По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-074-X, название «Delphi. Профессиональное программирование» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

HIGH TECH

Delphi

Профессиональное программирование

Дмитрий Осипов

СИМВОЛ®

Санкт-Петербург — Москва
2006

Серия «High tech»

Дмитрий Осипов

Delphi. Профессиональное программирование

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Редактор	*А. Петухов*
Художник	*В. Гренда*
Корректор	*О. Макарова*
Верстка	*Н. Гриценко*

Осипов Д.
Delphi. Профессиональное программирование. – СПб.: Символ-Плюс, 2006. – 1056 с., ил.
ISBN 5-93286-074-X

Книга Д. Осипова «Delphi. Профессиональное программирование» принципиально отличается от стандартных изданий на эту тему. Это и не скороспелое «полное» руководство по очередной версии Borland® Delphi™, и не рядовой справочник, содержащий перевод файлов помощи к среде программирования. Идея книги в другом. Автор системно и последовательно излагает концепцию Delphi, предоставляя читателю не просто инструмент, а профессиональную методику, позволяющую разрабатывать эффективные приложения для Windows.

Книга рассчитана на подготовленного пользователя ПК, желающего самостоятельно научиться программировать и разрабатывать приложения и базы данных в среде Delphi. Опытные программисты смогут использовать издание как справочник. В тексте подробно описаны более 80 компонентов VCL, функции Object Pascal и Win32 API. В первой части книги излагаются основы языка программирования Delphi, подробно рассматриваются библиотека визуальных компонентов и процесс разработки собственных компонентов, изучаются динамически подключаемые библиотеки, процессы, многопоточные приложения, особенности межпрограммного взаимодействия, программирование на Win32 API, особенности построения сетевого программного обеспечения, технологии COM и OLE-automation. Вторая часть книги посвящена проектированию и созданию реляционных баз данных. Рассматриваются реляционная модель данных и язык SQL, изучаются компоненты доступа к данным и отображения данных, базирующиеся на механизмах BDE, ADO и InterBase.

ISBN 5-93286-074-X

© Дмитрий Осипов, 2006
© Издательство Символ-Плюс, 2006

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 30.03.2006. Формат 70x100 1/16 . Печать офсетная.
Объем 66 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Введение 11

Часть I. Программирование для Windows в среде Delphi

1. Язык программирования Pascal 14
Простейшая программа на Object Pascal 15
Основные типы данных 19
Операторы и выражения 35
Резюме 43

2. Процедуры и функции 44
Процедуры 44
Функции 45
Особенности объявления и передачи параметров 46
Перегрузка методов 49
Структура программного модуля стандартного проекта Delphi 49
Резюме 51
Приложение 1: файлы проекта Delphi 51
Приложение 2: русификация консольных приложений 52

3. Базовые функции Delphi 55
Математические функции и процедуры 55
Функции проверки вхождения значения в диапазон 56
Тригонометрические функции и процедуры 57
Финансовые функции и процедуры 58
Статистические функции и процедуры 59
Процедуры и функции для работы со строками типа AnsiString 61
Процедуры и функции для работы со строками типа PChar 65
Работа с памятью 68
Процедуры управления ходом выполнения программы 69
Разные функции 70
Резюме 71

4. Основы работы с файлами 72
Классификация типов файлов 73
Низкоуровневые методы работы с файлами 89
Управление файлами, дисками и каталогами 91
Резюме 101

5. Введение в объектно-ориентированное программирование ... 102
Объект и класс ... 103
Инкапсуляция ... 107
Наследование ... 109
Полиморфизм ... 111
Программирование, управляемое событиями ... 112
Резюме ... 112

6. Невидимые классы ... 113
Основа основ – класс TObject ... 115
Класс TPersistent ... 119
Поток – TStream ... 120
Основа компонента – класс TComponent ... 121
Элемент управления – класс TControl ... 125
Оконный элемент управления – класс TWinControl ... 132
Обработка событий в классах TControl и TWinControl ... 136
Основа графических элементов управления – класс TGraphicControl ... 147
Резюме ... 148

7. Списки и коллекции ... 149
Набор строк – TStrings ... 150
Список – TList ... 152
Список строк – TStringList ... 154
Список объектов – класс TObjectList ... 155
Список компонентов – класс TComponentList ... 157
Коллекция – класс TCollection ... 157
Резюме ... 161

8. Стандартные компоненты ... 162
Компоненты для редактирования текста ... 162
Кнопки ... 172
Элементы управления – списки ... 179
Сетки ... 187
Меню ... 195
Резюме ... 207

9. Форма, интерфейсы SDI и MDI ... 208
Форма – TForm ... 208
Интерфейсы SDI и MDI ... 222
Приложение – класс TApplication ... 227
Особенности обработки событий в приложении и компонент TApplicationEvents ... 235
Экран – класс TScreen ... 235
Резюме ... 238

10. Графическая подсистема ... 239
Представление цвета в Windows ... 240
Перо – класс TPen ... 241
Кисть – класс TBrush ... 243
Шрифт – класс TFont ... 244

Холст – класс TCanvas . . . 246
Класс TGraphic . . . 254
Пиктограмма – класс TIcon . . . 257
Растровое изображение – класс TBitmap . . . 258
Метафайл – класс TMetafile . . . 261
Класс TJPEGImage . . . 263
Универсальное хранилище изображений – класс TPicture . . . 265
Графические компоненты VCL . . . 266
Работа с графикой методами Win32 API . . . 270
Резюме . . . 280

11. Компоненты Win32 . . . 281

Список закладок – TTabControl . . . 281
Блокнот – компонент TPageControl . . . 285
Иерархическая структура – TTreeView . . . 287
Графический список – TListView . . . 302
Панель инструментов – TToolBar . . . 313
Панель состояния – TStatusBar . . . 318
Линейка – TCoolBar . . . 321
Полоса управления – TControlBar . . . 323
Шкала – TTrackBar . . . 326
Резюме . . . 327

12. Для тех, кто ценит секунды . . . 328

Представление даты и времени в Delphi . . . 328
Процедуры и функции для работы с датой и временем . . . 329
Функции конвертирования даты и времени в другие типы данных . . . 331
Форматирование даты и времени . . . 332
Операционная система и таймер . . . 334
Таймер – компонент TTimer . . . 336
Компоненты-календари – базовый класс TCommonCalendar . . . 337
Резюме . . . 341

13. Работа с файлами инициализации и реестром Windows . . . 342

Файл инициализации – класс TIniFile . . . 342
Реестр Windows . . . 346
Низкоуровневый доступ к реестру – класс TRegistry . . . 349
Резюме . . . 355

14. Диалог с Microsoft® Windows® . . . 356

Диалоговые окна сообщений . . . 356
Диалог выбора каталога . . . 362
Диалоги доступа к базе данных . . . 363
Стандартные диалоговые окна Windows . . . 363
Резюме . . . 381

15. Обработка исключительных ситуаций . . . 382

Защищенные от ошибок секции . . . 383
Исключительные ситуации библиотеки VCL – класс Exception . . . 386
Принудительный вызов ИС – команда Raise . . . 393

Расширенные возможности конструкции try .. except 394
Обработка ИС в рамках события OnException приложения TApplication 395
Настройка поведения Delphi при обработке ИС 397
Резюме 398

16. Создание компонентов 399

Выбор предка 400
Эксперт компонентов 401
Шаблон кода компонента 401
Создание свойств 402
Создание методов 414
Создание событий 423
Пиктограмма компонента 427
Подключение файла справки к компоненту 428
Резюме 429

17. Централизованное управление приложением 430

Команда – класс TAction 431
Компоненты-контейнеры для командных объектов 436
Список команд – класс TActionList 438
Менеджер команд – класс TActionManager 439
Менеджер команд и компоненты пользовательского интерфейса 446
Резюме 454

18. Построение диаграмм 455

Компонент TChart 455
Резюме 477

19. Динамически подключаемые библиотеки 478

Назначение DLL 478
Создание шаблона динамической библиотеки в Delphi 481
Взаимодействие динамической библиотеки с проектом 489
Создание библиотеки ресурсов 494
Анализ DLL 495
Резюме 496

20. Процессы и потоки в среде Windows 497

Процессы и многозадачность 497
Понятие потока, многопоточность 507
Элементарный поток – класс TThread 508
Пример простого многопоточного приложения 513
Синхронизация процессов и потоков 517
Резюме 527

21. Службы Microsoft Windows NT 528

Администрирование служб в Windows NT 528
Управление службами из внешних приложений 529
Инкапсуляция системной службы в VCL – класс TService 538
Приложение-служба – класс TServiceApplication 547
Пример проекта службы 548

Советы по отладке системной службы . . . 549
Резюме . . . 550

22. Обмен данными между процессами . . . 551

Буфер обмена – класс TClipboard . . . 551
Обмен сообщениями между процессами . . . 559
Динамический обмен данными . . . 563
Файлы, отображаемые в память . . . 577
Резюме . . . 580

23. Обмен данными в сети . . . 581

Модель взаимодействия открытых систем . . . 582
Почтовые слоты . . . 583
Место класса THandleStream в обеспечении сетевого обмена данными . . . 587
Введение в Network DDE . . . 590
Каналы . . . 591
Интерфейс сокетов . . . 604
Реализация интерфейса WinSock в VCL . . . 607
Пример проекта WinSock для сети интранет . . . 618
Сокет – TRawSocket . . . 623
Резюме . . . 623

24. Многокомпонентная модель объектов (COM) . . . 624

Элементы COM-приложения . . . 625
COM-объект . . . 626
Интерфейс . . . 628
Порядок вызова сервера клиентским приложением . . . 631
Реализация COM-объекта в Delphi – класс TComObject . . . 636
Пример COM-проекта . . . 636
Резюме . . . 647
Приложение: редактор библиотеки типов . . . 647

25. Сотрудничество с Microsoft® Office . . . 650

Интерфейс IDispatch . . . 651
Инициализация и деинициализация объекта автоматизации . . . 652
Коллекция объектов . . . 653
Текстовый процессор Microsoft® Word . . . 655
Пример универсального генератора отчетов . . . 674
Электронные таблицы Microsoft® Excel . . . 676
Пример универсального генератора отчетов (продолжение) . . . 692
Резюме . . . 693

26. Связывание и внедрение объектов – технология OLE . . . 694

Место OLE-серверов в реестре Windows . . . 694
OLE-контейнер – компонент TOLEContainer . . . 696
Пример приложения OLE-контейнера . . . 702
Резюме . . . 706

27. Программирование на Win32 API . . . 707

Создание приложения без применения VCL . . . 708
Получение информации о системе . . . 719

Запуск программ . . . 723
Завершение работы . . . 724
Резюме . . . 725

28. Создание апплетов панели управления . . . 726

Стандартные апплеты панели управления Windows . . . 727
Апплет панели управления – класс TAppletModule . . . 728
Приложение панели управления – класс TAppletApplication . . . 729
Пример апплета панели управления . . . 731
Регистрация апплета панели управления . . . 732
Резюме . . . 732

29. Пространство имен оболочки Windows . . . 733

Идентификация объекта оболочки . . . 734
Интерфейс папки – IShellFolder . . . 740
Резюме . . . 747

30. Мультимедиа . . . 748

Проигрыватель мультимедиа – компонент TMediaPlayer . . . 748
Воспроизведение звука средствами Win32 API . . . 757
Резюме . . . 758

Часть II. Разработка баз данных в среде Delphi

31. Реляционная модель данных . . . 759

Ключевые термины реляционной базы данных . . . 763
Этапы проектирования базы данных . . . 764
Нормализация данных . . . 766
Модель данных «сущность–связь» . . . 772
Правила выбора первичного ключа . . . 774
Индексирование таблиц . . . 775
Представление (вид) . . . 776
Хранимая процедура . . . 777
Триггер . . . 777
Транзакции и управление их выполнением . . . 778
Резюме . . . 781

32. Структурированный язык запросов – SQL . . . 782

Назначение и состав языка SQL . . . 783
Основные типы данных SQL-92 . . . 784
Язык определения данных – DDL . . . 788
Язык запросов – DQL . . . 795
Язык манипулирования данными – DML . . . 803
Язык управления доступа к данным – DCL . . . 805
Язык обработки транзакций – TPL . . . 806
Язык управления курсором – CCL . . . 807
Резюме . . . 808

33. Универсальный набор данных – класс TDataSet . . . 809

Открытие и закрытие набора данных . . . 810
Обновление набора данных . . . 811

Перемещение по набору данных . . . 812
Создание закладок и переход к закладке . . . 814
Состояние набора данных . . . 814
Редактирование записей в наборе . . . 816
Организация доступа к отдельному полю . . . 818
Фильтрация набора данных . . . 821
Организация поиска данных . . . 822
Обработка событий . . . 824
Кэширование данных . . . 825
Взаимодействие с элементами управления данными . . . 825
Поддержка таблиц символов OEM и ANSI . . . 826
Резюме . . . 826

34. Работа с полями набора данных . . . 827

Поле таблицы – класс TField . . . 827
Числовые поля – класс TNumericField . . . 850
Текстовые поля – TStringField . . . 853
Логическое поле – TBooleanField . . . 855
Бинарные поля – TBinaryField, TBytesField и TVarBytesField . . . 855
Дата и время – поля TDateTimeField, TDateField и TTimeField . . . 855
Дата и время – поле TSQLTimeStampField . . . 856
Поля больших двоичных объектов – TBlobField, TGraphicField и TMemoField . . . 856
Резюме . . . 861

35. Применение механизма BDE для доступа к данным . . . 862

Введение в Borland Database Engine . . . 862
Компоненты доступа к данным BDE . . . 864
Набор данных BDE – класс TBDEDataSet . . . 865
Соединение с объектом данных – класс TDBDataSet . . . 872
Таблица – TTable . . . 874
Импорт данных – TBatchMove . . . 891
Запрос – TQuery . . . 893
Хранимая процедура – TStoredProc . . . 897
Модифицируемый запрос – компонент TUpdateSQL . . . 899
Резюме . . . 902

36. Элементы управления для работы с данными . . . 903

Источник данных – компонент TDataSource . . . 904
Общие черты компонентов отображения данных . . . 905
Сетка базы данных – компонент TDBGrid . . . 906
Статический текст БД – компонент TDBText . . . 916
Строка ввода БД – компонент TDBEdit . . . 917
Многострочный текстовый редактор БД – TDBMemo . . . 917
Редактор расширенного формата БД – TDBRichEdit . . . 918
Изображение БД – компонент TDBImage . . . 918
Список БД – компонент TDBListBox . . . 919
Комбинированный список БД – TDBComboBox . . . 919
Флажок БД – компонент TDBCheckBox . . . 919
Группа переключателей БД – компонент TDBRadioGroup . . . 920

Компонент TDBCtrlGrid . . . 920
Синхронный просмотр данных . . . 923
Навигатор – компонент TDBNavigator . . . 925
Резюме . . . 926

37. Элементы управления для работы с данными II . . . 927

Компоненты-списки . . . 927
Графический список – компонент TListView . . . 929
Сетка – компонент TStringGrid . . . 931
Иерархические данные . . . 933
Пример проекта иерархической БД . . . 935
Резюме . . . 944

38. Место BDE в клиент-серверных приложениях . . . 945

Сессия – класс TSession . . . 946
Список сессий – TSessionList . . . 954
База данных – класс TDatabase . . . 955
Резюме . . . 961

39. Технология объектов данных ADO . . . 962

Связь между объектной моделью Microsoft ADO и библиотекой VCL . . . 963
Строка соединения ADO . . . 967
Соединение с источником данных ADO – компонент TADOConnection . . . 968
Набор данных ADO – класс TCustomADODataSet, компонент TADODataSet . . . 979
Командный объект ADO – TADOCommand . . . 992
Таблица, запрос и хранимая процедура – компоненты TADOTable, TADOQuery и TADOStoredProc . . . 993
Сервисные методы модуля ADODB . . . 994
Резюме . . . 996

40. Компоненты InterBase . . . 997

Доступ к базе данных InterBase – компонент TIBDatabase . . . 997
Элементарный запрос – компонент TIBSQL . . . 1005
Экспорт и импорт данных . . . 1007
Характеристики наборов данных InterBase – компонент TIBDataSet . . . 1009
Запрос – компонент TIBQuery . . . 1015
Хранимая процедура – компонент TIBStoredProc . . . 1015
Таблица – компонент TIBTable . . . 1015
Транзакция – компонент TIBTransaction . . . 1016
Модифицируемый запрос InterBase – компонент TIBUpdateSQL . . . 1019
Информация об объектах БД – компонент TIBExtract . . . 1020
События InterBase – компонент TIBEvents . . . 1022
Информация о БД – компонент TIBDatabaseInfo . . . 1023
Резюме . . . 1024

Заключение . . . 1025

Литература . . . 1026

Алфавитный указатель . . . 1028

Введение

Не многие области науки могут похвастаться таким бурным развитием, какое претерпели за свою сравнительно недолгую историю существования электронно-вычислительная техника и шагающие с ней рука об руку языки программирования. Не так давно самые первые программы писались на языке машинных команд. Это был поистине каторжный труд. Программист тех старозаветных времен не просто знал язык первых машин, он обладал глубокими инженерными знаниями архитектуры электронно-вычислительной машины (ЭВМ), системы команд процессора, организации памяти и многого другого. Такой высококлассный специалист ценился на вес золота, а производительность его работы была до смешного мала. Процесс создания элементарной программы отдаленно напоминал шаманские обряды (кто видел перфоратор, тот меня поймет), а про программистов слагались легенды.

Такое положение вещей мало кого устраивало, посему учеными предпринимались активные попытки хотя бы в какой-то степени «очеловечить» язык машин. Первым успехом в этом направлении было создание компиляторов с языков ассемблера. Язык низкого уровня ассемблер по-прежнему был очень близок к машинным командам, однако в нем уже отдаленно просматривались и человеческие черты. В ассемблере машинным командам соответствовали англоязычные мнемонические коды. Синтаксис языка не отличался особой изысканностью – каждая команда ассемблера могла включать три элемента: поле метки, код операции и поле операндов. Не так густо, но по сравнению с машинными командами это был настоящий прорыв.

Хотя появление ассемблера и соответствующих компиляторов несколько упростило работу программиста, но, по сути, язык мнемокодов все еще значительно отличался от языка общения людей. Однако, без всякого сомнения, можно утверждать, что ассемблер некоторым образом расширил круг программистов и (к сожалению) снизил требования к их инженерной подготовке, скажем, с уровня шамана до уровня вождя племени. Но взамен было получено ощутимое преимущество: разработка программы на ассемблере ускорилась если не на порядок, то, по крайней мере, в разы. Подчеркну еще один немаловажный факт: умение творить на ассемблере не стало анахронизмом и актуально до сих пор, в особенности в области системного программного обеспечения.

Эра превосходства умеющих общаться с ЭВМ шаманов и вождей над обычным людом длилась совсем недолго. Конец неравенству положили новые языки высокого (третьего) уровня. Хотя новые системы программирования по-прежнему представляли собой компромисс между языком машин и людей, но они уже стали доброжелательными, наполнились существенным словарным запасом, плюс ко всему семантика конструкций существенно приблизилась к обычным человеческим фразам. Благодаря всем этим преимуществам, лишь прочитав введение в язык PL/1, ALGOL, ADA, Fortran или во что-нибудь еще, уверенные в своих силах студенты в два счета переводили в состояние ступора ЭВМ любой степени надежности.

Девяностые годы прошлого века ознаменовались рождением языков 4-го поколения – 4GL (fourth generation languages). В них впервые вместо скучных строк кода программист получил удивительную возможность оперировать графическими, интуитивно понятными образами, а для создания элементарного приложения стало достаточно лишь несколько раз щелкнуть кнопкой мыши. В одно время даже раздавались восторженные возгласы о том, что программирование стало доступным для домохозяек...

Не знаю, хорошо это или плохо, но создание профессионального программного продукта и в наши дни по-прежнему требует от человека глубоких и разносторонних знаний, терпения и внимательности, находчивости и сообразительности и, если не таланта, то, по крайней мере, творческой одаренности, потому что программирование уже давно перестало быть просто наукой – это уже и искусство.

В настоящей книге рассматривается один из безусловных лидеров среди современных систем программирования – среда программирования Delphi. Это глубоко продуманный, высокоэффективный и (что немаловажно) весьма удобный программный продукт, позволяющий создавать приложения практически любой сложности, предназначенные для работы под управлением операционных систем Microsoft® Windows® и Linux.

Изначально Delphi специализировалась только на создании программного обеспечения под Windows. Для этого среда снабжена глубоко проработанной и эффективной библиотекой визуальных компонентов (VCL, Visual Components Library), элементы которой не только инкапсулировали в себе функции прикладного программного интерфейса (API, Application Program Interface) Windows, но и внесли существенные усовершенствования. Благодаря этому библиотека VCL успешно конкурирует с библиотекой MFC (Microsoft Foundation Class), разработанной в корпорации Microsoft, и служит фундаментом альтернативным Microsoft Visual Studio средам программирования Borland Delphi и Borland C++.

В условиях жесткой конкуренции фирма Borland постоянно развивает и улучшает возможности среды разработки. Начиная с шестой версии Delphi в состав среды разработки вошел пакет кроссплатформенной разработки CLX (Borland Component Library for Cross-Platform), основанный на идеях, апробированных в VCL. Delphi 2005 впитала идеи создания распределенных программных продуктов, базирующихся на архитектуре Microsoft® .NET Framework.

Особенность пакета CLX в том, что он позволяет строить приложения не только для Windows, но и для набирающей обороты ОС Linux. Тем самым программисты Delphi получили еще одно существенное преимущество – переносимость приложений между разными операционными системами. Однако за универсальность платформы пришлось заплатить – приложения CLX вынуждены отказаться от вызова функций, специфичных для каждой из операционных систем. В связи с этим опора на CLX не столь рациональна в тех случаях, когда вы нацелены только на работу с Windows. Поскольку настоящая книга посвящена программированию для Windows, то мы больше не будем возвращаться к CLX и системе Kylix (дополнение к Delphi для работы с CLX).

Перечисляя заслуги Delphi, стоит упомянуть доступность и интуитивную понятность интерфейса среды, наглядность кодовых конструкций языка Object Pascal, надежную систему выявления ошибок, высокоэффективный компилятор, умение поддерживать самые распространенные форматы баз данных и многое другое.

Соглашения, принятые в книге

Впервые встречающиеся термины выделены полужирным шрифтом, а элементы интерфейса Delphi – шрифтом OfficinaSans.

Моноширинным шрифтом выделены имена файлов, переменных, констант, массивов, записей, методов, классов, свойств, процедур, функций, модулей и библиотек, а также код примеров и синтаксические конструкции. Зарезервированные слова выделены моноширинным полужирным шрифтом.

Для акцентирования внимания читателя на ключевых частях материала текст выделяется следующим образом:

На заметку.
Данный материал – советы, комментарии или замечания – следует принять к сведению.

Внимание!
Текст, отмеченный восклицательным знаком, однозначно описывает действия программиста в той или иной ситуации.

Стоп!
Однозначный запрет; внимание заостряется на характерных ошибках. Короче говоря, никогда так не делайте.

Такой картинкой отмечено описание методов из состава прикладного интерфейса пользователя Windows 32 API. Кроме того, они «переведены» с языка C на язык Pascal.

Подробный предметный указатель позволяет найти в тексте интересующий вас класс, компонент, свойство и метод по их названию.

1

Язык программирования Pascal

Столь популярный сегодня язык программирования Pascal своим рождением обязан профессору Цюрихской высшей технической школы Николаусу Вирту. Язык, названный в честь французского математика Блеза Паскаля, появился на свет в конце 1960-х годов и предназначался для обучения студентов основам структурного программирования.

Уже с самых первых дней своего существования Pascal был обречен на широчайшую популярность благодаря ряду своих неоспоримых достоинств: гибкости и надежности, простоте и наглядности конструкций, способности контроля правильности исходного кода на этапе компиляции, возможности построения новых типов данных и многим другим качествам, с которыми мы познакомимся в этой книге.

До появления Delphi 7.0 программисты говорили о ***«программировании на языке Object Pascal в среде Delphi»*** *и посмеивались над новичками, употреблявшими словосочетание* ***«язык Delphi»****. Вот и досмеялись… С официальным выходом Delphi 7.0 человек, сказавший, что он создает программные продукты на «языке Delphi», окажется прав. Действительно, за десятилетие язык Object Pascal был настолько усовершенствован, что Borland официально заговорила о языке Delphi. Вместе с этим, по крайней мере на мой взгляд, термин Object Pascal не стал атавизмом и вполне жизнеспособен. Поэтому эта глава и называется «Язык программирования Pascal».*

Достойным учеником профессора Н. Вирта стал основатель (1983 г.) фирмы Borland International, Inc. Филипп Кан. Совместно с А. Хейльсбергом им были созданы высокоскоростные Турбо-компиляторы для языков Pascal, BASIC, Prolog, C и Assembler.

Наилучшим примером упорного труда программистов Borland по совершенствованию языка Pascal наверное может стать приводимый ниже хронологический перечень:

- 1983 г. – год рождения пакета Turbo Pascal 1.0;
- 1985 г. – выход первой интегрированной среды Turbo Pascal 3.0;

- 1987 г. – Turbo Pascal 4.0, в состав пакета вошли графическая библиотека и средства раздельной компиляции;
- 1988 г. – Turbo Pascal 5.0, дальнейшее совершенствование среды;
- 1989 г. – Turbo Pascal 5.5, промежуточная версия, частично включившая в себя поддержку объектно-ориентированного программирования;
- 1990 г. – Turbo Pascal 6.0, многооконная среда программирования, поддержка мыши, объектно-ориентированная библиотека Turbo Vision;
- 1991 г. – Turbo Pascal for Windows, в состав пакета входит ключевая библиотека `ObjectWindows`;
- 1992 г. – Borland Pascal with Objects 7.0;
- 1995 г. – выход Borland Delphi 1.0 для работы под Microsoft Windows 3.x;
- 1996 г. – Borland Delphi 2.0 (первая 32-разрядная версия);
- 1997 г. – Borland Delphi 3.0;
- 1998 г. – Borland Delphi 4.0;
- 1999 г. – Borland Delphi 5.0;
- 2001 г. – Borland Delphi 6.0;
- 2002 г. – Borland Delphi 7.0;
- 2004 г. – Borland Delphi 8.0.

Простейшая программа на Object Pascal

Как-то раз я услышал неформальное, но, на мой взгляд, очень меткое определение программы: «*Программа – это Идея, которую программист изложил на языке программирования*». Такое определение во главу угла ставит не сотни строк безликого кода, а Ее Величество Идею, то, без чего немыслимо существование творческой личности. Это определение – достойный ответ спорщикам на тему: «Что такое программирование – ремесло или Искусство?»

Эта глава посвящена элементарным составным частям программы на языке Object Pascal. В целом разделы этой главы представляют фундамент, без которого изучение Delphi просто невозможно. Подчеркну, что это лишь предельно сжатый рассказ о возможностях языка Pascal.

При изложении материала, посвященного языку Pascal, мы по возможности абстрагируемся от среды Delphi и ее фундамента – библиотеки визуальных компонентов (VCL). На мой взгляд, первые шаги по изучению языка наиболее эффективны в консольных приложениях, где нет отвлекающих новичка элементов управления и код максимально прост и линеен.

При запуске интегрированной среды разработки Delphi автоматически подготавливается к работе новый проект стандартного приложения для Windows. Но для консольного приложения он не подходит, поэтому закройте созданный по умолчанию проект. Для этого в главном окне Delphi выберите пункт меню File → Close All. Затем найдите пункт File → New → Other... и щелкните по нему. Если все сделано правильно, появится окно New Items с открытой

страницей New. Найдите на этой странице пиктограмму Console Application и щелкните по кнопке OK. За этот каторжный труд Delphi отблагодарит нас заготовкой для самого простейшего приложения – консольного окна Windows.

```
program Project1;
{$APPTYPE CONSOLE}

uses SysUtils;

begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

Прежде чем я поясню, что содержится в шаблоне кода консольного приложения, научимся сохранять плоды своей деятельности в файл. Для этого выберем пункт меню File → Save и в открывшемся диалоговом окне присвоим своему первому проекту имя FirstPrj.dpr. В завершение нажмем кнопку OK.

Разрабатываемая программа может включать десятки или даже сотни файлов различного типа и назначения. Но как минимум программа состоит из одного файла – главного файла проекта Delphi. Файл такого типа идентифицируется расширением .dpr *(сокращение от Delphi Project).*

Теперь, когда наш проект сохранен на жестком диске компьютера, вновь обратите внимание на первую строку кода. Вместо имени проекта по умолчанию, Project1, после ключевого слова program появится предложенное нами название FirstPrj. Следующее ключевое слово uses применяется для подключения к проекту внешних модулей, как правило, содержащих библиотеки дополнительных подпрограмм. В частности, наш проект будет эксплуатировать наиболее часто используемую библиотеку системных утилит SysUtils. Шаблон завершается составным оператором begin..end, внутри которого размещается выполняемый код. Пока же здесь только текст комментария.

Комментарии в тексте программы

Во всех без исключения языках программирования предусмотрена возможность комментирования строк исходного кода. В комментарии программист в сжатом виде описывает, что делается в этих строках, для чего введена данная переменная, что произойдет после вызова процедуры. Другими словами, в комментариях разработчик кода кратко поясняет смысл рожденных в его голове команд. В результате листинг программы становится более понятным, более читаемым и доступным для изучения.

Для того чтобы при компиляции программы текст комментариев не воспринимался Delphi как исходный код программы и не служил источником ошибок, приняты следующие соглашения. Комментарием считается:

1. Отдельная строка, начинающаяся с двух наклонных черт //:

```
//Одна строка комментария
```

2. Весь текст, заключенный в фигурные скобки {} или в круглые скобки с символами звездочек (**):

```
{Текст комментария}
(*Это также комментарий*)
```

Текст комментария пропускается компилятором и не оказывает влияния на «жизнедеятельность» модуля.

Если внутри фигурных скобок на первой позиции окажется символ $, то это не что иное, как директива компилятора. В шаблоне только что созданного нами приложения такая директива есть:

```
{$APPTYPE CONSOLE}
```

В данном случае это означает, что наш проект является консольным приложением. Начинающему программисту не следует изменять ни содержимое таких строк, ни место их расположения, в противном случае есть риск привести свой проект в негодность.

Компиляция и запуск программы на выполнение

Теперь научимся компилировать программу. **Компилирование** – это процесс, переводящий программу с языка программирования (в нашем случае с языка Pascal) на язык машинных команд. Как-то неинтересно компилировать пустой проект, поэтому давайте научим его чему-нибудь полезному, например здороваться. В следующем листинге предложен пример такой исключительно воспитанной программы. А для того чтобы исходный код был понятнее, он буквально насквозь пропитан комментариями.

```
program FirstPrj;
{это листинг самой короткой и доброжелательной программы на свете}
{$APPTYPE CONSOLE}        //это директива компилятора, которую мы не трогаем

uses SysUtils;  (*строка подключения внешних библиотек подпрограмм, хотя, между нами
говоря, в этой программе внешние модули нам не нужны*)

begin
WriteLn('Hello, World!'); //выводим текст «Привет, Мир!»
ReadLn;                   //ожидаем ввод – нажатие любой клавиши завершит работу
end.
```

Повторив код программы, выберите пункт главного меню Delphi Run → Run. Если все повторено безошибочно, то за считанные доли секунды на экране появятся плоды нашего коллективного творчества – консольное окно со строкой «Hello, World!» Если же вдруг была допущена ошибка, то компилятор просигнализирует о ней, выделив в листинге строку, содержащую предполагаемую ошибку или следующую за ней строку.

Вместо утомительных поисков необходимого элемента в меню у программистов Delphi наибольшей популярностью пользуется быстрая клавиша запуска программы – функциональная клавиша F9. При нажатии этой клавиши осуществляется проверка синтаксиса проекта, его компиляция и запуск исполняемого exe-файла.

Как видите, вся программная логика сосредоточена внутри составного оператора `begin..end` и выполняется линейно в соответствии с очередностью следования строк.

Переменные и константы

Переменная – это хранилище для данных. В самом названии заключен смысл ее применения – переменная предназначена для работы с изменяющимися значениями. В языке Pascal могут быть объявлены переменные различных типов (об основных типах данных мы поговорим чуть позже). Для объявления переменной используется зарезервированное слово var (*сокр.* от variable). Синтаксис выглядит следующим образом:

```
var имя_переменной : тип_данных;
```

Например:

```
var  x : Integer;       //переменная x специализируется на работе с целыми числами
     y, z : real;       //переменные y и z могут хранить действительные числа
     s : char;          //s – символ
     MyVar : Boolean; //MyVar – логическая переменная
```

Как и переменная, константа также является хранилищем для данных, но, в отличие от переменной, константа задается раз и навсегда и не допускает редактирования своего содержимого. Для задания константы применяется зарезервированное слово const (*сокр.* от constant). Синтаксис определения константы следующий:

```
const <имя_константы> [: тип данных] = <значение>;
```

В квадратных скобках может быть отмечен необязательный указатель на тип константы. Приведем пример определения обычных констант:

```
const  A = 100;
       B = -3.1426;
       C = 'Текст';
```

Для задания значения, которое будет содержаться в обычной константе, допускается применение математических выражений и результатов, возвращаемых функциями.

```
const  D = 500;
       E = D+6;
       F = 3/Pi;
```

При определении типизированной константы явным образом указывается тип хранящихся в ней данных:

```
const  H : byte = 255;
       I : Boolean = true;
```

В отличие от обычных, или как их еще иногда называют «истинных», констант, типизированные константы не рекомендуется инициализировать выражениями. В Delphi в качестве констант могут определяться массивы, записи, указатели; кроме того, существуют и экзотические константы, например процедурные.

Важно знать, в каком именно месте листинга допускается объявление переменных и констант. В консольных проектах объявление осуществляется перед составным оператором begin..end.

```
program Project1;

{$APPTYPE CONSOLE}

uses SysUtils;
const X = 10;              //объявление константы
var   Y, Z : integer;      //объявление двух переменных
begin
Z:=X+Y;
//остальной код программы
end.
```

Идентификаторы

Идентификатор – это имя переменной, константы, массива, метода, модуля и всего остального, что должно иметь имя. В Delphi длина идентификатора не ограничена, но значащими являются только первые 255 символов. Идентификатор может содержать любые символы латинского алфавита, цифры и символ нижнего подчеркивания. Первый символ идентификатора обязательно должен быть буквенный. Само собой в роли идентификаторов не допускается применять зарезервированные слова.

В отличие от C, язык программирования Pascal не критичен к регистру символов, поэтому в Delphi следующие названия будут восприниматься как идентичные: `MyValue`, `myvalue`, `MYVALUE`, `mYvALUE`.

Если в рамках одного проекта существует несколько модулей с одинаковыми именами идентификатора, то для обращения к идентификатору требуется уточнить, кому он принадлежит.

```
Form1.Button1.Caption;
Form2.Button1.Caption;
Unit1.MyProcedure;
```

Основные типы данных

Одной из ключевых особенностей языка Object Pascal является жесткая типизация данных. Именно благодаря строгости в подходе к объявлению переменных, процедур и функций, Delphi может похвастаться одним из самых совершенных компиляторов. Что такое типизация? Все достаточно просто. Представьте себе педантичного джентльмена, любящего находить все свои вещи в отведенных им местах: зубную щетку – в шкафчике над умывальником, а смокинг – в платяном шкафу, и никак не наоборот. Если вдруг произойдет обратное, то Delphi потеряет к нам всякий интерес, отправив сообщение об ошибке.

Более того, размеры каждой вещи нашего педанта (другими словами, объем памяти, занимаемый переменной или объектом) соответствуют четко установленным правилам – не больше и не меньше. Однако тип данных не только накладывает ограничения на размер объекта, скажем переменной, но и строго определяет перечень операций, которые можно производить с этим объектом. И это правило весьма логично и последовательно – ведь совсем не

стоит, например, в переменную, предназначенную для хранения целого числа, помещать пусть даже очень хорошую строку «Hello, Word!».

На рис. 1.1. предложен вариант классификации типов данных, применяемых в Delphi.

Рис. 1.1. *Классификация основных типов данных языка Delphi*

Не надо отчаиваться при виде такой паутины (это, кстати, только верхушка айсберга). Нашими общими усилиями узелок за узелком она будет распутана. Итак, каждый тип данных предназначен для хранения информации определенного вида, и в самом общем случае можно говорить о существовании шести основных типов данных языка Delphi:

- простой
- строковый
- структурный
- указательный
- процедурный
- вариантный

Простые типы данных

Самым большим из представленных типов по праву считается простой. Он предназначен для хранения данных в форме чисел или некоторых упорядоченных последовательностей. Этот тип логически разделяется на две ветви: порядковые и действительные типы. К порядковым типам относятся:

- целые числа
- символьные типы
- логические типы
- перечислимые типы
- поддиапазоны

Если первые три группы в языке Delphi описаны самым жестким образом и не допускают каких-либо изменений, то два последних типа (перечисли-

мый и поддиапазон) могут определяться пользователем непосредственно во время процесса разработки программы.

Отношения между элементами любого из порядковых типов складываются вполне ординарно. Все элементы внутри одного типа могут располагаться в виде упорядоченной последовательности (следующий больше предыдущего). Для всех простых типов (за исключением целых чисел) справедливо утверждение, что первый (самый младший) элемент последовательности имеет индекс 0, второй – 1 и т. д. Целые же числа допускают хранение и отрицательных значений, поэтому здесь самый младший элемент последовательности может начинаться не с нуля. К еще одной особенности порядковых типов данных стоит отнести отсутствие знаков после запятой.

Целые числа

В табл. 1.1 представлены порядковые целочисленные значения. В первую очередь типы данных, описывающие целые числа, характеризуются пределом границ диапазона хранимых значений и возможностью описывать отрицательные величины. Чем больше предел допустимых значений, тем больший объем памяти будет занимать переменная этого типа. Как видно из таблицы, самым серьезным из предлагаемых типов является `Int64`, «пожирающий» целых 8 байт (64 бит) ОЗУ. Такой тип способен хранить величины, сопоставимые с количеством звезд во Вселенной. Как правило, для решения «земных» задач программисту достаточно диапазона значений типа `Integer`.

Таблица 1.1. Целые числа

Тип	Диапазон значений	Размер в байтах
`Int64`	$-2^{63} .. 2^{63}-1$	8
`Integer (Longint)`	−2147483648 .. 2147483647	4
`Smallint`	−32768 .. 32767	2
`Shortint`	−128 .. 127	1
`Byte`	0 .. 255	1
`Word`	0 .. 65535	2
`Cardinal (LongWord)`	0 .. 4294967295	4

Символьные типы

Основное назначение символьного типа данных – организация вывода информации на экран компьютера и принтер. В Windows обеспечена поддержка трех наиболее важных наборов символов:

1. OEM – набор символов по умолчанию для MS-DOS.
2. ANSI – набор символов по умолчанию для Windows 9.x.
3. Unicode – набор символов по умолчанию для Windows NT/2000.

Фундаментом наборов символов OEM и ANSI служит код ASCII, в котором каждый символ представлен значением от 0 до 127 (соответственно символ занимает 7 бит памяти). Кодам от 0 до 31 и 127 стандартный 8-битный набор ставит в соответствие управляющие символы (например, символы забоя, та-

буляции, конца строки и возврата каретки); остальные символы могут быть выведены на экран. Исторически сложилось, что оставшиеся символы были закреплены за латинскими буквами.

Вскоре был задействован и восьмой бит кода, что позволило расширить код ASCII до 256 символов («расширенный набор символов»). Этот набор символов был разработан производителями IBM PC и получил название OEM. Здесь коды от 32 до 126 унаследованы от ASCII, а оставшиеся коды включают дополнительные символы, в частности символы псевдографики для программ DOS.

В большинстве случаев Windows и приложения под Win32 используют «набор символов ANSI». Коды данного набора от 32 (0x20) до 127 (0x7F) соответствуют коду ASCII. Сравнительно недавно появилась еще одна кодировка, получившая название UNICODE. Один символ в такой кодировке занимает целых два байта, и благодаря этому он может принимать одно из 65535 значений.

Итак, для работы с отдельными символами Delphi предоставляет следующие типы данных:

Таблица 1.2. Символьные типы

Тип	Кодировка	Размер в байтах
`Char (AnsiChar)`	ANSI	1
`WideChar`	UNICODE	2

Логические (булевы) типы

Логический тип применяется для хранения логических данных, способных принимать только два значения: 1 (`true`/истина) и 0 (`false`/ложь).

Таблица 1.3. Логические типы

Тип	Диапазон значений	Размер в байтах
`Boolean`	0 – `false`; 1 – `true`;	1
`ByteBool`	от 0 до 255, где 0 – `false`, 1..255 – `true`	1
`WordBool`	от 0 до 65535, где 0 – `false`, 1..65535 – `true`	2
`LongBool`	от 0 до 4294967295, где 0 – `false`, 1..4294967295 – `true`	4

Перечислимые типы

Перечислимые типы относятся к типу данных, определяемых программистом. Перечислимый тип данных задается списком имен.

```
type TypeName = (Value1, Value2,..., Value19);
```

Числа, а также логические и символьные константы не могут быть элементами перечислимого типа. В качестве примера представим перечислимый тип, соответствующий дням недели:

```
type TypeWeekDay =(Mon, Tu, We, Th, Fr, Sa, Su);
. . .
var WDay1, WDay2 : TypeWeekDay;
begin
```

```
WDay1 : = Mon;
WDay2 : = Tu;
end;
```

Особенность перечислимого типа в том, что каждому его элементу соответствует порядковый номер, начиная с 0. Наличие порядкового номера позволяет проводить операции сравнения:

```
if WDay1<WDay2 then …
```

Совместно с данными перечислимого типа зачастую используют следующие функции:

```
function Pred(X);   // возвращает предшествующее значение аргумента
function Succ(X);   // возвращает следующее значение аргумента
```

Поддиапазоны

Переменная, входящая в поддиапазон, может принимать значения только в пределах границ диапазона.

```
type SubIntegerRange = 10 .. 100;
type SubCharRange =  'A' .. 'Z';
. . .
var IntValue  : SubIntegerRange;
    CharValue : SubCharRange;
. . .
MyValue : = 50;
CharValue : = 'X';
```

При попытке присвоить переменной `IntValue` значение вне диапазона `SubIntegerRange` компилятор Delphi откажется иметь с нами дело.

- Упрощает локализацию ошибок в проекте.
- Выпуск новой версии программного продукта может свестись к обновлению одной-двух библиотек, а не всего проекта в целом.
- Предоставляет группе программистов возможность одновременной работы над одним проектом. Разделение труда, когда один работает над исполняемым модулем, второй – с первой библиотекой и т. д., значительно упрощает и, что очень важно, ускоряет рождение программного продукта.
- Допускает использование DLL сторонними разработчиками ПО (и наоборот).
- Обеспечивает совместное использование библиотечного модуля несколькими процессами.

Именно благодаря приведенным выше преимуществам динамические библиотеки составляют основу архитектуры построения современных версий Windows. Операционная система превращается не в закостенелый ком, а в группу разумно взаимодействующих модулей, становясь соответственно более открытой и доступной для программиста.

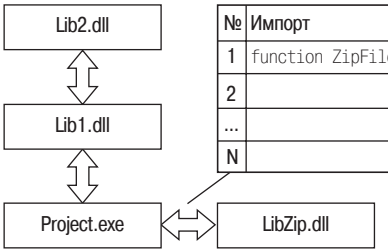
Использование динамической библиотекой стандартного расширения `*.DLL` *далеко не догма. В Windows широко применяются библиотеки с расширениями:* `*.EXE` *– специальные файлы Windows;* `*.DRV` *– драйверы устройств;* `*.SYS` *– системные файлы.*

При изучении Windows вы, наверное, уже сталкивались с такими файлами, как `KERNEL32.DLL`, `COMMDLG32.DLL` и `GDI32.DLL`. Эти и многие другие динамические библиотеки содержат функции интерфейса программирования (API) и ресурсы Windows, широко используемые программистами во время создания приложений. Например, модуль `KERNEL32` отвечает за управление памятью и инициализацию процессов, `COMMDLF32` содержит все стандартные диалоговые окна, а `GDI32` обслуживает графический интерфейс.

Проектирование приложения, использующего DLL

Разработка приложения, основанного на динамически подключаемых библиотеках, требует серьезного подхода уже на первоначальном этапе – этапе построения структуры программного продукта. Однозначно уяснив задачу или группу задач, решение которых должен обеспечить будущий шедевр программного искусства, мы разбиваем их на более мелкие составляющие – строим иерархию частных задач, без решения которых невозможно справиться с основной задачей. Детализация продолжается до тех пор, пока не появится возможность представить решение частной задачи на алгоритмическом языке в виде элементарного программного модуля.

После расчленения будущего приложения на модули настает очередь уточнения связей между ними. На этом этапе наиболее подготовленные разработчики ПО уже готовы говорить о потоке данных между основным модулем и взаимодействующими с ним элементами. Поток данных – это параметры, передаваемые импортируемым функциям, и значения, возвращаемые этими функциями.



№	Импорт	Результат
1	`function ZipFile(fName,fPath:string):boolean;`	в случае успеха – `true`
2	...	
...	...	
N	...	

***Рис. 19.1.** Структура проекта с таблицей импортируемых данных из библиотеки LibZip.dll*

Рассмотрим пример проекта, одна из частных задач которого состоит в архивировании каких-либо данных. Ядром программного продукта служит исполняемый модуль `Project.exe`. Непосредственно с ним взаимодействуют динамически подключаемая библиотека `Lib1.dll` (в свою очередь способная вызывать библиотеку `Lib2.dll`) и библиотека архивации `LibZip.dll`. Все связи между модулями целесообразно описать в виде таблиц импортируемых (экспортируемых) функций (рис. 19.1). В идеале в таблице импорта должны быть отмечены имена методов, список передаваемых параметров, тип возвращаемых значений и соглашения о вызовах.

Что может стать поводом для выноса части процедур и функций в отдельную библиотеку? Основной причиной является интуитивное понимание программистом, что задачи, решаемые модулем, могут повториться в перспективных проектах.

Второй причиной может стать предположение, что процедуры и функции, необходимые для построения полнофункционального проекта, будут вызываться нерегулярно (как операция архивирования в примере). В таком случае нет необходимости перегружать основной модуль излишним кодом и нерационально занимать дополнительные ресурсы компьютера. Достаточно загрузить динамическую библиотеку на период выполнения необходимых операций, а затем освободить захваченные ресурсы.

В-третьих, идея построения модульного проекта будет весьма кстати в случае, если вы предполагаете усовершенствование программного продукта в дальнейшем. При такой перспективе гораздо удобнее работать с отдельными модулями, чем с единым громоздким продуктом. Это далеко не полный перечень условий, при которых модульное построение проекта выглядит в более выигрышном свете, но на первых порах достаточно руководствоваться этими тремя причинами.

Операционная система и 32-разрядные DLL

Для того чтобы понять принцип работы динамических библиотек, надо знать ответ на вопрос: «Что происходит внутри современных версий Windows при запуске приложения?». В первую очередь операционная система создает *объект главного процесса* (*Process kernel object*). Процесс сразу запускает *пер-*

вичный поток (*Primary thread*). В 32-разрядных версиях Windows стартующий процесс получит в свое распоряжение отдельное адресное пространство размером 2^{32} (4 294 967 296) байт (4 Гбайт) виртуальной памяти.

Современные процессоры способны обращаться к 2^{36} (68 719 476 736) байт оперативной памяти (64 ГБайт). К тому же недалек тот день, когда рынок операционных систем будет захвачен 64-разрядными ОС.

Для поддержания адресации участок памяти распределяется не только из оперативной памяти, но и из системного страничного файла (файла подкачки). Для того чтобы избежать загрузки всего исполняемого файла в оперативную память или в страничный файл, после создания объекта главного процесса создается новый объект, именуемый как *объект отображения файла*. Его задача – отобразить выполняемый файл в адресное пространство процесса.

После успешного отображения исполняемого файла в виртуальную память операционная система разыскивает внутри этого файла все ссылки на динамически подключаемые библиотеки, и эти библиотеки отображаются в тот же процесс (где-то внутри 4 Гбайт участка адресуемой памяти главного процесса). Владельцем всех данных, объектов и переменных динамической библиотеки становится поток, вызвавший функцию DLL. Для каждого файла библиотеки создается свой собственный объект отображения файла. Если в свою очередь внутри динамической библиотеки существуют ссылки на другие файлы библиотек, то последние также отображаются в адресное пространство.

Главный процесс способен многократно загружать одну и ту же динамическую библиотеку путем запуска дополнительных потоков, но при этом отображение DLL в адресное пространство происходит только при первой загрузке. Последующие загрузки осуществляют приращение счетчика обращений к библиотеке. Если DLL загружалась *N* раз, то образ библиотеки в адресном пространстве процесса будет храниться до тех пор, пока не произойдет *N* выгрузок. С каждой выгрузкой динамической библиотеки счетчик обращений уменьшает свое значение на 1. Нулевое значение счетчика является признаком того, что библиотека более не нужна.

При завершении работы образ каждого dll-файла удаляется из памяти вместе с соответствующим объектом отображения. Процесс завершится до удаления образа exe-файла и объекта главного процесса. С завершением главного процесса автоматически освобождается используемая память.

Создание шаблона динамической библиотеки в Delphi

Для создания динамической библиотеки в среде Delphi предусмотрен отдельный мастер. Для его вызова в меню File выберите команду New и в диалоговом окне New Items на вкладке New найдите пиктограмму DLL Wizard (см. рис. 19.2). Нажав кнопку OK, вы получите заготовку проекта новой динамической библиотеки. Создайте проект DLL и сохраните его в отдельном каталоге под именем `mydll.dpr`. А теперь посмотрим листинг библиотеки `mydll`:

```
library mydll;
{Important note about DLL memory management: ShareMem must be the first unit in your
library's USES clause AND your project's (select Project-View Source) USES clause if
your DLL exports any procedures or functions that pass strings as parameters or func-
tion results. This applies to all strings passed to and from your DLL--even those that
are nested in records and classes. ShareMem is the interface unit to the BORLNDMM.DLL
shared memory manager, which must be deployed along with your DLL. To avoid using
BORLNDMM.DLL, pass string information using PChar or ShortString parameters.}

uses
 SysUtils, Classes;

{$R *.RES}

begin

end.
```

В отличие от уже знакомых нам листингов, начинающихся со слова `program` или `unit`, в заглавной строке вы встретите ключевое слово `library`. Это и есть визитная карточка исходного кода динамически подключаемой библиотеки. Стоит отметить, что шаблон DLL не имеет отдельных секций `interface` и `implementation`.

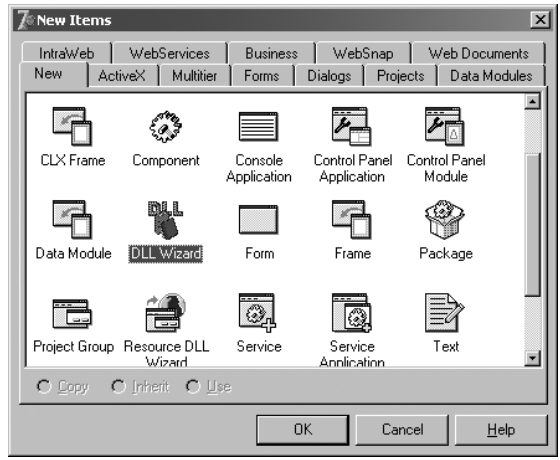


Рис. 19.2. *Диалоговое окно New Items*

Первое, что бросается в глаза, – это устрашающий по объему текст комментария. В его строках разработчики Delphi предупреждают программиста о том, что если хотя бы одна из экспортируемых функций DLL требует данные строкового типа (в качестве параметра или в виде возвращаемого значения), то **самым первым** в операторе uses должен стоять модуль `ShareMem`. Такая необходимость обусловлена особенностью строкового типа данных `AnsiString`.

Дело в том, что, в отличие от стандартных строк языка Pascal, `AnsiString` по существу представляет собой указатель на строку, завершающуюся нулевым символом. Но указатель адресуется к 13-му байту, так как непосредственно строковым данным предшествует 12-байтная структура, содержащая техническую информацию о размерности предназначенной для строки памяти, счетчик обращения к строке и ее физическую длину. Модуль `ShareMem` устраняет подобную несогласованность и делает так, что исполняемый и библиотечный модули начинают говорить на одном языке.

Единственный недостаток применения дополнительного модуля `SHAREMEM` *заключается в том, что при установке своего программного продукта на машину заказчика в состав проекта необходимо включить еще одну библиотеку –* `BORLNDMM.DLL` *(ее можно найти в каталоге \Bin). Успокаивает то, что ее размер не превышает 26 Кбайт.*

При желании можно отказаться от использования библиотеки BORLNDMM.DLL*, но в этом случае строковую информацию следует передавать при помощи* PChar *или* ShortString.

Инициализация динамической библиотеки

Код инициализации динамически подключаемой библиотеки описывается внутри конструкции begin..end модуля .dpr. Операторы, размещенные в этой части модуля, выполняются один-единственный раз сразу после загрузки DLL. Как правило, здесь устанавливаются начальные значения глобальных переменных, распределяется память и инициализируется обработчик исключительных ситуаций данного модуля.

Глобальный обработчик ошибок в DLL

Как уже говорилось, лучше ошибок не допускать. К сожалению, это утопичная идея. Самым обобщенным методом обработки исключительных ситуаций в динамически подключаемых библиотеках можно считать способ, по крайней мере извещающий пользователя о том, что «не все спокойно в Датском королевстве».

```
library MyDll;
uses  ShareMem, Forms, SysUtils, Classes;

{$R *.RES}

type TDllExcept= class(TObject)
    private
      procedure ExceptControl(Sender:TObject; E:Exception);
    end;
var
DllExcept : TDllExcept;

procedure TDllExcept.ExceptControl(Sender:TObject; E:Exception);
begin
Application.MessageBox(PChar(E.Message), PChar('Ошибка'), 1);
end;

{Код инициализации DLL}
begin
DllExcept:=TDllExcept.Create;
Application.OnException:= DllExcept.ExceptControl;
end.
```

Для оповещения пользователя о возникновении исключительной ситуации код инициализации DLL на базе объявленного класса TDllExcept создает объект DllExcept. В следующей строке листинга указывается, что в случае возникновения события OnException() приложения должен быть вызван метод ExceptControl() нового объекта. Обратите внимание, что строка uses динамической библиотеки пополнилась новым модулем forms.

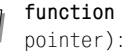
Не забудьте уничтожить созданный вами объект DllExcept *в момент выгрузки DLL. Момент выгрузки DLL нам подскажет метод* DllEntryPoint().

Точка входа/выхода для библиотеки DLL

Большинство библиотек динамической компоновки представляют собой обычные коллекции практически независимых друг от друга функций, экспортируемых в приложения и используемых в них. Но помимо функций, предназначенных для экспортирования, в каждой создаваемой в Delphi динамической библиотеке есть функция `DllMain()`, предназначенная для инициализации и очистки динамической библиотеки.

Для программиста, привыкшего работать с методами VCL, эта функция несколько необычна. Ее особенность в том, что она скрыта от наших глаз; мы не найдем ее в создаваемом по умолчанию шаблоне библиотеки `library`. Такое решение Borland обусловлено тем, что компилятор умеет самостоятельно подключать к проекту стандартную версию `DllMain()`. Однако это не означает, что мы не имеем право написать собственную усовершенствованную версию функции `DllMain()` и применять ее вместо классической.

Но прежде чем броситься в бой, стоит познакомиться с прототипом `DllMain` – функцией Windows API `DllEntryPoint()`. Это специальная пользовательская функция обратного вызова, которая вызывается при каждом подключении (отключении) нового процесса или потока (подзадачи) внутри процесса. Ее назначение – обеспечение единой точки входа/выхода в процессе инициализации или выгрузки DLL.




```
function DllEntryPoint(hinstDLL : THandle; dwReason : DWORD; Reserved :
pointer): Boolean;
```

Единственный параметр, на который стоит обратить внимание, именуется `dwReason`. Операционная система помещает в него значение (табл. 19.1), указывающее причину вызова функции `DllEntryPoint()`.

Таблица 19.1. Возможные значения параметра dwReason

Значение	Описание
`DLL_PROCESS_ATTACH`	DLL отображена в адресное пространство процесса. Происходит в результате запуска процесса или вызова функции `LoadLibrary()`.
`DLL_THREAD_ATTACH`	Значение посылается всем загруженным библиотекам при создании главным процессом нового (дочернего) потока. Имейте в виду,что если DLL используется и другими процессами, то значение `DLL_THREAD_ATTACH` ею получено не будет.
`DLL_THREAD_DETACH`	Главный процесс завершает выполнение дочернего потока.
`DLL_PROCESS_DETACH`	Библиотека DLL выгружается из процесса. Происходит по причине завершения процесса или вызова функции `FreeLibrary()`.

Связь между `DllEntryPoint()` и `DllMain()` отражена в параметре `dwReason`, описывающем причину вызова метода. Взгляните на немного доработанный листинг DLL `MyDll`. Обратите внимание, что в строке `uses` появился еще один модуль – `Windows`.

```
library MyDll;
uses ShareMem, Forms, SysUtils, Classes, Windows;
```

```
{$R *.RES}

type TDllExcept = class(TObject)
    private
      procedure ExceptControl(Sender:TObject; E:Exception);
    end;
var
DllExcept : TDllExcept;

procedure TDllExcept.ExceptControl(Sender:TObject; E:Exception);
begin
Application.MessageBox(PChar(E.Message), PChar('Ошибка'), 1);
end;

procedure DllMain(fdwReason : DWORD);
begin
  case fdwReason of
  DLL_PROCESS_ATTACH :
      begin
      DllExcept:=TDllExcept.Create;
      Application.OnException:=DllExcept.ExceptControl;
      end;
  DLL_THREAD_ATTACH  : {};
  DLL_THREAD_DETACH  : {};
  DLL_PROCESS_DETACH : DllExcept.Free;
  end;
end;

{Код инициализации DLL}
begin
DllProc:=@DllMain;
DllMain(DLL_PROCESS_ATTACH);
end.
```

Код процедуры `DllMain()` прозрачен. Параметр `fdwReason` отслеживает причину вызова динамической библиотеки. В зависимости от его значения в конструкции `case..end` выполняется соответствующий блок операторов. При отображении DLL в адресное пространство процесса (`DLL_PROCESS_ATTACH`) создается объект `DllExcept` (глобальный обработчик исключительных ситуаций). При выгрузке DLL из процесса (`DLL_PROCESS_DETACH`) освобождаются ресурсы, занимаемые объектом `DllExcept`.

Более интересен сам код инициализации, в котором описан процесс подключения процедуры `DllMain()`. И первое, что вызывает недоумение, – это происхождение «объекта» `DLLProc`, поскольку он не объявлен в секции `var` нашего модуля. Для того чтобы ответить на этот вопрос, удерживая клавишу Ctrl, щелкните левой кнопкой мыши по слову `DLLProc` прямо в редакторе кода Delphi (второй вариант: выделив слово `DllProc`, вызовите контекстное меню и в нем выберите пункт Find Declaration). Редактор кода откроет листинг модуля `SysInit`, подключенного к проекту в глубокой тайне от нас. Здесь и объявлена ссылка на `DLLProc`.

```
DllProc: TDLLProc; {Called whenever DLL entry point is called}
DllProcEx: TDLLProcEx absolute DllProc;
```

Не вдаваясь в тонкости кухни Delphi, отметим лишь одно: в момент, предшествующий инициализации файла .dpr, в метод DllMain() передается значение fdwReason. Это происходит благодаря процедуре _InitLib, обитающей в том же модуле SysInit. Остальное дело техники. В разделе инициализации DLLProc() присваивается значение адреса нашей процедуры DLLMain. Далее следует явный вызов DLLMain() с параметром DLL_PROCESS_ATTACH, после чего процедура DLLMain вызывается автоматически.

Экспортирование функций DLL

Для того чтобы приложение или другая библиотека могли обращаться к функциям внешней динамической библиотеки, эти функции должны быть помечены как экспортируемые. В связи с этим синтаксис объявления экспортируемых библиотекой методов имеет ряд небольших отличий от уже привычного описания. Четко выделяются две особенности:

1. Выбор **соглашения о вызовах** для передачи параметров от исполняемого модуля к динамической библиотеке.
2. Объявление экспортируемой функции с ключевым словом export и использование секции exports для указания имен экспортируемых процедур и функций.

Соглашение о вызовах

Прежде чем написать первую экспортируемую функцию, программист принимает решение о способе передачи данных между модулями проекта. В Delphi различают пять соглашений о вызовах (протоколов), отличающихся друг от друга:

- использованием стека (или регистров) для передачи параметров;
- направлением заполнения стека (слева направо или справа налево);
- очисткой стека.

Такое разнообразие протоколов объясняется стремлением разработчиков Delphi к построению максимально гибкого программного продукта, совместимого с другими языками программирования.

Таблица 19.2. Соглашения о вызовах

Директива	Используется	Порядок	Очистка стека
register	регистры, стек	слева направо	вызываемая процедура
pascal	стек	слева направо	вызываемая процедура
cdecl	стек	справа налево	вызывающая процедура
stdcall	стек	справа налево	вызываемая процедура
safecall	стек	справа налево	вызываемая процедура

Директива register (соглашение *Fast-call*) считается наиболее эффективным протоколом передачи данных и устанавливается по умолчанию. Для первых трех передаваемых параметров по возможности используются расширенные

регистры процессора (EAX, EDX и ECX). Остальные параметры (если они есть) передаются, как в директиве pascal. К сожалению, из-за уникальности протокола он может использоваться только с компиляторами Delphi или C Builder.

Соглашение о вызовах pascal по скорости слегка уступает register. К недостаткам протокола можно отнести отсутствие поддержки переменного числа параметров.

Директива cdecl предназначена для организации вызовов в духе языка C. Протокол допускает передачу переменного числа параметров, но, к сожалению, в среде Delphi это не всегда имеет смысл.

Директивы stdcall и safecall используются для обращения к API Windows. Протокол safecall предназначен для объявления методов с двойным интерфейсом. Например, это соглашение о вызовах используется для обращения к объектам автоматизации (Automation object).

Для того чтобы модуль был совместим с применяемыми по умолчанию методами Win32 API, используйте соглашение stdcall.

Применим наши знания на практике. Для этого откройте ранее начатый проект MyDLL. Опишем экспортируемую функцию SumFunc(), осуществляющую сложение двух чисел и возвращающую результат операции. Чтобы не перегружать заголовочный модуль library MyDll избыточным кодом, добавьте к проекту DLL новый unit. Для этого воспользуйтесь меню File → New и на вкладке New диалогового окна New Items выберите пиктограмму Unit.

Сохраните модуль Unit под именем MathFunc.pas. Обратите внимание, что в строке uses библиотеки library MyDll появилась ссылка на MathFunc.pas.

```
library MyDll;
uses ShareMem, Forms, SysUtils, Classes, Windows, MathFunc in 'MathFunc.pas';
```

Объявите экспортируемую функцию в разделе interface модуля MathFunc, не забыв при этом указать стандартное соглашение о вызове stdcall и ключевое слово export. В разделе implementation опишите саму функцию.

```
unit MathFunc;

interface

function SumFunc(iValue1, iValue2:integer):integer; stdcall; export;
implementation

function SumFunc(iValue1, iValue2:integer):integer;
begin
result:=iValue1+iValue2;
end;
end.
```

В заключение осталось опубликовать SumFunc() в разделе exports библиотеки MyDLL. Для этого сразу после списка используемых модулей uses добавьте новую строку:

```
exports SumFunc;
```

Хранение форм в DLL

При проектировании объемного программного пакета, содержащего несколько приложений, программисты стараются вынести совместно используемые приложениями формы в динамические библиотеки. Такое решение позволит отказаться от бессмысленного размножения одинаковых форм в каждом проекте, снизит размер программного продукта на жестком диске и, самое главное, существенно сократит временные ресурсы, затрачиваемые на создание приложения.

Доступ приложения к хранящимся во внешнем модуле формам существенно отличается от обращения к форме из проекта. Здесь команда `Form1.Show` или что-то в таком духе, к сожалению, не сработает. Для выполнения казалось бы элементарных действий в DLL придется описывать функцию-оболочку, и не одну. Как правило, в динамической библиотеке для каждого этапа работы с формой (создание, показ, уничтожение) описывается отдельная функция. А при необходимости обработки каких-либо данных в форме – и дополнительные функции.

Дополним наш проект `MyDLL` формой-календарем. Задача формы – возвратить в основную программу значение выбранной пользователем даты. Для внесения в динамическую библиотеку новой формы воспользуйтесь командой меню File → New Form. Переименуйте форму в `frmCalendar` и сохраните файл под именем `calendar.pas`. В соответствии с рис. 19.3 поместите на форме компонент `MonthCalendar1` и две кнопки `btnOk` и `btnCncl`.

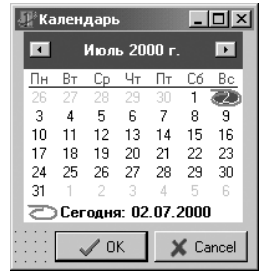


***Рис. 19.3.** Форма-календарь*

Весь код, необходимый для организации работы формы-календаря, должен быть спрятан внутри тела динамической библиотеки. Задача DLL – предоставить внешним модулям простые и понятные функции доступа к библиотеке.

```
unit calendar;

interface

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

type
  TfrmCalendar = class(TForm)
    MonthCalendar1 : TMonthCalendar;
    btnOk : TBitBtn;
    btnCncl : TBitBtn;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

function FrmShowAndFree(hWnd:THandle):Real; stdcall; export;

var frmCalendar: TfrmCalendar;
```

```
implementation

{$R *.DFM}

function FrmShowAndFree(hWnd:THandle):Real;
begin
 Application.Handle:=hWnd;
 frmCalendar:=TFrmCalendar.Create(Application);
 Result:=0;
 if frmCalendar.ShowModal=mrOK then Result:=frmCalendar.MonthCalendar1.Date;
 frmCalendar.Release;
end;
end.
```

Взаимодействие динамической библиотеки с проектом

Различают два варианта загрузки динамически подключаемых библиотек: *явная* (динамическая) и *неявная* (статическая) загрузка. Неявная загрузка предполагает жесткое подключение библиотеки к исполняемому модулю. В такой ситуации сразу после старта приложение приступает к загрузке всех необходимых ему интерфейсных модулей. В среде современной ОС статическая компоновка абсолютно неэффективна, так как приводит к неэкономному использованию одного из основных ресурсов – памяти. Каждое вновь стартующее приложение загружает в память компьютера один и тот же программный модуль; это далеко не самый рациональный подход.

Реализация явной загрузки несколько сложнее, но вместе с тем обеспечивает большую гибкость и надежность. Вам не придется подключать к программе никакие интерфейсные модули. Вместо этого приложение станет вызывать импортируемые функции при помощи методов Windows API.

Где следует размещать DLL

Операционная система Windows самостоятельно обнаружит необходимую библиотеку, но при условии, что файл DLL размещен в одном из каталогов:

- в каталоге, из которого запускается основной исполняемый файл `.EXE`;
- в текущем каталоге;
- в системном каталоге Windows;
- в каталоге Windows;
- в каталогах, включенных в переменную окружения `PATH`.

Явная загрузка DLL

При организации явной загрузки динамически подключаемой библиотеки программист должен пройти через четыре основных этапа:

1. Загрузка DLL.
2. Получение адреса нужной функции.

3. Вызов функции.
4. Выгрузка библиотеки из памяти.

Три из четырех действий реализуются при помощи функций Windows API. Отображение динамической библиотеки в адресное пространство процесса осуществляет функция LoadLibrary() или LoadLibraryEx():

Win32 API
```
function LoadLibrary(LibFileName: string):THandle;
function LoadLibraryEx(LibFileName: string; hFile :THandle;
Flags : DWORD):THandle;
```

Параметр LibFileName содержит имя подключаемой библиотеки. Примите во внимание, что совсем не обязательно передавать полный путь к файлу DLL.

Расширенный вариант функции загрузки библиотеки дополнен еще двумя параметрами. Параметр hFile зарезервирован для будущих версий функции и поэтому не используется (передавайте значение nil). Флаг Flags определяет действия при загрузке библиотеки и может принимать одно из следующих значений: DONT_RESOLVE_DLL_REFERENCES, LOAD_LIBRARY_AS_DATAFILE и LOAD_WITH_ALTERED_SEARCH_PATH.

Если флаг DONT_RESOLVE_DLL_REFERENCES не используется, то поведение функции LoadLibraryEx() аналогично LoadLibrary(), иначе операционная система не вызывает функцию DllEntryPoint() для процесса инициализации потока и завершения. Кроме того, система не загружает дополнительные выполняемые модули, которые вызываются данной библиотекой. Флаг поддерживается только Windows NT.

При включенном флаге LOAD_LIBRARY_AS_DATAFILE функция LoadLibraryEx() просто отображает файл библиотеки в адресное пространство. Функция загружает модуль, как будто это обычный файл данных. Обычно используется в том случае, если модуль DLL загружается с целью извлечь из него сообщения или ресурсы и не предполагается вызов каких-либо функций. В противном случае поведение функции идентично LoadLibrary().

Включенный флаг LOAD_WITH_ALTERED_SEARCH_PATH изменяет стратегию поиска файла библиотеки и, как правило, не используется.

В случае успеха обе функции возвращают дескриптор загрузившейся библиотеки. Если загрузить модуль не удается, выводится сообщение об ошибке (рис. 19.4).

Второй этап явной загрузки DLL – этап получения адреса конкретной функции, экспортируемой DLL. Для этого используется функция Windows API GetProcAddress():

Win32 API
```
function GetProcAddress(hModule: THandle; ProcName: string): Pointer;
```

Параметр hModule соответствует дескриптору библиотеки, полученному при помощи функции LoadLibrary() или LoadLibraryEx(). Имя функции описывается в параметре ProcName. Возвращаемое значение представляет собой указатель на вызываемую функцию.

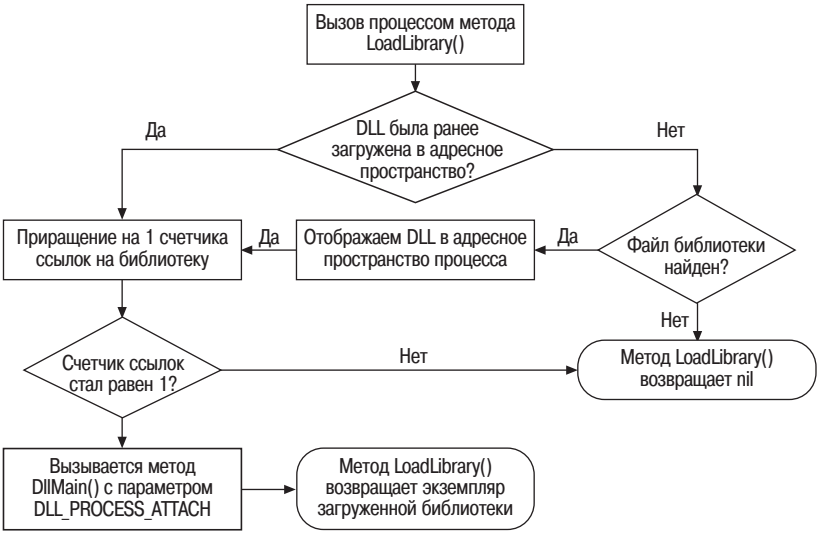


***Рис. 19.4.** Процесс загрузки библиотеки методом LoadLibrary( )*

В заключение необходимо выгрузить DLL из памяти системы. Для этого используют функцию Windows API `FreeLibrary()`:

Win32 API
```
function FreeLibrary(hModule:THandle): LongBool;
```

Параметр `hModule` нам уже знаком; это дескриптор ранее загруженной библиотеки. Функция `FreeLibrary()` уменьшает на 1 значение счетчика количества пользователей загруженного модуля в адресном пространстве процесса. Если счетчик примет нулевое значение, процесс выгружает библиотеку из памяти (рис. 19.5).

Как это выглядит на практике? Вернемся к виртуальному проекту, представленному на рис. 19.1, одной из задач которого было архивирование данных по окончании работы программы. Предположим, что библиотека архивации `LibZip.dll` написана ранее и за сжатие данных отвечает ее экспортируемая функция:

```
function ZipFile(fName, fPath : string) : Boolean; stdcall;
```

Способ организации явной загрузки динамической библиотеки приведен в листинге проекта:

```
. . .
public
    { Public declarations }
  end;
TZipFile=function(fName, fPath : string):boolean;
var frmMain : TfrmMain;
    hModule : THandle;
implementation

{$R *.DFM}
```

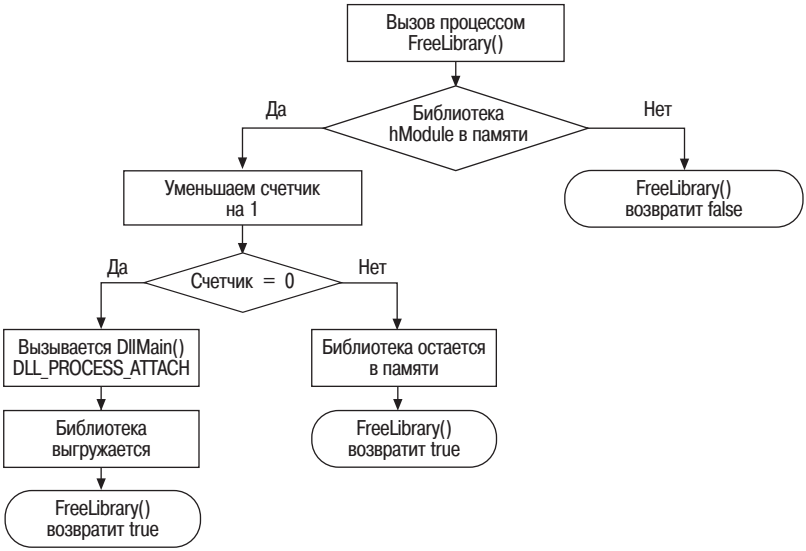


***Рис. 19.5.** Алгоритм выгрузки библиотеки из адресного пространства*

```
procedure TfrmMain.FormClose(Sender:TObject; var Action:TCloseAction);
var dwError:DWord;
    ZipFile:TZipFile;
begin
hModule:=LoadLibrary('LibZip');
if hModule=0 then
begin
dwError:=GetLastError();
ShowMessage(Format('Код ошибки %d',[dwError]));
Exit;
end;
@ZipFile:=GetProcAddress(hModule,'ZipFile');
if @ZipFile<>nil then
begin
  ZipFile('NewZip.Zip','C:\MyProject\ZipData');
  FreeLibrary(hModule);
end;
end;
```

Библиотека `LibZip.dll` загружается в момент закрытия главной формы проекта. Подключение библиотеки осуществляется функцией `LoadLibrary()`:

```
hModule:=LoadLibrary('LibZip');
```

Признаком успешного подключения будет ненулевое значение дескриптора `hModule`. В противном случае проект извещает пользователя о сбое и прекращает выполнение процедуры:

```
if hModule=0 then
begin
```

```
dwError:=GetLastError();
ShowMessage(Format('Код ошибки %d',[dwError]));
Exit;
end;
```

Код ошибки заносится в переменную `dwError` при помощи функции Windows API `GetLastError()`.

На этапе подключения динамической библиотеки к проекту весьма вероятно возникновение ошибки `ERROR_DLL_NOT_FOUND` *= 1157. Эта ошибка свидетельствует об отсутствии файла загружаемой библиотеки.*

После успешной загрузки файла `.dll` настает очередь вызова необходимой экспортируемой функции. Для этого в секции `interface` регистрируется прототип этой функции:

```
TZipFile = function(fName, fPath:string): boolean;
```

и объявляется в разделе переменных события `OnClose()` формы:

```
var . . .
    ZipFile:TZipFile;
```

Затем при помощи метода `GetProcAddress()` получаем адрес функции:

```
@ZipFile:=GetProcAddress(hModule,'ZipFile');
```

Далее осуществляем проверку на наличие такой функции в перечне экспортируемых функций и в случае успеха обращаемся к ней и... выгружаем динамически подключаемую библиотеку из памяти:

```
if @ZipFile<>nil then
begin
  ZipFile('NewZip.Zip','C:\MyProject\ZipData');
  FreeLibrary(hModule);
end;
```

Иногда вместо метода `FreeLibrary()` для выгрузки динамической библиотеки целесообразно использовать функцию `FreeLibraryAndExitThread()`.

```
procedure FreeLibraryAndExitThread(hModule : THandle; ExitCode : DWORD);
```

Особенность этого метода в том, что он кроме удаления библиотеки `hModule` из адресного пространства завершает работу потока `ExitCode`, вызванного из библиотеки.

Неявная загрузка DLL

Организация неявной загрузки динамически подключаемой библиотеки несколько проще и включает следующие этапы:

1. Регистрация в секции `interface` прототипов импортируемых функций и процедур с указанием соглашения о вызовах (см. табл. 19.2).
2. Объявление в разделе констант секции `implementation` имени подключаемой библиотеки и импортируемых из нее функций и процедур.

```
public
    { Public declarations }
end;

function SumFunc(iValue1, iValue2:integer):integer; stdcall;
function FrmShowAndFree(hWnd:THandle):Real; stdcall;

var frmMain : TfrmMain;

implementation

{$R *.dfm}
const MyDll='MyDll.Dll';

function SumFunc; external MyDll name 'SumFunc';
function FrmShowAndFree; external MyDll name 'FrmShowAndFree';
```

Непосредственно вызов экспортируемой функции ничем не отличается от вызова функции, объявленной внутри модуля.

```
procedure TfrmMain.btnSumFuncClick(Sender : TObject);
var Value1, Value2:integer;
begin
Value1:=StrToIntDef(EdValue1.Text,0);
Value2:=StrToIntDef(EdValue2.Text,0);
lbResult.Caption:=IntToStr(SumFunc(Value1, Value2));
end;

procedure TfrmMain.btnFrmShowAndFreeClick(Sender: TObject);
var dt : TDate;
begin
dt:=FrmShowAndFree(Application.Handle);
if dt>0 then lbData.Caption:=DateToStr(dt) else lbData.Caption:='';
end;
```

Создание библиотеки ресурсов

Еще одним полезным направлением в применении динамических библиотек является возможность хранить в них пиктограммы, растровые изображения, курсоры и т. д., тем самым получая универсальное и, что самое важное, разделяемое между несколькими приложениями хранилище ресурсов. Для того чтобы поместить в библиотеку ресурс, требуется (не поверите) в проекте DLL написать одну-единственную строку:

```
{$R MYRES.RES}
```

Это директива компилятору для подключения к библиотеке файла ресурсов `MYRES.RES`.

Для создания файла ресурсов с расширением `.RES` *программисты Delphi обычно используют утилиту Image Editor, входящую в стандартную поставку Delphi. Однако это не единственный путь. Сторонними разработчиками написаны десятки программ, решающих эту же задачу. Автор этих строк до сих пор пользуется программой Resource Workshop 4.5, созданной в недрах Borland еще в 1994 г.*

Многие редакторы ресурсов подготавливают файл ресурсов с расширением `.RC`. *Для того чтобы преобразовать его к формату, приемлемому для Delphi, вам понадобится компилятор ресурсов – файл* `BRCC32.EXE`, *хранящийся в каталоге \BIN. Для преобразования файла введите в командной строке Windows:* `brcc32 myres.rc`, *и вы получите файл* `myres.res`.

После того как мы научились создавать библиотеку ресурсов, нам осталось узнать, каким образом выудить из нее требуемую картинку. К счастью, и здесь код очень простой и не требует комментариев:

```
procedure TfrmMain.FormCreate(Sender : TObject);
var DLLInstance : INTEGER;
begin
  DLLInstance := LoadLibrary('myres.dll');
  Image1.Picture.Bitmap.LoadFromResourceName(DLLInstance, 'BITMAP1');
  FreeLibrary(DLLInstance);
end;
```

Анализ DLL

В состав поставки Delphi включена утилита Turbo Dump, работающая в режиме консольного приложения. Файл утилиты `tdump.exe` спрятан в каталоге *\BIN*. Рассматриваемая утилита умеет читать информацию из заголовка динамически подключаемой библиотеки и преобразовывать ее в текстовый вид. Утилита запускается в окне DOS, а синтаксис командной строки выглядит следующим образом:

```
TDUMP [опции] [файл DLL] [листинг] [опции]
```

Для получения пояснений по правилам применения утилиты воспользуйтесь стандартной командой DOS:

```
TDUMP /?
```

Самый простой вид команды предусматривает только ввод имени файла динамически подключаемой библиотеки:

```
TDUMP C:\MyProject\MyDll.Dll
```

В этом случае результаты анализа динамически подключаемой библиотеки будут выведены на экран монитора. При желании сохранить отчет в файл следует дополнить командную строку именем файла, в котором планируется сохранить листинг:

```
TDUMP C:\Windows\System\GDI32.DLL C:\GDI32_DLL.TXT
```

Наиболее важная информация листинга окажется в секциях `Import` и `Export`. Секция импорта предоставляет данные о взаимодействии анализируемой динамической библиотеки с другими библиотеками операционной системы; другими словами, здесь речь идет об импортируемых функциях (рис. 19.6).

```
***************************************************************
Section:             Import
  ImportLookUpTblRVA:00021068
  Time Stamp:        3546ABB0 : Wed Apr 29 08:25:20 1998
  Forwarder Chain:   FFFFFFFF (index of first forwarder reference)

Imports from KERNEL32.dll
    (hint = 025B) SetLastError
    (hint = 021B) SMapLS_IP_EBP_12
…
Imports from ADVAPI32.dll
    (hint = 00A9) RegSetValueExA
    (hint = 0082) RegCreateKeyExA
    (hint = 007E) RegCloseKey
    (hint = 0094) RegOpenKeyA
    (hint = 009D) RegQueryValueExA
```

Рис. 19.6. *Секция импортируемых функций библиотеки GDI32.DLL*

```
***************************************************************
Section:             Exports
  Flags:             00000000
  Time Stamp:        353EC26E : Thu Apr 23 08:24:14 1998
  Major Version:     0000
  Minor Version:     0000

Exports from GDI32.dll
  334 exported name(s), 339 export addresse(s).  Ordinal base is 100.
  Sorted by Name:
    RVA      Ord. Hint Name
    -------- ---- ---- ----
    0000284E  105 0000 AbortDoc =>
    000027FE  106 0001 AbortPath =>
    00002C61  107 0002 AddFontResourceA =>
    00007949  108 0003 AddFontResourceW =>
    00007976  109 0004 AngleArc =>
```

Рис. 19.7. *Начало секции экспортируемых функций библиотеки GDI32.DLL*

Секция экспорта информирует о функциях, ради которых в общем-то и создавалась данная библиотека. Например, если вы внимательно проанализируете список экспортируемых функций графической библиотеки `GDI32.DLL` (рис. 19.7), то встретите много старых знакомых (вспомните главу 10 «Графическая подсистема»).

К сожалению, выпытать дополнительную информацию об экспортируемых функциях DLL, такую как передаваемые параметры и возвращаемые значения, утилита Turbo Dump не способна, но она и так делает достаточно много.

Резюме

Динамически подключаемые библиотеки – один из столпов, на которых держится ОС Microsoft® Windows. Благодаря DLL расширяется функциональность приложения, упрощается управление проектом, достигается существенная экономия памяти и разделяются ресурсы между приложениями.

20

Процессы и потоки в среде Windows

История до сих пор хранит легенду об удивительных способностях знаменитого полководца, сенатора, а затем и властелина Великой Римской империи Гая Юлия Цезаря. Но славу ему принесли не только военные победы, грандиозные походы и роман с Клеопатрой. Наш герой умудрялся одновременно заниматься несколькими делами и, как гласит молва, весьма успешно. Воистину потрясающее качество, к сожалению редко присущее обычному человеку. Вспомните, сколько усилий тратил Труффальдино на то, чтобы услужить двум господам одновременно.

Что недоступно нам, с легкостью реализуют современные ОС. Большинство серьезных приложений под Windows, даже не имея представления о подвигах упомянутой выше исторической личности, творят не меньшие чудеса. Программа Microsoft Word одновременно со вводом текста разбивает текст на страницы и контролирует орфографию. Еще большим кудесником является сама ОС Windows. Она одновременно (по крайней мере, на первый взгляд) управляет десятками процессов: реагирует на нажатия клавиш на клавиатуре, в соответствии с расписанием запускает программы обслуживания компьютера, обеспечивает вывод задания на печать, воспроизводит звуковой компакт-диск, отправляет электронную почту, контролирует работу текстового редактора и т. д.

Процессы и многозадачность

В недавнем прошлом в старой доброй DOS пользователи имели право одновременно запускать не более одной программы. И только дождавшись завершения ее работы, можно было выполнить другую программу. В современной ОС одновременно работают десятки процессов, количество которых определяется вычислительной мощностью компьютера.

В самом общем случае ОС включает в себя блок кода, решающий задачи создания и удаления процессов. Современные операционные системы, такие как UNIX, OS/2, Windows, научились распределять вычислительные воз-

можности микропроцессора. Но неверно утверждение, что операционная система выделяет приложению некоторый отрезок времени. **Операционная система распределяет процессорное время между процессами, существующими в данный момент.**

Прежде чем приступить к изложению основной части материала, прокомментируем ряд старых и новых терминов. **Приложение**, или, как часто его называют, **программа**, – это статическая последовательность команд. При запуске программы в ОС Win32 создается отдельный **процесс** (process), или, как его иногда называют, **задача** (task). Процесс состоит из:

- исполняемой программы с ее кодом и данными;
- **виртуального адресного пространства** (address space), представляющего собой набор адресов виртуальной памяти, доступных для процесса;
- ресурсов, выделяемых процессу;
- как минимум одного потока, именуемого **потоком управления** (thread of execution), или **главным потоком** процесса. Именно поток управления и требует от операционной системы времени на обслуживание процесса.

Операционная система стремится максимально изолировать процессы друг от друга. Именно для этого каждому процессу выделяется свое виртуальное адресное пространство, причем весьма значительное (в Win32 процесс может адресовать до 2^{32} байт памяти). Такая забота в первую очередь продиктована желанием Windows защитить процесс от возможных некорректных действий его коллег.

В момент запуска задачи ОС создает дескриптор – специальную структуру, описывающую стартующий процесс. **Дескриптор процесса** содержит информацию, позволяющую Win32 управлять процессом. В частности в дескрипторе хранится:

- идентификатор процесса (PID – process identificator);
- информация о ресурсах, которыми владеет или собирается завладеть процесс;
- данные о приоритете процесса;
- **контекст задачи**, представляющий собой специальную защищенную область памяти, в которую записываются значения регистров процессора в момент приостановки задачи и откуда считываются данные в момент возобновления выполнения задачи.

За один и тот же интервал времени, называемый **квантом времени**, один процессор реально может выполнять не более одной задачи. Вместе с тем **многозадачная** (multitasking) ОС вполне способна одновременно обслуживать несколько процессов. Это достигается благодаря переключению процессора компьютера с исполнения одной задачи на другую. Такое переключение называется **переключением контекста** (context switching). При этом важно помнить, что время выделяется не процессу, а принадлежащим ему потокам (рис. 20.1). И чем выше производительность процессора, тем более правдоподобна иллюзия одновременного выполнении нескольких программ.

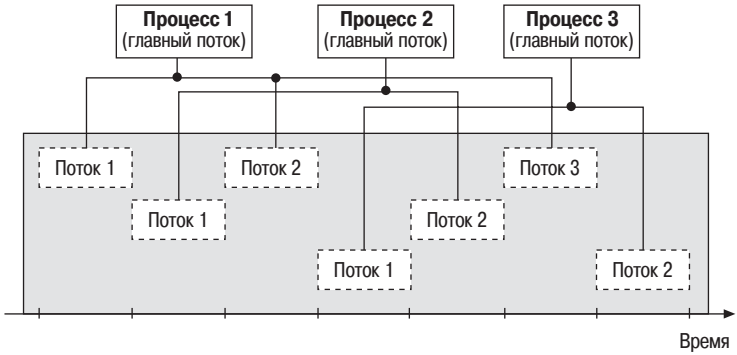


***Рис. 20.1.** Графическое представление многозадачности*

Итак, многозадачность – это совместное использование процессорного времени несколькими задачами (процессами), ожидающими выполнения. В уже устаревших версиях Microsoft Windows 3.x была реализована так называемая **кооперативная многозадачность** (cooperative multitasking), где за переключение контекста отвечала не операционная система, а сами процессы. Это означает, что все приложения по очереди используют процессор, передавая управление друг другу по истечении определенного интервала времени. Такой подход был чреват существенными недостатками. Во-первых, при «зависании» элементарной задачи рушилась вся операционная система. Во-вторых, кооперативная многозадачность требовала от программистов особой тщательности и аккуратности при построении кода передачи управления.

В современных версиях Windows в основу распределения процессорного времени между процессами положен принцип **вытесняющей многозадачности** (preemptive multitasking). Идея этого вида обслуживания в том, что решение на предоставление кванта времени тому или иному процессу принимается операционной системой, а точнее ее частью, называемой **диспетчером задач**.

Выбрав процесс из очереди, ОС загружает его контекст (элемент структуры дескриптора процесса, в котором хранятся данные о состоянии регистров микропроцессора на момент приостановки процесса) и запускает процесс на выполнение. По истечении отпущенного на процесс времени диспетчер задач приостанавливает его выполнение, сохраняет контекст и передает вычислительные мощности компьютера другому процессу. При вытесняющей многозадачности «зависание» одного процесса, как правило, не оказывает существенного влияния на выполнение остальных задач.

Создание процесса

Главную роль в формировании нового процесса в Windows выполняет метод `CreateProcess()`. Функция создает процесс и поток управления. Сам по себе этот метод достоин отдельной главы, так как обладает широчайшим спектром настроек и, кроме того, может использоваться для синхронизации работы нескольких процессов и потоков. К сожалению, в рамках этой книги мы вынуждены ограничиться лишь вводным описанием.

Win32 API
```
function CreateProcess(ApplicationName, CommandLine : pAnsiChar;
ProcessAttributes, ThreadAttributes : PSecurityAttributes; InheritHandles
: boolean; CreationFlags : Cardinal; Environment : Pointer;
CurrentDirectory : pAnsiChar; const StartupInfo : TStartupInfo;
var ProcessInformation : TProcessInformation) : Boolean;
```

В первых двух аргументах функции указываются имя запускаемого приложения и (при необходимости) передаваемые ему в командной строке параметры.

Третий и четвертый параметры метода `CreateProcess()` определяют атрибуты безопасности создаваемого процесса и главного потока, порожденного этим процессом. Физически передаваемые в аргументы значения представляют собой указатели на специальную структуру атрибутов защиты:

```
TSecurityAttributes = _SECURITY_ATTRIBUTES;
PSecurityAttributes = ^TSecurityAttributes;
  _SECURITY_ATTRIBUTES = record
    Length : cardinal;              // размер структуры в байтах
    SecurityDescriptor : Pointer;  // указатель на дескриптор защиты процесса
    InheritHandle : boolean;       // признак наследования возвращаемого дескриптора
  end;
```

Эти атрибуты применяются только в операционных системах, основанных на архитектуре Windows NT. Одной из особенностей ОС этого типа является возможность разграничения доступа пользователей к различным ее объектам. Так, например, пользователь, не обладающий правами администратора, не может просматривать системные каталоги или работать с каким-либо ресурсом. Более того, обычный пользователь не имеет права запускать процессы или управлять некоторыми процессами, имеющими высокие, не доступные для него атрибуты безопасности. Если вместо указателя этой функции передается `nil`, то включается защита по умолчанию и дескриптор создаваемого процесса наследоваться не может.

Особый интерес в структуре `PSecurityAttributes` представляет второе поле – указатель на дескриптор защиты. В этом дескрипторе сосредоточена информация о безопасности объекта. Для инициализации дескриптора используется метод `InitializeSecurityDescriptor()`, например:

```
var  SD : TSecurityDescriptor;
     SA : TSecurityAttributes;
begin
InitializeSecurityDescriptor(@SD, SECURITY_DESCRIPTOR_REVISION);
SA.nLength:=SizeOf(TSecurityAttributes);
SA.lpSecurityDescriptor:=@SD;
SA.bInheritHandle:=True;
…
end;
```

В операционных системах Windows Me/98/95 атрибуты безопасности игнорируются. Поэтому если ваше приложение рассчитано только для работы в этих (неумолимо устаревающих) версиях ОС, то в подобные параметры безопасности допускается передача неопределенных указателей `nil`.

Пятый параметр InheritHandles разрешает или запрещает создаваемому процессу наследовать дескрипторы из вызывающего процесса.

Набор флагов CreationFlags определяет флаги создания процесса и флаги приоритета. Шесть возможных значений приоритета приведены в следующем подразделе, а с флагами создания процесса можно познакомиться, обратившись к справочной системе. Параметр Environment – это указатель на настройки окружения нового процесса. CurrentDirectory – путь к текущему каталогу нового процесса. Параметр StartupInfo – это специальная структура, описывающая характеристики окна создаваемого приложения. ProcessInformation – указатель на структуру, используемую созданным процессом. Если функция выполнилась успешно, метод возвращает значение true.

Ниже приведен пример запуска блокнота (файл Notepad.exe). В командной строке процессу передается указание открыть файл autoexec.bat.

```
var StartupInfo : TStartupInfo;
    ProcessInformation : TProcessInformation;
    SD : TSecurityDescriptor;
    SA : TSecurityAttributes;

begin
//инициализация и заполнение атрибутов безопасности
InitializeSecurityDescriptor(@SD, SECURITY_DESCRIPTOR_REVISION);
SA.nLength:=SizeOf(TSecurityAttributes);
SA.lpSecurityDescriptor:=@SD;
SA.bInheritHandle:=True;

FillChar(StartupInfo, SizeOf(StartupInfo), #0);   //очищаем структуру StartupInfo
CreateProcess(PChar('c:\windows\notepad.exe'),    //запускаемое приложение
              PChar('open c:\autoexec.bat'),      //командная строка
              @SA, @SA, false,
              NORMAL_PRIORITY_CLASS,              //нормальный приоритет
              nil, nil, StartupInfo, ProcessInformation);
end;
```

Завершение процесса

Наилучший вариант завершения процесса – когда выполнение процесса заканчивается только после возврата управления входной функцией первичного потока. Только в этом случае можно утверждать, что процесс корректно освободит все свои объекты, а ОС очистит занимаемую стеком главного потока память и уменьшит счетчик на 1.

Но если процесс не желает уходить из жизни добровольно, то для завершения процесса с крайней неохотой применяют одну из функций: ExitProcess() или TerminateProcess(). Первый метод дает процессу шанс «покончить с собой» самостоятельно. Второй метод может быть вызван родительским процессом или любым другим процессом, знающим дескриптор разрушаемого процесса.

Win32 API

```
procedure ExitProcess(ExitCode : cardinal);
function TerminateProcess(hProcess : THandle;
                          ExitCode : cardinal) : Boolean;
```

В обоих методах параметр ExitCode – код завершения процесса. Это своеобразный последний привет, отправляемый в адрес породившего процесса. После завершения работы родительский процесс может определить код при помощи функции GetExitCodeProcess().

Сбор информации о процессах, протекающих в системе

В перечень методов Windows, обслуживающих процессы, входят информационные функции Tool Help. Этих методов немногим более десятка. Они нацелены на предоставление нам данных о задачах, работающих на момент опроса в системе. Особо подчеркну словосочетание «на момент опроса», и вот почему. Windows – динамическая система, в которой задачи могут рождаться и умирать чуть ли не ежесекундно. Поэтому функции Tool Help делают моментальный снимок состояния системы в указанный программистом момент времени и затем «препарируют» полученные данные, извлекая из снимка требуемую информацию.

Если есть необходимость в непрерывном отслеживании состояния системы, применяйте таймер, «фотографирующий» ОС с определенной периодичностью. Роль фотографа отводится методу:

Win32 API
```
function CreateToolhelp32Snapshot(Flags, ProcessID : Cardinal) : THandle;
```

Параметр Flags определяет, какую информацию мы планируем включать в снимок (табл. 20.1). Если исследуется только один процесс, то его дескриптор указывается в параметре ProcessID. При успешном выполнении метод возвращает указатель на структуру в памяти, называемую *снимком*.

Таблица 20.1. Возможные значения параметра Flags

Значение	Описание
TH32CS_INHERIT	Наследовать дескриптор снимка.
TH32CS_SNAPALL	Включить в снимок всю информацию.
TH32CS_SNAPHEAPLIST	Включить в снимок данные о кучах в процессе ProcessID.
TH32CS_SNAPMODULE	Включить в снимок данные о модулях в процессе ProcessID.
TH32CS_SNAPPROCESS	Включить в снимок данные о процессах.
TH32CS_SNAPTHREAD	Включить в снимок данные о потоках.

Для исследования имеющегося снимка на предмет получения информации о процессах применяют работающие в паре методы Process32First() и Process32Next():

Win32 API
```
function Process32First(Snapshot : THandle;
                        var lppe : TProcessEntry32) : boolean;
function Process32Next(Snapshot : THandle;
                       var lppe : TProcessEntry32) : boolean;
```

Параметры методов идентичны. Первый из них – дескриптор снимка, второй – указатель на структуру TProcessEntry32, в которую они помещают результаты исследований. Как мы уже отмечали, сбор данных производится

совместно. Первоначально вызывается метод Process32First(), проверяющий наличие интересующей нас информации в снимке. Если такая информация имеется, то метод возвращает true. Затем наступает черед функции Process32Next(), которая вызывается в цикле до тех пор, пока им не будет возвращено значение false. Это признак того, что все данные собраны. Рассмотрим структуру TProcessEntry32:

```
type
TProcessEntry32 = packed record
    dwSize: DWORD;              //размер записи
    cntUsage: DWORD;            //счетчик ссылок на процесс
    th32ProcessID: DWORD;       //идентификатор процесса
    th32DefaultHeapID: DWORD;   //идентификатор кучи процесса
    th32ModuleID: DWORD;        //модуль, связанный с процессом
    cntThreads: DWORD;          //количество дочерних потоков процесса
    th32ParentProcessID: DWORD; //родительский процесс
    pcPriClassBase: Longint;    //базовое значение приоритета процесса
    dwFlags: DWORD;             //не используется
    szExeFile: array[0..MAX_PATH - 1] of Char;   //имя исполняемого файла
end;
```

Перед первым вызовом запись типа TProcessEntry32 *должна быть инициализирована:*

```
var lppe : TProcessEntry32;
…
lppe:=SizeOf(TProcessEntry32);
```

Рассмотрим максимально упрощенный пример сбора данных о процессах системы. Создайте новое приложение и разместите на его поверхности компонент TListView. Переведите его в табличный режим просмотра (ViewStyle=vsReport) и в редакторе колонок создайте три колонки для отображения: идентификатора процесса, идентификатора родительского процесса и количества дочерних потоков. Подключите к проекту модуль tlhelp32 и повторите приведенный листинг:

```
uses tlhelp32;
…
procedure TfrmMain.FormCreate(Sender : TObject);
var PrcEntry : TProcessEntry32;
    Snapshot : THandle;
begin
PrcEntry.dwSize:=SizeOf(PrcEntry);         //установка размера структуры
Snapshot:=CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS,0); //получаем снимок процессов
{начало цикла перебора процессов в снимке}
if Process32First(Snapshot, PrcEntry)=true then
   repeat
        {в расширенном списке создаем новый элемент с информацией о процессе}
        with ListView1.Items.Add do
        begin
         Caption:=PrcEntry.szExeFile;                          //имя процесса
         SubItems.Add(IntToStr(PrcEntry.th32ProcessID));       //PID процесса
         SubItems.Add(IntToStr(PrcEntry.th32ParentProcessID)); //родительский PID
         SubItems.Add(IntToStr(PrcEntry.cntThreads));          //число потоков
```

```
        end;
    until not Process32Next(Snapshot, PrcEntry);   //условие продолжения перебора
CloseHandle(Snapshot);                 //освобождение дескриптора снимка процессов
end;
```

Кроме рассмотренных функций в состав Tool Help входят методы, позволяющие получить информацию о существующих в данный момент:

- *потоках* (`Thread32First` *и* `Thread32Next`);
- *модулях* (`Module32First` *и* `Module32Next`);
- *кучах* (`Heap32First`, `Heap32Next`, `Heap32ListFirst` *и* `Heap32ListNext`).

Приоритет процесса

В многозадачной операционной системе количество квантов процессорного времени, выделяемого процессу, в первую очередь определяется **приоритетом процесса**. Зависимость прямо пропорциональна: чем выше приоритет, тем большего внимания его удостоит процессор. Диспетчер задач вытесняет текущий процесс при появлении его коллеги с более высоким приоритетом. Система Windows поддерживает шесть категорий приоритетов процессов:

1. фоновый (`IDLE_PRIORITY_CLASS`);
2. нормальный заднего плана (`BELOW_NORMAL_PRIORITY_CLASS`);
3. нормальный (`NORMAL_PRIORITY_CLASS`);
4. нормальный переднего плана (`ABOVE_NORMAL_PRIORITY_CLASS`);
5. высокий (`HIGH_PRIORITY_CLASS`);
6. процесс реального времени (`REALTIME_PRIORITY_CLASS`).

Создание процессов с приоритетами `BELOW_NORMAL_PRIORITY_CLASS` *и* `ABOVE_NORMAL_PRIORITY_CLASS` *возможно только начиная с ОС Windows 2000.*

По умолчанию при создании нового процесса ему присваивается нормальный приоритет.

Для системного программиста большой практический интерес представляет возможность ускорения или замедления выполняющегося процесса. Для реализации этой коварной цели достаточно сделать два шага: во-первых, получить дескриптор процесса и, во-вторых, изменить его приоритет. Для выяснения дескриптора процесса, созданного нашим приложением, предназначен метод Win32 API:

```
function GetCurrentProcess : THandle;
```

Функция в аргументах не нуждается и возвращает дескриптор процесса, в рамках которого она была вызвана. Для выяснения дескриптора стороннего процесса предназначена функция:

```
function OpenProcess(DesiredAccess : cardinal; InheritHandle : Boolean;
ProcessId : cardinal) : THandle;
```

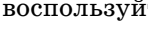
Для того чтобы воспользоваться услугами этого метода, как минимум надо знать идентификатор процесса `ProcessId`. Одним из вариантов решения этой

задачи может стать применение функций из состава недавно рассмотренного Tool Help, предоставляющего информацию о протекающих в системе процессах. Первый параметр DesiredAccess определяет права доступа к процессу, например для полного доступа применяйте константу PROCESS_ALL_ACCESS. Если параметр InheritHandle установлен в true, то порожденные процессы будут наследовать права доступа этого процесса.

Тем или иным способом получив дескриптор интересующего нас процесса, приступим к главному: поменяем приоритет задачи. Для этого вызовем метод:


```
function SetPriorityClass(Prc : THandle; PriorityClass : Cardinal) : Boolean;
```

Первый аргумент требует дескриптор процесса, а второй аргумент – это константа приоритета. Если планируется выяснить лишь приоритет процесса, воспользуйтесь функцией Windows API:



```
function GetPriorityClass(Prc : THandle) : Cardinal;
```

В зависимости от приоритета процесса функция возвратит одну из констант. Приведем пример, демонстрирующий способ выяснения приоритета выполняющейся задачи:

```
procedure TfrmMain.btnGetPriorityClassClick(Sender: TObject);
var prc : THandle;
    msg : string;
begin
prc:=GetCurrentProcess();                    // получаем дескриптор процесса
    case GetPriorityClass(prc) of
     IDLE_PRIORITY_CLASS     : msg:='IDLE_PRIORITY_CLASS';
     NORMAL_PRIORITY_CLASS   : msg:='NORMAL_PRIORITY_CLASS';
     HIGH_PRIORITY_CLASS     : msg:='HIGH_PRIORITY_CLASS';
     REALTIME_PRIORITY_CLASS : msg:='REALTIME_PRIORITY_CLASS';
    end;
ShowMessage(msg);
end;
```

Пример, демонстрирующий способ смены приоритета процесса:

```
var prc : THandle;
begin
prc:=GetCurrentProcess();                    // дескриптор процесса
SetPriorityClass(prc, IDLE_PRIORITY_CLASS);  // низший фоновый приоритет
end;
```

Время выполнения процесса

При анализе производительности системы часто возникает необходимость осуществить хронометраж процесса, т. е. выяснить, какое время уделил процессор компьютера тому или иному процессу. Эта задача достаточно просто решается в ОС Windows NT, так как в ней реализована специализированная функция:

```
function GetProcessTimes(Prc : THandle; var CreationTime, ExitTime,
KernelTime, UserTime : TFileTime) : boolean;
```

Здесь Prc – дескриптор процесса, CreationTime – время создания, ExitTime – время завершения процесса (если он остановлен), KernelTime – время выполнения процесса в режиме ядра (здесь хранится суммарное время выполнения всех потоков в режиме ядра), UserTime – время выполнения процесса в пользовательском режиме. При работе в Delphi обработка возвращаемых этой функцией данных сопряжена с некоторыми трудностями преобразования данных: последние четыре параметра (в которые помещаются результаты выполнения метода) представлены типом данных TFileTime.

```
type
TFileTime = record
dwLowDateTime  : Cardinal;
dwHighDateTime : Cardinal;
end;
```

В этом типе данных время представлено в несколько непривычном для программиста в среде Delphi формате: 32-разрядные элементы записи объединяются в учетверенное слово и образуют хранилище для 64-разрядного значения. За единицу измерения принят интервал в 100 нанасекунд, а за точку отсчета – полночь 1-го января 1601 года. Какими соображениями руководствовались в корпорации Microsoft, создавая такой потрясающий способ хранения времени, я не знаю, но только подчеркну, что работать с такими огромными значениями можно только с использованием типа данных INT64 или ряда специализированных методов. Например, можно воспользоваться функциями Win32 API FileTimeToSystemTime() и SystemTimeToFileTime():

```
function FileTimeToSystemTime(FileTime : TFileTime;
                              var SystemTime : TSystemTime) : boolean;
function SystemTimeToFileTime(SystemTime : TSystemTime;
                              var FileTime : TFileTime) : Boolean;
```

Методы обеспечивают преобразование TFileTime в TSystemTime, и наоборот. К сожалению, на этом наши мучения не закончились, так как тип данных TSystemTime по-прежнему не совместим с нашим любимым TDateTime. В этом нетрудно убедиться, взглянув на структуру TSystemTyme:

```
type PSystemTime = ^TSystemTime;
    TSystemTime = record
    wYear: Word;          //год, например 2005
    wMonth: Word;         //месяц, например 1,2, …,12
    wDayOfWeek: Word;     //день недели 1,2,…7
    wDay: Word;           //день месяца 1,2,…31
    wHour: Word;          //час 1,2…23
    wMinute: Word;        //минута 1,2,…59
    wSecond: Word;        //секунда 1,2,…59
    wMilliseconds: Word;  //миллисекунда 1,2,…999
  end;
```

Однако в этом формате записи время «разложено по полочкам», и перевод значений, представленных структурой TSystemTyme, в TDateTime – это уже дело техники; вспомните методы Delphi EncodeDateTime() и DecodeDateTime().

В заключение предлагаю пример использования данных функций:

```
var prc : THandle;
    CreationTime, ExitTime, KernelTime, UserTime : TFileTime;
    SystemTyme : TSystemTime;
    DateTime   : TDateTime;
begin
prc:=GetCurrentProcess();
GetProcessTimes(prc, CreationTime, ExitTime, KernelTime, UserTime);
FileTimeToSystemTime(CreationTime, SystemTyme);
DateTime:=EncodeDateTime(SystemTyme.wYear,
                         SystemTyme.wMonth,
                         SystemTyme.wDay,
                         SystemTyme.wHour,
                         SystemTyme.wMinute,
                         SystemTyme.wSecond,
                         SystemTyme.wMilliseconds);
{…}
end;
```

Понятие потока, многопоточность

Что представляет собой поток? Поток отображает одну из подзадач, существующих внутри процесса. Мы уже упоминали, что процесс владеет как минимум одним потоком – потоком управления (или, как его иногда именуют, главным потоком). В свою очередь поток управления имеет право создавать свои дочерние потоки.

Настал черед ввести еще один термин – **многопоточность**. Многопоточность – это поддержка нескольких потоков внутри одного процесса. Цели многопоточности самые благие – распределение частных задач между дочерними потоками в надежде ускорить работу приложения. Допустим, мы вызвали редактор электронных таблиц Microsoft Excel. Для этого Windows создает отдельный процесс. В рамках данного процесса мы осуществляем редактирование данных в ячейках таблицы (первый поток), «одновременно» электронная таблица производит вычисления (второй поток), идет подготовка листа к печати (третий поток) и т. д. Потоки выполняют разные части единой задачи в общем виртуальном адресном пространстве процесса, однако при этом они обладают своим собственным стеком, а начиная с Windows 2000 и собственным дескриптором защиты.

В самом общем случае можно говорить о следующих составляющих потока:

- идентификатор потока;
- **контекст потока**, в свою очередь состоящий из регистра, стеков и собственной области памяти потока.

Различают два основных типа потоков: **независимые** и **взаимодействующие**. Независимые потоки предназначены для выполнения различных задач, не разделяющих совместные ресурсы. Взаимодействующие потоки конкурируют за один и тот же ресурс, например переменную, файл или массив.

Не совсем корректно утверждать, что в рамках однопроцессорной системы могут существовать абсолютно независимые потоки. Ведь все потоки пользуются общим системным ресурсом – процессорным временем. Однако в Win32 конкуренция за этот ресурс контролируется операционной системой, так что мы его не будем принимать во внимание.

Элементарный поток – класс TThread

Для программной реализации потока в Delphi описан специальный класс `TThread`. Поскольку класс абстрактный, в первозданном виде он нам никогда не понадобится. Однако созданные на его основе объекты превосходно инкапсулируют полнофункциональные потоки.

На палитре компонентов Delphi элемент управления `TThread` вы не найдете. Для того чтобы самое обычное приложение сделать многопоточным, требуется к уже существующему приложению добавить специальный программный модуль. Для этого после создания нового приложения выберите пункт меню File → New → Other и на вкладке New диалогового окна New Items выберите пиктограмму ThreadObject (рис. 20.2).

После нажатия кнопки OK в новом диалоговом окне требуется ввести имя нового класса – потока, например с именем `TMyThread`. В результате Delphi создает представленный ниже шаблон, предназначенный для описания потока.

Обратите внимание, что в секции `protected` среда Delphi уже подготовила перекрытую процедуру `Execute()`. Именно в теле этой процедуры программист располагает программный код, который станет исполняться в отдельном потоке. Если подзадача вызывается многократно, то строки кода помещаются внутрь тела цикла, выход из которого завершает работу потока.

```
unit Unit2;

interface
```

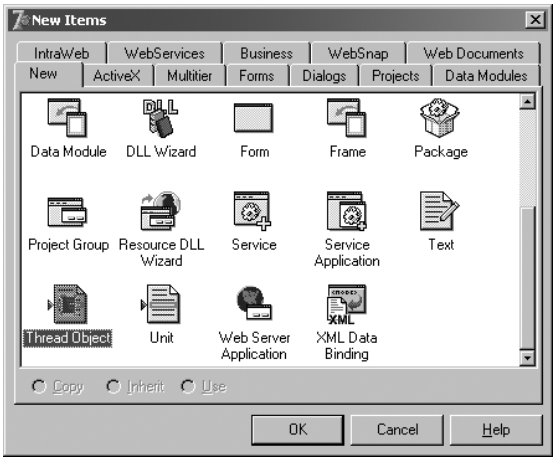


***Рис. 20.2.** Создание многопоточной программы*

```
uses Classes;
type
  TMyThread = class(TThread)   // класс TMyThread создается
                               // на базе абстрактного класса TThread
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;

implementation

procedure TMyThread.Execute;
begin
  {место для исходного кода, обслуживающего поток}
end;
end.
```

Для запуска потока вызывается метод Execute. Говоря образно, эта процедура заставит биться сердце потока.

```
procedure Execute;
```

При старте приложения экземпляр класса TMyThread автоматически не создается. Для рождения нового потока Delphi потребует явного обращения к его конструктору:

```
constructor Create(CreateSuspended : Boolean);
```

Единственный параметр конструктора определяет, каким образом будет осуществлен запуск потока. Если параметр CreateSuspended установлен в false, поток стартует сразу же после создания экземпляра класса. Если же передается значение true, то запуск потока откладывается до его явного вызова из какой-нибудь части кода программы.

```
var Thread: TMyThread;
…
procedure TfrmMain.FormCreate(Sender: TObject);
begin
MyThread:=TMyThread.Create(true);
end;
```

Для временной приостановки выполнения потока используйте процедуру:

```
procedure Suspend;
```

Для возобновления работы приостановленного потока:

```
procedure Resume;
```

Причем возобновление выполнения потока произойдет именно с того места (с той строки кода), где он был приостановлен. Два названных метода дублируются свойством:

```
property Suspended : Boolean;
```

Если установить свойство в true, поток приостановится, в false – вновь активизируется. Для временной приостановки потока программисты Delphi час-

то используют процедуру Win32 API `Sleep()`. В качестве параметра передается количество миллисекунд, на которые необходимо заморозить подзадачу.

```
procedure Sleep(milliseconds : Cardinal);
```

Поток, переведенный в состояние сна с помощью этого метода, не снижает производительность системы, так как не требует выделения ему квантов времени. Для полной остановки потока вызывается метод:

```
procedure Terminate;
```

Этот способ остановки называется мягким. Процедура лишь присваивает значение `true` свойству:

```
property Terminated: Boolean;
```

В этом случае Delphi не настаивает на немедленном прекращении работы потока и разрешает ему достичь логического конца. Поясню это утверждение на примере. Обычно исполняемый потоком код заключается в рамки цикла `while..do` или `repeat..until`, условием выхода из которого будет переход свойства `Terminated` в состояние `true`.

```
procedure TMyThread.Execute;
begin
    while Terminated=false do     //условие продолжения цикла
    begin
    {исполняемый код}
    end;
end;
```

Как видите, если в момент вызова метода `Terminated()` выполнялся код, расположенный внутри конструкции `begin..end`, то его выполнение не будет немедленно прервано. Останов произойдет только при проверке условия цикла. Раз есть мягкий способ остановки потока, то в недрах Windows можно обнаружить и «жесткий» способ. Если вы настаиваете на немедленном завершении потока, пригодится функция из арсенала Windows API:

```
function TerminateThread(hThread : THandle; ExitCode : Cardinal): boolean;
```

Здесь `hThread` – дескриптор потока, `ExitCode` – программный код выхода из потока (в простейшем случае передавайте значение 0). Если функция завершается успешно, она возвращает ненулевое значение. Дескриптор потока (так необходимый для работы с методами Win32 API) вы найдете в свойстве:

```
property Handle: THandle;
```

Например:

```
TerminateThread(Thread1.Handle, 0);
```

Останов потока сопровождается вызовом обработчика события:

```
property OnTerminate : TNotifyEvent;
```

Если установить в `true` свойство `FreeOnTerminate`, то сразу после останова поток уничтожится; в противном случае за вызов деструктора потока отвечаете вы.

```
property FreeOnTerminate : Boolean;
```

Если выполнение метода Execute() прервалось в результате ошибки и ошибка не была обработана в рамках этого метода, то в свойство FatalException (только для чтения) будет записан объект ошибки.

```
property FatalException: TObject; //только для чтения
```

Это свойство можно применять по крайней мере для проверки корректности завершения метода Execute(). Еще одним способом, косвенно проверяющим корректность завершения работы потока, является использование свойства:

```
property ReturnValue: Integer;
```

Это служебная переменная, в которую можно записывать данные, например о количестве запусков потока. Свойство ReturnValue часто применяется совместно с методом:

```
function WaitFor: LongWord;
```

Функция предназначена для организации ожидания, пока какой-нибудь другой поток не выполнит поставленную перед ним задачу. Она возвращает значение, содержащееся в свойстве ReturnValue ожидаемого потока.

Приоритет потока

Приоритет потока формируется из двух составляющих: приоритета процесса, которому принадлежит поток, и непосредственно приоритета этого потока. Приоритет потока определяется свойством:

```
property Priority : TThreadPriority;
```

Возможные значения приоритета приведены в табл. 20.2.

Таблица 20.2. Описание приоритета потока, класс TThreadPriority

Значение	Описание
tpIdle	Фоновый приоритет. Поток с таким приоритетом выполняется только при отсутствии хоть какой-либо загрузки процессора.
tpLowest	Низкий приоритет.
tpLower	Пониженный приоритет.
tpNormal	Нормальный приоритет, соответствующий поведению обычной программы.
tpHigher	Повышенный приоритет. Использование потоков с таким или более высоким приоритетом весьма критично скажется на производительности остальных потоков.
tpHighest	Высочайший приоритет. Применяется только для потоков, которые выполняются в кратчайшие промежутки времени.
tpTimeCritical	Приоритет реального времени. Приоритет такого процесса выше, чем у многих процессов ОС.

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
 Thread1:=TMyThread.Create(TRUE);
 Thread1.Priority:=tpIdle;
 Thread1.Resume;
end;
```

Старайтесь не создавать без особой на то необходимости потоки с приоритетом выше нормального. Это может привести к тому, что системные ресурсы будут перенацелены на обслуживание данной задачи, что в свою очередь снизит общую производительность системы.

Вновь вернемся к приоритетам потоков, порожденных внутри главного процесса. Приоритеты потоков, описанные в табл. 20.2, часто называют **относительными приоритетами**. Такое название объясняется тем, что два потока, скажем с приоритетом tpNormal, но порожденные процессами с различными приоритетами, не смогут претендовать на одинаковое процессорное время. **Реальный приоритет потока определяется приоритетом порождающего его процесса и своим приоритетом**. В результате приоритет потока оценивается значением, лежащим в диапазоне от 1 до 31 (где 1 – самый низкий приоритет). Примерные значения приоритетов потоков относительно породивших их процессов представлены в табл. 20.3.

Таблица 20.3. Приоритеты процессов и их потоков

Приоритет процесса	Приоритет потока					
	Фон	Низший	Пониженный	Нормальный	Повышенный	Высший
Фоновый	1	2	3	4	5	6
Нормальный заднего плана	1	5	6	7	8	9
Нормальный	1	6	7	8	9	10
Нормальный переднего плана	1	7	8	9	10	11
Высокий	1	11	11	12	14	15
Реального времени	16	22	23	24	25	26

В состав Delphi входит специальная утилита, предназначенная для просмотра потоков приложения. Для обращения к утилите выберите пункт меню View → Debug Windows → Threads.

Время выполнения потока

Операционная система позволяет оценить время процессора, затраченное на любой из потоков процесса. Для этого предназначен метод:

Win32 API

```
function GetThreadTimes(hThread : THandle; var CreationTime, ExitTime,
KernelTime, UserTime : TFileTime) : boolean;
```

Параметры и порядок применения метода аналогичны изученной в начале главы функции GetProcessTimes(), за исключением того, что здесь первый параметр – это дескриптор интересующего нас потока, а не процесса.

Синхронизация потока с методами VCL

Если из тела процедуры Execute() необходимо осуществить вызов какого-либо метода или обратиться к свойству компонента библиотеки VCL, то **обязательно** используйте синхронизирующий метод:

```
procedure Synchronize(Method : TThreadMethod);
```

Впрочем, об этом вам напомнит и сама среда Delphi: в каждый создаваемый модуль Thread Object вставляется предупреждение об этом с примером вызова метода Synchronize():

```
{ Important: Methods and properties of objects in VCL or CLX can only be used in a
method called using Synchronize, for example,
    Synchronize(UpdateCaption);
and UpdateCaption could look like,

    procedure TMyThread.UpdateCaption;
    begin
      Form1.Caption := 'Updated in a thread';
    end; }
```

Задача метода – защитить потоки от гонок (о которых мы поговорим несколько позже). Метод гарантирует, что к каждому объекту VCL одновременно получит доступ только один поток. В качестве аргумента процедуры передается имя вызываемого метода.

Пример простого многопоточного приложения

Для приобретения навыков напишем приложение, в составе которого будет функционировать три симметричных потока.

1. Создадим новый проект. Переименуем главную форму проекта в frmMain.dfm. Сохраним проект. Модуль unit1, соответствующий главной форме, назовем main.pas. Имя проекта выберите на свое усмотрение.
2. Поместите на главную форму три компонента TProgressBar, присвоив всем трем вертикальную ориентацию Orientation= pbVertical. Свойству Max, описывающему максимальное значение шкалы, назначим значение 1000. Под каждым компонентом TProgressBar поместите элемент управления TCheckBox (флажок). В правой части формы разместите три элемента TTrackBar (ползунок). У соответствующих компонентов TCheckBox и TTrackBar поменяйте значение свойства tag:

```
CheckBox1 и TrackBar1 - tag=1;
CheckBox2 и TrackBar2 - tag=2;
CheckBox3 и TrackBar3 - tag=3.
```

Последние три компонента проекта – метки TLabel – разместите над шкалами TProgressBar (рис. 20.3).

3. При помощи элементов управления TTrackBar мы будем изменять приоритеты соответствующих потоков. Одновременно выберите все компоненты TTrackBar и их свойству Max присвойте значение 3. Значение 0 будет соответствовать минимальному приоритету tpIdle, значение 3 – нормальному приоритету tpNormal.

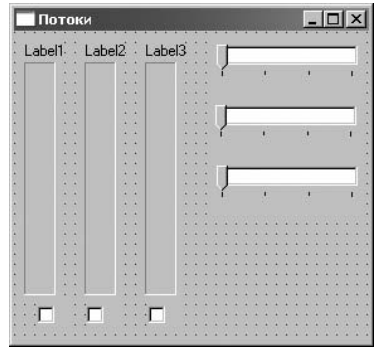


Рис. 20.3. Главная форма проекта

4. С настройкой элементов управления покончено. Займемся потоками. Для этого выберите пункт меню File → New → Other. На вкладке New выберите пиктограмму ThreadObject и нажмите OK. Назовите класс создаваемого потока TMyThread (рис. 20.4). Вновь созданный модуль с шаблоном кода потока сохраните под именем ThreadUnit.pas.

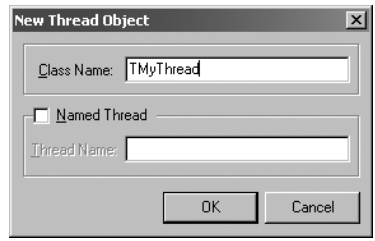


Рис. 20.4. Создание модуля с потоком

5. В соответствии с приведенным ниже листингом внесите изменения в модуль ThreadUnit:

```
unit ThreadUnit;

interface

uses Classes, Windows, SysUtils, StdCtrls,
ComCtrls;

type
  TMyThread = class(TThread)
  private
    { Private declarations }
    procedure UpdateControls;
  protected
    procedure Execute; override;
  public
    ProgressBar : TProgressBar; //поле потока для подключения к TProgressBar
    Lbl : TLabel;        //поле потока для подключения к информационной метке
  end;
implementation

procedure TMyThread.UpdateControls;
begin
ProgressBar.Position:=ProgressBar.Position+1;
if ProgressBar.Position>=ProgressBar.Max then ProgressBar.Position:=0;
Lbl.Caption:=IntToStr(ReturnValue);
end;

{ TMyThread }

procedure TMyThread.Execute;
var I:INTEGER;
begin
```

```
    while Terminated=FALSE do
    begin
      I:=0;
      while I<2000000 do INC(I);      // единственная задача цикла -
                                      // занять процессорное время
      ReturnValue:=ReturnValue+1;
      Synchronize(UpdateControls);    // синхронизируемся с методами VCL
    end;
  end;

  end.
```

В секции public потока опубликованы два поля ProgressBar : TProgressBar и lbl : TLabel, предназначенные для организации взаимодействия потока с размещенными на главной форме шкалой и меткой. Основной метод Execute() выполняется до тех пор, пока он не будет разрушен вызовом функции Terminate. Внутри метода реализован цикл while I<2000000 do INC(I), предназначенный только для «пожирания» времени процессора. В заключение из тела метода Execute() вызывается функция Synchronize(UpdateControls), организующая обновление данных в принадлежащих потоку шкале и метке.

6. Оставим модуль потоков в покое. Вернемся к главной форме проекта и опишем ряд методов. В обработчике события OnCreate() формы поместите строки кода, создающие и запускающие все три потока.

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
 Thread1:=TMyThread.Create(True);
 Thread1.Priority:=tpIdle;
 Thread1.ProgressBar:=ProgressBar1;
 Thread1.Lbl:=Label1;

 Thread2:=TMyThread.Create(True);
 Thread2.Priority:=tpIdle;
 Thread2.ProgressBar:=ProgressBar2;
 Thread2.Lbl:=Label2;

 Thread3:=TMyThread.Create(True);
 Thread3.Priority:=tpIdle;
 Thread3.ProgressBar:=ProgressBar3;
 Thread3.Lbl:=Label3;

 Thread1.Resume; Thread2.Resume; Thread3.Resume; //запускаем потоки
end;
```

В момент создания потоков мы связываем их с элементами управления TProgressBar и TLabel. Обращаю внимание, что все три экземпляра потоков создаются приостановленными и стартуют практически вместе после того, как к ним будут применены все настройки.

7. Чтобы не забыть, сразу опишем событие закрытия формы, в которой последовательно разрушим все три потока:

```
procedure TfrmMain.FormClose(Sender: TObject; var Action: TCloseAction);
begin
```

```
Thread1.Terminate;
Thread2.Terminate;
Thread3.Terminate;
end;
```

8. Настал звездный час элемента управления TTrackBar. Задача компонента – устанавливать приоритет для соответствующего потока. Выберите любой из этих компонентов и следующим образом опишите его обработчик события OnChange():

```
procedure TfrmMain.TrackBar1Change(Sender : TObject);
var Priority : TThreadPriority;
begin
{преобразуем позицию ползунка в значение приоритета}
    case (Sender as TTrackBar).Position of
    0 : Priority:=tpIdle;    //фоновый
    1 : Priority:=tpLowest;  //выше фонового
    2 : Priority:=tpLower;   //низкий
    3 : Priority:=tpNormal;  //нормальный
    end;

{по tag элемента управления выясняем, какому из потоков
 требуется изменить приоритет}
case (Sender as TTrackBar).Tag of
1 : Thread1.Priority:=Priority;
2 : Thread2.Priority:=Priority;
3 : Thread3.Priority:=Priority;
end;
end;
```

Назначьте описанный выше обработчик события OnChange() общим для всех компонентов TTrackBar.

9. Опишите обработчик события OnClick() любого из компонентов TCheckBox в соответствии с указанным ниже кодом и сделайте его общим для всех компонентов-флажков. Задача этой процедуры – приостановить/продолжить выполнение потока.

```
procedure TfrmMain.
CheckBox1Click(Sender: TObject);
var Suspended:boolean;
begin
Suspended:=(Sender as
TCheckBox).Checked;
    case (Sender as TCheckBox).Tag of
    1 : Thread1.Suspended:=Suspended;
    2 : Thread2.Suspended:=Suspended;
    3 : Thread3.Suspended:=Suspended;
    end;
end;
```

Рис. 20.5. Приложение в действии

- Компоненты обеспечения централизованного доступа к данным – TADOConnection и TRDSConnection. Эти компоненты обычно используются при разработке сложных приложений, базирующихся на архитектуре клиент–сервер.
- Компоненты обслуживания данных в стиле Microsoft ADO – командный компонент TADOCommand и набор данных TADODataSet.
- Компоненты, работающие с данными в классическом для программистов Borland стиле. Это компонент-таблица TADOTable, запрос TADOQuery и хранимая процедура TADOStoredProc.

Компонент TRDSConnection *с технологией ADO связан косвенно. Элемент управления поддерживает функциональные возможности службы удаленных данных (RDS, Remote Data Services). Эта технология обеспечивает удаленный доступ к источникам данных ADO и в первую очередь предназначена для построения многоуровневых систем.*

Интерфейс соединения – _Connection

Реализованный в модели MDAC объект ADO Connection предназначен для организации соединения с набором данных. Объект позволяет строить как простейшие соединения, так и работать с базами данных на принципах архитектуры клиент–сервер. Интерфейсная часть объекта описана в модуле ADOInt.

В компонентах VCL интерфейс и диспинтерфейс соединения доступны благодаря свойству:

```
property ConnectionObject: _Connection;
```

С интерфейсом _Connection способны взаимодействовать все элементы линейки ADO (за исключением TRDSConnection). Более подробно об организации доступа к данным с помощью этого интерфейса мы поговорим при изучении компонента TADOConnection.

Интерфейс команды – _Command

Общение с набором данных ADO осуществляется посредством передачи команд (SQL-инструкций) в адрес провайдера. Результат выполнения команды возвращается в виде множества строк – объекта RecordSet. Интерфейс и диспинтерфейс команды объявлены в модуле ADOInt библиотеки визуальных компонентов Delphi.

Интерфейс командного объекта инкапсулирован практически во всех компонентах линейки ADO (за исключением TRDSConnection*).*

В библиотеке VCL доступ к интерфейсу командного объекта представлен в виде свойства:

```
property CommandObject: _Command;
```

Текст отправляемой команды описывается в свойстве:

```
property CommandText: WideString;
```

Как мы уже говорили, это не что иное, как инструкция на языке SQL. Если команда содержит какие-то параметры, например аргументы хранимой процедуры, то определите параметры в свойстве:

```
property Parameters: TParameters;
```

Команды систематизированы по типам:

```
property CommandType: TCommandType;   //по умолчанию cmdText
type TCommandType = (cmdUnknown, cmdText, cmdTable, cmdStoredProc, cmdFile,
                     cmdTableDirect);
```

Определяя тип команды (табл. 39.1), программист явным образом указывает командному объекту ADO, какого рода инструкцию он планирует описать в его свойстве CommandText.

Таблица 39.1. Перечень типов команд ADO

Тип команды	Описание
cmdUnknown	Неизвестный тип команды.
cmdText	В свойстве CommandText определено текстовое описание инструкции SQL.
cmdTable	В свойстве CommandText указано имя таблицы.
cmdStoredProc	В свойстве CommandText указано имя хранимой процедуры.
cmdFile	В свойстве CommandText указано имя файла, в который сохранен набор записей.
cmdTableDirect	В свойстве CommandText указано имя таблицы; планируется получить все столбцы.

Например, если тип команды определен как cmdStoredProc, то вместо явного текстового описания SQL-инструкции в свойстве CommandText достаточно указать имя хранимой процедуры, которую мы намерены выполнить:

```
with ADOCommand1 do
begin
  CommandType := cmdStoredProc;
  CommandText := 'stp_ClearTempTable';
  Execute;
end;
```

Самый главный метод Execute() командного объекта ADO отправляет хранящиеся в CommandText инструкции SQL на выполнение. Различают три перегружаемые вариации этого метода:

```
function Execute: _Recordset; overload;
function Execute(const Parameters: OleVariant): _Recordset; overload;
function Execute(var RecordsAffected: Integer;
                 const Parameters: OleVariant): _RecordSet; overload;
```

В простейшем случае функция вообще не требует каких-либо аргументов. Первый параметр RecordsAffected проинформирует нас о том, сколько записей было охвачено отправленной командой. Аргумент Parameters – коллек-

ция передаваемых параметров. Если инструкция SQL предназначена для формирования какого-то набора записей, то метод создаст набор в виде множества записей _Recordset.

```
with ADOCommand1 do
begin
CommandType := cmdStoredProc;
CommandText := 'stp_MyStoredProc';
{выполнение команды с возвратом результата в объект RecordSet компонента ADODataSet1}
ADODataSet1.Recordset := Execute;
end;
```

Еще один пример, но на этот раз в адрес провайдера направляется текстовая SQL-инструкция. В этом случае вместо метода Execute() надо использовать метод Open():

```
with ADODataSet1 do
begin
  CommandText := 'SELECT PhoneNum, FName FROM Phones WHERE Street = :StreetParam';
  CommandType := cmdText;
  Parameters.ParamByName('StreetParam').Value := 'Невский проспект';
  Open;
end;
```

Время тайм-аута, по истечении которого компонент станет полагать, что выполнение команды закончилось неудачей, определяется целым числом в свойстве:

```
property CommandTimeout: Integer;
```

По умолчанию тайм-аут соответствует 30 секундам. Если за отведенное время отправленная к СУБД команда не будет выполнена, она автоматически отменяется.

Интерфейс множества записей – _Recordset

В рамках технологии ADO программистами корпорации Microsoft реализован специализированный объект, называемый множеством записей Recordset. Этот объект обеспечит доступ к набору записей таблицы или записей, полученных в результате выполнения какой-то команды. Физически Recordset представляет собой совокупность записей и колонок, создаваемых в момент обращения к данным или при выполнении SQL-инструкции. Что касается Delphi, то в составе изучаемого нами класса TCustomADODataSet множество записей Recordset инкапсулировано в свойстве:

```
property Recordset: _Recordset;
```

Свойство представляет собой указатель на интерфейс _Recordset и позволяет программисту манипулировать его свойствами и методами напрямую. Текущее состояние объекта Recordset описывается свойством:

```
property RecordsetState: TObjectStates;
type TObjectState = (stClosed, stOpen, stConnecting, stExecuting, stFetching);
TObjectStates = set of TObjectState;
```

Состояние stClosed удостоверяет пассивность объекта. Пока потомок класса TCustomADODataSet находится в процессе соединения, свойство возвратит значение stConnecting. Значение stOpen свидетельствует о том, что набор данных открыт и множество записей доступно. Значение stExecuting указывает, что этот элемент управления выполняет команду. Значение stFetching информирует нас о том, что в данный момент производится выборка строк из таблиц.

Для того чтобы набор данных мог выяснить, каким набором возможностей обладает обслуживаемый им объект Recordset, в его состав включен метод:

```
function Supports(CursorOptions: TCursorOptions): Boolean;
```

Возможные значения единственного параметра метода представлены в табл. 39.2. Если указанная опция поддерживается, метод возвратит значение true.

Таблица 39.2. Возможные значения класса TCursorOptions

Значение	Описание возможностей множества записей Recordset
coHoldRecords	Большинство записей может быть восстановлено без подтверждения отложенных обновлений.
coMovePrevious	Курсор может быть перемещен назад без использования закладки.
coAddNew	Поддерживается вставка новых записей.
coDelete	Поддерживается удаление записей.
coUpdate	Поддерживается модификация записей.
coBookmark	Закладка (свойство Bookmark) объекта Recordset может быть использована для перемещения курсора или для доступа к отмеченной записи.
coApproxPosition	Множество записей поддерживает свойство RecNo, что позволяет идентифицировать номер записи.
coUpdateBatch	Поддерживается пакетное обновление данных (допустим вызов методов UpdateBatch и CancelBatch).
coResync	Данные курсора могут обновляться асинхронным способом.
coNotify	Объект Recordset поддерживает уведомления и может генерировать события.
coFind	Для поиска данных допустимо применение метода Locate().
coSeek	Для поиска записи может использоваться метод Seek().
coIndex	Для задания текущего индекса может использоваться свойство IndexName.

Строка соединения ADO

Напомню, что при работе с ядром доступа к данным BDE для организации соединения с БД требуется указать путь к таблицам. К одному и тому же результату приходят двумя способами: посредством явного обращения к таблицам через файловый путь либо путем создания соответствующего псевдонима БД в утилите администрирования BDE. Нечто похожее потребуется про-

делать и для получения доступа к данным ADO. Только теперь параметры соединения описываются в текстовой **соединительной строке** либо в хранящем настроечные данные **UDL-файле**, называемом файлом связи с данными.

Соединительная строка представляет собой последовательность аргументов и значений, разделяемых точкой с запятой:

```
Параметр1=значение1;Параметр2=значение2;
```

Например, так выглядит строка доступа к базе данных DEMODB, расположенной на Microsoft SQL Server с именем MSSQL_HOST:

```
Provider=SQLOLEDB.1;Persist Security Info=False;User ID=SA;Initial
Catalog=DEMODB;Data Source=MSSQL_HOST
```

В качестве символа присваивания применяется знак равенства. Перечень наиболее часто встречаемых в соединительной строке аргументов представлен в табл. 39.3.

Таблица 39.3. Аргументы соединительной строки

Параметр	Описание
Provider	Имя поставщика интерфейса OLE DB. Если этот аргумент не определен, то по умолчанию будет использован провайдер Microsoft OLE DB для ODBC – «MSDASQL».
File name	Название файла, содержащего информацию о соединении.
User ID	Регистрационное имя пользователя по умолчанию.
Data Source	Имя источника данных, например SQL-сервера.
Initial Catalog	Каталог инициализации – имя базы данных.
Remote Provider	Имя провайдера, применяемого на клиентской стороне соединения. Этот параметр необходим только при организации работы со службами удаленных данных RDS.
Remote Server	Имя удаленного сервера. Этот параметр необходим только при работе с RDS.
URL	Абсолютный URL-адрес.

Чтобы сформировать «вручную» содержащий описание соединения UDL-файл, создайте пустой текстовый файл и измените его расширение с .txt *на* .udl*. Запустите файл на выполнение. Если на компьютере установлена служба MDAC, появится многостраничное диалоговое окно, позволяющее построить соединительную строку.*

Соединение с источником данных ADO – компонент TADOConnection

Элемент управления TADOConnection выполняет функции, схожие с задачами компонента Tdatabase. Но, в отличие от своего коллеги (получающего доступ к данным посредством обращения к технологии BDE), он нацелен на поддержание соединения с источником данных ADO. Для этого компонент инкапсулирует интерфейс _Connection.

Как правило, в приложении применяется не более одного компонента TADOConnection*, к которому в свою очередь подключаются все остальные элементы доступа к данным ADO.*

Хотя основной задачей компонента TADOConnection считается установка соединения с базой данных ADO, но помимо этого он умеет:

- регистрировать пользователя в СУБД;
- управлять транзакциями;
- отправлять в адрес СУБД различные команды (инструкции SQL);
- взаимодействовать с подчиненными компонентами – наборами данных ADO;
- получать от БД информацию об объектах БД и особенностях соединения.

Компоненты TADODataSet*,* TADOTable*,* TADOQuery*,* TADOStoredProc *и* TADOCommand *способны поддерживать соединение с базой данных как при помощи* TADOConnection*, так и самостоятельно.*

Установка соединения

Для описания параметров подключения к БД текст соединительной строки направляется в свойство:

```
property ConnectionString: WideString;
```

Если в строке отсутствует информация о поставщике услуг OLE DB, то его необходимо определить в свойстве:

```
property Provider: WideString;
```

По умолчанию в качестве провайдера назначается MSDASQL. Процесс соединения с описанной в соединительной строке базой данных инициируется посредством обращения к методу Open(). В простейшей ситуации процедура даже не требует параметров.

```
procedure Open; overload;
```

В перегружаемой версии процедуры Open() предусмотрена возможность передачи в адрес сервера имени и пароля пользователя. Но это только для случая, когда эта информация не включена в соединительную строку.

```
procedure Open(const UserID: WideString; const Password: WideString); overload;
```

Если вы полагаете, что при подключении к БД пользователь обязан пройти регистрацию и своими руками ввести имя и пароль, то целесообразно установить в True свойство:

```
property LoginPrompt: Boolean;
```

В результате перед соединением с базой данных вызывается стандартное диалоговое окно регистрации. Для проверки факта подключения компонента TADOConnection к БД контролируют состояние свойства:

```
property Connected: Boolean;
```

Значение true свидетельствует о соединении с базой данных, false – соединение отсутствует. Свойство доступно не только для чтения, поэтому его вполне можно применять для подключения или отключения от БД вместо методов Open() и Close().

Для поддержания постоянного соединения даже в случаях, когда приложение не обращается ни к одному из наборов данных, переведите в True свойство:

```
property KeepConnection: Boolean;
```

В противном случае клиентское приложение отключается от сервера сразу после закрытия последнего активного набора данных, и для повторного соединения потребуется вновь проходить утомительный этап подключения.

Различают два типа соединения: синхронное и асинхронное. За определение, каким образом наше приложение намерено работать с БД, отвечает свойство:

```
property ConnectOptions: TConnectOption;
type TConnectOption = (coConnectUnspecified, coAsyncConnect);
```

По умолчанию оно установлено в состояние coConnectUnspecified. В этом случае по возможности обеспечивается синхронное соединение. Принудительная организация асинхронного соединения (значение coAsyncConnect) рекомендуется только тогда, когда производительность СУБД оставляет желать лучшего и мы предполагаем, что у приложения могут возникнуть проблемы с получением данных.

Отнюдь не всегда факт успешного соединения свидетельствует о том, что пользователь приобрел абсолютные права на работу с данными. В реальности перечень возможностей пользователя может быть ограничен. Для того чтобы уточнить ограничения, следует обратиться к информационному свойству:

```
property Mode: TConnectMode;
```

Восемь возможных состояний этого свойства представлены в табл. 39.4.

Таблица 39.4. Описание TConnectMode

Константа	Описание
cmUnknown	Тип доступа явно не указан. Предоставляется установленный по умолчанию тип доступа.
cmRead	Данные доступны только для чтения.
cmWrite	Данные доступны только для записи.
cmReadWrite	Права на чтение и запись данных.
cmShareDenyRead	Монопольный доступ «только для чтения». Другие пользователи не смогут читать данные.
cmShareDenyWrite	Монопольные права на запись. Запрет на совместную запись.
cmShareExclusive	Данные забраны в эксклюзивное пользование и недоступны для других.
cmShareDenyNone	Монопольный доступ к источнику данных. Другие пользователи не смогут открыть источник.

Время ожидания ответа сервера БД после вызова метода Open() называется временем тайм-аута. Оно определяется свойством:

```
property ConnectionTimeout: Integer;
```

и по умолчанию составляет 15 секунд. Если за установленный промежуток времени компонент TADOConnection не получит ответа от СУБД, попытка соединения считается неудачной. В этом случае (по истечении времени тайм-аута) компонент TADOConnection самостоятельно обратится к методу Cancel, который прекращает попытки вызова сервера.

```
procedure Cancel;
```

С процессом установления соединения связана строгая последовательность событий (табл. 39.5), позволяющая программисту практически на всех этапах помогать или мешать (кому что нравится) этому процессу. Из шести событий три унаследованы от абстрактного класса TCustomConnection. Они уже упоминались в главе 38 в связи с компонентом TDatabase. Оставшиеся события OnWillConnect, OnConnectComplete и OnInfoMessage – вклад класса TADOConnection.

Таблица 39.5. Последовательность событий при соединении TADOConnection с набором данных

	Событие	Описание
1	`property BeforeConnect: TNotifyEvent;`	Вызывается перед началом соединения.
2	`property OnWillConnect: TWillConnectEvent;`	Генерируется после отправки запроса на соединение.
3	`property OnLogin: TLoginEvent;`	Регистрация пользователя.
4	`property OnConnectComplete : TConnectErrorEvent;`	Процесс соединения завершается.
5	`property OnInfoMessage: TInfoMessageEvent;`	Вызывается после успешного соединения в тот момент, когда провайдер возвращает дополнительную информацию о соединении.
6	`property AfterConnect: TNotifyEvent;`	Вызывается по окончании всего процесса соединения.

В перечне представленных в табл. 39.5 событий наибольший интерес представляет OnWillConnect(). Событие пригодится нам в тот момент, когда провайдер уже ответил согласием на соединение, но физически соединение еще не установлено.

```
TWillConnectEvent = procedure(Connection: TADOConnection;
                              var ConnectionString, UserID, Password: WideString;
                              var ConnectOptions: TConnectOption;
                              var EventStatus: TEventStatus) of object;
```

Здесь Connection – ссылка на компонент; ConnectionString, UserID, Password – соответственно строка соединения, имя пользователя и пароль; ConnectOptions – опции соединения (синхронное – coConnectUnspecified, асинхронное –

coAsyncConnect). Параметр EventStatus позволяет программисту определить статус операции. Если по какой-то причине в момент возникновения события OnWillConnect() вы вдруг захотите отказаться от соединения, присвойте этому параметру значение esCancel (на месте провайдера я бы таких клиентов на милю не подпускал к компьютеру). К счастью, кроме esCancel тип данных TEventStatus поддерживает и другие значения:

```
type TEventStatus = (esOK, esErrorsOccured, esCantDeny, esCancel, esUnwantedEvent);
```

Самое безобидное из них – esOK; оно свидетельствует о том, что все хорошо и мы готовы к подключению. Надо отметить, что параметр EventStatus еще неоднократно нам встретится в других событиях, поэтому познакомьтесь с описанием статуса события, предложенным в табл. 39.6.

Таблица 39.6. Описание статуса события TEventStatus

Значение	Описание
esOK	Операция выполняется без проблем.
esErrorsOccured	Во время выполнения операции произошла ошибка или ряд ошибок.
esCantDeny	Запрет на отмену текущей операции.
esCancel	Ожидающая обработки операция подключения была отменена прежде, чем было установлено соединение.
esUnwantedEvent	Прекратить уведомления о последующих событиях на время выполнения текущей операции.

После успешной регистрации пользователя генерируется событие OnConnectComplete().

```
TConnectErrorEvent = procedure(Connection: TADOConnection; Error: Error;
                               var EventStatus: TEventStatus) of object;
```

Здесь Connection – ссылка на компонент, осуществивший соединение; Error – зарегистрированный в ADO интерфейс, описывающий ошибку. По умолчанию во время обработки события параметр EventStatus, содержащий информацию о статусе события, принимает значение esOK, что свидетельствует о нормальном выполнении операции. Манипулируя другими значениями (см. табл. 39.6), программист приобретает некоторое влияние на процесс обработки события.

Если в приложении планируется обрабатывать события, в качестве параметра которых применяется интерфейс _Error *(например,* OnConnectComplete *и* OnInfoMessage*), то проверьте, чтобы к списку* uses *проекта был подключен модуль* ADOInt*, содержащий описания интерфейсов ADO.*

Третье событие, описанное в классе TADOConnection, позволяет программисту отреагировать на дополнительную информацию, возвращаемую провайдером клиентскому приложению после успешного соединения. Все параметры обработчика события вам уже знакомы:

```
TInfoMessageEvent = procedure(Connection: TADOConnection; const Error: Error;
                              var EventStatus: TEventStatus) of object;
```

Прямой доступ к интерфейсу объекта соединения предоставляется через свойство:

```
property ConnectionObject: _Connection;
```

Разрыв соединения

Для разрыва соединения воспользуйтесь методом `Close()` или установите в `false` свойство `Connection`. При отключении от набора данных последовательно генерируются три события, представленные в табл. 39.7.

Таблица 39.7. Последовательность событий при разрыве соединения TADOConnection с набором данных

	Событие	Описание
1	`property BeforeDisconnect: TNotifyEvent;`	Вызывается перед разрывом соединения.
2	`property OnDisconnect: TDisconnectEvent;`	Осуществляется разрыв.
3	`property AfterDisconnect: TNotifyEvent;`	Генерируется после разрыва.

Из трех событий, обслуживающих отключение от БД, лишь одно реализовано классом `TADOConnection`. Это событие `OnDisconnect()`, генерируемое сразу после отключения от набора данных.

```
TDisconnectEvent = procedure(Connection: TADOConnection;
                             var EventStatus: TEventStatus) of object;
```

Остальные события нам уже знакомы по классу `TCustomConnection`.

Информирование о соединении

Для выяснения номера установленной на компьютере версии технологии ADO обращаемся к свойству:

```
property Version: WideString; //только для чтения
```

При этом вовсе не обязательно, чтобы элемент управления `TADOConnection` состоял в соединении с какой-либо базой данных. Благодаря свойству `Version` программист может проверить наличие пакета MDAC на компьютере пользователя.

Текущее состояние соединения описывается свойством `State`. Перечень возможных состояний, описываемых типом данных `TobjectStates`, уже упоминался в этой главе при обсуждении интерфейса `_Recordset`.

```
property State: TObjectStates; //только для чтения
```

Наиболее информативен метод `OpenSchema()` (от англ. открытие схемы). Напомню, что в терминологии реляционной модели данных под схемой понимают взаимосвязанную совокупность объектов, составляющих базу данных. В перечень объектов схемы входят таблицы, домены, индексы, хранимые процедуры и т. п. Заголовок метода выглядит следующим образом:

```
procedure OpenSchema(const Schema: TSchemaInfo; const Restrictions: OleVariant;
                     const SchemaID: OleVariant; DataSet: TADODataSet);
```

Первый параметр метода Schema: TSchemaInfo определяет, какую именно информацию мы желаем получить. Например, передав в параметр константу siIndexes, мы получим информацию об индексах БД, siTables – о таблицах, siViews – о представлениях. Второй параметр Restrictions представляет собой массив ограничений на запрашиваемую информацию. Третий параметр SchemaID применяется только в ситуации, когда глобальный уникальный идентификатор (GUID) провайдера данных не определен в стандартной спецификации OLE DB. Как правило, такая необходимость возникает в случаях, когда аргумент Schema установлен в siProviderSpecific. Последний параметр DataSet является ссылкой на элемент управления, в свойство Recordset которого будут направлены результаты выполнения процедуры. Полученные данные программист может обрабатывать по своему усмотрению, например можно просто отобразить их в сетке TDBGrid.

Применим знания на практике. Разместите на форме нового проекта компоненты: соединение ADO – ADOConnection1; набор данных ADO – ADODataSet1; источник данных – DataSource1; сетку – DBGrid1.

Перейдите в Инспектор объектов. С помощью свойства Connection элемента управления ADODataSet1 свяжите компонент с ADOConnection1. Затем выберите источник данных DataSource1 и при помощи свойства DataSet соедините его с ADODataSet1. В заключение подключите сетку DBGrid1 к источнику данных DataSource1. На этом работа с Инспектором объектов завершена.

Переходим к программированию. На свое усмотрение создайте строку соединения с интересующей вас базой данных. В примере я связался с базой данных Master СУБД Microsoft SQL Server, размещенной на компьютере SQLSERVER. Затем опишите событие создания формы, как предложено в следующем листинге:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
ADOConnection1.ConnectionString:='Provider=SQLOLEDB.1;Persist Security
Info=False;User ID=sa;Initial Catalog=Master; Data Source=SQLSERVER';
ADOConnection1.Open;
ADOConnection1.OpenSchema(siTables, EmptyParam, EmptyParam,ADODataSet1);
end;
```

Если все сделано безошибочно, то в результате запуска проекта мы получим данные обо всех таблицах БД.

На основе процедуры OpenSchema() построен ряд справочных методов, предоставляющих данные о таблицах, полях таблиц и хранимых процедурах базы данных:

```
procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);
procedure GetFieldNames(const TableName: string; List: TStrings);
procedure GetProcedureNames(List: TStrings);
```

Результаты выполнения процедур передаются в список List.

Отправка команд

В терминологии ADO под **командой** понимается инструкция на языке SQL. Например, в качестве команды можно рассматривать операции удаления, вставки и редактирования строк набора данных.

Компонент TADOConnection может отправить SQL-инструкции провайдеру двумя способами:

1. Воспользоваться услугами любого подключенного к нему компонента линейки ADO Delphi, инкапсулирующего интерфейс _Command.
2. Передать команду, вызвав метод Execute().

Для хранения и передачи провайдеру наиболее часто вызываемых команд реализован специальный элемент управления TADOCommand.

В целях обеспечения эффективного взаимодействия компонента TADOConnection и подключенных к нему командных компонентов в классе TADOConnection объявлен ряд свойств. Количество подключенных компонентов, специализирующихся на отправке команд, хранится в свойстве:

```
property CommandCount: Integer; // только для чтения
```

Список связанных с TADOConnection активных командных компонентов находится в массиве:

```
property Commands[Index: Integer]: TADOCommand;
```

Например:

```
for i := 0 to (ADOConnection1.CommandCount - 1) do
ADOConnection1.Commands[i].Execute;
```

Если требуется передать в БД какую-то единичную инструкцию SQL, то вместо использования командных компонентов целесообразнее сформировать команду динамически и отправить ее при помощи одной из вариаций перегружаемого метода Execute():

```
function Execute(const CommandText: WideString;   //инструкция SQL
                 const CommandType: TCommandType = cmdText;  //тип команды
                 const ExecuteOptions: TExecuteOptions = []  //опции выполнения
                 ): _RecordSet; overload;          //возвращаем интерфейс Recordset

procedure Execute(const CommandText: WideString;  //инструкция SQL
                  var RecordsAffected: Integer;   //обработано строк
                  ExecuteOptions: TExecuteOptions = [eoExecuteNoRecords]
                  ); overload;
```

В обоих случаях единственный обязательный параметр – это содержащая инструкцию SQL константа CommandText.

```
var SQLText : string;
begin
  SQLText := ' CREATE TABLE Peoples
      (FNameChar(30),
       LNameChar(30),
```

```
      BornDate)';
  SQLConnection1.Execute(SQLText, nil, nil);
end;
```

С передачей команды связаны два события. Перед тем как команда будет выполнена, генерируется событие:

```
property OnWillExecute: TWillExecuteEvent;
TWillExecuteEvent = procedure(Connection: TADOConnection;
                    var CommandText: WideString;  //текст команды
                    var CursorType: TCursorType;  //тип курсора
                    var LockType: TADOLockType;  //особенности блокировки транзакции
                    var CommandType: TCommandType;  //тип команды
                    var ExecuteOptions: TExecuteOptions; //опции выполнения
                    var EventStatus: TEventStatus;  //индикатор выполнения операции
                    const Command: _Command;        //интерфейс командного объекта
                    const Recordset: _Recordset     //интерфейс множества записей
                    ) of object;
```

После успешного запуска команды вызывается событие:

```
property OnExecuteComplete: TExecuteCompleteEvent;
TExecuteCompleteEvent = procedure(Connection: TADOConnection;
                     RecordsAffected: Integer;
                     const Error: Error;              //интерфейс ошибки
                     var EventStatus: TEventStatus;  //индикатор выполнения операции
                     const Command: _Command;        //интерфейс командного объекта
                     const Recordset: _Recordset     //интерфейс множества записей
                     ) of object;
```

По параметру `RecordsAffected` программист может судить о том, сколько строк набора данных было задействовано в результате выполнения команды. Наличие ошибок при выполнении команды отразится на параметре `Error`. При необходимости переменная `EventStatus` позволит обработать ошибку. Если для отправки инструкции использовался командный компонент ADO, то ссылку на него мы обнаружим в свойстве `Command`.

Ряд параметров событий класса `TADOConnection` *предназначен для взаимодействия с реальными объектами ADO, а не с инкапсулирующими их компонентами библиотеки VCL. К таковым параметрам относятся:* `_Error` *– специализированный интерфейс ошибки,* `_Command` *– интерфейс командного объекта,* `_RecordSet` *– интерфейс множества записей.*

Управление транзакциями

Для старта новой транзакции предназначена функция:

```
function BeginTrans: Integer;
```

Метод возвращает целочисленное значение, соответствующее уровню вложения транзакции. При успешном запуске новой транзакции в `True` устанавливается свойство:

```
property InTransaction: Boolean;
```

Значение True свидетельствует о том, что TADOConnection инициировал транзакцию и она находится в стадии выполнения. С окончанием транзакции свойство вернется в состояние False. Сразу после успешного запуска новой транзакции происходит событие:

```
property OnBeginTransComplete: TBeginTransCompleteEvent;
TBeginTransCompleteEvent = procedure(Connection: TADOConnection;
                                     TransactionLevel: Integer; const Error: Error;
                                     var EventStatus: TEventStatus) of object;
```

Параметр EventStatus информирует нас, насколько успешно прошел процесс запуска транзакции (см. табл. 39.6). Если операция выполнилась корректно, в параметре окажется значение esOK.

Для корректного завершения транзакции программист вызывает процедуру:

```
procedure CommitTrans;
```

Сразу после сохранения транзакции возникает событие:

```
property OnCommitTransComplete: TConnectErrorEvent;
TConnectErrorEvent = procedure(Connection: TADOConnection; Error: Error;
                               var EventStatus: TEventStatus) of object;
```

Если в процесс выполнения транзакции вкралась ошибка, для ее отмены воспользуйтесь методом:

```
procedure RollbackTrans;
```

Откат транзакции сопровождается событием:

```
property OnRollbackTransComplete: TConnectErrorEvent;
```

Если разрабатываемая база данных станет эксплуатироваться в многопользовательском режиме, то особое внимание стоит уделить вопросам обеспечения параллельной обработки данных. Накал борьбы клиентских приложений за критический ресурс в БД можно снизить, установив соответствующий уровень изоляции транзакций:

```
property IsolationLevel: TIsolationLevel;
```

Уровень разграничения транзакций определяет возможность одновременного совместного доступа пользователей к одним и тем же данным. В табл. 39.8 приведены допустимые значения блокировок транзакций. Блокировки упорядочены сверху вниз по степени возрастания пессимистических настроений. Другими словами, каждая последующая строка все более и более затрудняет параллельную работу нескольких пользователей с одними и теми же данными, но зато снижает вероятность появления ошибок.

Таблица 39.8. Возможные уровни блокировки транзакции ADO-соединения – TIsolationLevel

Уровень изоляции	Описание
ilUnspecified	Сервер может быть настроен для работы в любом режиме изоляции транзакций, и приложение не предъявляет к СУБД каких-либо особых требований.

Таблица 39.8 (продолжение)

Уровень изоляции	Описание
ilChaos	Изменения, сделанные транзакцией с более высоким уровнем изоляции, не могут быть переписаны в текущем соединении.
ilReadUncommitted или ilBrowse	**Грязное чтение**. Допустимо чтение данных незавершенных транзакций.
ilCursorStability или ilReadCommitted	**Чистое чтение**. Данные доступны только после завершения транзакции.
ilRepeatableRead	**Повторяющееся чтение**. Изменения, произведенные другими транзакциями к предварительно прочитанным данным, не видны, пока не будет осуществлено повторное чтение данных.
ilSerializable или ilIsolated	**Полная изоляция**. Максимальная степень изоляции транзакций.

Процесс инициализации очередной транзакции может быть автоматизирован при помощи свойства:

```
property Attributes: TXactAttributes; //по умолчанию автоматизация отключена
```

В свойство может устанавливаться любая комбинация флагов из множества TxactAttributes:

```
TXactAttributes = set of TXactAttribute;
type TXactAttribute = (xaCommitRetaining, xaAbortRetaining);
```

Наличие флагов оказывает влияние на процесс инициализации очередной транзакции после сохранения или отката текущей транзакции. Флаг xaCommitRetaining *указывает, что сохранение изменений, сделанных корректно завершенной транзакцией, автоматически вызывает следующую транзакцию. Наличие флага* xaAbortRetaining *служит признаком того, что после отката сбойной транзакции предпринимается повторная попытка ее выполнения.*

При назначении атрибутов соединения необходимо ознакомиться с документацией провайдера ADO, иначе можно установить недопустимые флаги.

Библиотека курсоров

Полученный в результате выполнения запроса набор строк может обрабатываться одной из двух библиотек курсоров: библиотекой сервера или библиотекой клиента. Предпочтительный для нас режим обработки результирующего набора определяется свойством:

```
property CursorLocation: TCursorLocation; //по умолчанию clUseClient
type TCursorLocation = (clUseServer, clUseClient);
```

При подключении клиентской библиотеки (CursorLocation=clUseClient) запрошенные записи направляются в клиентское приложение и обслуживаются имеющимися в ее распоряжении драйверами. Как правило, возможности клиентского курсора значительно более гибкие, чем у сервера, и поддерживают расширенные возможности фильтрации и сортировки данных. Но все

выгоды клиентского курсора сводятся к нулю, если мы работаем с большими блоками данных на слабой клиентской машине. В этом случае отправка необработанной информации клиенту нерациональна, т. к. пересылка и обработка данных займут значительное количество времени и ресурсов. Поэтому при обработке больших массивов записей целесообразнее выбирать серверный курсор (`clUseServer`).

Набор данных ADO – класс TCustomADODataSet, компонент TADODataSet

Класс `TCustomADODataSet` служит связующим звеном между интерфейсами ADO и элементами управления библиотеки VCL. В своем первозданном виде класс не применяется, но на его основе (см. рис. 39.1) реализованы четыре компонента (`TADODataSet`, `TADOTable`, `TADOQuery` и `TADOStoredProc`), охватывающие весь спектр задач по управлению и обработке данных на клиентской стороне.

Обычно класс, подобный `TCustomADODataSet`, мы называли абстрактным, но в этом случае такое определение не совсем корректно. Более правильно классифицировать его как виртуальный базовый класс ADO. Развивая идеи, заложенные в TDataSet, и являясь пропагандистом универсальной, независимой от платформы технологии доступа к набору данных, класс `TCustomADODataSet` инкапсулирует интерфейсы объектов ADO, рожденных в стенах Microsoft. Таким образом, `TCustomADODataSet` предоставляет доступ к «инородным» интерфейсам в привычном для программиста Delphi виде – в качестве свойств и методов.

Компонент `TADODataSet` – ключевой элемент линейки ADO и прямой наследник класса `TCustomADODataSet` – предназначен для обслуживания набора данных. Попытка однозначной классификации компонента по функциональной направленности обречена на неудачу. Его нельзя отнести ни к привычным для программиста Borland компонентам-таблицам, ни к запросам, ни к хранимым процедурам. Скорее всего, `TADODataSet` – это универсальный элемент доступа к БД, впитавший в себя многие положительные качества своих коллег по цеху обслуживания данных.

Например, о наличии черт запроса говорит то, что:

- для подключения таблицы (или нескольких таблиц) к набору данных необходимо описать текст запроса SQL в свойстве `CommandText`;
- текст запроса допускает использование параметров.

Как и у классической таблицы:

- в компоненте `TADODataSet` полученный набор данных представляется в виде строк и столбцов;
- в свойствах `IndexName` и `IndexFileName` мы можем управлять подключением индексов.

Схожесть с хранимыми процедурами в том, что:

- компонент `TADODataSet` позволяет выполнять описанные на сервере процедуры.

Доступ к набору данных

Для подключения к набору данных TADODataSet способен воспользоваться услугами компонента TADOConnection. Для этого достаточно обратиться к компоненту-соединителю, сославшись на него в свойстве:

```
property Connection: TADOConnection;
```

Если же в приложении компоненту TADODataSet места не нашлось, он может самостоятельно сформировать строку соединения (свойство ConnectionString) или подключиться к файлу UDL.

Покончив с описанием параметров соединения, необходимо конкретизировать, какой именно набор данных нас интересует. И здесь нам пригодится свойство CommandText:

```
property CommandText: WideString;
```

В нем мы описываем инструкцию SELECT, определяющую таблицу (либо объединение таблиц), как представлено в следующем примере:

```
with ADODataSet1 do
begin
CommandType := cmdText; //тип команд
CommandText := 'SELECT *
            FROM (Phone LEFT JOIN Street ON Phone.Street_Cod = Street.Street_Cod);';
{Диалект SQL Microsoft Access}
Open;
end;
```

Указав интересующий нас набор данных, вызываем метод Open() либо устанавливаем в True свойство Active. После завершения работы метода Open(), в тот момент, когда набор данных будет готов к использованию, генерируется событие OnRecordsetCreate().

Выбор библиотеки курсора

По аналогии с TADOConnection класс TADODataSet позволяет выбирать библиотеку курсора с помощью свойства CursorLocation. По умолчанию набор данных ориентирован на работу с курсором на клиентской стороне (clUseClient). В этом случае данные с сервера передаются на клиентскую машину и затем обрабатываются ее средствами. Курсор такого рода обладает максимально гибкими возможностями, в особенности в части, касающейся сортировки и фильтрации данных. При использовании курсором клиентских библиотек роль сервера ограничивается только отбором данных согласно инструкции SQL. Серверный тип курсора (clUseServer) целесообразно выбирать только в случаях пересылки на пользовательскую машину больших объемов информации при очевидной «хилости» клиентского и сетевого оборудования.

Для централизованного назначения типа курсора для всех компонентов-наборов данных приложения разумно воспользоваться однотипным свойством CursorLocation *компонента* TADOConnection.

Редактирование данных

Названия почти всех методов, обеспечивающих процесс редактирования данных, унаследованы от универсального набора данных TDataSet. Одно из немногих исключений – процедура:

```
procedure DeleteRecords(AffectRecords: TAffectRecords = arAll);
type TAffectRecords = (arCurrent, arFiltered, arAll, arAllChapters);
```

специализирующаяся на удалении записей. Поведение метода определяется значением параметра AffectRecords. По умолчанию он устанавливается в arAll, что грозит полной очисткой таблицы. Если в параметр передано значение arCurrent, то удаляется текущая запись; arFiltered – удаляются записи, отобранные текущим фильтром; arAllChapters – удаляются все записи. В остальном перечень методов, нацеленных на модификацию данных, соответствует классическим методам универсального набора данных TDataSet.

Если множество записей обслуживается в пакетном режиме, то их физическое удаление произойдет только после вызова метода UpdateBatch().

Перемещение по множеству строк RecordSet

Номер текущей записи и общее количество записей, доступных в объекте RecordSet, выясняется из информационных свойств:

```
property RecNo: Integer;         //номер текущей строки
property RecordCount: Integer;   //всего строк
```

Перемещение по открытому набору данных осуществляется с помощью методов, знакомых еще по классическому классу TDataSet:

```
procedure First;  //переход к первой записи во множестве
procedure Prior;  //предыдущая запись
procedure Next;   //следующая запись
procedure Last;   //последняя запись во множестве
function MoveBy(Distance: Integer): Integer;
```

Только метод MoveBy() требует параметр. Он перемещает курсор на Distance записей вперед или назад (при отрицательном значении аргумента). Функция возвращает количество строк, на которые действительно удалось переместиться. О нахождении на первой записи во множестве свидетельствует значение True, возвращаемое свойством BOF. Если выбрана последняя запись набора, в True установится свойство EOF.

Помимо унаследованных от класса TDataSet обработчиков событий, связанных с перемещением по набору данных, в TCustomADODataSet представлены некоторые новинки. Это события OnWillMove(), OnMoveComplete() и OnEndOfRecordset(). Их описание вы найдете в табл. 39.11.

Размер записи в байтах укажет свойство:

```
property RecordSize: Word;  //только для чтения
```

Состояние текущей записи определяется при помощи свойства:

```
property RecordStatus: TRecordStatusSet; //только для чтения
```

Свойство возвращает константу, представленную в табл. 39.9. Львиная доля констант указывает причину отказа от сохранения записи.

Таблица 39.9. Статус записи, возможные значения TRecordStatusSet

Константа	Описание
rsOK	Строка была успешно модифицирована.
rsNew	Это новая строка.
rsModified	Строка изменилась.
rsDeleted	Строка была удалена.
rsUnmodified	Строка не подвергалась изменениям.
rsDBDeleted	Строка уже была удалена из источника данных.
Причина отказа сохранить изменения в строке:	
rsInvalid	Запись не сохранена из-за неверной закладки.
rsMultipleChanges	Недопустимое воздействие на другие записи.
rsPendingChanges	Процесс обновления не завершен.
rsCanceled	Отмена операции.
rsCantRelease	Строка заблокирована.
rsConcurrencyViolation	Запись захвачена конкурирующей транзакцией.
rsIntegrityViolation	Нарушение целостности данных.
rsMaxChangesExceeded	Ожидание большого числа изменений.
rsObjectOpen	Конфликт с открытым объектом.
rsOutOfMemory	Нехватка памяти.
rsPermissionDenied	Ограничение прав пользователя.
rsSchemaViolation	Сохранение приведет к нарушению структуры основной БД.

События TCustomADODataSet

Помимо событий, унаследованных от универсального набора данных TDataSet, в классе TCustomADODataSet реализована дюжина собственных обработчиков событий. В обслуживающих эти события типизированных процедурах объявлены схожие параметры. В первую очередь это DataSet: TCustomADODataSet – ссылка на компонент, ставший источником события. Второй по частоте появления параметр – EventStatus: TEventStatus. Это переменная, в которую будет возвращено описание статуса события (см. табл. 39.6). Исследовав EventStatus, программист делает вывод о том, насколько успешно завершилось то или иное событие.

В момент создания и инициализации множества записей Recordset генерируется событие:

```
property OnRecordsetCreate: TRecordsetCreate;
type TRecordsetCreate = procedure(DataSet: TCustomADODataSet;
                                  const Recordset: _Recordset) of object;
```

Второй параметр события возвращает интерфейс множества данных. Во время асинхронного получения данных периодически генерируется событие:

```
property OnFetchProgress: TFetchProgressEvent;
TFetchProgressEvent = procedure(DataSet: TCustomADODataSet; Progress, MaxProgress:
Integer; var EventStatus: TEventStatus) of object;
```

Параметры Progress и MaxProgress возвращают значения, позволяющие программисту информировать пользователя о проценте выполнения задачи – Trunc(Progress/MaxProgress * 100). Здесь MaxProgress – сколько всего строк будет передано в набор строк, а Progress – номер получаемой в данный момент времени строки. После «закачки» всех строк данных генерируется событие:

```
property OnFetchComplete: TRecordsetEvent;
TRecordsetEvent = procedure(DataSet: TCustomADODataSet; const Error: Error;
                            var EventStatus: TEventStatus) of object;
```

Половина предусмотренных у TCustomADODataSet событий парные: OnWill...() и On...Complete(). Событие OnWill...() сообщает о начале какого-то действия, а On...Complete() информирует о его завершении. Первая пара таких событий связана с изменением содержимого поля: OnWillChangeField() и OnFieldChangeComplete(). Перед изменением значения в поле ADO (или группе полей ADO) множества записей Recordset генерируется событие:

```
property OnWillChangeField: TWillChangeFieldEvent;
TWillChangeFieldEvent = procedure(DataSet: TCustomADODataSet;
                                  const FieldCount: Integer; const Fields: OLEVariant;
                                  var EventStatus: TEventStatus) of object;
```

Параметр FieldCount – число полей, подвергаемых изменениям; Fields – массив полей. Обратите внимание, что речь идет не о привычных для программистов Delphi классах TField. Ссылка на поля ADO возвращается в виде универсального типа данных OLEVariant.

После внесения изменений в поле (поля) ADO генерируется событие:

```
property OnFieldChangeComplete: TFieldChangeCompleteEvent;
TFieldChangeCompleteEvent = procedure(DataSet: TCustomADODataSet;
                        const FieldCount: Integer; const Fields: OleVariant;
                        const Error: Error; var EventStatus: TEventStatus) of object;
```

Все параметры события уже обсуждались. Напомню только, что в Error мы найдем интерфейс объекта-ошибки ADO.

Очередная пара событий обслуживает все изменения, происходящие со строкой (объект ADO Record) множества данных Recordset. При любых изменениях в состоянии строки (или нескольких строк) вызывается событие:

```
property OnWillChangeRecord: TWillChangeRecordEvent;
TWillChangeRecordEvent = procedure(DataSet: TCustomADODataSet;
                                   const Reason: TEventReason; const RecordCount: Integer;
                                   var EventStatus: TEventStatus) of object;
```

Что именно послужило причиной генерации этого события, выясняется по состоянию параметра Reason (табл. 39.10). Параметр RecordCount – количество измененных строк.

Таблица 39.10. Причина вызова события, возможные значения TEventReason

Константа	Описание
erAddNew	Добавление новой строки.
erDelete	Удаление записи.
erUpdate	Внесение изменений в строку.
erUndoUpdate	Отмена последних изменений.
erUndoAddNew	Отмена вставки новой строки.
erUndoDelete	Отмена удаления строки.
erRequery	Множество записей обновлено методом Requery().
erResynch	Множество записей обновлено методом Resynch().
erClose	Множество записей закрыто.
erMove	Курсор переместился.
erFirstChange	Первое изменение данных.
erMoveFirst	Курсор переместился на первую строку.
erMoveNext	Курсор переместился на следующую строку.
erMovePrevious	Курсор переместился на предыдущую строку.
erMoveLast	Курсор переместился на последнюю строку.

После изменения одной или нескольких записей вызывается завершающее событие:

```
property OnRecordChangeComplete: TRecordChangeCompleteEvent;
TRecordChangeCompleteEvent = procedure(DataSet: TCustomADODataSet;
                    const Reason: TEventReason; const RecordCount: Integer;
                    const Error: Error; var EventStatus: TEventStatus) of object;
```

Очередная пара событий, OnWillChangeRecordset() и OnRecordsetChangeComplete(), реагирует на изменения во множестве записей Recordset:

```
property OnWillChangeRecordset: TRecordsetReasonEvent;
property OnRecordsetChangeComplete: TRecordsetErrorEvent;
TRecordsetReasonEvent = procedure(DataSet: TCustomADODataSet;
              const Reason: TEventReason; var EventStatus: TEventStatus) of object;
```

Параметр Reason описан в табл. 39.10, остальные параметры вам уже знакомы. В табл. 39.11 представлены три события, связанные с перемещением курсора по множеству записей:

Таблица 39.11. События, вызываемые при перемещении по множеству строк

Событие	Описание
property OnWillMove: TRecordsetReasonEvent; TRecordsetReasonEvent = procedure(DataSet: TCustomADODataSet; const Reason: TEventReason; var EventStatus: TEventStatus) of object;	Событие вызывается перед началом перемещения к новой строке. Оно может быть инициировано методами First, Last, Next, Prior, MoveBy, Insert, Delete, Post, Requery и Resynch. Именно об этом и уведомляет параметр Reason.

Событие	Описание
`property OnMoveComplete: TRecordsetErrorEvent;` `TRecordsetErrorEvent = procedure(DataSet: TCustomADODataSet; const Reason: TEventReason; const Error: Error; var EventStatus: TEventStatus) of object;`	Сигнализирует о факте перехода к новой строке.
`property OnEndOfRecordset: TEndOfRecordsetEvent;` `TEndOfRecordsetEvent = procedure (DataSet: TCustomADODataSet; var MoreData: WordBool; var EventStatus: TEventStatus) of object;`	Событие генерируется в момент достижения последней строки из множества `RecordSet`. Здесь параметр `MoreData` в состоянии `true` сигнализирует о том, что в наборе `RecordSet` появились новые строки.

Обновление множества записей

При необходимости обновить полученное в результате выполнения команды множество записей обращаемся к процедуре `Refresh()` или `Requery()`. Вызов метода инициирует повторную отправку компонентом SQL-запроса в адрес провайдера, т. е. это аналог последовательного вызова методов `Close()` и `Open()`.

Работа с индексами

Для сбора имен доступных индексов предназначена процедура:

```
procedure GetIndexNames(List: TStrings);
```

Для определения текущего активного индекса необходимо передать его название в свойство:

```
property IndexName: string;
```

Количество полей, входящих в состав индекса, находится в свойстве

```
property IndexFieldCount: Integer; //только для чтения
```

Зная общее число полей в индексе, мы сможем получить доступ к отдельному полю при помощи свойства-массива:

```
property IndexFields[Index: Integer]: TField;
```

Достоинство свойства в том, что оно позволяет работать с полем в стиле VCL, т. к. возвращает `TField`.

Сортировка записей

Для изменения порядка следования строк в результирующем наборе пригодится свойство:

```
property Sort: WideString;
```

В свойство передается строка с текстом запроса, но без упоминания инструкции `ORDER BY`:

```
ADOQuery1.Sort := 'Phone DESC';
```

Альтернативный способ сортировки записей в наборе обеспечивается свойством:

```
property IndexFieldNames: string;
```

Это перечень названий полей, участвующих в сортировке. Физически сортировка осуществляется на стороне сервера, а клиент только получает результирующий набор данных. Если провайдер не способен поддерживать такой способ упорядочивания результирующего набора данных, то перед обращением к свойству IndexFieldNames необходимо перевести свойство CursorLocation в состояние clUseClient.

Поиск данных по индексу

Поиск записи в соответствии с текущим индексом производит функция:

```
function Seek(const KeyValues: Variant;
              SeekOption: TSeekOption = soFirstEQ): Boolean;
```

Возможные опции поиска представлены в табл. 39.12.

Таблица 39.12. Возможные опции поиска TSeekOption

Значение	Описание
soFirstEQ	Курсор устанавливается на первую найденную запись. Если ни одной записи не обнаружено, указатель переходит к последней строке набора.
soLastEQ	Курсор позиционируется на последней удовлетворяющей условиям поиска записи. Если ни одной записи не обнаружено, указатель переходит к последней строке набора.
soAfterEQ	Если запись обнаружена, курсор позиционируется на ней или сразу после нее.
soAfter	Курсор устанавливается на следующей записи после найденной строки.
soBeforeEQ	Если запись обнаружена, курсор позиционируется на ней или перед ней.
soBefore	Курсор устанавливается на строке, предшествующей найденной.

Особенности изоляции транзакций в наборе данных

При организации многопользовательской обработки данных обязательно стоит обратить внимание на определение уровня изоляции между параллельно выполняющимися транзакциями. Мы уже говорили об этом при обсуждении элемента управления TADOConnection. В свою очередь в классе TCustomADODataSet предусмотрен дополнительный механизм, определяющий уровни блокировки отдельной строки во множестве строк.

```
property LockType: TADOLockType; //по умолчанию ltOptimistic
type TADOLockType = (ltUnspecified, ltReadOnly, ltPessimistic, ltOptimistic,
ltBatchOptimistic);
```

Значение ltUnspecified свидетельствует о том, что не удалось детерминировать тип блокировки; ltReadOnly – запись доступна только для чтения; ltPes-

simistic – пессимистическая блокировка: после вашего обращения к записи она блокируется для других пользователей; ltOptimistic – оптимистическая блокировка заставляет провайдера блокировать запись только в том случае, когда вы приступаете к ее редактированию; ltBatchOptimistic – оптимистическая блокировка, допускающая пакетные обновления.

Уровень блокировки строки, LockType*, должен быть назначен еще до обращения к набору данных. По умолчанию он оптимистический –* ltOptimistic.

Фильтрация множества записей

Ряд свойств и методов класса предназначен для обеспечения процесса фильтрации данных во множестве строк:

```
property Filter: string;                    //текст фильтра
property FilterOptions: TFilterOptions;     //опции фильтрации
property Filtered: Boolean;                 //включение/отключение фильтра
```

На синтаксисе описания фильтра мы останавливаться не будем, т. к. он подробно изложен в главе 33 «Универсальный набор данных TdataSet». В процедуру передаются два аргумента: DataSet – ссылка на экземпляр класса TCustomADODataSet, к которому мы планируем применить фильтр, и FilterText – текст фильтра.

Кэширование записей

Таблицы БД могут содержать десятки или даже сотни тысяч записей. Как вести себя клиентским приложениям, запросившим от СУБД весь набор данных? Ведь может статься, что клиентский компьютер будет не в состоянии «проглотить» такой кусок пирога за один присест. Простейшим решением подобной проблемы является организация чтения данных порциями. Для управления количеством обслуживаемых набором данных строк используйте свойство:

```
property MaxRecords: Integer; //по умолчанию 0
```

По умолчанию это свойство установлено в 0, что свидетельствует об отсутствии каких-либо ограничений. Можно долго обсуждать вопрос о том, какое значение MaxRecords будет наиболее оптимальным для соблюдения баланса между скоростью доступа к данным и ресурсными ограничениями клиентского компьютера. Ведь результат зависит от многих показателей: пропускной способности сетевого оборудования, производительности сервера, числа пользователей, характеристик клиентского компьютера и качества нашего программного обеспечения.

Другим приемом оптимизации работы клиентских приложений БД является организация буферизации записей. Размер памяти, отводящейся для кэширования строк, определяется свойством:

```
property CacheSize: Integer;//по умолчанию 1
```

В качестве единицы измерения используются не байты, а записи. По умолчанию размер буфера установлен в 1. Это значит, что в памяти хранится всего

одна строка данных, и для того чтобы обратиться к очередной записи, элемент управления будет вынужден отправить запрос к провайдеру. Если увеличить размер кэша, скажем до 100, то в момент обращения к набору данных элемент управления затребует от провайдера сразу 100 строк и разместит их в памяти. Теперь перемещение по строкам будет происходить значительно быстрее, но до тех пор, пока не потребуется 101 строка. В этот момент повторится процедура запроса очередной сотни строк. У буфера большого размера еще один недостаток проявляется при работе в многопользовательском режиме. При параллельной обработке данных считанные в кэш записи не обновляются, т. е. не отражают изменения, которые возможно сделали другие пользователи.

Некоторого ускорения при чтении записей можно добиться, немного поэкспериментировав с размером блока чтения:

```
property BlockReadSize: Integer; //по умолчанию 0 - блочный режим чтения отключен
```

Свойство определяет количество буферизируемых записей за одну операцию чтения. При этом связанные с набором данных элементы отображения данных не обновляются. Для отключения режима поблочного чтения установите `BlockReadSize` в нулевое значение. При `BlockReadSize>0` состояние набора данных установится в `State=dsBlockRead`.

Организация пакетного обновления данных в ADO

Модель ADO поддерживает два основных режима обновления данных: **немедленное** и **пакетное** (отложенное) обновление.

Обычная работа пользователя с БД сводится к процессу вставки, редактирования и удаления данных. В режиме немедленного обновления при поступлении команды на изменение записи СУБД сразу инициирует транзакцию, в рамках которой в БД моментально вносятся все изменения, произведенные пользовательским приложением. Все происходит гладко, пока доступ клиентского приложения к ресурсу базы данных не ограничен ни скоростью обмена сети или модема, ни занятостью сервера в связи с обслуживанием транзакций, инициированных другими клиентами, ни чем-то еще подобным. Но как только возможности СУБД или сети начинают расходиться с пожеланиями пользователей, клиент сталкивается с проблемами внесения обновлений в данные. В таком случае по истечении времени тайм-аута приложение сигнализирует о невозможности сохранения модификаций, а работа оператора превращается в сплошное мучение.

В качестве одного из решений изложенной проблемы программистами предложено **пакетное обновление данных**. Идея пакетного обновления заключается в том, что все изменения в данных накапливаются в локальном буфере пользовательского компьютера и отправляются в адрес сервера одним пакетом в рамках единой транзакции. Такой подход значительно снижает нагрузку на СУБД и облегчает сетевой трафик. Однако наряду с преимуществами, отложенное обновление данных может стать потенциальным источником ошибок, особенно в случае многопользовательской работы с БД. Ведь нельзя исключить вероятность одновременного захвата двумя пользовате-

лями одних и тех же данных с целью редактирования, в результате чего можно получить абсолютно непредсказуемые последствия. Поэтому при использовании пакетного обновления особое внимание стоит уделить изоляции параллельных транзакций.

Для организации пакетного обновления данных в ADO программист должен выполнить следующий перечень действий:

- Свойство `LockType` у потомка класса `TCustomADODataSet` переводится в режим пакетного обновления – `ltBatchOptimistic`.
- Определяется степень изоляции пользовательского набора данных от изменений данных на сервере. Для этого тип курсора `CursorType` устанавливается в режим `ctKeySet` или `ctStatic`.
- При необходимости установить фильтр на передаваемые данные используется свойство `FilterGroup`.
- Для отправки на сервер пакета с измененными данными вызывается метод `UpdateBatch()`. При отказе от обновления измененные на клиентской стороне данные восстанавливаются методом `CancelBatch()`.

Степень защиты данных, загруженных в локальный кэш пользовательского компьютера, определяется не только уровнем изоляции транзакций, но и типом курсора, который будет «пробегать» строки набора данных:

```
property CursorType: TCursorType;
type TCursorType = (ctUnspecified, ctOpenForwardOnly, ctKeyset, ctDynamic, ctStatic);
```

Тип курсора (табл. 39.13) устанавливается еще до обращения к набору данных. Если провайдер не в состоянии обеспечить запрашиваемый тип курсора, он имеет право назначить курсор по своему усмотрению.

Таблица 39.13. Значения TCursorType

Тип курсора	Описание
`ctUnspecified`	Неопределенный (назначаемый по умолчанию) тип курсора.
`ctDynamic`	**Динамический курсор**. Добавления, изменения и удаления видны и другим пользователям. Разрешается перемещение по множеству записей в обоих направлениях.
`ctStatic`	**Статический курсор**. Статическая копия множества записей может использоваться для поиска данных или генерации отчетов. Добавления, изменения или удаления записи не видны другим пользователям.
`ctOpenForwardOnly`	**Однонаправленный курсор** (настройка по умолчанию). Идентичен статическому курсору, за исключением того, что перемещение по записям доступно только в одном направлении – вперед. Это весьма эффективный курсор, при условии, что планируется только один проход по множеству данных.
`ctKeySet`	Поведение подобно динамическому курсору, за исключением того, что вы не можете видеть записи, вносимые другими пользователями. Записи, удаленные другими пользователями, также недоступны. Изменения данных, проводимые другими пользователями, видимы.

Для обеспечения пакетной передачи данных курсор должен находиться в состоянии ctKeySet *или* ctStatic.

В результате работы часть помещенных в буфер записей может быть модифицирована, часть удалена, а часть оставлена без изменений. Поэтому в ADO предусмотрен фильтр, позволяющий отбирать записи из множества Recordset не на основе фактических данных, содержащихся в полях записи, а на основе контроля технического показателя состояния модифицируемой строки:

```
property FilterGroup: TFilterGroup;
```

Тип данных TFilterGroup представлен в табл. 39.14.

Таблица 39.14. Тип фильтрации TFilterGroup

Тип фильтра	Описание
fgUnassigned	Используется только внутри ADO-компонента; признак того, что фильтрация не применялась.
fgNone	Отключает любую фильтрацию. Аналог установки свойства Filtered в false.
fgPendingRecords	Показывает только модифицированные строки, но изменения пока не сохранялись (не был вызван метод UpdateBatch) или отменены (метод CancelBatch).
fgAffectedRecords	Показывает только последние модифицированные строки.
fgFetchedRecords	Показывает только строки в текущем кэше. Это результаты последнего запроса к базе данных.
fgPredicate	Показывает только удаленные строки.
fgConflictingRecords	Показывает только строки, которые подвергались модификациям, но не были сохранены из-за ошибок в процессе сохранения.

Для того чтобы получить возможность работать с фильтром FilterGroup*, необходимо перевести набор данных в пакетный режим модификации (*LockType=ltBatchOptimistic*), а свойство* Filtered *– в* True.

Для сохранения содержимого кэша в таблицах БД вызывается процедура:

```
procedure UpdateBatch(AffectRecords: TAffectRecords = arAll);
```

С классом TAffectRecords мы уже сталкивались при обсуждении метода DeleteRecords(). По умолчанию единственный параметр процедуры установлен в состояние arAll, что соответствует указанию сохранять все строки из буфера. Значение arCurrent отправит для сохранения только текущую строку; arFiltered – только строки, отобранные в результате фильтрации; arAllChapters воздействует на все записи во всех подчиненных множествах Recordset, включая даже те строки, которые скрыты фильтрацией.

Отменой пакетной передачи данных ведает процедура:

```
procedure CancelBatch(AffectRecords: TAffectRecords = arAll);
```

Указанный метод может вызываться не только в пакетном режиме, но и при работе в режиме немедленного обновления данных, но при условии, что в параметр AffectRecords передано значение arCurrent, что восстановит текущую запись. Значение arFiltered укажет на то, что метод будет применен только к записям, отобранным текущим фильтром. Значение arAll установит режим по умолчанию; в этом случае функция возвратит в исходное состояние все записи. Последний вариант arAllChapters определит, что обновление затронет все данные ADO, включая даже записи, которые не видны из-за включенного фильтра.

Если мы работаем с клиентским курсором (CursorLocation=clUseClient), то некоторые особенности возврата отредактированных данных на сервер зависят от состояния свойства:

```
property MarshalOptions: TMarshalOption; //по умолчанию moMarshalAll
type TMarshalOption = (moMarshalAll, moMarshalModifiedOnly);
```

При установке свойства в состояние moMarshalModifiedOnly набор данных отправит на сервер только те записи, которые подверглись изменениям на клиентской стороне. Если же это свойство останется в состоянии по умолчанию (moMarshalAll), то в адрес сервера будут переданы все данные.

Отправка команд

Объекты, построенные на основе класса TCustomADODataSet, инкапсулируют командный объект TADOCommand:

```
property Command: TADOCommand;
```

Данное свойство и ряд других свойств, связанных с интерфейсом _Command, объявлены в разделе protected и предназначены только для внутреннего пользования. Физическая недоступность командного интерфейса не означает, что потомки класса TCustomADODataSet не способны общаться с БД при помощи инструкций SQL. Просто каждый из компонентов реализует это своими силами. Например, в запросе TADOQuery инструкция SQL вносится в свойство SQL, а в хранимой процедуре TADOStoredProc имя процедуры передается через свойство ProcedureName.

Портфельный режим обработки данных

В наборе данных ADO предусмотрена пара процедур, позволяющих организовать обслуживание набора данных (например, полученного от удаленного сервера) на компьютере, временно отключенном от сети. Это так называемый портфельный режим. Пользователь загружает требуемые данные с сервера и сохраняет их в файл. Затем множество записей обрабатывается, а когда компьютер вновь включается в сеть, обновленные данные сбрасываются на сервер.

```
procedure SaveToFile(const FileName: string = ''; Format: TPersistFormat = pfADTG);
procedure LoadFromFile(const FileName: WideString);
```

Параметр FileName определяет имя файла. Формат хранения данных определяется параметром Format:

```
type TPersistFormat = (pfADTG, pfXML);
```

По умолчанию Delphi предлагает двоичный формат хранения данных ADTG (Advanced Data Tablegram). Второй вариант – расширенный язык разметки (XML, Extensible Markup Language).

Командный объект ADO – TADOCommand

Класс TADOCommand инкапсулирует реализованный в ADO командный объект. Элемент управления предназначен для отправки в адрес базы данных отдельных команд. Синтаксис команд полностью соответствует синтаксису структурированного языка запросов SQL-92. Однако есть одно существенное ограничение: отправив в адрес провайдера команду, элемент TADOCommand практически не интересуется результатами ее выполнения. Исходя из этой особенности в качестве команд целесообразно передавать SQL-инструкции, не возвращающие никаких данных. Вместе с тем в порядке исключения командный объект способен вернуть множество записей _Recordset, но при условии, что обработкой полученных результатов займется экземпляр класса TADODataSet.

С задачами командного компонента превосходно справится и экземпляр класса TADOQuery. *Однако применение не столь ресурсоемкого элемента* TADOCommand *может оказаться более эффективным при отправке SQL-команд, не возвращающих результата (*CREATE, DROP, INSERT, UPDATE, DELETE *и т. д.)*.

Для соединения с провайдером командный объект может выбрать один из двух путей. Если приложение подключается к данным централизованно (при посредничестве элемента управления TADOConnection), то стоит воспользоваться этой возможностью и связать командный компонент с элементом управления TADOConnection при помощи свойства Connection. В таком случае процесс подключения перекладывается на TADOConnection и перестает интересовать командный элемент управления. Второй путь – это самостоятельное подключение при помощи соединительной строки ConnectionString.

Текст отправляемой команды описывается в свойстве CommandText. Мы уже говорили, что это инструкция на языке SQL, причем желательно не возвращающая результатов, нуждающихся в последующей обработке. Если команда содержит какие-то параметры, например аргументы хранимой процедуры, определите их в свойстве:

```
property Parameters: TParameters;
```

Для того чтобы список параметров регенерировался автоматически при изменении команды, установите в True свойство:

```
property ParamCheck: Boolean;
```

Исключительно для удобства программиста команды систематизированы по типам с помощью свойства CommandType. Определяя тип команды (см. табл. 39.1), программист явным образом указывает объекту ADO, какого рода инструкцию он планирует описать в его свойстве CommandText. Например,

определив тип команды как cmdStoredProc, вместо явного текстового описания инструкции SQL в свойстве CommandText достаточно указать имя хранимой процедуры, которую мы планируем выполнить.

Самый главный метод Execute() командного объекта ADO отправляет хранящиеся в CommandText инструкции SQL на выполнение. Различают три перегружаемые вариации этого метода, которые были рассмотрены в начале главы в разделе, посвященном интерфейсу _Command. В простейшем случае функция вообще не требует каких-либо аргументов.

```
with ADOCommand1 do
begin
CommandType := cmdStoredProc;      //тип команды – хранимая процедура
CommandText := 'MyStoredProc';     //имя процедуры
ADODataSet1.Recordset := Execute; //выполнение команды
end;
```

Время, отводимое на выполнение команды, определяется свойством CommandTimeout и по умолчанию составляет 30 секунд. На порядок выполнения инструкции оказывают влияние опции (табл. 39.15), установленные в свойстве:

```
property ExecuteOptions: TExecuteOptions;
```

Таблица 39.15. Опции TExecuteOptions

Опция	Описание
eoAsyncExecute	Передавать команду в асинхронном режиме.
eoAsyncFetch	Указывает, что если кэш переполнен (размерность определяется свойством CacheSize класса TCustomADODataSet), но еще не все запрашиваемые строки получены, то оставшиеся «за бортом» строки должны передаваться асинхронно.
eoAsyncFetchNonBlocking	Асинхронное чтение. Если требуемая строка не была найдена, текущая строка автоматически перемещается в конец файла.
eoExecuteNoRecords	Командный объект не будет возвращать записи, независимо от передаваемой в адрес БД инструкции SQL.

*Асинхронной операцией с базой данных называют такую операцию, которая, отправив команду СУБД, сразу, не дожидаясь выполнения этой команды, возвращает управление приложению. При выполнении **синхронной операции** управление приложению не передается до тех пор, пока СУБД не выполнит команду. По умолчанию компоненты ADO настроены на работу в синхронном режиме.*

Таблица, запрос и хранимая процедура – TADOTable, TADOQuery и TADOStoredProc

Компоненты TADOTable, TADOQuery и TADOStoredProc являются обратной стороной медали универсального набора данных TADODataSet. Эти компоненты созданы лишь для того, чтобы обеспечить максимально удобный стиль работы

для программистов, которые привыкли работать с классическими для Borland компонентами-таблицами, запросами и хранимыми процедурами. Компоненты целиком и полностью построены на базе TCustomADODataSet, и за привычными еще со времен Delphi 1.0 названиями свойств и методов прячутся все те же интерфейсы ADO: _Command, _Recordset и _Connection. В доказательство этих слов предлагаю заголовок и реализацию класса хранимой процедуры TADOStoredProc:

```
  TADOStoredProc = class(TCustomADODataSet)
  public
    constructor Create(AOwner: TComponent); override;
    procedure ExecProc;
  published
    property CommandTimeout;
    property DataSource;
    property EnableBCD;
    property ProcedureName: WideString read GetCommandText write SetCommandText;
    property Parameters;
    property Prepared;
  end;
//…
constructor TADOStoredProc.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Command.CommandType := cmdStoredProc;
  Command.CommandTextAlias := 'ProcedureName'; { Do not localize }
end;

procedure TADOStoredProc.ExecProc;
begin
  InitializeMasterFields(Self);
  Command.Execute;
end;
```

Как видите, все отличия TADOStoredProc от ее предка TCustomADODataset сводятся к двум дополнительным строкам конструктора и косметическим изменениям в реализации метода ExecProc().

Сервисные методы модуля ADODB

В модуле ADODB объявлен ряд процедур и функций, позволяющих программисту создавать файл соединения программным способом. В первую очередь это процедура:

```
procedure CreateUDLFile(const FileName, ProviderName, DataSourceName: WideString);
```

Здесь FileName – имя будущего UDL-файла, ProviderName – имя провайдера, DataSourceName – имя источника данных.

```
uses ComObj, ADODB;
var FileName : string;
begin
FileName:=DataLinkDir+'\tst.udl';   //формируем имя файла
```

```
    try
    CreateUDLFile(FileName, 'MSDASQL.1', 'Phones');
    except on EOLESysError do
        begin
        //обработка ошибки
        end;
    end;
end;
```

Обратите внимание, что при формировании имени UDL-файла с помощью метода:

```
function DataLinkDir: string;
```

мы выясняем штатный каталог Windows для хранения всех UDL-файлов. По умолчанию это:

```
C:\Program Files\Common Files\System\Ole DB\Data Links\
```

Для того чтобы приложение смогло собрать весь перечень UDL-файлов из каталога, определенного в параметре `Directory`, нам пригодится метод:

```
function GetDataLinkFiles(FileNames: TStrings; Directory: string = ''): Integer;
```

Если в параметр `Directory` не передавать никаких значений, то поиск файлов будет произведен в каталоге по умолчанию. Собранные данные передаются в список `FileNames`, а функция возвращает количество найденных файлов.

Для построения полного списка имеющихся в системе провайдеров (драйверов OLE DB) пригодится метод:

```
procedure GetProviderNames(Names: TStrings);
```

Результаты анализа будут помещены в список имен `Names`.

Для вызова диалога выбора UDL-файла (рис. 39.3), воспользуйтесь методом:

```
function
PromptDataLinkFile(ParentHandle:
THandle; InitialFile: WideString):
WideString;
```

Здесь `ParentHandle` — указатель на форму, владеющую диалогом; `InitialFile` — файл, инициирующий каталог диалога и фильтр по расширению файла. Функция возвращает имя выбранного пользователем файла.

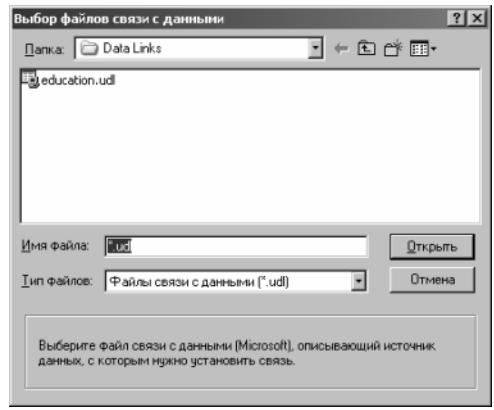


Рис. 39.3. *Стандартный диалог выбора файла связи с данными*

Если вместо UDL-файла планируется просто сформировать соединительную строку, то лучшим помощником станет метод:

```
function PromptDataSource(ParentHandle: THandle; InitialString: WideString):
WideString;
```

Метод вызовет диалоговое окно определения параметров строки соединения (рис. 39.4). Параметры аналогичны параметрам функции `PromptDataLinkFile()`, но в качестве результата метод возвратит текст с параметрами соединения.

Резюме

Наряду с фирменной технологией доступа к данным, построенной на основе механизма Borland Database Engine, среда Delphi открыта для работы с альтернативными методами доступа к БД. В частности, Delphi позволяет создавать высокоэффективные приложения баз данных, используя технологию Microsoft® ActiveX® Data Objects (ADO). Все необходимые интерфейсы объектов ADO инкапсулированы в семь компонентов визуальной библиотеки. При этом Delphi позволяет работать с данными как в стиле Microsoft, используя концепцию множества записей `Recordset`, так и в духе Borland, опираясь на функциональные возможности универсального набора данных `TDataSet`.

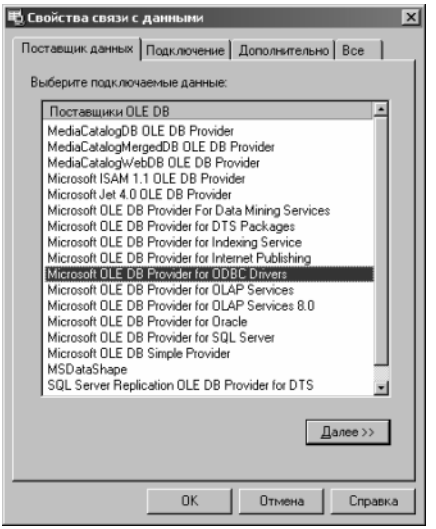


***Рис. 39.4.** Окно определения параметров строки соединения*

40

Компоненты InterBase

Если вы только начинаете изучать СУБД и мечтаете разработать свою первую клиент–серверную БД, то самой лучшей платформой для этого станет сервер InterBase фирмы Borland. И тому несколько причин. Во-первых, это весьма компактный программный продукт, для инсталляции которого достаточно 20 Мбайт. Во-вторых, в сравнении с такими монстрами, как Oracle, Informix и Microsoft SQL Server, СУБД InterBase предъявляет минимальные требования к ресурсам компьютера, на котором он будет функционировать. В-третьих, на работе с InterBase специализируется отдельная одноименная линейка компонентов Delphi. В-четвертых, для работы с InterBase не требуются услуги BDE. В-пятых, это действительно кроссплатформенная система, поддерживающая Windows и Linux.

Хорошая новость для недоброжелателей BDE: для организации работы клиентского приложения БД InterBase достаточно установить на компьютер лишь одну библиотеку – файл `gds32.dll`. *Данный файл (версии WI-V7.1.0.181) занимает всего 396 Кбайт! Согласитесь, это существенно скромнее, чем требования пакета BDE.*

Мир должен знать своих героев – основа кода компонентов InterBase (IBX) написана программистом Грегори Дилтцом в 1998 году.

Доступ к базе данных InterBase – компонент TIBDatabase

Компонент `TIBDatabase` предназначен для организации доступа клиентского приложения к БД, обслуживаемой сервером InterBase. Наличие этого компонента в клиентском приложении обязательно, т. к. его услугами пользуются практически все остальные элементы управления InterBase. Класс построен на основе хорошо нам знакомого `TCustomConnection` (рис. 40.1) и описан в модуле `IBDatabase`.

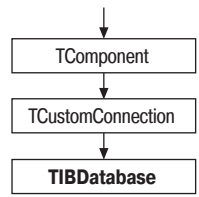


***Рис. 40.1.** Иерархия TIBDatabase*

Выбор диалекта SQL

Как и все SQL-ориентированные СУБД, сервер InterBase не только поддерживает, но и существенно расширяет набор инструкций стандартного структурированного языка запросов SQL-92. В рамках этой главы невозможно даже поверхностно поведать об особенностях InterBase SQL (ISQL), тем более, что к настоящему времени уже сменилось три диалекта ISQL. Порядка ради отметим, что первая версия ISQL предназначалась для InterBase 5 и более ранних релизов сервера. Второй диалект сыграл переходную роль и, по сути, не использовался. И наконец, СУБД InterBase версии 7 работает на диалекте ISQL 3. По умолчанию компонент `TIBDatabase` нацелен на работу с ISQL 3, но если вдруг вам потребуется обслуживать более ранние версии InterBase, понизьте номер диалекта в свойстве:

```
property SQLDialect: Integer;   //по умолчанию 3
```

Выбор диалекта осуществляется до подключения компонента к БД. Сразу оговорюсь, что все примеры этой главы ориентированы на ISQL 3. Свойство `SQLDialect` продублировано:

```
property DBSQLDialect: Integer; //только для чтения
```

Но здесь оно доступно только для чтения. При несоответствии уровней диалекта сервера и клиента `TIBDatabase` компонент генерирует событие:

```
property OnDialectDowngradeWarning: TNotifyEvent;
```

Создание и уничтожение базы данных

Компонент TIBDatabase вполне способен создать новую БД. Для этого предназначена процедура:

```
procedure CreateDatabase;
```

Перед вызовом метода программист должен определиться с диалектом, названием БД и регистрационными данными пользователя:

```
with frmDataModule.IBDatabase1 do
  begin
    SQLDialect:=3;              //диалект ISQL
    DatabaseName:= 'C:\IBData\IBDemoDB.gdb';  //название БД
    Params.Clear;
    Params.Add('USER "SYSDBA"');           //имя пользователя
    Params.Add('PASSWORD "masterkey"');    //пароль
    Params.Add('PAGE_SIZE 4096');          //размер страницы
    CreateDatabase;
  end;
```

Во время создания новой БД компонент `TIBDatabase` должен быть свободен (не состоять в соединении ни с какой из баз данных).

Для удаления базы данных предназначен метод:

```
procedure DropDatabase;
```

Но в этом случае (перед обращением к процедуре) компонент TIBDatabase подключается к удаляемой БД и активизирует соединение.

Соединение с базой данных

Для организации соединения в первую очередь необходимо указать имя БД, воспользовавшись свойством:

```
property DatabaseName: string;
```

Имя определяется названием файла базы данных, а формат его описания зависит от операционной системы, под которой выполняется СУБД Interbase, и сетевого протокола доступа к серверу. В простейшем случае, когда база данных размещается на том же компьютере, на котором установлено наше приложение, мы ограничиваемся обычным путем к файлу:

```
IBDatabase.DatabaseName:= 'C:\IBData\IBDemoDB.gdb';
```

Но если СУБД InterBase выполняется на удаленном сервере, то порядок определения имени слегка усложняется. Для соединения с СУБД можно выбрать любой из наиболее распространенных сетевых протоколов TCP/IP или NetBEUI, но по возможности лучше применять маршрутизируемый протокол TCP/IP. Так, для станций, работающих под управлением ОС Windows NT, путь к файлу данных предваряет сетевое имя компьютера:

```
IBDatabase.DatabaseName:= 'IBComputer:C:\IBData\IBDemoDB.gdb';
```

Если по какой-то причине транспортный протокол TCP/IP не установлен, то организуем работу с помощью протокола NetBEUI, но в этом случае несколько изменится порядок описания имени БД:

```
IBDatabase.DatabaseName:= '\\IBComputer\C:\IBData\IBDemoDB.gdb';
```

Параметры соединения описываются в массиве строк:

```
property Params: TStrings;
```

Например, на время разработки в это свойство можно внести данные об имени и пароле пользователя. При этом не забудьте отключить автоматический вызов диалога регистрации, установив в False свойство:

```
property LoginPrompt : Boolean;
```

Завершив все подготовительные операции, вызываем унаследованный от TCustomConnection метод:

```
procedure Open;
```

либо переводим в true свойство:

```
property Connected : Boolean;
```

Помимо прочего, свойство Connection может использоваться для проверки факта соединения с БД.

Для доступа к серверу InterBase необходимо указать имя и пароль пользователя. По умолчанию права системного администратора предоставит следующая учетная запись: пользователь «SYSDBA», пароль «masterkey».

```
procedure TfrmDataModule.DataModuleCreate(Sender: TObject);
begin
  with IBDatabase1 do
  begin
  DatabaseName:= 'C:\IBData\IBDemoDB.gdb';
  LoginPrompt:=False;
  Params.Clear;
  Params.Add('user_name=SYSDBA');
  Params.Add('password=masterkey');
  Connected:=True;
  end;
end;
```

Подчеркну, что такой код применяется только во время разработки проекта, а окончательная версия нашего программного продукта просто обязана потребовать от пользователя ввод регистрационных данных.

Большинство программистов предпочитают управлять ходом соединения клиентского приложения с БД вручную в коде программы, а не «железобетонно» переводить свойство `Connected` в состояние `true` еще в период визуального проектирования. Но зачастую контакт с БД необходим и во время разработки, например для инициализации свойств компонентов хранимых процедур или для генерации списка статических полей у компонентов наборов даных. После этого перед стартом программы важно не забыть вернуть свойство `Connected` в состояние `false`, иначе это может не понравиться Delphi. Если подключение/отключение надо произвести пару раз за день, то ничего страшного, но если ваш проект содержит сотни хранимых процедур, запросов и таблиц, то поверьте, это слегка утомляет... К счастью, (в отличие от старого доброго `TDatabase`) в компоненте `TIBDatabase` острота проблемы значительно снижена благодаря свойству:

```
property AllowStreamedConnected : Boolean; //по умолчанию true
```

Свойство предназначено исключительно для упрощения процесса отладки проекта. По умолчанию `AllowStreamedConnected` установлено в `true`. В этом случае поведение компонента ничем не отличается от повадок классического `TDatabase`. Включив в Инспекторе объектов свойство `Connected`, мы «привязываемся» к БД и получаем те же проблемы, что имели и раньше. Но переведя свойство `AllowStreamedConnected` в `false`, мы освобождаемся от необходимости постоянно включать/отключать `Connected` во время разработки. Компонент самостоятельно, без какого-то постороннего вмешательства, отключится от БД в момент старта проекта (ожидая явной команды на соединение в коде программы) и вновь подключится к БД после его остановки.

Еще одно полезное нововведение класса `TIBDatabase` – безопасное тестирование соединения. В процессе подготовки соединения перед вызовом метода `Open()` можно вызвать метод:

```
procedure TestConnected: Boolean;
```

Процедура проверит корректность параметров соединения и в случае успеха вернет значение `true`. При ошибке мы получим `false`, но (что очень важно) при этом не будет сгенерирована исключительная ситуация.

Для настройки параметров соединения во время разработки проекта программист может обратиться к контекстному меню компонента и воспользоваться пуктом Database edittor (рис. 40.2).

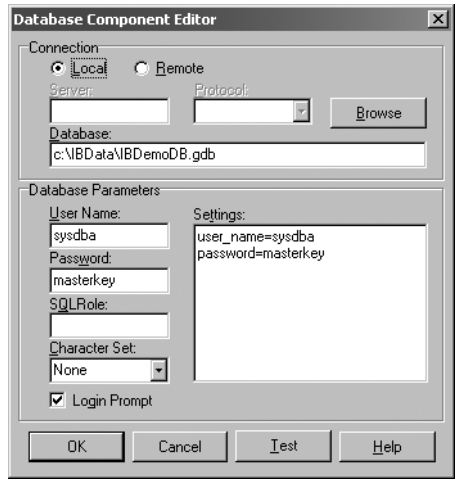


***Рис. 40.2.** Редактор параметров соединения*

Регистрация пользователя

Для вызова диалога регистрации пользователя установите в `true` свойство `LoginPrompt`. В этом случае обращение к методу `Open()` инициирует событие:

```
property OnLogin: TIBDatabaseLoginEvent;
TIBDatabaseLoginEvent =
procedure(Database: TIBDatabase;
LoginParams: TStrings) of object;
```

Аутентификация пользователя в компоненте `TIBDatabase` несколько отличается от знакомого нам по классам `TCustomConnection` и `TDatabase` одноименного события `OnLogin()`. В нашем случае регистрационные данные направляются в список строк `LoginParams`, как представлено в следующем примере:

```
procedure TfrmDataModule.IBDatabase1Login(Database: TIBDatabase; LoginParams:
TStrings);
var frmUserLogin: TfrmUserLogin;
begin
  frmUserLogin:=TfrmUserLogin.Create(nil); {создаем регистрационную форму}
  if frmUserLogin.ShowModal=mrOK then
    with LoginParams do
    begin
    Clear;
    Add('user_name='+frmUserLogin.edUser.Text);       //имя пользователя
    Add('password='+frmUserLogin.edPassword.Text);    //пароль
    end;
frmUserLogin.Release;
end;
```

Разрыв соединения

Кроме анализа состояния свойства `Connected` для проверки наличия соединения с БД программист может использовать методы:

```
procedure CheckActive;   //проверка активности
procedure CheckInactive; //проверка пассивности
```

Особенность процедур `Check…` состоит в том, что при невыполнении условия проверки они генерируют исключительную ситуацию.

```
uses IB;
…
procedure …
```

```
begin
  try
    IBDatabase1.CheckInactive;
    {какие-то операции, осуществление которых возможно только
     при отсутствии соединения}
  except on EIBClientError do
    begin
      MessageBox(Handle, PChar('Данная операция невозможна в активном соединении!'),
                         PChar('Ошибка'), MB_ICONERROR+MB_OK);
    end;
  end;
end;
```

Для отключения от БД обращаемся к методу `Close()` либо переводим в `false` свойство `Connected`. Пока ничего нового – все, как у компонента `TDatabase`. Но изредка возникают случаи, когда компонент `TIBDatabase` категорически отказывается разорвать соединение с БД. Это может быть объяснено внутренними проблемами сервера, например затянувшейся транзакцией.

Что же делать, если нужно все-таки прекратить работу клиентского приложения в случае его отказа подчиниться? Обращаемся к методу:

```
procedure ForceClose;
```

Процедура `ForceClose()` по возможности форсирует процесс отключения. Иногда вместо разрыва контакта с СУБД программисту необходимо отключить все активные наборы данных. Для этой цели подойдет процедура:

```
procedure CloseDataSets;
```

После обращения к методу все обслуживаемые нашим компонентом потомки `TDataSet` перейдут в пассивный режим (`Active=false`), но `TIBDatabase` по-прежнему сохранит контакт с базой.

При необходимости можно ограничить время простоя соединения, воспользовавшись свойством:

```
property IdleTimer: Integer; //по умолчанию 0 - ограничений нет
```

Время исчисляется в миллисекундах. По умолчанию свойство установлено в ноль, что означает отсутствие ограничений. По истечении времени простоя генерируется событие, которое может быть использовано для разрыва соединения с БД.

```
property OnIdleTimer: TNotifyEvent;
```

Информирование о составе БД

Для получения перечня таблиц БД необходимо вызвать процедуру:

```
procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);
```

Данные передаются в список `List`. Если необходимо собрать сведения и о системных таблицах, то передайте `true` в параметр `SystemTables`. Зная имя таблицы, мы сможем копнуть глубже и запросить список ее полей:

```
procedure GetFieldNames(const TableName: string; List: TStrings);
```

Здесь TableName – название таблицы, List – ссылка на список, в который будут собраны данные. Допустим, что на главной форме проекта frmMain размещены два компонента-списка TListBox, предназначенные для хранения перечня таблиц (компонент lbTables) и списка полей таблицы (компонент lbFields). Тогда в момент показа формы можно запросить информацию о таблицах:

```
procedure TfrmMain.FormShow(Sender: TObject);
begin
lbTables.Clear;
frmDataModule.IBDatabase1.GetTableNames(lbTables.Items, true);
if lbTables.Count> 0 then {если список не пуст, «щелкнем» по его первому элементу}
  begin
    lbTables.ItemIndex:=0;
    lbTables.OnClick(Self);
  end;
end;
```

Щелчок по элементу списка таблиц инициирует процесс сбора данных о полях выбранной таблицы:

```
procedure TfrmMain.lbTablesClick(Sender: TObject);
var TableName : string;
begin
if lbTables.ItemIndex>-1 then
  begin
    lbFields.Clear;
    TableName:=lbTables.Items.Strings[lbTables.ItemIndex];
    frmDataModule.IBDatabase1.GetFieldNames(TableName,lbFields.Items);
  end;
end;
```

Для получения перечня таких метаданных, как список таблиц, представлений, правил, индексов, целесообразно использовать компонент TIBExtract. *Более полную информацию о технических характеристиках БД предоставит компонент* TIBDatabaseInfo.

Управление транзакциями

В отличие от компонента TDatabase, предназначенного для обслуживания баз данных BDE, в TIBDatabase отсутствуют средства управления транзакциями. Но это не беда, поскольку к TIBDatabase могут быть присоединены элементы TIBTransaction, как раз и отвечающие за этот вопрос. Список всех связанных с TIBDatabase транзакций мы обнаружим в свойстве:

```
property Transactions[Index: Integer]: TIBTransaction;
```

Общее количество доступных транзакций можно узнать из свойства:

```
property TransactionCount: Integer; //только для чтения
```

Для выяснения индекса транзакции TR в массиве Transactions необходима функция:

```
function FindTransaction (TR: TIBTransaction): Integer;
```

Для того чтобы назначить транзакцию по умолчанию, обратитесь к свойству:

```
property DefaultTransaction: TIBTransaction;
```

Список Transactions может быть пополнен очередной транзакцией TR непосредственно из кода программы:

```
function AddTransaction(TR: TIBTransaction): Integer;
```

При успешном завершении метод возвратит новый индекс. Обратную операцию, исключающую транзакцию из списка, осуществит процедура:

```
procedure RemoveTransaction(Idx: Integer);
```

Метод отсоединит (но не уничтожит) транзакцию с индексом Idx от базы данных. Отключение всех компонентов TIBTransaction произведет процедура:

```
procedure RemoveTransactions;
```

Контроль за событиями

С компонентом TIBDatabase допустимо связывать произвольное число объектов, отслеживающих возникновение событий в базе данных InterBase:

```
procedure AddEventNotifier(Notifier: IIBEventNotifier);
```

В роли внешнего контролера может выступать экземпляр объекта TIBEvents либо интерфейс IIBEventNotifier. Удаление связи с уведомлением о событии осуществляется процедурой:

```
procedure RemoveEventNotifier(Notifier: IIBEventNotifier);
```

Совместная работа с SQL-монитором

В состав компонентов InterBase входит элемент управления TSQLMonitor, который позволяет производить мониторинг базы данных и управлять сведениями о выполнении SQL-запросов серверами InterBase. Происходящие с БД операции отслеживаются с помощью свойства:

```
property TraceFlags: TTraceFlags; //по умолчанию [tfQExecute,tfConnect]
```

Программист включает флаги, соответствующие интересующим его операциям (табл. 40.1).

Таблица 40.1. Возможные значения TTraceFlags

Флаг	Описание
tfQPrepare	Мониторинг подготовительных операций.
tfQExecute	Отслеживать команды на запуск.
tfQFetch	Контроль за операциями чтения Fetch.
tfError	Мониторинг за ошибками сервера.
tfStmt	Обрабатывать все команды SQL.
tfConnect	Контроль процесса соединения и разрыва соединения приложения с БД.

Флаг	Описание
tfTransact	Отслеживать команды управления транзакциями: StartTransaction, Commit и Rollback.
tfBlob	Мониторинг операций с BLOB-данными.
tfService	Обслуживать сервисы.
tfMisc	Мониторинг за всеми операциями, не связанными с рассмотренными ранее флагами.

Элементарный запрос – компонент TIBSQL

Основная отличительная черта компонента TIBSQL от своих собратьев по цеху InterBase – отсутствие в его иерархии наследования какого-либо намека на класс TDataset, обязательный для всех компонентов, обслуживающих наборы данных. Непосредственным предком TIBSQL выступает класс TComponent. В силу этого TIBSQL характеризуется минимальным спектром возможностей по обслуживанию данных, но это как раз тот случай, когда недостаток превращается в достоинство. Преимущество TIBSQL над другими элементами управления, специализирующимися на работе с данными, состоит в том, что компонент весьма аскетичен в плане потребления системных ресурсов.

Несмотря на отсутствие родственных связей между TIBSQL и универсальным набором данных TDataset (от которого берут начало все известные нам компоненты-наборы данных), при встрече с компонентом TIBSQL мы увидим знакомые имена свойств и методов. Дело в том, что при описании класса TIBSQL разработчики Delphi постарались сохранить привычные названия.

Подготовка к работе

Работа с компонентом начинается с подключения запроса к БД при помощи свойства:

```
property Database: TIBDatabase;
```

Заручившись поддержкой БД (в лице компонента TIBDatabase), находим еще одного помощника – компонент, отвечающий за обслуживание транзакций. Для этого воспользуемся свойством:

```
property Transaction: TIBTransaction;
```

Собрав всех своих друзей, переходим к подготовке инструкции SQL. Текст SQL-запроса заносится в набор строк:

```
property SQL: TStrings;
```

В момент изменения запроса генерируется событие:

```
property OnSQLChanging: TNotifyEvent;
```

Подготовка метода к выполнению производится процедурой:

```
procedure Prepare;
```

Подготовка запроса завершается переводом в состояние `true` свойства:

```
property Prepared: Boolean; //только для чтения
```

Если запрос содержит параметры, то доступ к ним осуществляется посредством свойства, содержащего массив параметров `TIBXSQLDA`:

```
property Params: TIBXSQLDA;
```

или благодаря функции, в которую передается имя параметра:

```
function ParamByName(Idx: string): TIBXSQLVAR;
```

Для передачи команды в адрес сервера используем процедуру:

```
procedure ExecQuery;
```

Например, как в следующем листинге, создающем таблицу `"STREETS"`:

```
with IBSQL1 do
begin
  sql.Clear;
  sql.Add('CREATE TABLE "STREETS"');
  sql.Add('("STREETKEY" INTEGER NOT NULL PRIMARY KEY,');
  sql.Add('"STREETNAME" VARCHAR(30) NOT NULL UNIQUE)');
  if Transaction.Active=false then Transaction.Active:=true;
  if Prepared=false then Prepare;
  ExecQuery;
end;
```

Для того чтобы определить, какой тип запроса был отправлен на выполнение, взгляните на свойство:

```
property SQLType: TIBSQLTypes; //только для чтения
```

После выполнения предыдущего примера свойство установится в состояние `SQLDDL`. Полное описание типов запросов `TIBSQLTypes` представлено в табл. 40.2.

Таблица 40.2. Тип выполняемого запроса – TIBSQLTypes

Константа	Описание
SQLCommit	Сохраняет результаты активной транзакции.
SQLDDL	Запускает на выполнение инструкцию определения данных DDL (Data Definition Language).
SQLDelete	Удаляет записи из таблицы или из активного набора курсора.
SQLExecProcedure	Вызывает хранимую процедуру.
SQLGetSegment	Читает сегмент из открытого объекта BLOB.
SQLInsert	Добавляет одну или несколько новых записей в таблицу.
SQLPutSegment	Записывает BLOB-сегмент.
SQLRollback	Производит откат транзакции.
SQLSetForUpdate	Хранимая процедура установлена для обновления.
SQLSetGenerator	Устанавливает новое значение в имеющийся генератор.

Константа	Описание
SQLSelect	Запрашивает данные из БД.
SQLStartTransaction	Запускает новую транзакцию.
SQLUnknown	Неизвестный тип SQL.
SQLUpdate	Модифицирует данные в строках таблицы, представления или в активном наборе курсора.

Обслуживание полученного набора данных

Если запрос возвращает результирующий набор данных, то для перебора всех записей в наборе пригодится следующая группа свойств и методов. Свойства:

```
property Bof: Boolean; //только для чтения
property Eof: Boolean; //только для чтения
```

сигнализируют о нахождении курсора в начале (Bof=true) или в конце набора (Eof=true). Все данные о текущей записи возвращает метод:

```
function Current: TIBXSQLDA;
```

Результат выполнения функции представляется в виде объекта TIBXSQLDA.

Компонент TIBSQL *способен просматривать записи лишь в одном направлении – от начала набора данных к его концу.*

Переход к следующей записи производит метод:

```
function Next: TIBXSQLDA;
```

Эта функция возвращает следующую строку из полученного набора данных. Для возврата к самой первой записи в наборе установите в true свойство:

```
property GoToFirstRecordOnExecute: Boolean;
```

Для организации доступа к полям текущей записи стоит воспользоваться услугами следующего трио:

```
property Fields[const Idx: Integer]: TIBXSQLVAR;     //массив полей
property FieldIndex: [FieldName: string]: Integer;   //индекс поля в массиве
function FieldByName[FieldName: string]: TIBXSQLVAR; //отдельное поле
```

Основной недостаток TIBSQL *состоит в том, что к набору данных, полученному в результате выполнения инструкции* SELECT*, нельзя получить доступ с помощью источника данных – компонента* TDataSource.

Экспорт и импорт данных

Для организации экспорта записей из наборов данных InterBase в файл и для обратной операции – импорта данных из файла – компоненты TIBSQL, TIBDataSet и TIBQuery вооружены парой методов:

```
procedure BatchInput(InputObject: TIBBatchInput);     //импорт данных
procedure BatchOutput(OutputObject: TIBBatchOutput);  //экспорт данных
```

Единственный параметр процедур – это ссылка на специализированный файловый объект – потомок класса TIBBatch.

На рис. 40.3 представлено дерево наследования класса TIBBatch. Сразу за абстрактным классом TIBBatch иерархия разветвляется на два направления. Ветвь TIBBatchInput отвечает за классы, обеспечивающие импорт данных, а ветвь TIBBatchOutput содержит классы поддержки экспорта. Каждая ветвь способна работать с двумя форматами файлов:

- размеченный файл (delimited file);
- двоичный файл (raw file).

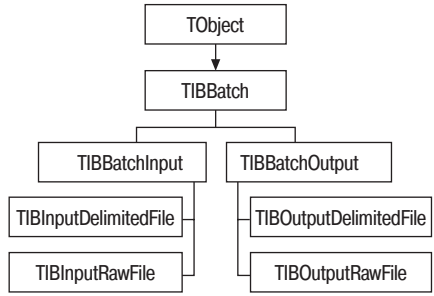


Рис. 40.3. *Потомки TIBBatch*

В первом случае (классы TIBInputDelimitedFile и TIBOutputDelimitedFile) программист может определить символы-разделители строк и колонок в свойствах:

```
property RowDelimiter: string;  //разделитель строк
property ColDelimiter: string;  //разделитель колонок
```

В случае файлов, инкапсулируемых классами TIBOutputRawFile и TIBInputRawFile, такая возможность отсутствует.

Основное свойство всех классов – имя файла:

```
property Filename: string;
```

Ниже представлен листинг, демонстрирующий способ сброса данных из таблицы в размеченный файл.

```
var OutFile : TIBOutputDelimitedFile;
begin
  frmDataModule.IBSQL1.SQL.Clear;
  frmDataModule.IBSQL1.SQL.Add('SELECT streetkey, streetname FROM streets');
  OutFile:= TIBOutputDelimitedFile.Create;
  try
    OutFile.FileName := 'c:\IBFile.txt';
    OutFile.RowDelimiter:=#13; //строки разделяем символом ввода
    OutFile.ColDelimiter:=#9;  //колонки разделяем символом табуляции
    frmDataModule.IBSQL1.BatchOutput(OutFile);
  finally
    OutFile.Free;
  end;
end;
```

В результате мы сможем просмотреть данные из файла в обычном блокноте. А вот пример кода загрузки данных из двоичного (неразмеченного) файла:

```
var RawFile : TIBInputRawFile;
begin
frmDataModule.IBSQL2.SQL.Clear;
```

```
frmDataModule.IBSQL2.SQL.Add('INSERT INTO streets (streetkey, streetname)
                              VALUES(:streetkey, :streetname)');
RawFile:= TIBInputRawFile.Create;
try
  RawFile.FileName := 'c:\IBFile.raw';
  frmDataModule.IBSQL2.BatchInput(RawFile);
finally
  RawFile.Free;
end;
end;
```

Характеристики наборов данных InterBase – компонент TIBDataSet

Предтеча всех наборов данных InterBase, класс TIBCustomDataSet, описан в модуле IBCustomDataSet. В нем сконцентрированы инженерные решения по обслуживанию баз данных InterBase из клиентского приложения.

На странице InterBase палитры компонентов Delphi находятся три элемента управления, предназначенные для обслуживания данных в классическом для Borland стиле, основанном на идеях, заложенных в универсальном наборе данных TDataSet. Это запрос TIBQuery, хранимая процедура TIBStoredProc и таблица TIBTable. Все перечисленные компоненты построены на основе общего предка – класса TIBCustomDataSet (рис. 40.4). Но запрос, хранимая процедура и таблица – лишь дань традициям, появившимся в первых версиях Delphi. Ключевым элементом управления InterBase по праву считается элемент управления TIBDataSet. Это мастер на все руки, обладающий возможностями всех своих коллег.

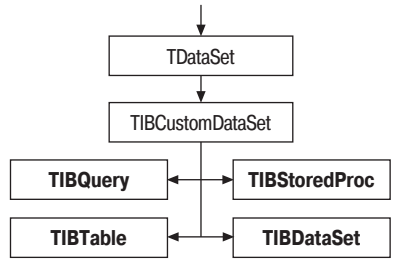


Рис. 40.4. *Компоненты для обслуживания наборов данных InterBase*

Возможности компонента TIBDataSet *практически перекрывают функциональный набор его коллег* TIBTable, TIBQuery *и* TIBStoredProc.

Подготовка к работе

Жизнедеятельность набора данных начинается с момента подключения к компоненту TIBDatabase. Для этого предназначено свойство:

```
property Database: TIBDatabase;
```

Второй взаимодействующий элемент управления – TIBTransaction. Связь с ним определяется в свойстве:

```
property Transaction: TIBTransaction;
```

Задача элемента управления TIBTransaction – организовать обмен между клиентским приложением и БД в границах отдельной транзакции.

Если внимательно изучить исходный код класса TIBCustomDataSet, то выяснится его главный секрет: класс инкапсулирует пять экземпляров TIBSQL:

```
TIBCustomDataSet = class(TDataset)
private
...
FQDelete, FQInsert, FQRefresh, FQSelect, FQModify : TIBSQL;
```

Каждый из экземпляров TIBSQL отвечает за решение специфических вопросов: удаление, вставка, обновление, выбор и модификация данных. Доступ к инкапсулированным объектам обеспечивают свойства:

```
property QDelete: TIBSQL;
property QInsert: TIBSQL;
property QRefresh: TIBSQL;
property QSelect: TIBSQL;
property QModify: TIBSQL;
```

Соответствующие инструкции SQL передаются в свойства:

```
property DeleteSQL: TStrings;  //удаление
property InsertSQL: TStrings;  //вставка
property RefreshSQL: TStrings; //обновление
property SelectSQL: TStrings;  //выбор
property ModifySQL: TStrings;  //модификация
```

Если запрос на выборку данных SelectSQL содержит параметры, то доступ к ним производится из свойства:

```
property Params: TIBXSQLDA;
```

Для подготовки всех инструкций вызовите метод:

```
procedure Prepare;
```

Противоположная цель достигается посредством обращения к методу:

```
procedure UnPrepare;
```

Ниже предложен пример инициализации элемента управления IBDataSet1:TIBDataSet для работы с таблицей "Streets":

```
with IBDataSet1 do
begin
Database:=IBDatabase1;
Transaction:=IBTransaction1;
SelectSQL.Text:='SELECT * FROM Streets ORDER BY StreetName';
RefreshSQL.Text:=SelectSQL.Text;
InsertSQL.Text:='INSERT INTO Streets (StreetName,StreetKey)
                 VALUES (:StreetName,:StreetKey)';
ModifySQL.Text:='UPDATE Streets SET StreetName=:StreetName WHERE
                 StreetKey=:Old_StreetKey';
DeleteSQL.Text:='DELETE FROM Streets WHERE (StreetKey=:Old_StreetKey)';
Prepare;
Open;
end;
```

Обратите внимание на особенность определения имен параметров при описании инструкций SQL:

1. Параметр, следующий за предложением `WHERE`*, называется* `OLD_+ имя_поля`.

2. Имена всех остальных параметров повторяют имена обслуживаемых полей.

Всегда следуйте этому соглашению.

Обработка событий

Помимо классического (унаследованного от `TDataSet`) набора событий потомки класса `TIBCustomDataSet` владеют специфичным для них перечнем событий (табл. 40.3).

Таблица 40.3. События класса TIBCustomDataSet

Событие	Описание
`property BeforeTransactionEnd: TNotifyEvent;`	Вызывается перед окончанием транзакции.
`property AfterTransactionEnd: TNotifyEvent;`	Вызывается по завершении транзакции.
`property BeforeDatabaseDisconnect: TNotifyEvent;`	Генерируется перед разрывом соединения со связанной БД.
`property AfterDatabaseDisconnect: TNotifyEvent;`	Генерируется после разрыва соединения со связанной БД.
`property TransactionFree: TNotifyEvent;`	Освобождение связанного с набором данных экземпляра транзакции (ссылка на объект находится в свойстве `Transaction`).
`property DatabaseFree: TNotifyEvent;`	Освобождение связанного с набором данных экземпляра БД (свойство `Database`).
`property OnUpdateRecord: TIBUpdateRecordEvent;`	Сохранение записи.
`property OnUpdateError: TIBUpdateErrorEvent;`	Ошибка при сохранении записи.

События `OnUpdateRecord()` *и* `OnUpdateError()` *более подробно рассмотрены в разделе «Работа в режиме кэширования обновлений» этой главы.*

Генератор значений

К одной из особенностей InterBase стоит отнести наличие в СУБД специальных объектов, называемых генераторами. Эти объекты предназначены для генерации числовых значений с целью их последующей вставки в поля таблиц, например в поля первичных ключей.

Генераторы обслуживаются инструкциями ISQL: `CREATE GENERATOR`*,* `SET GENERATOR` *и встроенной функцией* `GEN_ID()`*. Объекты-генераторы и соответствующие им инструкции InterBase SQL не описаны в классическом стандарте SQL-92 и являются новацией СУБД InterBase.*

При сохранении данных наследники TIBCustomDataSet способны автоматически присваивать значения для одного или нескольких полей обслуживаемого набора данных. Для этого предназначено свойство:

```
property GeneratorField: TIBGeneratorField;
```

Обращение к свойству во время разработки проекта приводит к вызову диалога настройки поля (рис. 40.5). Здесь программист должен выбрать:

1. Имя генератора (Generator).
2. Имя обслуживаемого поля таблицы (Filed).
3. Значение приращения (Increment By).
4. Событие, по которому новое значение будет передаваться в поле. Здесь предусмотрены три варианта: в момент вставки новой записи (On New Record), в момент сохранения (On Post) и поведение, определяемое сервером (On Server).

Рис. 40.5. *Диалог настройки свойства GeneratorField*

Редактирование данных

Компонент содержит доступные для редактирования данные, если свойство:

```
property CanModify: Boolean;  //только для чтения
```

возвращает значение true. Более полная информация предоставляется свойством:

```
property LiveMode: TLiveModes; //только для чтения
```

Это множество, способное хранить до 4 флагов:

```
TLiveModes = set of TLiveMode;
type TLiveMode = (lmInsert,  {разрешены операции вставки}
                  lmModify,  {разрешены операции модификации данных}
                  lmDelete,  {разрешено удаление}
                   lmRefresh); {разрешено обновление}
```

Если свойство пусто, то это свидетельствует о том, что набор данных доступен только для просмотра.

Для того чтобы всякий раз после сохранения изменений в данных компонент обновлял набор, установите в true свойство:

```
property ForcedRefresh: Boolean; //по умолчанию false
```

Работа в режиме кэширования обновлений

Для включения механизма кэширования установите в состояние true свойство:

```
property CachedUpdates: Boolean;
```

С этого момента все изменения в наборе данных начнут накапливаться в локальном буфере клиентского компьютера. Если набор данных работает в режиме кэширования обновлений, то благодаря свойству:

```
property UpdateRecordTypes: TIBUpdateRecordTypes;
type TIBUpdateRecordTypes = set of TCachedUpdateStatus;
```

мы получаем возможность определять, какие из находящихся в буфере записей будут видимы в наборе данных. Это своего рода фильтр, имеющий пять режимов отбора данных (табл. 40.4).

Таблица 40.4. Возможные значения TCachedUpdateStatus

Значение	Описание
cusModified	Модифицированные записи.
cusInserted	Вставленные записи.
cusDeleted	Удаленные записи.
cusUnmodified	Неизменявшиеся записи.
cusUninserted	Невставленные записи.

По умолчанию в состав UpdateRecordTypes множества включены флаги cusModified, cusInserted и cusUnmodified. Манипулируя элементами множества, мы пополним наш проект дополнительными сервисными возможностями, например позволим пользователю восстановить все удаленные записи из кэша:

```
procedure RestoreRecords;
var Old_URT : TIBUpdateRecordTypes;
begin
  with IBDataSet1 do
  begin
  Old_URT:=UpdateRecordTypes;
  UpdateRecordTypes := [cusDeleted];
    First;
    while EOF=False do
      begin
      Undelete;
      Next;
      end;

      UpdateRecordTypes := Old_URT;
  end;
end;
```

В примере мы воспользовались новым методом, восстанавливающим запись, помеченную на удаление:

```
procedure Undelete;
```

Если в буфере компонента содержатся модифицированные записи, ожидающие своего сохранения в БД, то свойство:

```
property UpdatesPending: Boolean;
```

вернет значение true. В этом случае для сохранения изменений в БД обратитесь к методу:

```
procedure ApplyUpdates;
```

В момент сохранения изменений в данных возникает событие:

```
property OnUpdateRecord: TIBUpdateRecordEvent;
TIBUpdateRecordEvent = procedure(DataSet: TDataSet; UpdateKind: TUpdateKind;
                                 var UpdateAction: TIBUpdateAction) of object;
```

Это событие генерируется для каждой записи из кэша. Параметр UpdateKind

```
type TUpdateKind = (ukModify, ukInsert, ukDelete);
```

указывает, какая именно операция (модификация, вставка или удаление) осуществляется. Переменная UpdateAction позволит программисту управлять развитием события (табл. 40.5).

Таблица 40.5. Возможные значения TIBUpdateAction

Значение	Описание
uaAbort	Прекращение обновления без генерации сообщения об ошибке.
uaApply	Для внутреннего пользования.
uaApplied	Сохранить изменения и очистить кэш.
uaFail	Прервать обновление и вызвать сообщение об ошибке.
uaRetry	Повторить операцию обновления.
uaSkip	Пропустить текущую запись с ее сохранением в кэше.

Если во время сохранения данных из локального кэша в БД возникнет ошибка, то генерируется событие:

```
property OnUpdateError: TIBUpdateErrorEvent;
TIBUpdateErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError;
            UpdateKind: TUpdateKind; var UpdateAction: TIBUpdateAction)of object;
```

Практически все параметры события повторяют параметры события OnUpdateRecord(), за исключением параметра E, который представляет собой указатель на возникшую исключительную ситуацию.

Для очистки кэша без отправки пакета изменений в БД вызовите процедуру:

```
procedure CancelUpdates;
```

Если же требуется вернуть в исходное состояние лишь отдельную запись в наборе, то лучше воспользоваться методом:

```
procedure RevertRecord;
```

Редактирование наборов данных, доступных в режиме «только для чтения»

Зачастую возвращаемый потомками TIBCustomDataSet набор данных доступен только для просмотра. Если логика работы приложения требует обеспечить возможность модификации таких данных, то на помощь приходит свойство:

```
property UpdateObject: TIBDataSetUpdateObject;
```

Это ссылка на потомка абстрактного класса TIBDataSetUpdateObject – элемент управления TIBUpdateSQL.

Перемещение по записям

Наборы данных InterBase владеют всеми классическими методами перемещения по записям (First, Prior, Next и Last). Кроме того, если набор данных поддерживает порядковые номера записей, а об этом свидетельствует значение true, возвращаемое функцией:

```
function IsSequenced: Boolean;
```

то по набору разрешено двигаться, передавая порядковый номер требуемой строки в свойство:

```
property RecNo: Integer;
```

Если свойство UniDirectional возвращает значение true, это означает, что курсор может перемещаться только в одном направлении – от первой записи к последней.

```
property UniDirectional: Boolean;
```

Фильтрация данных

Изучая универсальный набор данных TDataSet, мы познакомились с ключевыми свойствами и методами компонентов-наборов данных, предназначенных для фильтрации данных. В первую очередь это свойство Filter, определяющее условия фильтрации (см. табл. 33.4 в главе 33). Опции фильтра задаются в свойстве FilterOptions. Для включения/отключения фильтра обратитесь к свойству Filtered.

Запрос – компонент TIBQuery

Запрос TIBQuery построен на базе рассмотренного ранее TIBCustomDataSet и описан в модуле IBQuery. Основное назначение компонента – организация доступа к данным одной или нескольких таблиц. Как и у всех подобных элементов управления, текст запроса описывается в свойстве SQL, а отправка запроса реализуется методами Open() или ExecSQL().

Возвращаемый набор данных доступен только в режиме «для чтения». Чтобы добиться возможности его редактирования, нам понадобятся услуги TIBUpdateSQL, подключаемого к свойству UpdateObject.

Хранимая процедура – компонент TIBStoredProc

Хранимая процедура TIBStoredProc – прямой наследник класса TIBCustomDataSet. Компонент реализован в модуле IBStoredProc. Элемент управления предназначен для обращения к хранящимся в сервере БД хранимым процедурам. Название требуемой процедуры указывается в свойстве StoredProcName, ее параметры – в массиве Params. Для вызова процедуры потребуется метод ExecProc().

Таблица – компонент TIBTable

Таблица – прямой потомок класса TIBCustomDataSet. Класс описан в модуле IBTable. Компонент предназначен для просмотра и редактирования отдель-

ной таблицы БД. Для подключения к таблице необходимо указать ее название в свойстве TableName. Все свойства и методы класса подробно рассмотрены в главе 33.

При работе с компонентом TIBTable стоит определиться, какие типы таблиц будут доступны в свойстве TableName. Для этого предназначено свойство:

```
property TableTypes: TIBTableTypes;
TIBTableTypes = set of TIBTableType;
type TIBTableType = (ttSystem, ttView);
```

Если в состав элементов множества входит значение ttSystem, то это свидетельствует о том, что нам будут доступны системные таблицы и представления. Наличие элемента ttView – только представления, определенные пользователем.

Перед открытием набора данных стоит убедиться в наличии таблицы с именем TableName. Для этого предназначено свойство:

```
property Exists: Boolean;
```

Во всем остальном наш компонент схож с классическим TTable.

Транзакция – компонент TIBTransaction

Компонент TIBTransaction предоставляет услуги по управлению транзакциями из клиентского приложения с использованием одного или нескольких соединений с БД. В услугах транзакций нуждаются все потомки класса TIBCustomDataSet и компонент TIBSQL.

Перечень всех обслуживаемых соединений с БД мы обнаружим в свойстве:

```
property Databases[Index: Integer]: TIBDatabase;
```

Сколько всего экземпляров TIBDatabase связано с транзакцией, можно прочитать, обратившись к свойству:

```
property DatabaseCount: Integer;
```

База данных по умолчанию назначается в свойстве:

```
property DefaultDatabase: TIBDatabase;
```

Для программного пополнения списка обслуживаемых соединений с БД используется метод:

```
function AddDatabase(db: TIBDatabase): Integer;
```

Для удаления отдельного экземпляра TIBDatabase из списка Databases вызывается процедура:

```
procedure RemoveDatabase(Idx: Integer);
```

Метод требует указать индекс исключаемого компонента. Если этот индекс вам неизвестен, воспользуйтесь методом:

```
function FindDatabase (db: TIBDatabase): Integer;
```

Функция просмотрит весь список Databases с целью поиска в нем интересующего нас экземпляра TIBDatabase и возвратит его порядковый номер в списке. Для полной очистки списка понадобится процедура:

```
procedure RemoveDatabases;
```

Количество активных наборов данных InterBase и экземпляров TIBSQL, обслуживаемых базой данных TIBDatabase, мы найдем в свойстве:

```
property SQLObjectCount: Integer;            //только для чтения
```

Все они доступны благодаря свойству:

```
property SQLObjects[Index: Integer]: TIBBase; //только для чтения
```

Управление транзакцией

Об активности транзакции судят по свойству:

```
property Active: Boolean;
```

Запуск новой транзакции инициируется процедурой:

```
procedure StartTransaction;
```

При нормальном завершении транзакции вызываем процедуру:

```
procedure Commit;
```

фиксирующую изменения в БД. В противном случае для отката транзакции обращаемся к методу:

```
procedure Rollback;
```

Например:

```
procedure TMainForm.AddNewRecord;
begin
  IBDatabase1.Open;
  try
  IBTransaction1.StartTransaction;
      with Table1 do
      begin
      Insert;
      FieldByName('City').Value := ComboBox1.Text;
      Post;
      end;
  IBTransaction1.Commit;
  except
  IBTransaction1.Rollback;
  end;
end;
```

Если мы работаем с Interbase не ниже 6 версии, то для отката всех изменений в данных, осуществленных в рамках текущей транзакции, разрешено воспользоваться процедурой:

```
procedure RollbackRetaining;
```

При этом сохраняется контекст транзакции, т. е. транзакция не будет завершена. Для того чтобы зафиксировать все изменения в данных, сделанные в рамках текущей транзакции, с сохранением ее контекста, используйте метод:

```
procedure CommitRetaining;
```

Для проверки, выполняется ли в данный момент транзакция, обращаемся к свойству:

```
property InTransaction: Boolean; //только для чтения
```

Значение `true` свидетельствует, что транзакция еще не завершена.

Компонент `TIBTransaction` достаточно умен для того, чтобы самостоятельно завершить выполнение транзакции. Степень доверия программиста этому элементу управления определяется в свойстве:

```
property AutoStopAction : TAutoStopAction; //по умолчанию saNone
```

Варианты допустимых значений представлены в табл. 40.6.

Таблица 40.6. Возможные значения TAutoStopAction

Значение	Описание
saNone	Компонент не может самостоятельно управлять завершением транзакций.
saRollback	Компонент имеет право на самостоятельность при откате транзакции.
saCommit	Компоненту разрешено фиксировать транзакцию.
saRollback-Retaining	Поддерживается, начиная с Interbase 6.0. Транзакция не завершается, пока есть незакрытые наборы данных, но все изменения в связанных с транзакцией данных отменяются. Другими словами, производится откат транзакции с сохранением ее контекста.
saCommit-Retaining	Транзакция не завершается, пока есть незакрытые наборы данных, но все изменения в связанных с транзакцией данных фиксируются.

Свойство `AutoStopAction` *работает только после закрытия набора данных (наследника* `TDataSet`*). Если с транзакцией связан компонент* `TIBSQL`*, это свойство должно быть установлено в* `saNone`.

Тайм-аут транзакции

Время тайм-аута определяется в свойстве:

```
property IdleTimer: Integer;
```

По истечении установленного времени поведение компонента зависит от состояния свойства:

```
property DefaultAction: TTransactionAction;
```

Это операция по умолчанию; возможные варианты представлены в табл. 40.7.

Таблица 40.7. Варианты действия по истечении тайм-аута TTransactionAction

Значение	Описание
TARollback	Откат транзакции.
TACommit	Фиксация изменений.
TARollbackRetaining	Поддерживается, начиная с Interbase 6.0. Откат транзакции с удержанием ее контекста.
TACommitRetaining	Фиксация транзакции с сохранением ее контекста.

Кроме того, по завершении времени тайм-аута генерируется событие:

```
property OnIdleTimer: TNotifyEvent;
```

Параметры транзакции

Прикладной интерфейс программирования InterBase позволяет наделять транзакции своими параметрами. Доступ из кода приложения к содержащему параметры буферу обеспечивает свойство Params или TPB:

```
property Params: TStrings;
property TPB: PChar; //только для чтения
```

О размере буфера нас проинформирует свойство:

```
property TPBLength: Short;
```

Модифицируемый запрос InterBase – компонент TIBUpdateSQL

Компонент TIBUpdateSQL (рис. 40.6) приходит на помощь в тех случаях, когда возникает острая необходимость обеспечить возможность редактирования данных в наборах, не доступных для модификации. Например, такая ситуация вполне вероятна, когда компонент-потомок TIBCustomDataSet (TIBQuery, TIBStoredProc) возвращает данные от двух и более таблиц одновременно. Компонент TIBUpdateSQL описан в модуле IBUpdateSQL.

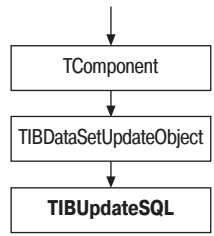


***Рис. 40.6.** Иерархия TIBUpdateSQL*

Заложенная в классе TIBUpdateSQL идея не нова и уже хорошо нам знакома. Вспомните описанный в главе 35 «Применение механизма BDE для доступа к данным» элемент управления TUpdateSQL.

Ссылка на обслуживаемый набор данных хранится в свойстве:

```
property DataSet: TIBCustomDataSet;
```

Для подключения модифицируемого запроса TIBUpdateSQL *к набору данных InterBase у потомков* TIBCustomDataSet *объявлено свойство* UpdateObject.

Внутри элемента TIBUpdateSQL скрываются три экземпляра класса TIBQuery, каждый из которых отвечает за решение отдельной задачи – вставки, редак-

тирования или удаления данных. Для того чтобы обратиться к требуемому экземпляру TIBQuery, можно воспользоваться свойством:

```
property Query[UpdateKind: TUpdateKind]: TIBQuery;
type TUpdateKind = (ukModify, ukInsert, ukDelete);
```

Для определения соответствующих инструкций SQL обращаемся к свойству:

```
property SQL[UpdateKind: TUpdateKind]: TStrings;
```

или к тройке свойств:

```
property InsertSQL: TStrings;
property ModifySQL: TStrings;
property DeleteSQL: TStrings;
```

Например:

```
with IBUpdateSQL do
begin
InsertSQL.Add('INSERT INTO Streets (StreetName) VALUES (:StreetName)');
ModifySQL.Add('UPDATE Streets SET StreetName=:StreetName WHERE
StreetKey=:Old_StreetKey');
DeleteSQL.Add('DELETE FROM Streets WHERE (StreetKey=:Old_StreetKey)');
end;
```

Сделаю очень важное замечание. При объявлении имен параметров программист обязан соблюдать следующие правила:

1. *В тексте команды* INSERT *имена параметров должны совпадать с физическими именами полей.*
2. *В командах* UPDATE *и* DELETE *(после ключевого слова* WHERE*) для обозначения в тексте запроса старых значений полей применяется префикс* OLD_*. Причина этого в том, что поиск записей для модификации или удаления производится по старым значениям.*

Перед запуском той или иной инструкции SQL на выполнение необходимо установить параметры запроса. Для этого предназначен метод:

```
procedure SetParams(UpdateKind: TUpdateKind);
```

И только после этого воспользуемся методом:

```
procedure ExecSQL(UpdateKind: TUpdateKind);
```

Чтобы сэкономить силы, вместо последовательного обращения к методам SetParams() и ExecSQL() вызываем процедуру:

```
procedure Apply(UpdateKind: TUpdateKind);
```

Прямой доступ к инструкции SQL, позволяющей обновить результирующий набор, обеспечивает свойство:

```
property RefreshSQL: TIBSQL;
```

Информация об объектах БД – компонент TIBExtract

Напомню о способности компонента TIBDatabase получать список таблиц БД и перечень входящих в них полей. Еще большим спектром возможностей об-

ладает элемент управления TIBExtract. Прямой наследник класса TComponent, компонент TIBExtract, описан в модуле IBExtract. Элемент управления сумеет выведать у СУБД такие метаданные, как перечень индексов, хранимых процедур и генераторов, т. е. список всех имеющихся в БД объектов.

Для организации работы с компонентом программист должен подключить TIBDatabase к базе данных (свойство Database) и соединиться с транзакцией (Transaction), а далее наступает черед ключевого метода:

```
procedure ExtractObject(ObjectType : TExtractObjectTypes;
                        ObjectName : string = ''; ExtractTypes : TExtractTypes = []);
```

Здесь ObjectType – запрашиваемый у БД тип метаданных (табл. 40.8). Необязательный параметр ObjectName указывает имя конкретного объекта БД. Необязательный параметр ExtractTypes позволяет наложить дополнительный фильтр на тип извлекаемых данных (табл. 40.9).

Таблица 40.8. Перечень значений TExtractObjectTypes

Значение	Описание
eoDatabase	Список всех определенных в БД объектов. Этот перечень может быть дополнительно отфильтрован при помощи параметров ObjectName и ExtractTypes.
eoDomain	Список всех доменов.
eoTable	Если параметр ObjectName не использовался, возвращается список всех таблиц, иначе мы будем проинформированы о таблице с именем ObjectName, причем тип информации должен быть определен в параметре ExtractTypes.
eoView	Если параметр ObjectName не использовался, возвращается список всех представлений, иначе мы будем проинформированы о представлении с именем ObjectName.
eoProcedure	Список хранимых процедур.
eoFunction	Список всех внешних функций. Если параметр ObjectName не пуст, то мы получим информацию об отдельной функции.
eoGenerator	Список генераторов.
eoException	Список исключений.
eoBLOBFilter	Список фильтров BLOB.
eoRole	Список правил.
eoTrigger	Перечень триггеров.
eoForeign	Перечень внешних ключей.
eoIndexes	Список индексов.
eoChecks	Ограничения.
eoData	Список всех данных с именем, определенным в параметре ObjectName.

Таблица 40.9. Описание TExtractTypes

Значение	Описание
etDomain	При выборке метаданных о таблице включить информацию обо всех принадлежащих таблице доменах.
etTable	Собрать данные только о таблице.
etRole	При выборке метаданных о таблице включить информацию обо всех связанных с таблицей правилах.
etTrigger	При выборке метаданных о таблице собирать данные об ее триггерах.
etForeign	При выборке метаданных о таблице собирать данные об ее внешних ключах.
etIndex	При выборке метаданных о таблице собирать данные об ее индексах.
etData	При выборке метаданных о таблице получить ее данные.
etGrant	При выборке метаданных о таблице включать информацию о правах пользователей.
etCheck	При выборке метаданных о таблице собирать данные об ограничениях.
etAlterProc	При сборе данных о хранимых процедурах исключить информацию о взаимных ссылках между процедурами.

Результат выполнения метода ExtractObject() помещается в свойство:

```
property Items: TStrings;
```

События InterBase – компонент TIBEvents

Благодаря компоненту TIBEvents клиентское приложение получает возможность обрабатывать события СУБД InterBase. Это наследник класса Tcomponent, описанный в модуле IBEvents. Объект TIBEvents умеет отслеживать возникающие на сервере события. Перед этим события должны быть определены разработчиком БД с помощью имеющихся в распоряжении ISQL-инструкций EVENT INIT и EVENT WAIT.

Прежде всего необходимо подключить компонент к базе данных, воспользовавшись свойством Database. Затем в свойстве:

```
property Events: TStrings;
```

соблюдая регистр символов, описываем перечень интересующих нас событий. Заполнив список событий, переводим в true свойство:

```
property Registered: Boolean;
```

или вызываем метод:

```
procedure RegisterEvents;
```

Таким образом, приложение укажет серверу, что его необходимо уведомлять об этих событиях. С этого момента при возникновении на сервере зарегистрированного события на клиентской стороне вызывается процедура:

```
property OnEventAlert: TEventAlert;
type TEventAlert = procedure( Sender: TObject; EventName: string;
EventCount: longint; var CancelAlerts: Boolean);
```

Здесь EventName – имя события, EventCount информирует об общем количестве данных событий с момента последнего вызова OnEventAlert(). Передав в переменную CancelAlerts значение true, мы укажем серверу, что больше не нуждаемся в информации об его событиях. Для возобновления процесса информирования вновь установите в true свойство Registered.

Информация о БД – компонент TIBDatabaseInfo

При разработке административных модулей хорошую службу сослужит компонент TIBDatabaseInfo. Это наследник класса Tcomponent, описанный в модуле IBDatabaseInfo. Подключившись к базе данных (с помощью свойства Database) компонент сообщит нам основные характеристики СУБД Interbase (табл. 40.10).

Таблица 40.10. Основные свойства компонента IBDatabaseInfo

Свойство	Описание
property Allocation: Long;	Количество распределенных страниц БД.
property BackoutCount: TStringList;	Возвратит число удаленных записей.
property BaseLevel: Long;	Второй байт числа содержит версию БД.
property CurrentMemory: Long;	Объем (в байтах) используемой СУБД памяти.
property DBFileName: string;	Название файла базы данных.
property DBIinplementationClass : Long;	Номер класса описания (значение в диапазоне от 1 до 12).
property DBImplementationNo: Long;	Номер описания.
property DBSiteName: string;	Имя сайта БД.
property DBSQLDialect: Long;	Диалект SQL.
property DeleteCount: TStringList;	Возвращает количество удалений; отсчет начинается с момента последнего обновления БД.
property ExpungeCount: TStringList;	Количество удаленных записей с момента последнего сохранения БД.
property Fetches: Long;	Количество чтений из буфера памяти.
property ForcedWrites: Long;	Режим операций: 0 – асинхронный, 1 – синхронный.
property InsertCount: TStringList;	Количество вставок данных в БД с момента последнего сохранения.
property Marks: Long;	Количество выполненных записей в кэш.
property MaxMemory: Long;	Максимальный размер памяти (в байтах), занимаемый БД с момента последнего сохранения.

Таблица 40.10 (продолжение)

Свойство	Описание
`property NoReserve: Long;`	Поддержка резервирования страниц: 0 – да, 1 – нет.
`property NumBuffers: Long;`	Число выделенных буферов.
`property ODSMajorVersion: Long;`	Версия ODS, старшее значение.
`property ODSMinorVersion: Long;`	Версия ODS, младшее значение.
`property PageSize: Long;`	Размер страницы.
`property PurgeCount: TStringList;`	Количество удаленных записей.
`property ReadIdxCount: TStringList;`	Количество чтений через индексы с момента последнего сохранения.
`property Readonly: Long;`	Доступ только для чтения: 0 – да, 1 – нет.
`property Reads: Long;`	Количество чтений из БД.
`property ReadSeqCount: TStringList;`	Количество последовательных просмотров таблицы.
`property SweepInterval: Long;`	Количество зафиксированных транзакций.
`property UpdateCount: TStringList;`	Количество обновлений БД с момента последнего сохранения.
`property UserNames: TStringList;`	Список активных пользователей.
`property Version: String;`	Версия базы данных.
`property Writes: Long;`	Количество постраничных записей.

Резюме

Компоненты InterBase специализируются исключительно на работе с фирменной СУБД компании Borland – InterBase. Палитра компонентов современной Delphi 2005 включает почти два десятка компонентов, нацеленных на обслуживание InterBase. Это весьма неприхотливая к ресурсам компьютера, эффективная, соответствующая стандарту SQL-92 СУБД. При разработке БД малой и средней степени сложности сервер InterBase способен составить конкуренцию таким лидерам в области реляционных баз данных, как Oracle, Informix и Microsoft SQL Server.

Заключение

Все, что мы успели обсудить на страницах этой книги, – всего лишь надводная часть айсберга. К сожалению, огромная часть информации так и осталась скрытой в глубине. Поэтому в завершение книги позволю себе дать ряд простых советов, которые могут пригодиться как в процессе программирования, так и для продолжения изучения Delphi.

В самостоятельной работе нашими лучшими помощниками станут: справочная система Delphi, SDK (Software Development Kit) Windows, листинги исходных модулей библиотеки VCL, система знаний MSDN. Это огромный кладезь ценнейшей официальной информации. Ни при каких условиях не пренебрегайте ими.

О популярности программного обеспечения программных продуктов Borland свидетельствует внушительное количество разнообразных сайтов во Всемирной паутине. Пальму первенства безусловно держит главный сайт компании Borland: *http://www.borland.com* и *http://www.borland.com/delphi_net*. Для русскоязычных программистов значительный интерес могут представлять ресурсы: *http://www.delphikingdom.ru*, *http://www.delphimaster.ru*, *http://delphid.dax.ru*, *http://www.vcl.ru*. Регулярно посещайте два-три сайта, подпишитесь на рассылку новостей, изучайте разделы FAQ. Короче говоря, будьте всегда в центре событий.

Среда программирования Delphi – это динамически развивающийся и непрерывно совершенствующийся продукт. В каждую новую версию, в каждый компонент вкладывается кропотливый труд сотен программистов компании Borland и энтузиастов-одиночек. Идеи талантливых инженеров воплощаются в программный код, в эффективные алгоритмы, в новые технологии и элементы управления. Поэтому никогда не «зацикливайтесь» на устаревших версиях ПО, будьте на шаг впереди своих конкурентов, не бойтесь осваивать новое. Но при этом никогда не спешите ставить новинку на вооружение без тщательного анализа ее достоинств и недостатков. Здоровый консерватизм еще никому не повредил.

Постоянно изучайте операционную систему. Ведь именно она, а не Delphi, служит фундаментом любого ПО.

Еще одним весьма важным аспектом вашего профессионального совершенствования должно стать общение с другими программистами. Это и обсуждение разрабатываемого ПО, и обмен опытом, и совместная работа над решением задач, и поиск ошибок. Кроме того, это именно тот случай, когда не грех похвастаться результатами своих трудов, тем более, что наиболее «благожелательные» друзья немедленно найдут в вашей программе полсотни недостатков и всласть покритикуют, казалось бы, безупречное творение. Воспринимайте это с пониманием (а если можете, то и с благодарностью), ведь для неглупого человека критика – это серьезный стимул к дальнейшему совершенствованию.

Литература

1. Справочная система Delphi.
2. Справочная система Microsoft Developers Network (MSDN Library). *http://msdn.microsoft.com*.
3. Jeffrey Richter «Programming Applications for Microsoft Windows», Fourth edition. – Microsoft Press, 1999.
4. David Sceppa «Programming ADO». – Microsoft Press, 2000.
5. Фролов А. В., Фролов Г. В. «Программирование для Windows NT», часть 1, том 26. – М.: Диалог-МИФИ, 1996. – 161 с.
6. Фролов А. В., Фролов Г. В. «Программирование для Windows NT», часть 2, том 27. – М.: Диалог-МИФИ, 1996. – 141 с.
7. Петзольд Ч. «Программирование для Windows 95», в двух томах. – Пер. с англ. – СПб.: BHV – Санкт-Петербург, 1997. – Том 1, 752 с., ил. Том 2, 368 с., ил.
8. Драхвелидзе П., Марков Е. «Delphi – среда визуального программирования». – СПб.: BHV – Санкт-Петербург, 1996. – 352 с.
9. Драхвелидзе П., Марков Е. «Delphi 4». – СПб.: БХВ – Санкт-Петербург, 1999. – 816 с., ил.
10. Сигнор Р., Стегман М. «Использование ODBC для доступа к базам данных». – Пер. с англ. – М.: Бином, Научная книга, 1995. – 384 с., ил.
11. Киммел П. и др. «Borland C++ 5». – Пер. с англ. – СПб.: BHV – Санкт-Петербург, 2000. – 976 с., ил.
12. Ковязин А., Востриков С. «Мир InterBase. Архитектура, администрирование и разработка приложений баз данных в InterBase/Firebird/Yaffil». 2-е изд. – М.: КУДИЦ-ОБРАЗ, 2002. – 496 с.
13. Краснов М. «Графика в проектах Delphi». – СПб.: БХВ – Санкт-Петербург, 2000. – 352 с., ил.
14. Орлик С. В. «Секреты Delphi на примерах». – М.: БИНОМ, 1996. – 316 с.
15. Коннолли Т., Бегг К., Страчан А. «Базы данных: проектирование, реализация и сопровождение. Теория и практика», 2-е изд. – Пер. с англ. – М.: Издательский дом «Вильямс», 2001. – 1120 с.

16. Грабер М. «Справочное руководство по SQL». – Пер. с англ. – М.: Издательство «Лори», 1998. – 291 с.

17. Тейлор Д. и др. «Delphi 3: библиотека программиста». – Пер. с англ. – СПб.: Питер, 1998. – 560 с., ил.

18. Тейксейра С., Пачеко К. «Delphi 5. Руководство разработчика» в двух томах. – Пер. с англ. – М.: Издательский дом «Вильямс», 2000. – Том 1, 832 с., ил. Том 2, 992 с., ил.

19. Тихомиров Ю. В. «Microsoft® SQL Server 7.0.» – СПб.: БХВ – Санкт-Петербург, 1999. – 720 с., ил.

20. Винкоп С. «Использование Microsoft® SQL Server 7.0. Специальное издание». – Пер. с англ. – К.: М.: СПб.: Издательский дом «Вильямс», 2001. – 816 с., ил.

21. Пильщиков В. Н. «Сборник упражнений по языку Паскаль: Учеб. пособие для вузов». – М.: Наука. Гл. ред. физ.-мат. лит., 1989. – 160 с.

22. Саймон Р. «Windows 2000 API. Энциклопедия программиста». – Пер. с англ. – К.: СПб.: ООО «ДиаСофтЮП», 2002. – 1088 с.

23. Румянцев П. В. «Азбука программирования в Win 32 API», 2-е изд. – М.: Радио и связь, Горячая линия – Телеком, 1999. – 272 с., ил.

24. Мэтчо Д. и др. «Delphi 2. Руководство для профессионалов». – Пер. с англ. СПб.: BHV – Санкт-Петербург, 1997. – 784 с., ил.

25. Гордеев А. В., Молчанов А. Ю. «Системное программное обеспечение». – СПб.: Питер, 2002 – 736 с., ил.

26. Кастер. Х. «Основы Windows NT и NTFS». – Пер. с англ. – М.: Издательский отдел «Русская редакция», ТОО «Channel Trading Ltd.», 1996. – 440 с., ил.

27. «Сетевые средства Microsoft® Windows NT® Server 4.0». – Пер. с англ. – СПб.: БХВ – Петербург, 2000. – 880 с., ил.

28. Марко Канту, Тим Гуч при участии Джона Лэма «Delphi. Руководство разработчика». – Пер. с англ. – К.: ВЕК+, М.: ЭНТРОП, М.: ДЕСС, 1999. – 752 с., ил.

29. Грегори К. «Использование Visual C++ 6. Специальное издание». – Пер. с англ. – М.: СПб.: К.: Издательский дом «Вильямс», 2000. – 864 с., ил.

Алфавитный указатель

A–B

ANSI, 165
BDE
  компоненты, 864
  объекты, 863
BDE API
  ConstraintCallBack, 867

C

Clustered, 794
COM (Component Object Model), многокомпонентная модель объектов), 624
_Command
  CommandObject, 964
  CommandText, 964
  CommandTimeout, 966
  Execute, 965
  TCommandType, 965
COM-объект, 626
Constructor, 105, 420

D

DDE, 563
  TDDEClientConv, 564
  TDDEClientItem, 570
  TDDEServerItem, 572
  имя сервиса, 571
  предмет сессии, 571
  тема сессии, 571
Destructor, 105, 420
DLL (Dynamic Link Library), 478
  DllEntryPoint, 483
  FreeLibrary, 491
  FreeLibraryAndExitThread, 493
  GetProcAddress, 490
  LoadLibrary, 490
  LoadLibraryEx, 490
  библиотека ресурсов, 494
  неявная загрузка, 493
  точка входа, 484
  хранение форм, 488
  явная загрузка, 489
drag-and-dock, 144
drag-and-drop, 141

E

ER-диаграмма, 773
Event, 112, 125
Exception
  Create, 387
  CreateFmt, 388
  CreateFmtHelp, 388
  CreateHelp, 388
  CreateRes, 388
  CreateResFmt, 388
  CreateResFmtHelp, 388
  CreateResHelp, 388
  HelpContext, 387
  Message, 387

G–H

GDI, 239
GUI, 239
HWND, 133

I

IClassFactory
  CreateInstance, 633
  LockServer, 633
IEnumIDList, 743
  Clone, 744
  Next, 743
  Reset, 743
  Skip, 744
IP-адрес, 605
IUnknown
  _AddRef, 631
  QueryInterface, 631
  _Release, 631

M–N

mailslot (почтовый слот), 583
Microsoft Excel, 676
MM_ANIISOTROPIC, 280
MM_ISOTROPIC, 278
MM_TEXT, 274
mutex (мьютекс), 524
Network DDE, 590

O

OEM, 165
OOP, 102
OSI (Open System Interconnec, 582

P–R

PQueryServiceConfig, 536
_Recordset
  Recordset, 966
  RecordsetState, 966
  Supports, 967
RTTI, 115
RUSSIAN_CHARSET, 165

S

SCM (Service Control Manager), 529
SQL, 782
  Add, 791
  Alter, 791
  Alter Table, 791
  Between, 797
  Cascade, 793
  Check, 790
  Close, 808
  Collate, 789
  Commit, 806
  Create Database, 789
  Create Index, 793
  Create Procedure, 794

Create Schema, 789
Create Table, 789
Create Trigger, 795
Create View, 792
Cross Join, 801
Declare Cursor, 807
Default, 789
Delete, 804
Drop Database, 789
Drop Index, 794
Drop Procedure, 795
Drop Schema, 789
Drop Table, 791
Drop Trigger, 795
Drop View, 793
Escape, 797
Foreign Key, 790
Full Join, 802
Global, 791
Grant, 805
Group By, 800
Having, 800
In, 798
Inner Join, 802
Insert, 803
Is Not Null, 798
Join Using, 802
Left Join, 802
Like, 797
Local, 791
Natural Join, 801
NULL, 798
Open, 807
Order By, 796
Outer Join, 802
Primary Key, 790
References, 790
Revoke, 806
Right Join, 802
Rollback, 807
Select, 795
Set, 804
Temporary, 791
Union Join, 802
Unique, 790
Update, 804
Where, 796
функции обобщающие, 799

T

TAction
ActionList, 436
Category, 436
Execute, 434
Hint, 436
Index, 436
OnExecute, 434
OnHint, 436
OnUpdate, 435
TActionClient
ContextItems, 449
Items, 449
TActionClientItem, 448
CheckUnused, 450
ContextItems, 449
Items, 449
LastSession, 450
UsageCount, 450
TActionList
ActionCount, 437
Actions, 437
AddAction, 437
ExecuteAction, 437
Images, 438
OnChange, 438
OnExecute, 434, 438
OnStateChange, 438
OnUpdate, 438
RemoveAction, 437
State, 437
TActionMainMenuBar, 453
AnimationStyle, 453
CloseMenu, 454
ExpandDelay, 453
InMenuLoop, 454
OnControlCreated, 453
OnEnterMenuLoop, 454
OnGetControlClass, 452
OnGetPopupClass, 454
OnPopup, 454
RootMenu, 454
Shadows, 453
TActionManager
ActionBars, 441
ActionCount, 437
AddAction, 437
DefaultActionBars, 441
ExecuteAction, 437
Images, 438
OnChange, 438
OnExecute, 434, 438
OnStateChange, 438
OnUpdate, 438
PrioritySchedule, 446
RemoveAction, 437
State, 437
Style, 445
TActionToolBar
OnControlCreated, 453
OnGetControlClass, 452
TADOCommand
ExecuteOptions, 993
ParamCheck, 992
Parameters, 992
TADOConnection
AfterConnect, 971
AfterDisconnect, 973
Attributes, 978
BeforeConnect, 971
BeforeDisconnect, 973
BeginTrans, 976
Cancel, 971
CommandCount, 975
Commands, 975
CommitTrans, 977
Connected, 969
ConnectionString, 969
ConnectionTimeout, 971
ConnectOptions, 970
CursorLocation, 978
Execute, 975
GetFieldNames, 974
GetProcedureNames, 974
GetTableNames, 974
InTransaction, 976
IsolationLevel, 977
KeepConnection, 970
LoginPrompt, 969
Mode, 970
OnBeginTransComplete, 977
OnCommitTransComplete, 977
OnConnectComplete, 971
OnDisconnect, 973
OnExecuteComplete, 976
OnInfoMessage, 971
OnLogin, 971
OnRollbackTransComplete, 977
OnWillConnect, 971
OnWillExecute, 976
Open, 969
OpenSchema, 973
Provider, 969
RollbackTrans, 977
State, 973
Version, 973
TADODataSet
BlockReadSize, 988
CacheSize, 987

CancelBatch, 990
Command, 991
CommandText, 980
Connection, 980
CursorLocation, 980
CursorType, 989
DeleteRecords, 981
Filter, 987
Filtered, 987
FilterGroup, 990
FilterOptions, 987
GetIndexNames, 985
IndexFieldCount, 985
IndexFieldNames, 986
IndexFields, 985
IndexName, 985
LoadFromFile, 991
LockType, 986, 989
MarshalOptions, 991
MaxRecords, 987
OnEndOfRecordset, 985
OnFetchComplete, 983
OnFetchProgress, 983
OnFieldChangeComplete, 983
OnMoveComplete, 985
OnRecordChangeComplete, 984
OnRecordsetChangeComplete, 984
OnRecordsetCreate, 982
OnWillChangeField, 983
OnWillChangeRecord, 983
OnWillChangeRecordset, 984
OnWillMove, 984
RecordSize, 981
RecordStatus, 981
SaveToFile, 991
Seek, 986
Sort, 985
UpdateBatch, 990
TAnimate
Active, 269
AutoSize, 269
CanAutoSize, 269
CommonAVI, 268
FileName, 268
FrameCount, 268
FrameHeight, 268
FrameWidth, 268
OnClose, 270
OnOpen, 270
OnStart, 270
OnStop, 270
Open, 269
Play, 269
Repetitions, 269
Reset, 269
ResHandle, 268
ResID, 268
ResName, 268
Seek, 269
StartFrame, 269
Stop, 269
StopFrame, 269
Timers, 269
TAppletApplication
ControlPanelHandle, 730
ModuleCount, 730
Modules, 730
TAppletModule
AppletIcon, 728
Caption, 728
Data, 728
Help, 728
OnActivate, 728
OnInquire, 729
OnNewInquire, 729
OnStartWParms, 728
OnStop, 729
ResidIcon, 728
ResidInfo, 728
ResidName, 728
TApplication, 227
Active, 230
CancelHint, 234
CreateForm, 230
CurrentHelpFile, 234
DialogHandle, 230
ExeName, 229
Handle, 230
HandleException, 395
HandleMessage, 232
HelpCommand, 234
HelpContext, 234
HelpJump, 234
Hint, 233
HintColor, 233
HintHidePause, 233
HintPause, 233
HintShortCuts, 233
HintShortPause, 233
Icon, 229
Initalize, 230
MainForm, 229
Minimize, 234
OnActionExecute, 434
OnException, 396
OnHelp, 234
OnHint, 233
OnMessage, 560
OnMinimize, 234
Restore, 234
Run, 231
ShowMainForm, 232
Terminate, 232
Title, 229
TApplicationEvents
OnActionExecute, 235
OnActionUpdate, 235
OnActivate, 235
OnDeactivate, 235
OnException, 235
OnHint, 235
OnIdle, 235
OnMessage, 235
OnMinimize, 235
OnRestore, 235
OnShortCut, 235
OnShowHint, 235
TAutoIncField, 852
TBaseSocket, 607
Active, 610
BlockMode, 608
BytesReceived, 611
BytesSent, 611
Close, 610
Domain, 608
MapDomain, 608
MapSockType, 608
OnCreateHandle, 609
OnDestroyHandle, 610
OnError, 611
OnReceive, 611
OnSend, 611
Open, 609
PeekBuf, 611
Protocol, 609
ReceiveBuf, 611
Receiveln, 611
Select, 610
SendBuf, 610
Sendln, 610
SendStream, 610
SockType, 608
WaitForData, 610
TBatchMove
AbortOnKeyViol, 892
AbortOnProblem, 892
ChangedTableName, 892
CommitCount, 892

Destination, 891
Execute, 891
KeyViolCount, 892
KeyViolTableName, 892
Mode, 891
MovedCount, 892
ProblemCount, 892
ProblemTableName, 892
RecordCount, 892
Source, 891
Transliterate, 892
TBCDField, 852
Currency, 853
Precision, 853
TBDEDataSet, 865
ApplyUpdates, 869
CacheBlobs, 870
CachedUpdates, 867
CancelUpdates, 868
CanModify, 865
CommitUpdates, 869
ConstraintsDisabled, 866
DisableConstraints, 840, 866
EnableConstraints, 840, 866
ExpIndex, 872
FetchAll, 870
FlushBuffers, 870
GetIndexInfo, 872
Handle, 865
KeySize, 872
OnUpdateError, 871
OnUpdateRecord, 871
RevertRecord, 868
UpdateObject, 866
UpdateRecordTypes, 868
UpdatesPending, 867
UpdateStatus, 868
TBevel
Shape, 267
Style, 267
TBinaryField, 855
TBitBtn
Glyph, 174
Kind, 174
Layout, 174
Margin, 174
NumGlyphs, 174
Spacing, 174
Style, 175
TBitmap, 258
Dormant, 260
FreeImage, 260
Handle, 259
HandleType, 258
IgnorePalette, 260
LoadFromResourceID, 260
LoadFromResourceName, 260, 495
Mask, 261
MaskHandle, 260
Monochrome, 259
Palette, 259
PixelFormat, 259
ReleaseHandle, 259
ReleaseMaskHandle, 261
ReleasePalette, 259
ScanLine, 259
TransparentColor, 260
TransparentMode, 260
TBlobField
Assign, 857
BlobType, 856
Clear, 858
LoadFromFile, 857
LoadFromStream, 857
Modified, 857
SaveToFile, 857
SaveToStream, 857
SetFieldType, 857
Transliterate, 857
TBookmarkList
Clear, 908
CurrentRowSelected, 908
Delete, 909
Find, 908
IndexOf, 908
Refresh, 908
TBooleanField
DisplayValues, 855
TBrush, 243
Bitmap, 244
Color, 243
Style, 243
TButton
Cancel, 173
Default, 173
TBytesField, 855
TCanvas, 246
Arc, 251
Brush, 246
BrushCopy, 252
Chord, 251
ClipRect, 247
CopyMode, 247, 253
CopyRect, 252
Draw, 257
DrawFocusRect, 253
Ellipse, 251
FillRect, 253
FloodFill, 253
Font, 246
FrameRect, 253
Handle, 246
LineTo, 249
MoveTo, 249
Pen, 246
PenPos, 247
Pie, 251
Pixels, 246
PolyBexierTo, 254
PolyBezier, 254
Polygon, 250
Polyline, 250
Rectangle, 250
RoundRect, 250
StretchDraw, 257
TextExtent, 248
TextFlags, 248
TextHeight, 247
TextOut, 247
TextRect, 247
TextWidth, 247
TChart
AddSeries, 457
AllowZoom, 473
AnimatedZoom, 474
AnimatedZoomSteps, 474
BottomAxis, 469
Chart3dPercent, 477
CopyToClipboardBitmap, 475
CopyToClipboardMetafile, 475
DepthAxis, 469
Draw, 476
Height3D, 477
LeftAxis, 469
Legend, 468
LoadChartFromFile, 475
LoadChartFromURL, 475
MaxPointsPerPage, 474
MaxZOrder, 477
MonochromePrinting, 476
NextPage, 474
NumPages, 474
OnGetAxisLabel, 472
OnPageChange, 474
OnUndoZoom, 474
OnZoom, 473
Page, 474

PreviousPage, 474
Print, 476
Printing, 476
PrintLandscape, 476
PrintMargins, 476
PrintOrientation, 476
PrintPartial, 476
PrintPortrait, 476
PrintRect, 476
RefreshSeries, 462
RemoveAllSeries, 458
RemoveSeries, 458
RightAxis, 469
SaveChartToFile, 475
SaveToBitmapFile, 475
SaveToMetaFile, 475
SaveToMetafileEnh, 475
ScaleLastPage, 474
Series, 457
SeriesCount, 457
SeriesDown, 477
SeriesHeight3D, 477
SeriesUp, 477
SeriesWidth3D, 477
TeeCreateMetafile, 475
TopAxis, 469
UndoZoom, 474
View3d, 477
View3dWalls, 477
Width3D, 477
ZoomPercent, 473
ZoomRect, 473
ZOrder, 477
TChartAxis
Automatic, 470
AutomaticMaximum, 470
AutomaticMinimum, 470
Axis, 472
AxisValuesFormat, 471
AxisVisible, 469
CalcMinMax, 470
DateTimeFormat, 471
Grid, 472
Horizontal, 469
Increment, 472
LabelsAngle, 471
LabelsFont, 471
LabelsSize, 471
LabelStyle, 471
Logarithmic, 471
Maximum, 470
Minimum, 470
OtherSide, 469
SetMinMax, 470
Visible, 472
TChartLegend
RectLegend, 468
TextStyle, 468
Visible, 468
TChartSeries, 458
Add, 460
AddNull, 461
AddXY, 461
AddY, 460
AssignValues, 461
Clear, 462
Clicked, 466
ColorEachPoint, 463
ColorRange, 463
ColorSource, 463
Count, 460
Create, 459
DataSource, 461
Delete, 462
FillSampleValues, 461
First, 460
GetCursorValueIndex, 465
GetCursorValues, 466
Last, 460
Locate, 460
Marks, 466
MaxValue, 460
MinValue, 460
Monochrome, 463
OnAfterAdd, 461
OnBeforeAdd, 461
OnClearValues, 462
OnClick, 466
ParentChart, 459
SeriesColor, 462
Total, 460
TotalABS, 460
Value, 459
ValueList, 459
ValueSource, 462
XLabel, 460, 461
XLabelsSource, 462
XScreenToValue, 466
XValue, 460
XValues, 459
YScreenToValue, 466
YValue, 460
YValues, 459
TChartSeriesList, 456
TChartValueList, 459
TCheckBox
AllowGrayed, 176
Checked, 177
State, 176
TCheckListBox
AllowGrayed, 185
Checked, 185
Clear, 180
Header, 185
HeaderBackgroundColor, 185
HeaderColor, 185
ItemEnabled, 185
ItemIndex, 180
Items, 179
OnClickCheck, 185
OnDrawItem, 181
OnMeasureItem, 182
State, 185
TClientSocketThread, 616
ClientSocket, 616
TClipboard, 551
Assign, 554
AsText, 553
Clear, 554
Close, 555
FormatCount, 552
Formats, 552
GetAsHandle, 554
GetComponent, 554
GetTextBuf, 554
HasFormat, 553
Open, 555
SetAsHandle, 554
SetComponent, 554
SetTextBuf, 554
TCollection, 159
Add, 160
BeginUpdate, 160
Clear, 160
Collection, 158
Create, 159
Delete, 160
Destroy, 159
EndUpdate, 161
FindItemID, 161
ID, 159
Index, 158
Insert, 160
ItemClass, 159
Items, 159
Owner, 159
TCollectionItem, 158
Create, 158
Destroy, 158
TColorDialog
Color, 378

CustomColors, 378
Execute, 378
Options, 377
TColumn
Alignment, 911
Assign, 912
AssignedValues, 912
ButtonStyle, 913
Color, 911
DefaultAlignment, 912
DefaultColor, 912
DefaultFont, 912
DefaultReadOnly, 912
DefaultWidth, 912
Depth, 914
DisplayName, 913
DropDownRows, 913
Expanded, 914
Field, 911
FieldName, 911
Font, 911
Grid, 914
ParentColumn, 914
PickList, 913
PopupMenu, 914
ReadOnly, 911
RestoreDefaults, 912
Showing, 914
Title, 912
Visible, 911
Width, 911
TComboBox
CharCase, 186
Clear, 180
DropDownCount, 186
DroppedDown, 186
ItemIndex, 180
Items, 179
MaxLength, 186
OnChange, 186
OnCloseUp, 186
OnDrawItem, 181
OnDropDown, 186
OnMeasureItem, 182
OnSelect, 186
SelectAll, 186
SelLength, 186
SelStart, 186
SelText, 186
Style, 183
Text, 186
TCommonCalendar
BoldDays, 339
Date, 338
DateTime, 337
EndDate, 338
FirstDayOfWeek, 338
MaxDate, 338
MaxSelectRange, 338
MinDate, 338
MultiSelect, 338
OnGetMonthInfo, 339
ShowToday, 339
ShowTodayCircle, 339
TMonthCalColors, 339
WeekNumbers, 339
TCommonDialog, 364
Execute, 364
MessageHook, 365
OnCanClose, 370
OnClose, 365
OnFolderChange, 370
OnIncludeItem, 370
OnSelectionChange, 370
OnShow, 365
OnTypeChange, 370
Template, 364
TComObject
Controller, 636
Factory, 636
RefCount, 636
TComObjectFactory
ClassID, 634
ClassName, 634
ComClass, 634
ComServer, 634
Create, 635
ErrorIID, 636
Instancing, 634
LicString, 636
ShowErrors, 636
SupportsLicensing, 636
ThreadingModel, 634
TComponent, 121
ComObject, 124
ComponentCount, 122
ComponentIndex, 122
Components, 122
ComponentState, 123
Create, 122
Destroy, 122
DestroyComponents, 123
FindComponent, 123
FreeNotification, 124
InsertComponent, 123
Loaded, 422
Name, 121
Notification, 124
Owner, 122
RemoveComponent, 123
Tag, 122
VCLComObject, 124
TComponentList, 157
Items, 157
TComServer
HelpFileName, 640
IsInprocServer, 640
LoadTypeLib, 640
ObjectCount, 640
OnLastRelease, 640
ServerFileName, 640
ServerKey, 640
ServerName, 640
SetServerName, 639
StartMode, 640
TypeLib, 640
UIInteractive, 640
UpdateRegistry, 640
TConfigMode, 948
TContainedAction, 436
TControl, 125
Action, 130
Align, 127
Anchors, 127
BeginDrag, 142
BoundsRect, 126
BringToFront, 128
Caption, 128
ClientHeight, 126
ClientRect, 126
ClientToScreen, 126
ClientWidth, 126
Color, 129
Constraints, 126
ControlState, 131
ControlStyle, 131
DragCursor, 142
DragKind, 141
DragMode, 141
Enabled, 128
EndDrag, 142
ExecuteAction, 130
Font, 129
Height, 126
Hide, 128
Hint, 129
Left, 126
OnClick, 136
OnDblClick, 136
OnDragDrop, 143
OnDragOver, 143
OnEndDrag, 143

OnMouseDown, 136
OnMouseMove, 137
OnMouseUp, 136
OnMouseWheel, 138
OnMouseWheelDown, 139
OnMouseWheelUp, 139
OnStartDrag, 143
Parent, 125
ParentColor, 129
PopupMenu, 130, 206
ScreenToClient, 126
SendToBack, 128
Show, 128
ShowHint, 129
TBasicAction, 432
Text, 128
Top, 126
UpdateAction, 130
Visible, 128
Width, 126
TControlBar
AutoDock, 324
AutoSize, 325
OnBandDrag, 325
OnBandInfo, 324
OnBandMove, 324
OnBandPaint, 325
Picture, 325
RowSize, 325
TControlScrollBar, 213
ButtonSize, 214
Increment, 214
Margin, 214
Position, 213
Range, 213
Size, 214
Style, 214
TCoolBand
Bitmap, 322
Break, 323
Control, 322
FixedSize, 323
ImageIndex, 322
Images, 322
MinHeight, 323
MinWidth, 323
ParentBitmap, 322
Text, 323
TCoolBar
Add, 321
Align, 322
Bands, 321
Count, 321
FindBand, 322
Items, 321
OnChange, 322
Vertical, 322
TCriticalSection, 520
TCurrencyField, 852
Precision, 852
TCustomActionBar, 450
TCustomActionList, 436
TCustomADODataSet, 979
StoreDefs, 885
TCustomControlBar
StickControls, 324
TCustomEdit
AutoSelect, 164
AutoSize, 166
CanUndo, 165
CharCase, 163
Clear, 165
ClearSelection, 164
CopyToClipboard, 165
CutToClipboard, 165
GetSelTextBuf, 164
HideSelection, 164
Modified, 165
OEMConvert, 165
OnChange, 165
PasswordChar, 164
PasteFromClipboard, 165
ReadOnly, 163
SelectAll, 164
SelLength, 164
SelStart, 164
SelText, 164
SetSelTextBuf, 164
Text MaxLength, 164
Undo, 165
TCustomEdiText, 163
TCustomIPClient, 617
TCustomizeDlg
ActionManager, 442
FileName, 443
LoadFromFile, 443
LoadFromStream, 443
OnClose, 443
OnShow, 443
ResetActionBar, 443
ResetUsageData, 443
SaveToFile, 443
SaveToStream, 443
Show, 442
StayOnTop, 442
TCustomListBox
AddItem, 180
Clear, 180
ItemIndex, 180
Items, 179
OnDrawItem, 181
OnMeasureItem, 182
SetItemData, 928
Sorted, 180
Style, 181
TCustomTCPServer, 613, 614
TDatabase
AfterConnect, 957
AfterDisconnect, 957
AliasName, 956
BeforeConnect, 957
BeforeDisconnect, 957
Close, 957
Commit, 959
Connected, 956
DatabaseName, 956
DriverName, 956
InTransaction, 960
KeepConnection, 957
LoginPrompt, 958
OnLogin, 958
Open, 956
Params, 956
Rollback, 960
Session, 955
SessionName, 955
StartTransaction, 959
TransIsolation, 960
TDataSet
Active, 811
ActiveBuffer, 825
ActiveRecord, 825
AfterCancel, 818
AfterClose, 811
AfterDelete, 818
AfterEdit, 817
AfterInsert, 817
AfterOpen, 811
AfterPost, 817
AfterRefresh, 811
AfterScroll, 813
Append, 816
AppendRecord, 816
AutoCalcFields, 820
BeforeCancel, 818
BeforeClose, 811
BeforeDelete, 818
BeforeEdit, 817
BeforeInsert, 817
BeforeOpen, 811
BeforePost, 817

BeforeRefresh, 811
BeforeScroll, 813
Bof, 812
Bookmark, 814
BookmarkSize, 814
BufferCount, 825
Buffers, 825
Cancel, 818
CanModify, 816
CheckBrowseMode, 817
Close, 811
ControlsDisabled, 826
CurrentRecord, 825
DataSource, 826
DefaultFields, 819
Delete, 818
DisableControls, 813
Edit, 817
EnableControls, 813
Eof, 813
FieldByName, 819
FieldByNumber, 819
FieldCount, 819
FieldDefList, 820
FieldDefs, 819
FieldList, 819
Fields, 819
Filter, 821
Filtered, 822
FilterOptions, 822
FindField, 819
First, 812
Insert, 816
InsertRecord, 816
IsEmpty, 812
IsLinkedTo, 826
Last, 812
Locate, 822
Lookup, 823
Modified, 817
MoveBy, 812
Next, 812
ObjectView, 821
OnCalcFields, 820
OnDeleteError, 818
OnEditError, 818
OnFilterRecord, 822
OnNewRecord, 817
OnPostError, 818
Open, 811
Post, 817
Prior, 812
RecNo, 812
RecordCount, 812
Refresh, 811
Resync, 811
State, 814
TBookmarkStr, 814
Translate, 826
UpdateStatus, 818
TDataSource
AutoEdit, 905
DataSet, 904
Edit, 905
IsLinkedTo, 905
OnDataChange, 905
OnStateChange, 905
OnUpdateData, 905
State, 905
TDateField
DisplayFormat, 855
TDateTimeField
DisplayFormat, 855
TDateTimePicker, 340
Checked, 341
DateFormat, 341
DroppedDown, 341
Format, 341
Kind, 340
OnChange, 341
OnCloseUp, 341
OnDropDown, 341
OnUserInput, 340
ParseInput, 340
ShowCheckbox, 341
TDBCheckBox
Checked, 920
State, 920
ValueChecked, 919
ValueUnchecked, 919
TDBCtrlGrid
AllowInsert, 922
ColCount, 921
DoKey, 922
EditMode, 922
OnPaintPanel, 922
Orientation, 921
PanelCount, 922
PanelHeight, 921
PanelIndex, 922
PanelWidth, 921
RowCount, 921
SelectedColor, 922
ShowFocus, 922
TDBDataSet, 872
AutoRefresh, 874
CloseDatabase, 873
Database, 872
DatabaseName, 873
DBHandle, 874
DBLocale, 874
DBSession, 873
OpenDatabase, 873
UpdateMode, 873
TDBEdit
DataField, 906
DataSource, 906
Field, 906
ReadOnly, 906
TDBGrid
Columns, 909
Count, 908
DataSource, 906
DefaultDrawing, 915
FieldCount, 907
Fields, 906
Items, 908
ObjectView, 914
OnCellClick, 915
OnColEnter, 915
OnColExit, 915
OnColumnMoved, 915
OnDrawColumnCell, 915
OnDrawDataCell, 916
OnEditButtonClick, 915
OnTitleClick, 915
Options, 907
SelectedField, 907
SelectedIndex, 907
SelectedRows, 908
ShowPopupEditor, 914
TitleFont, 912
TDBGridColumns
Add, 910
Columns, 909
Grid, 910
Items, 909
LoadFromFile, 910
LoadFromStream, 910
RebuildColumns, 911
RestoreDefaults, 911
SaveToFile, 910
SaveToStream, 910
TDBImage
AutoDisplay, 918
LoadPicture, 918
Picture, 918
TDBListBox
Items, 919
TDBLookupComboBox
CanModify, 924
DataField, 923

DataSource, 923
DropDown, 925
DropDownAlign, 925
DropDownRows, 925
DropDownWidth, 925
KeyField, 923
ListField, 923
ListFieldIndex, 924
ListFields, 923
ListSource, 923
NullValueKey, 924
OnCloseUp, 925
OnDropDown, 925
ReadOnly, 924
Text, 925
TDBLookupListBox
CanModify, 924
DataSource, 923
KeyField, 923
ListField, 923
ListFieldIndex, 924
ListFields, 923
ListSource, 923
NullValueKey, 924
ReadOnly, 924
RowCount, 924
SelectedItem, 924
TDBMemo
AutoDisplay, 917
DataField, 906
DataSource, 906
LoadMemo, 918
TDBNavigator
BeforeAction, 926
ConfirmDelete, 926
Flat, 926
Hints, 926
OnClick, 926
VisibleButtons, 925
TDBRadioGroup
ItemIndex, 920
Items, 920
Value, 920
Values, 920
TDBText
AutoSize, 917
DataSource, 906
WordWrap, 917
TDDEClientConv
CloseLink, 566
ConnectMode, 565
Conv, 567
DdeFmt, 567
DdeService, 564
DdeTopic, 564
ExecuteMacro, 566
ExecuteMacroLines, 566
FormatChars, 567
OnClose, 566
OnOpen, 566
OpenLink, 565
PasteLink, 565
PokeData, 567
PokeDataLines, 567
RequestData, 566
ServiceApplication, 565
SetLink, 565
StartAdvise, 566
WaitStat, 566
TDDEClientItem
DdeConv, 570
DdeItem, 570
Lines, 571
OnChange, 571
Text, 571
TDDEServerConv
ExecuteMacro, 572
OnExecuteMacro, 571
TDDEServerItem
Lines, 572
OnChange, 572
OnPokeData, 572
PokeData, 572
ServerConv, 572
Text, 572
TDefCollection, 884
TDependencies
Items, 540
TDrawGrid
CellRect, 190
Col, 190
ColCount, 187
ColWidths, 188
DefaultColWidth, 188
DefaultDrawing, 191
DefaultRowHeight, 188
FixedColor, 188
FixedCols, 188
FixedRows, 188
GridHeight, 188
GridLineWidth, 188
GridWidth, 188
LeftCol, 188
MouseToCell, 190
OnColumnMoved, 190
OnDrawCell, 191
OnGetEditMask, 191
OnGetEditText, 191
OnRowMoved, 190
OnSelectCell, 190
OnSetEditText, 191
OnTopLeftChanged, 189
Options, 189
Row, 190
RowCount, 187
RowHeights, 188
ScrollBars, 188
TopRow, 188
VisibleColCount, 188
VisibleRowCount, 188
TEvent
Create, 523
LastError, 524
ResetEvent, 523
SetEvent, 523
WaitFor, 523
TField, 819, 827
Alignment, 841
AsBCD, 836
AsBoolean, 836
AsCurrency, 836
AsDateTime, 836
AsFloat, 836
AsInteger, 836
Assign, 834
AssignValue, 834
AsSQLTimeStamp, 837
AsString, 837
AsVariant, 837
AutoGenerateValue, 835
Calculated, 830
CanModify, 834
Clear, 834
ConstraintErrorMessage, 840
CurValue, 835
CustomConstraint, 839
DataSet, 832
DataSize, 837
DataType, 830, 831
DefaultExpression, 838
DisplayLabel, 841
DisplayName, 841
DisplayText, 841
DisplayWidth, 841
EditMask, 840
EditMaskPtr, 840
FieldKind, 829
FieldName, 832
FieldNo, 832
FocusControl, 841
GetData, 836

HasConstraints, 840
ImportedConstraint, 840
Index, 832
IsIndexField, 840
IsNull, 832
KeyFields, 845
Lookup, 830, 846
LookupCache, 846
LookupDataSet, 845
LookupKeyFields, 845
LookupResultField, 845
NewValue, 835
OldValue, 835
OnChange, 842
OnGetText, 842
OnSetText, 842
OnValidate, 842
Origin, 832
ProviderFlags, 835
ReadOnly, 834
RefreshLookupList, 846
Required, 834
SetData, 836
Size, 838
Text, 841
Validate, 842
Value, 833
TFieldDef, 885
Attributes, 885
ChildDefs, 886
DataType, 885
DisplayName, 885
FieldClass, 885
FieldNo, 886
HasChildDefs, 886
InternalCalcField, 886
Name, 885
ParentDef, 886
Precision, 885
Required, 886
TFieldDefs, 884
AddFieldDef, 884
Find, 884
Items, 884
ParentDef, 884
Update, 885
TFindDialog
FindText, 371
Options, 371
Position, 372
TFloatField, 852
Precision, 852
TFont, 244
Charset, 244
Color, 244
FontAdapter, 244
Handle, 244
Height, 244
Name, 244
Pitch, 244
PixelsPerInch, 244
Size, 244
Style, 244
TFontDialog
Device, 375
Execute, 377
Font, 377
MaxFontSize, 376
MinFontSize, 376
Options, 376
TForm
Active, 212
ActiveControl, 212
ActiveMDIChild, 226
ActiveOLEControl, 212
ArrangeIcons, 226
AutoScroll, 213
BorderIcon, 210
BorderStyle, 210
BorderWidth, 210
Cascade, 226
ClientHandle, 213
ClientHeight, 213
ClientRect, 213
ClientWidth, 213
Close, 219
Create, 215
CreateNew, 216
DefocusControl, 212
DropTarget, 213
FocusControl, 212
FormState, 211
FormStyle, 208
Handle, 213
HelpFile, 211
Hide, 217
HorzScrollBar, 213
Icon, 211
KeyPreview, 222
MDIChildCount, 226
MDIChildren, 226
Menu, 211
ModalResult, 217
Next, 227
OleFormObject, 211
OnActivate, 218
OnCanResize, 218, 220
OnClose, 219, 220
OnCloseQuery, 219
OnCreate, 217
OnDeactivate, 219
OnDestroy, 219, 220
OnHide, 219
OnPaint, 218
OnResize, 218, 220
OnShortCut, 221
OnShow, 218
PopupMenu, 211
Position, 209
Previous, 227
Release, 217
Show, 216
ShowModal, 216
Tile, 226
VertScrollBar, 213
Visible, 212
WindowMenu, 211
WindowState, 210
TGraphic, 254
Empty, 255
Height, 255
LoadFromClipboardFormat, 255
LoadFromFile, 255
LoadFromStream, 255
Modified, 255
OnProgress, 256
Palette, 255
PaletteModified, 255
SaveToClipboardFormat, 255
SaveToFile, 255
SaveToStream, 255
Transparent, 255
TGraphicControl
Canvas, 148
Paint, 148
TGraphicField, 856
TGuidField
AsGuid, 855
THandleStream, 587
Create, 588
Handle, 588
Position, 588
Read, 588
Seek, 588
SetSize, 588
Size, 588
Write, 588
TIBDatabase
AddEventNotifier, 1004
AddTransaction, 1004

AllowStreamedConnected, 1000
CheckActive, 1001
CheckInactive, 1001
CloseDataSets, 1002
Connected, 999
CreateDatabase, 998
DatabaseName, 999
DBSQLDialect, 998
DefaultTransaction, 1004
DropDatabase, 998
FindTransaction, 1004
ForceClose, 1002
GetFieldNames, 1003
GetTableNames, 1002
IdleTimer, 1002
LoginPrompt, 999
OnDialectDowngradeWarning, 998
OnIdleTimer, 1002
OnLogin, 1001
Open, 999
Params, 999
RemoveEventNotifier, 1004
RemoveTransaction, 1004
RemoveTransactions, 1004
SQLDialect, 998
TestConnected, 1000
TraceFlags, 1004
TransactionCount, 1003
Transactions, 1003
TIBDataSet
AfterDatabaseDisconnect, 1011
AfterTransactionEnd, 1011
ApplyUpdates, 1013
BatchInput, 1007
BatchOutput, 1007
BeforeDatabaseDisconnect, 1011
BeforeTransactionEnd, 1011
CachedUpdates, 1012
CancelUpdates, 1014
CanModify, 1012
Database, 1009
DatabaseFree, 1011
DeleteSQL, 1010
Filtered, 1015
ForcedRefresh, 1012
GeneratorField, 1012
InsertSQL, 1010
IsSequenced, 1015
LiveMode, 1012
ModifySQL, 1010
OnUpdateError, 1011, 1014
OnUpdateRecord, 1011, 1014
Params, 1010
Prepare, 1010
QDelete, 1010
QInsert, 1010
QModify, 1010
QRefresh, 1010
QSelect, 1010
RecNo, 1015
RefreshSQL, 1010
RevertRecord, 1014
SelectSQL, 1010
Transaction, 1009
TransactionFree, 1011
Undelete, 1013
UniDirectional, 1015
UnPrepare, 1010
UpdateObject, 1014
UpdateRecordTypes, 1013
UpdatesPending, 1013
TIBEvents
Events, 1022
OnEventAlert, 1022
Registered, 1022
RegisterEvents, 1022
TIBExtract
ExtractObject, 1021
TIBInputDelimitedFile
ColDelimiter, 1008
FileName, 1008
RowDelimiter, 1008
TIBInputRawFile
FileName, 1008
TIBOutputDelimitedFile
ColDelimiter, 1008
FileName, 1008
RowDelimiter, 1008
TIBOutputRawFile
FileName, 1008
TIBQuery
BatchInput, 1007
BatchOutput, 1007
Database, 1009
Transaction, 1009
TIBSQL
BatchInput, 1007
BatchOutput, 1007
Bof, 1007
Current, 1007
Database, 1005
Eof, 1007
ExecQuery, 1006
FieldByName, 1007
FieldIndex, 1007
Fields, 1007
GoToFirstRecordOnExecute, 1007
Next, 1007
OnSQLChanging, 1005
ParamByName, 1006
Params, 1006
Prepare, 1005
Prepared, 1006
SQL, 1005
SQLType, 1006
Transaction, 1005
TIBStoredProc
Database, 1009
Transaction, 1009
TIBTable
Database, 1009
Exists, 1016
TableTypes, 1016
Transaction, 1009
TIBTransaction
AddDatabase, 1016
AutoStopAction, 1018
Commit, 1017
CommitRetaining, 1018
DatabaseCount, 1016
Databases, 1016
DefaultAction, 1018
DefaultDatabase, 1016
FindDatabase, 1016
IdleTimer, 1018
InTransaction, 1018
OnIdleTimer, 1019
RemoveDatabase, 1016
RemoveDatabases, 1017
Rollback, 1017
RollbackRetaining, 1017
SQLObjectCount, 1017
SQLObjects, 1017
StartTransaction, 1017
TIBUpdateSQL
Apply, 1020
DataSet, 1019
DeleteSQL, 1020
ExecSQL, 1020
InsertSQL, 1020
ModifySQL, 1020

RefreshSQL, 1020
SetParams, 1020
SQL, 1020
TIcon
Handle, 257
Height, 257
ReleaseHandle, 257
Width, 257
TImage
Center, 267
IncrementalDisplay, 268
Picture, 267
Stretch, 267
Transparent, 267
TIndexDef, 887
CaseInsFields, 888
DescFields, 888
Expression, 888
FieldExpression, 888
Fields, 888
GroupingLevel, 888
Options, 888
Source, 888
TIndexDefs
Add, 886
AddIndexDef, 886
Assign, 886
Find, 887
FindIndexForFields, 887
GetIndexForFields, 887
Items, 886
Update, 886
TIniFile, 343
Create, 343
DeleteKey, 346
EraseSection, 345
FileName, 343
ReadBool, 343
ReadDate, 344
ReadDateTime, 344
ReadFloat, 343
ReadInteger, 344
ReadSection, 344
ReadSections, 344
ReadSectionValues, 344
ReadString, 344
ReadTime, 344
SectionExists, 346
ValueExists, 346
WriteBool, 344
WriteDate, 344
WriteDateTime, 345
WriteFloat, 344
WriteInteger, 344
WriteString, 344
WriteTime, 345
TIntegerField, 852
MaxValue, 852
MinValue, 852
TIPSocket, 611
GetSocketAddr, 612
LocalDomainName, 612
LocalHost, 611
LocalHostAddr, 612
LocalHostName, 612
LocalPort, 611
LookupHostAddr, 612
LookupHostName, 612
LookupPort, 612
LookupProtocol, 612
ReceiveFrom, 613
RemoteHost, 611
RemotePort, 611
SendTo, 612
TJPEGImage, 263
Assign, 263
Compress, 264
CompressionQuality, 263
DIBNeeded, 263
JPEGNeeded, 263
Performance, 264
PixelFormat, 264
ProgressiveDisplay, 264
ProgressiveEncoding, 264
Scale, 264
Smoothing, 264
TLabel
AutoSize, 270
Caption, 270
OnMouseEnter, 270
OnMouseLeave, 270
Transparent, 270
WordWrap, 270
TLargeintField, 852
TList
Add, 153
Capacity, 152
Clear, 153
Count, 152
Delete, 153
Exchange, 153
Expand, 153
First, 153
IndexOf, 153
Insert, 153
Items, 152
Last, 153
List, 153
Move, 153
Pack, 153
Remove, 153
Sort, 153
TListBox
AddItem, 180
BorderStyle, 184
Clear, 180
Columns, 184
ExtendedSelect, 183
IntegralHeight, 184
ItemAtPos, 184
ItemIndex, 180
ItemRect, 185
Items, 179
MultiSelect, 183
OnDrawItem, 181
OnMeasureItem, 182
SelCount, 183
Selected, 184
TabWidth, 184
TListColumn, 306
Alignment, 306
AutoSize, 306
Caption, 306
Index, 306
MaxWidth, 307
MinWidth, 307
Width, 307
TListColumns
Count, 306
Items, 306
TListItem, 302
CancelEdit, 303
Caption, 302, 930
Checked, 303
Data, 303, 930
Delete, 304, 930
EditCaption, 303
Focused, 302
GetPosition, 303
ImageIndex, 305
Index, 303, 930
Left, 303
OverlayIndex, 305
Position, 303
Selected, 302
SetPosition, 303
StateIndex, 305
SubItems, 302, 930
Top, 303
TListItems
Add, 308, 930
BeginUpdate, 308

Clear, 308, 931
Count, 307
Delete, 308, 931
EndUpdate, 308
IndexOf, 308
Insert, 308, 930
Item, 307
ItemIndex, 307
Owner, 308
TopItem, 307
ViewOrigin, 308
TListView
AlphaSort, 310
Checkboxes, 305
ClearSelection, 309
Columns, 305
CopySelection, 309
DeleteSelected, 309
Deleting, 304
DisplayRect, 313
FindCaption, 311
FindData, 311
GridLines, 306
Handle, 302
HideSelection, 309
HotTrack, 305
HotTrackStyles, 305
HoverTime, 305
IconOptions, 305
ItemFocused, 309
Items, 302, 930
LargeImages, 304
MoveSelection, 309
MultiSelect, 309
OnCompare, 310
OnCustomDraw, 312
OnCustomDrawItem, 312
OnCustomDrawSubItem, 313
OnDrawItem, 312
OnEdited, 304
OnEditing, 303
SelCount, 309
SelectAll, 309
Selected, 309
ShowColumnHeaders, 306
SmallImages, 304
SortType, 310
StateImages, 304
ViewStyle, 304, 318
VisibleRowCount, 306
TMainMenu
AutoMerge, 206
GetOle2AcceleratorTable, 206
Merge, 206
SetOle2MenuHandle, 206
Unmerge, 206
TMaskEdit
EditMask, 166
EditText, 167
GetTextLen, 168
ValidateEdit, 167
TMediaPlayer
AutoEnable, 752
AutoOpen, 749
AutoRewind, 753
Back, 753
Capabilities, 751
Close, 756
ColoredButtons, 752
DeviceID, 751
DeviceType, 749
Display, 753
DisplayRect, 753
Eject, 753
EnabledButtons, 752
EndPos, 751
Error, 755
ErrorMessage, 756
FileName, 749
Frames, 753
Length, 750
Mode, 754
Next, 753
Notify, 755
NotifyValue, 755
OnClick, 754
OnNotify, 755
OnPostClick, 754
Open, 749
Pause, 752
Play, 752
Position, 751
Previous, 753
Resume, 752
Rewind, 753
Save, 755
Shareable, 749
Start, 750
StartPos, 751
StartRecording, 753
Step, 753
TimeFormat, 750
TrackLength, 750
TrackPosition, 750
Tracks, 750
Wait, 754
TMemo
Alignment, 168
CaretPos, 168
Lines, 168
ScrollBars, 168
WantReturns, 169
WantTabs, 169
WordWrap, 168
TMemoField, 856
TMenu, 205
Items, 205
OnChange, 205
TMenuItem, 196
Add, 200
AutoCheck, 198
AutoHotkeys, 196
AutoLineReduction, 203
Bitmap, 199
Break, 202
Caption, 196
Checked, 198
Clear, 202
Count, 199
Create, 200
Default, 197
Delete, 202
Destroy, 200
Find, 199
GetParentMenu, 196
GroupIndex, 198
ImageIndex, 199
IndexOf, 199
Insert, 200
InsertNewLineAfter, 203
InsertNewLineBefore, 203
IsLine, 203
Items, 199
MenuIndex, 199
NewBottomLine, 203
NewTopLine, 203
OnAdvancedDrawItem, 204
OnClick, 197
OnDrawItem, 204
OnMeasureItem, 204
Parent, 199
RadioItem, 198
Remove, 202
RethinkHotkeys, 197
RethinkLines, 203
ShortCut, 197
SubMenuImages, 200
TMetafile, 261

CreatedBy, 262
Description, 262
Enhanced, 261
Handle, 261
Height, 261
Inch, 261
MMHeight, 261
MMWidth, 261
Palette, 261
Width, 261
TMetafileCanvas, 262
Create, 262
CreateWithComment, 262
TNamedItem, 884
TNavigateBtn, 925
TNumericField
DisplayFormat, 850
EditFormat, 851
TObject, 110, 115
ClassInfo, 116
ClassName, 117
ClassNameIs, 117
ClassParent, 117
ClassType, 117
Constructor, 116
Destructor, 116
FieldAddress, 117
Free, 116
FreeInstance, 116
GetInterface, 119
GetInterfaceEntry, 119
GetInterfaceTable, 119
InheritsFrom, 117
MethodAddress, 117
MethodName, 117
NewInstance, 116
TObjectField
FieldCount, 848
Fields, 848
FieldValues, 849
FullName, 850
HasParent, 850
ObjectType, 850
ParentField, 850
TObjectList
Add, 156
Create, 156
Extract, 156
FindInstanceOf, 156
First, 155
IndexOf, 157
Insert, 156
Items, 155
Last, 155
OwnsObjects, 155
Remove, 156
TOLEContainer
AllowInPlace, 699
AutoActivate, 699
AutoVerbMenu, 700
CanPaste, 701
Close, 700
CopyOnSave, 701
CreateLinkToFile, 697
CreateObject, 697
CreateObjectFromFile, 697
CreateObjectFromInfo, 698
DestroyObject, 700
Iconic, 699
InsertObjectDialog, 696
Linked, 699
LoadFromStream, 697
ObjectVerbs, 700
OleClassName, 697
OleObject, 698
OleObjectInterface, 698
OnActivate, 702
OnDeactivate, 702
OnObjectMove, 702
OnResize, 702
Paste, 701
PasteSpecialDialog, 701
PrimaryVerb, 700
SaveAsDocument, 700
SaveToFile, 700
SaveToStream, 700
State, 698
UpdateObject, 699
TOpenDialog
DefaultExt, 366
Execute, 368
FileEditStyle, 366
FileName, 368
Filter, 365
FilterIndex, 365
HistoryList, 366
InitialDir, 366
Options, 366
OptionsEx, 368
Title, 366
TPageControl
ActivePage, 285
ActivePageIndex, 285
FindNextPage, 286
PageCount, 285
Pages, 285
SelectNextPage, 286
TPageSetupDialog
MarginLeft, 380
MinMarginBottom, 380
MinMarginLeft, 380
MinMarginRight, 380
MinMarginTop, 380
Options, 381
PageHeight, 380
PageWidth, 380
Units, 380
TPaintBox
OnPaint, 266
Paint, 266
TParaAttributes, 169
Alignment, 169
FirstIndent, 169
LeftIndent, 169
Numbering, 170
RightIndent, 169
Tab, 170
TabCount, 170
TPen, 241
Color, 242
Handle, 242
TPenMode, 243
Width, 242
TPersistent
Assign, 119
AssignTo, 119
TPicture, 265
Bitmap, 265
Icon, 265
LoadFromClipboardFormat, 265
LoadFromFile, 265
Metafile, 265
SaveToClipboardFormat, 265
SaveToFile, 265
TPopupMenu
AutoPopup, 207
MenuAnimation, 207
OnPopup, 207
Popup, 207
PopupAlignment, 207
PopupComponent, 206
TPrintDialog, 379
Copies, 380
Execute, 379, 380
FromPage, 379
MaxPage, 379
MinPage, 379
Options, 379
PrintRange, 379

PrintToFile, 380
ToPage, 379
TPrinterSetupDialog, 379
Execute, 379
Options, 379
TQuery
Constrained, 897
ExecSQL, 894
Open, 894
ParamByName, 896
ParamCount, 895
Params, 895
Prepare, 893
Prepared, 893
RequestLive, 896
RowsAffected, 897
SQL, 893
Text, 893
UnPrepare, 894
TRadioButton
Alignment, 175
Checked, 175
TRadioGroup
Columns, 176
ItemIndex, 176
Items, 175
TRegistry, 349
Access, 349
CloseKey, 351
Create, 349
CreateKey, 351
CurrentKey, 350
CurrentPath, 350
DeleteKey, 351
DeleteValue, 353
GetDataInfo, 354
GetDataSize, 354
GetDataType, 354
GetKeyInfo, 354
GetKeyNames, 354
GetValueNames, 352, 354
HasSubKeys, 351
KeyExists, 351
LazyWrite, 351
LoadKey, 355
MoveKey, 351
OpenKey, 351
OpenKeyReadOnly, 351
ReadBinaryData, 352
ReadBool, 352
ReadCurrency, 352
ReadDate, 352
ReadDateTime, 352
ReadFloat, 352
ReadInteger, 352
ReadString, 352
ReadTime, 352
RegistryConnect, 350
RenameValue, 353
ReplaceKey, 355
RestoreKey, 355
RootKey, 350
SaveKey, 354
UnLoadKey, 355
ValueExists, 352
WriteBinaryData, 353
WriteBool, 353
WriteCurrency, 353
WriteDate, 353
WriteDateTime, 353
WriteFloat, 353
WriteInteger, 353
WriteString, 353
WriteTime, 353
TRegistryIniFile, 347
Access, 349
CurrentKey, 349
OpenKey, 349
OpenKeyReadOnly, 349
RegIniFile, 347
RootKey, 348
TReplaceDialog
FindText, 371
OnReplace, 374
Options, 371
Position, 372
ReplaceText, 374
TRichEdit
Paragraph, 169
FindText, 171
OnProtectChange, 171
OnResizeRequest, 171
OnSaveClipboard, 172
OnSelectionChange, 171
PlainText, 169
TSaveDialog
DefaultExt, 366
Execute, 368
FileEditStyle, 366
FileName, 368
Filter, 365
FilterIndex, 365
HistoryList, 366
InitialDir, 366
Options, 366
OptionsEx, 368
Title, 366
TScreen, 235
ActiveControl, 237
ActiveCustomForm, 238
ActiveForm, 237
Cursor, 236
Cursors, 236
CustomFormCount, 238
CustomForms, 238
DataModuleCount, 238
DataModules, 238
DesktopHeight, 236
DesktopLeft, 236
DesktopTop, 236
DesktopWidth, 236
Fonts, 237
FormCount, 237
Forms, 237
Height, 236
HintFont, 237
IconFont, 237
MenuFont, 237
MonitorCount, 236
Monitors, 236
PixelsPerInch, 236
ResetFonts, 237
Width, 236
TSeriesMarks
Arrow, 467
ArrowLength, 467
BackColor, 467
Clicked, 468
Clip, 467
Font, 467
Frame, 467
Style, 467
Transparent, 467
Visible, 467
TServerSocketThread, 615
ClearThreadPool, 616
ServerSocket, 616
ThreadCacheSize, 616
ThreadPool, 616
TService
ErrCode, 546
LogMessage, 547
OnExecute, 545
ServiceCount, 547
Terminated, 545
WaitHint, 546
Win32ErrCode, 546
TServiceAppliaction
ServiceCount, 547
TServiceApplication, 547
TServiceThread, 544
ProcessRequests, 545

TServiceType
AfterInstall, 542
AfterUninstall, 542
AllowPause, 542
AllowStop, 542
BeforeInstall, 542
BeforeUninstall, 542
Dependencies, 540
DisplayName, 539
ErrorSeverity, 541
Interactive, 538
LoadGroup, 540
Name, 539
OnContinue, 543
OnPause, 543
OnShutdown, 543
OnStart, 543
OnStop, 543
Param, 543
ParamCount, 543
Password, 539
ReportStatus, 544
ServiceStartName, 539
ServiceThread, 544
ServiceType, 538
StartType, 540
Status, 543
TagID, 540
TSession
Active, 946
AddAlias, 949
AddDriver, 949
AddPassword, 953
AddStandardAlias, 950
AutoSessionName, 947
ConfigMode, 948
DatabaseCount, 947
Databases, 947
DeleteAlias, 951
DeleteDriver, 949
DropConnections, 948
GetAliasDriverName, 951
GetAliasNames, 950
GetAliasParams, 951
GetConfigParams, 951
GetDatabaseNames, 952
GetDriverNames, 949
GetDriverParams, 949
GetFieldNames, 953
GetPassword, 954
GetStoredProcNames, 953
GetTableNames, 952
Handle, 948
IsAlias, 950
KeepConnections, 947
Locale, 949
ModifyAlias, 951
ModifyDriver, 949
NetFileDir, 948
OnPassword, 953
OnStartup, 947
PrivateDir, 948
RemoveAllPasswords, 954
RemovePassword, 954
SessionName, 947
SQLHourGlass, 948
TSessionList, 954
Count, 954
CurrentSession, 955
FindSession, 955
GetSessionNames, 955
List, 954
OpenSession, 955
Sessions, 954
TShape
Brush, 267
Pen, 267
Shape, 267
TSmallintField, 852
TSpeedButton
AllowAllUp, 179
Down, 179
Flat, 178
Glyph, 178
GroupIndex, 179
Transparent, 179
TStatusBar
Add, 320
AutoHint, 320
Delete, 320
Insert, 320
Items, 319
OnCreatePanelClass, 320
OnDrawPanel, 321
OnHint, 320
Panels, 319
SimplePanel, 319
SimpleText, 319
Style, 321
TStatusPanel, 319
Text, 319
TStatusPanels, 319
TStoredProc
ExecProc, 898
GetResults, 899
Open, 898
Overload, 898
ParamBindMode, 898
Params, 898
StoredProcName, 898
TStream, 120
CopyFrom, 121
Position, 120
Read, 120
ReadBuffer, 120
Seek, 121
Size, 120
Write, 120
WriteBuffer, 120
TStringField
FixedChar, 854
Transliterate, 854
TStringGrid
Cells, 192
Col, 190
ColCount, 187
Cols, 192
DefaultDrawing, 191
FixedCols, 187
FixedRows, 187
Objects, 193, 932
OnColumnMoved, 190
OnDrawCell, 191
OnGetEditMask, 191
OnGetEditText, 191
OnRowMoved, 190
OnSelectCell, 190
OnSetEditText, 191
OnTopLeftChanged, 189
Options, 189
Row, 190
RowCount, 187
Rows, 192
TStringList
Duplicates, 155
Find, 155
OnChange, 155
OnChanging, 155
Sort, 154
Sorted, 154
TStrings, 150, 168
Add, 151
AddObject, 151
AddStrings, 151
Append, 151
Clear, 151
Count, 150
Delete, 151
Equals, 151
Exchange, 151
GetText, 150
IndexOf, 151

IndexOfName, 151
IndexOfObject, 151
Insert, 152
InsertObject, 152
LoadFromFile, 150
LoadFromStream, 150
Move, 152
Objects, 150
SaveToFile, 150
SaveToStream, 150
SetText, 150
Strings, 150
Values, 151
TTabControl
GetHitTestInfoAt, 284
Images, 285
IndexOfTabAt, 283
MultiLine, 283
MultiSelect, 282
OnChange, 285
OnChanging, 284
OnDrawTab, 285
OnGetImageIndex, 285
RowCount, 283
ScrollOpposite, 283
ScrollTabs, 283
Style, 281
TabHeight, 282
TabIndex, 282
TabPosition, 282
TabRect, 284
Tabs, 281
TabWidth, 282
TTable
AddIndex, 877
ApplyRange, 879
BatchMove, 890
CancelRange, 880
CanModify, 876
CloseIndexFile, 879
CreateTable, 883
DefaultIndex, 876
DeleteIndex, 878
DeleteTable, 890
EditKey, 882
EditRangeEnd, 880
EditRangeStart, 880
EmptyTable, 889
Exclusive, 876
Exists, 875
FieldDefs, 883
FindKey, 882
FindNearest, 883
GetIndexNames, 876
GotoKey, 882
GotoNearest, 882
IndexDefs, 883
IndexFieldCount, 876
IndexFieldNames, 877
IndexFields, 876
IndexFiles, 878
IndexName, 877, 881
KeyExclusive, 880
LockTable, 890
MasterFields, 881
MasterSource, 881
OpenIndexFile, 879
ReadOnly, 876
RenameTable, 889
SetKey, 882
SetRange, 880
SetRangeEnd, 879
SetRangeStart, 879
TableName, 875
TableType, 875
UnlockTable, 890
TTabSheet
Highlighted, 287
ImageIndex, 287
PageControl, 286
PageIndex, 286
TabIndex, 287
TabVisible, 287
TTCPClient
Connect, 618
Connected, 617
DisConnect, 618
TTCPServer
Accept, 615
BlockMode, 613
Close, 615
Listening, 614
OnAccept, 615
OnCreateHandle, 614
OnDestroyHandle, 615
OnGetThread, 617
OnListening, 614
ServerSocketThread, 615
TTextAttributes, 170
DefAttributes, 170
SelAttributes, 170
TThread
Create, 509
Execute, 509
FatalException, 511
FreeOnTerminate, 511
OnTerminate, 510
Priority, 511
Resume, 509
ReturnValue, 511
Suspend, 509
Synchronize, 513
Terminate, 510
Terminated, 510
WaitFor, 511
TTimeField
DisplayFormat, 855
TTimer
Enabled, 337
Interval, 336
OnTimer, 336
TToolBar
ButtonCount, 315
ButtonHeight, 315
Buttons, 315
ButtonWidth, 315
Caption, 314
Customizable, 315
CustomizeKeyName, 315
CustomizeValueName, 315
DisabledImages, 316
EdgeBorders, 314
EdgeInner, 314
EdgeOuter, 314
Flat, 314
HotImages, 316
Images, 316
Indent, 314
List, 314
Menu, 315
ShowCaptions, 315
Transparent, 314
Wrapable, 314
TToolButton, 316
AllowAllUp, 316
CheckMenuDropdown, 318
Down, 317
DropdownMenu, 317
Grouped, 316
Indeterminate, 317
Marked, 317
Style, 316
TTrackBar
Frequency, 326
Max, 326
Min, 326
OnChange, 326
Orientation, 326
Position, 326
SetTick, 327
ThumbLength, 327
TickMarks, 327

TickStyle, 326
TTreeNode
AbsoluteIndex, 292
Collapse, 297
Cut, 296
Data, 289
DeleteChildren, 295
Expand, 297
Expanded, 297
Focused, 296
FullCollapse, 297
FullExpand, 297
GetFirstChild, 292
GetLastChild, 292
GetNext, 292
GetNextChild, 292
GetNextSibling, 292
GetNextVisible, 293
GetPrev, 292
GetPrevChild, 292
GetPrevSibling, 292
GetPrevVisible, 293
HasAsParent, 291
HasChildren, 291
ImageIndex, 299
Index, 292
IsEditing, 296
IsVisible, 292
Level, 292
MoveTo, 296
OverlayIndex, 299
Owner, 289
Parent, 291
SelectedIndex, 299
StateIndex, 299
Text, 289
TreeView, 289
TTreeNodes, 287
Item, 288
TTreeView
Add, 293
AddChild, 293
AddChildFirst, 293
AddChildObject, 294
AddChildObjectFirst, 294
AddFirst, 293
AddObject, 294
AddObjectFirst, 294
BeginUpdate, 294
Clear, 295
Count, 288
Delete, 295
Deleting, 295
Deselect, 289
DisplayRect, 300
EndUpdate, 294
FindNextToSelect, 291
GetFirstNode, 288
GetHitTestInfoAt, 290
GetNodeAt, 290
Handle, 288
HideSelection, 288
Images, 299
Insert, 293
InsertNode, 294
InsertObject, 294
MultiSelect, 290
MultiSelectStyle, 290
OnAddition, 294
OnAdvancedCustomDraw, 301
OnAdvancedCustomDrawItem, 301
OnChange, 289
OnChanging, 289
OnCollapsed, 298
OnCollapsing, 298
OnCustomDraw, 300
OnCustomDrawItem, 300
OnDeletion, 295
OnEdited, 296
OnEditing, 296
OnExpanded, 298
OnExpanding, 298
OnGetImageIndex, 300
OnGetSelectedIndex, 300
Owner, 288
Selected, 289
SelectionCount, 290
Selections, 290
ShowButtons, 298
ShowLines, 298
ShowRoot, 298
StateImages, 299
TUDPClient
Connect, 618
Connected, 617
Disconnect, 618
TUpdateSQL, 866
Apply, 900
DeleteSQL, 900
ExecSQL, 900
InsertSQL, 900
ModifySQL, 900
Query, 899
SetParams, 900
TUpDown
AlignButton, 178
ArrowKeys, 178
Associate, 178
Increment, 177
Max, 177
Min, 177
Orientation, 178
Position, 177
Thousands, 178
Wrap, 177
TVarBytesField, 855
TWinControl, 132
AutoSize, 144
Broadcast, 134
CanFocus, 135
ContainsControl, 134
ControlAtPos, 134
ControlCount, 134
Controls, 134
CreateParams, 133
CreateWindowHandle, 133
CreateWnd, 133
Dock, 147
DockClientCount, 144
DockClients, 145
DockSite, 144
DragKind, 145
DragMode, 145
FloatingDockSiteClass, 145
Focused, 135
GetTabOrderList, 135
Handle, 133
HandleAllocated, 133
HandleNeeded, 133
ManualDock, 147
ManualFloat, 147
OnDockDrop, 146
OnDockOver, 146
OnEndDock, 147
OnEnter, 135
OnExit, 135
OnGetSiteInfo, 146
OnKeyDown, 139
OnKeyPress, 140
OnKeyUp, 139
OnStartDock, 146
OnUnDock, 147
RemoveControl, 134
SetFocus, 135
TabOrder, 136
TabStop, 135
TWordField, 852

U–V

UpdateObject, 866
VCL (Visual Component Library), 114

W

Win32 API
AngleArc, 252
BindToObject, 742
CallNamedPipe, 599
ChangeServiceConfig, 537
CharToOem, 54
CloseServiceHandle, 531
CompareIDs, 745
ConnectNamedPipe, 596
ControlService, 533
CopyFile, 92
CreateDC, 271
CreateFileMapping, 577
CreateFontIndirect, 246
CreateMailslot, 585
CreateMutex, 524
CreateNamedPipe, 592
CreateProcess, 499
CreateSemaphore, 526
CreateService, 532
CreateToolhelp32Snapshot, 502
CreateViewObject, 745
CreateWindow, 711
DateTimeToSystemTime, 329
DeleteCriticalSection, 519
DeleteService, 538
DisconnectNamedPipe, 597
DllEntryPoint, 484
DrawText, 248
EnterCriticalSection, 519
EnumObjects, 743
EnumServicesStatus, 534
ExitProcess, 501
ExitWindows, 724
ExitWindowsEx, 724
ExtractAssociatedIcon, 257
FileTimeToSystemTime, 506
FindWindow, 559
FreeLibrary, 491
FreeLibraryAndExitThread, 493
GetAttributesOf, 745
GetBValue, 241
GetComputerName, 719
GetCurrentDirectory, 98
GetCurrentProcess, 504
GetDC, 271
GetDeviceCaps, 272
GetDiskFreeSpaceEx, 99
GetDisplayNameOf, 742
GetDoubleClickTime, 138
GetDriveType, 99
GetFileSizeEx, 95
GetGValue, 241
GetKeyState, 320
GetLastError, 392
GetMailslotInfo, 586
GetMapMode, 275
GetMessage, 713
GetNamedPipeInfo, 597
GetNearestColor, 378
GetPriorityClass, 505
GetProcAddress, 490
GetProcessTimes, 505
GetRValue, 241
GetSystemDirectory, 98
GetSystemInfo, 722
GetTempPath, 98
GetThreadTimes, 512
GetTickCount, 336
GetUIObjectOf, 747
GetUserName, 719
GetVersionEx, 720
GetVolumeInformation, 100
GetWindowsDirectory, 98, 182
GlobalMemoryStatus, 720
InitializeCriticalSection, 518
InitializeSecurityDescriptor, 601
InvalidateRect, 272
KillTimer, 335
LeaveCriticalSection, 519
LoadCursor, 236
LoadLibrary, 490
LoadLibraryEx, 490
LockWorkStation, 725
MapViewOfFile, 578
MessageBeep, 118, 748
MessageBoxEx, 358
MoveFile, 92
OpenMutex, 525
OpenProcess, 504
OpenSCManager, 530
OpenSemaphore, 526
OpenService, 532
ParseDisplayName, 741
PeekMessage, 138
PlaySound, 757
PostMessage, 718
Process32First, 502
Process32Next, 502
QueryServiceConfig, 536
ReadFileEx, 90
RegisterClass, 711
RegisterClipboardFormat, 552, 556
RegisterWindowMessage, 560
ReleaseMutex, 525
ReleaseSemaphore, 527
RGB, 241
SendMessage, 718
SetMailslotInfo, 586
SetMapMode, 274
SetNamedPipeHandleState, 596
SetNameOf, 747
SetPriorityClass, 505
SetTimer, 334
SetViewportExtEx, 278
SetViewportOrgEx, 275, 276
SetVolumeLabel, 101
SetWindowExtEx, 278
SetWindowOrgEx, 276
SHBrowseForFolder, 735
ShellExecute, 723
SHGetDesktopFolder, 740
SHGetPathFromIDList, 740
SHGetSpecialFolderPath, 98, 737
ShowWindow, 712
Sleep, 510
StartService, 533
SystemTimeToDateTime, 329
SystemTimeToFileTime, 506
TerminateProcess, 501
TerminateThread, 510
UnMapViewOfFile, 579
UpdateWindow, 713
WaitForMultipleObjects, 521
WaitForSingleObject, 521
WaitNamedPipe, 600

WindowProc, 715
WinExec, 723
WriteFileEx, 90
Windows Shell, 733

А

абзац, 169
абстрактный класс, 115
асинхронная операция, 90
атрибуты защиты, 500

Б

блокировка таблицы, 890
буфер обмена
Clipboard, 552

В

ввод-вывод, 72
время, 328
всплывающая подсказка, 129
всплывающее меню, 130
вывод текста, 247
вычисляемое поле, 820

Г

генерация случайного числа
RandG, 60
Random, 60
Randomize, 60

Д

дата, 328
дескриптор контекста, 271
деструктор, 116, 421
диалоги и сообщения
CreateMessageDialog, 360
InputBox, 361
InputQuery, 361
LoginDialog, 363
LoginDialogEx, 363
MessageBoxEx, 358
MessageDlg, 357
MessageDlgPos, 358
RemoteLoginDialog, 363
SelectDirectory, 362
ShowMessage, 356
ShowMessageFmt, 356
динамический обмен данными (DDE), 563
директива
Abstract, 415, 420
Cdecl, 487
Default, 413
Dynamic, 415, 417
Implements, 630
Index, 412
NoDefault, 413
Overload, 44, 49
Override, 417
Pascal, 487
Register, 487
Reintroduce, 418
Safecall, 487
Static, 415
Stdcall, 487
Stored, 413
Virtual, 415, 417
директива компилятора
{$APPTYPE CONSOLE}, 17
{$ASSERTIONS OFF}, 392
{$ASSERTIONS ON}, 392
{$C+}, 392
{$C-}, 392
{$I+}, 88
{$I-}, 88
{$R *.DFM}, 51
{$WARN SYMBOL_PLATFORM OFF}, 638

З

запись
TBrowseInfo, 735
TCreateInfo, 698
TFileTime, 506
TFMTBCDField, 853
TGUID, 854
TInterfaceTable, 119
TLogFont, 245
TMemoryStatus, 720
TMsg, 560, 714
TOSVersionInfo, 721
TOverlapped, 91
TPoint, 126
TProcessEntry32, 503
TRect, 126, 252
TRegKeyInfo, 354
TSearchRec, 96, 193
TSecurityAttributes, 500
TSize, 248
TSQLTimeStamp, 856
TStringItem, 154
TSystemInfo, 722
TSystemTime, 506
TWMKey, 221
TWndClass, 710
поля записи, 30
с вариантным полем, 31
защищенная секция
try..except, 384
try..finally, 385

И

именованный канал, 591
имя компьютера, 719
индекс, 775
индексирование, 793
интерфейс
IClassFactory, 633
IDispatch, 651
IEnumIDList, 743
IInterface, 629
IShellFolder, 740
IUnknown, 631
папки, 740
приложения
MDI, 223
SDI, 227
реализация, 629
исключительная ситуация, 382
EAbort, 389, 390
EAbstractError, 389
EAssertionFailed, 389, 390
EConvertError, 332, 389
EExternal, 389
EHeapException, 389
EInOutError, 389
EInvalidCast, 389
EOSError, 389, 393
EPackageError, 389
EPropReadOnly, 389
EPropWriteOnly, 389
ESafecallException, 389
EVariantError, 389
Exception, 387

К

кисть, 243
класс, 103
ключ первичный, 764
ключевое слово
array, 27
const, 18
file, 73

finalization, 50
function, 45
implementation, 49
in, 33
Inherited, 418
initialization, 50
interface, 49
library, 481
nil, 34
out, 46
overload, 49
procedure, 44
program, 16
set, 32
uses, 50
var, 18
команда
TAction, 431
компонент
TActionList, 438
TActionMainMenuBar, 453
TADODataSet, 979
TADOQuery, 993
TADOStoredProc, 993
TADOTable, 993
TAnimate, 268
TApplicationEvents, 235, 397
TBevel, 267
TBitBtn, 174
TButton, 172
TChart, 455
TCheckBox, 176
TCheckListBox, 185
TColorDialog, 377
TComboBox, 186
TControlBar, 323
TCoolBar, 321
TCustomActionBar, 450
TCustomizeDlg, 442
TDatabase, 872, 955
TDataSource, 826, 904
TDBChart, 455
TDBCheckBox, 919
TDBComboBox, 919
TDBCtrlGrid, 920
TDBEdit, 917
TDBImage, 918
TDBListBox, 919
TDBMemo, 917
TDBRadioGroup, 920
TDBRichEdit, 918
TDBText, 916
TDDEClientConv, 564
TDDEClientItem, 570
TDDEServerItem, 572
TDrawGrid, 187
TEdit, 166
TFindDialog, 371
TFontDialog, 375
TForm, 208
TIBDataSet, 1009
TIBEvents, 1022
TIBExtract, 1021
TIBQuery, 1015
TIBStoredProc, 1015
TIBTransaction, 1016
TIBUpdateSQL, 1019
TImage, 267
TLabel, 270
TListView, 302
TMainMenu, 205
TMaskEdit, 166
TMediaPlayer, 749
TMonthCalendar, 340
TOpenDialog, 365
TOpenPictureDialog, 370
TPageControl, 285
TPageSetupDialog, 380
TPaintBox, 266
TPopupMenu, 206
TPrintDialog, 379
TPrinterSetupDialog, 378
TQuery, 893
TRadioButton, 175
TRadioGroup, 175
TRawSocket, 623
TReplaceDialog, 371
TRichEdit, 169
TSaveDialog, 365
TSavePictureDialog, 370
TSession, 873, 946
TShape, 266
TSpeedButton, 178
TStandardColorMap, 445
TStatusBar, 318
TStoredProc, 897
TStringGrid, 192
TTabControl, 281
TTable, 874
TTCPClient, 617
TTimer, 336
TToolBar, 313
TTrackBar, 326
TTreeView, 287
TUDPClient, 617
TUpdateSQL, 899
TUpDown, 177
TXPColorMap, 445
константа, 18
конструктор, 116, 420
координаты
логические, 273
мировые, 273
устройства, 273
кривые Безье, 254

M

массив, 27
Array, 27
FillChar, 70
Packed array, 28
SetLength, 28
Slice, 71, 250
динамический, 28
статический, 28
метод
абстрактный, 420
виртуальный, 416
динамический, 416
создание, 414
статический, 415
метод класса, 106, 116–117
многозадачность
вытесняющая, 499
кооперативная, 499
многокомпонентная модель объектов (COM), 624
многопоточность, 507
модальный результат, 173, 216
модуль
Chart, 455
Clipbrd, 552
ComObj, 652, 676
ComServ, 639
Contnrs, 155
CtlPanel, 728, 729
DB, 810, 906
DBActns, 432
DBCtrls, 906
DBGrids, 906
DBLogDlg, 363
DBTables, 946
ExtActns, 432
FileCtrl, 91, 93, 362
FMTBcd, 853
Forms, 208
IBCustomDataSet, 1009
IBDatabase, 997
IBEvents, 1022

IBExtract, 1021
IBUpdateSQL, 1019
Jpeg, 263, 371
Masks, 93
Math, 55, 59
MMSystem, 758
OleCtnrs, 696
Registry, 347
ShareMem, 482
ShellApi, 723, 734
ShlObj, 734
StdActns, 432
SvcMgr, 532
SyncObjs, 520
SysUtils, 387
TeEngine, 458, 469
Tlhelp32, 503
WinSvc, 532
монопольный доступ, 876
мьютекс (mutex), 524

Н

набор символов
ANSI, 21, 53
OEM, 21, 53
Unicode, 21, 26
набор строк, 150
наследование, 109
начало координат, 275
нормализация данных, 767

О

область видимости, 108
оболочка Windows, 733
обработка ошибок
ErrorCode, 393
GetLastError, 392
OleCheck, 741
Raise, 393
RaiseLastOSError, 393
SysErrorMessage, 392
try..except, 384
try..finally, 385
обслуживание ИС
Abort, 390
Assert, 391
объединение таблиц, 801
объект, 103
объект синхронизации
критическая секция, 518
мьютекс, 524
событие, 522
оконная процедура, 715
оконный элемент управления, 132
округление чисел
Abs, 55
Ceil, 56
Floor, 55
Frac, 55
Int, 55
Odd, 56
Round, 55
Trunc, 56
оператор
as, 106
begin..end, 38
case..of, 39
if..then, 38
if..then..else, 38
is, 106
with..do, 42
класса, 106
присваивания, 35
селектор, 39
составной, 38
условный, 38
цикл, 40
операция
арифметическая, 36
Div, 36
Mod, 36
логическая, 36
AND, 37
NOT, 37
OR, 37
XOR, 37
перетаскивания, 141, 144
побитовая
SHL, 38
SHR, 38
операции ввода-вывода, 72
ошибки, 87
отладка проекта, 390, 397
Assert, 70
Beep, 70
отладка системной службы, 549
ошибки ввода-вывода, 87

П

перегрузка методов, 49
переменная, 18
Application, 227
DateSeparator, 334
DecimalSeparator, 65
FileMode, 76, 81
Sessions, 947, 954
ThousandSeparator, 65
TimeSeparator, 334
перо, 241
поле
вычисляемое, 843
набора данных, 827
объекта, 103
процедурного типа, 425
синхронного просмотра, 845
составное, 821
порт TCP/IP, 606
поток, 507
приоритет, 511
почтовый слот (mailslot), 583
правила Кодда, 761
представление, 776, 792
преобразование даты/времени
DateTimeToFileDate, 331
DateTimeToStr, 332
DateTimeToSystemTime, 331
DateTimeToTimeStamp, 331
DateToStr, 332
DecodeDate, 331
DecodeDateFully, 332
DecodeDateMonthWeek, 332
DecodeDateTime, 332
DecodeTime, 332
EncodeDate, 331
EncodeDateMonthWeek, 332
EncodeDateTime, 331
EncodeTime, 331
FileDateToDateTime, 331
FloatToDateTime, 331
StrToDate, 332
StrToDateTime, 332
StrToTime, 332
SystemTimeToDateTime, 331
TimeStampToDateTime, 331
TimeToStr, 332
TryEncodeDate, 331
TryEncodeTime, 331

преобразование координат, 277
преобразование типов
Chr, 70
CurrToStr, 62
FloatToDecimal, 786
FloatToStr, 62
FloatToStrF, 62
FormatCurr, 62
FormatFloat, 62
FormatMaskText, 62
IntToHex, 62
IntToStr, 62
IntToStrDef, 62
StrToCurr, 62
StrToInt, 62
приложение, 227
процедура, 44
процедуры ADO
CreateUDLFile, 994
DataLinkDir, 995
GetDataLinkFiles, 995
GetProviderNames, 995
PromptDataLinkFile, 995
PromptDataSource, 996
процедуры управления программой
Abort, 69
Break, 69
Continue, 70
Exit, 70, 383
Halt, 70, 383
RunError, 70
процесс
дескриптор, 498
завершение, 501
создание, 499
пункт меню, 196

P

работа с дисками, 98
работа с памятью, 68
раздел
интерфейса, 49
реализации, 49
реестр Windows, 346
режим отображения
анизотропный, 280
изотропный, 278
текстовый, 274
реляционная модель, 761
ресурс формы, 51

C

свойство, 108
read, спецификатор, 404
write, спецификатор, 404
перечислимое, 407
простое, 406
публикация, 404
создание, 402
тип свойства, 406
типа массив, 410
типа множество, 407
типа объект, 408
секция
private, 51
public, 51
синхронный просмотр данных, 923
событие, 112, 125, 298
AfterCancel, 818
AfterClose, 811
AfterConnect, 957
AfterDatabaseDisconnect, 1011
AfterDelete, 818
AfterDisconnect, 957
AfterEdit, 817
AfterInsert, 817
AfterInstall, 542
AfterOpen, 811
AfterPost, 817
AfterScroll, 813
AfterTransactionEnd, 1011
AfterUninstall, 542
BeforeAction, 926
BeforeCancel, 818
BeforeClose, 811
BeforeConnect, 957
BeforeDatabaseDisconnect, 1011
BeforeDelete, 818
BeforeDisconnect, 957
BeforeEdit, 817
BeforeInsert, 817
BeforeOpen, 811
BeforePost, 817
BeforeScroll, 813
BeforeTransactionEnd, 1011
BeforeUninstall, 542
DatabaseFree, 1011
OnAccept, 615
OnActionExecute, 235, 434
OnActionUpdate, 235
OnActivate, 218, 235, 702, 728
OnAddition, 294
OnAdvancedCustomDraw, 301
OnAdvancedCustomDrawItem, 301
OnAdvancedDrawItem, 204
OnAfterAdd, 461
OnBandDrag, 325
OnBandInfo, 324
OnBandMove, 324
OnBandPaint, 325
OnBeforeAdd, 461
OnCalcFields, 824
OnCanClose, 370
OnCanResize, 220
OnCellClick, 915
OnChange, 155, 165, 186, 205, 285, 289, 322, 326, 341, 438, 572, 842
OnChanging, 155, 284, 289
OnClearValues, 462
OnClick, 136, 172, 197, 754
OnClickCheck, 185
OnClose, 219, 220, 270, 365, 443, 566
OnCloseQuery, 219
OnCloseUp, 186, 341, 925
OnColEnter, 915
OnColExit, 915
OnCalcFields, 820
OnCollapsed, 298
OnColumnMoved, 190, 915
OnCompare, 310
OnContinue, 543
OnControlCreated, 453
OnCreate, 217
OnCreateHandle, 609, 614
OnCreatePanelClass, 320
OnCustomDraw, 300, 312
OnCustomDrawItem, 300, 312
OnCustomDrawSubItem, 313
OnDataChange, 905
OnDblClick, 136

OnDeactivate, 219, 235, 702
OnDeleteError, 818, 824
OnDeletion, 295
OnDestroy, 219, 220
OnDestroyHandle, 610, 615
OnDialectDowngrade-Warning, 998
OnDockDrop, 146
OnDockOver, 146
OnDragDrop, 143
OnDragOver, 143
OnDrawCell, 191, 194
OnDrawItem, 181, 204, 311
OnDrawPanel, 321
OnDrawTab, 285
OnDropDown, 186, 341, 925
OnEditButtonClick, 915
OnEdited, 296, 304
OnEditError, 818, 825
OnEditing, 296, 303
OnEndDock, 147
OnEndDrag, 143
OnEnter, 135
OnEnterMenuLoop, 454
OnError, 611
OnEventAlert, 1022
OnException, 235, 396
OnExecute, 434, 438
OnExecuteMacro, 571
OnExit, 135
OnExpanded, 298
OnExpanding, 298
OnFilterRecord, 822
OnFolderChange, 370
OnGetAxisLabel, 472
OnGetControlClass, 452
OnGetEditMask, 191
OnGetEditText, 191
OnGetImageIndex, 285, 299
OnGetMonthInfo, 339
OnGetPopupClass, 454
OnGetSelectedIndex, 300
OnGetSiteInfo, 146
OnGetText, 842
OnGetThread, 617
OnHelp, 234
OnHide, 219
OnHint, 233, 235, 436
OnIdle, 235
OnIdleTimer, 1002
OnIncludeItem, 370
OnInquire, 729
OnKeyDown, 139
OnKeyPress, 140
OnKeyUp, 139
OnLastRelease, 640
OnLogin, 958, 1001
OnMeasureItem, 182, 204
OnMessage, 235, 560
OnMinimize, 234, 235
OnMouseDown, 136
OnMouseEnter, 270
OnMouseLeave, 270
OnMouseMove, 137
OnMouseUp, 136
OnMouseWheel, 138
OnMouseWheelDown, 139
OnMouseWheelUp, 139
OnNewInquire, 729
OnNewRecord, 817
OnNotify, 755
OnObjectMove, 702
OnOpen, 270, 566
OnPageChange, 474
OnPaint, 218, 266
OnPaintPanel, 922
OnPassword, 953
OnPause, 543
OnPokeData, 572
OnPopup, 207
OnPostClick, 754
OnPostError, 818, 825
OnProgress, 256
OnProtectChange, 171
OnReceive, 611
OnReplace, 374
OnResize, 218, 220, 702
OnResizeRequest, 171
OnRestore, 234, 235
OnRowMoved, 190
OnSaveClipboard, 172
OnSelect, 186
OnSelectCell, 190
OnSelectionChange, 171, 370
OnSend, 611
OnSetEditText, 191
OnSetText, 842
OnShortCut, 221, 235
OnShow, 218, 365, 442
OnShowHint, 235
OnShutdown, 543
OnSQLChanging, 1005
OnStart, 270, 543
OnStartDock, 146
OnStartDrag, 143
OnStartup, 947
OnStartWParms, 728
OnStateChange, 438, 905
OnStop, 270, 543, 729
OnTerminate, 510
OnTimer, 336
OnTitleClick, 915
OnTopLeftChanged, 189
OnTypeChange, 370
OnUnDock, 147
OnUndoZoom, 474
OnUpdate, 435, 438
OnUpdateData, 905
OnUpdateError, 871, 1011
OnUpdateRecord, 871, 1011, 1014
OnUserInput, 340
OnValidate, 842
OnZoom, 473
TransactionFree, 1011
клавиатуры, 139
мыши, 136
создание, 423
соглашение о вызовах
cdecl, 487
pascal, 487
register, 487
safecall, 487
stdcall, 487
сообщение
NM_CUSTOMDRAW, 301
WM_CHAR, 714
WM_COMMAND, 718
WM_COPYDATA, 563
WM_MOUSEMOVE, 425
WM_PAINT, 148, 713
WM_PALETTCHANGED, 131
WM_QUIT, 714
WM_TIMER, 335
составные поля, 821
состояние набора данных, 814
сравнение строк
ANSI, 61
PChar, 67
страница блокнота, 286
структура модуля, 50

Т

тип данных
AnsiString, 25, 61
Boolean, 22
Byte, 21
ByteBool, 22
Char, 22
Comp, 24
Currency, 24
Extended, 24
HRESULT, 642
Int64, 21
Integer, 21
LongBool, 22
ModalResult, 173
PAnsiChar, 34
PChar, 34
PWideChar, 34
Real48, 24
Shortint, 21
ShortString, 25
Single, 24
Smallint, 21
String, 25
TAffectRecords, 981
TAlign, 127
TAnchorKind, 127
TAnimationStyle, 453
TBevelShape, 267
TBitBtnKind, 174
TBorderIcon, 210
TBrushStyle, 243
TButtonLayout, 174
TButtonStyle, 175
TCalDayOfWeek, 338
TCheckBoxState, 176, 185
TClassInstancing, 634
TCloseAction, 220
TColor, 241
TColorDialogOptions, 377
TComboBoxStyle, 183
TCommandType, 965
TCommonAVI, 268
TConnectMode, 970
TControlState, 131
TControlStyle, 131
TCursorOptions, 967
TCustomDrawStage, 301
TDataSetState, 814
TDate, 329
TDateTime, 94, 328
TDBGridOptions, 907
TDisplayCode, 313
TDragKind, 141, 145
TDragMode, 141
TDuplicates, 155
TEditCharCase, 186
TFieldAttribute, 885
TFieldKind, 829
TFieldType, 830
TFillStyle, 253
TFindOptions, 371
TFontDialogDevice, 375
TFontDialogOptions, 376
TFontPitch, 244
TFontStyle, 244
TFormBorderStyle, 210
TFormState, 211
TFormStyle, 208
TGridDrawState, 915
TGridOptions, 189
THitTest, 284
TileMode, 227
TIsolationLevel, 977, 978
TJPEGPerformance, 264
TJPEGPixelFormat, 264
TJPEGQualityRange, 263
TJPEGScale, 264
TListBoxStyle, 181
TMenuAnimation, 207
TMenuBreak, 202
TMenuItemAutoFlag, 196, 203
TModalResult, 216
TMouseButton, 137
TMsgDlgBtn, 357
TMsgDlgType, 357
TMultiSelectStyle, 291
TNodeAttachMode, 297
TNumberingStyle, 170
TOpenOptions, 366
TOwnerDrawState, 181, 204
TPageSetupDialogOptions, 381
TPenMode, 243
TPenStyle, 242
TPixelFormat, 259
TPosition, 209
TPrintDialogOptions, 379
TProgressStage, 256
TRecordStatusSet, 982
TScrollStyle, 168
TSearchType, 171
TSeriesMarksStyle, 467
TServerSocketBlockMode, 613
TShapeType, 267
TShiftState, 137
TSocketDomain, 608
TSocketProtocol, 609
TSortType, 310
TTabPosition, 282
TTabStyle, 281
TThreadingModel, 635
TTime, 329
TToolButtonStyle, 316
TTransIsolation, 960
TTransparentMode, 260
TUpdateStatus, 818
TViewStyle, 304
TWaitResult, 523
TWindowState, 210
WideChar, 22
WideString, 25
Word, 21
WordBool, 22
действительное число, 24
логический, 22
массив, 27
множество, 31
поддиапазон, 23
простой, 20
символьный, 21
строковый, 25
указатель, 33
целое число, 21
типы данных SQL, 784
транзакция, 778
изоляция, 779

У

уведомление
EMPNotify, 754
EMPPostNotify, 754
ENavClick, 926
TActionEvent, 434, 438
TActivateEvent, 728
TAdvancedMenuDrawItemEvent, 204
TAxisOnGetLabel, 472
TBandDragEvent, 325
TBandInfoEvent, 324
TBandMoveEvent, 324
TBandPaintEvent, 325
TCanResizeEvent, 220
TCloseEvent, 220
TCloseQueryEvent, 219, 370

TConnectErrorEvent, 972, 977
TDatabaseLoginEvent, 958
TDataChangeEvent, 905
TDataSetErrorEvent, 818, 825
TDataSetNotifyEvent, 811
TDBGridClickEvent, 915
TDockDropEvent, 146
TDockOverEvent, 146
TDragDropEvent, 143
TDragOverEvent, 143
TDrawCellEvent, 191
TDrawColumnCellEvent, 915
TDrawItemEvent, 181
TDrawPanelEvent, 321
TDrawTabEvent, 285
TDTParseInputEvent, 340
TEndDragEvent, 143
TEventAlert, 1023
TExceptionEvent, 396
TExecuteCompleteEvent, 976
TExitMenuEvent, 454
TFieldGetTextEvent, 842
TFieldSetTextEvent, 842
TFilterRecordEvent, 822
TGetControlClassEvent, 452
TGetEditEvent, 191
TGetPopupClassEvent, 454
TGetSiteInfoEvent, 146
TGetThreadEvent, 617
THelpEvent, 234
THintEvent, 436
TIBDatabaseLoginEvent, 1001
TIBUpdateErrorEvent, 1014
TIBUpdateRecordEvent, 1014
TIncludeItemEvent, 370
TInquireEvent, 729
TItemCreatedEvent, 453
TKeyEvent, 139, 424
TKeyPressEvent, 140
TLastReleaseEvent, 640
TLVCompareEvent, 310
TLVCustomDrawEvent, 312
TLVCustomDrawItemEvent, 312
TLVCustomDrawSubItemEvent, 313
TLVDrawItemEvent, 312
TLVEditedEvent, 304
TLVEditingEvent, 304
TMacroEvent, 571
TMeasureItemEvent, 182
TMenuChangeEvent, 205
TMenuDrawItemEvent, 204
TMenuMeasureItemEvent, 204
TMenuPopupEvent, 454
TMessageEvent, 560
TMouseEvent, 136, 424
TMouseMoveEvent, 137
TMouseWheelEvent, 138
TMouseWheelUpDownEvent, 139
TMovedEvent, 190, 915
TNewInquireEvent, 729
TNoifyEvent, 423
TObjectMoveEvent, 702
TOnBeginTransComplete, 977
TOnGetMonthInfoEvent, 339
TPaintPanelEvent, 922
TPasswordEvent, 953
TRichEditProtectChange, 171
TRichEditResizeEvent, 171
TRichEditSaveClipboard, 172
TSBCreatePanelClassEvent, 320
TSelectCellEvent, 190
TSeriesClick, 466
TSeriesOnAfterAdd, 461
TSeriesOnBeforeAdd, 461
TSeriesOnClear, 462
TServiceEvent, 542
TSetEditEvent, 191
TShortCutEvent, 221
TSocketAcceptEvent, 615
TSocketDataEvent, 611
TSocketErrorEvent, 611
TSocketNotifyEvent, 609
TStartDockEvent, 146
TStartDragEvent, 143
TStartWParmsEvent, 728
TStopEvent, 729
TTabChangingEvent, 284
TTabGetImageEvent, 285
TTVAdvancedCustomDrawEvent, 301
TTVChangedEvent, 289
TTVChangingEvent, 289
TTVCollapsingEvent, 298
TTVCustomDrawEvent, 300
TTVCustomDrawItemEvent, 300, 301
TTVEditedEvent, 296
TTVEditingEvent, 296
TTVExpandedEvent, 294, 295, 298
TTVExpandingEvent, 298
TUnDockEvent, 147
TUpdateErrorEvent, 871
TUpdateRecordEvent, 871
TWillConnectEvent, 971
TWillExecuteEvent, 976
удаление пробелов, 61
указатель, 33, 133
  неопределенный, 34
управление каталогами, 97
условный оператор
  if..then, 38
  if..then..else, 38
утилита
  TDUMP, 495
  редактор библиотеки типов, 648

Ф

фабрика класса
  TComObjectFactory, 634
файл
  *.bmp, 258
  *.bpg, 51
  *.bpl, 51
  *.cpl, 730
  *.dcr, 427
  *.dcu, 51
  *.dfm, 51, 400, 413
  *.dib, 258
  *.dll, 479
  *.dof, 51
  *.dpk, 51
  *.dpr, 51
  *.dsk, 51
  *.emf, 261
  *.idl, 648
  *.jpg, 263
  *.pas, 51

*.rec, 495
*.res, 51
*.tlb, 648
*.udl, 968
*.wmf, 261
borlndmm.dll, 69, 483
gds32.dll, 997
атрибуты, 94
дата и время создания, 93
двоичный, 85
имя файла, 92
инициализация, 342
размер, 95
текстовый, 75
типизированный, 80
файловая переменная, 73
файлы проекта Delphi, 51
фокус ввода, 135
форматирование даты/времени
DateTimeToStr, 332
DateTimeToString, 332
DateToStr, 332
FormatDateTime, 332
TimeToStr, 332
форматирование строк - ANSI, 61
Format, 63, 64
форматирование строк PChar, 66
форматирование текста, 170
функции автоматизации
CreateOleObject, 652
GetActiveOleObject, 653
функции вхождения в диапазон
High, 23
Low, 23
функции даты/времени
CurrentYear, 329
Date, 329
DateOf, 330
DateTimeToSQLTimeStamp, 787
DayOfTheWeek, 329
DayOfWeek, 329
DaysBetween, 330
HoursBetween, 330
IncDay, 329
IncMonth, 330
IsLeapYear, 330
MilliSecondsBetween, 330
MinutesBetween, 330
MonthsBetween, 330
Now, 329
ReplaceDate, 330
ReplaceTime, 330
SecondsBetween, 330
Time, 329
TimeOf, 330
WeeksBetween, 330
YearsBetween, 330
функции для работы с памятью
AllocMem, 68
CompareMem, 69
Dispose, 68, 933
Finalize, 29
FreeMem, 68
GetHeapStatus, 68
GetMem, 68
GetMemoryManager, 68
GlobalAlloc, 557
GlobalUnLock, 557
IsMemoryManagerSet, 69
New, 68, 932
ReallocMem, 69
SetMemoryManager, 69
SysFreeMem, 69
SysGetMem, 69
SysReallocMem, 69
функции для работы с указателями
Assigned, 70
функции для работы с файлами
Append, 76
AssignFile, 74
BlockRead, 85
BlockWrite, 85
ChangeFileExt, 92
ChDir, 97
CloseFile, 74
CreateDir, 97
DateTimeToFileDate, 94
DeleteFile, 91
DirectoryExists, 97
DiskFree, 98
DiskSize, 99
Eof, 78
EoLn, 79
Erase, 84
ExcludeTrailingBackslash, 92
ExpandFileName, 92
ExpandUNCFileName, 92
ExtractFileDir, 93
ExtractFileDrive, 93
ExtractFileExt, 93
ExtractFileName, 93
ExtractFilePath, 93
ExtractRelativePath, 93
ExtractShortPathName, 93
FileAge, 93
FileClose, 89
FileCreate, 89
FileDateToDateTime, 94
FileExists, 91
FileGetAttr, 94
FileGetDate, 89, 94
FileOpen, 89
FilePos, 81
FileRead, 89
FileSeek, 89
FileSetAttr, 94
FileSetDate, 89, 94
FileSetReadOnly, 95
FileSize, 82
FileWrite, 89
FindClose, 96, 193
FindFirst, 96, 193
FindNext, 96, 193
FindSearch, 97
Flush, 78
ForceDirectories, 98
GetCurrentDir, 97
GetDir, 97
IncludeTrailingBackslash, 92
IOResult, 88
IsPathDelimiter, 93
MatchesMask, 93
MinimizeName, 93
MoveFile, 98
MoveFileEx, 98
ProcessPath, 93
Read, 78
ReadLn, 78
RemoveDir, 98
Rename, 84
Reset, 76
Rewrite, 76
Seek, 81
SeekEof, 79
SeekEoLn, 79
SetCurrentDir, 97
Write, 78
WriteLn, 78
функции математические, 55
Exp, 56

Frexp, 56
IntPower, 56
Ldexp, 56
Ln, 56
LnXP1, 56
Log10, 56
Log2, 56
LogN, 56
Power, 56
Sqr, 56
Sqrt, 56
функции ожидания, 520
функции порядковые
Chr, 23
Dec, 23
Inc, 23
Odd, 23
Ord, 23
Pred, 23
Succ, 23
функции статистические
Max, 59
MaxIntValue, 59
MaxValue, 59
Mean, 60
MeanAndStdDev, 60
Min, 59
MinIntValue, 59
MinValue, 60
Norm, 60
PopnVariance, 60
StdDev, 60
Sum, 60
SumAndSquares, 60
SumInt, 60
SumOfSquares, 60
TotalVariance, 60
Variance, 60
функции строк ANSI
AdjustLineBreaks, 63
AnsiQuotedStr, 63
AnsiReverseString, 63
AnsiSameText, 61
CompareStr, 61
CompareText, 61
Concat, 63
Copy, 63
Delete, 63
FmtStr, 61
Format, 61
Insert, 63
Length, 63
LowerCase, 61
Pos, 63
QuotedStr, 62
Trim, 61
TrimLeft, 61
TrimRight, 61
UpperCase, 61
функции строк PChar
FormatBuf, 67
StrAlloc, 65
StrBufSize, 65
StrCat, 66
StrComp, 67
StrCopy, 66
StrDispose, 65
StrECopy, 66
StrEnd, 67
StrFmt, 66
StrIComp, 67
StrIPos, 67
StrIScan, 67
StrLCat, 66
StrLComp, 67
StrLCopy, 66
StrLen, 67
StrLFmt, 66
StrLIComp, 67
StrLower, 66
StrMove, 66
StrNew, 65
StrPCopy, 66
StrPLCopy, 66
StrRScan, 67
StrUpper, 66
функции тригонометрические
ArcCos, 57
ArcCosh, 57
ArcSin, 57
ArcSinh, 57
ArcTan, 57
ArcTan2, 57
ArcTanh, 57
Cos, 57
Cosh, 57
Cotan, 57
CycleToRad, 58
DegToRad, 58
GradToRad, 58
Hypot, 58
Pi, 57
RadToCycle, 58
RadToDeg, 58
RadToGrad, 58
Sin, 57
SinCos, 57
Sinh, 57
Tan, 57
функции финансовые
DoubleDecliningBalance, 58
FutureValue, 58
InterestPayment, 58
InterestRate, 58
InternalRateOfReturn, 59
NetPresentValue, 59
NumberOfPeriods, 59
Payment, 59
PeriodPayment, 59
PresentValue, 59
SLNDepreciation, 59
SYDDepreciation, 59
функция проверки вхождения значения в диапазон
EnsureRange, 57
High, 57
InRange, 56
Low, 57
VarInRange, 57

Х–Ц

холст, 246
цикл
break, 42
continue, 42
for..do, 40
for..downto, 40
repeat..until, 41
while..do, 40
обработки сообщений, 713
прерывание, 41
с параметром, 40
с постусловием, 41
с предусловием, 40

Ш–Я

шаблон маски, 166
шрифт, 244
экран, 235
элемент управления, 114, 125
язык определения данных, 788

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-074-X, название «Delphi. Профессиональное программирование» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.