Михаил Фленов

Библия Delphi

3-е издание

Программирование в Delphi от А до Я

Динамические библиотеки

Подготовка и вывод документов на печать

Создание локальных, клиент-серверных и трехуровневых баз данных

Практические рекомендации и примеры

Большое количество дополнительной информации на CD

+CD

bhv

Михаил Фленов

Библия Delphi

3-е издание

Санкт-Петербург

«БХВ-Петербург»

2011

УДК 681.3.068+800.92Delphi
ББК 32.973.26-018.1
Ф69

Фленов М. Е.

Ф69 Библия Delphi. — 3-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2011. — 688 с.: ил. + CD-ROM

ISBN 978-5-9775-0667-0

Книга посвящена программированию на языке Delphi от самых основ до примеров построения конкретных приложений. Подробно описывается логика выполнения каждого участка кода, чтобы читатель смог использовать эти знания при решении собственных задач. Книга содержит большое количество примеров практического программирования; некоторые из них вынесены в качестве дополнительной информации на прилагаемый компакт-диск. В третьем издании материал исправлен и переработан с учетом новых возможностей пакета. Компакт-диск содержит исходные коды программ, дополнительную справочную информацию, а также готовые изображения и компоненты.

Для программистов

УДК 681.3.068+800.92Delphi
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Алексей Семенов*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 29.11.10.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 55,47.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

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

ISBN 978-5-9775-0667-0

Оглавление

Введение **1**

Замечания к третьему изданию 2

Глава 1. Основные принципы работы компьютера **5**

1.1. Двоичная система работы процессора 5
1.2. Машинный язык 10
1.3. История языков программирования 11
1.4. Исполнение машинных команд 15

Глава 2. Машинная математика **17**

2.1. Основы машинной математики 17
2.2. Блок-схемы 20
2.3. Машинная логика и циклы 22
2.4. Программирование машинной логики 24

Глава 3. Начальные сведения о Delphi **27**

3.1. Оболочка Delphi 27
3.2. Главное меню 30
3.3. Настройка 31

Глава 4. Визуальная модель Delphi **39**

4.1. Процедурное программирование 39
4.2. Объектно-ориентированное программирование 43
4.3. Компонентная модель 48
4.4. Наследственность 49
4.5. Полиморфизм 50
4.6. Инкапсуляция 51

Глава 5. Основы языка программирования Delphi **53**

5.1. "Hello World", или Из чего состоит проект 53
5.2. Язык программирования Delphi 62
5.3. Типы данных в Delphi 68

5.3.1. Целочисленные типы данных ... 68
5.3.2. Вещественные типы данных ... 69
5.3.3. Символьные типы данных ... 70
5.3.4. Булевы типы ... 75
5.3.5. Массивы ... 77
5.3.6. Странный *PChar* ... 78
5.3.7. Константы ... 79
5.3.8. Всемогущий *Variant* ... 80
5.4. Процедуры и функции в Delphi ... 81
5.5. Рекурсивный вызов процедур ... 89
5.6. Встроенные процедуры ... 91
5.7. Возврат значений через параметры ... 92
5.8. Перегрузка ... 93
5.9. Методы объектов ... 94
5.10. Наследование объектов ... 95

Глава 6. Работа с компонентами ... 99

6.1. Основная форма и ее свойства ... 99
6.2. Событийная модель Windows ... 108
6.3. События главной формы ... 110
6.4. Палитра компонентов ... 111

Глава 7. Палитра компонентов *Standard* ... 113

7.1. Кнопка (*TButton*) ... 113
7.2. Изменение свойств кнопки (логические операции) ... 116
7.3. Надписи (*TLabel*) ... 120
7.4. Строки ввода (*TEdit*) ... 121
7.5. Многострочное поле ввода (*TMemo*) ... 122
7.6. Класс *TStrings* ... 126
7.6.1. Свойства *TStrings* ... 126
7.6.2. Методы объекта *TStrings* ... 127
7.7. Компонент *CheckBox* ... 127
7.8. Панели (*TPanel*) ... 128
7.9. Кнопки выбора *TRadioButton* ... 130
7.10. Списки выбора (*TListBox*) ... 131
7.11. Ниспадающие списки (*TComboBox*) ... 133
7.12. Полосы прокрутки (*TScrollBar*) ... 134
7.13. Группировка объектов (*TGroupBox*) ... 135
7.14. Группа компонентов *RadioButton* (*TRadioGroup*) ... 135
7.15. Список действий *TActionList* ... 136

Глава 8. Учимся программировать ... 139

8.1. Циклы *for...to...do* ... 139
8.2. Циклы *while* ... 142

8.3. Циклы *Repeat* ... 144
8.4. Управление циклами ... 145
8.5. Логические операторы ... 149
8.6. Работа со строками ... 152
8.6.1. Функция *Length* ... 152
8.6.2. Функция *Copy* ... 152
8.6.3. Функция *Delete* ... 153
8.6.4. Функция *Pos* ... 153
8.6.5. Функция *Insert* ... 154
8.7. Исключительные ситуации ... 154
8.8. Классы исключительных ситуаций ... 157

Глава 9. Создание рабочих приложений ... 161

9.1. Создание главного меню программы ... 161
9.2. Создание дочерних окон ... 165
9.3. Модальные и немодальные окна ... 168
9.4. Обмен данными между формами ... 169
9.5. Многодокументные MDI-окна ... 171
9.6. Инициализация окон ... 174
9.7. Фреймы ... 179

Глава 10. Основные приемы программирования ... 181

10.1. Работа с массивами ... 181
10.2. Многомерные массивы ... 186
10.3. Работа с файлами ... 187
10.4. Работа с текстовыми файлами ... 191
10.5. Приведение типов ... 194
10.5.1. Преобразование целых чисел в строку и обратно ... 195
10.5.2. Преобразование даты в строку и обратно ... 196
10.5.3. Преобразование вещественных чисел ... 197
10.6. Преобразование совместимых типов (преобразование строк) ... 199
10.6.1. Приведение классов ... 199
10.7. Указатели ... 201
10.8. Структуры, записи ... 204
10.9. Храним структуры в динамической памяти ... 208
10.10. Поиск файлов ... 210
10.11. Работа с системным реестром ... 214
10.12. Множества ... 220
10.13. Потоки ... 222
10.14. Концентрация на объекте ... 223

Глава 11. Обзор дополнительных компонентов Delphi ... 225

11.1. Дополнительные кнопки Delphi (*TSpeedButton* и *TBitBtn*) ... 225
11.2. Самостоятельная подготовка иконок ... 229

11.3. Маскированная строка ввода (*TMaskEdit*) ... 230
11.4. Сеточки (*TStringGrid, TDrawGrid*) ... 231
11.5. Компоненты *TImage, TShape, TBevel* ... 237
11.6. Панель с полосами прокрутки (*TScrollBox*) ... 240
11.7. Маркированный список (*TCheckListBox*) ... 241
11.8. Полоса разделения (*TSplitter*) ... 242
11.9. Многострочный текст (*TStaticText*) ... 243
11.10. Редактор параметров (*TValueListEditor*) ... 243
11.11. Набор вкладок (*TTabControl*) ... 246
11.12. Набор страниц (*TPageControl*) ... 250
11.13. Набор картинок (*TImageList*) ... 252
11.14. Ползунки (*TTrackBar*) ... 253
11.15. Индикация процесса (*TProgressBar*) ... 254
11.16. Простейшая анимация (*TAnimate*) ... 257
11.17. Ниспадающий список выбора даты (*TDateTimePicker*) ... 258
11.18. Календарь (*TMonthCalendar*) ... 258
11.19. Дерево элементов (*TTreeView*) ... 259
11.20. Профессиональное использование компонента *TreeView* ... 264
11.21. Список элементов (*TListView*) ... 268
11.22. Простейший файловый менеджер ... 269
11.23. Улучшенный файловый менеджер (с возможностью запуска файлов) ... 279
11.24. Подсказки для чайников (*TStatusBar*) ... 281
11.25. Панель инструментов (*TToolBar* и *TControlBar*) ... 283
11.26. Перемещаемые панели и меню в стиле MS (Docking) ... 285
11.27. Меню и панели на основе Action ... 288
11.28. Всплывающее меню на основе Action ... 292
11.29. Практика использования Action ... 292
11.30. События приложения ... 297
11.31. Поле ввода с меткой ... 297
11.32. Коробка с цветом ... 298
11.33. Иконка в SystemTray ... 298

Глава 12. Графические возможности Delphi ... 301

12.1. Графическая система Windows ... 301
12.2. Первый пример работы с графикой ... 303
12.3. Свойства карандаша ... 304
12.4. Свойства кисти ... 307
12.5. Работа с текстом в графическом режиме ... 311
12.6. Вывод текста под углом ... 313
12.7. Работа с цветом ... 318
12.8. Методы объекта *TCanvas* ... 321
12.8.1. *Pixels* ... 321
12.8.2. *TextWidth* и *TextHeight* ... 322
12.8.3. *Arc* ... 322

12.8.4. *CopyRect* ... 322
12.8.5. *Draw* ... 323
12.8.6. *Ellipse* ... 324
12.8.7. *FillRect* ... 324
12.8.8. *FloodFill* ... 324
12.9. Компонент работы с графическими файлами (*TImage*) ... 324
12.10. Рисование на стандартных компонентах ... 328
12.11. Работа с экраном ... 332
12.12. Режимы рисования ... 334
12.13. Сканирование данных ... 338

Глава 13. Печать в Delphi ... 343

13.1. Объект *TPrinter* ... 343
13.2. Получение информации об установленном принтере ... 347
13.3. Текстовая печать ... 350
13.4. Печать содержимого формы ... 351
13.5. Вывод на печать изображения ... 356
13.6. Еще немного о печати ... 358
13.7. Это интересно ... 360

Глава 14. Delphi и базы данных ... 365

14.1. Теория реляционных баз данных ... 366
14.1.1. Локальные базы данных ... 367
14.1.2. Delphi и базы данных ... 369
14.2. Создание первой базы данных Access ... 370
14.3. Пример работы с базами данных ... 373
14.3.1. Свойства компонента *TADOTable* ... 377
14.3.2. Методы компонента *TADOTable* ... 379
14.4. Управление отображением данных ... 380
14.5. Поисковые поля ... 386
14.6. Улучшенный пример с поисковыми полями ... 393
14.7. Сортировка ... 395
14.8. Фильтрация данных ... 397
14.9. Язык запросов SQL ... 401
14.10. Связанные таблицы ... 406
14.11. Вычисляемые поля ... 412
14.12. Цветные сетки *DBGrid* ... 415
14.13. Подключение к базе данных во время выполнения программы ... 418
14.14. Расширения ADO ... 420
14.15. Обработка базы данных ... 426
14.16. Бинарные данные ... 429
14.17. События наборов данных ... 433
14.18. События *DataSource* ... 435
14.19. Позиционирование ... 436

Глава 15. Создание отчетности ... 439

15.1. Создание отчетности в Excel ... 440
15.2. Отчетность в Word ... 448
15.3. Отчетность в Quick Reports ... 449
15.4. Печать таблиц с помощью Quick Reports ... 455
15.5. Печать связанных таблиц ... 456
15.6. Дополнительные возможности ... 457

Глава 16. Работа с DBF, Paradox, XML и клиент-серверными базами данных ... 459

16.1. Создание таблицы Paradox ... 459
16.2. Русификация таблиц Paradox и DBF ... 465
16.3. Быстрый поиск ... 466
16.4. Создание псевдонимов ... 467
16.5. Работа с XML-таблицами ... 470
16.6. Теория клиент-серверных баз данных ... 471
16.7. Пример работы с SQL Server ... 473
16.8. Многоуровневые приложения для баз данных ... 477
16.8.1. Реализация сервера бизнес-логики ... 479
16.8.2. Клиент для бизнес-логики ... 482

Глава 17. Потоки ... 487

17.1. Теория потоков ... 487
17.2. Простейший поток ... 489
17.3. Дополнительные возможности потоков ... 493
17.4. Подробней о синхронизации ... 494
17.5. Объект события *Event* ... 496
17.6. Критические секции ... 500

Глава 18. Динамически компонуемые библиотеки ... 503

18.1. Что такое DLL ... 503
18.1.1. Решение № 1 ... 503
18.1.2. Проблема № 1 ... 504
18.1.3. Проблема № 2 ... 504
18.1.4. Решение № 2 ... 505
18.1.5. Из чего сделан Windows ... 506
18.2. Простой пример создания DLL ... 508
18.3. Замечания по использованию библиотек ... 511
18.4. Хранения формы в динамических библиотеках ... 512
18.5. Немодальные окна в динамических библиотеках ... 515
18.6. Явная загрузка библиотек ... 518
18.7. Точка входа ... 520
18.8. Вызов из библиотек процедур основной программы ... 522

Глава 19. Разработка собственных компонентов ... 525

19.1. Пакеты ... 526
19.2. Подготовка к созданию компонента ... 532
19.3. Создание первого компонента ... 534
19.4. Создание иконки компонента ... 543
19.5. События в компонентах ... 545
19.6. Когда создавать компоненты ... 547

Глава 20. Технология OLE ... 549

20.1. Теория OLE ... 549
20.2. OLE-контейнер ... 552
20.3. Создание собственного окна вставки OLE-объекта ... 556

Глава 21. Компоненты ActiveX ... 561

21.1. Использование Internet Explorer ... 561
21.2. Пример создания ActiveX-форм ... 566
21.3. Создание компонентов ActiveX ... 570

Глава 22. Технология COM ... 577

22.1. Модель COM ... 577
22.2. Информация о COM ... 578
22.3. Интерфейс и реализация ... 579

Глава 23. Буфер обмена ... 583

23.1. Буфер обмена и стандартные компоненты Delphi ... 583
23.2. Объект *Clipboard* ... 584
23.3. Картинки и буфер обмена ... 586
23.4. Создание собственного формата для работы с буфером ... 591

Глава 24. Дополнительная информация ... 599

24.1. Тестирование и отладка ... 599
24.2. Работа с редактором ... 606
  24.2.1. Закладки ... 606
  24.2.2. Копирование строк ... 607
  24.2.3. Code Explorer ... 608
  24.2.4. Редактор кода ... 609
24.3. Создание программ инсталляции ... 609
24.4. Как писать и распространять программы ... 620

Глава 25. Практика ... 625

25.1. Создание ScreenSaver ... 625
25.2. Компоненты в runtime ... 630

25.3. Тест на прочность 635
25.4. Сохранение и загрузка теста 646
25.5. Тестер 650

ПРИЛОЖЕНИЯ 657

Приложение 1. Основные классы библиотеки VCL 659

П1.1. *TObject* 659
П1.2. *TPersistent* 659
П1.3. *TComponent* 660
П1.4. *TControl* 660
П1.5. *TWinControl* 663
П1.6. *TApplication* 665

Приложение 2. Описание компакт-диска 667

Литература 669

Предметный указатель 671

Введение

Данная книга посвящена одному из наиболее популярных в нашей стране и перспективному во всем мире языку программирования Delphi. Она предназначена для программистов всех уровней, от начинающего до опытного. Как показывает практика, большинство людей научились программированию по книгам. Однако далеко не все из этих книг объясняют принципиальные основы работы Windows и компьютера в целом. Отсутствие базовых знаний в этой области не позволяет писать эффективные программы.

Эта книга может научить многому. Однако без самостоятельного стремления совершенствоваться в данной области вы не сможете самостоятельно писать "хорошие" программы. В этой книге будут рассматриваться различные методы, некоторые шаблоны и приемы программирования на языке Delphi, однако описать абсолютно все, как вы понимаете, здесь просто невозможно. Программирование — это такая область, в которой требуется постоянное обучение. В связи с этим нельзя останавливаться на достигнутом, прочитав только одну книгу. Нужно постоянно совершенствоваться и обучатся.

За что я люблю компьютеры, так это за то, что они являются безграничным источником знаний, которые нельзя изучить в полном объеме. Даже если вы сможете узнать все, пока вы будете обучаться, появятся новые технологии. Именно поэтому нет человека, который знал бы все. Я тоже не знаю, но люблю изучать что-то новое. Но если даже представить себе, что я смогу изучить все, то лично мне жить станет скучно.

Прежде чем приступить к изучению самой книги, необходимо сделать несколько замечаний. Первое из них касается терминологии. В тексте часто будет использоваться выражение "Язык программирования Delphi". Многие утверждают, что Delphi — это среда разработки, которая использует язык программирования Pascal (Паскаль). В принципе, здесь не утверждается, что это ошибка. И все же в Delphi от старого Паскаля осталось очень мало, поэтому я считаю, что это не просто среда разработки, а самостоятельный язык программирования. Это лично мое мнение как автора, и вы можете с ним соглашаться или нет. Но даже разработчик среды разработки Delphi уже тоже воспринимает Delphi как самостоятельный язык.

Теперь о содержимом книги. В ней сделана попытка представить изучаемый материал таким образом, чтобы было понятно даже человеку, который только недавно познакомился с компьютером. Возможно, опытным программистам какие-то части

читать будет скучно, но даже здесь будут описываться достаточно специфичные вещи, среди которых можно найти для себя довольно много полезного. Поверьте, это действительно так и связано с тем, что большинство книг по данной проблематике упускают из виду некоторые очень важные тонкости, которые желательно знать для понимания принципа работы программ. Без этого понимания тяжело двигаться дальше, и любые новые технологии будут казаться тяжелыми и сложными.

Прежде чем приступить к чтению книги, учтите один совет. Книгу желательно читать полностью, от начала и до конца, потому что материал излагается постепенно, и некоторые вещи могут быть непонятны, если что-то пропустить вначале. Как только вы почувствуете, что получили достаточно знаний и способны самостоятельно писать хотя бы простейшие программы, можете сделать единственный скачок на *главу 25*. В ней дается материал, касающийся отладки приложений, потому что при самостоятельном написании программ всегда появляются ошибки или опечатки. Мы люди, и нам свойственно ошибаться. Эта глава объясняет, как находить такие ошибки. В ней вы также узнаете некоторые приемы по работе с редактором кода, которые могут пригодиться в будущем при программировании собственных приложений, да и при работе с примерами, которые представлены в этой книге. После прочтения этой главы можно вернуться к той, на которой вы остановились ранее, и продолжить чтение книги уже без каких-либо скачков. Иначе какой-то важный момент может быть упущен, и нагнать потом будет очень тяжело, потому что вы можете не заметить, что что-то упустили.

> **ВНИМАНИЕ.** Я не читал ни одной книги по Delphi на русском языке, только англоязычные материалы. Единственная книга, которую я видел (я даже не прочитал ее полностью, а просмотрел несколько глав) была по Borland Pascal. Это было в 1994 году, поэтому я даже не помню ее названия. Именно поэтому некоторые мои термины могут отличаться от таких же в другой литературе.

И последнее, некоторые термины, встречающиеся в книге, могут отличаться от аналогичных, которые изложены в другой технической литературе, относящейся к данному вопросу. Это связано с особенностями перевода англоязычного текста на русский язык. В любом случае терминология, которая приводится в книге, делает ее намного проще и понятней как начинающим, так и опытным программистам.

Замечания к третьему изданию

Если вы читали предыдущий вариант книги, вам также будет полезно прочитать эту книгу, потому что данный вариант переработан полностью от начала и до конца. Помимо этого, на компакт-диске, прилагаемом к данной книге, представлено много новой и полезной документации, которая поможет вам двигаться дальше после прочтения книги.

Некоторая информация была перенесена из книги на компакт-диск в виде электронных документов. Этим я сэкономил место, чтобы дать больше полезной информации на страницах книги. Если бы всю эту информацию превратить в печатные страницы, то книга стала бы как минимум в два раза толще. Пришлось бы

выпускать два тома, но это оказалось бы очень дорого для многих читателей, а я хочу, чтобы книга оставалась доступной и давала максимум информации.

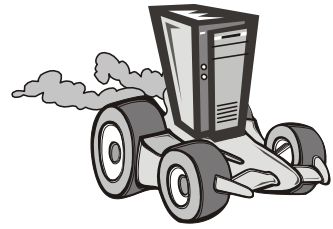
Когда я планировал написать эту книгу, то основная цель была не заработать как можно больше денег, а поделиться знаниями и дать как можно больше. Пусть книга будет дешевле, а я заработаю меньше, но вы уж точно не будете жалеть, что потратили свои деньги.

Над вторым изданием я работал в два раза дольше, несмотря на то, что это было всего лишь обновление. Первое издание было создано на скорую руку и поэтому содержало много ошибок. На этот раз я старался не торопиться, чтобы совершать меньше ошибок. Я человек и могу совершать ошибки или опечатки, поэтому, если вы найдете их в данной книге, просьба сообщить мне об этом через мой сайт **www.flenov.info**.

Для работы над третьим изданием я использовал Borland Delphi версии 2006. Но все, что написано в книге, будет применимо и к более поздним версиям. Некоторые до сих пор используют 7-ю версию, поскольку были разочарованы 8-й версией и Delphi 2005. Если вы тоже используете старую версию, то я советую вам попробовать что-то более современное. Начиная с версии 2006, оболочка Delphi стала намного лучше и стабильнее. Описываемый в книге материал и примеры будут работать на любой версии Delphi, начиная с 7-й.

Наша цель — изучить язык программирования, который не сильно отличается от оболочки. С выходом новых версий в языках появляются новые возможности и функции, но основа остается той же самой.

> **ПРИМЕЧАНИЕ.** В 2006 году среду разработки Delphi выпускал Borland, а в конце года эта задача была передана созданному самостоятельному подразделению CodeGear. А через некоторое время, все подразделение было продано компании Embarcadero Technologies. Теперь если попытаться зайти на сайт **www.codegear.com**, то будет загружен **http://www.embarcadero.com**.



Глава 1

Основные принципы работы компьютера

Прежде чем начинать создавать программы, необходимо понять, как работает компьютер. Программирование — это постоянная борьба с машиной. Нужно заставлять ее делать то, что вам нужно. Поэтому любой программист обязан понимать, как будет выполняться программа. Это позволит вам максимально эффективно использовать все доступные ресурсы и лучше писать код.

1.1. Двоичная система работы процессора

Компьютеры изобрели достаточно давно. В те далекие времена электроника только начинала зарождаться, поэтому первые компьютеры были ламповыми и занимали очень много места. Для того чтобы управлять такой махиной, нужно было очень много обслуживающего персонала. Со временем лампы начали вытесняться электронными компонентами, и компьютеры стали уменьшаться в размерах. Сейчас же мы видим результат прогресса и на наших рабочих столах находятся небольшие системные блоки, которые занимают мало места, а по производительности могут обойти серверы 5—10-летней давности.

В конце 1990-х годов я работал на большом предприятии, на котором сохранились компьютеры 1970-х годов. Это были шкафы (в прямом смысле этого слова), произведенные в Советском Союзе. Вы помните, что была такая страна ☺? Так вот, в то время у меня дома стоял Pentium 100, который был в сто раз меньше и в тысячу раз быстрее.

Однако основные принципы работы компьютера, заложенные во времена их рождения, действуют до сих пор. Суть их заключается в следующем. Данные передаются с помощью какого-то сигнала (для нас не имеет значения какого, потому что мы не электронщики) методом "есть сигнал или нет" или, по-другому, "включен или выключен". Так появился "бит" (bit). Бит — это единица информации, которая может принимать значение 0 или 1, т. е. "включен или выключен". Восемь бит объединяются в байт, один байт равен 8 битам. Почему именно 8? Да потому что первые компьютеры были восьмиразрядными и могли работать одновременно только с 8 разрядами (битами), например, 010000111.

Все первые нули можно удалять, поэтому число 010000111 можно записать как — 10000111. Это то же самое, что и в привычной для нас десятичной системе исчисления, где каждый разряд может принимать значения от 0 до 9. Здесь также

никто не будет писать число 5743 как 0005743, потому что первые нули не имеют никакого значения.

В один байт можно записать любое число от 0 до 255. Почему? Об этом немного позже. Указанный диапазон чисел довольно мал. Поэтому чаще используют более крупные градации:

- два байта = слово;
- два слова = двойное слово.

Итак, компьютер стал работать в двоичной системе исчисления. Но как же тогда записать число 135, если у нас единица информации может принимать значения 0 или 1? Это можно сделать в двоичной системе. Давайте разберемся, как это работает.

Для начала вспомним, как действует десятичная система исчисления, к которой мы привыкли. Для этого рассмотрим число 19 578 246. Я специально выбрал такое число, чтобы оно состояло из восьми разрядов (цифр). Теперь запишем его, как показано на рис. 1.1.

Номер разряда	7	6	5	4	3	2	1	0
	1	9	5	7	8	2	4	6

Рис. 1.1. Число 19 578 246 в десятичной системе исчисления

Как видите, я пронумеровал разряды начиная с нуля до семи и справа налево. Теперь представьте себе, что это не целое число, а просто набор разрядов. 1, 9, 5, 7, 8, 2, 4 и 6. Как из этих разрядов получить целое число в десятичной системе? Наверное, некоторые скажут, что надо просто записать их подряд. А если я спрошу почему? Вот тут появляется математика. Нужно каждый разряд умножить на 10 (степень исчисления), возведенную в степень номера разряда. Непонятно? Попробую оформить сказанное в виде формулы, показанной на рис. 1.2.

$$\text{Разряд 0} * (10^{\text{Номер разряда}}) + \text{Разряд 1} * (10^{\text{Номер разряда}}) + \ldots$$

Рис. 1.2. Работа с десятичными числами

Давайте посчитаем значение числа по этой формуле начиная с нулевого разряда. Получается, что 6 нужно умножить на 10 в нулевой степени $6 \times 10^0 = 6$. Потом прибавить 4×10 в 1-й степени, или $4 \times 10^1 = 40$ (итого уже 46). Потом 2×10 во 2-й степени, $2 \times 10^2 = 200$ (итого 246). Потом 8×10 в 3-й степени, $8 \times 10^3 = 8000$ (итого 8246) и т. д. В итоге получится число 19 578 246. Магия чисел? Нет, это просто математика, и в школе далеко не всегда нам показывают, что означает десятичная система исчисления, а ведь она достаточно проста.

А теперь рассмотрим двоичную систему исчисления. Здесь каждый разряд может принимать значение 0 или 1 (два состояния). Кстати, в десятичной системе у нас каждый разряд мог принимать значения от 0 до 9, т. е. имел десять состояний.

Давайте рассмотрим следующий байт — 10000111. Запишем его на листке бумаги так, как показано на рис. 1.3.

Номер разряда	7	6	5	4	3	2	1	0
Биты	1	0	0	0	0	1	1	1

Рис. 1.3. Двоичное число

Здесь действует та же самая формула, только нужно возводить в степень не 10, а число 2, потому что это число в двоичной системе. Опять же произведем расчет начиная с нулевого разряда, т. е. справа налево. Получается, что первую 1 мы должны умножить на 2 в нулевой степени ($1 \times 2^0 = 1$). Следующую единицу нужно умножить на 2^1, получается 2 (итого 2 + 1 = 3) и т. д. Вот как это будет выглядеть полностью:

$(1 \times 2^0) + (1 \times 2^1) + (1 \times 2^2) + (0 \times 2^3) + (0 \times 2^4) + (0 \times 2^5) + (0 \times 2^6) + (1 \times 2^7) = 135$.

Вот так, оказывается, выглядит в двоичной системе исчисления число 135. Давайте теперь научимся пересчитывать числа из десятичной системы в двоичную. Для этого нужно число 135 разделить на 2. Получается 67 и остаток 1 (запомним 1, т. к. она определяет первый двоичный разряд искомого числа). Теперь 67 снова делим на 2, получается 33 и остаток 1 (таким образом получено уже две двоичные единицы, т. е. 11). 33 делим на 2, получаем 16 и остаток 1 (в результате получаем три двоичные единицы, 111). 16 делим на 2, получаем 8 и остаток 0 (результат 0111). И наконец, 8 делим на 2 = 4, остаток от деления при этом будет 0 (получаем 00111); 4 делим на 2 = 2, остаток 0 (получаем 000111); 2 делим на 2 = 1, остаток 0 (итого 0000111). Оставшаяся единица частного на два не делится, значит, это последний, самый старший разряд искомого числа. Просто дописываем ее к ранее сформированным разрядам и получаем окончательный ответ — 10000111. Получилось первоначальное число.

Вот так происходит преобразование чисел в двоичную систему исчисления. Таким же образом можно перевести число в любую другую систему (двоичную, восьмеричную, шестнадцатеричную). Для более полного закрепления материала в табл. 1.1 показано соответствие десятичных чисел двоичным. Попробуйте сами перевести пару чисел из одной системы исчисления в другую.

Таблица 1.1. *Таблица соответствия десятичных и двоичных чисел*

Десятичное	Двоичное	Десятичное	Двоичное
0	0	6	110
1	1	7	111
2	10	8	1000
3	11	9	1001
4	100	10	1010
5	101		

Например, попробуйте перевести число, состоящее из 8 бит (1 байт), у которого все биты состоят из единиц, в десятичную систему. Вы должны получить 255. Это максимальное число, которое можно записать в одном байте, потому что все его биты равны 1. Вот так и получается, что в 8 бит можно записать числа от 0 до 255. В 16 битах (2 байта или слово) можно записать число от 0 до 65 535. В 32 битах (двойное слово) можно уже записать число от 0 до 4 294 967 295.

В компьютере принято вести расчет в двоичной или шестнадцатеричной системе. Вторая вошла в обиход, когда компьютеры стали 16-разрядными. Да, когда мы будем писать свои программы на Delphi, то будем использовать привычную нам десятичную систему, потому что перед нами интеллектуальный компилятор, который во время компиляции сам переведет все числа в нужный процессору вид, но понимать, какими числами думает процессор, просто необходимо.

Шестнадцатеричная система выглядит немного по-другому. Каждый разряд содержит уже не 2 состояния (как в двоичной системе) или десять (как в десятичной системе), а шестнадцать. Поэтому один разряд может принимать значения: 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Буква "A" соответствует цифре 10 в десятичной системе, "B" соответствует 11 и т. д. Например, число 1A в шестнадцатеричной системе равно 26 в десятичной. Почему? Да в соответствии все с той же формулой. Только здесь нужно возводить уже 16 в степень номера разряда. "A" — десять, нужно умножить на 16^0. В результате получится 10. 1 — первый разряд нужно умножить на 16^1, получится значение 16. Затем полученные результаты складываются и определяется искомое число — 10 + 16 = 26. В результате, как показано в табл. 1.2, можно установить соответствие между числами, записанными в различных системах исчисления.

Таблица 1.2. *Таблица соответствия десятичных, двоичных и шестнадцатеричных чисел*

Десятичное	Двоичное	Шестнадцатеричное
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C

***Таблица 1.2** (окончание)*

Десятичное	Двоичное	Шестнадцатеричное
13	1101	D
14	1110	E
15	1111	F
16	10 000	10
17	10 001	11
18	10 010	12
19	10 011	13
20	10 100	14

На протяжении всей книги мы будем иногда встречаться с шестнадцатеричной системой исчисления (без этого никуда не денешься). В этом случае, когда нужно будет показать, что число шестнадцатеричное, перед ним будет ставиться знак решетки #, например, #13. В других языках, например Assembler или C++, принято ставить в конце числа букву h (13h). Но эта книга о Delphi, поэтому здесь будем писать так, как принято в этой среде разработки, чтобы потом не возникало никаких вопросов.

До сих пор рассматривались целые числа. С числами с плавающей точкой (имеющими дробную часть) совершенно другая история, и ее мы рассматривать не будем.

Теперь разберемся со знаком у чисел. Если заранее предусмотрено, что число может быть отрицательным, то его длина сокращается ровно на один бит (этот бит отводится под знак числа). Так, неотрицательное целое число может быть 8-битным, тогда как число со знаком будет 7-битным. Первый бит будет означать знак. Если первый бит равен 1, то число отрицательное, иначе положительное.

В дробных числах один байт может быть отведен для целой части и один для дробной. Никогда не смешивают целую и дробную части в одно целое. За счет этого дробные числа всегда будут занимать больше памяти, и операции с ними будут проходить намного дольше.

В первых процессорах вообще не было команд для работы с вещественными числами. Со временем разработчики поняли, что работать с вещественными числами через команды целочисленных вычислений достаточно накладно, и в компьютеры стали устанавливать математические сопроцессоры. Этот модуль был выполнен в виде отдельного процессора. В современных компьютерах сопроцессор реализован в виде модуля внутри основного процессора.

На первый взгляд перевод чисел очень сложный процесс, но вручную им заниматься не обязательно. Человек уже давно придумал для себя хорошего помощника — калькулятор. С его помощью без проблем можно перевести число в любую систему исчисления.

Запустите встроенный в Windows калькулятор (**Пуск | Программы | Стандартные | Калькулятор**). Теперь выберите из меню **Вид** пункт **Инженерный**. На рис. 1.4 показано окно, которое вы должны увидеть.

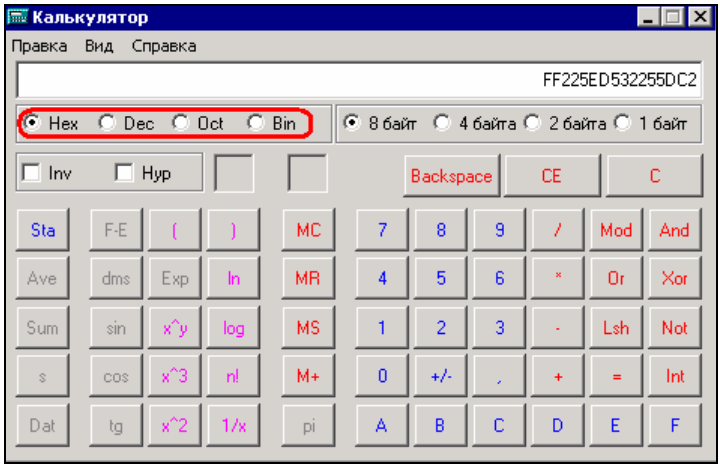


Рис. 1.4. Внешний вид калькулятора

Для перевода числа в другую систему просто наберите его в окне ввода калькулятора и потом выберите нужную систему исчисления. На рисунке кнопки переключения из одной системы исчисления в другую выделены овалом:

- **Hex** — шестнадцатеричная;
- **Dec** — десятичная;
- **Oct** — восьмеричная;
- **Bin** — двоичная.

Возникает вопрос — зачем здесь так долго рассказывалось о преобразованиях чисел, когда так легко можно воспользоваться калькулятором? Ответ прост — это нужно знать. Поверьте мне. Если вы будете понимать, как происходит преобразование, то вам потом легче будет работать с этими числами. Уметь использовать компьютер и язык программирования — это хорошо, но понимать, как и что происходит — намного лучше.

1.2. Машинный язык

Данные на диске хранятся в двоичном виде. Даже текстовые файлы на диске выглядят в виде нулей и единиц. Точно так же выглядит и любая программа, только ее называют *машинным кодом*. Давайте познакомимся с ним немного поближе.

Любая программа представляет собой последовательность команд. Эти команды называются *процессорными инструкциями*. В точном соответствии этим инструкциям процессор определяет, что и как ему нужно делать. Когда вы запускаете программу, компьютер загружает ее машинный код в оперативную память и начинает выполнять команду за командой. Наша задача как программистов написать эти инструкции таким образом, чтобы компьютер понял, что мы от него хотим.

Реальная программа, которую выполняет компьютер, представляет собой последовательность единиц и нулей. Такую последовательность называют машинным языком. Человек не способен эффективно думать единицами и нулями. Для нас легче воспринимается осмысленный текст, а не сумасшедшие числа в двоичной системе исчисления, с которой мы не привыкли работать. Например, *команда сложения* двух

регистров в шестнадцатеричной системе выглядит так: $03C3. Это мало о чем говорит, и запомнить такую команду очень тяжело. Намного проще написать "сложить число 1 и число 2".

Первое время программисты писали программы в машинных кодах, пока кому-то не пришла в голову идея: "Почему бы не писать текст программы на понятном языке, а потом заставлять компьютер переводить этот текст в машинный код?" Идея действительно заслуживала внимания. Так появился первый компилятор — программа, которая переводила текст программ в машинный код. Таким образом, пользователи стали писать программы более осмысленно, а всю рутинную работу по переводу текста программы в машинный код возложили на сам компьютер.

Здесь настало время сделать паузу и рассказать вам небольшую историю языков программирования. Она достаточно интересна и поучительна. Ну а потом мы продолжим изучение принципов работы компьютера и познакомимся со структурой процессора и его работой при выполнении программы.

1.3. История языков программирования

Как мы уже выяснили, компьютер — примитивное существо, которое мыслит нулями и единицами, из которых складываются числа. Основное его устройство — процессор. Все, что может делать процессор, — это оперировать двоичными числами. Программы — это тоже числа, которые воспринимаются процессором как некоторая совокупность команд с целью выполнения им определенных действий.

Мы также выяснили, что первые программисты писали программы в машинных кодах. Тогда еще не было компиляторов и приходилось все писать числами. Вы даже представить себе не можете, какой это адский труд — постоянно держать в памяти таблицу машинных кодов (это не таблица умножения). Например, вам понятно шестнадцатеричное число 8BC3? Нет? А это обычная команда копирования между двумя ячейками регистров. Регистр — это память в процессоре определенного объема, которая может использоваться для хранения значений, с которыми будет работать процессор. С ними работа происходит быстрее, чем с оперативной памятью, а размер зависит от архитектуры процессора.

Это был пример машинного кода, потому что тогда регистры были другие и процессоры были намного проще. Со временем компьютер стал умнеть, но самое главное, он все так же оперировал двоичными числами, однако делал это намного быстрее.

Программист — это человек, а не железка, и ему очень тяжело создавать логику работы программы в числах. Намного легче работать с привычными словами. Например, все ту же команду пересылки удобней записать словами типа "скопировать `ebx` в `eax`" (`eax` и `ebx` — это регистры процессора). Но что делать, если компьютер не понимает слов, а только числа? Выход есть — написать такую программу, которая будет превращать написанный текст, понятный человеку, в машинные коды. Пусть компьютер сам создает *байт*-код. Программу, выполняющую эти действия, назвали компилятором, а язык, на котором писался исходный текст программы, — языком программирования.

```
MainUnit.pas.131: InitViewGrid;
00527F7C 8BC3            mov eax,ebx
00527F7E E84D010000      call TSborDataForm.InitViewGrid
MainUnit.pas.132: InitProgramm;
00527F83 8BC3            mov eax,ebx
00527F85 E8B6030000      call TSborDataForm.InitProgramm
MainUnit.pas.133: if lDeviceNum>=0 then
00527F8A 83BB1004000000  cmp dword ptr [ebx+$00000410],$00
00527F91 7C26            jl +$26
MainUnit.pas.135: ScanPortThr := TScanPortThread.Create(true);
00527F93 B101            mov cl,$01
00527F95 B201            mov dl,$01
00527F97 A1D4A45200      mov eax,[$0052a4d4]
00527F9C E8EB94EFFF      call TThread.Create
00527FA1 8BF0            mov esi,eax
00527FA3 89B314040000    mov [ebx+$00000414],esi
MainUnit.pas.136: ScanPortThr.lDeviceNum:=lDeviceNum;
00527FA9 8B8310040000    mov eax,[ebx+$00000410]
00527FAF 89464C          mov [esi+$4c],eax
MainUnit.pas.137: ScanPortThr.Resume;
00527FB2 8BC6            mov eax,esi
00527FB4 E81F98EFFF      call TThread.Resume
MainUnit.pas.141: try
00527FB9 33C0            xor eax,eax
00527FBB 55              push ebp
00527FBC 680B805200      push $0052800b
00527FC1 64FF30          push dword ptr fs:[eax]
```

Рис. 1.5. Программа в машинных кодах и Assembler

И вот был разработан первый компилятор. Эту программу назвали *Assembler*, что переводится как "сборщик". Писать на нем практически так же сложно, как и в машинных кодах, однако теперь уже использовались не числа, а понятные, как показано на рис. 1.5, человеку слова. Текст на рисунке можно разделить на три колонки:

- адрес инструкции;
- машинный код инструкции;
- код на языке Ассемблера.

Теперь все та же команда копирования регистров выглядела так: `mov eax, ebx`. В данном случае `mov` — это команда языка программирования, которая происходит от английского слова move, двигать. `eax` и `ebx` — имена регистров. Получается, что приведенная выше команда может читаться как двигать в регистр `eax` значение из `ebx`. Да, код на ассемблере не совсем нагляден, но зато намного удобнее, чем то же самое, но в машинных кодах.

Вроде все прекрасно и удобно, но почему-то среди программистов возникли споры и разногласия. Кто-то воспринял новый метод с удовольствием. А кто-то говорил, что машинные коды лучше. Любители языка Assembler хвалили компилятор за то, что программировать стало проще и быстрее, а противники утверждали, что программа, написанная в кодах, работает быстрей. Говорят, что эти споры доходили до драк и иногда лучшие друзья становились врагами.

В принципе, и те и другие были правы. На языке Assembler действительно программу писать легче и быстрей, но программа, написанная в машинных кодах, работала быстрее и более гибко. Когда программа пишется в машинных кодах, то программист ничем не ограничен, а при работе с ассемблером наши руки более связаны. Мы не всегда можем повлиять на результат и зависим от того, как он

захочет превратить наш текст в программу и какие инструкции процессора при этом будет использовать.

Тогда никто не мог себе представить, чем же все может закончиться. Но время показало свое. С помощью Assembler программы писались быстрее, а это один из основных факторов успеха любой программы на рынке. Люди начинают пользоваться тем продуктом, который выходит на рынок первым. Даже если более поздний вариант лучше, человека трудно переубедить перейти на другую версию. Здесь начинает играть большую роль фактор привычки. К тому же, когда программист напишет свою первую версию программы в машинных кодах, программист на языке Assembler выпустит уже пару новых версий.

Вот так и получилось, что те, кто программировал на языке Assembler, превратились в убегающих вперед, а те, кто программировал в машинных кодах, превратились в постоянно догоняющих. В конце концов, первые убежали настолько "далеко", что вторые не смогли догнать и вынуждены были или перейти на Assembler или отойти от программирования совсем.

С этого момента начался бум. Языки программирования стали появляться один за другим. Так появились *C, ADA, FoxPro, Fortran, Basic, Pascal* и др. Некоторые из них были предназначены только для школьников или обучения, другие были ориентированы на профессиональных программистов. И тут споры перенеслись в другую плоскость — какой язык лучше. Некоторые говорили, что это Pascal, другие утверждали, что C, ну а кое-кто утверждал, что это Basic. Этот спор длится уже около 30 лет, и конца ему не видно. При этом все спорные вопросы разделились на две части.

1. Какой язык лучший?
2. Что лучше — язык высокого уровня или низкого?

Спор по первому пункту не может закончиться до сих пор. Каждый пытается доказать, что его язык программирования самый мощный, удобный и создает самый быстрый программный код. Мне кажется, что этот спор не закончится никогда. В принципе, такое положение дел устраивает всех, потому что это своеобразная конкуренция. Благодаря ей происходит развитие языков программирования, и мы быстро продвигаемся вперед.

Так все же, какой язык лучше? На этот вопрос можно дать ответ, но немного позже.

Наиболее интересным был спор: "Что лучше — язык высокого уровня или низкого?" Язык низкого уровня — это тот, который ориентирован на команды процессора, т. е. Assembler. К языкам высокого уровня относят C, Pascal, Basic и другие (на то время это были структурные языки программирования, они имели более высокий уровень по сравнению с ассемблером). Они ориентированы на людей и создают им максимум удобства при написании программ. Этот спор проходил в той же манере, как и спор между любителями Assembler и любителями программирования в машинных кодах. Только теперь приверженцы языка Assembler утверждали, что их код самый быстрый, а любители языков высокого уровня утверждали, что они напишут программу быстрей, чем самый лучший программист на языке Assembler.

Спор продолжался достаточно долгое время. И опять победила скорость разработки и "удобство" языка программирования. Любителям Assembler пришлось отступить, потому что теперь они превратились в " догоняющих".

Конечно же, нельзя сказать, что машинные коды и Assembler окончательно ушли из нашей жизни. Они используются до сих пор, но в очень ограниченном количестве. Язык Assembler используется только в качестве вставок для языков высокого уровня, а машинные коды используются для написания того, что не может сделать компилятор (да и для написания самого компилятора они могут потребоваться). Ушедшие технологии живут и будут жить, но рядовой программист очень редко встречается с ними.

Следующей ступенью стало объектно-ориентированное программирование. Язык C превратился в *C++*, Pascal превратился в *Object Pascal* и т. д. И снова борьба. И снова скорость разработки против скорости выполнения программного кода. Опять споры, драки и оскорбления.

Война длилась несколько лет. Сколько времени было потрачено в спорах, сколько волос было вырвано на голове в процессе доказательств силы именно его программного кода. А в результате победила скорость и удобство разработки, т. е. объектно-ориентированное программирование (ООП).

Последней крупной революцией, происходящей в программировании, считается переход на визуальное программирование. Этот переход происходит прямо на наших глазах. Визуальность дает нам еще более удобные средства разработки для более быстрого написания кода, но проигрывает ООП по быстроте работы. Вот почему многие начинающие программисты стоят на перекрестке, не зная, какой язык выбрать.

Лидером в разработке визуальных языков программирования является Borland, а приверженцем ООП остается Microsoft. Конечно же, Билл Гейтс пытается встроить в свои языки визуальность, но она примитивна по сравнению с такими гигантами, как *Delphi, Kylix* или *C++ Builder*. Это связано с изначально неправильной разработкой MFC (Microsoft Foundation Classes — Основные Классы Microsoft), которые не могут работать визуально. Нужна глобальная переработка кода, которую почему-то не хотят делать. Вот народ и стоит на двух атомных бомбах, ожидая взрыва одной из них. Как вы думаете, какая бомба рванет? Что победит — скорость разработки или скорость кода? Я не буду отвечать на этот вопрос. История говорит сама за себя, а мы подождем подтверждения этому.

К моменту написания этих строк, победитель уже начинает вырисовываться. Классический C++ уходит в сторону, а вместо него появляется C# и мощные визуальные средства, идеология которых была позаимствована у Java и Delphi. Но Delphi уже давно визуальный и ориентирован на объекты и в нашей стране получил большую популярность, а C# только пытается захватить рынок.

Считается, что прогресс не будет стоять на одном месте и переход на новые технологии программирования рано или поздно состоится. Поэтому я уже перешел на Delphi. Если вы хотите успеть за прогрессом, то тоже обязаны вступить в партию любителей Borland и его подразделения по средствам разработки CodeGear. Выбирайте любой из его компиляторов, и вы не ошибетесь. Для вас есть все, что угодно, — Delphi, JBuilder, Kylix или C++ Builder. Как видите, у корпорации Borland есть визуальные варианты всех языков, и они действительно лучшие.

Осталось только ответить на вопрос: "Какой язык программирования лучше?" Я уже несколько лет пытаюсь ответить на этот вопрос, но окончательного решения

вынести не могу. Даже у того же *Visual C++* от Microsoft есть свои плюсы, хотя я его не очень люблю за корявость объектной модели. Как это ни странно, но положительные стороны есть у всех. Вопрос остается только за тем, какие программы будут создаваться? Здесь можно дать примерно такую градацию.

1. Если вы будете писать базы данных, программы общего значения или утилиты, то ваш язык Delphi или C++ Builder.
2. Если это игры, то желательно Visual C++ плюс знание Assembler. Но это не значит, что нельзя использовать Delphi или C++ Builder. В этих средах вы потеряете не намного больше в скорости работы, поэтому в большинстве игр можно не обращать внимания на эту потерю. Если правильно использовать свои знания, то и на самом медленном и слабом языке программирования можно создать шедевр.
3. Если это будут драйверы и работа с "железом", где критичен размер файла, ваш язык чистый C или Assembler.

И все же большую массу программ занимают утилиты и базы данных. А тут визуальность необходима, если вы хотите оказаться впереди. Визуальные языки будут жить и за ними будущее. На протяжении всей этой книги будет рассматриваться лучший (это на мой взгляд, и он может отличаться от других) язык программирования — Delphi.

Корпорация Microsoft тоже движется в сторону визуальности и простоты разработки программных продуктов. Об этом говорит их новая платформа .NET и язык разработки C#.

1.4. Исполнение машинных команд

Прежде чем переходить к дальнейшему изучению материала, необходимо рассмотреть ряд понятий.

Мы уже разобрались с байтом и его размером. Теперь рассмотрим другие существующие размерности:

- 1 килобайт = 1024 байт;
- 1 мегабайт = 1024 килобайт;
- 1 гигабайт = 1024 мегабайт.

Почему именно 1024? Это число 2 в 10-й степени. В компьютере большинство значений являются степенью числа 2, потому что компьютер оперирует двоичной системой, и таким образом можно максимально эффективно использовать его возможности.

Сегмент — это область внутренней (оперативной) памяти компьютера (внешняя память представлена накопителями на магнитных дисках и лентах). Раньше, когда операционные системы были 16-битными, процессор не мог работать с памятью размером более 64 килобайт, потому что это максимальный размер области памяти, который можно адресовать, используя в этих целях адрес длиной в два байта. Поэтому память делилась на сегменты по размеру и по назначению. На данный момент мы используем 32-разрядную ОС, которая может адресовать до 4 Гбайт оперативной памяти (длина адреса — 32 бита или 4 байта). Поэтому можно

сказать, что память стала сплошной. Однако деление ее по назначению все-таки осталось.

Существуют следующие сегменты памяти:

- *сегмент кода* — в эту область памяти загружается машинный код, который будет потом выполняться процессором;
- *сегмент данных* — это область памяти для хранения данных;
- *сегмент стека* — область памяти для хранения временных (локальных) данных и адресов возврата из процедур.

Каждой запущенной программе отводится свой сегмент кода, данных и стека. Поэтому данные одной программы не могут пересекаться с данными или кодом другой программы. Да, существуют методы взаимодействия между процессами (программами), но это отдельный случай.

Регистр — ячейка памяти в процессоре. Размер ячейки зависит от ее разрядности. В 32-разрядных процессорах ячейки 32-битные, но есть и 64-битные. Таких регистров у процессора несколько, и каждый из них предназначен для определенных целей. Существуют также регистры общего значения, которые программа может использовать на свое усмотрение.

На рис. 1.6 показан регистр `EAX`. Полная его длина 32 бита, но младшая половина — это регистр `AX` (16-битный вариант регистра). То есть если мы "попросим" процессор показать нам содержимое регистра `AX`, то мы увидим половину регистра `EAX`. Иногда это очень удобно, особенно когда требуется прочитать только половину числа из регистра.

> **ПРИМЕЧАНИЕ.** Очень часто, прежде чем выполнить какую-то команду, процессор загружает необходимые данные в регистры и только после этого выполняет необходимую инструкцию. Но возможны варианты, когда вычисления идут напрямую с памятью.

Я думаю, что этого будет достаточно для понимания основ работы компьютера и программ. Если вы хотите узнать больше, то советую почитать спецификации Intel, которых можно найти великое множество на сайте **www.intel.com**.

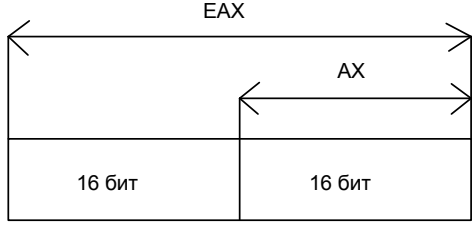


Рис. 1.6. Регистры

Глава 2

Машинная математика

До начала перестройки в нашей стране практически не готовили программистов. Большинство программистов были выходцами с кафедр математики, на которых очень часто изучались определенные учебные курсы с уклоном в сторону информатики. На этих курсах учили писать *блок*-схемы, которые в свою очередь помогали понять логику работы программы.

С *блок*-схемами знакомятся на начальном этапе обучения программированию. Первое впечатление тех, кто столкнулся с *блок*-схемами, — абсолютно ненужная ерунда. Однако со временем становятся понятны их достоинства. Возможно, что и вам покажется это слишком просто, но все же желательно прочитать эту главу полностью. Здесь рассматривается теория всего процесса программирования, что позволяет понять логику выполнения команд в процессе выполнения программы на процессоре. В дальнейшем останется только познакомиться с практикой.

Когда мне пришлось изучать информатику в институте, то нас тоже заставляли писать блок схемы. Самое страшное, что нас именно заставляли. Прежде чем начать что-то писать, мы должны были описать весь процесс в виде логики из блоков. Я не собираюсь вам говорить, что вы должны поступать так же. Все равно таким образом можно описать только небольшой участок логики или простой алгоритм. В большинстве случаев можно приступать к программированию сразу, но если вы запутались или что-то не получается, попробуйте отложить клавиатуру, взять листок бумаги и набросать логику с помощью блоков. Очень часто это помогает.

2.1. Основы машинной математики

В любом языке программирования можно выполнять математические операции любой сложности. Delphi в этом вопросе не исключение. Но пока мы не будем рассматривать все возможности этого языка в данной области, а остановимся только на математических операциях. В табл. 2.1 представлены основные математические операции языка, которые мы будем использовать.

Таблица 2.1. *Основные математические операции Delphi*

Математическая операция	Описание
*	Умножить
/	Разделить
Sqr	Квадрат
Sqrt	Квадратный корень
+	Сложение
-	Вычитание
:=	Присвоить значение

Все эти операции выполняются в том же порядке, в котором перечислены. Например, результатом вычисления выражения `2+2*2` будет 6, а не 8, потому что сначала выполняется операция умножения, а потом сложения. Если вы хотите сначала выполнить сложение, а потом вычитание, то, как и в математике, нужно использовать скобки `(2+2)*2=8`. В этом случае результат будет совершенно другим.

Для изучения компьютерной математики необходимо уяснить ряд понятий и мы начнем с понятия *переменной*.

Переменная — это ячейка оперативной памяти, в которую можно записывать различные значения. Чаще всего этой ячейке памяти ставится в соответствие какое-нибудь имя. Например, можно определить переменную с именем `F`. Ей, в свою очередь, можно присваивать значения, например, 5. Для этого достаточно записать выражение `F:=5`. Последовательность символов — знак двоеточия и равно означают здесь операцию "присвоить".

Почему для присвоения значения используется именно `:=`, тогда как мы привыкли к простому знаку равенства? Это необходимо, чтобы отделить операцию присваивания от операции сравнения. Знак равенства в Delphi используется для сравнения чисел, а `:=` для присваивания значения переменным.

Значения переменных можно копировать. Допустим, имеется еще одна переменная `G`. Ей можно присвоить значение переменной `F` с помощью простого присваивания `G:=F`. После этого в переменной `G` тоже будет значение 5.

Переменной можно присваивать результаты каких-то вычислений, например: `F:=10/2`. Это достаточно простой пример. А вот уже целое выражение с использованием переменных:

```
F:=5;
G:=10;
F:=G/2.
```

Имя переменной может состоять как из одной, так и из нескольких букв. Например, переменная может иметь имя `Str` или `MyVariable`. Единственное ограничение, которое следует здесь учитывать, — это то, что имя должно состоять из английских букв и не должно использовать зарезервированные слова (о зарезервированных

словах будет сказано немного позже). Вы также можете в имени переменной использовать числа (желательно в конце), например, `Str1`, `Str2`, `Str3` и т. д.

> **СОВЕТ.** Назначайте переменным осмысленные имена. Когда вы начнете писать большие программы, тяжело будет разобраться, что означает переменная `i`, `b`, `Str1` или `Temp`.

Тип переменной — это тип значения, которое можно присвоить переменной. Очень часто используют термин *тип данных*, потому что это действительно тип данных, хранящихся в переменной. Он показывает, какого типа информация может быть присвоена переменной (помещена в выделенную область памяти). В Delphi принято обязательно указывать типы переменных, чтобы сразу можно было определить, какую информацию можно туда записать, а какую нет.

Язык Delphi строго типизирован и требует, чтобы каждая переменная имела свой тип. Да, есть возможность выделить память, не указывая тип или указав тип `Variant` (забегая вперед, скажу, что этот тип позволяет хранить любые типы данных). Но для выполнения операций нетипизированные данные должны быть приведены к определенному типу.

Существует несколько основных типов переменных, которые на данный момент времени необходимо четко представлять (табл. 2.2).

Таблица 2.2. *Основные типы данных в Delphi*

Название типа	Описание	Дополнительная информация
`Integer`	Целое число	Переменная этого типа может принимать в качестве значения любые целые числа, как положительные, так и отрицательные
`Real`	Вещественное число	Переменная этого типа может принимать в качестве значения целые и дробные числа со знаком и без
`String`	Строка	Переменная этого типа может принимать в качестве значения любые символы и наборы символов
`Boolean`	Булево значение	Переменная может принимать значение `true` или `false` (истина или ложь). Этот тип очень часто используется для организации логики

В табл. 2.2 приведены только основные типы данных. Реально их намного больше. Когда мы перейдем к программированию, вы познакомитесь с большим количеством типов.

Строки — это любые символы или наборы символов. В языке Delphi они выделяются одинарными кавычками, например, `'Привет'`. Строки так же можно присваивать переменным, как и любое другое значение. Например:

```
Str — строковая переменная.
Str:='Привет!!!'
```

Изложенного материала будет достаточно для понимания переменных и перехода к блок-схемам.

2.2. Блок-схемы

Давайте сразу зададим какой-нибудь простой пример, на котором попробуем определить логику его решения компьютером. Допустим, нам надо получить сумму двух чисел. В логике человека мы должны выполнить следующие операции:

Листинг 2.1. Сложение двух чисел

```
Старт.
Ввести число 1.
Ввести число 2.
Прибавить к числу 1 число 2.
Вывести результат.
```

Это простейшая и подробная логика, которой оперирует человек. Но машина так не может мыслить, и по ее логике нужно рассуждать немного иначе. Для отображения машинной логики определение вычислительных шагов неудобно, потому что решение может быть извилистым, а не прямолинейным. Поэтому давайте знакомиться с блок-схемами на этой линейной задаче.

Блок-схемы принято чертить различными квадратами, овалами и прямоугольниками. Я особо не буду придерживаться стандартов, потому что это не принципиально в решении, но некоторых особенностей мы будем придерживаться. Основные типы блоков, которые можно увидеть на рис. 2.1.

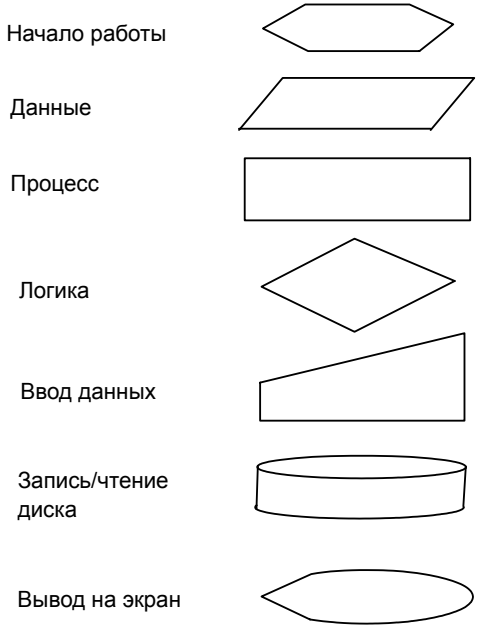


Рис. 2.1. Основные фигуры блок-схемы