbrd, Mask, ComCtrls;
type

  TMainForm = class(TForm)
    edtFirstName: TEdit;
    edtLastName: TEdit;
    edtMI: TEdit;
    btnCopy: TButton;
    btnPaste: TButton;
    meAge: TMaskEdit;
    btnClear: TButton;
    lblFirstName: TLabel;
    lblLastName: TLabel;
```

```
    lblMI: TLabel;
    lblAge: TLabel;
    lblBirthDate: TLabel;
    memAsText: TMemo;
    lblCustom: TLabel;
    lblText: TLabel;
    dtpBirthDate: TDateTimePicker;
    procedure btnCopyClick(Sender: TObject);
    procedure btnPasteClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

uses cbdata;

{$R *.DFM}

procedure TMainForm.btnCopyClick(Sender: TObject);
// Этод метод копирует данные из управляющих элементов формы в буфер обмена
var
  DataObj: TData;
begin
  DataObj := TData.Create;
  try
    with DataObj.Rec do
    begin
      FName     := edtFirstName.Text;
      LName     := edtLastName.Text;
      MI        := edtMI.Text;
      Age       := StrToInt(meAge.Text);
      BirthDate := dtpBirthDate.Date;
      DataObj.CopyToClipboard;
    end;
  finally
    DataObj.Free;
  end;
end;

procedure TMainForm.btnPasteClick(Sender: TObject);
{ Этот метод вставляет данные в формате CF_DDGDATA из буфера
  обмена в управляющие элементы формы. Текстовая версия этих
  данных копируется в компонент TMemo формы }
```

```
var
  DataObj: TData;
begin
  btnClearClick(nil);
  DataObj := TData.Create;
  try
    // Проверка, доступен ли формат CF_DDGDATA
    if Clipboard.HasFormat(CF_DDGDATA) then
      // Копирование данных в формате CF_DDGDATA в элементы управления формы
      with DataObj.Rec do
      begin
        DataObj.GetFromClipboard;
        edtFirstName.Text := FName;
        edtLastName.Text  := LName;
        edtMI.Text        := MI;
        meAge.Text        := IntToStr(Age);
        dtpBirthDate.Date := BirthDate;
      end;
  finally
    DataObj.Free;
  end;
  // Копирование текстовой версии данных в компонент формы TMemo
  if Clipboard.HasFormat(CF_TEXT) then
    memAsText.PasteFromClipboard;
end;

procedure TMainForm.btnClearClick(Sender: TObject);
var
  i: integer;
begin
  // Очищает содержимое элементов управления формы
  for i := 0 to ComponentCount - 1 do
    if Components[i] is TCustomEdit then
      TCustomEdit(Components[i]).Text := '';
end;

end.
```

Когда пользователь щелкает на кнопке Copy, данные, содержащиеся в объектах `TEdit` и `TmaskEdit`, копируются в поле `TDataRec` объекта `TData`. Затем вызывается метод `TData.CopyToClipboard()`, помещающий эти данные в буфер обмена.

По щелчку на кнопке Paste происходит обратный процесс. Если данные в буфере обмена имеют формат `CF_DDGDATA`, они помещаются в элементы формы. Текстовое представление данных также копируется и вставляется в компонент `TMemo`. Результат операции Paste показан на рис. 17.2. Естественно, текстовый вариант данных можно вставить и в любое другое Windows-приложение.

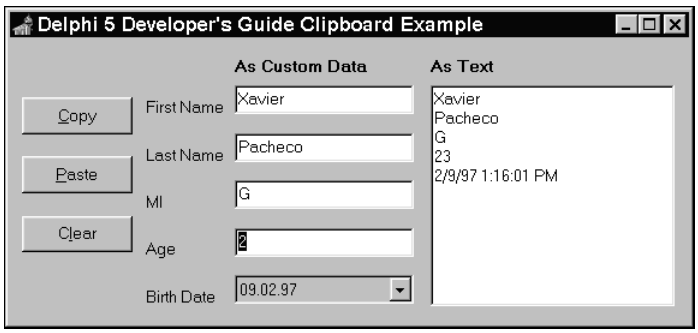


Рис. 17.2. Данные вставлены в главную форму

Кнопка Clear очищает содержимое всех элементов управления главной формы.

Резюме

Обмен данными между различными приложениями — чрезвычайно эффективная технология. Позволяя своим приложениям обмениваться данными с другими приложениями, вы делаете их более удобными в эксплуатации и экономите время пользователей. В этой главе мы обсудили, как использовать встроенные функции буфера обмена для работы с управляющими элементами формы. Кроме того, было показано, как создать собственный формат для работы с буфером обмена. Другим, еще более эффективным способом взаимодействия процессов является технология COM — подробное ее описание дается в последующих главах.

Мультимедиа-программирование в Delphi

Глава

18

Компонент Delphi TMediaPlayer — блестящее подтверждение поговорки “Мал золотник, да дорог”. Незначительный на первый взгляд компонент инкапсулирует большинство функциональных возможностей Windows Media Control Interface (MCI) — части Windows API, ответственной за управление устройствами мультимедиа.

Delphi делает мультимедиа-программирование очень простым, окончательно отодвигая в прошлое изрядно надоевшую программу “Hello, World”. И действительно, зачем писать на экране “Hello, World”, если это же приветствие можно воспроизвести в звуковом или даже видеоварианте с одинаковой легкостью?

В этой главе вы узнаете, как создать простой, но работоспособный медиаплейер и даже полнофункциональный проигрыватель звуковых компакт-дисков. Здесь же описаны особенности работы компонента TMediaPlayer. Однако вы извлечете гораздо больше пользы из материала этой главы, если ваш компьютер оснащен соответствующими устройствами мультимедиа: звуковой платой и приводом компакт-дисков.

Создание простого медиаплейера

Лучший способ чему-либо научиться — просто попытаться это сделать. Чтобы быстро создать медиаплейер, достаточно поместить в форму компоненты TMediaPlayer, Tbutton и TOpenDialog. Подобная форма показана на рис. 18.1.

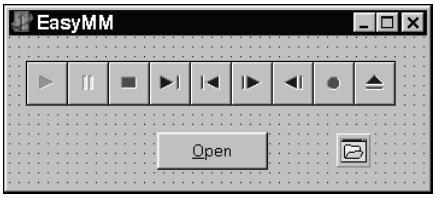


Рис. 18.1. Медиаплейер EasyMM

Медиаплейер EasyMM работает следующим образом: по щелчку на кнопке раскрывается диалоговое окно OpenDialog с предложением выбрать файл. Далее медиаплейер готовится к тому, чтобы воспроизвести файл, который был выбран в окне OpenDialog. После завершения подготовки, для воспроизведения файла достаточно щелкнуть на кнопке Play. Следующий фрагмент кода принадлежит методу OnClick, в котором выполняется выбор файла и его открытие:

```
procedure TMainForm.BtnOpenClick(Sender: Tobject);
begin
  if OpenDialog1.Execute then
  begin
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open;
  end;
end;
```

В начале работы этого метода вызывается диалоговое окно OpenDialog1, и, если имя файла выбрано, свойство FileName окна OpenDialog1 копируется в свойство FileName объекта MediaPlayer1. Метод Open объекта MediaPlayer вызывается для того, чтобы подготовить указанный файл к воспроизведению.

Можно ограничить перечень файлов, представляемых для выбора в окне OpenDialog, только мультимедиа-файлами. Компонент TMediaPlayer поддерживает большой набор разнообразных форматов файлов, но в данном примере нам вполне достаточно WAVE-, AVI- и MIDI-файлов. Класс OpenDialog включает такую возможность, и мы можем ею воспользоваться. Выделите компонент OpenDialog1 в окне инспектора объектов, выберите свойство Filter и щелкните на кнопке с многоточием в правой части строки свойства. На экране раскроется окно Filter Editor редактора свойства Filter. Занесите в него информацию о требуемых типах файлов с расширениями .WAV, .AVI, и .MID, как показано на рис. 18.2.

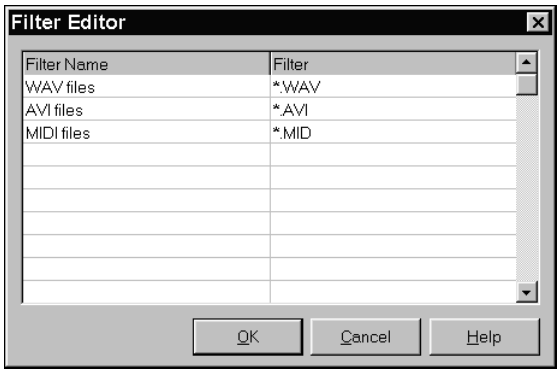


*Рис. 18.2. Окно **Filter Editor** редактора свойства **Filter***

Этот проект сохранен в файле EasyMM.dpr, а основной его модуль — в файле Main.pas. Медиаплейер готов к работе. Запустите программу и протестируйте ее с одним из мультимедиа-файлов, имеющихся на жестком диске. Надо полагать, вас неоднократно пытались убедить в том, что мультимедиа-программирование — крайне сложная вещь. Теперь у вас есть доказательства, что это не так.

Использование WAV-файлов в приложениях

Стандартным форматом хранения аудиоинформации в Windows является формат WAV. Как следует из названия (*wave* — волна), WAV-файл хранит звуковую информацию в двоичном представлении, математически описывающем звуковую волну. Главным достоинством WAV-файлов стало признание этого формата в компьютерной отрасли, благодаря чему он получил широчайшее распространение. Основной недостаток WAV-файлов — их громоздкость: несколько песен могут занять солидную часть жесткого диска.

Компонент TMediaPlayer позволяет легко интегрировать WAV-звуки в создаваемые приложения. Как уже было показано, воспроизведение WAV-файлов — дело несложное: сообщите TMediaPlayer имя файла, откройте этот компонент и воспроизведите требуемый файл. Даже небольшие аудиовозможности добавят самому скромному приложению необходимую законченность и внешний лоск.

Если воспроизведение WAV-файлов — единственное, что требуется достичь, то нет необходимости использовать компонент TMediaPlayer. Вместо этого можно воспользоваться функцией Win32 API PlaySound(), объявление которой содержится в модуле MMSystem. Эта функция определяется следующим образом:

```
function PlaySound(pszSound: Pchar; hmod: HMODULE; fdwSound: DWORD): BOOL; stdcall;
```

Функция PlaySound() обладает возможностью воспроизводить WAV-звук из файла, из памяти или из ресурсного файла, связанного с приложением. Ей передается три параметра.

- Первый параметр, pzsSound, представляет собой переменную типа Pchar, которая задает имя файла, псевдоним, имя ресурса, запись в Registry , запись в разделе [sounds] файла WIN.INI или указатель на WAV-данные, расположенные в памяти, — одним словом, идентифицирует данные, которые следует воспроизвести.
- Второй параметр, hmod, является указателем на выполняемый файл, содержащий ресурс для загрузки. Он должен быть равен 0, если параметр fdwSound не равен snd_Resource.
- Третий параметр, fdwSound, содержит флаги, описывающие, как должен воспроизводиться звук. Флаг может содержать любую комбинацию значений, приведенных ниже.

Флаг	Описание
SND_APPLICATION	Звук воспроизводится приложением, ассоциированным с данным типом звуковой информации
SND_ALIAS	В первом параметре указано имя системного события, определенного в Registry или в файле WIN.INI. Не используйте этот флаг совместно с флагами SND_FILENAME или SND_RESOURCE, потому что они взаимно исключают друг друга
SND_ALIAS_ID	В первом параметре указан предопределенный идентификатор звука
SND_FILENAME	Первый параметр задает имя файла
SND_NOWAITE	Флаг указывает, что, если драйвер занят, следует немедленно вернуть управление без воспроизведения звука
SND_PURGE	Прекращается воспроизведение всех звуков. Если параметр pszSound не равен нулю, останавливается воспроизведение всех экземпляров указанного звука. Если параметр pszSound равен нулю, останавливается воспроизведение всех звуков, инициированных данным заданием. Для остановки обработки событий SND_RESOURCE необходимо указать правильный дескриптор
SND_RESOURCE	Первый параметр указывает на идентификатор ресурса. При использовании этого флага параметр hmod должен определять объект, включающий требуемый ресурс
SND_ASYNC	Требуется асинхронное воспроизведение звука с немедленным возвратом управления из функции. Подобный подход используется для получения эффекта фонового звучания
SND_LOOP	Многократное воспроизведение звука в цикле, до тех пор пока этот процесс не будет остановлен извне. При использовании этого флага необходимо также использовать флаг SND_ASYNC
SND_MEMORY	Воспроизводится WAV-файл из области памяти, задаваемой параметром pszSound
SND_NODEFAULT	Если звук не найден, функция PlaySound() прекращает свое выполнение немедленно, без воспроизведения звука, указанного в Registry по умолчанию

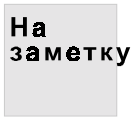
Флаг	Описание
SND_NOSTOP	Воспроизводит звук только в том случае, если в данный момент он еще не воспроизводится. Функция PlaySound() возвращает значение True, если звук воспроизводится, и False — если не воспроизводится. Если этот флаг не определен, Win32 остановит воспроизведение любых других звуков, чтобы воспроизвести звук, заданный параметром pszSound
SND_SYNC	Воспроизведение звука в синхронном режиме. Возврата из функции не происходит до тех пор, пока воспроизведение звука не завершится

Для остановки воспроизведения асинхронно проигрываемого WAV-звука вызовите функцию PlaySound() и передайте значения Nil или 0 всем ее параметрам:

```
PlaySound(Nil,0,0); //Останавливает все воспроизводимые WAV-звуки
```

Чтобы остановить любые звуки, в том числе и не WAV, добавьте флаг snd_Purge:

```
PlaySound(Nil,0,snd_Purge); // Останавливает все воспроизводимые звуки
```



Win32 API все еще поддерживает функцию sndPlaySound(), являющуюся частью Windows 3.x API. Это делается лишь из соображений обратной совместимости, однако вполне может оказаться, что эта функция будет отсутствовать в последующих реализациях Win32 API. Для обеспечения совместимости в будущем предпочтительнее использовать функцию Win32 API PlaySound().

Воспроизведение видео

Формат AVI (сокращение от *audio-video interleave* — аудио/видеочередование) — один из наиболее известных форматов файлов для одновременного хранения аудио- и видеоинформации. В частности, несколько AVI-файлов содержится и на компакт-диске с Delphi 5, в каталоге \Runimage\Delphi50\Demos\Coolstuf.

Для отображения AVI-файлов можно использовать ту же простенькую программу медиаплейера, которая обсуждалась выше. Достаточно выбрать AVI-файл в окне OpenDialog1 и щелкнуть на кнопке Play. Обратите внимание на то, что AVI-файлы всегда воспроизводятся в собственном окне.

Показ первого кадра

Возможна ситуация, когда требуется отобразить первый кадр AVI-файла перед тем, как запустить его на воспроизведение. Такое действие создает эффект замороженного кадра. С этой целью сначала вызовите метод Open() компонента TMediaPlayer, установите его свойство Frames равным 1, после чего вызовите метод Step(). Свойство Frames указывает компоненту MediaPlayer, сколько кадров перемещать при вызове методов Step() или Back(). Метод Step() обновляет окно компонента TMediaPlayer и отображает в нем текущий кадр:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    with MediaPlayer1 do
```

```
    begin
      Filename := OpenDialog1.Filename;
      Open;
      Frames := 1;
      Step;
      Notify := True;
    end;
end;
```

Использование свойства Display

Свойству TMediaPlayer.Display можно присвоить значение, позволяющее воспроизводить AVI-файл в указанном окне, без создания своего собственного. Для этого добавьте в окно медиаплейера компонент TPane, как показано на рис. 18.3. После этого сохраните проект в новой папке под именем DDGMPlay.dpr.

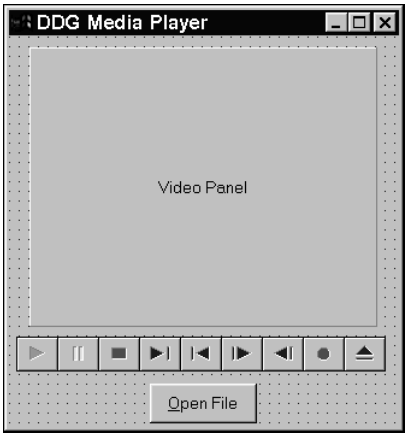


Рис. 18.3. Главное окно проекта DDGMPlay

В окне инспектора объектов щелкните на кнопке со стрелкой вниз, расположенной в строке свойства Display объекта MediaPlayer1. Обратите внимание: в раскрывшемся списке перечислены все компоненты проекта. В качестве значения свойства Display установите Panel1.

Теперь, если запустить программу и выбрать для воспроизведения AVI-файл, он будет отображаться на указанной панели. Обратите также внимание на то, что AVI-файл не занимает всего окна, поскольку в нем самом задается размер поля, в котором он будет воспроизводиться.

Использование свойства DisplayRect

Свойство DisplayRect имеет тип TRect и задает размер окна, в котором будет воспроизводиться AVI-файл. Обычно свойство используют, чтобы AVI-файл выводился с растяжкой или сжатием изображения до указанных размеров. Например, если требуется, чтобы AVI-файл занимал всю область Panel1, свойству DisplayRect следует назначить размер этой панели:

```
MediaPlayer1.DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
```

Эту строку кода можно добавить к обработчику события кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then begin
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open;
    MediaPlayer1.DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
  end;
end;
```

Установить значение свойства DisplayRect можно только после вызова метода TMediaPlayer.Open().

События класса TMediaPlayer

Компонент TMediaPlayer имеет два специфических события: OnPostClick и OnNotify.

Событие OnPostClick очень похоже на событие OnClick, но последнее возникает непосредственно по щелчку на компоненте, тогда как событие OnPostClick — по завершении действий, вызванных подобным щелчком. Например, если во время работы компонента TMediaPlayer щелкнуть на кнопке Play, будет сгенерировано событие OnClick, событие же OnPostClick генерируется сразу по завершении воспроизведения данных устройством мультимедиа.

Событие OnNotify представляет бо́льший интерес. Оно генерируется, как только компонент TMediaPlayer завершает выполнение какого-либо из методов управления мультимедиа (Back, Close, Eject, Next, Open, Pause, PauseOnly, Play, Previous, Resume, Rewind, StartRecording, Step или Stop), но лишь тогда, когда его свойство Notify имеет значение True. Чтобы проиллюстрировать работу события OnNotify, добавим обработчик этого события в проект DDGMPlay. В подпрограмме обработки этого события будем выводить диалоговое окно с сообщением — оно будет появляться на экране по завершении выполнения команды:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  MessageDlg(Media control method executed', mtInformation, [mbOk], 0);
end;
```

Не забудьте после открытия медиаплейера в обработчике события кнопки Button1 OnClick назначить свойству Notify значение True:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    with MediaPlayer1 do
    begin
      Filename := OpenDialog1.Filename;
      Open;
      DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
      Notify := True;
    end;
end;
```

Обратите внимание на то, что весь код, работающий с компонентом MediaPlayer, помещен в конструкцию with..do. Как уже неоднократно упоминалось в предыдущих главах, эта конструкция позволяет достичь максимальной ясности программного текста и способствует повышению его производительности за счет упрощения имен всех используемых свойств и методов.

Исходный код проекта DDGMPlay

Теперь вам известны основные методы воспроизведения файлов WAV и AVI в Delphi. В листингах 18.1 и 18.2 содержатся полный исходный текст проекта DDGMPlay и исходный код модуля Main.pas.

Листинг 18.1. Исходный текст файла проекта DDGMPlay.dpr

```
program DDGMPlay;

uses
  Forms,
  Main in 'MAIN.PAS' {MainForm};

{$R *.RES}

begin
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

Листинг 18.2. Исходный код модуля Main.pas

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    MediaPlayer1: TMediaPlayer;
    OpenDialog1: TOpenDialog;
    Button1: TButton;
    Panel1: TPanel;
    procedure Button1Click(Sender: TObject);
    procedure MediaPlayer1Notify(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;
```

```
var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    with MediaPlayer1 do
    begin
      Filename := OpenDialog1.Filename;
      Open;
      DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
      Notify := True;
    end;
end;

procedure TMainForm.MediaPlayer1Notify(Sender: TObject);
begin
  MessageDlg('Выполняется метод управления мультимедиа', mtInformation, [mbOk],
0);
end;

end.
```

Поддержка устройств

Компонент `TMediaPlayer` работает с большим количеством устройств, поддерживаемых Win32 MCI. Тип устройства, которым управляет компонент `TMediaPlayer`, задается свойством `DeviceType`. В табл. 18.1 описаны различные допустимые значения свойства `DeviceType`.

Таблица 18.1. Значения свойства DeviceType объекта TMediaPlayer

Значения DeviceType	Устройство
`dtAutoSelect`	Компонент `TMediaPlayer` автоматически выберет правильное устройство исходя из имени воспроизводимого файла
`dtAVIVideo`	Файл AVI. Эти файлы имеют расширение `.AVI` и содержат как звук, так и видео кинематографического качества
`dtCDAudio`	Звуковые компакт-диски, проигрываемые в приводе компакт-дисков компьютера
`dtDAT`	Цифровой кассетный аудиоплейер (DAT), соединенный с компьютером
`dtDigitalVideo`	Цифровое видеоустройство (например, цифровая видеокамера)
`dtMMMovie`	Формат multimedia movie
`dtOther`	Неопределенный мультимедийный формат

Окончание табл. 18.1.

Значения DeviceType	Устройство
dtOverlay	Устройство наложения изображения
dtScanner	Сканер, подключенный к компьютеру
dtSequencer	Устройство-секвенсор, способное воспроизводить MIDI-файлы. Обычно эти файлы имеют расширения имени .MID или .RMI
dtVCR	Видеомагнитофон (VCR), подключенный к компьютеру
dtVideodisc	Проигрыватель видеокомпакт-дисков, подключенный к компьютеру
dtWaveAudio	Звуковой файл WAVE. Эти файлы имеют расширение .WAV

Хотя, как вы могли убедиться, TMediaPlayer поддерживает множество разнообразных форматов, в этой главе в основном рассматриваются форматы WAV, AVI и CD Audio, получившие наибольшее распространение в Windows.

На заметку

Компонент TMediaPlayer является производным от класса TWinControl, а это означает, что с помощью мастеров Delphi 5 он может быть легко инкапсулирован в элемент управления ActiveX. Важным следствием из этого факта является возможность вставить компонент TMediaPlayer в Web-страницу с целью расширения ее мультимедийных характеристик. Вдобавок к этому, с помощью нескольких команд на языке JavaScript или VBScript можно предоставить проигрыватель компакт-дисков любому пользователю Internet или intranet, работающему с броузером в среде Windows.

Создание проигрывателя музыкальных компакт-дисков

Теперь мы обсудим более интересные возможности компонента TMediaPlayer, позволяющие создать на его основе полнофункциональный проигрыватель музыкальных компакт-дисков. На рис. 18.4 показан вид главной формы приложения CDPlayer.dpr. Ее текст содержится в модуле CDMain.pas.

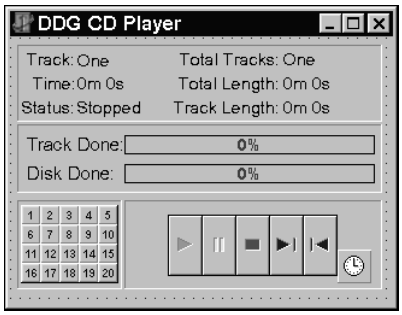


Рис. 18.4. Главная форма приложения проигрывателя компакт-дисков

В табл. 18.2 представлены важнейшие свойства, которые следует установить компонентам, содержащимся в главной форме проигрывателя компакт-дисков.

Таблица 18.2. Важные свойства компонентов проигрывателя компакт-дисков

Компонент	Свойство	Значение
mpCDPlayer	DeviceType	dtAudioCD
sbTrack1 - sbTrack20	Caption	'1'- '20'
sbTrack1 - sbTrack20	Tag	1 - 20

Отображение заставки

Когда программа проигрывателя компакт-дисков запускается, проходит несколько секунд; еще несколько секунд требуется для инициализации компонента `TMediaPlayer` после вызова его метода `Open()`. Задержка между щелчком на пиктограмме и реальным появлением окна программы может привести пользователя в недоумение, и он успеет несколько раз запустить программу до появления первого ее экземпляра на экране. Задержка эта вызвана тем, что Windows требуется определенное время для инициализации подсистемы мультимедиа, происходящей во время открытия компонента `TMediaPlayer`. Чтобы избежать этой проблемы, снабдим проигрыватель компакт-дисков заставкой, которая будет появляться сразу же после запуска программы и подтверждать, что программа действительно начала загружаться.

Первым шагом в разработке заставки будет создание для нее формы. Обычно такая форма содержит панель, но без рамки или заголовка — это придает ей особый, "заставочный" вид. Поместите на панели один или несколько компонентов `TLabel` и компонент `TImage` с графическим изображением или пиктограммой.

Форма заставки для проигрывателя компакт-дисков показана на рис. 18.5, а текст модуля `Splash.pas` приведен в листинге 18.3.

Рис. 18.5. Форма заставки для проигрывателя компакт-дисков

Листинг 18.3. Исходный код модуля `Splash.pas`

```
unit Splash;

interface

uses Windows, Classes, Graphics, Forms, Controls, StdCtrls, ExtCtrls;

type
  TSplashScreen = class(TForm)
    StatusPanel: TPanel;
  end;
```

```
var
  SplashScreen: TSplashScreen;

implementation

{$R *.DFM}

begin
  { Так как заставка отображается раньше основного окна,
    то и создана она должна быть раньше. }
  SplashScreen := TSplashScreen.Create(Application);
  SplashScreen.Show;
  SplashScreen.Update;
end.
```

В отличие от обычной формы, форма заставки создается и выводится в секции инициализации модуля. Поскольку эта секция во всех модулях выполняется перед основным программным блоком, форма появится на экране перед запуском основной части программы.

Не используйте метод `Application.CreateForm()` для создания экземпляра формы-заставки. Delphi всегда делает первую создаваемую этим методом форму главной формой приложения, что несколько не соответствует нашим целям.

Создание проигрывателя компакт-дисков

Создадим обработчик события `OnCreate` формы, в котором программа проигрывателя компакт-дисков будет инициализироваться и открываться. Для этого сначала вызовем метод `Open()`, который проверит, позволяет ли система проигрывать компакт-диски, а затем инициализирует устройство воспроизведения. Если метод `Open()` будет завершен некорректно, генерируется исключительная ситуация `EMCIDeviceError`. В этом случае выполнение программы следует прекратить:

```
try
  { Открыть устройство проигрывателя компакт-дисков. }
  mpCDPlayer.Open;
except
  { Если происходит ошибка, возможно, система
    не способна воспроизводить компакт-диски. }
  on EMCIDeviceError do
  begin
    MessageDlg('Error Initializing CD Player. Program will now exit.',
        mtError, [mbOk], 0);
    Application.Terminate; { Закрытие приложения }
  end;
end;
```

На заметку

Предпочтительным способом завершения приложения Delphi является вызов метода `Close()` главной формы или метода `Application.Terminate`.

После открытия компонента CDPlayer следует установить значение его свойства EnableButtons для того, чтобы появились необходимые для управления устройством кнопки. Какие кнопки станут доступными, зависит от текущего состояния устройства чтения компакт-дисков. Если, например, устройство уже проигрывает компакт-диск, когда вызывается метод Open(), требуется заблокировать кнопку Play. Для проверки текущего статуса устройства воспроизведения компакт-дисков следует проанализировать значение свойства Mode объекта CDPlayer. Все возможные значения этого свойства подробно описаны в интерактивной справочной системе. Оно предоставляет информацию о текущем состоянии устройства: *воспроизведение*, *выключено*, *пауза*, *поиск* и т.п. В нашем случае, представляют интерес только три режима устройства: *выключено*, *пауза* и *воспроизведение*. Вот код, который активизирует нужные кнопки:

```
case mpCDPlayer.Mode of
  mpPlaying:  mpCDPlayer.EnabledButtons :=[btPause, btStop, btNext, btPrev];
  mpStopped, { Когда устройство остановлено, отображаются кнопки,
               заданные по умолчанию. }
  mpPaused : mpCDPlayer.EnabledButtons := [btPlay, btNext, btPrev];
end;
```

Следующий фрагмент программного текста представляет полный исходный код метода TMainForm.FormCreate(). Обратите внимание на то, что после успешного открытия компонента CDPlayer вызывается несколько методов. Целью вызова этих методов является обновление отображаемых данных программы воспроизведения компакт-дисков (например, количество дорожек данного компакт-диска). Эти методы подробно описаны далее в главе.

```
procedure TMainForm.FormCreate(Sender: TObject);
{ Этот метод вызывается при создании формы.
  Он открывает и инициализирует проигрыватель. }
begin
  try
    mpCDPlayer.Open;   // Открываем устройство проигрывания
    { Если компакт-диск уже воспроизводится, достаточно просто
      отобразить статус воспроизведения. }
    if mpCDPlayer.Mode = mpPlaying then
    LblStatus.Caption := 'Playing';
    GetCDTotals;      // Показать общее время выполнения и количество
                      // дорожек на текущем компакт-диске
    ShowTrackNumber;  // Показать текущую дорожку
    ShowTrackTime;    // Показать минуты и секунды текущей дорожки
    ShowCurrentTime;  // Показать текущую позицию компакт-диска
    ShowPlayerStatus; // Обновить статус проигрывателя компакт-дисков
  except
    { Если происходит ошибка, возможно, система не способна
      воспроизводить компакт-диски. }
    on EMCIDeviceError do
    begin
      MessageDlg('Error Initializing CD Player. Program will now exit.',
                 mtError, [mbOk], 0);
      Application.Terminate;
    end;
  end;
```

```
  { Проверяется текущий режим проигрывателя и активизируются
    соответствующие кнопки. }
  case mpCDPlayer.Mode of
    mpPlaying: mpCDPlayer.EnabledButtons := PlayButtons;
    mpStopped, mpPaused: mpCDPlayer.EnabledButtons := StopButtons;
  end;
  SplashScreen.Release; // Закрытие и освобождение окна заставки
end;
```

Последняя строка этого фрагмента закрывает форму заставки. Событие основной формы `OnCreate` — самое подходящее место для этого действия.

Обновление информации проигрывателя компакт-дисков

В процессе воспроизведения информацию в форме `CDPlayerForm` можно постоянно обновлять с помощью компонента `TTimer`. Каждый раз, когда происходит событие таймера, можно вызывать соответствующий метод для обновления, как показано в методе `OnCreate` нашей формы. В результате вид окна на экране постоянно будет отвечать текущему состоянию проигрывателя. Вот исходный код обработки этого события:

```
procedure TMainForm.tmUpdateTimerTimer(Sender: TObject);
{ Это метод является центральным звеном проигрывателя.
  Он обновляет всю информацию через установленный временной интервал }
begin
  if mpCDPlayer.EnabledButtons = PlayButtons then
  begin
    mpCDPlayer.TimeFormat := tfMSF;
    ggDiskDone.Progress := (mci_msf_minute(mpCDPlayer.Position) * 60 +
        mci_msf_second(mpCDPlayer.Position));
    mpCDPlayer.TimeFormat := tfTMSF;
    ShowTrackNumber; // Показывает номер дорожки
    ShowTrackTime; // Показывает общее время текущей дорожки
    ShowCurrentTime; // Показывает истекшее время текущей дорожки
  end;
end;
```

Помимо вызова различных методов обновления, этот метод также обновляет компонент `DiskDoneGauge` для указания истекшего времени компакт-диска. Чтобы получить значение истекшего времени, метод уменьшает значение свойства `TimeFormat` компонента `mpCDPlayer` на значение `tfMSF`, после чего преобразует значение свойства `Position` в минуты и секунды, используя для этого функции `mci_msf_Minute()` и `mci_msf_Second()`. Этот момент заслуживает более подробного разъяснения.

Свойство `TimeFormat`

Свойство `TimeFormat` компонента `TMediaPlayer` определяет способ интерпретации значений свойств `StartPos`, `Length`, `Position`, `Start` и `EndPos`. В табл. 18.3 приведены допустимые значения свойства `TimeFormat`. Эти значения представляют информацию, упакованную в переменную типа `Longint`.

Таблица 18.3. Значения свойства TMediaPlayer.TimeFormat

Значение	Формат хранения времени
tfBytes	Количество байтов
tfFrames	Кадры
tfHMS	Часы, минуты и секунды
tfMilliseconds	Время в миллисекундах
tfMSF	Минуты, секунды и кадры
tfSamples	Число образцов
tfSMPTE24	Часы, минуты, секунды и кадры, исходя из 24 кадров в секунду
tfSMPTE25	Часы, минуты, секунды и кадры, исходя из 25 кадров в секунду
tfSMPTE30	Часы, минуты, секунды и кадры, исходя из 30 кадров в секунду
tfSMPTE30Drop	Часы, минуты, секунды и кадры, исходя из 30 завершенных кадров в секунду
tfTMSF	Дорожки, минуты, секунды и кадры

Подпрограмма преобразования значений времени

Windows API содержит подпрограммы для получения информации о времени в различных упакованных форматах, приведенных в табл. 18.4. При использовании *упакованного формата* различные данные пакуются (кодируются) в одно значение типа `Longint`. Эти функции находятся в библиотеке `MMSystem.dll`, так что при их использовании необходимо обязательно объявить модуль `MMSystem` в секции `Uses`.

Таблица 18.4. Функции распаковки форматов времени мультимедиа

Функция	Работает с	Возвращает
mci_HMS_Hour()	tfHMS	Часы
mci_HMS_Minute()	tfHMS	Минуты
mci_HMS_Second()	tfHMS	Секунды
mci_MSF_Frame()	tfMSF	Кадры
mci_MSF_Minute()	tfMSF	Минуты
mci_MSF_Second()	tfMSF	Секунды
mci_TMSF_Frame()	tfTMSF	Кадры
mci_TMSF_Minute()	tfTMSF	Минуты
mci_TMSF_Second()	tfTMSF	Секунды
mci_TMSF_Track()	tfTMSF	Дорожки

Методы обновления состояния формы проигрывателя компакт-дисков

Как уже отмечалось в этой главе, для регулярного отображения обновленной информации в окне проигрывателя компакт-дисков используется несколько методов. Главная цель каждого из них — обновление текста надписей в верхней части формы и обновление состояния индикаторов в средней ее части.

Метод GetCDTotals()

Метод GetCDTotals(), текст которого показан ниже, служит для получения информации о длине дорожек на текущем компакт-диске и их общем количестве. Эта информация впоследствии используется для обновления нескольких надписей и индикатора DiskDoneGauge. В исходном тексте присутствует также метод AdjustSpeedButtons(), активизирующий нужное количество кнопок SpeedButton, соответствующее количеству дорожек. Кстати, этот метод использует свойство TimeFormat и ранее описанные процедуры преобразования временных форматов:

```
procedure TMainForm.GetCDTotals;
{ Этот метод используется для получения общего времени записи
  и количества дорожек, а также для отображения этой информации.}
var
  TimeValue: longint;
begin
  mpCDPlayer.TimeFormat := tfTMSF; // Установка формата времени
  TimeValue := mpCDPlayer.Length;  // Общая длина записи на компакт-диске
  TotalTracks := mci_Tmsf_Track(mpCDPlayer.Tracks); // Количество дорожек
  TotalLengthM := mci_msf_Minute(TimeValue); // Общее время записи в мин.
  TotalLengthS := mci_msf_Second(TimeValue); // Общее время записи в сек.
  { Изменение надписи Total Tracks }
  LblTotTrk.Caption := TrackNumToString(TotalTracks);
  { Изменение надписи Total Time }
  LblTotalLen.Caption := Format(MSFormatStr, [TotalLengthM, TotalLengthS]);
  { Инициализация индикатора }
  ggDiskDone.MaxValue := (TotalLengthM * 60) + TotalLengthS;
  { Активизирует нужное количество кнопок }
  AdjustSpeedButtons;
end;
```

Метод ShowCurrentTime()

Ниже приведен исходный текст метода ShowCurrentTime(). Он предназначен для определения продолжительности уже воспроизведенной части текущей дорожки и обновления соответствующих управляющих элементов. В нем также используются процедуры преобразования форматов времени, содержащиеся в модуле MMSystem:

```
procedure TMainForm.ShowCurrentTime;
{ Этот метод отображает текущее время воспроизведения дорожки }
```

```
begin
  { Длина дорожки в минутах }
  m := mci_Tmsf_Minute(mpCDPlayer.Position);
  { Секунды }
  s := mci_Tmsf_Second(mpCDPlayer.Position);
  { Обновление надписи Track Time }
  LblTrackTime.Caption := Format(MSFormatStr, [m, s]);
  { Обновление индикатора }
  ggTrackDone.Progress := (60 * m) + s;
end;
```

Метод ShowTrackTime()

Этот метод, текст которого приведен ниже, получает информацию об общем времени воспроизведения текущей дорожки в минутах и секундах и обновляет соответствующую надпись. И в этом случае также используются процедуры преобразования временных форматов. Дополнительно следует проверить, не изменилась ли текущая дорожка со времени последнего вызова функции, — это позволит избежать ненужных вызовов функций и излишнего обновления компонентов:

```
procedure TMainForm.ShowTrackTime;
{ Этот метод обновляет общее время воспроизведения текущей дорожки. }
var
  Min, Sec: Byte;
  Len: Longint;
begin
  { Не обновляет информацию, пока воспроизводится одна и та же дорожка. }
  if CurrentTrack <> OldTrack then
  begin
    Len := mpCDPlayer.TrackLength[mci_Tmsf_Track(mpCDPlayer.Position)];
    Min := mci_msf_Minute(Len);
    Sec := mci_msf_Second(Len);
    ggTrackDone.MaxValue := (60 * Min) + Sec;
    LblTrackLen.Caption := Format(MSFormatStr, [m, s]);
  end;
  OldTrack := CurrentTrack;
end;
```

Исходный текст программы проигрывателя компакт-дисков

Итак, рассмотрены все аспекты проигрывателя компакт-дисков, имеющие отношение к мультимедиа. В листингах 18.4 и 18.5 содержатся полные исходные тексты модулей CDPlayer.dpr и CDMain.pas. В модуле CDMain также продемонстрированы способы манипулирования кнопками типа SpeedButton с помощью их свойства Tag, а также другие интересные методы обновления состояния элементов управления формы.

Листинг 18.4. Исходный текст модуля CDPlayer.dpr

```
program CDPlayer;

uses

  Forms,
  Splash in 'Splash.pas' {SplashScreen},
  CDMain in 'CDMain.pas' {MainForm};

begin
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

Листинг 18.5. Исходный текст модуля CDMain.pas

```
unit CDMain;

interface

uses
  SysUtils, Windows, Classes, Graphics, Forms, Controls, MPlayer, StdCtrls,
  Menus, MMSystem, Messages, Buttons, Dialogs, ExtCtrls, Splash,Gauges;
type
  TMainForm = class(TForm)
  tmUpdateTimer: TTimer;
    MainScreenPanel: TPanel;
    LblStatus: TLabel;
    Label2: TLabel;
    LblCurTrk: TLabel;
    Label4: TLabel;
    LblTrackTime: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    LblTotTrk: TLabel;
    LblTotalLen: TLabel;
    Label12: TLabel;
    LblTrackLen: TLabel;
    Label15: TLabel;
    CDInfo: TPanel;
    SBPanel: TPanel;
    Panel1: TPanel;
    mpCDPlayer: TMediaPlayer;
    sbTrack1: TSpeedButton;
    sbTrack2: TSpeedButton;
```

```
    sbTrack3: TSpeedButton;
    sbTrack4: TSpeedButton;
    sbTrack5: TSpeedButton;
    sbTrack6: TSpeedButton;
    sbTrack7: TSpeedButton;
    sbTrack8: TSpeedButton;
    sbTrack9: TSpeedButton;
    sbTrack10: TSpeedButton;
    sbTrack11: TSpeedButton;
    sbTrack12: TSpeedButton;
    sbTrack13: TSpeedButton;
    sbTrack14: TSpeedButton;
    sbTrack15: TSpeedButton;
    sbTrack16: TSpeedButton;
    sbTrack17: TSpeedButton;
    sbTrack18: TSpeedButton;
    sbTrack19: TSpeedButton;
    sbTrack20: TSpeedButton;
    ggTrackDone: TGauge;
    ggDiskDone: TGauge;
    Label1: TLabel;
    Label3: TLabel;
    procedure tmUpdateTimerTimer(Sender: TObject);
    procedure mpCDPlayerPostClick(Sender: TObject; Button: TMPBtnType);
    procedure FormCreate(Sender: TObject);
    procedure sbTrack1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Закрытые объявления }
    OldTrack, CurrentTrack: Byte;
    m, s: Byte;
    TotalTracks: Byte;
    TotalLengthM: Byte;
    TotalLengthS: Byte;
    procedure GetCDTotals;
    procedure ShowTrackNumber;
    procedure ShowTrackTime;
    procedure ShowCurrentTime;
    procedure ShowPlayerStatus;
    procedure AdjustSpeedButtons;
    procedure HighlightTrackButton;
    function TrackNumToString(InNum: Byte): String;
  end;

var

  MainForm: TMainForm;
```

```
implementation

{$R *.DFM}

const
  { Массив из двадцати строковых элементов, представляющих числа: }
  NumStrings: array[1..20] of String[10] =
      ( 'One',     'Two',       'Three',    'Four',     'Five',
        'Six',     'Seven',     'Eight',    'Nine',     'Ten',
        'Eleven',  'Twelve',    'Thirteen', 'Fourteen', 'Fifteen',
        'Sixteen', 'Seventeen', 'Eighteen', 'Nineteen', 'Twenty');
  MSFormatStr = '%dm %ds';
  PlayButtons: TButtonSet = [btPause, btStop, btNext, btPrev];
  StopButtons: TButtonSet = [btPlay, btNext, btPrev];
function TMainForm.TrackNumToString(InNum: Byte): String;
{ Эта функция возвращает строковое представление чисел от 1 до 20.
  Если номер больше 20, то выводится его числовое значение в виде строки. }
begin
  if (InNum > High(NumStrings)) or (InNum < Low(NumStrings)) then
    Result := IntToStr(InNum)   { Если значение выходит за пределы
                                  массива, то просто выводится число. }
  else
    Result := NumStrings[InNum]; { Возвращает строку из массива NumStrings. }
end;

procedure TMainForm.AdjustSpeedButtons;
{ Этот метод активизирует нужное количество кнопок. }
var
  i: integer;
begin
  { Итерация в массиве компонентов формы... }
  for i := 0 to SBPanel.ControlCount - 1 do
    if SBPanel.Controls[i] is TSpeedButton then // Это кнопка?
    { Отключает кнопки, ненужные для количества дорожек,
      присутствующих на данном компакт-диске. }
      with TSpeedButton(SBPanel.Controls[i]) do Enabled := Tag <= TotalTracks;
end;

procedure TMainForm.GetCDTotals;
{ Этот метод вычисляет количество дорожек, общее время
  записи на компакт-диске и отображает эту информацию. }
var
  TimeValue: longint;
begin
  mpCDPlayer.TimeFormat := tfTMSF; // Формат времени
  TimeValue := mpCDPlayer.Length; // Продолжительность записи
  TotalTracks := mci_Tmsf_Track(mpCDPlayer.Tracks); // Количество дорожек
```

```
  TotalLengthM := mci_msf_Minute(TimeValue); // Общее время записи в минутах
  TotalLengthS := mci_msf_Second(TimeValue); // Секунды
  { Обновление надписи Total Tracks }
  LblTotTrk.Caption := TrackNumToString(TotalTracks);
  { Обновление надписи Total Time }
  LblTotalLen.Caption := Format(MSFormatStr, TotalLengthM, TotalLengthS]);
  { Инициализация индикатора }
  ggDiskDone.MaxValue := (TotalLengthM * 60) + TotalLengthS;
  { Активизирует нужное количество кнопок }
  AdjustSpeedButtons;
end;

procedure TMainForm.ShowPlayerStatus;
{ Этот метод отображает информацию о статусе проигрывателя
  компакт-дисков и проигрываемого компакт-диска. }
begin
  if mpCDPlayer.EnabledButtons = PlayButtons then
    with LblStatus do
    begin
      case mpCDPlayer.Mode of
        mpNotReady: Caption := 'Not Ready';
        mpStopped:  Caption := 'Stopped';
        mpSeeking:  Caption := 'Seeking';
        mpPaused:   Caption := 'Paused';
        mpPlaying:  Caption := 'Playing';
      end;
    end
  { Эти кнопки отображаются, когда проигрыватель остановлен. }
  else if mpCDPlayer.EnabledButtons = StopButtons then
    LblStatus.Caption := 'Stopped';
end;

procedure TMainForm.ShowCurrentTime;
{ Этот метод показывает текущее время воспроизведения дорожки. }
begin
  { Минуты }
  m := mci_Tmsf_Minute(mpCDPlayer.Position);
  { Секунды }
  s := mci_Tmsf_Second(mpCDPlayer.Position);
  { Обновление надписи Track Time }
  LblTrackTime.Caption := Format(MSFormatStr, [m, s]);
  { Обновление индикатора дорожки }
  ggTrackDone.Progress := (60 * m) + s;
end;

procedure TMainForm.ShowTrackTime;
{ Этот метод отображает продолжительность текущей выбранной дорожки. }
```

```
var
  Min, Sec: Byte;
  Len: Longint;
begin
  { Не обновляет информацию, пока проигрыватель воспроизводит одну и ту же дорожку. }
  if CurrentTrack <> OldTrack then
  begin
    Len := mpCDPlayer.TrackLength[mci_Tmsf_Track(mpCDPlayer.Position)];
    Min := mci_msf_Minute(Len);
    Sec := mci_msf_Second(Len);
    ggTrackDone.MaxValue := (60 * Min) + Sec;
    LblTrackLen.Caption := Format(MSFormatStr, [m, s]);
  end;
  OldTrack := CurrentTrack;
end;

procedure TMainForm.HighlightTrackButton;
{ Эта процедура изменяет цвет шрифта на кнопке текущей дорожки
  на красный; у других кнопок этот цвет синий. }
var
  i: longint;
begin
  { Перебор компонентов формы }
  for i := 0 to ComponentCount - 1 do
    { это кнопка? }
    if Components[i] is TSpeedButton then
      if TSpeedButton(Components[i]).Tag = CurrentTrack then
        { Если это – номер текущей дорожки, то изменить цвет на красный. }
        TSpeedButton(Components[i]).Font.Color := clRed
      else
        { Если дорожка не текущая, изменить цвет на синий. }
        TSpeedButton(Components[i]).Font.Color := clNavy;
end;

procedure TMainForm.ShowTrackNumber;
{ Этот метод отображает номер текущей дорожки. }
var
  t: byte;
begin
  t := mci_Tmsf_Track(mpCDPlayer.Position); // Получить номер текущей дорожки
  CurrentTrack := t; // Инициализация переменной
  LblCurTrk.Caption := TrackNumToString(t); // Обновление надписи
  HighlightTrackButton; // Подсвечивание текущей кнопки
end;

procedure TMainForm.tmUpdateTimerTimer(Sender: TObject);
{ Этот метод обновляет всю информацию через установленный
```

```
  для таймера интервал времени }
begin
  if mpCDPlayer.EnabledButtons = PlayButtons then
  begin
    mpCDPlayer.TimeFormat := tfMSF;
    ggDiskDone.Progress := (mci_msf_minute(mpCDPlayer.Position) * 60 +
      mci_msf_second(mpCDPlayer.Position));
    mpCDPlayer.TimeFormat := tfTMSF;
    ShowTrackNumber; // Показывает текущий номер дорожки
    ShowTrackTime;   // Общее время текущей дорожки
    ShowCurrentTime; // Текущее время воспроизведения дорожки
  end;
end;

procedure TMainForm.mpCDPlayerPostClick(Sender: TObject; Button: TMPBtnType);
{ Этот метод позволяет правильно отобразить кнопки, если одна из них нажата. }
begin
  Case Button of
    btPlay:
      begin
       mpCDPlayer.EnabledButtons := PlayButtons;
       LblStatus.Caption := 'Playing';
      end;
    btPause:
      begin
        mpCDPlayer.EnabledButtons := StopButtons;
        LblStatus.Caption := 'Пауза';
      end;
    btStop:
      begin
        mpCDPlayer.Rewind;
        mpCDPlayer.EnabledButtons := StopButtons;
        LblCurTrk.Caption := 'One';
        LblTrackTime.Caption := '0m 0s';
        ggTrackDone.Progress := 0;
        ggDiskDone.Progress := 0;
        LblStatus.Caption := 'Stopped';
      end;
    btPrev, btNext:
     begin
       mpCDPlayer.Play;
       mpCDPlayer.EnabledButtons := PlayButtons;
       LblStatus.Caption := 'Playing';
      end;
  end;
end;
```

```
procedure TMainForm.FormCreate(Sender: TObject);
{ Этот метод вызывается при создании формы. Он открывает и
  инициализирует проигрыватель компакт-дисков. }
begin
  try
    mpCDPlayer.Open; // Открывает устройство проигрывателя компакт-дисков
    { Если компакт-диск уже воспроизводится, отобразить его статус. }
    if mpCDPlayer.Mode = mpPlaying then
      LblStatus.Caption := 'Playing';
    GetCDTotals;      // Общее количество дорожек и
                      // продолжительность записи на диске
    ShowTrackNumber;  // Текущая дорожка
    ShowTrackTime;    // Минуты и секунды текущей дорожки
    ShowCurrentTime;  // Текущая позиция компакт-диска
    ShowPlayerStatus; // Обновить статус проигрывателя
  except
    { Если произошла ошибка, вполне возможно, что система просто
      неспособна воспроизводить компакт-диски. }
    on EMCIDeviceError do
    begin
      MessageDlg('Error Initializing CD Player. Program will now exit.',
                 mtError, [mbOk], 0);
      Application.Terminate;
    end;
  end;
  { Проверяет текущий режим проигрывателя и активизирует
    соответствующие кнопки. }
    case mpCDPlayer.Mode of
    mpPlaying: mpCDPlayer.EnabledButtons := PlayButtons;
    mpStopped, mpPaused: mpCDPlayer.EnabledButtons := StopButtons;
  end;
  SplashScreen.Release; // Закрывает и освобождает окно заставки
end;

procedure TMainForm.sbTrack1Click(Sender: TObject);
{ Этот метод устанавливает текущую дорожку, когда пользователь щелкает
  на одной из кнопок дорожки. Метод анализирует все 20 кнопок, проверяя
  параметр 'Sender', и определяет, какая из них была выбрана. }
begin
  mpCDPlayer.Stop;
  { Новая стартовая позиция на компакт-диске для
    воспроизведения выбранной дорожки. }
  mpCDPlayer.StartPos := mpCDPlayer.TrackPosition[(Sender as TSpeedButton).Tag];
  { Начало воспроизведения с новой позиции }
  mpCDPlayer.Play;
  mpCDPlayer.EnabledButtons := PlayButtons;
  LblStatus.Caption := 'Playing';
```

```
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  mpCDPlayer.Close;
end;

end.
```

Резюме

В этой главе были рассмотрены основные принципы работы компонента Delphi `TMediaPlayer`. На нескольких конкретных примерах мы продемонстрировали мощь и простоту использования этого компонента. Кроме того, здесь содержится информация о работе с файлами наиболее распространенных мультимедиа-форматов (`WAVE`, `AVI`), а также со звуковыми компакт-дисками.

Отладка и тестирование

Глава

19

Некоторые программисты считают, что обширные знания и применение проверенной методологии программирования и разработки приложений сделают процедуру отладки излишней. Однако эти факторы отнюдь не исключают, а наоборот, дополняют друг друга. Человек, одинаково хорошо овладевший и тем, и другим, получает значительное преимущество. Это замечание особенно справедливо в тех случаях, когда над разными частями одной программы работают сразу несколько человек. Невозможно полностью исключить из расчетов вероятность совершения человеком ошибок.

Многие наивно полагают, что если их программа компилируется, то в ней нет ошибок. Чепуха! Нет никакой связи между компилируемостью программы и наличием в ней ошибок. Дело в том, что синтаксическая и логическая корректность кода — совершенно разные понятия. К тому же, если какая-то часть кода успешно функционировала раньше в той или иной программе, это вовсе не значит, что она свободна от программных ошибок. И будь программные ошибки (bug) настоящими жуками, множество легко компилирующихся и не имеющих ни единого замечания программ превратились бы в огромные термитники…

Естественно, компилятор может оказать большую помощь при разработке программы. Компилятор Delphi, например, позволяет включить различные режимы диагностики ошибок во вкладке Compiler диалогового окне Project⇨Options, показанной на рис. 19.1. Некоторые специальные директивы компилятора можно поместить непосредственно в текст программы. Кроме того, для получения дополнительной информации о коде полезно установить флажки опций Show Hints (Вывод советов) и Show Warnings (Вывод предупреждений), расположенных в той же вкладке диалогового окна свойств проекта. Зачастую программист тратит несколько часов в бесплодных поисках одной-единственной ошибки, которую компилятор, используя эти вспомогательные средства, смог бы найти немедленно. (Кстати, авторы данной книги постоянно пользуются этими вспомогательными средствами.)

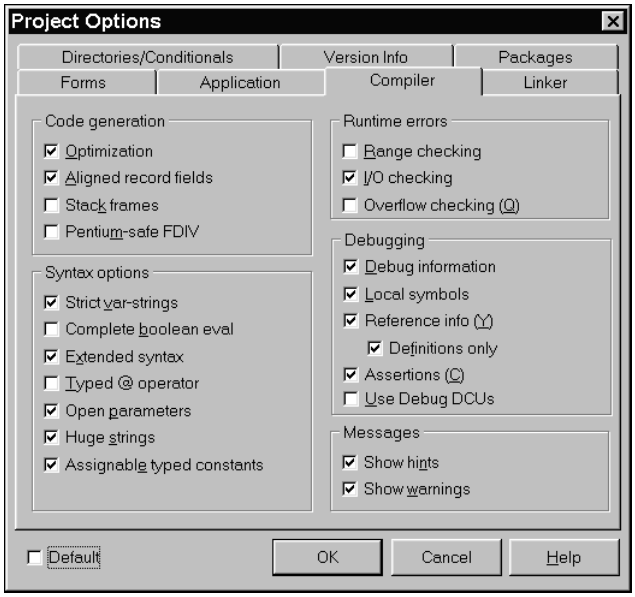


Рис. 19.1. Вкладка Compiler диалогового окна Project Options

В табл. 19.1 описаны различные опции контроля ошибок времени выполнения, существующие в Delphi.

Таблица 19.1. Классификация ошибок времени выполнения в Delphi

Проверка	Директива	Функция
Range Checking (Проверка диапазона)	`{$R+}`	Позволяет убедиться, что индекс массива, строки или значение скалярной переменной не вышли за пределы допустимого диапазона
I/O Checking (Контроль ввода-вывода)	`{$I+}`	Контролирует ошибки ввода-вывода после каждого вызова функций ввода-вывода, например, таких как `ReadLn()` и `WriteLn()`. Эту опцию желательно всегда включать
Overflow Checking (Контроль переполнения)	`{$Q+}`	Проверяет, не превышает ли результат вычислений размер регистра

На заметку

Имейте в виду, что эти директивы снижают производительность вашей программы, следовательно, когда отладка входит в финальную фазу и вы готовитесь передать программу заказчикам, можно повысить производительность, отменив некоторые из директив. Чаще всего отключают проверку всех ошибок, кроме ошибок ввода-вывода.

Наиболее распространенные программные ошибки

В этом разделе вы ознакомитесь с наиболее типичными случаями ошибок, приводящими к некорректной работе программного обеспечения. Зная, что именно следует прежде всего искать в тексте отлаживаемой программы, можно существенно сократить время, затрачиваемое на поиск ошибок.

Использование переменной класса без ее создания

Это одна из наиболее распространенных ошибок программирования в Delphi. Посмотрим на следующий фрагмент кода:

```
procedure Form1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList.Assign(ListBox1.Items);
end;
```

Класс `TStringList` переменной `MyStringList` объявлен, но экземпляр этого класса не создан, что и приведет к ошибке при обращении к этой переменной. Вы должны быть уверены в том, что все используемые переменные классов были получены путем создания экземпляра соответствующего класса. В следующем фрагменте показан корректный способ создания переменной, но в нем содержится другая ошибка. Сможете ли вы ее найти?

```
procedure Form1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
```

```
begin
  MyStringList := TStringList.Create;
  MyStringList.Assign(ListBox1.Items);
end;
```

Если ваш ответ: "Класс TStringList не освобожден", — поздравляем вас. Конечно, такая ошибка не приведет к сбою или аварийному завершению программы, но в итоге можно бесполезно израсходовать очень много памяти — ведь при каждом вызове этого метода в нем будет создаваться новый объект TStringList, вызывая постоянную утечку памяти. Хотя по завершении работы приложения Win32 API освободит всю используемую им память, утечка памяти во время работы приложения может привести к серьезным проблемам. Например, приложение с утечкой памяти в процессе своего выполнения будет потреблять все больше и больше системных ресурсов, вызывая непрерывное возрастание количества операций страничного обмена и снижая быстродействие всей системы.

Правильная версия предыдущего фрагмента кода выглядит следующим образом (пока без необходимых улучшений, которые будут внесены в следующем подразделе):

```
procedure Form1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create; // Создание
  MyStringList.Assign(ListBox1.Items); // Использование
  { Выполнение необходимых операций с экземпляром TStringList }
  MyStringList.Free; // Освобождение
end;
```

Убедитесь, что экземпляры класса освобождены

Предположим, что при выполнении программы из предыдущего примера сразу после создания экземпляра объекта TStringList возникла исключительная ситуация. Это привело к немедленному прекращению выполнения процедуры, и ее оставшаяся часть не была выполнена, что, в свою очередь, привело к утечке памяти. Для того чтобы освободить все экземпляры класса даже после генерации исключения, следует воспользоваться конструкцией try...finally, как показано ниже.

```
procedure Form1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create; // Создание
  try
    MyStringList.Assign(ListBox1.Items); // Использование
    { Выполнение необходимых операций с экземпляром TStringList }
  finally
    MyStringList.Free; // Освобождение
  end;
end;
```

Прочитав раздел “Точки останова” (далее в этой главе), попытайтесь продолжить эксперимент и поместите следующую строку сразу же после строки, в которой классу `TStringList` присваивается элемент `ListBox1`:

```
raise Exception.Create('Проверка на исключение');
```

Затем поместите точку останова в начало кода метода и пошагово выполняйте его. Вы увидите, что экземпляр объекта `TStringList` освобождается даже после генерации исключения.

Приручение „диких“ указателей

Дикий указатель — типичная ошибка, приводящая к разрушению данных в памяти, когда эти указатели используются для выполнения операции записи в память. Существует две разновидности “диких” указателей: неинициализированный и устаревший.

Неинициализированный указатель может появиться, если переменная указателя используется до распределения памяти для него. При использовании подобных указателей запись выполняется в ту область памяти, адрес которой случайно оказался в поле, отведенном под указатель. Вот пример использования неинициализированного указателя:

```
var
  P: ^Integer;
begin
  P^ := 1971; // А ведь указатель P еще не инициализирован!
```

Устаревший указатель ссылается на область памяти, которая когда-то была правильно распределена, а затем освобождена:

```
var
  P: ^Integer;
begin
  New(P);
  P^ := 1971;
  Dispose(P);
  P^ := 4; // Ошибка! Указатель P устарел!
```

Если вам повезет, при попытке записи по “дикому” указателю будет получено сообщение об ошибке доступа. А если нет — произойдет перезапись (т.е. разрушение) данных, используемых другой частью приложения. Для отладки программы с такой ошибкой нередко требуются поистине колоссальные усилия. На одной машине указатель может адресовать безобидную область памяти и все будет прекрасно работать, тогда как при переносе приложения на другую машину (возможно, с внесением незначительных изменений) произойдет сбой. Логично будет предположить, что сбой вызван именно последними изменениями в исходном коде или же проблемами с аппаратным обеспечением на второй машине. Если попасться в эту ловушку, то никакие, даже самые лучшие методологии программирования не смогут вам помочь. В попытке решить проблему вы начнете добавлять вызовы функции `ShowMessage()` к различным участкам кода, но это только изменит размещение кода в памяти, и ошибка проявится в другом месте программы или, что еще хуже, вообще таинственным образом исчезнет! Лучший способ борьбы с “дикими” указателями состоит в самой тщательной профилактике их появления еще на этапе написания кода. Если вам нужно работать с указателями и непосредственным распределением памяти, тщательно (и не один раз) проверьте алгоритмы на наличие в них этой довольно распространенной ошибки.

Использование неинициализированных переменных типа PChar

Очень часто "дикие" указатели появляются при работе с переменными типа PChar. Поскольку тип PChar представляет собой обыкновенный указатель на строку символов, не забывайте распределять память для переменных этого типа с помощью функции StrAlloc(), GetMem(), StrNew(), GlobalAlloc() или VirtualAlloc(), а также не забывайте освобождать эту память с помощью GlobalFree() или VirtualFree().

Вы сможете избежать потенциальных ошибок в программе, работая (везде, где только это возможно) с переменными типа string, а не PChar. Тип string всегда можно привести к типу PChar — там, где это необходимо. В остальных случаях использование обычных строк только упростит код, поскольку строковые переменные автоматически размещаются в памяти и так же освобождаются, что избавит вас от усилий, связанных с распределением памяти.

Очень важно помнить об этом при переносе приложения Delphi 1.0 в Delphi последующих версий. В Delphi 1.0 переменные типа PChar — это неизбежное зло. В любой 32-разрядной Delphi их применение оправдано лишь в редких случаях. Не пожалейте времени на замену переменных типа PChar на string при переносе приложения в 32-разрядную Delphi — это обязательно оправдает себя.

Разыменовывание указателя со значением nil

Еще одной из самых распространенных ошибок является разыменовывание указателя со значением nil (указателя со значением 0). Это действие операционная система всегда расценивает как ошибку доступа. Хотя вы, конечно, не захотите сохранить в своем приложении такую ошибку, она не столь страшна, так как при этом не портится память, а для продолжения выполнения программы можно организовать соответствующую обработку исключительных ситуаций. Продемонстрируем это следующей процедурой:

```
procedure I_AV;
var
  P: PByte;
begin
  P := Nil;
  try
    P^ := 1;
  except
    on EAccessViolation do
      MessageDlg('Этого делать нельзя!', mtError, [mbOk], 0);
  end;
end;
```

Если поместить эту процедуру в программу, то при ее выполнении появится диалоговое окно с данным сообщением, но программа продолжит свою работу.

Использование встроенного отладчика

В Delphi существует мощный отладчик, встроенный непосредственно в интегрированную среду разработки (IDE). Команды вызова большинства его функций находятся в меню Run. Набор поддерживаемых им функций включает все, что можно ожидать от отладчика: задание параметров командной строки при запуске приложения, трассировку и пошаговое выполнение, установку точек останова, добавление и просмотр контролируемых значений, вычисление и модификацию данных, а также просмотр содержимого стека.

Использование параметров командной строки

Если отлаживаемая программа использует параметры командной строки, их можно задать в диалоговом окне Run Parameters. В этом окне параметры вводятся точно так же, как в командной строке или в диалоговом окне Windows Start⇨Run.

Точки останова

Точки останова (*breakpoints*), или просто *остановы*, позволяют при выполнении определенных условий приостановить работу программы. Чаще всего точки останова размещаются в определенной строке кода, при этом останов происходит в тот момент, когда данная строка должна начать выполняться. Такой останов можно установить, щелкнув слева от строки кода в окне Code Editor, выбрав соответствующую команду в контекстном меню или выбрав команду Run⇨Add Breakpoint. Если необходимо проанализировать поведение программы внутри определенной процедуры или функции, достаточно просто установить точку останова в ее первой строке. Пример установки такой точки останова показан на рис. 19.2.

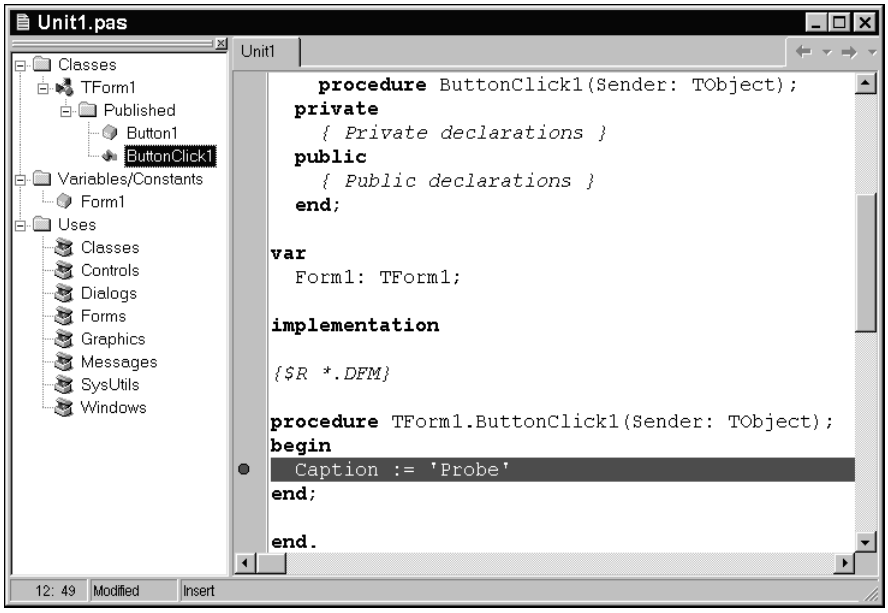


Рис. 19.2. Установка точки останова в окне Code Editor

Условная точка останова

Можно задать дополнительное условие к точке останова, и тогда программа будет приостанавливаться по достижении определенной строки кода только при выполнении этого условия. Типичным случаем применения такой точки останова будет проверка кода внутри цикла. Вероятно, вы не захотите останавливать и опять запускать программу всякий раз, когда выполняется цикл — а это происходит сотни, а то и тысячи раз. Вместо того чтобы непрерывно нажимать клавишу <F9> для повторного запуска программы, просто установите точку останова по достижении какой-то переменной определенного значения. Например, в новом проекте поместите в главную форму объект TButton и добавьте следующий код в программу обработчика событий кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 100 do
  begin
    Caption := IntToStr(I); // Обновление формы
    Button1.Caption := IntToStr(I); // Обновление кнопки
    Application.ProcessMessages; // Произвести обновления
  end;
end;
```

А теперь поместите точку останова в следующую строку:

```
Caption := IntToStr(I); // Обновление формы
```

После этого выберите команду View⇨Debug Windows⇨Breakpoints, открывающую диалоговое окно Breakpoint List. Точка останова должна быть показана в нем. Щелкните в окне правой кнопкой мыши и в раскрывшемся контекстном меню выберите команду Properties. На экран будет выведено диалоговое окно Source Breakpoint Properties, показанное на рис. 19.3. В поле Condition введите значение I=50 и щелкните на кнопке OK. В результате установленная ранее точка останова приостановит выполнение программы только, когда переменная I станет равной 50.

На рис. 19.3 представлены новые возможности работы с точками останова, которые появились только в Delphi 5. Все они объединены в группу Action (Действие) и позволяют точно определить поведение отладчика при достижении программой данной точки останова. Тип требуемых действий задается с помощью трех флажков опций, показанных на рисунке. Опция Break, как можно догадаться, предписывает отладчику приостановить выполнение программы, когда та достигнет данной точки останова. Опция Ignore Subsequent Exception (Игнорировать последующие исключения) требует от отладчика воздержаться от остановов, если в ходе выполнения программы после точки останова будут обнаружены исключительные ситуации. Опция Handle Subsequent Exception требует от отладчика сохранить принимаемое по умолчанию поведение в отношении точек останова, если в ходе выполнения программы после точки останова будут обнаружены исключительные ситуации.

Две последние опции целесообразно использовать в паре. Например, пусть в программе имеется определенный блок операторов, который мешает отладке из-за возникновения в нем некоторой исключительной ситуации. Можно поместить в начало этого блока точку останова и установить флажок блокировки исключительных ситуаций после ее прохождения. В последний оператор данного блока помещается вторая точка останова, восстанавливающая нормальную обработку исключений отладчиком.

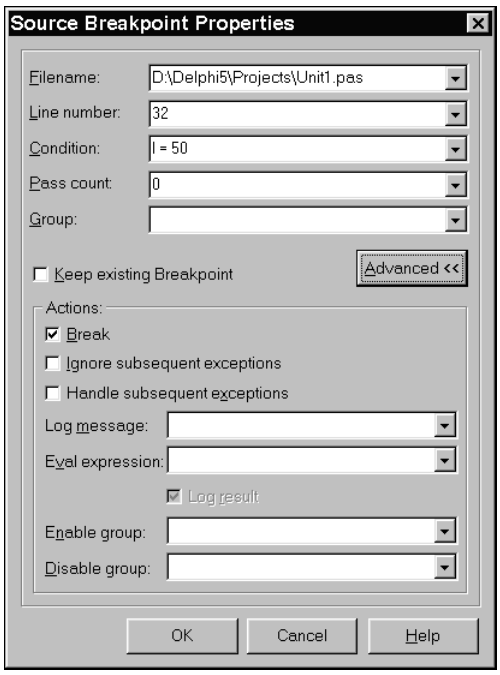


Рис. 19.3. Диалоговое окно Source Breakpoint Properties

Точка останова по обращению к данным

Этот тип точки останова приостанавливает выполнение программы при модификации определенного участка памяти. Он применяется для низкоуровневой отладки, когда отслеживаются ошибки присвоения значений переменным. Точку останова по данным можно установить, выбрав в главном меню команду Run⇨Add Breakpoint⇨Data Breakpoint или в контекстном меню окна Breakpoint List команду Add⇨Data Breakpoint. На экране появится диалоговое окно Add Data Breakpoint, показанное на рис. 19.4. В этом окне можно ввести начальный адрес области памяти, которую требуется контролировать, и ее размер (в байтах). Устанавливая необходимый размер, можно контролировать переменную любого типа — от `Char` (1 байт) и `Integer` (4 байта) до массива или записи произвольного размера. Как и в случае точки останова по условию, в диалоговом окне Add Data Breakpoint можно ввести выражение, которое будет критерием останова при изменении области памяти. Это позволяет выявлять ошибки, которые проявляются только после n-го обновления значения переменной. Если требуется останавливать программу при изменении значения определенной переменной, достаточно ввести ее имя в поле Address.

Точка останова по адресу

Останов по адресу реализуется, когда выполняется инструкция, находящаяся в памяти по заданному адресу. Эти точки обычно устанавливаются из контекстного меню в окне CPU, если невозможно установить обычную точку останова в определенной строке кода из-за отсутствия исходного кода для соответствующего модуля. Как и в случае других видов точек останова, можно задать условия приостановки выполнения программы.

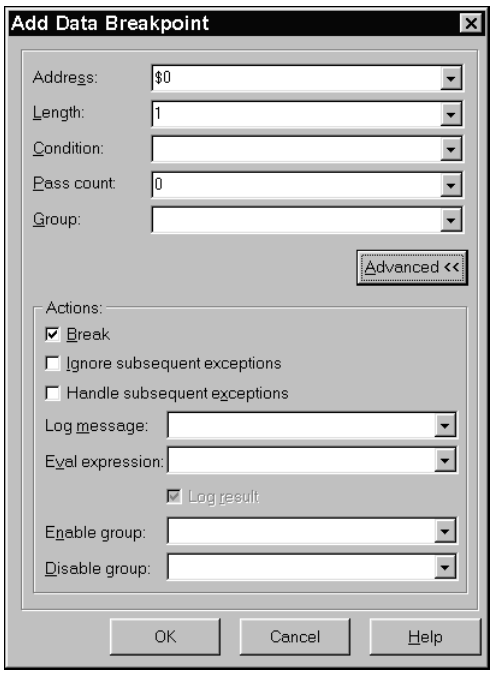


Рис. 19.4. Диалоговое окно Add Data Breakpoint

Точка останова по загрузке модуля

Как следует из названия, точка останова по загрузке модуля останавливает выполнение отлаживаемой программы при загрузке указанного модуля. Как правило, эти точки устанавливают из контекстного меню в окне Modules, хотя их можно установить и из главного меню с помощью команды Run⇨Add Breakpoint⇨Module Load Breakpoint.

Группы точек останова

Группы точек останова представляют собой одну из самых мощных и эффективных функций интегрированного отладчика. С помощью механизма групп, любая точка останова может быть активизирована или деактивизирована другой точкой останова, что позволяет строить весьма сложные алгоритмы функционирования групп точек останова, предназначенные для обнаружения самых трудноуловимых и специфических ошибок. Допустим, предполагается наличие ошибки в методе `Paint()`, которая проявляется только после выбора определенной команды меню. Можно попытаться решить проблему "в лоб", т.е. поместить точку останова в метод `Paint()`, запустить программу и затем методично возобновлять ее выполнение при каждом из сотен вызовов данного метода. Но существует более удобный вариант. Точка останова в методе `Paint()` сохраняется, но объявляется неактивной, и, следовательно, обрабатываться не будет. Затем новая точка останова добавляется в метод обработки выбора требуемой команды меню, которая активизирует точку останова в методе `Paint()`. Теперь приложение будет выполняться с нормальной скоростью, и отладчик остановит его работу только при первом обращении к методу `Paint()` после выбора нужной команды меню.

Пошаговое выполнение программы

Программу можно выполнять последовательно, строка за строкой, используя команды Step Over или Trace Into (по умолчанию клавиши <F8> и <F7> соответственно). Команда Trace Into при выполнении программы обеспечивает вход в вызываемые процедуры и функции, а команда Step Over немедленно их выполняет и представляет как одно действие. Этими опциями удобно пользоваться после останова программы в каком-то месте ее текста. Запомните клавиши <F8> и <F7> — они вам обязательно пригодятся!

Можно также дать указание Delphi выполнять программу до того места, в котором в настоящий момент находится курсор, — с помощью команды Run to Cursor (клавиша <F4>). Эту возможность удобно использовать для пропуска многократно выполняющегося цикла, чтобы непрерывно не нажимать клавиши <F8> и <F7>. Не забывайте, что установить точки останова в окне редактора кода можно в любой момент выполнения программы, а поэтому нет необходимости заранее устанавливать их сразу все.

Если при отладке вы случайно вошли в тело функции, которую трудно или слишком долго выполнять в пошаговом режиме, выберите в главном меню команду Run⇨Run Until Return. Отладчик остановит программу сразу после завершения работы данной процедуры или функции.

Используя команду Run⇨Program Pause, работу программы можно остановить динамически в любой момент ее выполнения. Эта команда часто помогает определить, не зациклилась ли данная программа. Однако нужно помнить, что при выполнении любых программ большую часть времени работает код подпрограмм библиотеки VCL, так что очень часто при динамическом останове вас может "выбросить" за пределы текста вашей программы.

Отлаживая ваши приложения, вы, вероятно, уже обратили внимание на синие точки, расположенные слева в окне редактора кода. Каждая из этих точек находится возле такой строки программы, для которой генерируется машинный код. Нельзя установить точку останова или пошагово выполнить строку исходного кода, если она не отмечена подобной точкой, поскольку с этой строкой не связан никакой машинный код.

Использование окна Watch

Окно Watch можно использовать для контроля значений переменных по ходу выполнения программы. Имейте в виду, что программа должна находиться в режиме просмотра кода программы (т.е. выполняться какая-либо из точек останова — только в этом случае содержимое окна Watch будет корректным. В этом окне можно ввести некоторое выражение Object Pascal или указать определенное в программе имя, как показано на рис. 19.5.

Инспекторы отладки

Инспекторы отладки представляют собой разновидность окна инспектора данных. Они обладают б*о*льшими возможностями, и их легче использовать, чем окно Watch. Для вызова инспектора во время отладки воспользуйтесь командой Run⇨Inspect. На экране раскроется простое диалоговое окно, в котором можно ввести требуемое выражение. Щелкните на кнопке OK, после чего раскроется окно инспектора отладки для заданного выражения. Например, на рис. 19.6 показан инспектор отладки для пустого приложения Delphi.

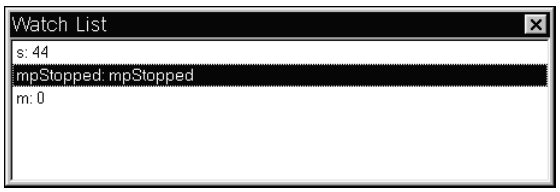


Рис. 19.5. Использование окна Watch

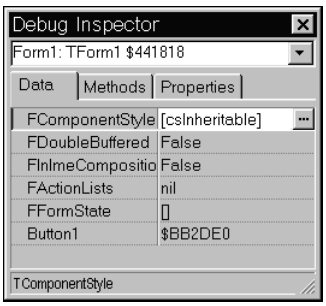


Рис. 19.6. Инспектор отладки для формы

В окне инспектора отладки можно просматривать содержимое данных, состоящих из множества индивидуальных элементов, например, таких как классы и записи. Щелкните справа от столбца значений для изменения содержимого поля. Можно добраться даже до каждого члена записи или класса — с помощью двойного щелчка на соответствующем поле в списке.

Использование команд Evaluate и Modify

Команды Evaluate и Modify позволяют соответственно просматривать и изменять содержимое переменных, включая массивы и записи, "на лету", во время выполнения приложения в интегрированном отладчике. (Однако они не предоставляют доступ к функциям и переменным вне области видимости.)

Вычисление и изменение переменных, вероятно, наиболее мощная функция интегрированного отладчика, однако не забывайте, что она налагает на вас ответственность за непосредственный доступ к памяти. Вы должны быть очень осторожны при изменении значений переменных, так как это может привести к непредсказуемым результатам.

Доступ к стеку вызовов

Получение доступа к стеку вызовов достигается с помощью команды View⇨Debug Windows⇨Call Stack. Она позволяет просмотреть вызовы функций и процедур вместе с переданными им параметрами, в той последовательности, в которой они выполнялись до определенного момента выполнения программы. На рис. 19.7 показан типичный вид содержимого окна Call Stack.

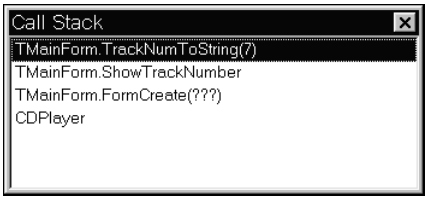


Рис. 19.7. Окно Call Stack

Чтобы просмотреть процедуру или функцию в окне Call Stack, просто дважды щелкните в окне на соответствующей строке. Это отличный способ вернутся к функции, которую вы случайно пропустили в процессе пошаговой отладки.

Просмотр потоков

Если приложение использует множество потоков, встроенный отладчик позволяет получить информацию о каждом из них в окне Thread Status, вызываемом с помощью команды главного меню View⇨Debug Windows⇨Threads. Во время паузы в выполнении приложения (например, после точки останова) можно воспользоваться контекстным меню в этом окне для того, чтобы сделать текущим другой поток или просмотреть код, ассоциированный с конкретным процессом. Помните, что, если вы измените текущий поток, выполнение программы продолжится с нового места! Вид окна Thread Status показан на рис. 19.8.

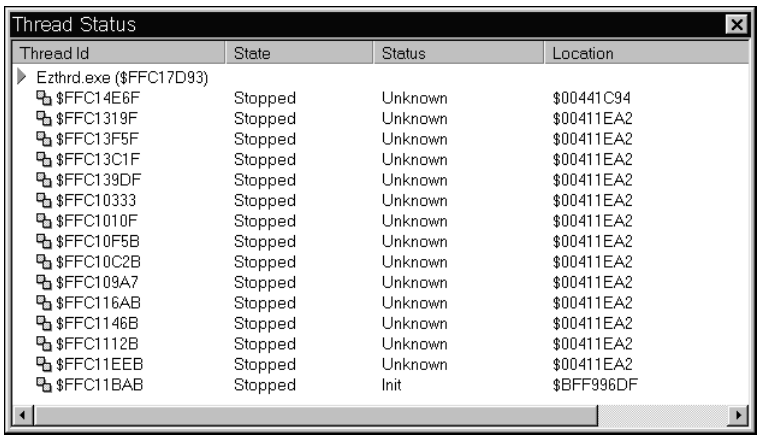


Рис. 19.8. Окно Thread Status

Журнал событий (Event Log)

Журнал событий — это место, в которое отладчик записывает информацию о различных событиях. Окно Event Log (рис. 19.9) вызывается по команде View⇨Debug Windows⇨Event Log. Для настройки окна можно использовать его контекстное меню или вкладку Debugger диалогового окна Environment Options, вызываемого по команде Tools⇨Environment Options.

Типы событий, сведения о которых могут помещаться в журнал, включают запуск и останов процесса, загрузку модулей отладчиком, направленные в приложение сообщения Windows и данные, выводимые приложением с помощью функции `OutputDebugString()`.

Функция Win32 API `OutputDebugString()` — удобное средство отладки приложений. Единственным ее параметром является переменная типа `PChar`. Строка-параметр направляется в отладчик и (в случае Delphi) добавляется в журнал событий `Event Log`. Это позволяет контролировать значения переменных и получать подробные отладочные сведения без использования информационно-избыточных окон отладчика.

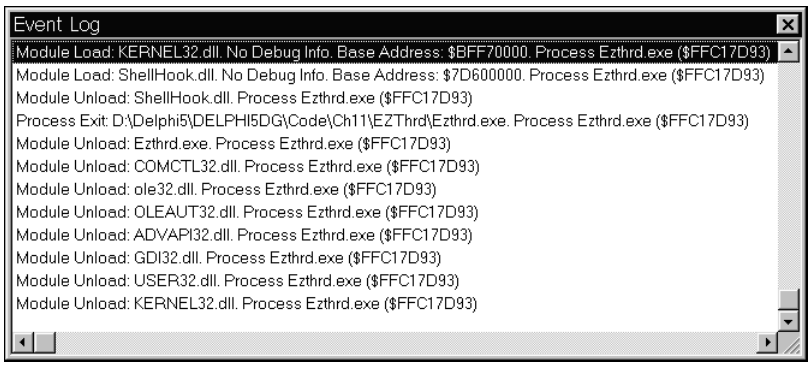


Рис. 19.9. Окно Event Log

Просмотр модулей

Функция просмотра модулей позволяет получить информацию обо всех используемых модулях (`EXE`, `DLL`, `BPL` и т.д.), загружаемых во время отладки приложением. Соответствующее окно Modules показано на рис. 19.10. Кроме подробной информации о модулях, оно также позволяет установить точки останова по загрузке модулей.

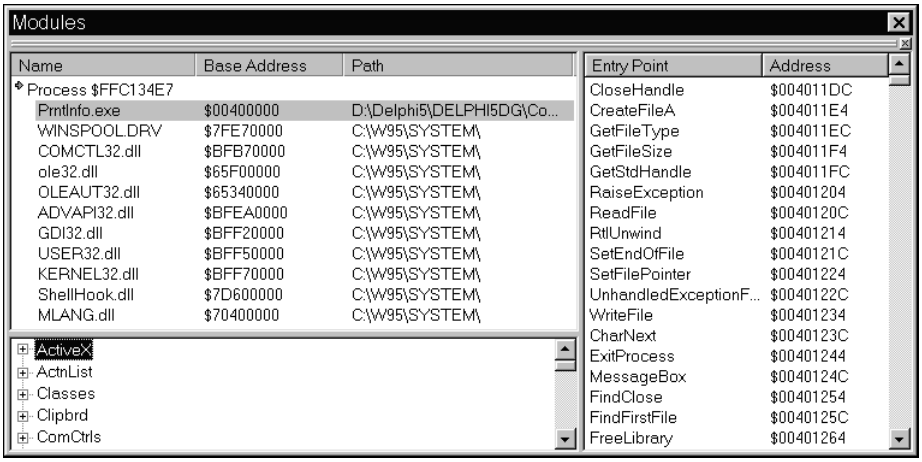


Рис. 19.10. Окно Modules

Отладка DLL

В интегрированном отладчике существует возможность отладки динамически компонуемых библиотек (DLL). При этом в качестве *основного* (*host*) можно использовать любое приложение. Это довольно просто: откройте вашу библиотеку и выберите команду Run⇨Parameters главного меню. Затем определите основное приложение в появившемся диалоговом окне (рис. 19.11).

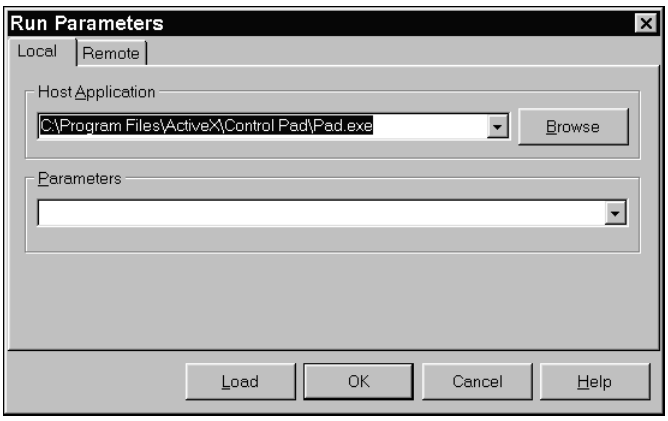


Рис. 19.11. Задание основного приложения для отладки DLL

Основное приложение представляет собой выполняемый файл, загружающий и использующий отлаживаемую библиотеку. После задания основного приложения вы можете воспользоваться всеми возможностями отладчика, доступными при отладке обычного выполняемого файла: установкой точек останова, пошаговым выполнением, трассировкой и т.п.

Эта возможность наиболее эффективна при отладке элементов ActiveX и незавершенных COM-серверов, запускаемых из контекста другого процесса. Например, эту функцию можно использовать для отладки элемента ActiveX, включенного в приложение Visual Basic.

Окно CPU

Окно CPU вызывается с помощью команды View⇨Debug Windows⇨CPU и содержит пять информационных панелей: CPU, Memory Dump, Register, Flags и Stack (рис. 19.12). С его помощью разработчик получает возможность узнать, что именно происходит в машине. Каждая из панелей позволяет во время отладки следить за важными аспектами функционирования процессора.

В панели CPU отображаются коды операций и мнемоники дизассемблированного кода, выполняемого в данный момент. Вы можете просмотреть код своего приложения по любому адресу и произвольно выбрать новую инструкцию для текущего выполнения. Это позволит изучить поведение ассемблерного кода вашей программы. Опытным разработчикам прекрасно известно, что множество ошибок находится и устраняется при проверке сгенерированного ассемблерного кода. Те же, кто не знаком с языком ассемблера, не в состоянии быстро справиться с подобными ошибками.

Контекстное меню окна CPU позволяет настроить внешний вид окна, просмотреть различные адреса, перейти к инструкции, выполняемой в данный момент, осуществить поиск, вернуться к просмотру исходного кода и т.п. Кроме того, можно выбрать контекст потока, в котором будет просматриваться информация CPU.

На панели Memory Dump можно просмотреть содержимое любой области памяти. Это содержимое может быть представлено по-разному: как `Byte`, `Word`, `DWORD`, `QWORD`, `Single`, `Double` или `Extended`. Можно выполнить поиск в памяти определенной последовательности байтов, модифицировать текущие данные и перейти к следующим, либо последовательно, либо используя текущие данные в качестве указателя.

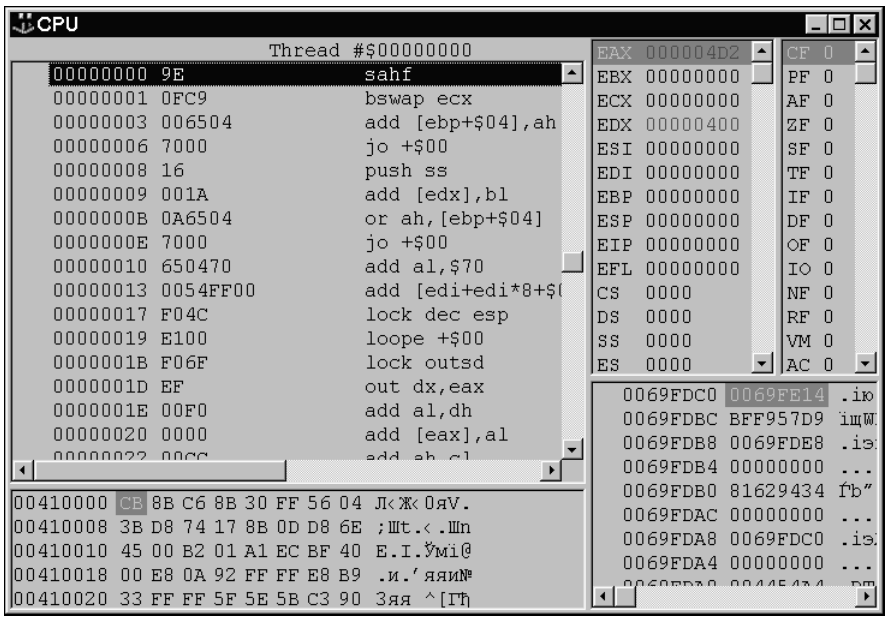


Рис. 19.12. Окно CPU

Назначение панелей Register и Flags достаточно очевидно: в них отображаются и могут быть изменены все регистры и флаги процессора.

На панели Stack можно просмотреть память, используемую приложением в качестве стека. Можете также изменять значения и перемещаться по адресам.

Резюме

В этой главе вам было предложено совершить предельно краткий экскурс по этапам отладочного процесса. Рассматривались чаще всего встречающиеся при отладке проблемы и обсуждались особенности встроенного отладчика. Важно помнить, что отладка — такая же часть программирования, как и написание кода. Отладчик будет вашим верным союзником при создании корректного кода, так что найдите время изучить его.

В следующей части книги, размещенной в отдельном томе, открывается завеса над загадочной областью программирования, в которой используются компоненты COM и VCL.

Предметный указатель

.

$

A

B

C

D

F

G

I

M

N

O

P

Q

R

S

T

U

V

W

Z

А

Б

В

Г

Д

Е

Ж

З

И

К

Л

М

H

O

П

P

C

T

У

Ф

X

Ц

Ш