

А. Чиртик, В. Борисок, Ю. Корвель

трюки & эффекты

DELPHI



**Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Новосибирск · Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск**

2007

ББК 32.973.23-018.2

УДК 004.42

Ч-65

Чиртик А. А., Борисок В. В., Корвель Ю. И.

Ч-65 Delphi. Трюки и эффекты (+CD). — СПб.: Питер, 2007. — 400 с.: ил. — (Серия «Трюки и эффекты»).

ISBN 978-5-91180-219-6

5-91180-219-8

«Delphi. Трюки и эффекты», как и все издания данной серии, адресована тем, кто хочет научиться делать с помощью уже знакомых программных пакетов новые, интересные вещи. В первой части книги многое говорится о среде разработки Delphi (самых последних версий) и программировании на языке Object Pascal. Благодаря этому издание подходит и новичкам, и начинающим программистам. Вторая (основная) часть книги описывает удивительные возможности, скрытые в языке, и на примерах учит читателя программистским фокусам — от «мышек-невидимок» и «непослушных окон» до воспроизведения MP3 и управления офисными программами Word и Excel из приложений Delphi. Купив эту книгу, вы пройдете непростой путь к вершинам программистского мастерства весело и интересно.

ББК 32.973.23-018.2

УДК 004.42

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-91180-219-6

© ООО «Питер Пресс», 2007

Краткое содержание

Введение	9
От издательства	9
Глава 1. Окна	10
Глава 2. Уменьшение размера EXE-файла. Использование Windows API	49
Глава 3. Мышь и клавиатура	91
Глава 4. Диски, каталоги, файлы	111
Глава 5. Мультимедиа	154
Глава 6. Использование Windows GDI	174
Глава 7. Системная информация и реестр Windows	201
Глава 8. Обмен данными между приложениями	242
Глава 9. Возможности COM в Microsoft Word и Microsoft Excel ..	255
Глава 10. Окна других приложений	267
Глава 11. Сетевое взаимодействие	297
Глава 12. Шифрование	332
Заключение	373
Приложение 1. Коды и обозначения основных клавиш	374
Приложение 2. Оконные стили	378
Приложение 3. Сообщения	387

Оглавление

Введение	9
От издательства	9
Глава 1. Окна	10
1.1. Привлечение внимания к приложению	11
Инверсия заголовка окна	11
Активизация окна	14
1.2. Окно приложения	15
1.3. Полупрозрачные окна	17
1.4. Окна и кнопки нестандартной формы	20
Регионы. Создание и использование	20
Закругленные окна и многоугольники	24
Комбинированные регионы	31
1.5. Немного о перемещении окон	37
Перемещение за клиентскую область	38
Перемещаемые элементы управления	40
1.6. Масштабирование окон	43
1.7. Добавление пункта в системное меню окна	45
1.8. Отображение формы поверх других окон	47
Глава 2. Уменьшение размера EXE-файла. Использование Windows API ..	49
2.1. Источник лишних килобайт	51
2.2. Создание окна вручную	54
2.3. Окно с элементами управления	58
Создание элементов управления	58
Использование элементов управления	62
Реакция на события элементов управления	67
Пример приложения	68

2.4. Стандартные диалоговые окна Windows	72
Окно открытия/сохранения файла	73
Окно выбора цвета	75
Окно выбора шрифта	77
Окно выбора папки	78
Окна подключения и отключения сетевого ресурса	80
Системное окно «О программе»	81
Демонстрационное приложение	82
2.5. Установка шрифта элементов управления	87
Глава 3. Мышь и клавиатура	91
3.1. Мышь	92
Координаты и указатель мыши	92
Захват указателя мыши	94
Ограничение перемещения указателя	95
Изменение назначения кнопок мыши	96
Подсчет расстояния, пройденного указателем мыши	97
Подсвечивание элементов управления	101
3.2. Клавиатура	102
Определение информации о клавиатуре	103
Опрос клавиатуры	104
Имитация нажатия клавиш	106
«Бегущие огни» на клавиатуре	108
Глава 4. Диски, каталоги, файлы	111
4.1. Диски	112
Сбор информации о дисках	112
Изменение метки диска	118
Программа просмотра свойств дисков	118
4.2. Каталоги и пути	121
Системные папки WINDOWS и system	121
Имена для временных файлов	122
Прочие системные пути	124

Определение и установка текущей папки	127
Преобразование путей	127
Поиск	133
Построение дерева каталогов	140
4.3. Файлы	144
Красивое копирование файла	144
Определение значков, ассоциированных с файлами	147
Извлечение значков из EXE- и DLL-файлов	150
Глава 5. Мультимедиа	154
5.1. Воспроизведение звука с помощью системного динамика	155
5.2. Использование компонента MediaPlayer	156
5.3. Компонент Animate	161
5.4. Разработка звукового проигрывателя	165
5.5. Видеопроигрыватель	170
Глава 6. Использование Windows GDI	174
6.1. Графические объекты	175
6.2. Аппаратно-независимый графический вывод	176
6.3. Контекст устройства	176
Экранный контекст устройства	177
Контекст устройства принтера	178
Контекст устройства памяти	179
Информационный контекст устройства	179
6.4. Графические режимы	180
6.5. Работа со шрифтами	180
6.6. Рисование примитивов	181
6.7. Работа с текстом	186
6.8. Работа с растровыми изображениями	191
6.9. Альфа-смешивание	194
Глава 7. Системная информация и реестр Windows	201
7.1. Системная информация	202
Версия операционной системы	202
Имя компьютера	207
Имя пользователя	207

Состояние системы питания компьютера	208
Состояние памяти компьютера	211
7.2. Системное время	213
Давно ли запущена операционная система?	213
Аппаратный таймер	214
Мультимедиа-таймер	216
Создание программного таймера высокой точности	218
7.3. Реестр	221
Краткие сведения о реестре Windows	221
Средства работы с реестром	222
Хранение настроек программы в реестре	224
Автозапуск программ	228
Запуск приложения из командной строки	232
Регистрация типов файлов	233
Программа для просмотра реестра	236
Глава 8. Обмен данными между приложениями	242
8.1. Сообщение WM_COPYDATA	243
8.2. Использование буфера обмена	246
8.3. Проецируемые в память файлы	250
Глава 9. Возможности COM в Microsoft Word и Microsoft Excel	255
9.1. Технология OLE	256
9.2. Технология COM	256
9.3. Использование OLE в Delphi	257
Microsoft Office с точки зрения COM	257
Объект Application	258
Класс TOLEServer	259
9.4. Управление Microsoft Word и Microsoft Excel	260
Трюки в Microsoft Word	260
Трюки в Microsoft Excel	264
Глава 10. Окна других приложений	267
10.1. Ловушки Windows	268
Виды ловушек	269
Расположение функции-ловушки и DLL	272

10.2. Программа «Оконный шпион»	273
Составление списка открытых окон	274
Получение информации об окне	276
Изменение оконных стилей	281
Перехват сообщений	283
Глава 11. Сетевое взаимодействие	297
11.1. Краткое описание сетевых компонентов	298
11.2. Простой обмен данными	300
11.3. Слежение за компьютером по сети	302
11.4. Многопользовательский разговорник	309
Требования к клиентскому и серверному приложениям	309
Формат сообщений клиента и сервера	310
Реализация сервера	311
Реализация клиентского приложения	324
Глава 12. Шифрование	332
12.1. Основы криптографии	334
12.2. Шифр простой подстановки	336
12.3. Транспозиция	345
12.4. Шифр Виженера и его варианты	352
12.5. Шифр с автоключом	359
12.6. Взлом	365
Заключение	373
Приложение 1. Коды и обозначения основных клавиш	374
Приложение 2. Оконные стили	378
Приложение 3. Сообщения	387

Введение

В настоящее время количество книг, посвященных различным языкам программирования, настолько велико, что иногда просто не знаешь, какую выбрать. Цель этой книги — не просто тривиальное изложение материала о Delphi. Она поможет вам получить опыт в решении многих задач. В итоге у вас будет необходимый базис знаний, который даст возможность легко и быстро усваивать что-то новое. Здесь вы найдете ответы на вопросы, которые возникают у большинства людей при разработке своих собственных приложений. Вам больше не придется задумываться над тем, как решать мелкие задачи, которые являются повседневной работой большинства программистов. У вас появится возможность тратить больше времени именно на основную цель, поставленную перед вами, а не на второстепенные.

Данная книга рассчитана на читателей, которые уже имеют некий опыт в программировании, причем достаточный, чтобы не излагать тривиальные вещи заново. Но сразу отмечу, пусть даже вы делаете свои первые шаги на пути к написанию приложений на высоком уровне, книга окажет вам неоценимую помощь. Она построена так, чтобы вы смогли с высокой степенью эффективности узнавать новый материал. В конце книги есть приложения в удобном для восприятия виде. В них вы найдете информацию, которая часто используется при написании программ.

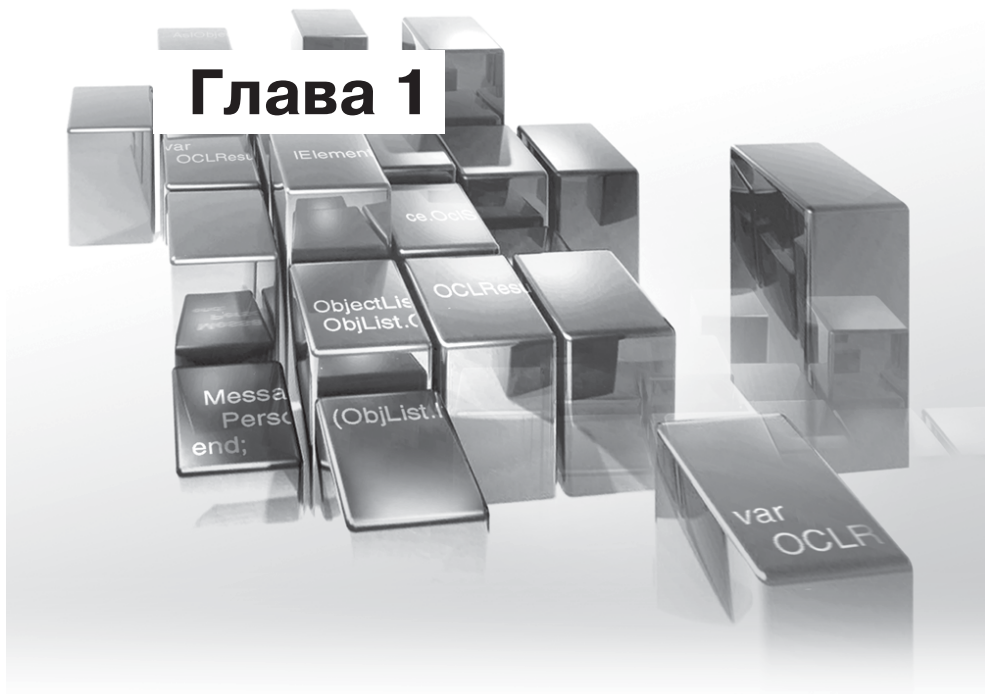
Зачастую люди выбирают Delphi за его простоту. Программа подкупает начинающих пользователей, которые хотят почти сразу писать программы, а не разбираться в особенностях синтаксиса языка. Простота в совокупности с мощностью дают вам целый набор инструментов для воплощения задуманного. Однако запомните: чтобы научиться хорошо программировать, недостаточно иметь огромный объем теоретических знаний, хотя и он немаловажен. Следует научиться думать в концепции выбранного вами языка, и тогда вас ждет успех. Ведь не понимая, зачем все это нужно, вы не сможете эффективно воспользоваться ресурсами языка для наиболее удачного решения поставленных задач.

В этой книге описано множество примеров. Есть как относительно простые, так и довольно сложные. Но пусть последнее вас не пугает. К тому моменту, когда вы начнете их рассматривать, они не покажутся вам особенно трудными.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты dgurski@minsk.piter.com (издательство «Питер», компьютерная редакция).

На сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.



Окна

- ☐ Привлечение внимания к приложению
- ☐ Окно приложения
- ☐ Полупрозрачные окна
- ☐ Окна и кнопки нестандартной формы
- ☐ Немного о перемещении окон
- ☐ Масштабирование окон
- ☐ Добавление пункта в системное меню окна
- ☐ Отображение формы поверх других окон

Было решено начать книгу именно с необычных приемов использования оконного интерфейса. Причиной стало то, что при работе с операционной системой Windows мы видим окна постоянно и везде (отсюда, собственно, и название). Речь идет не только об окнах приложений, сообщений, свойств — понятие о таких окнах есть у любого начинающего пользователя Windows.

В своих собственных окнах рисуются и элементы управления (текстовые поля, панели инструментов, таблицы, полосы прокрутки, раскрывающиеся списки и т. д.). Взгляните на интерфейс, например, Microsoft Word. Здесь вы увидите, что даже содержимое документа находится в своем собственном окне с полосами прокрутки (правда, это не обязательно элемент управления). Окна элементов управления отличаются от «самостоятельных» окон (упрощенно) отсутствием стиля, позволяющего им иметь заголовков, а также тем, что являются дочерними по отношению к другим окнам. Понимание этого момента является важным, так как на нем основана часть примеров данной главы.

Рассматриваемые примеры частично используют средства, которые предусмотрены в Borland Delphi, а частично — возможности «чистого» API (см. гл. 2). Практически все API-функции работы с окнами требуют задания параметра типа `HWND` — дескриптора окна. Это уникальное значение, идентифицирующее каждое существующее в текущем сеансе Windows окно. В Delphi дескриптор окна формы и элемента управления хранится в параметре `Handle` соответствующего объекта.

Нужно также уточнить, что в этой главе термины «окно» и «форма» употребляются как синонимы, когда речь идет о форме (в том смысле, который это слово имеет при программировании с использованием Delphi). Когда же речь идет об элементах управления, то так и говорится: окно элемента управления.

1.1. Привлечение внимания к приложению

Начнем с простых примеров, позволяющих привлечь внимание пользователя к определенному окну приложения. Это может пригодиться в различных ситуациях: от уведомления пользователя об ошибке программы до простой сигнализации ему, какое окно в данный момент времени ожидает пользовательского ввода.

Инверсия заголовка окна

Вероятно, вы не раз могли наблюдать, как некоторые приложения после выполнения длительной операции или при возникновении ошибки как бы подмигивают. При этом меняется цвет кнопки приложения на Панели задач, а также состояние окна с активного на неактивное. Такой эффект легко достигим при использовании API-функции `FlashWindow` или ее усовершенствованного, но более сложного варианта — функции `FlashWindowEx`.



ПРИМЕЧАНИЕ

Здесь сказано, что функции изменяют цвет кнопки приложения на Панели задач. Однако этого не происходит при выполнении приведенных ниже примеров. Почему так получается и как с этим бороться, рассказано в разд. 1.2.

Первая из этих функций позволяет один раз изменить состояние заголовка окна и кнопки на **Панели задач** (листинг 1.1).

Листинг 1.1. Простая инверсия заголовка окна

```
procedure TForm1.cmbFlashOnceClick(Sender: TObject);
begin
    FlashWindow(Handle, True);
end;
```

Как видите, функция принимает дескриптор нужного окна и параметр (тип BOOL) инверсии. Если значение флага равно `True`, то состояние заголовка окна изменится на противоположное (из активного становится неактивным и наоборот). Если значение флага `False`, то состояние заголовка окна восстанавливается в свое первоначальное значение (активно или неактивно).

Более сложная функция `FlashWindowEx` в качестве дополнительного параметра (кроме дескриптора окна) принимает структуру `FLASHWINFO`, заполняя поля которой, можно настроить параметры мигания кнопки приложения и/или заголовка окна.

В табл. 1.1 приведено описание полей структуры `FLASHWINFO`.

Таблица 1.1. Поля структуры `FLASHWINFO`

Поле	Тип	Назначение
cbSize	UINT	Размер структуры <code>FLASHWINFO</code> (для отслеживания версий)
hwnd	HWND	Дескриптор окна
dwFlags	DWORD	Набор флагов, задающий режим использования функции <code>FlashWindowEx</code> . Значения этих флагов и их описания приведены после таблицы
uCount	UINT	Количество инверсий заголовка окна и/или кнопки на Панели задач
dwTimeout	DWORD	Время между изменениями состояния заголовка окна и/или кнопки на Панели задач . Если значение равно нулю, используется системное значение таймута

Значение параметра `dwFlags` формируется из приведенных ниже флагов с использованием операции побитового ИЛИ:

- `FLASHW_CAPTION` — инвертирует состояние заголовка окна;
- `FLASHW_TRAY` — заставляет мигать кнопку на **Панели задач**;
- `FLASHW_ALL` — сочетание `FLASHW_CAPTION` и `FLASHW_TRAY`;
- `FLASHW_TIMER` — периодическое изменение состояния заголовка окна и/или кнопки на **Панели задач** вплоть до того момента, пока функция `FlashWindowEx` не будет вызвана с флагом `FLASHW_STOP`;
- `FLASHW_TIMERNOFG` — периодическое изменение состояния заголовка окна и/или кнопки на **Панели задач** до тех пор, пока окно не станет активным;

- `FLASHW_STOP` — восстанавливает исходное состояние окна и кнопки на Панели задач.

Далее приведены два примера использования функции `FlashWindowEx`.

В первом — состояние заголовка окна и кнопки на Панели задач изменяется десять раз каждые 0,2 секунды (листинг 1.2).

Листинг 1.2. Десятикратная инверсия заголовка окна

```
procedure TForm1.cmbInverse10TimesClick(Sender: TObject);
var
    fl: FLASHWINFO;
begin
    fl.cbSize := SizeOf(fl);
    fl.hwnd := Handle;
    fl.dwFlags := FLASHW_CAPTION or FLASHW_TRAY; // аналогично
    FLASHW_ALL
    fl.uCount := 10;
    fl.dwTimeout := 200;
    FlashWindowEx(fl);
end;
```

Второй пример демонстрирует использование флагов `FLASHW_TIMER` и `FLASHW_STOP` для инверсии заголовка окна в течение заданного промежутка времени (листинг 1.3).

Листинг 1.3. Инверсия заголовка окна в течение определенного промежутка времени

```
//Запуск процесса периодической инверсии заголовка
procedure TForm1.cmbFlashFor4SecClick(Sender: TObject);
var
    fl: FLASHWINFO;
begin
    fl.cbSize := SizeOf(fl);
    fl.hwnd := Handle;
    fl.dwTimeout := 200;
    fl.dwFlags := FLASHW_ALL or FLASHW_TIMER;
    fl.uCount := 0;
    FlashWindowEx(fl);

    Timer1.Enabled := True;
end;
```

```
//Остановка инверсии заголовка
procedure TForm1.Timer1Timer(Sender: TObject);
var
    fl: FLASHWINFO;
begin
    fl.cbSize := SizeOf(fl);
    fl.hwnd := Handle;
    fl.dwFlags := FLASHW_STOP;
    FlashWindowEx(fl);

    Timer1.Enabled := False;
end;
```

В данном примере подразумевается использование таймера, срабатывающего каждые четыре секунды. Таймер первоначально неактивен. Конечно, можно было бы не использовать его, а просто посчитать количество инверсий, попадающих в нужный интервал времени (в данном случае четыре секунды), и задать его в поле `uCount`. Но приведенный пример рассчитан именно на демонстрацию использования флагов `FLASHW_TIMER` и `FLASHW_STOP`.

Активизация окна

Рассмотрим другой, гораздо более гибкий способ привлечение внимания к окну приложения. Он базируется на использовании API-функции `SetForegroundWindow`. Функция принимает один единственный параметр — дескриптор окна. Если выполняется ряд условий, то окно с заданным дескриптором будет выведено на передний план и пользовательский ввод будет направлен в это окно. Функция возвращает нулевое значение, если не удалось сделать окно активным.

В приведенном ниже примере окно активизируется при каждом срабатывании таймера (листинг 1.4).

Листинг 1.4. Активизация окна

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    SetForegroundWindow(Handle);
end;
```

В операционных системах старше Windows 95 и Windows NT 4.0 введен ряд ограничений на действие функции `SetForegroundWindow`. Так, приведенный выше пример как раз и является одним из случаев недружественного использования активизации окна, но это всего лишь пример.

Чтобы активизировать окно, процесс должен быть не фоновым либо должен иметь право устанавливать активное окно, назначенное ему другим процессом с таким

правом, и т. п. Все возможные нюансы в пределах одного трюка рассматривать не имеет смысла. Стоит отметить, что в случае, когда окно не может быть активизировано, автоматически вызывается функция `FlashWindow` для окна приложения (заставляет мигать кнопку этого приложения на **Панели задач**). Поэтому даже при неудачном вызове функции `SetForegroundWindow` приложение, нуждающееся во внимании, не останется незамеченным.

1.2. Окно приложения

Обратите внимание на то, что название приложения, указанное на кнопке, расположенной на **Панели задач**, совпадает в названием проекта (можно установить на вкладке **Application** диалога **Project options**, вызываемого командой меню **Project ▶ Options**), но не с заголовком главной формы приложения. Взгляните на приведенный ниже код, который можно найти в DPR-файле (несущественная часть опущена).

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

В конструкторе класса `TApplication`, экземпляром которого является глобальная переменная `Application` (ее объявление находится в модуле `Forms`), происходит неявное создание главного окна приложения. Заголовок именно этого окна отображается на **Панели задач** (кстати, этот заголовок можно также изменить с помощью свойства `Title` объекта `Application`). Дескриптор главного окна приложения можно получить при помощи свойства `Handle` объекта `Application`.

Главное окно приложения делается невидимым (ему задается нулевая высота и ширина), чтобы создавалась иллюзия его отсутствия и можно было считать, что главной является именно создаваемая первой форма.

Для подтверждения вышесказанного можно отобразить главное окно приложения, используя следующий код (листинг 1.5).

Листинг 1.5. Показываем окно приложения

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    SetWindowPos(Application.Handle, 0, 0, 0, 200, 100,
        SWP_NOZORDER or SWP_NOMOVE);
end;
```

В результате ширина окна станет равной 200, а высота — 100, и мы сможем посмотреть на главное окно. Кстати, можно заметить, что при активизации этого окна

(например, щелчке кнопкой мыши на заголовке) фокус ввода немедленно передается созданной первой, то есть главной, форме.

Теперь должно стать понятно, почему не мигала кнопка приложения при использовании функций `FlashWindow` или `FlashWindowEx`. Недостаток этот можно легко устранить, например, следующим образом (листинг 1.6).

Листинг 1.6. Мигание кнопки приложения на Панели задач

```
procedure TForm1.Button2Click(Sender: TObject);
var
    fl: FLASHWINFO;
begin
    fl.cbSize := SizeOf(fl);
    fl.hwnd := Application.Handle;
    fl.dwFlags := FLASHW_ALL;
    fl.uCount := 10;
    fl.dwTimeout := 200;
    FlashWindowEx(fl);
end;
```

В данном случае будет одновременно инвертироваться заголовок окна приложения. Убедиться в этом можно, предварительно применив листинг 1.5. Наконец, чтобы добиться одновременного мигания кнопки приложения на Панели задач и заголовка формы (произвольной, а не только главной), можно применить листинг 1.7.

Листинг 1.7. Мигание кнопки приложения и инверсия заголовка формы

```
procedure TForm1.Button3Click(Sender: TObject);
var
    fl: FLASHWINFO;
begin
    //Мигание кнопки
    fl.cbSize := SizeOf(fl);
    fl.hwnd := Application.Handle;
    fl.dwFlags := FLASHW_TRAY;
    fl.uCount := 10;
    fl.dwTimeout := 200;
    FlashWindowEx(fl);

    //Инверсия заголовка
    fl.cbSize := SizeOf(fl);
    fl.hwnd := Handle;
```

```

fl.dwFlags := FLASHW_CAPTION;
fl.uCount := 10;
fl.dwTimeout := 200;
FlashWindowEx(fl);
end;

```

В данном случае инвертируется заголовок формы `Form1`. Кнопка на Панели задач может не только мигать, но и, например, быть скрыта или показана, когда в этом есть необходимость. Так, для скрытия кнопки приложения можно применить API-функцию `ShowWindow` следующим образом:

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

Чтобы показать кнопку приложения, можно ту же функцию `ShowWindow` вызвать со вторым параметром, равным `SW_NORMAL`.

1.3. Полупрозрачные окна

В Windows 2000 впервые появилась возможность использования прозрачности окон (в англоязычной документации такие полупрозрачные окна называются *Layered windows*). Достигается это заданием дополнительного стиля окна (о назначении и использовании оконных стилей можно узнать в гл. 2). Здесь мы не будем рассматривать использование API-функций для работы с полупрозрачными окнами, так как их поддержка реализована для форм Delphi. Соответствующие свойства включены в состав класса `TForm`.

- `AlphaBlend` — включение/выключение прозрачности. Если `True`, то прозрачность включена, если `False`, то выключена.
- `AlphaBlendValue` — значение, обратное прозрачности окна (от 0 до 255). Если 0, то окно полностью прозрачно, если 255, то окно непрозрачно.

Значения перечисленных свойств можно изменять как из окна *Object Inspector*, так и во время выполнения программы (рис. 1.1).

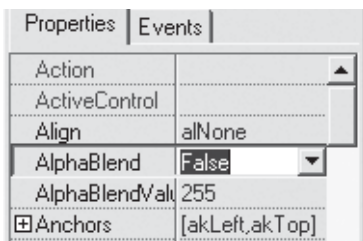


Рис. 1.1. Свойства для настройки прозрачности в окне *Object Inspector*

На рис. 1.2 наглядно продемонстрировано, как может выглядеть полупрозрачное окно (форма Delphi).

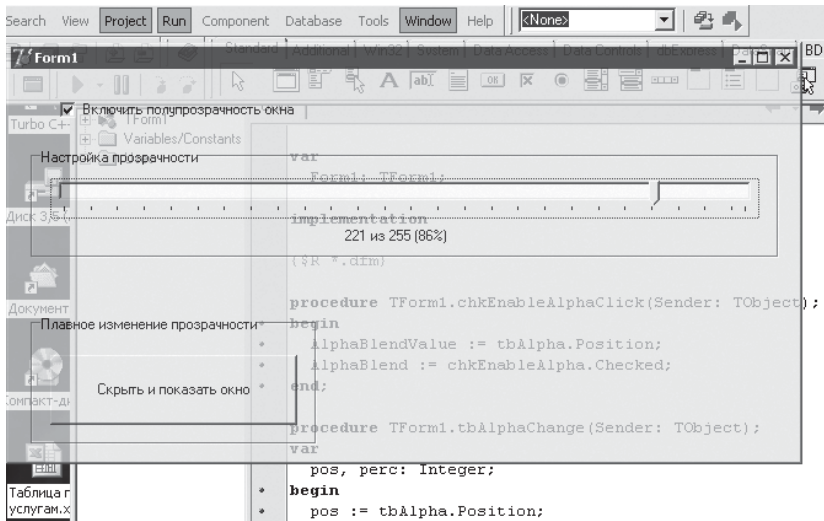


Рис. 1.2. Форма, прозрачная на 14 %

Ниже для примера рассмотрим, как применяются свойства `AlphaBlend`, а также `AlphaBlendValue` для задания прозрачности окна во время выполнения программы (сочетание ползунка `tbAlpha`, флажка `chkEnableAlpha` и подписи `lblCurAlpha` на форме рис. 1.2) (листинг 1.8).

Листинг 1.8. Динамическое изменение прозрачности окна

```

procedure TForm1.chkEnableAlphaClick(Sender: TObject);
begin
  AlphaBlendValue := tbAlpha.Position;
  AlphaBlend := chkEnableAlpha.Checked;
end;

procedure TForm1.tbAlphaChange(Sender: TObject);
var
  pos, perc: Integer;
begin
  pos := tbAlpha.Position;
  //Новое значение прозрачности
  AlphaBlendValue := pos;
  //Обновим подпись под ползунком
  perc := pos * 100 div 255;
  lblCurAlpha.Caption := IntToStr(pos) + ' из 255 (' +
    IntToStr(perc) + '%)';
end;

```

Довольно интересный эффект постепенного исчезновения, а затем появления формы можно реализовать, применив следующий код (листинг 1.9).

Листинг 1.9. Исчезновение и появление формы

```
implementation
var
    isInc: Boolean; // Если True, то значение AlphaBlend формы
                    // увеличивается, если False, то уменьшается
                    // (форма скрывается)
procedure TForm1.cmbHideAndShowClick(Sender: TObject);
begin
    if AlphaBlend then chkEnableAlpha.Checked := False;
    //Включаем прозрачность (подготовка к плавному скрывтию)
    AlphaBlendValue := 255;
    AlphaBlend := True;
    Refresh;

    //Запускаем процесс скрывтия формы
    isInc := False;
    Timer1.Enabled := True;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var val: Integer;
begin
    if not isInc then
    begin
        //"Растворение" окна
        val := AlphaBlendValue;
        Dec(val, 10);
        if val <= 0 then
        begin
            //Окно полностью прозрачно
            val := 0;
            isInc := True;
        end
    end
    else begin
```

```
//Появление окна
val := AlphaBlendValue;
Inc(val, 10);
if val >= 255 then
begin
    //Окно полностью непрозрачно
    val := 255;
    Timer1.Enabled := False; //Процесс закончен
    AlphaBlend := False;
end
end;
AlphaBlendValue := val;
end;
```

Главная сложность приведенного в листинге 1.9 алгоритма кроется в использовании таймера (Timer1) для инициирования изменения прозрачности окна. Так сделано для того, чтобы окно могло принимать пользовательский ввод, даже когда оно скрывается или постепенно показывается, и чтобы приложение не «съедало» все ресурсы на относительно слабой машине. Попробуйте сделать плавное изменение прозрачности в простом цикле, запустите его на каком-нибудь Pentium III 600 МГц без навороченной видеокарты и сами увидите, что станет с бедной машиной.

Грамотное, а главное уместное, использование прозрачности окон может значительно повысить привлекательность интерфейса приложения (взгляните хотя бы на Winamp 5 при включенном параметре прозрачности окон).

1.4. Окна и кнопки нестандартной формы

Сейчас мы рассмотрим некоторые стандартные возможности Windows, которые можно использовать для достижения большего разнообразия и привлекательности элементов оконного интерфейса за счет изменения формы элементов управления и, естественно, самих окон приложений.

Регионы. Создание и использование

Рассматриваемые далее эффекты по изменению формы окон базируются на использовании регионов (областей) отсечения — в общем случае сложных геометрических фигур, ограничивающих область рисования окна. По умолчанию окна (в том числе и окна элементов управления) имеют область отсечения, заданную прямоугольным регионом с высотой и шириной, равными высоте и ширине самого окна.

Однако использование прямоугольных регионов для указания областей отсечения совсем не обязательно. Использование отсечения по заданному непрямоугольному региону при рисовании произвольного окна наглядно представлено на рис. 1.3: *a* — изначальный прямоугольный вид формы; *b* — используемый регион, формирующий область отсечения; *в* — настоящий вид формы в результате рисования с отсечением по границам заданного региона.



Рис. 1.3. Использование области отсечения при рисовании окна

Рассмотрим операции, позволяющие создавать, удалять и модифицировать регионы.

Создание и удаление регионов

Создать регионы различной формы можно с помощью следующих API-функций:

```
function CreateRectRgn(p1, p2, p3, p4: Integer): HRGN;  
function CreateEllipticRgn(p1, p2, p3, p4: Integer): HRGN;  
function CreateRoundRectRgn(p1, p2, p3, p4, p5, p6: Integer): HRGN;
```

Все перечисленные здесь и ниже функции создания регионов возвращают дескриптор GDI-объекта «регион». Он впоследствии и передается в различные функции, работающие с регионами.

Итак, первая из приведенных функций (`CreateRectRgn`) предназначена для создания регионов прямоугольной формы. Параметры этой функции необходимо толковать следующим образом:

- `p1` и `p2` — горизонтальная и вертикальная координаты левой верхней точки прямоугольника;
- `p3` и `p4` — горизонтальная и вертикальная координаты правой нижней точки прямоугольника.

Следующая функция — `CreateEllipticRgn` — предназначена для создания региона эллиптической формы. Параметры этой функции — координаты прямоугольника (аналогично `CreateRectRgn`), в который вписывается эллипс.

Третья функция — `CreateRoundRectRgn` — создает регион — прямоугольник с округленными углами. При этом первые четыре параметра функции аналогичны соответствующим параметрам функции `CreateRectRgn`. Параметры `p5` и `p6` — ширина и высота сглаживающих углы эллипсов (рис. 1.4).

Трех приведенных функций достаточно даже в том случае, если нужно создавать регионы очень сложной формы. Это достигается при помощи многочисленных операций над простыми регионами, как в приведенном далее примере создания региона по битовому шаблону. Однако рассмотрим еще одну несложную функцию, которая позволяет сразу создавать регионы-многоугольники по координатам точек — вершин многоугольников:

```
function CreatePolygonRgn(const Points; Count, FillMode: Integer): HRGN;
```

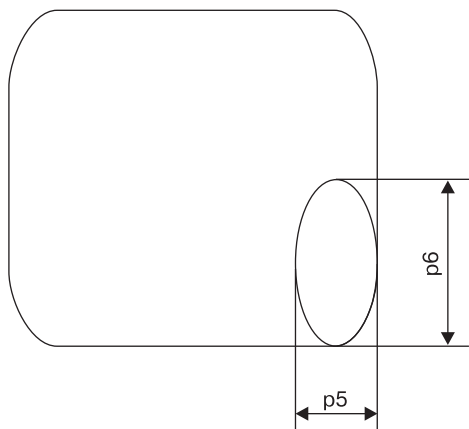


Рис. 1.4. Округление прямоугольника функцией `CreateRoundRectRgn`

Функция `CreatePolygonRgn` принимает следующие параметры:

- `Points` — указатель на массив записей типа `TPoint`, каждый элемент массива описывает одну вершину многоугольника, координаты не должны повторяться;
- `Count` — количество записей в массиве, на который указывает `Points`;
- `FillMode` — режим заливки региона (в данном случае, попадает ли внутренняя область многоугольника в регион).

Параметр `FillMode` принимает значения `WINDING` (попадает любая внутренняя область) или `ALTERNATE` (попадает внутренняя область, если она находится между нечетной и следующей четной сторонами многоугольника).



ПРИМЕЧАНИЕ

При создании регионов с помощью любой из указанных выше функций координаты точек задаются в системе координат того окна, в котором предполагается использовать регион. Так, если у нас есть кнопка 40×30 , левый верхний угол которой расположен на форме в точке $(100; 100)$, то для того, чтобы создать для кнопки прямоугольный регион 20×15 с левой верхней точкой $(0; 0)$ относительно начала координат кнопки, следует вызвать функцию `CreateRectRgn` с параметрами $(0, 0, 19, 14)$, а не $(100, 100, 119, 114)$.

Поскольку регион является GDI-объектом (подробнее в гл. 6), то для его удаления, если он не используется системой, применяется функция удаления GDI-объектов `DeleteObject`:

```
function DeleteObject(p1: HGDIOBJ): BOOL;
```

Регион как область отсечения при рисовании окна

Было сказано, что регион нужно удалять в том случае, если он не используется системой. Так вот, после того как регион назначен окну в качестве области отсечения, удалять его не следует. Функция назначения региона окну имеет следующий вид:

```
function SetWindowRgn(hWnd: HWND; hRgn: HRGN; bRedraw: BOOL): Integer;
```

Функция возвращает 0, если произвести операцию не удалось, и ненулевое значение в противном случае. Параметры функции `SetWindowRgn` следующие:

- `hWnd` — дескриптор окна, для которого устанавливается область отсечения (свойство `Handle` формы или элемента управления);
- `hRgn` — дескриптор региона, назначаемого в качестве области отсечения (в простейшем случае является значением, возвращенным одной из функций создания региона);
- `bRedraw` — флаг перерисовки окна после назначения новой области отсечения, для видимых окон обычно используется значение `True`, для невидимых — `False`.

Чтобы получить копию региона, формирующего область отсечения окна, можно использовать API-функцию `GetWindowRgn`:

```
function GetWindowRgn(hWnd: HWND; hRgn: HRGN): Integer;
```

Первый параметр функции — дескриптор (`Handle`) интересующего нас окна. Второй параметр — дескриптор предварительно созданного региона, который в случае успеха модифицируется функцией `GetWindowRgn` так, что становится копией региона, формирующего область отсечения окна. Описания целочисленных констант — возможных возвращаемых значений функции:

- `NULLREGION` — пустой регион;
- `SIMPLEREGION` — регион в форме прямоугольника;
- `COMPLEXREGION` — регион сложнее, чем прямоугольник;
- `ERROR` — при выполнении функции возникла ошибка (либо окну задана область отсечения).

Далее приводится пример использования функции `GetWindowRgn` (предполагается, что приведенный ниже код является телом одного из методов класса формы).

```
var rgn: HRGN;
begin
  rgn := CreateRectRgn(0,0,0,0); //Первоначальная форма
                                //региона не важна
  if GetWindowRgn(Handle, rgn) <> ERROR then
    begin
      //Операции с копией региона, формирующего область отсечения
      //окна...
    end;
  DeleteObject(rgn); //Мы пользовались копией региона, которую
                    //должны удалить (здесь или в ином месте,
                    //но сами)
end;
```

Операции над регионами

При рассказе о функциях создания регионов неоднократно упоминалось о возможности комбинирования регионов для получения сложных форм. Пришло время кратко рассмотреть операции над регионами. Все операции по комбинированию регионов осуществляются при помощи функции `CombineRgn`:

```
function CombineRgn(p1, p2, p3: HRGN; p4: Integer): Integer;
```

Параметры этой функции:

- `p1` — регион (предварительно созданный), куда сохранить результат;
- `p2, p3` — регионы-аргументы операции;
- `p4` — тип операции над регионами.

Более подробно действие `CombineRgn` при различных значениях параметра `p4` поясняется в табл. 1.2.

Таблица 1.2. Операции функции `CombineRgn`

Значение <code>p4</code>	Операция	Пример
<code>RGN_AND</code>	Пересечение регионов	
<code>RGN_OR</code>	Объединение регионов	
<code>RGN_DIFF</code>	Разность регионов (часть региона <code>p2</code> , не являющаяся частью <code>p3</code>)	
<code>RGN_XOR</code>	Так называемое исключающее ИЛИ (объединение непересекающихся частей регионов <code>p2</code> и <code>p3</code>)	

Кроме приведенных выше в табл. 1.2 констант, в качестве параметра `p4` функции `CombineRgn` можно использовать `RGN_COPY`. В этом случае копируется регион, заданный параметром `p2`, в регион, заданный параметром `p1`.

Тщательно рассчитывая координаты точек регионов-аргументов, можно с использованием функции `CombineRgn` создавать регионы самых причудливых форм, в чем вы сможете убедиться далее.

Наконец, после теоретического отступления рассмотрим несколько примеров создания и преобразования регионов с последующим их использованием для формирования области отсечения окон (форм и элементов управления на формах).

Закругленные окна и многоугольники

Сначала самые простые примеры: создание регионов без операций над ними. Формы всех трех приведенных здесь примеров содержат по три кнопки различной ширины и высоты, которым также задаются области отсечения.

**ПРИМЕЧАНИЕ**

В приведенных далее примерах регионы для области отсечения окна создаются при обработке события `FormCreate`. Однако это сделано только для удобства отладки и тестирования примеров и ни в коем случае не должно наталкивать вас на мысль, что этот способ является единственно правильным. На самом деле, если в приложении много окон, использующих области отсечения сложной формы, то старт приложения (время от момента запуска до показа первой формы) может длиться по крайней мере несколько секунд. Так происходит потому, что все формы создаются перед показом первой (главной) формы (см. DPR-файл проекта). Исправить ситуацию можно, создавая формы вручную в нужный момент времени либо создавая регионы для областей отсечения, например, перед первым отображением каждой конкретной формы.

Итак, в приведенном ниже обработчике события `FormCreate` создается окно в форме эллипса с тремя кнопками такой же формы (листинг 1.10).

Листинг 1.10. Окно и кнопки в форме эллипсов

```
procedure TfrmElliptic.FormCreate(Sender: TObject);
var
    formRgn, but1Rgn, but2Rgn, but3Rgn: HRGN;
begin
    //Создаем регионы кнопок
    but1Rgn := CreateEllipticRgn(0, 0, Button1.Width-1, Button1.
Height-1);
    SetWindowRgn(Button1.Handle, but1Rgn, False);

    but2Rgn := CreateEllipticRgn(0, 0, Button2.Width-1, Button2.
Height-1);
    SetWindowRgn(Button2.Handle, but2Rgn, False);

    but3Rgn := CreateEllipticRgn(0, 0, Button3.Width-1, Button3.
Height-1);
    SetWindowRgn(Button3.Handle, but3Rgn, False);

    //Регион для окна
    formRgn := CreateEllipticRgn(0, 0, Width-1, Height-1);
    SetWindowRgn(Handle, formRgn, True);
end;
```

Ширина и высота эллипсов в приведенном примере равна соответственно ширине и высоте тех окон, для которых создаются регионы (формы и каждой из кнопок). При необходимости это можно изменить, например, если требуется, чтобы

все кнопки были одной величины независимо от размера, установленного во время проектирования формы.

Результат работы листинга 1.10 можно увидеть на рис. 1.5.

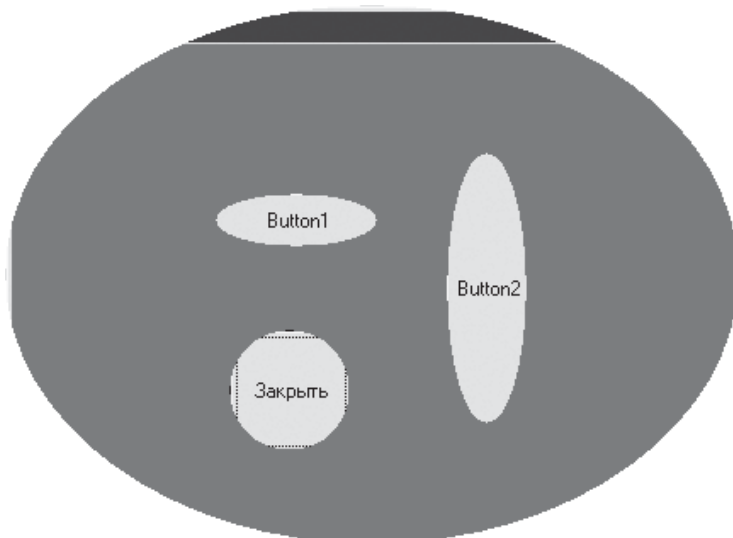


Рис. 1.5. Окно и кнопки в форме эллипсов

Далее рассмотрим не менее интересный (возможно, даже более полезный на практике) пример, а именно округление углов формы и кнопок на ней, то есть применение к ним области отсечения в форме прямоугольника с округленными углами. Ниже приводится реализация соответствующего обработчика события `FormCreate` (листинг 1.11).

Листинг 1.11. Окно и кнопки с округленными краями

```
procedure TfrmRoundRect.FormCreate(Sender: TObject);
var
    formRgn, but1Rgn, but2Rgn, but3Rgn: HRGN;
begin
    //Создаем регионы для кнопок
    but1Rgn := CreateRoundRectRgn(0, 0, Button1.Width-1,
                                   Button1.Height-1,
                                   Button1.Width div 5,
                                   Button1.Height div 5);
    SetWindowRgn(Button1.Handle, but1Rgn, False);

    but2Rgn := CreateRoundRectRgn(0, 0, Button2.Width-1,
                                   Button2.Height-1,
```

```
        Button2.Width div 5,  
        Button2.Height div 5);  
SetWindowRgn(Button2.Handle, but2Rgn, False);  
  
but3Rgn := CreateRoundRectRgn(0, 0, Button3.Width-1,  
        Button3.Height-1,  
        Button3.Width div 5,  
        Button3.Height div 5);  
SetWindowRgn(Button3.Handle, but3Rgn, False);  
  
//Регион для окна  
formRgn := CreateRoundRectRgn(0, 0, Width-1, Height-1,  
        Width div 5, Height div 5);  
SetWindowRgn(Handle, formRgn, False);  
end;
```

В листинге 1.11 размеры округляющих эллипсов вычисляются в зависимости от размеров конкретного окна (20 % от ширины и 20 % от высоты). Это смотрится не всегда красиво. В качестве альтернативы для ширины и высоты скругляющих эллипсов можно использовать фиксированные небольшие значения.

Результат работы листинга 1.11 можно увидеть на рис. 1.6.

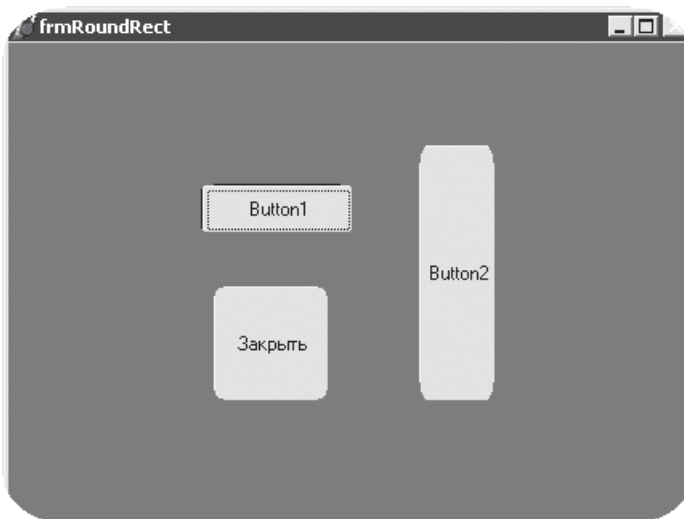


Рис. 1.6. Окно и кнопки с округленными краями

Теперь самый интересный из предусмотренных примеров — создание окна и кнопок в форме многоугольников, а конкретно: окна в форме звезды, кнопок в форме треугольника, пяти- и шестиугольника (рис. 1.7).

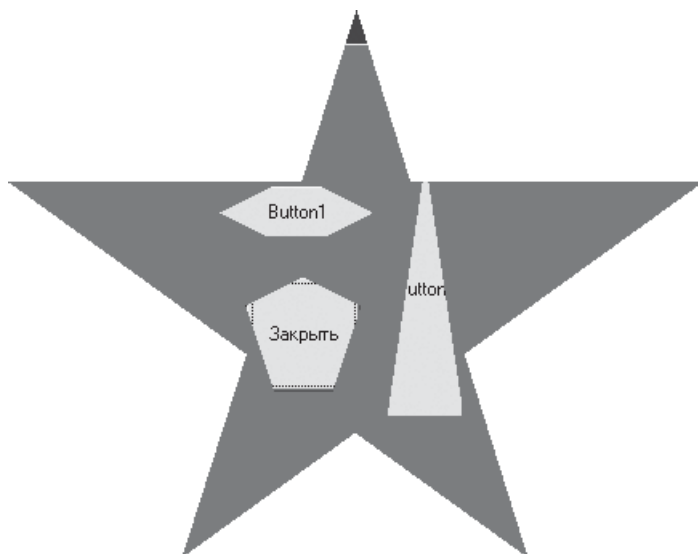


Рис. 1.7. Окно и кнопки в форме многоугольников

Создание регионов для областей отсечения формы, показанной на рис. 1.7, выглядит следующим образом (листинг 1.12).

Листинг 1.12. Окно и кнопки в форме многоугольников

```
procedure TfrmPoly.FormCreate(Sender: TObject);
var
  points: array [0..5] of TPoint;
  formRgn, but1Rgn, but2Rgn, but3Rgn: HRGN;
begin
  //Создаем регионы для окна и кнопок
  //...шестиугольная кнопка
  Make6Angle(Button1.Width, Button1.Height, points);
  but1Rgn := CreatePolygonRgn(points, 6, WINDING);
  SetWindowRgn(Button1.Handle, but1Rgn, False);
  //...треугольная кнопка
  Make3Angle(Button2.Width, Button2.Height, points);
  but2Rgn := CreatePolygonRgn(points, 3, WINDING);
  SetWindowRgn(Button2.Handle, but2Rgn, False);
  //...пятиугольная кнопка
  Make5Angle(Button3.Width, Button3.Height, points);
  but3Rgn := CreatePolygonRgn(points, 5, WINDING);
  SetWindowRgn(Button3.Handle, but3Rgn, False);
```

```
//...форма в виде звезды
MakeStar(Width, Height, points);
formRgn := CreatePolygonRgn(points, 5, WINDING);
SetWindowRgn(Handle, formRgn, False);
end;
```

Особенностью создания регионов в приведенном листинге является использование дополнительных процедур для заполнения массива `points` координатами точек-вершин многоугольников определенного вида. Все эти процедуры принимают, помимо ссылки на сам массив `points`, ширину и высоту прямоугольника, в который должен быть вписан многоугольник. Процедура создания треугольника приводится в листинге 1.13.

Листинг 1.13. Создание треугольника

```
procedure Make3Angle(width, height: Integer; var points: array
of TPoint);
begin
  points[0].X := 0;
  points[0].Y := height - 1;
  points[1].X := width div 2;
  points[1].Y := 0;
  points[2].X := width - 1;
  points[2].Y := height - 1;
end;
```

В листинге 1.14 приведена процедура создания шестиугольника.

Листинг 1.14. Создание шестиугольника

```
procedure Make6Angle(width, height: Integer; var points: array
of TPoint);
begin
  points[0].X := 0;
  points[0].Y := height div 2;
  points[1].X := width div 3;
  points[1].Y := 0;
  points[2].X := 2 * (width div 3);
  points[2].Y := 0;
  points[3].X := width - 1;
  points[3].Y := height div 2;
  points[4].X := 2 * (width div 3);
  points[4].Y := height - 1;
```

```
points[5].X := width div 3;  
points[5].Y := height - 1;  
end;
```

Листинг 1.15 содержит процедуру создания пятиугольника (неправильного).

Листинг 1.15. Создание пятиугольника

```
procedure Make5Angle(width, height: Integer; var points: array  
of TPoint);  
var a: Integer; //Сторона пятиугольника  
begin  
  a := width div 2;  
  
  points[0].X := a;  
  points[0].Y := 0;  
  points[1].X := width - 1;  
  points[1].Y := a div 2;  
  points[2].X := 3 * (a div 2);  
  points[2].Y := height - 1;  
  points[3].X := a div 2;  
  points[3].Y := height - 1;  
  points[4].X := 0;  
  points[4].Y := a div 2;  
end;
```

Пятиугольная звезда, используемая как область отсечения формы, создается при помощи приведенной в листинге 1.15 процедуры Make5Angle. После изменяется порядок следования вершин пятиугольника, чтобы их обход при построении региона выполнялся так, как рисуется звезда карандашом на бумаге (например, 1–3–5–2–4) (листинг 1.16).

Листинг 1.16. Создание пятиугольной звезды

```
procedure MakeStar(width, height: Integer; var points: array  
of TPoint);  
begin  
  Make5Angle(width, height, points);  
  //При построении звезды точки пятиугольника обводятся не по  
  //порядку, а через одну  
  Swap(points[1], points[2]);  
  Swap(points[2], points[4]);  
  Swap(points[3], points[4]);  
end;
```

Процедура `MakeStart` (листинг 1.16) использует дополнительную процедуру `Swap`, меняющую местами значения двух передаваемых в нее аргументов. Процедура `Swap` реализуется чрезвычайно просто и потому в тексте книги не приводится.

Комбинированные регионы

Вы уже научились создавать и использовать простые регионы. Однако многим может показаться недостаточным тех форм окон, которые получаются с использованием лишь одного несложного региона в качестве области отсечения. Пришло время заняться созданием окон более сложной формы, применяя рассмотренные ранее операции над регионами.

«Дырявая» форма

Этот простейший пример сомнительной полезности предназначен для знакомства с операциями над регионами. Здесь применяется только одна из возможных операций — операция XOR для формирования «дырок» в форме (рис. 1.8).



Рис. 1.8. «Дырки» в форме

На рис. 1.8 явно видно, как в «дырках» просвечивается одно из окон среды разработки Delphi. При этом сообщения от мыши, когда указатель находится над «дыркой», получает не наше окно, а те, часть которых видна в «дырке».

Программный код, приводящий к созданию формы столь необычного вида, приведен в листинге 1.17.

Листинг 1.17. Создание «дырок» в форме

```
procedure TfrmHole.FormCreate(Sender: TObject);
var
    rgn1, rgn2: HRGN; //"Регионы-дырки" в форме
    formRgn: HRGN;
begin
```

```
//Создание региона для формы
formRgn := CreateRectRgn(0, 0, Width - 1, Height - 1);
//Создание регионов для "дырок"
rgn1 := CreateEllipticRgn(10, 10, 100, 50);
rgn2 := CreateRoundRectRgn(10, 60, 200, 90, 10, 10);

//Создание "дырок" в регионе формы
CombineRgn(formRgn, formRgn, rgn1, RGN_XOR);
CombineRgn(formRgn, formRgn, rgn2, RGN_XOR);
SetWindowRgn(Handle, formRgn, True);

//Регионы для "дырок" больше не нужны
DeleteObject(rgn1);
DeleteObject(rgn2);
end;
```

Сложная комбинация регионов

Теперь пришла очередь более сложного, но и гораздо более интересного примера. Последовательное применение нескольких операций над регионами приводит к созданию формы, показанной на рис. 1.9 (белое пространство — это вырезанные части формы).

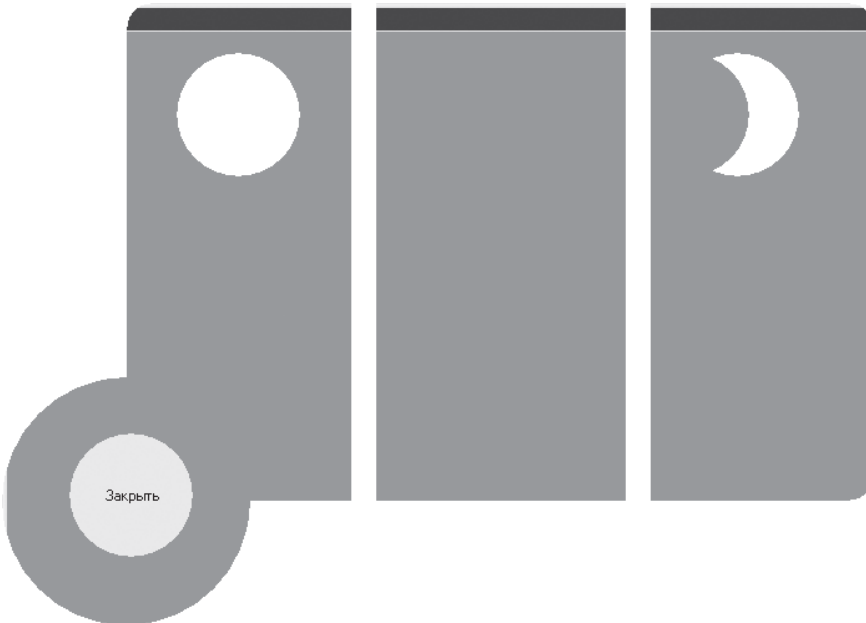


Рис. 1.9. Сложная комбинация регионов

Процедура, в которой производятся операции над регионами, приведена в листинге 1.18.

Листинг 1.18. Сложная комбинация регионов

```
procedure TfrmManyRgn.FormCreate(Sender: TObject);
var
  r1, r2, r3, r4, r5, r6, r7: HRGN;
  formRgn: HRGN;
  butRgn: HRGN;
begin
  //Создание регионов
  r1 := CreateRoundRectRgn(100, 0, 700, 400, 40, 40);
  r2 := CreateRectRgn(280, 0, 300, 399);
  r3 := CreateRectRgn(500, 0, 520, 399);
  r4 := CreateEllipticRgn(140, 40, 240, 140);
  r5 := CreateEllipticRgn(0, 300, 200, 500);
  r6 := CreateEllipticRgn(500, 40, 600, 140);
  r7 := CreateEllipticRgn(540, 40, 640, 140);

  //Комбинирование
  //..разрезы в основном регионе
  CombineRgn(r1, r1, r2, RGN_XOR);
  CombineRgn(r1, r1, r3, RGN_XOR);
  //..круглая "дырка" в правой стороне
  CombineRgn(r1, r1, r4, RGN_XOR);
  //..присоединение круга в левой нижней части
  CombineRgn(r1, r1, r5, RGN_OR);
  //..создание "дырки" в форме полумесяца
  CombineRgn(r7, r7, r6, RGN_DIFF);
  CombineRgn(r1, r1, r7, RGN_XOR);

  formRgn := CreateRectRgn(0, 0, 0, 0);
  CombineRgn(formRgn, r1, 0, RGN_COPY);

  DeleteObject(r1);
  DeleteObject(r2);
  DeleteObject(r3);
  DeleteObject(r4);
  DeleteObject(r5);
```

```
DeleteObject(r6);  
DeleteObject(r7);  
  
//Создание круглой кнопки закрытия  
butRgn := CreateEllipticRgn(50, 50, 150, 150);  
SetWindowRgn(Button1.Handle, butRgn, False);  
SetWindowRgn(Handle, formRgn, True);  
end;
```

В листинге подписано, какие операции для создания каких элементов итогового региона предназначены. В операциях участвуют семь регионов. Расположение используемых в операциях регионов показано на рис. 1.10.

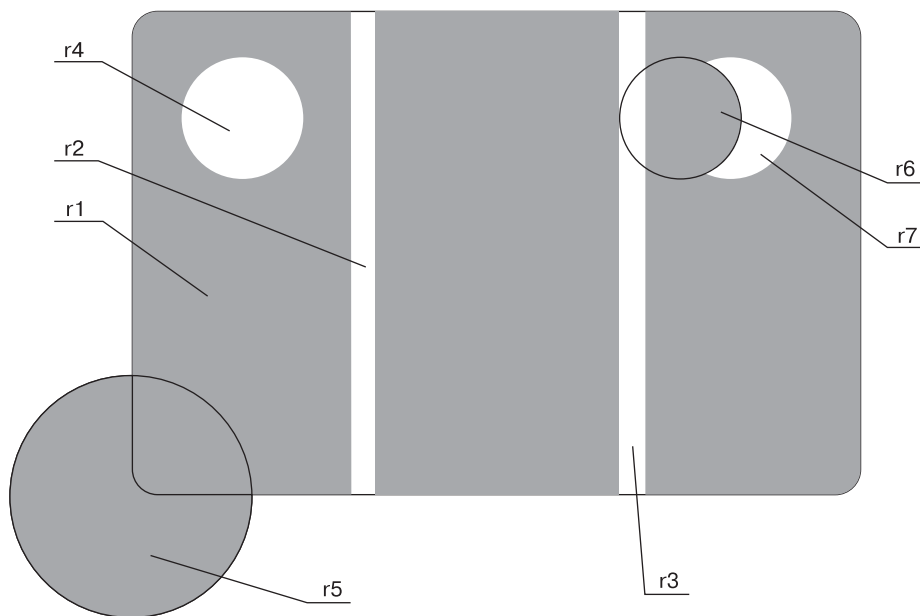


Рис. 1.10. Элементарные регионы, используемые для получения формы на рис. 1.9.

Использование шаблона

Предыдущий пример наглядно демонстрирует мощь функции `CombineRgn` при построении регионов сложной формы. Однако существует огромное количество предметов, контуры которых крайне сложно повторить, комбинируя простые регионы. Построение многоугольных регионов с большим количеством точек может в этом случае нас выручить, но ведь это крайне нудно и утомительно.

Если есть изображение предмета, контуры которого должны совпадать с контурами региона, то гораздо проще при построении региона обрабатывать само изображение, отбирая все точки, для которых выполняется определенное усло-

вие. Изображение и будет тем шаблоном, по которому «вырезается» регион нужной формы.

Рассмотрим простейший пример: есть монохромное изображение, каждая точка которого должна попасть в результирующий регион, если ее цвет не совпадает с заданным цветом фона. При этом изображение анализируется по так называемым «скан-линиям», то есть построчно. Из подряд идущих точек не фонового цвета формируются прямоугольные регионы, которые объединяются с результирующим регионом. Пример возможного шаблона приведен на рис. 1.11.



Рис. 1.11. Пример растрового изображения-шаблона

Функция построения региона указанным способом приведена в листинге 1.19.

Листинг 1.19. Построение региона по шаблону

```
function RegionFromPicture(pict: TPicture; bgcolor: TColor):  
    HRGN;  
var  
    rgn, resRgn: HRGN;  
    x, y, xFirst: Integer;
```

```

begin
    resRgn := CreateRectRgn(0, 0, 0, 0); //Результирующий регион
    //Анализируем каждую скан-линию рисунка (по горизонтали)
    for y := 0 to pict.Height - 1 do
    begin
        x := 0;
        while x < pict.Width do
        begin
            if (pict.Bitmap.Canvas.Pixels[x, y] <> backcolor) then
            begin
                xFirst := x;
                Inc(x);
                //Определим часть линии, окрашенной не цветом фона
                while (x < pict.Width) and
                    (pict.Bitmap.Canvas.Pixels[x, y] <> backcolor) do Inc(x);
                //Создаем регион для части скан-линии и добавляем его
                //к результирующему региону
                rgn := CreateRectRgn(xFirst, y, x-1, y+1);
                CombineRgn(resRgn, resRgn, rgn, RGN_OR);
                DeleteObject(rgn);
            end;
            Inc(x);
        end;
    end;
    RegionFromPicture := resRgn;
end;

```

Загрузка изображения-шаблона и создание региона может происходить, например, при создании формы следующим образом (листинг 1.20).

Листинг 1.20. Создание региона для области отсечения формы

```

procedure TfrmTemplate.FormCreate(Sender: TObject);
var
    pict: TPicture;
begin
    //Загрузка изображения и создание региона (считаем, что
    //цвет фона – белый)
    pict := TPicture.Create;

```

```
pict.LoadFromFile('back.bmp');  
SetWindowRgn(Handle, RegionFromPicture(pict, RGB(255,255,255)),  
True);  
end;
```

В листинге 1.20 подразумевается использование файла `back.bmp`, находящегося в той же папке, что и файл приложения. Цвет фона — белый. Таким образом, если шаблон, показанный на рис. 1.11, хранится в файле `back.bmp`, то получим форму, как на рис. 1.12.

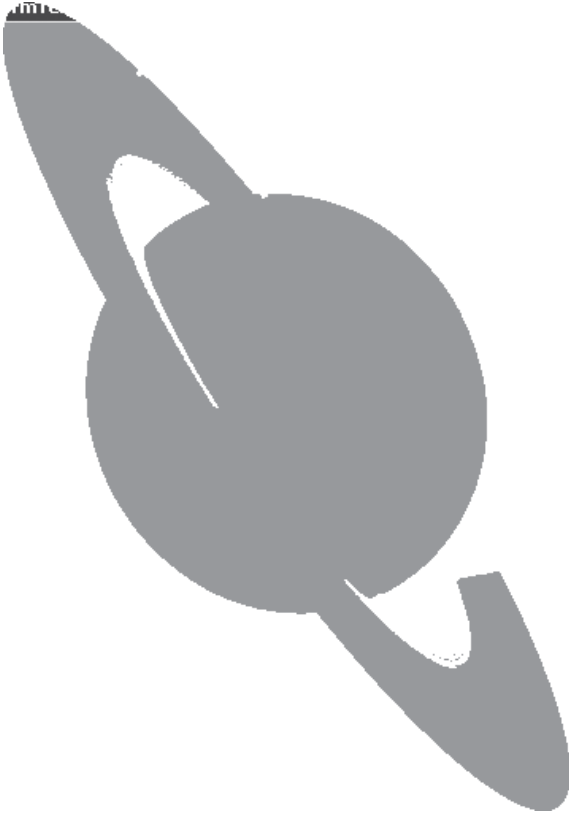


Рис. 1.12. Результат построения региона по шаблону

1.5. Немного о перемещении окон

Кроме придания необычного вида окнам приложения способами, рассмотренными выше, можно также несколько разнообразить интерфейс за счет оригинального использования перемещения окон. Далее показано, как самостоятельно назначать области, за которые можно перетаскивать (и не только) форму. Еще один пример демонстрирует способ дать пользователю возможность самому задавать расположение элементов управления на форме.

Перемещение за клиентскую область

Здесь на конкретном примере (перемещение формы за любую точку клиентской области) продемонстрировано, как можно самостоятельно определять положение некоторых важных элементов окна. Под элементами окна здесь подразумеваются:

- строка заголовка (не только предназначена для отображения текста заголовка, но и служит областью захвата при перемещении окна мышью);
- границы окна (при щелчке кнопкой мыши на верхней, нижней, правой и левой границе можно изменять размер окна, если, правда, стиль окна это допускает);
- четыре угла окна (предназначены для изменения размера окна при помощи мыши);
- системные кнопки — закрытия, разворачивания, сворачивания, контекстной справки (обычно расположены в строке заголовка окна);
- полосы прокрутки — горизонтальная и вертикальная;
- системное меню (раскрывается при щелчке кнопкой мыши на значке окна);
- меню — полоса меню, обычно вверху окна;
- клиентская область — по умолчанию все пространство окна, кроме строки заголовка, меню и полос прокрутки.

Каждый раз, когда над окном перемещается указатель мыши либо происходит нажатие кнопки мыши, система посылает соответствующему окну сообщение `WM_NCHITTEST` для определения того, над какой из перечисленных выше областей окна находится указатель. Обработчик этого сообщения, вызываемый по умолчанию, информирует систему о расположении элементов окна в привычных для нас местах: заголовка — сверху, правой границы — справа и т. д.

Как вы, скорее всего, уже догадались, реализовав свой обработчик сообщения `WM_NCHITTEST`, можно изменить назначение элементов окна. Этот прием как раз и используется в листинге 1.21.

Листинг 1.21. Перемещение окна за клиентскую область

```
procedure TfrmMoveClient.WMNCHitTest (var Message: TWMNCHitTest);
var
    rc: TRect;
    p: TPoint;
begin
    //Если точка приходится на клиентскую область, то заставим
    //систему считать эту область частью строки заголовка
    rc := GetClientRect();
    p.X := Message.XPos;
    p.Y := Message.YPos;
    p := ScreenToClient(p);
```

```
if PtInRect(rc, p) then
    Message.Result := HTCAPTION
else
    //Обработка по умолчанию
    Message.Result := DefWindowProc(Handle, Message.Msg, 0,
                                     65536 * Message.YPos + Message.XPos);
end;
```

Приведенный в листинге 1.21 обработчик переопределяет положение только строки заголовка, возвращая значение `HTCAPTION`. Этот обработчик может возвращать следующие значения (целочисленные константы, возвращаемые функцией `DefWindowProc`):

- `HTBORDER` — указатель мыши находится над границей окна (размер окна не изменяется);
- `HTBOTTOM`, `HTTOP`, `HTLEFT`, `HTRIGHT` — над нижней, верхней, левой или правой границей окна соответственно (размер окна можно изменить, «потянув» за границу);
- `HTBOTTOMLEFT`, `HTBOTTOMRIGHT`, `HTTOPLEFT`, `HTTOPRIGHT` — в левом нижнем, правом нижнем, левом верхнем или правом верхнем углу окна (размер окна можно изменять по диагонали);
- `HTSIZE`, `HTGROWBOX` — над областью, предназначенной для изменения размера окна по диагонали (обычно в правом нижнем углу окна);
- `HTCAPTION` — над строкой заголовка окна (за это место окно перемещается);
- `HTCLIENT` — над клиентской областью окна;
- `HTCLOSE` — над кнопкой закрытия окна;
- `HTHELP` — над кнопкой вызова контекстной справки;
- `HTREDUCE`, `HTMINBUTTON` — над кнопкой минимизации окна;
- `HTZOOM`, `HTMAXBUTTON` — над кнопкой максимизации окна;
- `HTMENU` — над полоской меню окна;
- `HTSYSTEMMENU` — над значком окна (используется для вызова системного меню);
- `HTHSCROLL`, `HTVSCROLL` — указатель находится над вертикальной или горизонтальной полосой прокрутки соответственно;
- `HTTRANSPARENT` — если возвращается это значение, то сообщение пересылается окну, находящемуся под данным окном (окна должны принадлежать одному потоку);
- `HTNOWHERE` — указатель не находится над какой-либо из областей окна (например, на границе между окнами);
- `HTERROR` — то же, что и `HTNOWHERE`, только при возврате этого значения обработчик по умолчанию (`DefWindowProc`) воспроизводит системный сигнал, говорящий об ошибке.

Перемещаемые элементы управления

В завершение материала о перемещении окон приведем один совсем несложный, но довольно интересный пример, позволяющий прямо «на лету» модифицировать внешний вид приложения, перемещая и изменяя размер элементов управления так, как будто это обычные перекрывающиеся окна.

Чтобы вас заинтересовать, сразу приведем результат работы примера. Итак, на рис. 1.13 показан внешний вид формы в начале работы примера.

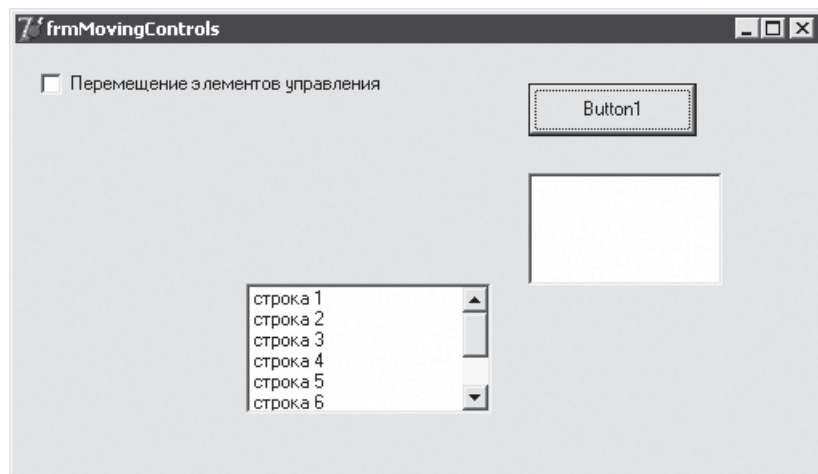


Рис. 1.13. Первоначальный вид формы

Теперь устанавливаем флажок Перемещение элементов управления и получаем результат, показанный на рис. 1.14.

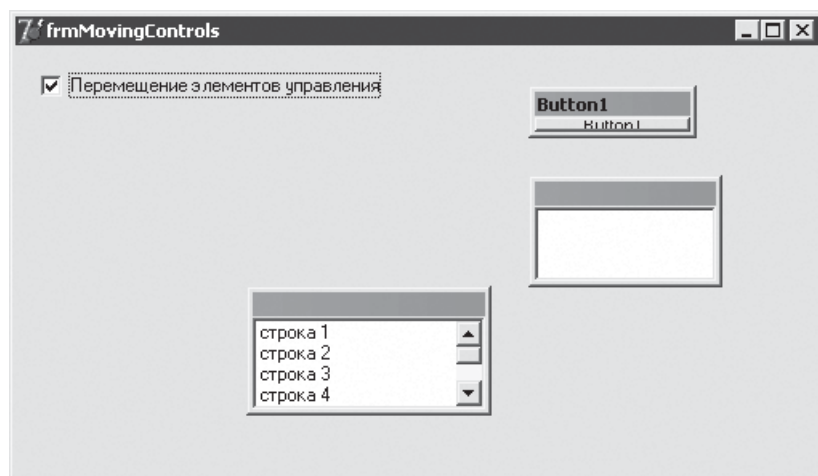


Рис. 1.14. Элементы управления можно перемещать (флажок не учитывается)

Выполняем произвольные перемещения, изменение размера окон, занявших место элементов управления, снимаем флажок и получаем результат — измененный интерфейс формы (рис. 1.15).

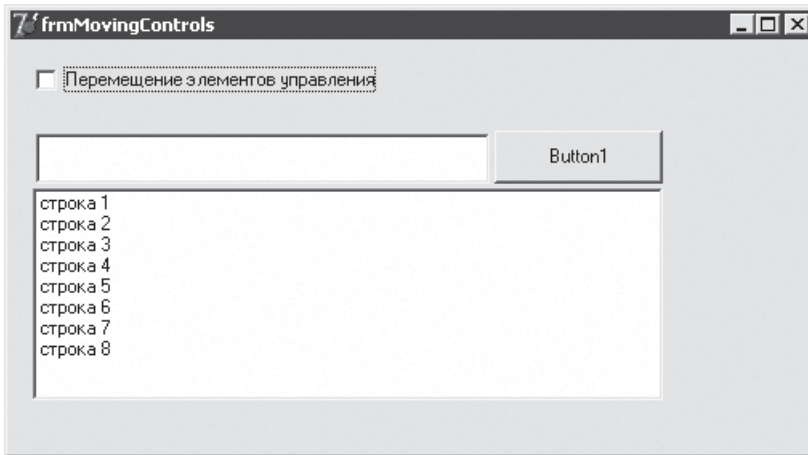


Рис. 1.15. Внешний вид формы после перемещения элементов управления

Как же достигнут подобный эффект? Очень даже просто. Ведь вы уже знаете, что элементы управления рисуются внутри своих собственных окон (дочерних по отношению к окну формы). Окна элементов отличается отсутствие в их стиле флагов (подробнее в гл. 2), позволяющих отображать рамку, изменять размер окна элемента управления. Это легко исправить, самостоятельно задав нужные флаги в стиле окна при помощи API-функции `SetWindowLong`. Для удобства можно написать отдельную процедуру, которая будет дополнять стиль окна флагами, необходимыми для перемещения и изменения размера (как, собственно, и сделано в примере) (листинг 1.22).

Листинг 1.22. Разрешение перемещения и изменения размера

```
procedure MakeMovable(Handle: HWND);
var
    style: LongInt;
    flags: UINT;
begin
    //Разрешаем перемещение элемента управления
    style := GetWindowLong(Handle, GWL_STYLE);
    style := style or WS_OVERLAPPED or WS_THICKFRAME or WS_CAPTION;
    SetWindowLong(Handle, GWL_STYLE, style);

    style := GetWindowLong(Handle, GWL_EXSTYLE);
    style := style or WS_EX_TOOLWINDOW;
```

```
SetWindowLong(Handle, GWL_EXSTYLE, style);

//Перерисуем в новом состоянии
flags := SWP_NOMOVE or SWP_NOSIZE or SWP_DRAWFRAME or
  SWP_NOZORDER;
SetWindowPos(Handle, 0, 0, 0, 0, 0, flags);
end;
```

Как можно увидеть, задание дополнительных флагов происходит в два этапа. Сначала считывается старое значение стиля окна. Потом при помощи двоичной операции ИЛИ стиль (а это целочисленное значение) дополняется новыми флагами. Это делается для того, чтобы не пропали ранее установленные значения стиля окна.

Вообще, процедура `MakeMovable` изменяет два стиля окна: обычный и расширенный. Расширенный стиль окна изменяется лишь для того, чтобы строка заголовка получившегося окна занимала меньше места (получаем так называемое окно панели инструментов). Полный перечень как обычных, так и расширенных стилей можно посмотреть в приложении 2.

Логично также реализовать процедуру, обратную `MakeMovable`, запрещающую перемещение окон элементов управления (листинг 1.23).

Листинг 1.23. Запрещение перемещения и изменения размера

```
procedure MakeUnmovable(Handle: HWND);
var
  style: LongInt;
  flags: UINT;
begin
  //Запрещаем перемещение элемента управления
  style := GetWindowLong(Handle, GWL_STYLE);
  style := style and not WS_OVERLAPPED and not WS_THICKFRAME
    and not WS_CAPTION;
  SetWindowLong(Handle, GWL_STYLE, style);

  style := GetWindowLong(Handle, GWL_EXSTYLE);
  style := style and not WS_EX_TOOLWINDOW;
  SetWindowLong(Handle, GWL_EXSTYLE, style);

  //Перерисуем в новом состоянии
  flags := SWP_NOMOVE or SWP_NOSIZE or SWP_DRAWFRAME or
    SWP_NOZORDER;
  SetWindowPos(Handle, 0, 0, 0, 0, 0, flags);
end;
```

Осталось только реализовать вызовы процедур `MakeMovable` и `MakeUnmovable` в нужном месте программы. В нашем примере вызовы заключены внутри обработчика изменения состояния флажка на форме (листинг 1.24).

Листинг 1.24. Управление перемещаемостью элементов управления

```
procedure TfrmMovingControls.chkSetMovableClick(Sender: TObject);
begin
    if chkSetMovable.Checked then
    begin
        //Разрешаем перемещение элементов управления
        MakeMovable(Memo1.Handle);
        MakeMovable(ListBox1.Handle);
        MakeMovable(Button1.Handle);
    end
    else
    begin
        //Запрещаем перемещение элементов управления
        MakeUnmovable(Memo1.Handle);
        MakeUnmovable(ListBox1.Handle);
        MakeUnmovable(Button1.Handle);
    end;
end;
```

1.6. Масштабирование окон

Возможность масштабирования окон (форм) является интересным приемом, который может быть заложен в дизайн приложения.

При этом имеется в виду масштабирование в буквальном смысле этого слова: как пропорциональное изменение размера элементов управления формы, так и изменение размера шрифта.

Использовать масштабирование при работе с Delphi крайне просто, ведь в класс `TWinControl`, от которого наследуются классы форм, встроены методы масштабирования. Вот некоторые из них:

- `ScaleControls` — пропорциональное изменение размера элементов управления на форме;
- `ChangeScale` — пропорциональное изменение размера элементов управления с изменением шрифта, которым выводится текст в них.

Оба приведенных метода принимают два целочисленных параметра: числитель и знаменатель нового масштаба формы. Пример задания параметров для методов масштабирования приводится в листинге 1.25.

Листинг 1.25. Масштабирование формы с изменением шрифта

```
procedure TfrmScaleBy.cmbSmallerClick(Sender: TObject);
begin
    ChangeScale(80, 100); //Уменьшение на 20 % (новый масштаб — 80 %)
end;
procedure TfrmScaleBy.cmbBiggerClick(Sender: TObject);
begin
    ChangeScale(120, 100); //Увеличение на 20 % (новый масштаб — 120 %)
end;
```

Чтобы размер шрифта правильно устанавливался, для элементов управления нужно использовать шрифты семейства TrueType (в нашем примере это шрифт Times New Roman).

На рис.1.16 приводится внешний вид формы до изменения масштаба.

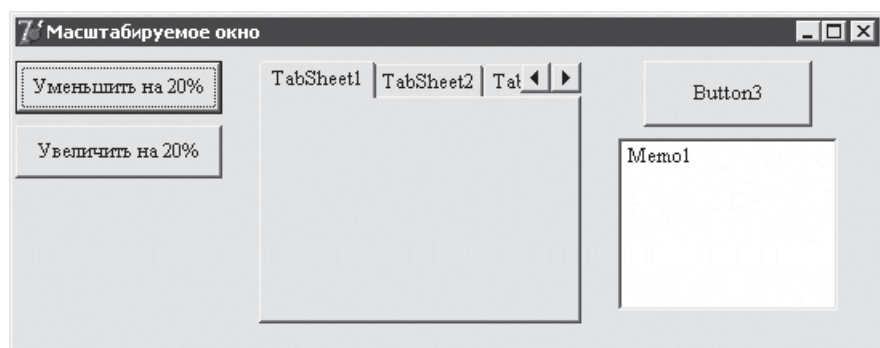


Рис. 1.16. Форма в оригинальном масштабе

Внешний вид формы после уменьшения масштаба в 1,25 раза (новый масштаб составляет 80 % от первоначального) демонстрируется на рис. 1.17.

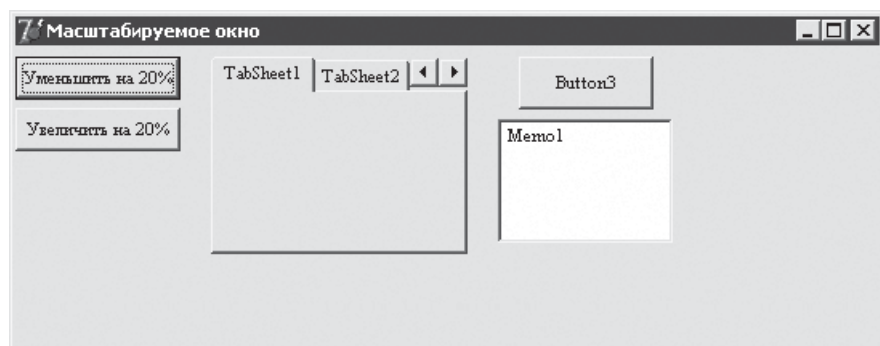


Рис. 1.17. Форма в масштабе 80 %

То, что форма не изменяет своего размера при масштабировании, можно легко исправить, установив, например, свойство `AutoSize` в `True` при помощи редактора свойств объектов (`Object Inspector`).

Если по каким-либо причинам использование свойства `AutoSize` вас не устраивает, то можно рассчитать новый размер формы самостоятельно. Только пересчитывать нужно размер не всего окна, а его клиентской области, ведь строка заголовка при масштабировании остается без изменений. Расчет размера окна можно выполнить так:

1. Получить прямоугольник клиентской области окна (`GetClientRect`).
2. Посчитать новый размер клиентской области.
3. Рассчитать разницу между новой и первоначальной шириной, новой и первоначальной высотой клиентской области; сложить полученные значения с первоначальными размерами самой формы.

Пример расчета приводится ниже (для увеличения размера клиентской области в 1,2 раза):

```
GetClientRect(Handle, rc);  
newWidth := (rc.Right - rc.Left) * 120 div 100;  
newHeight := (rc.Bottom - rc.Top) * 120 div 100;  
Width := Width + newWidth - (rc.Right - rc.Left);  
Height := Height + newHeight - (rc.Bottom - rc.Top);
```



ПРИМЕЧАНИЕ

Чтобы после уменьшения или увеличения масштаба формы можно было вернуться в точности к первоначальному масштабу (при помощи соответствующей обратной операции), нужно для уменьшения или увеличения использовать коэффициенты, произведение которых равно 1. Например, при уменьшении масштаба на 20 % (в 0,8 раза) его нужно увеличивать при обратной операции на 25 % (в $1/0,8 = 1,25$ раза).

1.7. Добавление пункта в системное меню окна

Обратите внимание на меню, раскрывающееся при щелчке кнопкой мыши на значке окна. В этом системном меню обычно присутствуют пункты, выполняющие стандартные действия над окном, такие, как закрытие, минимизация, максимизация и др. Однако есть функции, позволяющие получить доступ к этому меню, что дает возможность использовать его в своих целях.

Для получения дескриптора (`HMENU`) системного меню окна используем API-функцию `GetSystemMenu`, а для добавления пункта в меню — функцию `AppendMenu`. Пример процедуры, добавляющей пункты в системное меню, приведен в листинге 1.26.

Листинг 1.26. Добавление пунктов в системное меню окна

```
procedure TForm1.FormCreate(Sender: TObject);
var hSysMenu: HMENU;
begin
    hSysMenu := GetSystemMenu(Handle, False);
    AppendMenu(hSysMenu, MF_SEPARATOR, 0, '');
    AppendMenu(hSysMenu, MF_STRING, 10001, 'Увеличить на 20%');
    AppendMenu(hSysMenu, MF_STRING, 10002, 'Уменьшить на 20%');
end;
```

В результате системное меню формы `Form1` станет похожим на меню, показанное на рис. 1.18.

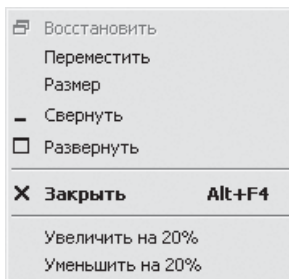


Рис. 1.18. Пользовательские пункты в системном меню

Однако мало просто создать пункты меню, нужно предусмотреть обработку их выбора. Это делается в обработчике сообщения `WM_SYSCOMMAND` (листинг 1.27).

Листинг 1.27. Обработка выбора пользовательских пунктов в системном меню

```
procedure TForm1.WMSysCommand(var Message: TWMSysCommand);
begin
    if Message.CmdType = 10001 then
        //Увеличение масштаба
        ChangeScale(120, 100)
    else if Message.CmdType = 10002 then
        //Уменьшение масштаба
        ChangeScale(80, 100)
    else
        //Обработка по умолчанию
        DefWindowProc(Handle, Message.Msg, Message.CmdType,
            65536 * Message.YPos + Message.XPos);
end;
```

Обратите внимание на то, что числовые значения, которые переданы в функцию `AppendMenu`, используются для определения, какой именно пункт меню выбран. Чтобы меню вело себя обычным образом, все поступающие от него команды должны быть обработаны. Поэтому для всех команд, реакция на которые не заложена в реализованном нами обработчике, вызывается обработчик по умолчанию (функция `DefWindowProc`).

1.8. Отображение формы поверх других окон

Иногда вам может пригодиться возможность отображения формы поверх всех окон. За примером далеко ходить не надо: посмотрите на окно Диспетчера задач Windows. А теперь вспомните, терялось ли хоть раз окно Свойства: Экран среди других открытых окон. Это происходит из-за того, что оно перекрывается другими окнами и при этом не имеет никакого значка на Панели задач (правда, это окно все же можно найти с помощью Диспетчера задач).

Из сказанного выше можно заключить, что как минимум в двух случаях отображение поверх других окон может пригодиться: для важных окон приложения (например, окно ввода пароля) и/или в случае, если значок приложения не выводится на Панели задач (как скрыть значок, было рассказано выше).

После небольшого отступления рассмотрим способы, позволяющие задать положение формы так, чтобы другие окна не могли ее закрыть.

Первый способ прост «до безобразия»: достаточно задать свойству `FormStyle` в окне `Object Inspector` значение `fsStayOnTo`. Результат этого действия показан на рис. 1.19 (обратите внимание, что форма закрывает Панель задач, которая по умолчанию также отображается поверх всех окон).



Рис. 1.19. Форма, отображаемая поверх других окон

Второй способ пригодится, если форма отображается постоянно как обычно, однако в определенные моменты времени требует к себе пристального внимания, для чего и помещается наверх. Способ основан на использовании API-функции `SetWindowPos`, которая кроме позиции и размера окна может еще устанавливать порядок рисования окна (Z-order).

**ПРИМЕЧАНИЕ**

Под Z-order подразумевается порядок следования окон вдоль оси Z, направленной перпендикулярно экрану (оси X и Y лежат в плоскости экрана).

Вызов функции `SetWindowPos` для помещения окна наверх выглядит следующим образом (`Handle` — дескриптор нужного окна):

```
SetWindowPos(Handle, HWND_TOPMOST, 0, 0, 0, 0, SWP_NOMOVE or  
SWP_NOSIZE)
```

В таком случае окно может быть закрыто другим окном, также отображающимся поверх других (например, Диспетчером задач).

Чтобы восстановить нормальное положение (порядок рисования) окна, можно вызвать функцию `SetWindowPos` со следующим набором параметров:

```
SetWindowPos(Handle, HWND_NOTOPMOST, 0, 0, 0, 0, SWP_NOMOVE or  
SWP_NOSIZE)
```

После этого другие, неотображаемые поверх остальных, окна могут снова перекрывать нашу форму.



Глава 2

Уменьшение размера EXE-файла. Использование Windows API

- ☐ Источник лишних килобайт
- ☐ Создание окна вручную
- ☐ Окно с элементами управления
- ☐ Стандартные диалоговые окна Windows
- ☐ Установка шрифта элементов управления

Не секрет, что размер скомпилированного EXE-файла Delphi часто значительно превосходит размер программ, написанных с использованием сред разработки от Microsoft (например, Visual C++, Visual Basic).



ПРИМЕЧАНИЕ

Здесь и далее имеются в виду приложения с оконным интерфейсом (не консольные).

При разработке крупных проектов этот факт абсолютно не смущает. Однако что же делать, если программисту на Delphi нужно написать программу, занимающую как можно меньше места (например, инсталлятор) или загружающуюся за минимальное время (например, сервисную программу). Конечно, такое приложение можно написать на C++, но что делать, если осваивать новый язык программирования нет времени?

В этой главе будут рассмотрены два способа уменьшения размера EXE-файла: отказ от библиотеки Borland за счет прямого использования Windows API и разбиение приложения на несколько DLL. Первый способ позволяет реально уменьшить размер приложения. Однако написание Delphi-приложения (да еще и с оконным интерфейсом) с использованием только API-функций является задачей весьма трудоемкой, хотя и интересной, да к тому же и экзотичной. Второй же способ не уменьшает размера проекта в целом, но может сэкономить время запуска приложения.

Вначале небольшое отступление. Итак, операционная система (в нашем случае это Windows) предоставляет интерфейс для программирования внутри себя — набор функций, заключенных в нескольких системных библиотеках, называемый Windows API (Windows Application Programming Interface — интерфейс программирования Windows-приложений). Любой проект под Windows на любом языке программирования в конечном счете сводится именно к приложению, использующему функции Windows API. Только использование этих самых функций может быть как явным, так и скрытым за использованием библиотек, поставляемых вместе со средой программирования.

И еще один момент. В тексте постоянно говорится о Windows API, а не просто API. Это потому, что само понятие Application Programming Interface применяется ко многим системам, а не только к ОС, и уж тем более не только к Windows. Вот несколько примеров: UNIX API, Linux API, Oracle API (интерфейс для работы с СУБД Oracle) и т. д.



ПРИМЕЧАНИЕ

В книге описаны только те возможности Window API, которые непосредственно используются в примерах. Полное описание Windows API является слишком большой задачей, для которой не хватит и книги. Если вам захочется изучить или хотя бы узнать больше о Windows API, то можно обратиться к специализированным изданиям по этой теме. Однако никакое издание не заменит MSDN (огромная справочная система от Microsoft для Visual Studio).

Теперь выясним, за счет чего разрастается EXE-файл приложения при использовании среды программирования Delphi.

2.1. Источник лишних килобайт

Для начала создадим новый проект Windows-приложения (Project1.exe). По умолчанию оно создает и показывает одну пустую форму (объявлена в модуле Unit1.pas). Ничего менять не будем, просто скомпилируем и посмотрим размер EXE-файла. Больше 300 Кбайт — не многовато ли для такого простого приложения?

Кстати, простейшее оконное приложение, написанное на Visual C++ 6.0 (в Release-конфигурации, то есть без отладочной информации в EXE-файле) без использования MFC, имеет размер 28 Кбайт, с использованием библиотеки MFC (правда, окно диалоговое) — 20 Кбайт. Простейшее оконное приложение на Visual Basic 6.0 занимает всего 16 Кбайт.

Из-за чего такая разница? Посмотрим, какие библиотеки используются приложениями, написанными на этих языках программирования. Это можно сделать, например, с помощью программы Dependency Walker, входящей в комплект Microsoft Visual Studio (рис. 2.1).

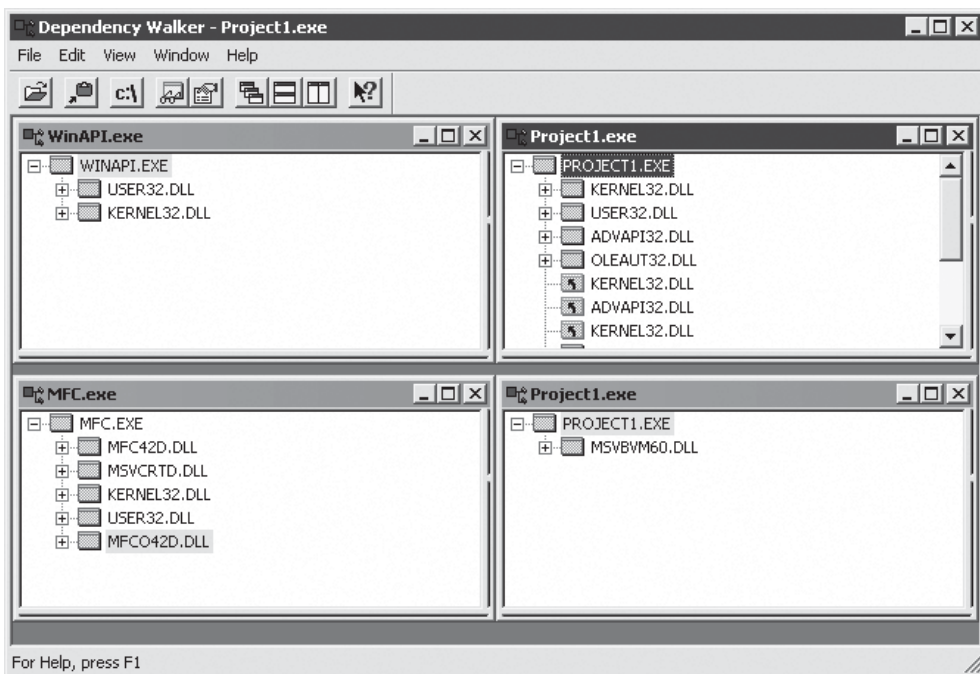


Рис. 2.1. Библиотеки, используемые приложениями

Как видим, приложение на Delphi (правый верхний угол окна на рис. 2.1) использует приличный набор функций, помещенных в стандартные библиотеки операционной

системы Windows. Кроме библиотек операционной системы, приложение на Delphi ничего не использует.

Приложение `WinAPI.exe` (левое верхнее окно на рис. 2.1) является примером чистого Windows API приложения в том смысле, что в нем не задействованы библиотеки оболочки над API-функциями, каким-либо образом облегчающие программирование. Собственно, столько реально и «весит» простейшее оконное приложение.

С приложением `MFC.exe` уже интереснее: размер самого EXE-файла уменьшился за счет того, что часть кода работы с API-функциями переместилась в библиотеки. С приложением на Visual Basic (правое нижнее окно) еще интереснее — оно фактически представляет собой вызовы функций одной библиотеки, в которой и реализована вся поддержка программирования на этом языке (при детальном рассмотрении этой библиотеки в ней можно найти объявления встроенных функций Visual Basic).

К чему это все? А к тому, что приложения на других языках программирования (в данном случае речь идет о продуктах Microsoft) совсем не менее «тяжеловесны», чем приложения, написанные на Borland Delphi, если при их написании программист пользуется не только API-функциями. Особенно примечателен в этом случае пример исполняемого файла Visual Basic, который хотя и имеет малый размер, но требует наличия библиотеки, размер которой около 1,32 Мбайт. Программа на Visual C++ с использованием, например, MFC, в которой реализованы классы оболочки над функциями Windows API (правда, не только они), требует наличия нескольких DLL. Для Microsoft это не проблема, так как операционная система Windows выпускается именно этой компанией, а следовательно, обеспечить переносимость (здесь — работоспособность без установки) приложений, написанных с использованием ее же сред разработки, очень просто: достаточно добавить нужные библиотеки в состав ОС.

Что же в таком случае осталось сделать Borland? Дабы не лишать программиста возможности пользоваться библиотеками с реализацией самых полезных классов (VCL и не только), код с реализацией этих самых классов приходится компоновать в один файл с самой программой. Вот и получается, что реализация этих самых классов в EXE-файле может занимать места гораздо больше, чем реализация собственно приложения. Так в нашем случае и получилось.



ПРИМЕЧАНИЕ

Кстати, проект на Visual C++ также можно статически скомпоновать с библиотекой MFC (то есть включить код реализации классов в сам EXE-файл). Таким способом можно добиться независимости приложения от различных библиотек, кроме тех, что гарантированно поставляются с Windows. Но при этом размер EXE-файла рассмотренного выше приложения (в Release-конфигурации) возрастает до 192 Кбайт.

Теперь обратимся к нашему проекту на Delphi. Посмотрим, что записано в файлах `Unit1.pas` и `Project1.dpr`. Текст файла `Unit1.pas` приводится ниже (листинг 2.1).

Листинг 2.1. Содержимое Unit1.pas

```
unit Unit1;
interface
uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms,
    Dialogs;
type
    TForm1 = class(TForm)
    private
        { Private declarations }
    public
        { Public declarations }
    end;
var
    Form1: TForm1;
implementation
{$R *.dfm}
end.
```

Обратите внимание на секцию `uses`. Здесь можно увидеть подключение девяти модулей, объявление собственно класса формы `TForm1`, а также строку, указывающую компилятору на использование файла ресурсов. Все модули, кроме первых двух, — это уже труды компании Borland, облегчающие жизнь простым программистам. Модуль такого же рода используется и в файле `Project1.dpr` (листинг 2.2).

Листинг 2.2. Содержимое файла Project1.dpr

```
program Project1;
uses
    Forms,
    Unit1 in 'Unit1.pas' {Form1};
{$R *.res}
begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

Теперь обратим внимание на модули `Windows` и `Messages`. В первом определены константы, структуры данных, необходимые для работы с функциями `Windows API`, и, конечно же, объявлены импортируемые из системных библиотек

API-функции. В модуле `Messages` можно найти определения констант и структур для работы с Windows-сообщениями (об этом в подразд. «Реакция на события элементов управления» разд. 2.3).

Собственно, этих двух модулей должно хватить для того, чтобы реализовать оконное приложение, правда, использующее только стандартные функции `WindowsAPI`, стандартные элементы управления. В листинге 2.3 приведен пример элементарного Windows-приложения. Главное, на что сейчас стоит обратить внимание, — это размер приложения: всего 15 Кбайт.

Листинг 2.3. Элементарное приложение

```
program WinAPI;
uses
  Windows, Messages;
{$R *.res}
begin
  MessageBox(0, 'This is a test', 'Little application', MB_OK);
end.
```

Зачастую неоправданно полностью отказываться от классов, реализованных Borland. Но для чистоты эксперимента в этой главе рассмотрим радикальные примеры, построенные на использовании только `Windows API`.

2.2. Создание окна вручную

Раз уж речь зашла о приложениях с оконным интерфейсом, то самое время приступить к его реализации средствами `Windows API`. Итак, чтобы создать и заставить работать окно приложения, нужно выполнить следующие операции:

1. Зарегистрировать класс окна с использованием функции `RegisterClass` или `RegisterClassEx`.
2. Создать экземпляр окна зарегистрированного ранее класса.
3. Организовать обработку сообщений, поступающих в очередь сообщений.

Пример того, как можно организовать регистрацию класса окна, приведен в листинге 2.4.

Листинг 2.4. Регистрация класса окна

```
function RegisterWindow():Boolean;
var
  wcx: WNDCLASSEX;
begin
  ZeroMemory(Addr(wcx), SizeOf(wcx));
```

```
//Формирование информации о классе окна
wcx.cbSize := SizeOf(wcx);
wcx.hInstance := GetModuleHandle(nil);
wcx.hIcon := LoadIcon(0, IDI_ASTERISK); //Стандартный значок
wcx.hIconSm := wcx.hIcon;
wcx.hCursor := LoadCursor(0, IDC_ARROW); //Стандартный указатель
wcx.hbrBackground := GetStockObject(WHITE_BRUSH); //Серый
                                                    //цвет фона
wcx.style := 0;

//..самые важные параметры
wcx.lpszClassName := 'MyWindowClass'; //Название класса
wcx.lpfnWndProc := Addr(WindowFunc); //Адрес функции
                                                    //обработки сообщений

//Регистрация класса окна
RegisterWindow := RegisterClassEx(wcx) <> 0;
end;
```

Здесь существенным моментом является обязательное заполнение структуры WNDCLASSEX информацией о классе окна. Самой необычной вам должна показаться следующая строка:

```
wcx.lpfnWndProc := Addr(WindowFunc); //Адрес функции
                                                    //обработки сообщений
```

Здесь мы сохранили адрес функции WindowFunc (листинг 2.5) — обработчик оконных сообщений (называемый также оконной процедурой). После вызова функции RegisterClassEx система запомнит этот адрес и будет вызывать нашу функцию-обработчик каждый раз при необходимости обработать сообщение, пришедшее окну. Простейшая реализация функции WindowFunc приводится в листинге 2.5.

Листинг 2.5. Функция обработки сообщений

```
//Функция обработки сообщений
function WindowFunc(hWnd:HWND; msg:UINT; wParam:WPARAM;
                    lParam:LPARAM):LRESULT; stdcall;
var
  ps: PAINTSTRUCT;
begin
  case msg of
    WM_CLOSE:
      if (hWnd = hMainWnd) then
```

```

        PostQuitMessage(0);    //При закрытии окна — выход
WM_PAINT:
begin
    //Перерисовка содержимого окна
    BeginPaint(hWnd, ps);
    TextOut(ps.hdc, 10, 10, 'Текст в окне', 12);
    EndPaint(hWnd, ps);
end;
else
begin
    //Обработка по умолчанию
    WindowFunc := DefWindowProc(hWnd, msg, wParam, lParam);
    Exit;
end;
end;

WindowFunc := S_OK; //Сообщение обработано
end;

```

В этой функции реализована обработка сообщения WM_PAINT — запроса на перерисовку содержимого окна. Обработка сообщения WM_CLOSE предусмотрена для того, чтобы при закрытии главного окна происходил выход из приложения. Для всех остальных сообщений выполняется обработка по умолчанию.

Обратите особое внимание на прототип этой функции: типы возвращаемых значений, типы параметров и способ вызова функции должны быть именно такими, как в листинге 2.5. Возвращаемое значение зависит от конкретного сообщения. Чаще всего это S_OK (константа, равная 0) в случае успешной обработки сообщения.

Далее в листинге 2.6 приводится часть программы, собственно использующая регистрацию, создание окна, а также организующая обработку сообщений для созданного окна.

Листинг 2.6. Регистрация и создание окна. Цикл обработки сообщений

```

program Window;
uses
    Windows, Messages;
{$R *.res}
var
    hMainWnd: HWND;
    mess: MSG;
...

```

```
begin
  //Создание окна
  if not RegisterWindow() then Exit;
  hMainWnd := CreateWindow(
    'MyWindowClass', //Имя класса окна
    'Главное окно', //Заголовок окна
    WS_VISIBLE or WS_OVERLAPPEDWINDOW, //Стиль окна
                                     //(перекрывающееся, видимое)
    CW_USEDEFAULT, //Координата X по умолчанию
    CW_USEDEFAULT, //Координата Y по умолчанию
    CW_USEDEFAULT, //Ширина по умолчанию
    CW_USEDEFAULT, //Высота по умолчанию
    HWND(nil), //Нет родительского окна
    HMENU(nil), //Нет меню
    GetModuleHandle(nil),
    nil);

  //Запуск цикла обработки сообщений
  while (Longint(GetMessage(mess, HWND(nil), 0, 0)) <> 0)
  do begin
    TranslateMessage(mess);
    DispatchMessage(mess);
  end;
end.
```

В приведенном листинге 2.6 на месте многоточия должны находиться коды функций `WindowFunc` и `RegisterWindow`. При создании окна использовались только стили `WS_VISIBLE` и `WS_OVERLAPPEDWINDOWS`. Но это далеко не все возможные стили окон. В приложении 2 приводится список всех стилей окон (если другого не сказано, то стили можно комбинировать при помощи оператора `or`). Кроме функции `CreateWindow`, для создания окон можно использовать функцию `CreateWindowEx`. При этом появится возможность указать дополнительный (расширенный) стиль окна (первый параметр функции `CreateWindowEx`). Список расширенных стилей приводится все в том же приложении 2.

В конце листинга 2.6 расположен цикл обработки сообщений:

```
while (Longint(GetMessage(mess, hMainWnd, 0, 0)) > 0)
do begin
  TranslateMessage(mess);
  DispatchMessage(mess);
end;
```

Здесь API-функция `GetMessage` возвращает значения больше 0, пока в очереди не обнаружится сообщение `WM_QUIT`. В случае возникновения какой-либо ошибки функция `GetMessage` возвращает значение -1. Функция `TranslateMessage` преобразует сообщения типа `WM_KEYDOWN`, `WM_KEYUP`, `WM_SYSKEYDOWN` и `WM_SYSKEYUP` в сообщения символьного ввода (`WM_CHAR`, `WM_SYSCHAR`, `WM_DEADCHAR`, `WM_SYSDEADCHAR`). Функция `DispatchMessage` в общем случае (за исключением сообщения `WM_TIMER`) вызывает функцию обработки сообщений нужного окна.

Внешний вид самого окна, созданного в этом примере, приводится на рис. 2.2.



Рис. 2.2. Окно, созданное вручную

Кстати, пока размер приложения всего 16 Кбайт.

2.3. Окно с элементами управления

После того как мы рассмотрели создание простейшего окна, самое время позаботиться о его наполнении элементами управления. Для стандартных элементов управления в системе уже зарегистрированы классы окон. Их перечень следующий:

- `BUTTON` — оконный класс, реализующий работу обычной кнопки, флажка, переключателя и даже рамки для группы элементов управления (`GroupBox`);
- `COMBOBOX` — раскрывающийся список;
- `EDIT` — текстовое поле, может быть как однострочным, так и многострочным, с полосами прокрутки или без;
- `LISTBOX` — список;
- `SCROLLBAR` — полоса прокрутки;
- `STATIC` — статический текст (он же `Label`, надпись, метка и пр.), кроме текста, может содержать изображение.

Ввиду большого количества возможных стилей окон элементов управления их перечень здесь не приводится, но его можно найти в приложении 2.

Создание элементов управления

Целесообразно написать более краткие функции создания элементов управления, чтобы, формируя интерфейс формы «на лету», не приходилось «украшать» код

громоздкими вызовами функций `CreateWindow` или `CreateWindowEx`. Этим мы сейчас и займемся. Сразу необходимо отметить: предполагается, что все функции помещены в модуль (модуль `Controls` в файле `Controls.pas`), в котором объявлены глобальные переменные `hAppInst` и `hParentWnd`. Эти переменные инициализируются перед вызовом первой из перечисленных ниже процедур или функций создания и работы с элементами управления (инициализацию можно посмотреть в листинге 2.21).

**ВНИМАНИЕ**

Обратите внимание на параметр `id` функций создания и манипулирования элементами управления. Это целочисленное значение идентифицирует элементы управления в пределах родительского окна.

Итак, для создания обычных кнопок можно использовать функцию из листинга 2.7 (все рассмотренные далее функции создания элементов управления возвращают дескриптор созданного окна).

Листинг 2.7. Создание кнопки

```
function CreateButton(x, y, width, height, id: Integer;
                     caption : String):HWND;
begin
    CreateButton :=
        CreateWindow('BUTTON', PAnsiChar(caption), WS_CHILD or
                    WS_VISIBLE or BS_PUSHBUTTON or WS_TABSTOP,
                    x, y, width, height, hParentWnd, HMENU(id),
                    hAppInst, nil);
end;
```

Приведенная в листинге 2.8 функция создает флажок и устанавливает его.

Листинг 2.8. Создание флажка

```
function CreateCheck(x, y, width, height, id: Integer;
                    caption: String; checked: Boolean):HWND;
var
    res: HWND;
begin
    res :=
        CreateWindow('BUTTON', PAnsiChar(caption), WS_CHILD or
                    WS_VISIBLE or BS_AUTOCHECKBOX or WS_TABSTOP,
                    x, y, width, height, hParentWnd, HMENU(id),
                    hAppInst, nil);
    if ((res <> 0) and checked) then
```

```

    SendMessage(res, BM_SETCHECK, BST_CHECKED, 0);
                                                    //Флажок установлен

    CreateCheck := res;
end;
```

Следующая функция (листинг 2.9) создает переключатель. Если нужно, то он устанавливается. Новый переключатель может начинать новую группу переключателей, для чего нужно параметру `group` присвоить значение `True`.

Листинг 2.9. Создание переключателя

```

function CreateOption(x, y, width, height, id: Integer;
                    caption: String; group: Boolean;
                    checked: Boolean):HWND;

var
    res: HWND;
    nGroup: Integer;
begin
    if (checked) then nGroup := WS_GROUP else nGroup := 0;
    res :=
        CreateWindow('BUTTON', PAnsiChar(caption), WS_CHILD or
                    WS_VISIBLE or BS_AUTORADIOBUTTON or nGroup or
                    WS_TABSTOP, x, y, width, height, hParentWnd,
                    HMENU(id), hAppInst, nil);
    if ((res <> 0) and checked) then
        //Переключатель установлен
        SendMessage(res, BM_SETCHECK, BST_CHECKED, 0);
    CreateOption := res;
end;
```

Для создания подписанной рамки, группирующей элементы управления, можно воспользоваться функцией `CreateFrame`, приведенной в листинге 2.10.

Листинг 2.10. Создание рамки

```

function CreateFrame(x, y, width, height, id: Integer;
                    caption: String):HWND;

begin
    CreateFrame:=
        CreateWindow('BUTTON', PAnsiChar(caption), WS_CHILD or
                    WS_VISIBLE or BS_GROUPBOX, x, y, width, height,
                    hParentWnd, HMENU(id), hAppInst, nil);
end;
```

Для того чтобы создать раскрывающийся список (`ComboBox`), можно использовать функцию `CreateCombo` из листинга 2.11.

Листинг 2.11. Создание раскрывающегося списка

```
function CreateCombo(x, y, width, height, id: Integer):HWND;
begin
    CreateCombo:=
        CreateWindow('COMBOBOX', nil, WS_CHILD or WS_VISIBLE or
            CBS_DROPDOWN or CBS_AUTOHSCROLL or WS_TABSTOP,
            x, y, width, height, hParentWnd,
            HMENU(id), hAppInst, nil);
end;
```

Для создания простого списка (ListBox) вполне подойдет функция `CreateList` из листинга 2.12.

Листинг 2.12. Создание простого списка

```
function CreateList(x, y, width, height, id: Integer):HWND;
begin
    CreateList:=
        CreateWindowEx(WS_EX_CLIENTEDGE, 'LISTBOX', nil, WS_CHILD or
            WS_VISIBLE or LBS_NOTIFY or WS_BORDER or
            WS_TABSTOP, x, y, width, height,
            hParentWnd, HMENU(id), hAppInst, nil);
end;
```

Функция `CreateLabel` в листинге 2.13 создает статическую надпись (Label), предназначенную только для вывода текста.

Листинг 2.13. Создание надписи

```
function CreateLabel(x, y, width, height, id: Integer;
    caption: String):HWND;
begin
    CreateLabel:=
        CreateWindow('STATIC', PAnsiChar(caption), WS_CHILD or
            WS_VISIBLE, x, y, width, height, hParentWnd,
            HMENU(id), hAppInst, nil);
end;
```

Однострочное текстовое поле с привычной рамкой создается функцией `CreateEdit` (листинг 2.14).

Листинг 2.14. Создание однострочного текстового поля

```
function CreateEdit(x, y, width, height, id: Integer;
    strInitText: String):HWND;
begin
```

```
CreateEdit:=  
    CreateWindowEx(WS_EX_CLIENTEDGE, 'EDIT',  
        PAnsiChar(strInitText), WS_CHILD or  
        WS_VISIBLE or ES_AUTOHSCROLL or WS_TABSTOP,  
        x, y, width, height, hParentWnd,  
        HMENU(id), hAppInst, nil);  
end;
```

Создание многострочного текстового поля (Мемо) отличается от создания однострочного поля только указанием дополнительного флага `ES_MULTILINE` (листинг 2.15).

Листинг 2.15. Создание многострочного текстового поля

```
function CreateMemo(x, y, width, height, id: Integer;  
    strInitText: String):HWND;  
begin  
    CreateMemo:=  
        CreateWindowEx(WS_EX_CLIENTEDGE, 'EDIT',  
            PAnsiChar(strInitText),  
            WS_CHILD or WS_VISIBLE or ES_AUTOVSCROLL or  
            ES_MULTILINE or WS_TABSTOP,  
            x, y, width, height, hParentWnd,  
            HMENU(id), hAppInst, nil);  
end;
```

Приведенные здесь функции не претендуют на абсолютную универсальность и гибкость. Они введены для того, чтобы упростить создание элементов управления в тех частных случаях, которые приводятся далее в примерах этой главы.

Использование элементов управления

Элементы управления, как и все окна, управляются путем отправки им сообщений. Этим же способом они уведомляют родительские окна о некоторых произошедших событиях (например, выделение элемента в списке, нажатие кнопки и т. д.).

Описание наиболее используемых сообщений для рассматриваемых элементов управления приводится в приложении 3. Мы же рассмотрим, как можно упростить работу с элементами управления в некоторых частных случаях, написав для этого специальные функции.

Итак, в демонстрационном проекте для управления переключателями и флажками предусмотрены следующие функции и процедуры (листинг 2.16).

Листинг 2.16. Управление флажками и переключателями

```
//Установка/снятие флажка (установка/снятие переключателя)  
procedure SetChecked(id: Integer; checked: BOOL);
```

```
var state: Integer;
begin
    if (checked) then state := BST_CHECKED
    else state := BST_UNCHECKED;
    SendDlgItemMessage(hParentWnd, id, BM_SETCHECK, state, 0);
end;
//Получение информации о том, установлен ли флажок
//(установлен ли переключатель)
function GetChecked(id: Integer):BOOL;
begin
    if (SendDlgItemMessage(hParentWnd, id, BM_GETCHECK, 0, 0) =
        BST_CHECKED)
    then GetChecked := True
    else GetChecked := False;
end;
```

Функции и процедуры листинга 2.17 предназначены для управления элементом **ComboBox**.

Листинг 2.17. Управление раскрывающимся списком

```
//Добавление строки в список
procedure AddToCombo(id: Integer; str: String);
begin
    SendDlgItemMessage(hParentWnd, id, CB_ADDSTRING, 0,
        Integer(PAnsiChar(str)));
end;
//Удаление строки из списка
procedure DeleteFromCombo(id: Integer; index: Integer);
begin
    SendDlgItemMessage(hParentWnd, id, CB_DELETESTRING, index, 0);
end;
//Выделение строки с заданным номером
procedure SetComboSel(id: Integer; index: Integer);
begin
    SendDlgItemMessage(hParentWnd, id, CB_SETCURSEL, index, 0);
end;
//Получение номера выделенной строки (CB_ERR, если нет выделения)
function GetComboSel(id: Integer): Integer;
begin
    GetComboSel := SendDlgItemMessage(hParentWnd, id,
        CB_GETCURSEL, 0, 0);
end;
```

```
end;
//Получение количества строк
function GetComboCount(id: Integer): Integer;
begin
    GetComboCount := SendDlgItemMessage(hParentWnd, id,
                                         CB_GETCOUNT, 0, 0);
end;
//Получение текста строки по ее индексу
function GetComboItemText(id: Integer; index: Integer): String;
var buffer: String;
begin
    SetLength(buffer,
               SendDlgItemMessage(hParentWnd, id, CB_GETLBTEXTLEN,
                                   index, 0)
    );
    SendDlgItemMessage(hParentWnd, id, CB_GETLBTEXT, index,
                       Integer(Addr(buffer)));
    GetComboItemText := buffer;
end;
```

Сходные функции и процедуры в листинге 2.18 предназначены для управления элементом `ListBox`.

Листинг 2.18. Управление списком

```
//Добавление строки в список
procedure AddToList(id: Integer; str: String);
begin
    SendDlgItemMessage(hParentWnd, id, LB_ADDSTRING, 0,
                       Integer(PAnsiChar(str)));
end;
//Удаление строки из списка
procedure DeleteFromList(id: Integer; index: Integer);
begin
    SendDlgItemMessage(hParentWnd, id, LB_DELETETESTRING, index, 0);
end;
//Выделение строки с заданным номером
procedure SetListSel(id: Integer; index: Integer);
begin
    SendDlgItemMessage(hParentWnd, id, LB_SETCURSEL, index, 0);
end;
```

```
//Получение номера выделенной строки (LB_ERR, если нет выделения)
function GetListSel(id: Integer): Integer;
begin
    GetListSel := SendDlgItemMessage(hParentWnd, id,
                                     LB_GETCURSEL, 0, 0);
end;
//Получение количества строк
function GetListCount(id: Integer): Integer;
begin
    GetListCount := SendDlgItemMessage(hParentWnd, id,
                                       LB_GETCOUNT, 0, 0);
end;
//Получение текста строки по ее индексу
function GetListItemText(id: Integer; index: Integer):String;
var buffer: String;
begin
    SetLength(buffer,
               SendDlgItemMessage(hParentWnd, id, LB_GETTEXTLEN,
                                index, 0)
    );
    SendDlgItemMessage(hParentWnd, id, LB_GETTEXT, index,
                      Integer(Addr(buffer)));
    GetListItemText := buffer;
end;
```

Функции и процедуры листинга 2.19 дадут возможность управлять текстовыми полями (Edit и Memo).

Листинг 2.19. Управление текстовыми полями

```
//Получение позиции первого выделенного символа (нумерация с нуля)
function GetSelStart(id: Integer): Integer;
var selStart, selEnd: Integer;
begin
    SendDlgItemMessage(hParentWnd, id, EM_GETSEL,
                      Integer(Addr(selStart)),
                      Integer(Addr(selEnd)));
    GetSelStart := selStart;
end;
//Получение длины выделенного фрагмента текста
function GetSelLength(id: Integer): Integer;
```

```
var selStart, selEnd: Integer;
begin
    SendDlgItemMessage(hParentWnd, id, EM_GETSEL,
        Integer(Addr(selStart)),
        Integer(Addr(selEnd)));
    GetSelLength := selEnd - selStart;
end;
//Выделение фрагмента текста (позиция первого символа с нуля)
procedure SetSel(id: Integer; start, length: Integer);
begin
    SendDlgItemMessage(hParentWnd, id, EM_SETSEL, start,
        start + length);
end;
//Получение выделенного фрагмента текста
function GetSelText(id: Integer): String;
var allText: String;
begin
    allText := GetText(id);
    GetSelText := Copy(allText, GetSelStart(id)+1, GetSelLength(id));
end;
//Замена выделенного текста
procedure ReplaceSelText(id: Integer; newText: String);
begin
    SendDlgItemMessage(hParentWnd, id, EM_REPLACESEL,
        0, Integer(PAnsiChar(newText)));
end;
```

В листинге 2.20 приводятся функции и процедуры, которые можно с одинаковым успехом применять ко всем элементам управления.

Листинг 2.20. Общие функции и процедуры

```
//Установка текста окна
procedure SetText(id: Integer; str: String);
begin
    SetWindowText(GetDlgItem(hParentWnd, id), PAnsiChar(str));
end;
//Получение текста окна
function GetText(id: Integer): String;
var buffer: String;
```

```
begin
    SetLength(buffer, GetWindowTextLength(hParentWnd));
    GetWindowText(hParentWnd, PAnsiChar(buffer), Length(buffer));
    GetText := buffer;
end;
//Активизация/деактивизация окна
procedure SetEnabled(id: Integer; fEnabled: BOOL);
begin
    EnableWindow(GetDlgItem(hParentWnd, id), fEnabled);
end;
```

Реакция на события элементов управления

При возникновении какого-либо предусмотренного для элемента управления события родительскому окну посылается сообщение `WM_COMMAND`.



ПРИМЕЧАНИЕ

Сообщение `WM_COMMAND` приходит также при перерисовке так называемых «самоперерисовывающихся» (Owner Draw) элементов управления. Однако ввиду специфики данного вопроса и ограниченности объема главы мы его рассматривать не будем.

Итак, когда родительское окно получает сообщение `WM_COMMAND`, то из двух прилагающихся параметров (`lParam` и `wParam`) можно извлечь следующие сведения:

- старшие 16 бит `wParam` представляют собой целочисленный код уведомления, позволяющий определить, что же именно произошло с элементом управления;
- младшие 16 бит `wParam` представляют собой идентификатор элемента управления, состояние которого изменилось (именно этот идентификатор мы передавали вместо дескриптора меню при создании элементов управления);
- `lParam` содержит дескриптор (`HWND`) окна элемента управления, состояние которого изменилось.

Для выделения старших 16 бит из 32-битного значения можно использовать функцию `HiWord`. Для получения младших 16 бит можно использовать функцию с именем `LoWord`. Обе они объявлены в модуле `Windows`.

В качестве примеров можно привести следующие коды уведомлений:

- `BN_CLICKED` — нажата кнопка;
- `EN_CHANGE` — изменен текст в текстовом поле;
- `LBN_SELCHANGE` — изменилось выделение в списке;
- `CBN_SELCHANGE` — изменилось выделение в раскрывающемся списке.

Эти и все остальные константы уведомлений стандартных элементов управления объявлены в модуле Messages.



ПРИМЕЧАНИЕ

Коды уведомлений рассматриваемых в этой главе элементов управления приводятся в приложении 3.

Пример приложения

Рассмотрим небольшой пример, иллюстрирующий принцип работы с элементами управления, помещенными на форму описанным ранее способом. Проект этого приложения называется ControlsDemo.

Не будем заострять внимание на регистрации класса главного окна приложения, так как она аналогична приведенной в листинге 2.4. Рассмотрим создание окна с элементами управления в нем (листинг 2.21).

Листинг 2.21. Создание главного окна приложения (с элементами управления)

```
program ControlsDemo;
uses
  Windows, Messages,
  Controls in 'Controls.pas';
{$R *.res}
var
  hMainWnd: HWND;
  hInst: Cardinal;
  mess: MSG;

//Функция обработки сообщений
...

//Создание окна и цикла обработки сообщений
begin
  hInst := GetModuleHandle(nil);

  //Регистрация и создание главного окна
  if not RegisterWindow() then Exit;
  hMainWnd := CreateWindow(
    'MyWindowClass', //Имя класса окна
    'Главное окно', //Заголовок окна
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, //Координата X по умолчанию
```

```
CW_USEDEFAULT,    //Координата Y по умолчанию
CW_USEDEFAULT,    //Ширина по умолчанию
CW_USEDEFAULT,    //Высота по умолчанию
HWND(nil),        //Нет родительского окна
HMENU(nil),       //Нет меню
hInst,
nil);
if (hMainWnd = HWND(nil)) then Exit;

//Инициализация модуля Controls для работы с главным окном
Controls.hParentWnd := hMainWnd;
Controls.hAppInst := hInst;

//Создание элементов управления
CreateFrame(10, 80, 170, 70, -1, 'Кнопки');
CreateButton(20, 100, 70, 30, 1001, 'Кнопка 1');
CreateButton(100, 100, 70, 30, 1002, 'Кнопка 2');

CreateFrame(200, 10, 200, 180, -1, 'Флажки и переключатели');
CreateCheck(210, 30, 180, 20, 2001, 'Флажок 1');
CreateCheck(210, 60, 180, 20, 2002, 'Флажок 2', True);
CreateOption(210, 100, 180, 20, 3001, 'Переключатель 1', True);
CreateOption(210, 130, 180, 20, 3002, 'Переключатель 2', False,
True);
CreateOption(210, 160, 180, 20, 3003, 'Переключатель 3', True);

CreateFrame(420, 10, 300, 180, -1, 'Списки и статические
написи');
CreateLabel(430, 30, 70, 20, -1, 'Надпись');
CreateCombo(510, 30, 200, 100, 4001);
CreateList(430, 60, 280, 120, 5001);

CreateFrame(200, 200, 200, 240, -1, 'Текстовые поля');
CreateEdit(210, 220, 180, 20, 6001, 'Текст в текстовом поле');
CreateMemo(210, 250, 180, 180, 6002, 'Текст в многострочном'
+ #13 + #10 + 'текстовом поле');

//Добавление строк в списки
AddToCombo(4001, 'Строка 1');
AddToCombo(4001, 'Строка 2');
```

```
AddToCombo(4001, 'Строка 3');  
AddToList(5001, 'Строка 1');  
AddToList(5001, 'Строка 2');  
AddToList(5001, 'Строка 3');  
  
ShowWindow(hMainWnd, SW_NORMAL);  
  
//Запуск цикла обработки сообщений  
while (Longint(GetMessage(mess, 0, 0, 0)) <> 0)  
do begin  
    TranslateMessage(mess);  
    DispatchMessage(mess);  
end;  
end.
```

Листинг 2.21 заодно демонстрирует использование некоторых из приведенных ранее функций работы с элементами управления. Выглядит созданное окно так, как показано на рис. 2.3.

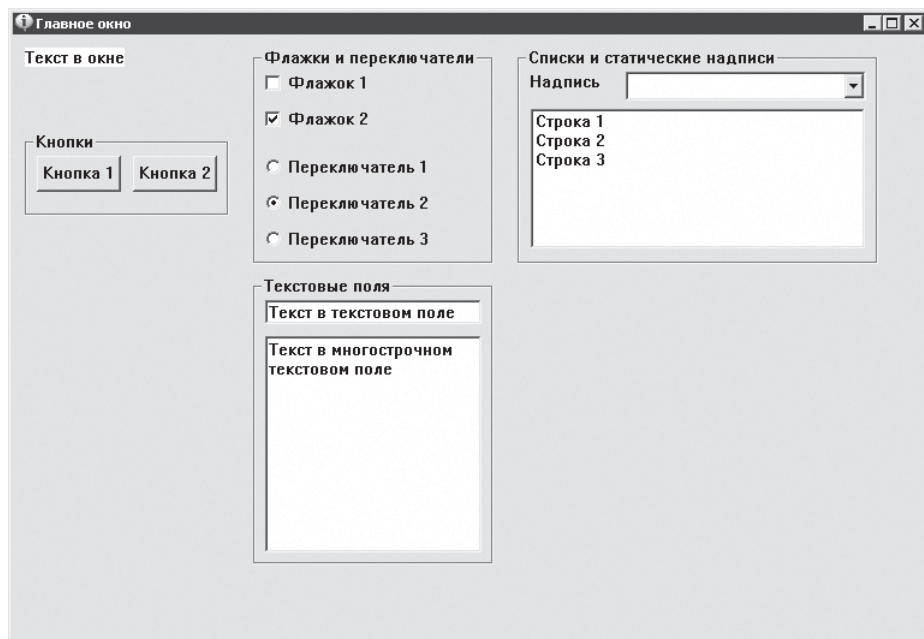


Рис. 2.3. Окно с элементами управления

Принцип построения функции обработки сообщений для этого окна приведен в листинге 2.22.

Листинг 2.22. Функция обработки сообщений

```
//Функция обработки сообщений
function WindowFunc(hWnd:HWND; msg:UINT; wParam:WPARAM;
                    lParam:LPARAM):LRESULT; stdcall;

var
  ps: PAINTSTRUCT;
begin
  case msg of
    WM_PAINT:
      begin
        //Перерисовка содержимого окна
        BeginPaint(hWnd, ps);
        TextOut(ps.hdc, 10, 10, 'Текст в окне', 12);
        EndPaint(hWnd, ps);
      end;
    WM_CLOSE:
      if (hWnd = hMainWnd) then
        PostQuitMessage(0); //При закрытии этого окна
                           //завершается приложение
    WM_COMMAND:
      begin
        case LOWORD(wParam) of
          //нажата "Кнопка 1"
          1001: if HIWORD(wParam) = BN_CLICKED then;
          //нажата "Кнопка 2"
          1002: if HIWORD(wParam) = BN_CLICKED then;
          //установлен "Флажок 1"
          2001: if HIWORD(wParam) = BN_CLICKED then;
          //установлен "Флажок 2"
          2002: if HIWORD(wParam) = BN_CLICKED then;
          //установлен "Переключатель 1"
          3001: if HIWORD(wParam) = BN_CLICKED then;
          //установлен "Переключатель 2"
          3002: if HIWORD(wParam) = BN_CLICKED then;
          //установлен "Переключатель 3"
          3003: if HIWORD(wParam) = BN_CLICKED then;
          //выделение в ComboBox
          4001: if HIWORD(wParam) = CBN_SELCHANGE then;
```

```
//выделение в ListBox
5001: if HIWORD(wParam) = LBN_SELCHANGE then;
//изменен текст в Edit
6001: if HIWORD(wParam) = EN_CHANGE then;
//изменен текст в Мемо
6002: if HIWORD(wParam) = EN_CHANGE then;
end;
end;
else
begin
//Обработка по умолчанию
WindowFunc := DefWindowProc(hWnd, msg, wParam, lParam);
Exit;
end;
end;
```

WindowFunc := S_OK; //Сообщение обработано

end;

Приведенная в листинге 2.22 функция отнюдь не претендует на то, чтобы быть эталоном в порядке классификации сообщений от элементов управления. Иногда бывает полезно сразу классифицировать сообщения не по элементам управления, которые их прислали, а по типу. К тому же в ряде случаев можно предусмотреть один обработчик сообщений сразу для нескольких элементов управления, например для группы переключателей. В таком случае полезным окажется параметр `lParam` сообщения `WM_COMMAND`.

Кстати, размер исполняемого файла этого приложения всего 19 Кбайт.

2.4. Стандартные диалоговые окна Windows

Теперь рассмотрим, как можно только при помощи функций Windows API вызывать некоторые распространенные диалоговые окна. Чтобы использовать API-функции и структуры с информацией для этих диалоговых окон, необходимо подключить следующие модули:

- `CommDlg` — для окон открытия/сохранения файла, выбора цвета и шрифта, поиска и замены текста;
- `ShlObj` и `ActiveX` — для окна выбора папки (второй модуль нужен для доступа к интерфейсу `IMalloc`, зачем — будет рассказано далее);
- `Windows` — помимо объявления основных структур и API-функций, этот модуль содержит объявления функций для работы с окнами подключения и отключения от сетевого ресурса (сетевого диска);
- `ShellAPI` — для системного окна 0 программе.

Вариант использования рассматриваемых в этом разделе диалоговых окон приведен в подразд. «Демонстрационное приложение».



ПРИМЕЧАНИЕ

В приведенных далее примерах вызова диалоговых окон можно увидеть не объявленные, но используемые в программах переменные `hApplInst` и `hParentWnd`. Подразумевается, что это глобальные переменные, которые инициализируются вне процедур и функций, приведенных в примерах. Для инициализации этих переменных можно также написать специальную процедуру, например с именем `Init`, в которую и передавать значения для `hParentWnd` и `hApplInst`.

Окно открытия/сохранения файла

Чтобы воспользоваться возможностями окна открытия файла, достаточно задействовать листинг 2.23.

Листинг 2.23. Окно открытия файла

```
function ShowOpen(strFilter: string; nFilterIndex: Integer = 0;
                  strInitFileName: string = '');
var
  ofn: OPENFILENAME;
begin
  ZeroMemory(Addr(ofn), SizeOf(ofn));
  //Формирование буфера (260 символов)
  SetLength(strInitFileName, MAX_PATH);
  PrepareFilterString(strFilter);

  //Заполнение структуры для диалога
  ofn.lStructSize := SizeOf(ofn);
  ofn.hWndOwner := hParentWnd;
  ofn.hInstance := hAppInst;
  ofn.lpstrFilter := PAnsiChar(strFilter);
  ofn.nFilterIndex := nFilterIndex;
  ofn.lpstrFile := PAnsiChar(strInitFileName);
  ofn.nMaxFile := MAX_PATH;
  ofn.lpstrTitle := pAnsiChar(strTitle);
  ofn.Flags := OFN_FILEMUSTEXIST or OFN_PATHMUSTEXIST or
              OFN_HIDEREADONLY;

  //Отображение окна диалога и обработка результата
  if (GetOpenFileName(ofn) = True) then
```

```
ShowOpen := ofn.lpstrFile;
end;
```

Приведенная в листинге 2.23 функция возвращает не пустую строку — полный путь файла в случае, если пользователь выбрал или ввел имя файла. Здесь главной трудностью является заполнение довольно большой структуры `OPENFILENAME`. В данном примере используются только базовые возможности диалога открытия файла и лишь некоторые из поддерживаемых им флагов (поле `Flags`):

- `OFN_FILEMUSTEXIST` — при успешном завершении работы диалогового окна можно быть уверенным, что результирующий путь является путем существующего файла;
- `OFN_PATHMUSTEXIST` — не дает ввести имя файла в несуществующей папке (например, при вводе `c:\docs\mydoc1.doc`, если папки `docs` не существует, будет выдано соответствующее сообщение);
- `OFN_HIDEREADONLY` — не показывать флажок Только для чтения.

Отдельно рассмотрим, зачем в приведенном примере вызывается дополнительная функция `PrepareFilterString` (листинг 2.24).

Листинг 2.24. Преобразование строки фильтра

```
procedure PrepareFilterString(var strFilter: string);
var
  i: Integer;
begin
  for i := 1 to length(strFilter) do
    if (strFilter[i] = '|') then strFilter[i] := #0;
  end;
```

Дело в том, что при задании фильтров (поле `lpstrFile`) требуется, чтобы каждое их название и обозначение были отделены символом `#0`, а за последним фильтром шла последовательность из двух нулевых символов. На практике задавать строку из нескольких фильтров в следующем виде не особо удобно:

```
'Текстовые файлы' + #0 + '*.txt' + #0 + 'Все файлы' + '*.*' + #0 + #0
```

Поэтому часто применяются другие разделители, которые впоследствии преобразуются в символы `#0`. В нашем случае в качестве разделителя используется символ `|`, поэтому приведенная выше строка фильтра может быть записана так:

```
'Текстовые файлы|*.txt|Все файлы|*.*||'
```

Согласитесь, что получилось более кратко и понятно.

Теперь обратимся к диалоговому окну сохранения файла. Для его вызова достаточно переделать пример из листинга 2.23 следующим образом (листинг 2.25).

Листинг 2.25. Окно сохранения файла

```
function ShowSave(strFilter: string; nFilterIndex: Integer = 0;
                 strInitFileName: string = '';
                 strTitle: string = 'Сохранение файла'):string;
var
    ofn: OPENFILENAME;
begin
    ZeroMemory(Addr(ofn), SizeOf(ofn));
    //Формирование буфера (260 символов)
    SetLength(strInitFileName, MAX_PATH);
    PrepareFilterString(strFilter);

    //Заполнение структуры для диалога
    ofn.lStructSize := SizeOf(ofn);
    ofn.hWndOwner := hParentWnd;
    ofn.hInstance := hAppInst;
    ofn.lpstrFilter := PAnsiChar(strFilter);
    ofn.nFilterIndex := nFilterIndex;
    ofn.lpstrFile := PAnsiChar(strInitFileName);
    ofn.nMaxFile := MAX_PATH;
    ofn.lpstrTitle := pAnsiChar(strTitle);
    ofn.Flags := OFN_PATHMUSTEXIST or OFN_OVERWRITEPROMPT;

    //Отображение окна диалога и обработка результата
    if (GetSaveFileName(ofn) = True) then
        ShowSave := ofn.lpstrFile;
end;
```

Здесь дополнительно к упомянутому ранее флагу `OFN_PATHMUSTEXIST` применен флаг `OFN_OVERWRITEPROMPT` для того, чтобы при указании имени уже существующего файла был задан вопрос о желании пользователя заменить старый файл.

Окно выбора цвета

Вызов следующего диалогового окна — окна выбора цвета — приводится в листинге 2.26.

Листинг 2.26. Окно выбора цвета

```
function ShowChooseColor(lastColor: COLORREF = 0):COLORREF;
```

```
var
    choose: TChooseColor;
begin
    ZeroMemory(Addr(choose), SizeOf(choose));
    //Заполнение структуры для диалогового окна
    choose.lStructSize := SizeOf(choose);
    choose.hWndOwner := hParentWnd;
    choose.hInstance := hAppInst;
    choose.rgbResult := lastColor;
    choose.lpCustColors := Addr(colors);
    choose.Flags := CC_RGBINIT or CC_ANYCOLOR or CC_FULLOPEN;
    //Отображение окна диалога и обработка результата
    if (ChooseColor(choose) = True) then ShowChooseColor :=
                                                choose.rgbResult
    else ShowChooseColor := lastColor;
end;
```

Здесь также заполняется структура похожего назначения. Используются следующие флаги диалогового окна:

- **CC_RGBINIT** — использовать значение поля `rgbResult` в качестве предустановленного значения цвета (по умолчанию как ранее выбранного);
- **CC_ANYCOLOR** — отображать все доступные предопределенные цвета (левая часть, рис. 2.4);
- **CC_FULLOPEN** — раскрывать панель подбора цвета (правая часть, рис. 2.4).

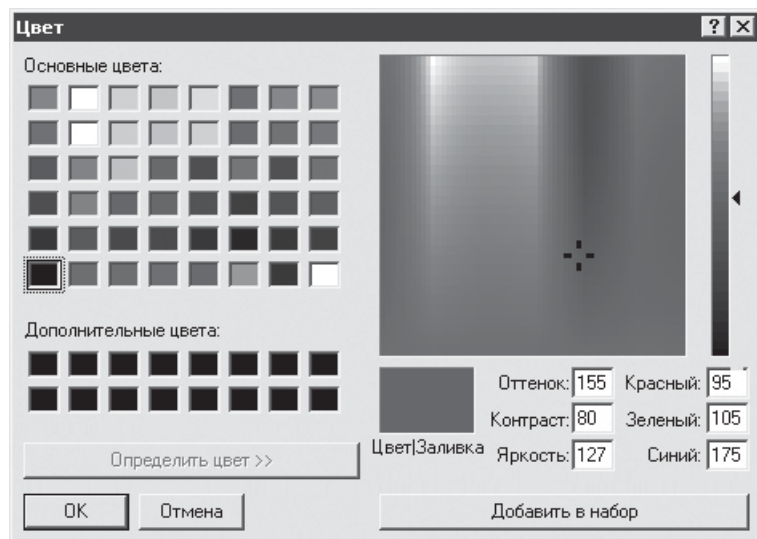


Рис. 2.4. Окно выбора цвета

Поясним, что за переменная, а точнее, ее адрес, сохраняется в поле `lpCustColors` — это массив из 16 значений типа `COLORREF`:

```
colors: array [1..16] of COLORREF;
```

Обратите внимание на 16 квадратов в левой нижней области окна (рис. 2.4) — это места для определенных пользователем цветов. Для заполнения этой области окна и используются значения из массива `colors`. Массив может быть как локальным, так и глобальным (что удобнее, так как значения определенных пользователем цветов сохраняются между вызовами диалогового окна).

Окно выбора шрифта

Для вывода диалогового окна выбора шрифта вполне подойдет функция, приведенная в листинге 2.27.

Листинг 2.27. Окно выбора шрифта

```
function ShowChooseFont(var font: LOGFONT):BOOL;
var
    choose: TChooseFont;
begin
    ZeroMemory(Addr(choose), SizeOf(choose));
    //Заполнение структуры для диалогового окна
    choose.lStructSize := SizeOf(choose);
    choose.hWndOwner := hParentWnd;
    choose.hInstance := hAppInst;
    choose.lpLogFont := Addr(font);
    choose.Flags := CF_BOTH or CF_INITTOLOGFONTSTRUCT;
    //Отображение окна диалога и обработка результата
    if (ChooseFont(choose) = True) then
        begin
            CopyMemory(Addr(font), choose.lpLogFont, SizeOf(font));
            ShowChooseFont := True;
        end
    else ShowChooseFont := False;
end;
```

Здесь используются флаги окна, имеющие следующие значения:

- `CF_BOTH` — отображать экранные и принтерные шрифты (для показа экранных или принтерных шрифтов можно использовать флаги `CF_SCREENFONTS` и `CF_PRINTERFONTS` соответственно);
- `CF_INITTOLOGFONTSTRUCT` — выбрать в диалоговом окне шрифт, соответствующий (или максимально похожий) шрифту, описываемому структурой `LOGFONT`, указатель на которую сохраняется в поле `lpLogFont`.

Окно выбора папки

Чтобы иметь возможность пользоваться окном Обзор папок для выбора папки, можно использовать листинг 2.28.

Листинг 2.28. Окно выбора папки

```
function ShowChooseFolder(strTitle: string):string;
var
    choose: BROWSEINFO;
    buffer: string;
    pidl: PItemIDList;
begin
    ZeroMemory(Addr(choose), SizeOf(choose));
    SetLength(buffer, MAX_PATH);
    //Заполнение структуры для диалога
    choose.hwndOwner := hParentWnd;
    choose.pidlRoot := nil;           //Корень – папка Рабочего стола
    choose.pszDisplayName := PAnsiChar(buffer);
    choose.lpszTitle := PAnsiChar(strTitle);
    choose.ulFlags := 0;

    //Вывод диалогового окна и обработка результата
    pidl := SHBrowseForFolder(choose);
    if (pidl <> nil) then
    begin
        //Получение полного пути выбранной папки
        SHGetPathFromIDList(pidl, PAnsiChar(buffer));
        ShowChooseFolder := buffer;
        DeletePIDL(pidl);
    end
    else
        ShowChooseFolder := '';
end;
```

В листинге 2.28 функция ShowChooseFolder возвращает полный путь указанной папки, если она выбрана, и пустую строку в противном случае. Само окно Обзор папок показано на рис. 2.5.

Особенностью использованной в данном примере функции SHBrowseForFolder является то, что она возвращает не путь выбранного каталога, а указатель на структуру ItemIDList (что-то вроде внутреннего представления путей). Для извлече-

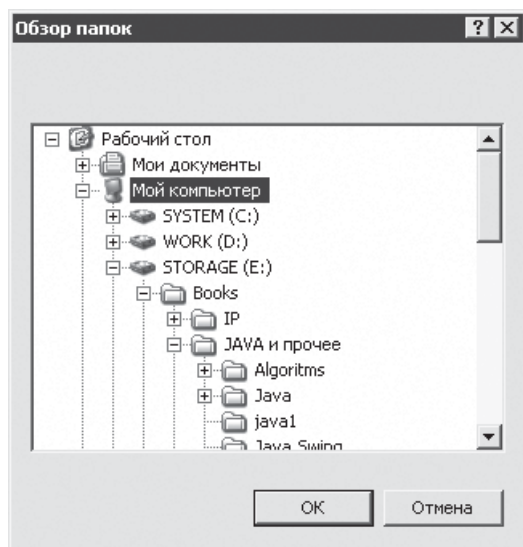


Рис. 2.5. Окно выбора папки

ния построения пути по содержимому этой структуры используется функция `SHGetPathFromIDList`. После этого структура нам больше не нужна, и ее следует правильно удалить (с использованием специального интерфейса `IMalloc`). Для этого используется процедура `DeletePIDL`, реализованная в листинге 2.29.

Листинг 2.29. Удаление структуры `ItemIDList`

```
procedure DeletePIDL(pidl: PItemIDList);
var
    pMalloc: IMalloc;
begin
    SHGetMalloc(pMalloc);
    if (pMalloc <> nil) then
    begin
        pMalloc.Free(pidl);
        pMalloc._Release();
    end;
end;
```



ПРИМЕЧАНИЕ

Освобождение памяти, занимаемой данными структуры `ItemIDList`, можно выполнить и более простым способом: использовать API-функцию `CoTaskMemFree`, передав ей адрес структуры в качестве следующего параметра: `CoTaskMemFree(pidl)`.

Вообще функцию `SHBrowseForFolder` (листинг 2.28) можно использовать для указания принтеров или компьютеров. Для этого достаточно установить флаги `BIF_BROWSEFORCOMPUTER` и `BIF_BROWSEFORPRINTER` соответственно:

```
choose.ulFlags := BIF_BROWSEFORCOMPUTER;
```

или

```
choose.ulFlags := BIF_BROWSEFORPRINTER;
```

Чтобы в окне отображались еще и значки файлов, необходимо установить флаг `BIF_BROWSEINCLUDEFILES`.

Окно подключения и отключения сетевого ресурса

Часто бывает удобно осуществлять доступ к сетевым папкам как к локальным дискам компьютера (с использованием того же принципа построения пути). Окна подключения и отключения сетевого ресурса позволяют дать пользователю возможность выбрать, какие папки считать сетевыми дисками и какие сетевые диски можно отключить.

Окно подключения сетевого ресурса в Windows XP выглядит так, как показано на рис. 2.6.

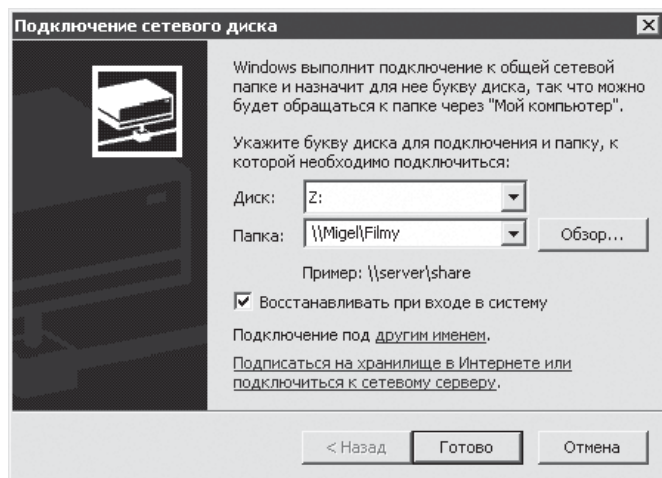


Рис. 2.6. Окно подключения сетевого диска

Для вызова диалогового окна подключения сетевого ресурса можно использовать функцию, приведенную в листинге 2.30.

Листинг 2.30. Окно подключения сетевого ресурса

```
function ShowConnection(): BOOL;  
begin
```

```
ShowConnection :=  
  WNetConnectionDialog(hParentWnd, RESOURCETYPE_DISK) = NO_ERROR;  
end;
```

Функция `ShowConnection` возвращает `True` в случае удачного подключения и `False` в противном случае.

Окно отключения сетевого диска приведено на рис. 2.7.

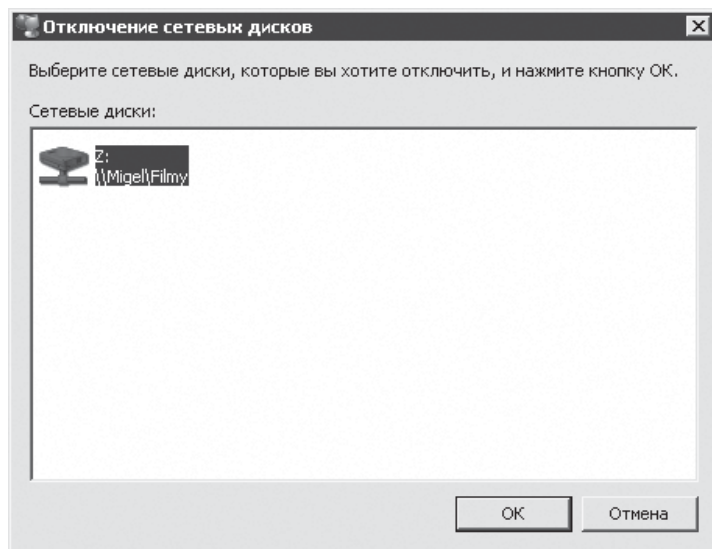


Рис. 2.7. Отключение сетевого ресурса

Функция, показывающая окно отключения сетевого диска, приведена в листинге 2.31.

Листинг 2.31. Окно отключения сетевого ресурса

```
function ShowDisconnect(): BOOL;  
begin  
  ShowDisconnect :=  
    WNetDisconnectDialog(hParentWnd, RESOURCETYPE_DISK) = NO_ERROR;  
end;
```

Аналогично `ShowConnection` функция `ShowDisconnect` возвращает `True`, если отсоединен хотя бы один диск, и `False` в противном случае.

Системное окно «О программе»

Этот последний и довольно экзотичный пример приведен на случай, если возникнет желание или необходимость использовать красивое окно `О программе`, которое

выводится для самой операционной системы Windows и ее компонентов. Процедура вывода этого окна приведена в листинге 2.32.

Листинг 2.32. Окно «О программе»

```
procedure ShowAbout(strAppName: string; strInfo: string);
begin
    ShellAbout(hParentWnd, PAnsiChar(strAppName),
               PAnsiChar(strInfo), LoadIcon(0, IDI_ASTERISK));
end;
```

Правда, в окне 0 программе Windows XP на информацию о приложении отведено всего две строки (и место для значка слева от окна). Все остальное место занимают информация о регистрации операционной системы и фирменная эмблема Microsoft Windows XP.

Демонстрационное приложение

Теперь пришла очередь рассмотреть небольшое приложение, использующее описанные выше диалоговые окна (проект StandartWindows). Окно этого приложения приводится на рис. 2.8.

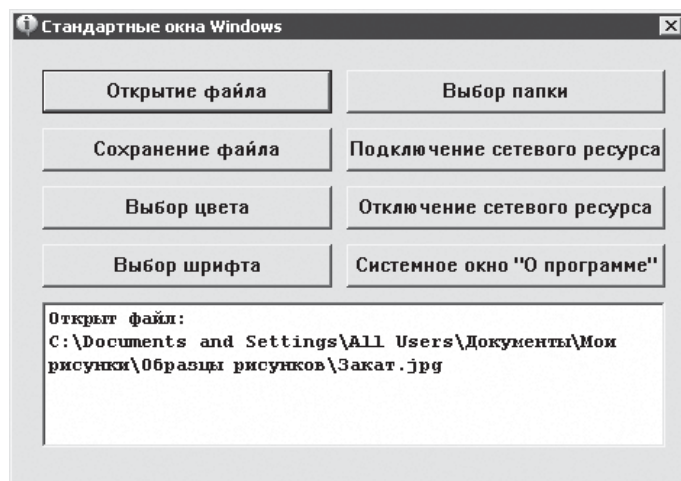


Рис. 2.8. Окно демонстрационного приложения

Размер EXE-файла приложения равен 22 Кбайт.

В листинге 2.33 приводятся объявления используемых глобальных переменных, а также код, реализующий создание окна и элементов управления в нем, цикл обработки сообщений (файл StandartWindows.dpr). Функции работы с рассмотренными выше диалоговыми окнами вынесены в отдельный модуль StdWindows (файл StdWindows.pas). В этом и следующем листинге используются уже знакомые вам функции из модуля Controls.

Листинг 2.33. Глобальные переменные, код создания окна и цикл обработки сообщений

```
program StandartWindows;
{$R *.res}
uses
  Windows, Messages, CommDlg,
  Controls in 'Controls.pas',
  StdWindows in 'StdWindows.pas';
var
  hMainWnd: HWND;
  hInst: Cardinal;
  mess: MSG;
  curColor: COLORREF;
  font: LOGFONT;
  hCurFont: HFONT;
...
function RegisterWindow():Boolean;
...
begin
  hInst := GetModuleHandle(nil);
  //Регистрация и создание главного окна
  if not RegisterWindow() then Exit;
  hMainWnd := CreateWindow(
    'MyWindowClass', //Имя класса окна
    'Стандартные окна Windows', //Заголовок окна
    WS_CAPTION or WS_SYSMENU or WS_CLIPCHILDREN or WS_CLIPSIBLINGS,
    CW_USEDEFAULT, //Координата X по умолчанию
    CW_USEDEFAULT, //Координата Y по умолчанию
    470, 420,
    HWND(nil), //Нет родительского окна
    HMENU(nil), //Нет меню
    hInst,
    nil);
  if (hMainWnd = HWND(nil)) then Exit;

  //Инициализация модуля Controls для работы с главным
  //окном приложения
  Controls.hParentWnd := hMainWnd;
  Controls.hAppInst := hInst;
```

```
//Инициализация модуля StdWindows для работы с главным
//окном приложения
StdWindows.hParentWnd := hMainWnd;
StdWindows.hAppInst := hInst;

//Создание кнопок для открытия диалоговых окон
CreateButton(20, 20, 200, 30, 1001, 'Открытие файла');
CreateButton(20, 60, 200, 30, 1002, 'Сохранение файла');
CreateButton(20, 100, 200, 30, 1003, 'Выбор цвета');
CreateButton(20, 140, 200, 30, 1004, 'Выбор шрифта');
CreateButton(20, 180, 200, 30, 1005, 'Окно поиска текста');
CreateButton(20, 220, 200, 30, 1006, 'Окно поиска и замены');
CreateButton(230, 20, 220, 30, 1010, 'Выбор папки');
CreateButton(230, 60, 220, 30, 1011, 'Подключение сетевого
ресурса');
CreateButton(230, 100, 220, 30, 1012, 'Отключение сетевого
ресурса');
CreateButton(230, 140, 220, 30, 1013, 'Системное окно "О про-
грамме"');
//Текстовое поле для результата
CreateMemo(20, 270, 430, 100, 2001);

ShowWindow(hMainWnd, SW_NORMAL);

//Запуск цикла обработки сообщений
while (Longint(GetMessage(mess, 0, 0, 0)) <> 0) do
begin
    if (IsDialogMessage(hMainWnd, mess) = False) then
    begin
        TranslateMessage(mess);
        DispatchMessage(mess);
    end;
end;
end.
```

Код функции RegisterWindow опущен, так как он аналогичен приведенному в листинге 2.4. Функции работы с рассмотренными ранее диалоговыми окнами вынесены в модуль StdWindows (файл StdWindows.pas).

Особенностью цикла обработки сообщений в этом примере является использование API-функции IsDialogMessage, которая позволяет реагировать на некото-

рые действия пользователя так, как это делается в диалоговых окнах. Примером может быть перемещение фокуса между окнами при нажатии клавиши `Tab`.

Перед функцией `RegisterWindow` (на месте многоточия перед ее объявлением в листинге 2.33) находится функция обработки сообщений, имеющая следующий вид (листинг 2.34).

Листинг 2.34. Функция обработки сообщений

```
function WindowFunc(hWnd:HWND; msg:UINT; wParam:WPARAM;
                    lParam:LPARAM):LRESULT; stdcall;

var
    hOldFont: HFONT;
    strBuf: String;
    hEditDC: HDC;
begin
    case msg of
        WM_CLOSE:
            if (hWnd = hMainWnd) then PostQuitMessage(0);
        WM_CTLCOLOREDIT: //Сообщения от Edit перед перерисовкой
            begin
                //Зададим тексту Edit выбранный цвет
                hEditDC := HDC(wParam);
                SetTextColor(hEditDC, curColor);
                GetCurrentObject(hEditDC, OBJ_BRUSH);
            end;
        WM_COMMAND:
            if (HIWORD(wParam) = BN_CLICKED) then
                begin
                    //Определим, какая кнопка нажата
                    case LOWORD(wParam) of
                        1001: //Открытие файла
                            begin
                                SetText(2001, 'Открыт файл:' + #13 + #10 +
                                    ShowOpen('Все файлы|*.*|'));
                            end;
                        1002: //Сохранение файла
                            begin
                                SetText(2001, 'Путь для сохранения:' + #13 + #10 +
                                    ShowSave('Все файлы|*.*|'));
                            end;
                    end;
                end;
            end;
```

```
1003: //Выбор цвета
begin
    curColor := ShowChooseColor(curColor);
    Str(curColor, strBuf);
    SetText(2001, 'Выбранный цвет:' + #13 + #10 + strBuf);
end;
1004: //Выбор шрифта
begin
    if (ShowChooseFont(font) = True) then
    begin
        //Замена шрифта в Edit
        hOldFont := HFONT(
            SendDlgItemMessage(hMainWnd, 2001, WM_GETFONT, 0, 0));
        hCurFont := CreateFontIndirect(font);
        SendDlgItemMessage(hMainWnd, 2001, WM_SETFONT,
            Integer(hCurFont),
            Integer(True));
        SetText(2001, 'Текст, записанный выбранным
                                шрифтом');

        if (hOldFont <> 0) then DeleteObject(hOldFont);
    end;
end;
1010: //Выбор папки
begin
    SetText(2001, 'Выбранная папка:' + #13 + #10 +
        ShowChooseFolder());
end;
1011: //Подключение сетевого ресурса
begin
    ShowConnection();
end;
1012: //Отключение сетевого ресурса
begin
    ShowDisconnect();
end;
1013: //Окно "О программе"
begin
    ShowAbout('Standart windows',
        'Демонстрация использования стандартных ' +
        'окон диалога из чистого API-приложения');
```

```
        end;
    end;
end;
else
    begin
        //Обработка по умолчанию
        WindowFunc := DefWindowProc(hWnd, msg, wParam, lParam);
        Exit;
    end;
end;

WindowFunc := S_OK; //Сообщение обработано
end;
```

Обработка сообщений здесь довольно проста, за исключением изменения шрифта текстового поля. Обратите внимание на следующий отрывок листинга 2.34:

```
//Замена шрифта в Edit
hOldFont := HFONT(SendDlgItemMessage(hMainWnd, 2001, WM_GETFONT, 0, 0));
hCurFont := CreateFontIndirect(font);
SendDlgItemMessage(hMainWnd, 2001, WM_SETFONT,
    Integer(hCurFont), Integer(True));
SetEditText(2001, 'Текст, записанный выбранным шрифтом');
if (hOldFont <> 0) then DeleteObject(hOldFont);
```

Этот довольно объемный фрагмент кода всего лишь заменяет шрифт в текстовом поле. Подобную операцию можно использовать для задания шрифта любого элемента управления. В частности, в приведенных в этой главе примерах текст на кнопках, надписях и т. д. выглядит довольно невзрачно потому, что используется системный шрифт, установленный по умолчанию.

Способ, которым можно установить шрифт всех элементов управления окна, показан далее. Теперь еще один существенный момент: не забывайте удалять объекты GDI (в данном случае — шрифт) после того, как они стали не нужны. Дело в том, что приложение может владеть не более чем 65 000 объектов GDI. И при наличии так называемой «утечки» ресурсов GDI может наступить момент (при продолжительной работе программы), когда вдруг окна приложения начинают отрисовываться по меньшей мере странно (если вообще отрисовываются).

2.5. Установка шрифта элементов управления

Есть множество способов установки шрифта текста, отображаемого в элементах управления. Можно, например, при создании каждого элемента управления посылать

ему сообщение WM_SETFONT, передавая дескриптор (HFONT) созданного ранее объекта шрифта. В таком случае код создания и установки шрифта элементов управления (с использованием рассмотренных в этой главе функций) может выглядеть, как в листинге 2.35.

Листинг 2.35. Установка шрифта по ходу создания элементов управления

```
//Шрифт для элементов управления
font := CreateFont(16, 0, 0, 0, FW_NORMAL, 0, 0, 0, ANSI_CHARSET,
                  OUT_CHARACTER_PRECIS, CLIP_DEFAULT_PRECIS,
                  DEFAULT_QUALITY, DEFAULT_PITCH, 'Courier new');

//Создание элементов управления
ctrl := CreateButton(20, 30, 70, 30, 1001, 'Кнопка 1');
SendMessage(ctrl, WM_SETFONT, HFONT(font), 1);

ctrl := CreateButton(100, 30, 70, 30, 1002, 'Кнопка 2');
SendMessage(ctrl, WM_SETFONT, HFONT(font), 1);

ctrl := CreateCheck(210, 30, 180, 20, 2001, 'Флажок 1');
SendMessage(ctrl, WM_SETFONT, HFONT(font), 1);

ctrl := CreateCheck(210, 60, 180, 20, 2001, 'Флажок 2', True);
SendMessage(ctrl, WM_SETFONT, HFONT(font), 1);

ctrl := CreateOption(210, 100, 180, 20, 3001, 'Переключатель 1',
True);
SendMessage(ctrl, WM_SETFONT, HFONT(font), 1);

ctrl := CreateOption(210, 130, 180, 20, 3002, 'Переключатель 2',
False, True);
SendMessage(ctrl, WM_SETFONT, HFONT(font), 1);

ctrl := CreateOption(210, 160, 180, 20, 3003, 'Переключатель 3',
True);
SendMessage(ctrl, WM_SETFONT, HFONT(font), 1);

//Запуск цикла обработки сообщений
while (Longint(GetMessage(mess, 0, 0, 0)) <> 0)
do begin
    TranslateMessage(mess);
```

```
DispatchMessage (mess) ;  
end;  
//Удаление шрифта  
DeleteObject (font);
```

Выглядит окно с элементами управления, шрифт которых установлен любым из рассмотренных способов, так, как показано на рис. 2.9.

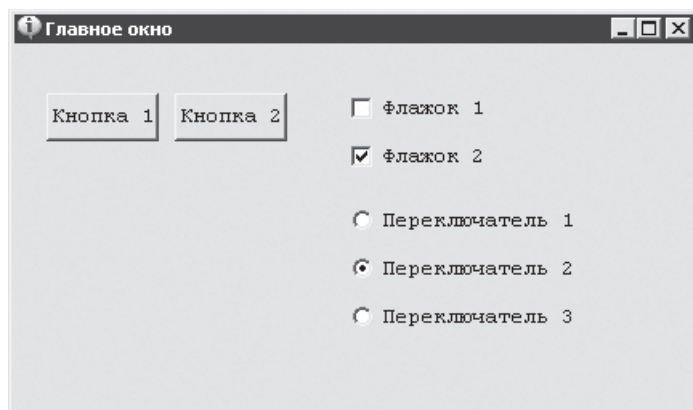


Рис. 2.9. Шрифт элементов управления, от личный от системного

Способ задания шрифта, приведенный в листинге 2.35, легко реализовать. Его существенным недостатком является двукратное увеличение количества строк кода, выполняющих создание элементов управления. Для окон, содержащих большое количество элементов управления, можно предложить более универсальный способ (листинг 2.36).

Листинг 2.36. Установка шрифта перебором элементов управления

```
//Шрифт для элементов управления  
font := CreateFont(16, 0, 0, 0, FW_NORMAL, 0, 0, 0, ANSI_CHARSET,  
OUT_CHARACTER_PRECIS, CLIP_DEFAULT_PRECIS,  
DEFAULT_QUALITY, DEFAULT_PITCH, 'Courier new');  
  
//Создание элементов управления  
CreateButton(20, 30, 70, 30, 1001, 'Кнопка 1');  
CreateButton(100, 30, 70, 30, 1002, 'Кнопка 2');  
CreateCheck(210, 30, 180, 20, 2001, 'Флажок 1');  
CreateCheck(210, 60, 180, 20, 2001, 'Флажок 2', True);  
CreateOption(210, 100, 180, 20, 3001, 'Переключатель 1', True);  
CreateOption(210, 130, 180, 20, 3002, 'Переключатель 2', False,  
True);  
CreateOption(210, 160, 180, 20, 3003, 'Переключатель 3', True);
```

```
//Установка шрифта элементов управления
EnumChildWindows(hMainWnd, Addr(EnumFunc), font);

//Запуск цикла обработки сообщений
while (Longint(GetMessage(mess, 0, 0, 0)) <> 0)
do begin
    TranslateMessage(mess);
    DispatchMessage(mess);
end;
DeleteObject(font);
```

Собственно за установление шрифта отвечает в приведенном листинге только одна строка:

```
EnumChildWindows(hMainWnd, Addr(EnumFunc), font);
```

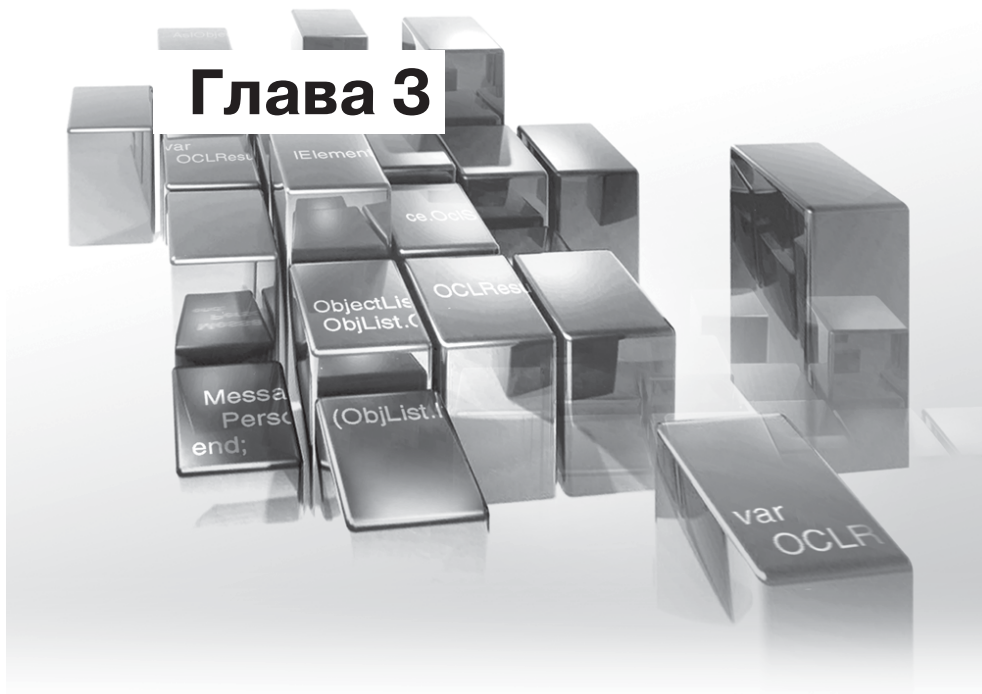
Правда, при этом нужно определить функцию обратного вызова (в данном случае это функция EnumFunc), которая будет вызываться по одному разу для каждого дочернего окна. В нашем примере функция EnumFunc имеет следующий вид (листинг 2.37).

Листинг 2.37. Реализация функции EnumFunc

```
function EnumFunc(wnd: HWND; param: LPARAM):BOOL; stdcall;
begin
    SendMessage(wnd, WM_SETFONT, WPARAM(param), LPARAM(True));
    EnumFunc := True; //Продолжать перечисление
end;
```

В принципе, имя этой функции и названия параметров могут быть любыми. А вот типы параметров и возвращаемого значения, а также способ вызова функции должны быть именно такими, как в листинге 2.37. Функция должна возвращать True, если нужно продолжать перечисление окон, и False в противном случае. Значение, которое было передано в качестве третьего параметра API-функции EnumChildWindows, передается в функцию обратного вызова. В нашем случае этим параметром является дескриптор шрифта.

Глава 3



Мышь и клавиатура

- ☐ Мышь
- ☐ Клавиатура

Самыми распространенными средствами для ввода информации в компьютер являются мышь и клавиатура. Уже сложно представить себе персональный компьютер без таких устройств, так как клавиатура обеспечивает полноценный ввод текстовой информации, а мышь — это наиболее простое, интуитивно понятное средство для работы с графическим интерфейсом. В этой связи существует масса возможностей по созданию различного рода хитростей и трюков, связанных с мышью и клавиатурой.

3.1. Мышь

Начнем с простых операций с мышью. Вероятно, простота этого средства определяет то, как легко использовать в программе данные, получаемые от мыши. Поэтому при работе с мышью большинство сложностей состоит именно в особых алгоритмах обработки данных, а не в получении этих данных (по сравнению, например, с клавиатурой), в чем вы сами сейчас сможете убедиться.

Координаты и указатель мыши

Для начала программным путем определим присутствие мыши в системе. Один из способов определения наличия мыши демонстрирует следующий пример (листинг 3.1).

Листинг 3.1. Как узнать, присутствует ли мышь

```
function MousePresent : Boolean;
begin
    //При помощи вызова GetSystemMetrics определяем
    //наличие мыши в системе
    if GetSystemMetrics(SM_MOUSEPRESENT) <> 0 then
        Result := True
    else
        Result := False;
end;
```

Описанная выше функция `MousePresent` позволяет проверить наличие мыши. Когда мышь присутствует, `MousePresent` возвращает `True`, в противном случае — `False`.

После того как мы обнаружили мышь, можем приступить к определению ее координат на экране (листинг 3.2).

Листинг 3.2. Определение координат указателя мыши

```
procedure MouseForm.Button1Click(Sender: TObject);
var
    pt: TPoint;
begin
```

```
//Получаем координаты указателя мыши
GetCursorPos(pt);
ShowMessage( '(' + IntToStr(pt.X) + ' , ' + IntToStr( pt.Y ) + ' ) ' );
end;
```

Для определения координат мыши использовалась API-функция `GetCursorPos`. Передав в эту функцию переменную `pt` типа `TPoint`, мы получим текущие экранные координаты указателя.

Рассмотрим пример, в котором указатель мыши при нажатии кнопки `Button2` скрывается, а при нажатии кнопки `Button3` (например, при помощи клавиатуры) показывается (листинг 3.3).

Листинг 3.3. Скрытие указателя мыши

```
procedure MouseForm.Button2Click(Sender: TObject);
begin
    //Прячем указатель
    ShowCursor(False);
end;

procedure MouseForm.Button3Click(Sender: TObject);
begin
    //Показываем указатель
    ShowCursor(True);
end;
```

В приведенном примере для управления видимостью указателя мыши используется функция `ShowCursor`, которая либо скрывает его (принимая значение `False`), либо снова показывает (принимая значение `True`). По причине того что указатель может скрываться и управление мышью будет невозможно, исходный текст, осуществляющий управление видимостью указателя, помещен в обработчики нажатия кнопок формы. В то время, когда указатель будет скрыт, можно использовать клавишу `Tab` для выбора и нажатия кнопки.

Существуют и другие способы скрыть указатель. Рассмотрим пример управления его видимостью посредством установки свойства `Cursor` компонента:

```
TempForm.Cursor := crNone;
```

В данном случае указатель делается невидимым только для формы, за ее пределы он становится видимым. Если на форме присутствуют компоненты (элементы управления), то при наведении на них указатель мыши становится видимым. Если мы хотим сделать его невидимым во всей области экрана, то следует применить следующий исходный текст:

```
Screen.Cursor := crNone;
```

Мышь можно передвигать и программным путем. Следующий пример демонстрирует, каким образом это можно сделать (листинг 3.4).

Листинг 3.4. Изменение координат мыши

```
procedure TForm1.Button1Click(Sender: TObject);
var
    pt : TPoint;
begin
    Application.ProcessMessages;
    Screen.Cursor := CrHourglass;
    GetCursorPos(pt);
    SetCursorPos(pt.x + 1, pt.y + 1);
    Application.ProcessMessages;
    SetCursorPos(pt.x - 1, pt.y - 1);
end;
```

Захват указателя мыши

Существует ряд задач, для выполнения которых бывает полезно иметь возможность получать сообщения от мыши даже тогда, когда указатель находится за пределами формы. За примером далеко ходить не надо: откройте редактор Paint, сделайте размер его окна меньше размера холста, после чего, нажав кнопку мыши, нарисуйте линию так, чтобы в ходе рисования указатель вышел за пределы окна редактора. Есть ли на рисунке часть линии, которую вы рисовали, двигая указатель за пределами окна (должна быть)?

Захват указателя полезен и в других случаях, потому мы рассмотрим, как его реализовать (а сделать это действительно просто). В листинге 3.5 приводятся обработчики нажатия и отпускания кнопки мыши, которые реализуют захват указателя на время от нажатия до отпускания кнопки.

Листинг 3.5. Захват и освобождение указателя мыши

```
procedure TForm1.FormMouseDown(Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    //Захватываем указатель мыши
    SetCapture(Handle);
end;

procedure TForm1.FormMouseUp(Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
```

```
//Отменяем захват указателя  
ReleaseCapture();  
end;
```

Вся хитрость состоит в использовании API-функций захвата `SetCapture`, а также `ReleaseCapture`. При вызове первой функции происходит регистрация окна, которое захватывает указатель мыши: окно будет получать сообщения от мыши даже тогда, когда указатель будет находиться за его пределами. Функция возвращает дескриптор окна, которое захватило указатель ранее, либо `0`, если такого окна нет. Соответственно, функция `ReleaseCapture` используется для отмены захвата указателя.



ПРИМЕЧАНИЕ

При использовании `SetCapture` окно будет получать сообщения, когда указатель находится не над окном только в случае, если кнопка мыши нажата либо если указатель находится над одним из окон, созданных тем же потоком (независимо от нажатия кнопки мыши).

Можно также упомянуть о API-функции `GetCapture`. Функция не принимает аргументов и возвращает дескриптор окна, захватившего указатель ранее. С помощью этой функции можно, например, удостовериться, что захватом указателя мыши мы не нарушим работу другого приложения (что маловероятно).

Ограничение перемещения указателя

При помощи несложных манипуляций можно также ограничить перемещение указателя мыши определенной областью экрана (прямоугольником). Для этого используется API-функция `ClipCursor`. Она принимает в качестве параметра структуру `TRect` с координатами прямоугольника, в пределах которого может перемещаться указатель, и в случае успешной установки ограничения возвращает отличное от нуля значение.

С `ClipCursor` тесно связана функция `GetClipCursor`, позволяющая получить координаты прямоугольника, которым в данный момент ограничено перемещение указателя.

Использование функций `ClipCursor` и `GetClipCursor` приведено в листинге 3.6.

Листинг 3.6. Ограничение перемещения указателя

```
var  
    lastRect: TRect;  
    cursorClipped: Boolean = False;  
procedure SetCursorRect(newRect: TRect);  
begin
```

```
if not cursorClipped then
begin
    //Сохраняем старую область перемещения указателя
    GetClipCursor(lastRect);
    //Устанавливаем ограничение на перемещения указателя
    cursorClipped := ClipCursor(Addr(newRect)) <> False;
end;
end;
procedure RestoreCursorRect();
begin
    if cursorClipped then
    begin
        //Восстанавливаем область перемещения указателя
        cursorClipped := ClipCursor(Addr(lastRect)) = False;
    end;
end;
```

Здесь реализована пара процедур, первая из которых (`SetCursorRect`) ограничивает перемещение указателя мыши заданной областью экрана (параметр `newRect`). Перед ограничением на перемещение указателя в процедуре `SetCursorRect` происходит сохранение области перемещения, установленной ранее, чтобы действие процедуры можно было отменить. Для отмены ограничения перемещения указателя служит вторая процедура — `RestoreCursorRect`.



ПРИМЕЧАНИЕ

Вообще, задание ограничения на перемещение указателя мыши не считается хорошим тоном. Потому для использования такой возможности в реальном приложении должны быть действительно веские причины.

Изменение назначения кнопок мыши

Как известно, операционная система Windows дает возможность работать за компьютером широкому кругу людей. Со стороны разработчиков было бы глупо не предусмотреть возможность простой адаптации манипулятора «мышь» к правше или левше. К тому же мышь адаптировать к таким различиям намного проще: конструкцию изменять не надо, достаточно программно поменять функции кнопок мыши.

Как поменять функции левой и правой кнопок мыши, демонстрирует листинг 3.7.

Листинг 3.7. Изменение назначения кнопок мыши

```
procedure TForm1.Button1Click(Sender: TObject);
begin
```

```
//Меняем местами функции левой и правой кнопок мыши
SwapMouseButton(True);
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
    //Восстанавливаем функции кнопок мыши
    SwapMouseButton(False);
end;
```

В листинге 3.7 не учитывается тот факт, что инверсия мыши уже может быть установлена при запуске программы (например, если за компьютером работает левша). Чтобы точно знать, была ли ранее применена инверсия к кнопкам мыши, можно использовать значение, возвращаемое функцией `SwapMouseButton`. Если это значение отлично от нуля, то ранее функции кнопок мыши были инвертированы.

Подсчет расстояния, пройденного указателем мыши

Рассмотрим небольшую программу, которая носит скорее познавательный, чем практический характер. Она умеет подсчитывать, сколько же метров (в буквальном смысле) пробегает указатель мыши за время ее работы. Внешний вид формы приложения показан на рис. 3.1.

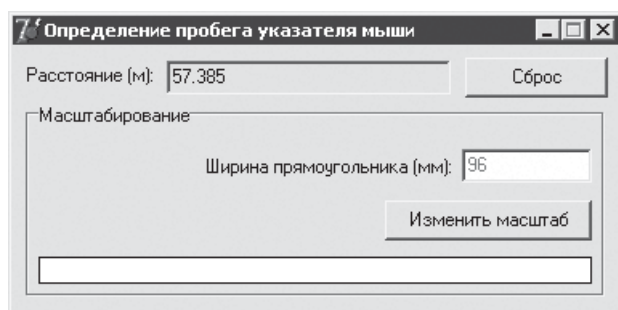


Рис. 3.1. Программа измерения пробега указателя мыши

Использование такой программы крайне просто: сразу после запуска она начинает измерять пройденное указателем мыши расстояние в пикселах. Нижняя группа элементов управления нужна для правильного вывода пройденного расстояния в метрах. При нажатии кнопки **Изменить масштаб** становятся активными два текстовых поля (для ввода ширины и высоты прямоугольника). Чтобы программа правильно преобразовывала пройденное расстояние, нужно линейкой измерить ширину белого прямоугольника и ввести полученное значение (в мм) в текстовое поле. При повторном нажатии этой кнопки введенные значения принимаются, и с этого момента показания пройденного расстояния переводятся в метры с учетом текущего разрешения и размера монитора.

Теперь приступим к рассмотрению реализации этого приложения. В табл. 3.1 приводятся сведения по настройке элементов управления, не являющихся рамками или статическими надписями.

Таблица 3.1. Параметры элементов управления формы, показанной на рис. 3.1

Название (свойство name)	Описание	Измененные свойства и их значения
Timer1	При каждом срабатывании таймера происходит получение положения указателя мыши	Interval = 1
txtDistance	В этом текстовом поле выводится пройденное указателем расстояние	ReadOnly = True Color = clBtnFace Text = Расстояние (м)
cmbClear	При нажатии кнопки сбрасываются показания счетчика пройденного расстояния	Caption = Сброс
Shape1	Размеры этого прямоугольника измеряются при задании масштаба	Ширина и высота произвольны, но чем больше, тем точнее масштабирование
txtWidth	Текстовое поле для ввода ширины прямоугольника Shape1	Text = Ширина прямоугольника (мм)
cmbScale	При нажатии этой кнопки программа переходит в режим установки масштаба	Caption = Изменить масштаб

В листинге 3.8 приводятся объявления переменных (членов класса TForm1) и методов, добавленных вручную.

Листинг 3.8. Форма для измерения пробега указателя

```

type
  TForm1 = class(TForm)
  ...
  private
    isUpdating: Boolean; //Если равен False, то показания
                        //в txtDistance
                        //не обновляются
    lastPos: TPoint;    //Координаты указателя во время
                        //прошлого замера
    distance: Real;     //Пройденное расстояние в пикселах
    procedure StartUpdating();
    procedure StopUpdating();
    procedure ShowDistance();
  end;

```

Суммарное расстояние в пикселах, пройденное указателем, сохраняется в переменной `distance`. Рассмотрим, как осуществляется перевод этого расстояния в метры (листинг 3.9).

Листинг 3.9. Перевод расстояния в метры с учетом масштаба

```
procedure TForm1.ShowDistance();
var
    scale: Real;
    distanceMeters: Real;
begin
    //Пересчитываем текущий пробег в метры и показываем его
    //в текстовом поле
    //..определяем масштаб для перевода измерений в метры
    scale := 0.001 * StrToInt(txtWidth.Text) / Shape1.Width;
    //..подсчитываем расстояние с учетом масштаба
    distanceMeters := scale * distance;
    //..округляем до трех знаков (мм) и показываем
    distanceMeters := Int(distanceMeters * 1000) * 0.001;
    txtDistance.Text := FloatToStr(distanceMeters);
end;
```

В приведенном расчете нет ничего сложного, как , собственно, нет ничего сложного и во всем примере. Главная процедура приложения — обработчик для таймера `Timer1`. Таймер срабатывает с максимальной для него частотой (не 1 мс, конечно, но где-то 18 раз в секунду). Текст обработчика `Timer1Timer` приводится в листинге 3.10.

Листинг 3.10. Подсчет разницы между положениями указателя мыши

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
    curPos: TPoint;
    delta: Real;
begin
    if (curPos.X <> lastPos.X) or (curPos.Y <> lastPos.Y) then
    begin
        GetCursorPos(curPos);
        //Вычисляем разницу между текущим и прошлым
        //положением мыши
        delta := Sqrt(Sqr(curPos.X - lastPos.X) + Sqr(curPos.Y -
lastPos.Y));
        distance := distance + delta;
```

```
//Не забываем сохранить новые координаты указателя
lastPos := curPos;
if isUpdating then
begin
    //Обновим показания в текстовом поле
    ShowDistance();
end;
end;
end;
```

Как можно увидеть при внимательном рассмотрении листинга 3.10, обновление показаний происходит при истинном значении переменной `isUpdating`. Значение этой переменной устанавливается в `False` во время задания масштаба, чтобы во время ввода значений в текстовые поля не выводились неправильные цифры (листинг 3.11).

Листинг 3.11. Активизация/деактивизация режима ввода масштаба

```
procedure TForm1.cmbScaleClick(Sender: TObject);
begin
    if cmbScale.Caption = 'Изменить масштаб' then
    begin
        //Начинаем изменение масштаба
        StopUpdating();
        cmbScale.Caption := 'Принять масштаб';
        txtWidth.Enabled := True;
    end
    else
    begin
        //Заканчиваем изменение масштаба
        txtWidth.Enabled := False;
        cmbScale.Caption := 'Изменить масштаб';
        StartUpdating();
    end;
end;
```

Процедуры `StartUpdating` и `StopUpdating` скрывают действия, которые необходимо произвести для остановки или возобновления отображения пройденного расстояния в текстовом поле. В нашем примере они выглядят крайне просто (листинг 3.12).

Листинг 3.12. Включение/выключение обновления результатов измерения

```
procedure TForm1.StartUpdating();
begin
```

```
//Включаем обновление показаний в текстовом поле
isUpdating := True;
end;
procedure TForm1.StopUpdating();
begin
    //Отключаем обновление показаний в текстовом поле
    isUpdating := False;
end;
```

В завершение остается реализовать код инициализации при запуске программы и обработчик события Click для кнопки cmbClear (листинг 3.13).

Листинг 3.13. Инициализация при запуске и код сброса счетчика

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    //Инициализируем координаты мыши
    GetCursorPos(lastPos);
    StartUpdating();
end;
procedure TForm1.cmbClearClick(Sender: TObject);
begin
    //Сбрасываем счетчик пройденного расстояния
    distance := 0;
    GetCursorPos(lastPos); //Начинаем отсчет с текущей
                           //позиции указателя
    ShowDistance();
end;
```

Вот, собственно, и все, что нужно для работы рассматриваемой программы. Остается лишь уточнить, что способ установки масштаба, используемый в программе, предназначен для таких разрешений мониторов, при которых нет искажений по горизонтали или вертикали. Чаще всего это такие разрешения, при которых размеры изображения по горизонтали и вертикали подчиняются пропорции 4:3 (640 × 480, 800 × 600 и т. д.). При этом такими же пропорциями должен обладать экран монитора.

Подсвечивание элементов управления

В завершение рассмотрим несложный, но достаточно полезный пример, позволяющий сделать более «живым» интерфейс приложения: изменение внешнего вида элементов управления при наведении на них указателя мыши.

В листинге 3.14 показано, как можно сделать статическую надпись похожей на гиперссылку (для большего эффекта для такой надписи можно установить свойство Cursor равным crHandPoint на этапе проектирования формы).

Листинг 3.14. Подчеркивание и изменение цвета надписи

```
procedure TForm1.lblUnderlineMouseEnter(Sender: TObject);
begin
    lblUnderline.Font.Style := [fsUnderline];
    lblUnderline.Font.Color := RGB(0, 0, 255);
end;

procedure TForm1.lblUnderlineMouseLeave(Sender: TObject);
begin
    lblUnderline.Font.Style := [];
    lblUnderline.Font.Color := RGB(0, 0, 0);
end;
```

Осталось добавить обработчик события Click для надписи, и получится довольно правдоподобная гиперссылка, правда, выполнять она может любое действие.

Начертание шрифта можно также изменить для стандартной кнопки. Как это можно сделать, показано в листинге 3.15.

Листинг 3.15. Изменение начертания шрифта

```
procedure TForm1.cmbItalicBoldMouseMove(Sender: TObject;
    Shift: TShiftState; X, Y: Integer);
begin
    cmbItalicBold.Font.Style := [fsItalic, fsBold];
end;

procedure TForm1.lblItalicMouseEnter(Sender: TObject);
begin
    lblItalic.Font.Style := [fsItalic];
end;
```

В листинге 3.15 используется обработчик MouseMove для кнопки потому, что, к великому сожалению, обработчики событий MouseEnter и MouseLeave для нее (по крайней мере, с вкладки Standard) не предусмотрены.

3.2. Клавиатура

Клавиатура является основным средством для ввода информации в компьютер, поэтому не будем обходить стороной и рассмотрим некоторые не так часто используемые или не такие очевидные возможности работы с ней.

Определение информации о клавиатуре

Начнем с небольшого примера, позволяющего определить некоторую информацию о клавиатуре (листинг 3.16). Пример основан на использовании API-функции `GetKeyboardType`.

Листинг 3.16. Определение информации о клавиатуре

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    //Определяем тип клавиатуры
    case GetKeyboardType(0) of
        1: txtType.Text := 'PC/XT или совместимая (83 клавиши)';
        2: txtType.Text := 'Olivetti "ICO" (102 клавиши)';
        3: txtType.Text := 'PC/AT (84 клавиши) или похожая';
        4: txtType.Text := 'Расширенная (101 или 102 клавиши)';
        5: txtType.Text := 'Nokia 1050 или похожая';
        6: txtType.Text := 'Nokia 9140 или похожая';
        7: txtType.Text := 'японская';
    end;
    //Определяем код типа производителя
    txtSubtype.Text := IntToStr(GetKeyboardType(1));
    //Определяем количество функциональных клавиш
    txtKeys.Text := IntToStr(GetKeyboardType(2));
end;
```

При создании формы происходит заполнение текстовых полей информацией о типе клавиатуры, коде типа, присвоенном производителем, и количестве функциональных клавиш.

Пример возможного результата определения информации о клавиатуре приводится на рис. 3.2.

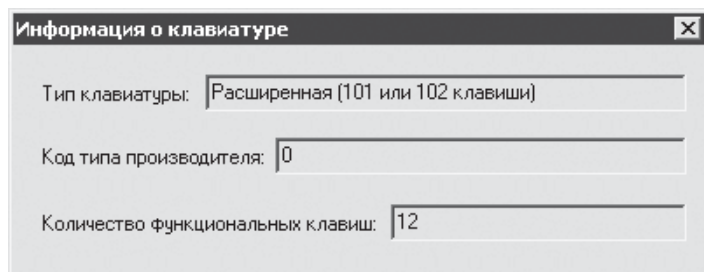


Рис. 3.2. Информация о клавиатуре

Опрос клавиатуры

Существует достаточно удобная альтернатива обработке событий клавиатурного ввода, которая может оказаться особенно полезной, когда нужно знать состояние сразу нескольких клавиш.

В листинге 3.17 приводится пример обработчика события `Timer1Timer`, определяющего, нажаты ли клавиши \uparrow , \downarrow , \leftarrow , \rightarrow , а также пробел, Enter, Ctrl (правый) и Alt (правый).

Листинг 3.17. Определение состояния некоторых клавиш

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  buttons: TKeyboardState;
begin
  //Получаем состояния клавиш
  GetKeyboardState(buttons);
  //Отобразим состояния клавиш
  //..пробел
  if buttons[VK_SPACE] and 128 <> 0 then
    SendMessage(cmbSpace.Handle, BM_SETSTATE, BST_CHECKED, 0)
  else
    SendMessage(cmbSpace.Handle, BM_SETSTATE, BST_UNCHECKED, 0);
  //..enter
  if buttons[VK_RETURN] and 128 <> 0 then
    SendMessage(cmbEnter.Handle, BM_SETSTATE, BST_CHECKED, 0)
  else
    SendMessage(cmbEnter.Handle, BM_SETSTATE, BST_UNCHECKED, 0);
  //..правый Ctrl
  if buttons[VK_RCONTROL] and 128 <> 0 then
    SendMessage(cmbRCtrl.Handle, BM_SETSTATE, BST_CHECKED, 0)
  else
    SendMessage(cmbRCtrl.Handle, BM_SETSTATE, BST_UNCHECKED, 0);
  //..правый Alt
  if buttons[VK_RMENU] and 128 <> 0 then
    SendMessage(cmbRAlt.Handle, BM_SETSTATE, BST_CHECKED, 0)
  else
    SendMessage(cmbRAlt.Handle, BM_SETSTATE, BST_UNCHECKED, 0);
  //..правый Shift
  if buttons[VK_RSHIFT] and 128 <> 0 then
    SendMessage(cmbRShift.Handle, BM_SETSTATE, BST_CHECKED, 0)
```

```
else
    SendMessage(cmbRShift.Handle, BM_SETSTATE, BST_UNCHECKED, 0);
//..вверх
if buttons[VK_UP] and 128 <> 0 then
    SendMessage(cmbUp.Handle, BM_SETSTATE, BST_CHECKED, 0)
else
    SendMessage(cmbUp.Handle, BM_SETSTATE, BST_UNCHECKED, 0);
//..вниз
if buttons[VK_Down] and 128 <> 0 then
    SendMessage(cmbDown.Handle, BM_SETSTATE, BST_CHECKED, 0)
else
    SendMessage(cmbDown.Handle, BM_SETSTATE, BST_UNCHECKED, 0);
//..влево
if buttons[VK_LEFT] and 128 <> 0 then
    SendMessage(cmbLeft.Handle, BM_SETSTATE, BST_CHECKED, 0)
else
    SendMessage(cmbLeft.Handle, BM_SETSTATE, BST_UNCHECKED, 0);
//..вправо
if buttons[VK_RIGHT] and 128 <> 0 then
    SendMessage(cmbRight.Handle, BM_SETSTATE, BST_CHECKED, 0)
else
    SendMessage(cmbRight.Handle, BM_SETSTATE, BST_UNCHECKED, 0);
end;
```

Для того чтобы определить состояние клавиш, можно использовать API-функцию `GetKeyboardState`, которая заполняет массив `buttons` (на самом деле тип `TKeyboardState` определен как `array[0..255] of Byte`) значениями, характеризующими, нажата ли клавиша. Причем значения в массиве `buttons` трактуются следующим образом:

- если установлен старший бит (проверка чего и делается в листинге 3.17), то клавиша в данный момент нажата;
- если установлен младший бит, то функция, закрепленная за этой клавишей (например, `Caps Lock`), включена.

Для индексации массива можно использовать ASCII-коды символов, а также константы, соответствующие несимвольным клавишам (обозначения и коды для таких клавиш приводятся в приложении 1).

Каждой контролируемой клавише (листинг 3.17) соответствует кнопка на форме. Для принудительной установки кнопки в нажатое или ненажатое состояние используется посылка сообщения `BM_SETSTATE`. Пример определения состояния клавиш в некоторый момент времени показан на рис. 3.3.

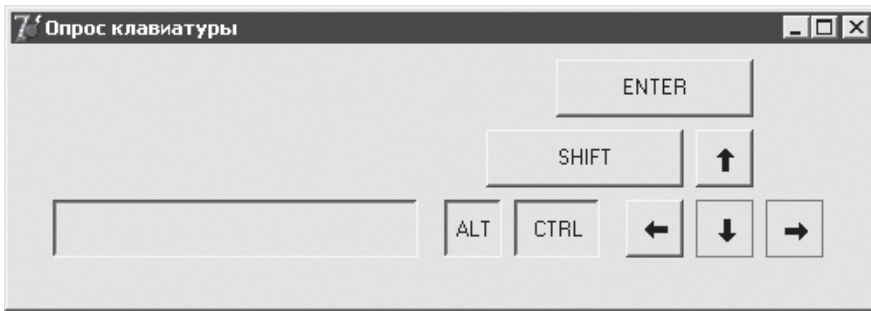


Рис. 3.3. Состояние некоторых клавиш клавиатуры

Интересно, что рассмотренный способ работы с клавиатурой можно использовать даже для определения неисправных клавиш на клавиатуре, например, как это сделано в одной из программ пакета Norton Utilities.

Имитация нажатия клавиш

Состояние клавиш на клавиатуре можно не только определять, его также можно программно изменять. Рассмотрим один из способов программного нажатия клавиш, который крайне прост благодаря наличию API-функции `keybd_event`, как раз и предназначенной для имитации клавиатурного ввода.

Назначения параметров этой функции поясним на примере (листинг 3.18).

Листинг 3.18. Показываем меню Пуск

```
procedure TForm1.cmbStartClick(Sender: TObject);
begin
    //Имитируем нажатие клавиши Windows
    keybd_event(VK_LWIN, 0, 0, 0);
    //Имитируем отпускание клавиши Windows
    keybd_event(VK_LWIN, 0, KEYEVENTF_KEYUP, 0);
end;
```

Нас интересуют, прежде всего, первый и третий параметры функции `keybd_event` (второй не используется, а третий предназначен для установки дополнительной информации, относящейся к нажатию клавиши). Первым параметром функции передается код «нажимаемой» или «отпускаемой» клавиши. Третий параметр равен нулю при «нажатии» и константе `KEYEVENTF_KEYUP` при «отпускании» клавиши.



ВНИМАНИЕ

При использовании `keybd_event` главное — не забывать «отпускать» программно нажатые клавиши (как это делается в приведенных здесь примерах). Иначе есть риск изрядных «глюков» клавиатурного ввода.

Аналогичный приведенному в листинге 3.18 пример программного нажатия клавиши Print Screen (снятия копии экрана) приводится в листинге 3.19.

Листинг 3.19. Снятие копии экрана

```
procedure TForm1.cmbPrintScreenClick(Sender: TObject);
begin
    //Нажимаем Print Screen
    keybd_event(VK_SNAPSHOT, 0, 0, 0);
    keybd_event(VK_SNAPSHOT, 0, KEYEVENTF_KEYUP, 0);
end;
```

В листинге 3.20 приводится пример нажатия комбинации из нескольких клавиш (Windows+M для сворачивания всех окон).

Листинг 3.20. Сворачивание всех окон

```
procedure TForm1.cmbMinimizeAllClick(Sender: TObject);
begin
    //Нажимаем Windows+M
    keybd_event(VK_LWIN, 0, 0, 0);
    keybd_event(Byte('M'), 0, 0, 0);
    keybd_event(Byte('M'), 0, KEYEVENTF_KEYUP, 0);
    keybd_event(VK_LWIN, 0, KEYEVENTF_KEYUP, 0);
end;
```

Добавление к этой комбинации клавиши Shift приведет к восстановлению первоначального состояния окон.

Последний пример иллюстрирует, как можно использовать программное нажатие клавиш для ускорения быстрого доступа к программам. Имеется в виду программное нажатие сочетаний клавиш, ассоциированных с ярлыками, расположенными на Рабочем столе или находящимися в меню Пуск. Допустим, на компьютере используется сочетание клавиш Ctrl+Alt+E для запуска Internet Explorer. Пример программного нажатия этой комбинации клавиш приведен в листинге 3.21.

Листинг 3.21. Быстрый запуск программы

```
procedure TForm1.cmbEExplorerClick(Sender: TObject);
begin
    //Нажимаем комбинацию Ctrl+Alt+E
    keybd_event(VK_CONTROL, 0, 0, 0);
    keybd_event(VK_MENU, 0, 0, 0);
    keybd_event(Byte('E'), 0, 0, 0);
    keybd_event(Byte('E'), 0, KEYEVENTF_KEYUP, 0);
end;
```

```

    keybd_event(VK_MENU, 0, KEYEVENTF_KEYUP, 0);
    keybd_event(VK_CONTROL, 0, KEYEVENTF_KEYUP, 0);
end;
```

Последний пример особенно полезен для запуска сразу нескольких программ (для этого ярлыкам этих программ должны быть назначены сочетания клавиш).

«Бегущие огни» на клавиатуре

В завершение рассмотрим довольно забавный пример, так же, как и предыдущий, основанный на программном нажатии клавиш Caps Lock, Num Lock и Scroll Lock. Как известно, этим клавишам соответствуют три лампочки (по крайней мере, на большинстве клавиатур). Суть примера состоит в последовательном включении/выключении упомянутых клавиш, которое автоматически сопровождается включением/выключением соответствующих лампочек.

Перед рассмотрением основных процедур примера приведем текст процедуры PressKey, которая далее используется практически на каждом шагу (листинг 3.22). Она имитирует нажатие одной клавиши с переданным кодом.

Листинг 3.22. Нажатие одной клавиши

```

procedure PressKey(keyCode: Byte);
begin
    keybd_event(keyCode, 0, 0, 0);
    keybd_event(keyCode, 0, KEYEVENTF_KEYUP, 0);
end;
```

Запуск и остановка огней осуществляется при нажатии кнопки (помимо кнопки, на форме должно быть текстовое поле, в котором вводится интервал между смежной состояниями огней, а также таймер со свойством Enabled, равным False) (листинг 3.23).

Листинг 3.23. Запуск и остановка огней

```

var
    initCaps, initNum, initScroll: Boolean; //Первоначальные
                                           //состояния клавиш
    curCaps, curNum, curScroll: Boolean;    //Текущие состояния
                                           //клавиш

procedure TForm1.cmbStartClick(Sender: TObject);
begin
    if cmbStart.Caption = 'Старт' then
    begin
        //Сохраняем первоначальные состояния клавиш
        initCaps := (GetKeyState(VK_CAPITAL) and 1) <> 0;
```

```

    initNum := (GetKeyState(VK_NUMLOCK) and 1) <> 0;
    initScroll := (GetKeyState(VK_SCROLL) and 1) <> 0;
    //Включаем только Caps Lock
    if not initCaps then PressKey(VK_CAPITAL);
    curCaps := True;
    if initNum then PressKey(VK_NUMLOCK);
    curNum := False;
    if initScroll then PressKey(VK_SCROLL);
    curScroll := False;
    //Запускаем "бегущие огни"
    Timer1.Interval := StrToInt(txtInterval.Text);
    Timer1.Enabled := True;
    cmbStart.Caption := 'Стоп';
end
else
begin
    //Останавливаем "бегущие огни"
    Timer1.Enabled := False;
    cmbStart.Caption := 'Старт';
    //Восстанавливаем первоначальные состояния клавиш
    if initCaps <> curCaps then PressKey(VK_CAPITAL);
    if initNum <> curNum then PressKey(VK_NUMLOCK);
    if initScroll <> curScroll then PressKey(VK_SCROLL);
end;
end;

```

В начале листинга 3.23 приведены используемые глобальные переменные:

- `initCaps`, `initNum`, `initScroll` — для сохранения первоначального состояния клавиш `Caps Lock`, `Num Lock` и `Scroll Lock` с целью его восстановления при остановке огней, чтобы не раздражаться необходимостью вручную устанавливать состояния этих клавиш;
- `curCaps`, `curNum`, `curScroll` — для быстрого определения текущего состояния клавиш (вместо постоянного обращения к функциям типа `GetKeyboardState`).

Перемещение огней происходит при каждом срабатывании таймера `Timer1` (листинг 3.24).

Листинг 3.24. Перемещение огней

```

procedure TForm1.Timer1Timer(Sender: TObject);
begin
    //Изменяем состояние лампочек на клавиатуре
    if curCaps then

```

```
begin
    //C Caps Lock на Num Lock
    PressKey(VK_NUMLOCK);
    PressKey(VK_CAPITAL);
    curCaps := False;
    curNum := True;
end
else if curNum then
begin
    //C Num Lock на Scroll Lock
    PressKey(VK_SCROLL);
    PressKey(VK_NUMLOCK);
    curNum := False;
    curScroll := True;
end
else
begin
    //C Scroll Lock на Caps Lock
    PressKey(VK_CAPITAL);
    PressKey(VK_SCROLL);
    curScroll := False;
    curCaps := True;
end;
end;
```



ПРИМЕЧАНИЕ

Если у вашей клавиатуры порядок следования лампочек отличается от приведенного в примере (в какую-нибудь сторону), то следует изменить порядок переключения в листинге 3.24, чтобы «бегущие огни» действительно «бежали».

Теперь можно запустить соответствующую заставку и получить неплохое украшение, например, для новогодней елки... из компьютера.



Глава 4

Диски, каталоги, файлы

- ☐ Диски
- ☐ Каталоги и пути
- ☐ Файлы

В этой главе вы познакомитесь с некоторыми возможностями получения полезной информации о файловой системе (и от файловой системы). Примеры главы целиком основаны на использовании API-функций для получения информации, так сказать, из первых рук.

Конечно, разработчики Borland не проигнорировали эту тему при написании библиотеки для Delphi: в модуле `SysUtils` можно найти ряд функций, позволяющих работать с объектами файловой системы. Поэтому в этой главе в основном рассматриваются API-функции, позволяющие получить информацию, недоступную при использовании процедур и функций модуля `SysUtils`, дабы полностью не дублировать функционал этой библиотеки.

4.1. Диски

Начнем с получения информации о дисках компьютера. Как вы, наверное, не раз могли убедиться, ряд приложений (хотя бы тот же Internet Explorer) обладают гораздо большей информацией о дисках, нежели их обозначение (буква) или размер. Далее рассмотрено, как определить буквы всех установленных на компьютере дисков, метки дисков, серийные номера томов и другую информацию о файловой системе. Вы также узнаете, как программно поменять метки дисков.

Все рассмотренные ниже функции работы с дисками вы можете найти в модуле `DriveTools`, расположенном на диске, прилагаемом к книге, в папке с названием раздела.

Сбор информации о дисках

Итак, начнем по порядку. Получить список дисков компьютера (строк вида <буква>:\) поможет функция из листинга 4.1.

Листинг 4.1. Определение букв дисков

```
function GetDriveLetters(letters: TStrings):Integer;
var
  buffer: String;
  i, len, start: Integer;
begin
  SetLength(buffer, 110);
  len := GetLogicalDriveStrings(110, PAnsiChar(buffer));
  //Разбираем строку вида 'c:\#0d:\#0...\#0#0',
  //возвращаемую функцией GetLogicalDriveStrings
  start := 1;
  for i := 2 to len do
    if (buffer[i] = #0) and (start <> i) then
      begin
        //Нашли обозначение очередного диска
```

```
        letters.Append(Copy(buffer, start, i-start));
        start := i+1;
    end;
    GetDriveLetters := letters.Count;
end;
```

Функция принимает ссылку на список и заполняет его строками с путями корневых папок каждого из дисков (например, c:\). Вся сложность этой функции состоит в необходимости выделения путей из строки, заполняемой API-функцией GetLogicalDriveStrings. Функция GetDriveLetters возвращает количество строк, добавленных в список letters.

Кроме API-функции GetLogicalDriveStrings, для получения информации о том, за какими буквами закреплены диски, можно использовать еще как минимум одну функцию — GetLogicalDrives. Она не имеет аргументов и возвращает значение типа DWORD, представляющее собой битовую маску. Состояние каждого бита маски (от 1 до 26) соответствует наличию либо отсутствию диска под соответствующей номеру буквой латинского алфавита. Выделение информации из маски (и соответственно составление списка дисков) может выглядеть, как в листинге 4.2.

Листинг 4.2. Составление списка дисков

```
function GetDriveLetters(letters: TStrings):Integer;
var
    mask: DWORD;
    i: Integer;
    letter: Char;
begin
    //Получаем маску, характеризующую наличие дисков
    mask := GetLogicalDrives();
    //Разбираем маску (определяем значения первых 26 битов)
    i := 1;
    for letter := 'A' to 'Z' do
        begin
            if mask and i <> 0 then
                //Есть диск под текущей буквой
                letters.Append(letter + ':\');
            i := i * 2; //Переходим к следующему биту
        end;
    GetDriveLetters := letters.Count;
end;
```

Теперь напомним несложные функции, позволяющие определить полный размер и размер свободного пространства на диске (листинг 4.3).

Листинг 4.3. Определение полного размера и размера свободного пространства диска

```
//Функция возвращает полный размер диска в байтах
function GetDriveSize(root: String): Int64;
var
    freeToCaller, totalBytes, freeBytes: Int64;
begin
    if GetDiskFreeSpaceEx(PAnsiChar(root), freeToCaller,
        totalBytes, PLargeInteger(Addr(freeBytes))) <> False
    then
        GetDriveSize := totalBytes
    else
        GetDriveSize := -1;
end;

//Функция возвращает размер свободного места на диске (в байтах)
function GetDriveFreeSpace(root: String): Int64;
var
    freeToCaller, totalBytes, freeBytes: Int64;
begin
    if GetDiskFreeSpaceEx(PAnsiChar(root), freeToCaller,
        totalBytes, PLargeInteger(Addr(freeBytes))) <> False
    then
        GetDriveFreeSpace := freeBytes
    else
        GetDriveFreeSpace := -1;
end;
```

В обеих функциях листинга 4.3 для достижения двух разных целей используется API-функция GetDiskFreeSpaceEx:

```
function GetDiskFreeSpaceEx(lpDirectoryName: PChar;
    var lpFreeBytesAvailableToCaller,
    lpTotalNumberOfBytes;
    lpTotalNumberOfFreeBytes: PLargeInteger): BOOL;
```

Функция принимает путь (любой) файла или папки на интересующем диске и заполняет три параметра:

- lpFreeBytesAvailableToCaller — размер свободного пространства, доступного пользователю, под чьими правами работает поток, вызывающий функцию (в байтах);

- `lpTotalNumberOfBytes` — полный размер диска (в байтах);
- `lpTotalNumberOfFreeBytes` — размер свободного пространства на диске (в байтах).

Все перечисленные значения являются 64-битными, чтобы можно было оперировать размерами дисков более 4 Гбайт. Если вызов функции `GetDiskFreeSpaceEx` оказывается неудачным, то возвращается значение `False`. В этом случае функции листинга 4.3 возвращают `-1`, сигнализируя об ошибке.

Теперь самое интересное — определение детальной информации о файловой системе на дисках. Много интересного о файловой системе на каждом диске можно узнать при помощи API-функции `GetVolumeInformation`. Она имеет следующий вид:

```
function GetVolumeInformation(lpRootPathName: PChar;  
    lpVolumeNameBuffer: PChar; nVolumeNameSize: DWORD;  
    lpVolumeSerialNumber: PDWORD; var lpMaximumComponentLength,  
    lpFileSystemFlags: DWORD; lpFileSystemNameBuffer: PChar;  
    nFileSystemNameSize: DWORD): BOOL;
```

Объявление функции выглядит довольно громоздким за счет большого количества параметров. Однако использовать функцию `GetVolumeInformation` очень просто. Чтобы не вдаваться в долгое описание ее параметров, рассмотрим ее использование на примере (листинг 4.4).

Листинг 4.4. Определение информации о диске

```
//Функция определяет информацию о диске  
//Возвращает False, если возникла ошибка  
function GetDriveInformation(root: String;  
    var info: DriveInfo):Boolean;  
  
var  
    bufDriveName, bufFSName: String;  
    SN: DWORD;  
    maxFileName, fsOptions: Cardinal;  
begin  
    SetLength(bufDriveName, 101);  
    SetLength(bufFSName, 101);  
    //Определение информации о диске  
    if GetVolumeInformation(PAnsiChar(root),  
        PAnsiChar(bufDriveName), 100,  
        Addr(SN), maxFileName, fsOptions,  
        PAnsiChar(bufFSName), 100) <> False  
then  
    begin
```

```
//Заполняем структуру информацией о диске
with info do
begin
    DriveLabel := bufDriveName;
    FileSystemName := bufFSName;
    SerialNumber := SN;
    MaxFileNameLen := maxFileName;
    //..параметры файловой системы
    with info.FileSystemOptions do
    begin
        CaseSensitive := fsOptions and FS_CASE_SENSITIVE <> 0;
        SupportCompression := fsOptions and
                                FS_FILE_COMPRESSION <> 0;
        IsCompressed := fsOptions and FS_VOL_IS_COMPRESSED <> 0;
    end;
end;
//Функция отработала успешно
GetDriveInformation := True;
end
else
    //Ошибка
    GetDriveInformation := False;
end;
```

Если проанализировать приведенный листинг, то можно увидеть, что функции `GetVolumeInformation`, кроме пути, принадлежащего диску, передается также:

- буфер для метки диска (и длина этого буфера);
- указатель на переменную типа `DWORD` для записи в нее серийного номера тома диска (присваивается при каждом создании файловой системы, например, после форматирования диска);
- ссылка на переменную типа `Cardinal` для сохранения в ней максимальной длины компонента пути (имени файла или папки);
- ссылка на переменную типа `Cardinal` для сохранения в ней набора битовых флагов с некоторыми параметрами файловой системы;
- буфер для названия файловой системы (и его длина).

Как вы могли заметить, результатом работы приведенной в листинге 4.4 функции `GetDriveInformation` является заполнение структуры (записи) `DriveInfo`. Определение этой структуры (а также вложенной в нее структуры, хранящей некоторые извлеченные из битовой маски `fsOptions` флаги) приводится в листинге 4.5.

Листинг 4.5. Определение записей для хранения информации о диске

Type

```
//Запись некоторых параметров о файловой системе
FSOptions = record
    CaseSensitive: Boolean;           //При уравнивании путей
                                     //учитывает регистр
    SupportCompression: Boolean;     //Файловая система
                                     //поддерживает сжатие
    IsCompressed: Boolean;           //Диск сжат
end;
//Запись, содержащая информацию о диске
DriveInfo = record
    DriveLabel: String;              //Метка диска
    FileSystemName: String;          //Файловая система диска
    FileSystemOptions: FSOptions;    //Параметры файловой системы
    SerialNumber: DWORD;             //Серийный номер тома
    MaxFileNameLen: Cardinal;        //Максимальная длина имени
                                     //файла
end;
```

Напоследок рассмотрим еще одну полезную возможность — определение типа носителя диска при помощи API-функции `GetDriveType`. Она принимает единственный параметр, задающий корневую папку диска (например, `C:\`, причем обратный слэш на конце обязателен). Функция `GetDriveType` возвращает целочисленное значение, идентифицирующее тип диска. Вариант получения текстового описания типов дисков с использованием этой API-функции приведен в листинге 4.6.

Листинг 4.6. Определение типа носителя диска

```
function GetDriveTypeName(root: String): String;
begin
    case GetDriveType(PAnsiChar(root)) of
        DRIVE_UNKNOWN: GetDriveTypeName := 'Не определен';
        DRIVE_REMOVABLE: GetDriveTypeName := 'Сменный';
        DRIVE_FIXED: GetDriveTypeName := 'Фиксированный';
        DRIVE_REMOTE: GetDriveTypeName := 'Удаленный (сетевой)';
        DRIVE_CDROM: GetDriveTypeName := 'Компакт-диск';
        DRIVE_RAMDISK: GetDriveTypeName := 'RAM-диск';
    else
        GetDriveTypeName := '' //Возвращается в случае ошибки
    end;
end;
```

Изменение метки диска

Как вы думаете, сложно ли изменить метку диска? Совсем нет: вся сложность состоит в отыскании нужной функции. В данном случае можно применить API-функцию `SetVolumeLabel` (листинг 4.7).

Листинг 4.7. Изменение метки диска

```
function SetDriveLabel(root, newLabel: String): Boolean;
begin
    SetDriveLabel :=
        SetVolumeLabel(PAnsiChar(root), PAnsiChar(newLabel)) <> False;
end;
```

В листинге 4.7 приведена функция-оболочка для API-функции изменения метки диска, избавляющая нас от необходимости преобразования типов и интерпретации значения, возвращаемого API-функцией.

Программа просмотра свойств дисков

В завершение темы работы с дисками рассмотрим еще небольшой пример, обобщающий сказанное выше. Для этого создадим небольшое приложение, выводящее информацию о любом из дисков компьютера. Приложение должно использовать возможности всех рассмотренных выше функций.

Окно этого приложения приведено на рис. 4.1.

Работа формы, приведенной на рис. 4.1, организована предельно просто. Сначала при создании формы получаем список дисков (а также выделяем первый диск и загружаем информацию о нем) (листинг 4.8).

Листинг 4.8. Составление списка дисков

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    //Загрузка букв дисков
    if GetDriveLetters(cboDrives.Items) > 0 then
        begin
            //Выделим первый диск
            cboDrives.ItemIndex := 0;
            cboDrivesSelect(self);
        end
    else
        Button1.Enabled := False;
end;
```

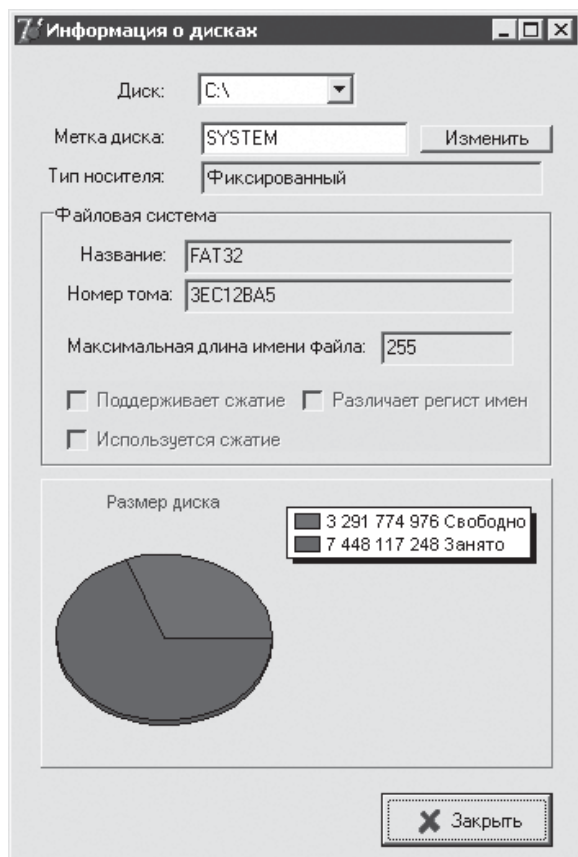


Рис. 4.1. Окно с информацией о дисках

Загрузка информации о дисках происходит при выборе буквы диска в списке (листинг 4.9).

Листинг 4.9. Загрузка информации о выбранном диске

```
procedure TForm1.cboDrivesSelect(Sender: TObject);
var info: DriveInfo;
    root: String;
    fullSize, freeSize: Int64;
begin
    root := cboDrives.Items[cboDrives.ItemIndex];
    //Загружаем информацию о выбранном диске
    GetDriveInformation(root, info);
    //Общая информация о диске и файловой системе
    txtLabel.Text := info.DriveLabel;
```

```
txtDriveType.Text := GetDriveTypeName(root);
txtFSName.Text := info.FileSystemName;
txtSN.Text := IntToHex(Int64(info.SerialNumber), 8);
txtMaxFileName.Text := IntToStr(Integer(info.MaxFileNameLen));
//Флажки некоторых свойств файловой системы
chkCaseSensitive.Checked := info.FileSystemOptions.CaseSensitive;
chkCompression.Checked := info.FileSystemOptions.SupportCompression;
chkCompressed.Checked := info.FileSystemOptions.IsCompressed;
//Размер диска
fullSize := GetDriveSize(root);
if fullSize <> -1 then
    freeSize := GetDriveFreeSpace(root)
else
begin //Ошибка при обращении к диску
    fullSize := 0;
    freeSize := 0;
end;
//..формирование диаграммы
driveSize.Series[0].Clear;
driveSize.Series[0].Add( freeSize, 'Свободно');
driveSize.Series[0].Add( fullSize - freeSize, 'Занято')
end;
```

При нажатии кнопки **Изменить** производится попытка присвоить выбранному в списке диску метку, введенную в соответствующее текстовое поле (txtLabel) (листинг 4.10).

Листинг 4.10. Задание новой метки диска

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    //Изменение метки диска
    if not SetDriveLabel(cboDrives.Items[cboDrives.ItemIndex],
        txtLabel.Text)
    then
        MessageBox(self.Handle, 'Не удалось поменять метку диска',
            'Ошибка', MB_ICONEXCLAMATION)
    else
        //Перечитаем информацию о диске
```

```
cboDrivesSelect(self);  
end;
```

Табличное или иное описание свойств элементов управления не приводится, так как имена элементов управления соответствуют виду информации, помещаемой в них. Только следует уточнить, что в элементе управления TChart создан один ряд типа Pie (круговая диаграмма). У этого ряда отключено отображение подписей к диаграмме, чтобы не дублировать данные, приведенные в легенде.

4.2. Каталоги и пути

В этом разделе описываются некоторые полезные примеры, позволяющие узнавать расположение важных каталогов операционной системы Windows. Здесь также рассматриваются вопросы преобразования путей и приводятся некоторые алгоритмы обхода каталогов, применяемые для поиска.

Прежде чем рассматривать решения конкретных задач, следует уточнить, что за магическое число, а точнее, целочисленная константа используется в некоторых примерах, приведенных далее. Речь идет о константе `MAX_PATH`, равной 260. Она используется явно или неявно (функциями API) в качестве максимально возможной длины пути. Здесь налицо небольшой парадокс: хотя такая файловая система как FAT32, и реализована так, что может поддерживать неограниченную вложенность каталогов, в реальности не получится создать даже два вложенных каталога с именем длиной 255 символов.



ПРИМЕЧАНИЕ

При доскональной проверке вышесказанного выяснилось, что не получится создать даже один каталог с именем длиной 255 символов в корневой папке диска (например, C:\). Каталог может иметь имя длиной максимум 244 символа. С учетом длины имени корневой папки (получается 247) можно предположить, что в таком случае система резервирует оставшиеся 13 символов, чтобы в папке можно было сохранять файлы с именем в формате 8.3 (MS-DOS).

Системные папки WINDOWS и system

Приходилось ли вам хоть раз писать приложения, работоспособность которых зависела от расположения системных папок Windows? Если да, то вы наверняка хорошо знаете, как неустойчиво предположение о том, что папка `WINDOWS` всегда `C:\WINDOWS`, а `system` всегда `C:\WINDOWS\system`. Ведь при установке операционной системы ничто не мешает поместить ее, например, на диск `E:\`, а папку для Windows назвать `Linux`. Кроме того, системная папка Windows на платформе NT имеет имя `system32`, и кто знает, какое имя она будет иметь в следующей версии Windows. В таких и многих других случаях выручат API-функции: `GetWindowsDirectory` и `GetSystemDirectory`. Они обе принимают в качестве параметров строковый буфер и его длину и возвращают количество символов, записанных в переданный буфер, или 0 в случае ошибки.

Для этих функций удобно реализовывать функции-оболочки, работающие со стандартными для Delphi строками, что, собственно, и сделано при написании этой главы (все реализованные функции вы можете найти в модуле `PathFunctions`, расположенном на диске, прилагаемом к книге, в папке с названием подраздела). Итак, функция определения папки Windows приведена в листинге 4.11.

Листинг 4.11. Определение папки WINDOWS

```
function GetWinDir(): String;
var
    buffer: String;
    len: UINT;
begin
    SetLength(buffer, MAX_PATH + 1);
    len := GetWindowsDirectory(PAnsiChar(buffer), MAX_PATH);
    SetLength(buffer, len);
    GetWinDir := buffer;
end;
```

По аналогии реализуется функция определения расположения системной папки, только вместо `GetWindowsDirectory` вызывается функция `GetSystemDirectory`.

Имена для временных файлов

Для централизованного хранения временных данных, необходимых при работе приложений, в Windows предусмотрена специальная папка `Temp`. Ее расположение может варьироваться. Причем в многопользовательских версиях Windows (NT, 2000, XP) местоположение папки для временных файлов может быть различным для различных пользователей. Итак, расположение папки `Temp` поможет определить API-функция `GetTempPath`. Она принимает следующие параметры: строковый буфер и длину этого буфера. Возвращает количество символов, записанных в переданную строку, или 0, если возникла ошибка. Функция-оболочка, скрывающая работу со строковым буфером и преобразование типов, реализуется аналогично двум ранее рассмотренным функциям (листинг 4.12).

Листинг 4.12. Определение расположения папки для временных файлов

```
function GetTempDir(): String;
var
    buffer: String;
    len: UINT;
begin
    SetLength(buffer, MAX_PATH + 1);
    len := GetTempPath(MAX_PATH, PAnsiChar(buffer));
```

```
SetLength(buffer, len);  
GetTempDir := buffer;  
end;
```

Кроме того, Windows API предусматривает очень полезную функцию, избавляющую программиста от необходимости подбирать имена временных файлов так, чтобы они были уникальными в пределах заданной папки (это не обязательно должна быть папка `Temp`). Имя этой функции — `GetTempFileName`. Пример ее использования приведен в листинге 4.13.

Листинг 4.13. Определение имени временного файла

```
function GetTempFile(prefix: String = '~mytmp'): String;  
var  
    buffer, dir: String;  
begin  
    dir := GetTempDir();  
    //Получение имени временного файла (система сама определяет  
имя,  
    //уникальное для заданной папки)  
    SetLength(buffer, MAX_PATH + 1);  
    GetTempFileName(PAnsiChar(dir), PAnsiChar(prefix), 0,  
                    PAnsiChar(buffer));  
    GetTempFile := buffer;  
end;
```

Приведенная в листинге 4.13 функция в качестве папки для временных файлов использует папку `Temp`. Однако функцию `GetTempFileName` можно использовать для получения имен файлов в пределах любой папки.

Кроме пути папки, в которой необходимо создать временный файл, функция `GetTempFileName` принимает строку-префикс для имени временного файла и целочисленное значение (третий параметр). Если третий параметр не равен нулю, то его значение в шестнадцатеричной форме просто прибавляется справа к строке `prefix`. Никаких проверок на уникальность получившегося имени файла при этом не производится. Если же третий параметр установить в `0`, то система сама сформирует шестнадцатеричное значение так, чтобы имя файла было уникальным в заданной папке. Кроме того, при этом создается и сам файл.

Буфер (последний параметр функции `GetTempFileName`) должен вмещать как минимум `MAX_PATH` символов, так как функция записывает в него полный путь временного файла.

Пример работы функций определения папки для временных файлов, получения имени для временного файла, а также определения системных папок Windows приводится на рис. 4.2.

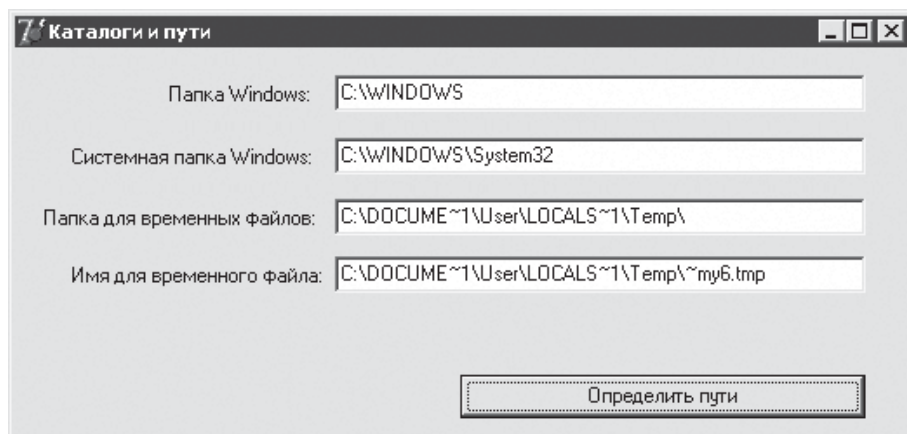


Рис. 4.2. Папки WINDOWS, system, Temp и имя для временного файла

Прочие системные пути

В Windows существует ряд других системных путей, которые так или иначе могут пригодиться. Определяются они не менее просто, чем пути к системным папкам (листинг 4.14).

Листинг 4.14. Определение прочих системных путей

```
function GetSpecialDir(dirtype: Integer): String;
var
    buffer: String;
begin
    SetLength(buffer, MAX_PATH + 1);
    SHGetSpecialFolderPath(0, PAnsiChar(buffer), dirtype, False);
    GetSpecialDir := buffer;
end;
```

Здесь используется функция командной оболочки файловой системы (Windows Shell) `SHGetSpecialFolderPath`, ее объявление находится в модуле `ShlObj`. Среди параметров этой функции самыми значимыми для нас (кроме буфера длиной минимум `MAX_PATH` символов для помещения в него пути) являются два последних. Третий параметр функции `SHGetSpecialFolderPath` используется для указания того, расположение какой именно папки нас интересует. Если четвертый параметр функции `SHGetSpecialFolderPath` не равен `False`, то запрошенная папка будет создана, если до этого она не существовала.

Пример использования функции `GetSpecialDir` для составления списка (в элементе управления `ListView`) некоторых системных путей приведен в листинге 4.15. Из него вы также сможете узнать имена целочисленных констант, идентифицирующих некоторые папки.

Листинг 4.15. Использование функции GetSpecialDir

```
procedure TForm3.Button1Click(Sender: TObject);
var
    item: TListItem;
begin
    lvwPathes.Clear;
    //Определение путей некоторых системных каталогов
    //..Рабочий стол
    item := lvwPathes.Items.Add();
    item.Caption := 'Рабочий стол';
    item.SubItems.Insert(0, GetSpecialDir(CSIDL_DESKTOPDIRECTORY));
    //..Избранное
    item := lvwPathes.Items.Add();
    item.Caption := 'Избранное';
    item.SubItems.Insert(0, GetSpecialDir(CSIDL_FAVORITES));
    //..Шрифты
    item := lvwPathes.Items.Add();
    item.Caption := 'Шрифты';
    item.SubItems.Insert(0, GetSpecialDir(CSIDL_FONTS));
    //..Мои документы
    item := lvwPathes.Items.Add();
    item.Caption := 'Мои документы';
    item.SubItems.Insert(0, GetSpecialDir(CSIDL_PERSONAL));
    //..Последние документы
    item := lvwPathes.Items.Add();
    item.Caption := 'Последние документы';
    item.SubItems.Insert(0, GetSpecialDir(CSIDL_RECENT));
    //..История
    item := lvwPathes.Items.Add();
    item.Caption := 'История';
    item.SubItems.Insert(0, GetSpecialDir(CSIDL_HISTORY));
    //..Отправить
    item := lvwPathes.Items.Add();
    item.Caption := 'Отправить';
    item.SubItems.Insert(0, GetSpecialDir(CSIDL_SENDTO));
    //..Меню Пуск
    item := lvwPathes.Items.Add();
    item.Caption := 'Пуск';
```

```

item.SubItems.Insert(0, GetSpecialDir(CSIDL_STARTMENU));
//..Меню Программы
item := lvwPathes.Items.Add();
item.Caption := 'Программы';
item.SubItems.Insert(0, GetSpecialDir(CSIDL_PROGRAMS));
//..Меню Автозагрузки
item := lvwPathes.Items.Add();
item.Caption := 'Автозагрузка';
item.SubItems.Insert(0, GetSpecialDir(CSIDL_STARTUP));
//..Папка с шаблонами документов
item := lvwPathes.Items.Add();
item.Caption := 'Шаблоны';
item.SubItems.Insert(0, GetSpecialDir(CSIDL_TEMPLATES));
end;
```

Результат работы процедуры из листинга 4.14 приводится на рис. 4.3.

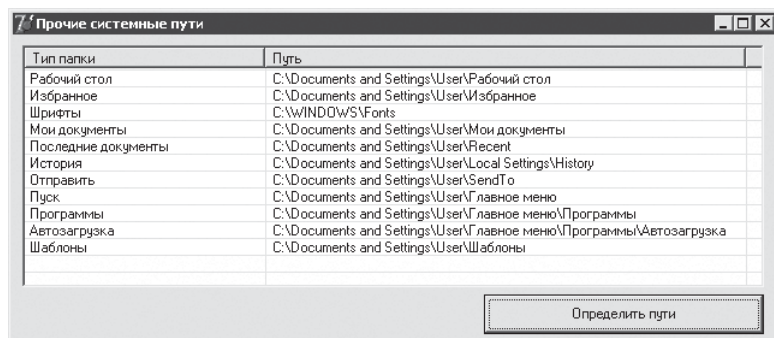


Рис. 4.3. Прочие системные пути Windows

В приведенной в листинге 4.15 процедуре определены не все пути, доступные с использованием функции `SHGetSpecialFolderPath`. Дело в том, что существует ряд виртуальных (не существующих реально на диске) папок: **Мой компьютер**, **Принтеры**, **Сетевое окружение** и т.д.

Для некоторых упоминаемых в листинге 4.15 папок есть также аналогичные папки, содержимое которых доступно всем пользователям:

- `CSIDL_COMMON_DESKTOPDIRECTORY` — содержимое этой папки отображается на Рабочем столе всех пользователей;
- `CSIDL_COMMON_DOCUMENTS` — общие документы;
- `CSIDL_COMMON_FAVORITES` — общие элементы папки **Избранное**;
- `CSIDL_COMMON_PROGRAMS` — общие для всех пользователей программы (пункт **Программы** меню **Пуск**);

- CSIDL_COMMON_STARTMENU — общие элементы, отображаемые в меню Пуск;
- CSIDL_COMMON_STARTUP — общие элементы меню Автозагрузка;
- CSIDL_COMMON_TEMPLATES — папка с общими для всех пользователей шаблонами документов.

**ПРИМЕЧАНИЕ**

Большинство из перечисленных выше путей определяются только в системах Windows на ядре NT, но не в Windows 95/98/Me.

Определение и установка текущей папки

Во время работы каждого приложения для него запоминается папка, которая считается текущей (для этого приложения). При грамотном управлении текущей папкой удобно использовать рассмотренные далее относительные пути.

Для определения текущей папки приложения можно воспользоваться функцией `GetCurrentDir`, приведенной в листинге 4.16.

Листинг 4.16. Определение текущей папки

```
function GetCurrentDir(): String;  
var  
    len: Integer;  
    buffer: String;  
begin  
    SetLength(buffer, MAX_PATH + 1);  
    len := GetCurrentDirectory(MAX_PATH, PAnsiChar(buffer));  
    GetCurrentDir := Copy(buffer, 1, len);  
end;
```

Функция определения пути текущей папки основана на применении соответствующей API-функции `GetCurrentDirectory`. Вполне естественно, что она имеет пару — функцию для задания текущего каталога `SetCurrentDirectory`. Объявление этой функции:

```
function SetCurrentDirectory(lpPathName: PChar): BOOL; stdcall;
```

Функция принимает путь папки и возвращает ненулевое значение в случае успешного выполнения.

Преобразование путей

Рассмотрим несколько функций, которые могут пригодиться, если возникнет необходимость преобразования путей. Имеется в виду прежде всего преобразование

имен файлов в формат MS-DOS и обратно. Этот вид преобразования наглядно продемонстрирован на рис. 4.4 (верхняя часть формы).

Иногда оказывается полезным представлять пути относительно какой-нибудь папки, но не относительно корневого каталога диска. Например, представьте, что вы разрабатываете приложение, документы которого, являющиеся неделимыми для пользователя, могут фактически состоять из большого количества файлов, расположенных в разных папках (Images, Movies, Embed). Сами папки расположены в том же каталоге, где и основной файл документа, или ниже по иерархии (во вложенных папках). Как добиться того, чтобы при копировании приложения со всеми нужными папками в другое место (на другой диск или компьютер, в другую папку) его по-прежнему можно было открыть, при этом рассчитывая, что в папках Images, Movies и Embed содержится не только нужная информация. Последнее говорит о том, что приложение должно «знать», какие файлы и в каких папках ему действительно необходимы. В таком случае пригодится относительный путь, который несет в себе информацию о количестве и направлении переходов из каталога, заданного в качестве корневого, для того чтобы мы смогли найти указанный в этом пути файл или папку.

Преобразование из абсолютного в относительный путь и наоборот продемонстрировано на рис. 4.4 (нижняя часть формы). При этом в качестве исходного пути берется содержимое текстового поля Исходный длинный путь, а в качестве пути папки для построения относительного пути — содержимое поля Текущая папка.

Преобразование путей

Длинные пути в короткие и наоборот

Исходный длинный путь: C:\Documents and Settings\User\borland\registry.slm

Короткий путь: C:\DOCUME~1\User\BORLAN~1\registry.slm

Длинный путь, полученный из короткого: C:\Documents and Settings\User\borland\registry.slm

Преобразовать

Абсолютные пути в относительные (по отношению к заданной папке) и наоборот

Текущая папка: C:\Documents and Settings\Администратор

Относительный путь: ..\User\borland\registry.slm

Абсолютный путь, полученный из относительного: C:\Documents and Settings\User\borland\registry.slm

Преобразовать

Рис. 4.4. Преобразование путей

На всякий случай нужно уточнить, что в относительном пути элемент `.` указывает на текущую папку (никуда переходить не надо), а элемент `..` означает папку, расположенную на один уровень выше (родительскую папку). Также следует уточнить, что под абсолютным путем понимается путь, корневым элементом которого является `\\` или `<диск>:\` (`C:\`, `D:\` и т.д.).

**ПРИМЕЧАНИЕ**

Все приведенные далее функции преобразования вы можете найти в модуле PathConvert, расположенном на диске, в папке с названием подраздела.

Преобразование длинных имен файлов в короткие и наоборот

Теперь рассмотрим реализацию преобразования путей. Сначала — преобразование между длинной и короткой формами. Выполняется это предельно просто, благодаря Windows API предусматривает соответствующие функции.

Преобразование длинного пути в короткий приводится в листинге 4.17.

Листинг 4.17. Преобразование пути из длинной в короткую форму

```
function LongPathToShort(path: String): String;
var
  buffer: String;
  len: Integer;
begin
  SetLength(buffer, MAX_PATH);
  len := GetShortPathName(PAnsiChar(path), PAnsiChar(buffer),
MAX_PATH);
  SetLength(buffer, len);
  LongPathToShort := buffer;
end;
```

Соответственно, обратное преобразование пути может выглядеть следующим образом (листинг 4.18).

Листинг 4.18. Преобразование пути из короткой в длинную форму

```
function ShortPathToLong(path: String): String;
var
  buffer: String;
  len: Integer;
begin
  SetLength(buffer, MAX_PATH);
  len := GetLongPathName(PAnsiChar(path), PAnsiChar(buffer),
MAX_PATH);
  SetLength(buffer, len);
  ShortPathToLong := buffer;
end;
```

При тестировании последнего листинга в Delphi 7 выяснилось, что API-функция GetLongPathName объявлена в модуле Windows. Возможно, в более старых или

новых версиях Delphi это не так. Но в любом случае импортировать эту функцию из библиотеки `Kernel32.dll` предельно просто, достаточно поместить в модуль следующую строку:

```
function GetLongPathName(lpszLongPath: PChar;  
    lpszShortPath: PChar; cchBuffer: DWORD): DWORD;  
    stdcall; external kernel32 name 'GetLongPathNameA';
```

Преобразование абсолютного пути в относительный и наоборот

Теперь пришла очередь рассмотреть реализацию преобразований между абсолютной и относительной формами путей. Однако сначала рассмотрим небольшую, но полезную процедуру, используемую при преобразованиях. Процедура `GetPathElements` (листинг 4.19) формирует список строк из компонентов переданного ей пути (имен каталогов и имени целевого файла или каталога).

Листинг 4.19. Разбиение пути на составляющие

```
procedure GetPathElements(path: String; elements: TStrings);  
var  
    start, pos: Integer;  
begin  
    start := 1;  
    for pos := 1 to Length(path) do  
        if path[pos] = '\' then  
            begin  
                if start <> pos then  
                    //Выделим имя каталога  
                    elements.Add(Copy(path, start, pos - start))  
                else  
                    //Сочетание типа '\\\ в середине пути пропускаем  
                    ;  
                start := pos + 1;  
            end;  
        pos := Length(path) + 1;  
        if start <> pos then  
            //Выделим имя последнего каталога или файла  
            elements.Add(Copy(path, start, pos - start));  
    end;
```

После применения процедуры `GetPathElements` работать с компонентами пути становится очень удобно, да к тому же и упрощается код функций преобразования,

так как при их написании не нужно уделять внимание правильному выделению подстрок из строки полного пути.

Функция преобразования абсолютного пути в относительный (от заданной в параметре `curdir` папки) приводится в листинге 4.20.

Листинг 4.20. Преобразование абсолютного пути в относительный

```
function AbsPathToRelative(path, curdir: String): String;
var
  pathElements, curElements: TStrings;
  outPath: String;
  i, j: Integer;
begin
  if Copy(path, 1, 2) <> Copy(curdir, 1, 2) then
    begin
      //Папки на разных дисках
      AbsPathToRelative := path;
      Exit;
    end;
  //Получение составляющих абсолютного и текущего пути
  pathElements := TStringList.Create;
  GetPathElements(path, pathElements);
  curElements := TStringList.Create;
  GetPathElements(curdir, curElements);
  //Пропускаем одинаковые папки
  i := 0;
  while (i < curElements.Count) and (i < pathElements.Count)
    and (CompareText(curElements[i], pathElements[i]) = 0) do Inc(i);
  //Добавляем необходимое количество переходов вверх для того,
  //чтобы из папки curdir попасть в общую для path и curdir папку
  for j := i to curElements.Count-1 do
    outPath := outPath + '..\';
  //Заходим из папки полученной (общей) папки в папку path
  for j := i to pathElements.Count - 2 do
    outPath := outPath + pathElements[j] + '\';
  //Последним добавляем имя конечной папки или файла
  AbsPathToRelative := outPath + pathElements[pathElements.Count - 1];
  //Списки строк больше не нужны
  pathElements.Free;
```

```
curElements.Free;
end;
```

При преобразовании нужно учитывать, что пути, не принадлежащие одной иерархии (например, локальный и сетевой или пути, принадлежащие разным дискам, не могут быть представлены один относительно другого: у них нет общего родительского каталога.

Обратное преобразование относительного пути в абсолютный приведено в листинге 4.21. Здесь нужно отметить, что если путь папки `curdir` относительный, то в итоге получим также относительный путь (только относительно другой папки). Поэтому функция и называется `RelativePathToRelative`, а не `RelativePathToAbs`.

Листинг 4.21. Преобразование относительного пути в абсолютный

```
function RelativePathToRelative(path, curdir: String): String;
var
  pathElements, curElements: TStringList;
  outPath: String;
  i: Integer;
begin
  //Получение списка составляющих абсолютного и текущего пути
  pathElements := TStringList.Create;
  GetPathElements(path, pathElements);
  curElements := TStringList.Create;
  GetPathElements(curdir, curElements);
  //Изначально находимся в последней папке пути curdir
  //"Путешествуем" от текущей папки вверх или вниз
  //по дереву каталогов
  //(прибавляя или удаляя компоненты пути в список curElements)
  for i := 0 to pathElements.Count-1 do
    begin
      if pathElements[i] = '..' then
        //Вверх по дереву
        if (curElements.Count > 0) then
          curElements.Delete(curElements.Count - 1)
        else
          curElements.Append('..')
      else if pathElements[i] <> '.' then
        //Вниз по дереву (знак текущей папки "." не изменяет
        //положение)
        curElements.Append(pathElements[i]);
    end;
```

```
//Формируем результирующий путь
if (curElements.Count > 0) then outPath := curElements[0];
for i := 1 to curElements.Count-1 do
    outPath := outPath + '\' + curElements[i];
RelativePathToRelative := outPath;
//Списки строк больше не нужны
pathElements.Free;
curElements.Free;
end;
```

Поиск

Поиск является неотъемлемой частью работы с файловой системой. Даже простой просмотр содержимого любого каталога сопряжен с использованием простейших, но все-таки поисковых средств (перебор и, возможно, отсеивание элементов каталога). Поэтому далее мы рассмотрим возможные варианты реализации двух удобных функций поиска: поиск по маске и атрибутам файлов в пределах заданной папки и такой же поиск по всему дереву каталогов, начиная от заданной корневой папки. Все рассмотренные далее функции поиска можно найти в модуле `Search`, расположенном на диске, в папке с названием подраздела.

Но сначала немного сведений о масках для поиска и атрибутах файлов (и папок).

Маски и атрибуты

Маска имени файла или папки представляет собой строку, в которой неизвестный одиночный символ можно менять на `?`, а произвольное количество (0 и более) неизвестных заранее символов — на `*`. Остальные (допустимые в имени) символы обозначают сами себя. Например, имена файлов `SomeFile.exe` и `Some.exe` удовлетворяют каждой из масок: `Some*` и `Some*.exe`.

Атрибуты определяют некоторые важные особенности файла. Так, например, при просмотре каталога при помощи API-функций папка может отличаться от файла только наличием атрибута `FILE_ATTRIBUTE_DIRECTORY`. Вообще содержимое папки (директории, каталога) записано на диске в самый обычный файл. Его отличает наличие указанного неизменяемого вручную атрибута и строго заданный формат записей, а также наличие специальных функций, скрывающих от нас все особенности работы с данными каталога (открытие файла, поиск нужных записей).

Итак, далее об атрибутах. Ниже приводится перечень наиболее часто используемых атрибутов файлов и каталогов (идентификаторы целочисленных констант, объявленных в модуле `Windows`). Если не сказано иное, атрибут можно изменить.

- `FILE_ATTRIBUTE_ARCHIVE` — архивный файл или каталог (на опыте замечено, что этот атрибут появляется практически у всех файлов, находящихся на диске некоторое время);
- `FILE_ATTRIBUTE_DIRECTORY` — атрибут каталога (атрибут нельзя самостоятельно снять или назначить);

- `FILE_ATTRIBUTE_HIDDEN` — скрытый файл или каталог;
- `FILE_ATTRIBUTE_NORMAL` — означает отсутствие особых атрибутов у файла или каталога (у последнего, естественно, всегда установлен атрибут `FILE_ATTRIBUTE_DIRECTORY`);
- `FILE_ATTRIBUTE_READONLY` — файл или каталог только для чтения;
- `FILE_ATTRIBUTE_SYSTEM` — системный файл или каталог;
- `FILE_ATTRIBUTE_TEMPORARY` — временный файл (файловая система стремится по возможности хранить все содержимое открытого временного файла в памяти для ускорения доступа к находящимся в нем данным).

Были рассмотрены основные атрибуты, которые могут быть присвоены объектам файловой системы (файлам и папкам), но не было сказано, как получить или установить атрибуты файла или каталога. Атрибуты можно получить при просмотре содержимого каталога (как в рассмотренных далее функциях поиска). А можно использовать для этого API-функцию `GetFileAttributes`. Она принимает путь файла (`PChar`) и возвращает значение типа `DWORD` (32-битное целое значение), представляющее собой битовую маску. Если функция `GetFileAttributes` завершается неудачно, то возвращаемое значение равно `$FFFFFFFF` (-1 при переводе к беззнаковому целому).

Каждому из рассмотренных атрибутов соответствует бит в возвращаемом функцией `GetFileAttributes` значении. Вот отрывок программы, определяющей, является ли файл системным:

```
var attrs: DWORD;
begin
    attrs := GetFileAttribute(PAnsiChar('C:\boot.ini'));
    if (attrs and FILE_ATTRIBUTE_SYSTEM <> 0) then {файл системный};
```

Атрибуты устанавливаются при помощи API-функции `SetFileAttributes`. Она принимает два параметра: путь файла или папки (`PChar`) и битовую маску атрибутов. Возвращает 0 (`False`) в случае неудачи и ненулевое значение в противном случае.

Поскольку в функцию `SetFileAttributes` передается маска, хранящая сведения сразу обо всех атрибутах файла или папки, то изменять атрибуты нужно аккуратно (чтобы не удалить установленные ранее). Пример (отрывок программы) «включения» одного и одновременного «выключения» другого атрибута файла приведен в листинге 4.22 (проверка ошибок для простоты не производится).

Листинг 4.22. Изменение атрибутов файла

[illegible]

```
attrs := attrs and not FILE_ATTRIBUTE_ARCHIVE; //Снятие
                                           //атрибута "архивный"
SetFileAttributes('C:\text.txt', attrs);
```

Поиск в указанной папке

Поиск в пределах одной папки представляет собой простой перебор всех элементов каталога с отбором тех, имена которых удовлетворяют маске и заданному набору атрибутов. В приведенном ниже примере (листинг 4.23) используется API-функция `FindFirstFile`, которая начинает просмотр заданного каталога, автоматически отсеивая имена файлов и папок, не удовлетворяющие маске. Функция возвращает дескриптор (`THandle`), используемый для идентификации начатого просмотра папки при продолжении поиска (в функции `FindNextFile`).

После окончания просмотра папки вызывается функция `FindClose`, завершающая просмотр папки. Очень напоминает работу с обычным файлом (открытие, просмотр, закрытие), не так ли?

Листинг 4.23. Поиск в заданной папке

```
function SearchInFolder(folder: String; mask: String; flags: DWORD;
    names: TStrings; addpath: Boolean = False): Boolean;
var
    hSearch: THandle;
    FindData: WIN32_FIND_DATA;
    strSearchPath: String;
    bRes: Boolean; //Если равен True, то нашли хотя бы один
                  //файл или каталог
begin
    strSearchPath := folder + '\' + mask;
    bRes := False;
    //Начинаем поиск
    hSearch := FindFirstFile(PAnsiChar(strSearchPath), FindData);
    if (hSearch <> INVALID_HANDLE_VALUE) then
    begin
        //Ищем все похожие элементы (информация о первом элементе
        //уже записана в FindData функцией FindFirstFile)
        repeat
            if (String(FindData.cFileName) <> '..') and
                (String(FindData.cFileName) <> '.') then
                //Пропускаем . и ..
            begin
                if MatchAttrs(flags, FindData.dwFileAttributes) then
```

```
begin
    //Нашли подходящий объект
    if addpath then
        names.Add(folder + '\' + FindData.cFileName)
    else
        names.Add(FindData.cFileName);
    bRes := True;
end;
end;
until FindNextFile(hSearch, FindData) = False;
//Заканчиваем поиск
FindClose(hSearch);
end;
SearchInFolder := bRes;
end;
```

Результатом работы функции `SearchInFolder` является заполнение списка `names` именами или, если значение параметра `addpath` равно `True`, полными путями найденных файлов и каталогов. Значение параметра `flags` (битовая маска атрибутов) формируется так же, как для функции `SetFileAttributes`. Только одновременно можно установить любые интересующие программиста атрибуты. При нахождении хотя бы одного файла или каталога `SearchInFolder` возвращает значение `True`.

В функции поиска проверка соответствия атрибутов найденных файлов и каталогов производится при помощи дополнительной функции `MatchAttrs`. Код этой функции приведен в листинге 4.24.

Листинг 4.24. Фильтр атрибутов

```
function MatchAttrs(flags, attrs: DWORD): Boolean;
begin
    MatchAttrs := (flags and attrs) = flags;
end;
```

Может показаться, что проверка из одной строки — слишком слабый аргумент для создания отдельной функции. В рассматриваемом примере отдельная функция `MatchAttrs` выделена для того, чтобы сделать отсеивание файлов (и папок) по атрибутам более очевидным.

В листинге 4.24 приводится реализация нестроого фильтра: он принимает файл или папку, если они имеют все установленные в `flags` атрибуты, независимо от наличия файла или папки дополнительных атрибутов. Так, если мы задали `flags := FILE_ATTRIBUTE_READONLY`, то будут найдены как файлы, так и каталоги, а также скрытые, системные и прочие файлы, также имеющие атрибут


```

if (chkArchive.Checked) then flags := flags or
                                FILE_ATTRIBUTE_ARCHIVE;

lblFound.Caption := 'Поиск...';
lstFiles.Clear;
Refresh;
//Поиск (файлы записываются прямо в список на форме)
if not SearchInFolder(txtFolder.Text, txtMask.Text, flags,
                    lstFiles.Items)

then
    lblFound.Caption := 'Поиск не дал результатов'
else
    lblFound.Caption := 'Найдено объектов: ' +
                        IntToStr(lstFiles.Count);

end;

```

Поиск по всему дереву каталогов

В листинге 4.26 приводится одна из возможных реализаций рекурсивного поиска по дереву каталогов. Алгоритм поиска работает следующим образом.

1. Выполняется поиск в папке `folder` (все найденные файлы или папки добавляются в список `names`).
2. Функция `SearchInTree` вызывается для каждого подкаталога `vfolder` для продолжения поиска в поддереве, определяемом подкаталогом.

Листинг 4.26. Поиск по дереву каталогов

```

function SearchInTree(folder, mask: String; flags: DWORD;
                    names: TStrings; addpath: Boolean = False): Boolean;
var
    hSearch: THandle;
    FindData: WIN32_FIND_DATA;
    bRes: Boolean; //Если равен True, то нашли хотя бы один файл
                  или каталог
begin
    //Осуществляем поиск в текущей папке
    bRes := SearchInFolder(folder, mask, flags, names, addpath);
    //Продолжим поиск в каждом из подкаталогов
    hSearch := FindFirstFile(PAnsiChar(folder + '\*'), FindData);
    if (hSearch <> INVALID_HANDLE_VALUE) then
    begin
        repeat
            if (String(FindData.cFileName) <> '..') and
                (String(FindData.cFileName) <> '.') then

```

```
//Пропускаем . и ..
begin
  if (FindData.dwFileAttributes and
      FILE_ATTRIBUTE_DIRECTORY <> 0)
  then
    //Нашли подкаталог — выполним в нем поиск
    if SearchInTree(folder + '\' + String(FindData.cFileName),
        mask, flags, names, addpath)
    then
      bRes := True;
    end;
  until FindNextFile(hSearch, FindData) = False;
  FindClose(hSearch);
end;
SearchInTree := bRes;
end;
```

В функции `SearchInTree` не используется просмотр каталога `folder` вручную (при помощи API-функций) из соображений эффективности. Если захотите, можете реализовать поиск подкаталогов при помощи функции `SearchInFolder`. Правда, при этом нужно будет завести дополнительный список (`TStringList`) для сохранения найденных в текущем каталоге подкаталогов. Элементы списка будут использоваться только один раз: для поиска в подкаталогах.

Возможный результат поиска с использованием функции `SearchInTree` приводится на рис. 4.6.

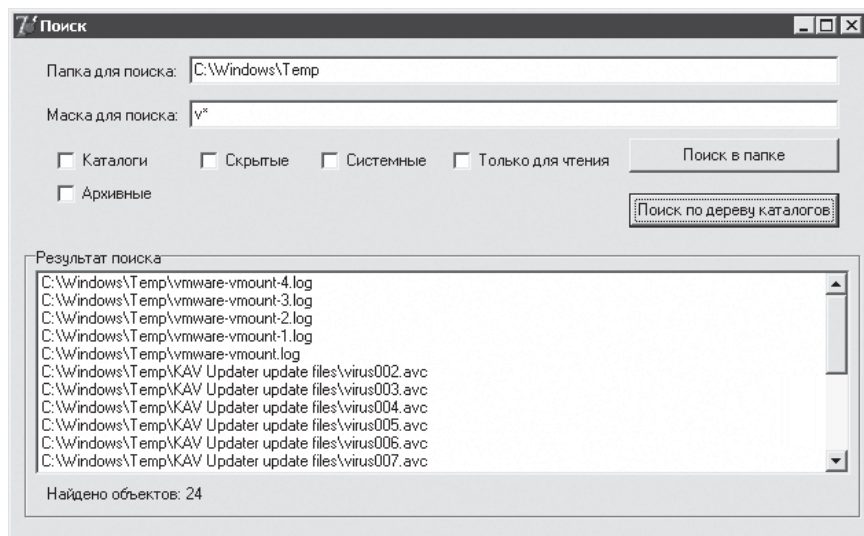


Рис. 4.6. Поиск по дереву каталогов

С небольшими модификациями алгоритм рекурсивного обхода дерева каталогов, реализованный в листинге 4.25, можно использовать и при операциях, отличных от простого поиска: например, при копировании или удалении дерева каталогов. Для этого достаточно выполнять нужные операции над каждым найденным объектом.

Построение дерева каталогов

Рассмотрим довольно интересный пример, основанный на использовании функции поиска `SearchInFolder`, — построение дерева каталогов для определенного диска (рис. 4.7).

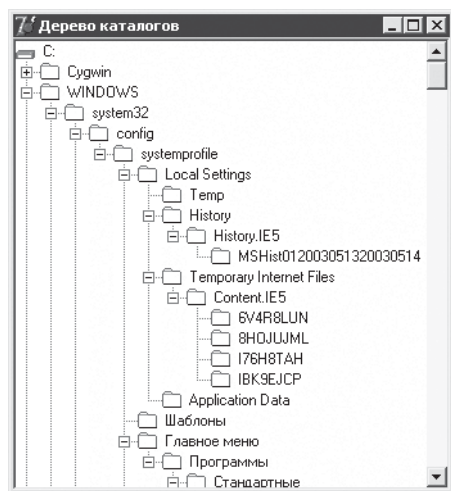


Рис. 4.7. Дерево каталогов

Для простоты (и чтобы не отвлекать внимания от построения дерева) диск задается в программе жестко. При необходимости это можно легко исправить (как определять диски, вы уже знаете).

Рассмотрим работу приложения по порядку. Элемент управления `TreeView` на форме имеет имя `tree`. Содержимое списка изображений (`ImageList`), используемого деревом, приведено на рис. 4.8.

Первый элемент дерева (соответствует диску) образуется при создании формы (листинг 4.27).

Листинг 4.27. Создание первого элемента дерева — диска

```
procedure TForm3.FormCreate(Sender: TObject);
begin
    //Корневой элемент дерева (диск)
    root := tree.Items.Add(tree.Items.GetFirstNode, 'C:');
    root.ImageIndex := 0;
```


Если после создания элементов дерева процедура `SetExpanded` вызывается с параметром `isExpanded`, равным `False` (как в листинге 4.27), то для переданного в процедуру элемента дерева создается фиктивный дочерний элемент. Это делается для того, чтобы не зачитывать содержимое каждого не развернутого еще элемента дерева (для папок с большим количеством файлов программа будет сильно «тормозить»). А так у каждого еще не развернутого элемента отображается символ `+`, позволяющий развернуть его в нужный момент. При этом не нужно забывать удалять созданный фиктивный элемент дерева (что и делает `SetExpanded` с параметром `isExpanded`, равным `True`).

Каждый не развернутый еще элемент дерева помечается значением поля `Node.Data`, равным 0. Каждый элемент, содержимое которого уже прочитано с диска, помечается значением поля `Node.Data`, равным 1. Для проверки, было ли прочитано содержимое каталога, соответствующего элементу дерева, используется простая функция `IsExpanded` (листинг 4.29).

Листинг 4.29. Проверка, загружено ли содержимое каталога

```
function TForm3.IsExpanded(Node: TTreeNode): Boolean;
begin
    IsExpanded := Integer(Node.Data) = 1;
end;
```

Загрузка содержимого каталога и одновременное формирование дочерних элементов в дереве происходят при разворачивании элемента дерева (листинг 4.30).

Листинг 4.30. Загрузка содержимого каталога

```
procedure TForm3.treeExpanding(Sender: TObject; Node: TTreeNode;
                               var AllowExpansion: Boolean);
var
    strFolder: String;
    subfolders: TStrings;
    i: Integer;
    item: TTreeNode;
begin
    if not IsExpanded(Node) then
        //Содержимое каталога нужно зачитать
        SetExpanded(Node, True)
    else
        begin
            //Список подкаталогов для выделенного каталога
            //был составлен ранее
            AllowExpansion := True;
            Exit;
        end;
```

```
//Составление списка подкаталогов
strFolder := NodeToFolderPath( Node );
subfolders := TStringList.Create;
if SearchInFolder(strFolder, '*', FILE_ATTRIBUTE_DIRECTORY,
subfolders)
then begin
    //Добавим в дерево элементы, соответствующие подкаталогам
    for i := 0 to subfolders.Count - 1 do
    begin
        item := tree.Items.AddChild(Node, subfolders[i]);
        item.ImageIndex := 1;
        item.SelectedIndex := 2;
        SetExpanded(item, False); //Содержимое подкаталога еще
                                //не прочитано
    end;
    AllowExpansion := True;
end
else
    //В каталоге нет подкаталогов
    AllowExpansion := False;

subfolders.Free;
end;
```

В листинге 4.30 для определения пути каталога, заданного элементом дерева, используется функция `NodeToFolderPath`. Реализуется она совсем несложно (листинг 4.31).

Листинг 4.31. Определение полного пути элемента дерева

```
function TForm3.NodeToFolderPath(Node: TTreeNode): String;
var
    path: String;
    item: TTreeNode;
begin
    item := Node;
    while item <> nil do
    begin
        if path <> '' then
            path := item.Text + '\' + path
        else
            path := item.Text;
```

```
    item := item.Parent;  
end;  
NodeToFolderPath := path;  
end;
```

Приведенный здесь пример построения дерева может пригодиться при решении некоторых задач. Дополнительно же нужно сказать, что на вкладке **Samples (Delphi 7)** можно найти компоненты, прекрасно подходящие для построения пользовательского интерфейса приложений для просмотра содержимого не только физически существующих дисков: полное дерево каталогов **ShellTreeView** (включая корневой элемент **Рабочий стол** и прочие виртуальные каталоги), список основных элементов системы каталогов (**ShellComboBox**), а также элемент управления для просмотра содержимого папки (**ShellListView**).

4.3. Файлы

В завершение главы рассмотрим три несложных примера работы с файлами: копирование файла (с отображением хода копирования в **ProgressBar**), определение значений, ассоциированных с файлами, и извлечение значков из EXE- и DLL-файлов.

Красивое копирование файла

Казалось бы, что особенного в организации копирования большого файла с отображением процесса: читай файл порциями, записывая прочитанные данные в файл назначения, попутно показывая в **ProgressBar** или где-то еще отношение объема переписанной информации к размеру файла. Однако зачем такие сложности? Ведь у API-функции **CopyFile**, осуществляющей простое копирование файла, есть расширенный вариант — функция **CopyFileEx**, в которую встроена поддержка отображения процесса копирования (и не только это). Вот прототип функции **CopyFileEx**:

```
function CopyFileEx(lpExistingFileName, lpNewFileName: PChar;  
    lpProgressRoutine: TFNProgressRoutine; lpData: Pointer;  
    pbCancel: PBool; dwCopyFlags: DWORD): BOOL; stdcall;
```

Итак, кроме пути исходного и конечного файлов, а также флагов (последний параметр), функция принимает ряд дополнительных параметров: адрес функции обратного вызова (**lpProgressRoutine**), указатель на данные, передаваемые в функцию обратного вызова (**lpData**), а также адрес переменной типа **BOOL** (**pbCancel**), при установке значения которой в **True** копирование прерывается.

Пример использования функции **CopyFileEx** в программе приведен в листинге 4.32. Здесь подразумевается, что кнопка **сmbCopy** используется как для запуска, так и для остановки процесса копирования. Также на форме присутствуют следующие элементы управления:

- индикатор **pbCopyProgress**, диапазон значений которого от 0 до 100;
- текстовое поле **txtFrom** с именем копируемого файла;
- текстовое поле **txtTo** с именем файла назначения.

Листинг 4.32. Использование функции CopyFileEx

```
procedure TForm1.cmbCopyClick(Sender: TObject);
begin
    if cmbCopy.Caption = 'Копировать' then
    begin
        //Запускаем копирование
        progress := pbCopyProgress; //Настроен от 0 до 100 %
        bCancelCopy := False;
        cmbCopy.Caption := 'Отмена';
        if CopyFileEx(PAnsiChar(txtFrom.Text), PAnsiChar(txtTo.Text),
                     Addr(CopyProgressFunc), nil, Addr(bCancelCopy),
                     COPY_FILE_FAIL_IF_EXISTS) = False
        then
            MessageBox(Handle, 'Не удастся скопировать файл',
                       'Копирование', MB_ICONEXCLAMATION);
        end
    else
    begin
        //Останавливаем процесс копирования
        bCancelCopy := True;
        cmbCopy.Caption := 'Копировать';
    end;
end;
```

Из листинга 4.32 можно увидеть, что в качестве значения последнего параметра функции CopyFileEx можно передавать константу COPY_FILE_FAIL_IF_EXISTS (функция вернет False, если файл назначения уже существует, и не будет осуществлять копирование).

На самом деле значение параметра dwCopyFlags функции CopyFileEx может быть комбинацией значений COPY_FILE_FAIL_IF_EXISTS и COPY_FILE_RESTARTABLE, то есть представляет собой битовый флаг. Последнее значение используется для того, чтобы в случае прерывания копирования файла можно было возобновить. Функция CopyFileEx в этом случае сохраняет в файле назначения информацию, достаточную для возобновления процесса копирования.

В листинге 4.32 изменяется переменная progress — глобальная переменная-ссылка на TProgressBar, которая используется в функции обратного вызова. Переменная bCancelCopy, адрес которой передается в функцию CopyFileEx, также объявлена глобальной (в пределах модуля).

Теперь, наконец, рассмотрим функцию обратного вызова, осуществляющую в нашем случае отображение хода копирования на индикаторе (листинг 4.33).

Листинг 4.33. Функция, показывающая ход копирования файла

```
function CopyProgressFunc( TotalFileSize: Int64;
                          TotalBytesTransferred: Int64;
                          StreamSize: Int64;
                          StreamBytesTransferred: Int64;
                          dwStreamNumber: DWORD;
                          dwCallbackReason: DWORD;
                          hSourceFile: THandle;
                          hDestinationFile: THandle;
                          lpData: Pointer): DWORD; stdcall;

begin
    progress.Position := 100 * TotalBytesTransferred div
TotalFileSize;
    Application.ProcessMessages;    //Чтобы не "зависал"
                                   //интерфейс приложения
    CopyProgressFunc := PROGRESS_CONTINUE;
end;
```

Пусть вас не смущает большое количество параметров функции `CopyProgressFunc`. Применять их все далеко не обязательно (но они должны быть объявлены), хотя ничего сложного здесь нет. В листинге 4.33 использование параметров реализовано наиболее простым (на наш взгляд) и очевидным образом: значения параметров `TotalBytesTransferred` и `TotalFileSize` применяются для определения доли скопированной информации.

В листинге 4.33 вызов метода `ProcessMessages` объекта `Application` используется потому, что функция `CopyFileEx` возвращает управление программе только после завершения (или прерывания) копирования. Иначе пришлось бы создавать для копирования отдельный поток, усложняя листинг и отвлекая вас от главной цели этого примера.

Теперь несколько слов о возвращаемых функцией `CopyProgressFunc` значениях (в нашем примере используется только одно из четырех доступных значений). Список целочисленных констант, значения которых может возвращать функция `CopyProgressFunc`, таков:

- `PROGRESS_CONTINUE` — продолжать процесс копирования;
- `PROGRESS_CANCEL` — отмена копирования;
- `PROGRESS_STOP` — остановка копирования (можно возобновить);
- `PROGRESS_QUIET` — при возврате этого значения система перестает вызывать функцию `CopyProgressFunc`.

Внешний вид формы при копировании большого файла приводится на рис. 4.9.

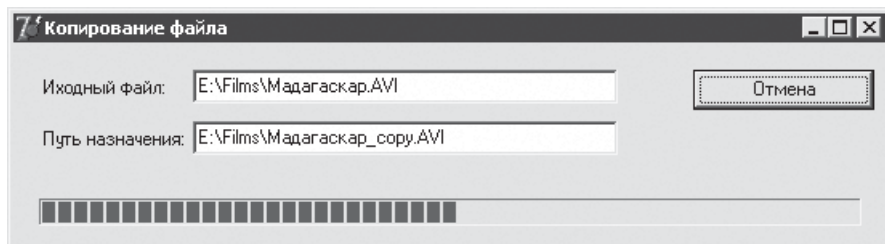


Рис. 4.9. Копирование большого файла

Только не нужно забывать останавливать копирование при закрытии приложения или в прочих экстренных ситуациях. Так, если не предусмотреть обработку события `CloseQuery` для формы (рис. 4.9), то закрыть ее в ходе копирования обычным способом не удастся. Зато после завершения копирования (или при нажатии кнопки `Отмена`) форма тут же исчезнет. Странное поведение, не правда ли? Вариант более-менее адекватной реакции на закрытие формы приводится в листинге 4.34.

Листинг 4.34. Остановка копирования при закрытии формы

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose:
Boolean);
begin
    //Останавливаем процесс копирования
    bCancelCopy := True;
end;
```

Как вариант, можно запретить закрытие формы (установить `CanClose` в `False`), не останавливая копирования.

В том случае, когда копируется несколько файлов, можно ввести дополнительный элемент управления `ProgressBar`, отображающий ход всего процесса копирования. Только при этом придется заранее определить общий размер копируемых файлов.

Определение значков, ассоциированных с файлами

Рассмотрим еще один интересный пример, позволяющий получить значок файла, показываемый, например, в Проводнике Windows. Приведенная в листинге 4.35 функция принимает в качестве параметра путь файла и флаг, определяющий, какой нужен значок — малый или большой. Она возвращает дескриптор экземпляра значка, ассоциированного с файлом. Реализация функции находится в модуле `ShellFunctions`, расположенном на диске, прилагаемом к книге, в папке с названием раздела.

Листинг 4.35. Определение значка файла

```
function GetFileIcon(filename: String; small: Boolean = False ):
HICON;
```

```
var
    info: SHFILEINFO;
    flags: Cardinal;
begin
    flags := SHGFI_ICON;
    if small then
        //Получение малого значка
        flags := flags or SHGFI_SMALLICON
    else
        //Получение большого значка
        flags := flags or SHGFI_LARGEICON;
    ZeroMemory(Addr(info), SizeOf(info));
    //Получение значка
    SHGetFileInfo(PAnsiChar(filename), 0, info, SizeOf(info), flags);

    GetFileIcon := info.hIcon;
end;
```

Используемая в листинге 4.35 API-функция `SHGetFileInfo` объявлена в модуле `ShellApi`. Там же объявлена структура `SHFILEINFO`.

В листинге 4.36 приведен пример использования функции `GetFileIcon`: здесь полученные значки сохраняются в элементах управления `Image` (по одному для большого и малого значков).

Листинг 4.36. Пример получения значка заданного файла (или папки)

```
procedure TForm1.cmbLoadIconClick(Sender: TObject);
begin
    //Определение большого и малого значков файла
    imgLarge.Picture.Icon.Handle := GetFileIcon(txtFile.Text);
    imgSmall.Picture.Icon.Handle := GetFileIcon(txtFile.Text, True);
end;
```

Пример определения значка файла приводится на рис. 4.10.

На самом деле функция из листинга 4.35 может определять значки не только файлов, но и каталогов, дисков и виртуальных папок (`Мой компьютер`, `Рабочий стол`, `Панель управления` и т. д.). Правда, в последнем случае используемая в листинге API-функция `SHGetFileInfo` требует первый параметр специального вида (не строка). Частично работа с таким представлением путей рассмотрена в подразд. «Окно выбора папки» разд. 2.4.

В заключение скажем несколько слов о прочих полезных возможностях API-функции `SHGetFileInfo`. Недаром она называется не `SHGetFileIcon` или что-то

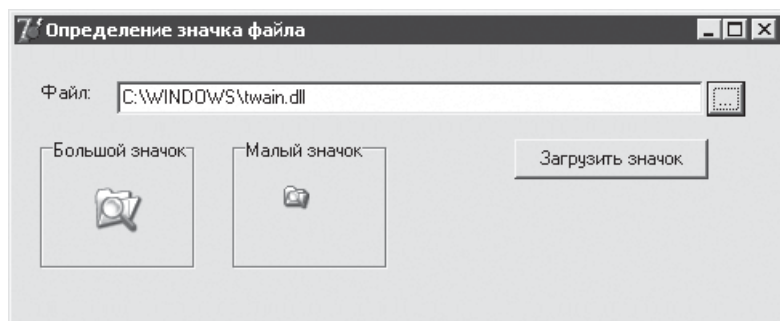


Рис. 4.10. Определение значка, ассоциированного с файлом

подобное: она позволяет получить гораздо больше информации, нежели просто значок файла. Эта информация зависит от набора флагов, передаваемых в функцию в качестве последнего параметра. Но сначала рассмотрим, из каких полей состоит структура `SHFILEINFO`, потому что результат (за редким исключением) помещается именно в ее поля:

- `hIcon` (типа `HICON`) — дескриптор значка заданного путем объекта (первый параметр функции `SHGetFileInfo`);
- `iIcon` (типа `Integer`) — номер значка в системном компоненте `ImageList`;
- `dwAttributes` (типа `DWORD`) — атрибуты заданного путем объекта;
- `szDisplayName` (типа `array[0..MAX_PATH-1] of AnsiChar`) — буфер для имени заданного объекта (например, сочетание имени и метки диска, отображаемое в Проводнике Windows);
- `szTypeName` (типа `array[0..79] of AnsiChar`) — буфер для названия типа файла (например, Документ Microsoft Word).

На полях `dwAttributes` и `iIcon` подробно останавливаться не будем, зато рассмотрим, как заставить функцию `SHGetFileInfo` заполнить остальные поля структуры (их проще всего использовать в Delphi). Вот используемые для этого флаги (имена целочисленных констант):

- `SHGFI_ICON` — поле `hIcon` заполняется дескриптором значка, ассоциированного с объектом; если при использовании дескриптор не сохраняется в каком-либо контейнере или прочем объекте, автоматически удаляющем ненужные значки (как в листинге 4.36), то после использования значков нужно удалить вручную (API-функция `DestroyIcon`);
- `SHGFI_LARGEICON`, `SHGFI_SMALLICON` — они применяются в комбинации с `SHGFI_ICON` для получения большого или малого значков соответственно; использование флагов вместе не имеет смысла (будет получен малый значок);
- `SHGFI_DISPLAYNAME` — при наличии этого флага поле `szDisplayName` будет содержать дружественное пользователю имя объекта (например, `System(C:)`);
- `SHGFI_EXETYPE` — при наличии этого атрибута поле `szTypeName` будет заполнено текстовым описанием типа файла.

Значения в приведенном списке можно, если не сказано иное, комбинировать при помощи операции битового ИЛИ (`or`).

Извлечение значков из EXE- и DLL-файлов

Наверняка вы знаете, что исполняемый файл, помимо кода программы, данных и прочей системной информации, может содержать также ресурсы. Так, из секции ресурсов берутся значки для EXE-файлов. Также в EXE- или DLL-файлах помещаются значки, используемые для ассоциированных с приложениями документов. Итак, в завершение главы рассмотрим еще один графический пример: создадим программу, способную извлекать упомянутые значки из DLL- или EXE-файлов (работает также и для ICO-файлов).

Пусть мы имеем путь файла, а также два списка (`TImageList`) для больших и малых значков соответственно. Тогда процедура, заполняющая списки значками, извлеченными из файла, может выглядеть следующим образом (листинг 4.37).

Листинг 4.37. Составление списков значков

```
procedure LoadIcons(filename: String; lgImages,
                    smImages: TImageList);
var
    icon: TIcon;
    smIconHandle, lgIconHandle: HICON;
    i: Integer;
begin
    //Загрузка каждого значка (неоптимально, но просто)
    i := 0;
    while Integer(
        ExtractIconEx(PAnsiChar(filename), i, lgIconHandle,
                     smIconHandle, 1)
    ) > 0 do
    begin
        Inc(i);
        //Большой значок
        icon := TIcon.Create;
        icon.Handle := lgIconHandle;
        lgImages.AddIcon(icon);
        //Малый значок
        icon := TIcon.Create;
        icon.Handle := smIconHandle;
        smImages.AddIcon(icon);
    end;
```

```
end;  
end;
```

В листинге 4.37 для извлечения значков из файла используется очередная полезная функция модуля `ShellApi` — `ExtractIconEx`. Прототип функции таков:

```
function ExtractIconEx(lpszFile: PChar; nIconIndex: Integer;  
    var phiconLarge, phiconSmall: HICON;  
    nIcons: UINT): UINT;
```

Функция `ExtractIconEx` принимает следующие параметры:

- `lpszFile` — путь файла, из которого извлекаются значки;
- `nIconIndex` — номер первого извлекаемого значка; нумерация начинается с нуля (если номер равен `-1` и параметры `piconLarge` и `piconSmall` нулевые, то функция возвращает количество значков в файле);
- `piconLarge, piconSmall` — ссылки на переменные типа `HICON` (либо на первые элементы массива `array...of HICON`) для помещения в них дескрипторов больших и малых значков соответственно;
- `nIcons` — количество извлекаемых значков (по сути, может быть количество элементов в передаваемых в функцию массивах: лишние элементы не будут заполнены).

Функция возвращает количество значков, извлеченных из файла, или количество значков в файле при соответствующем значении параметра `nIconIndex`.

В листинге 4.36 используется не совсем оптимальный способ извлечения значков из файла — по одному. Однако он подойдет для большинства случаев. Другой (но не единственный) вариант — использование массива. Тогда функции `ExtractIconEx` передаются первые элементы массивов для дескрипторов значков (функции нужен адрес начала массива), а в качестве последнего параметра — количество элементов в массиве. Таким образом, если количество значков в файле превзойдет количество элементов в массиве, то вызов функции `ExtractIconEx` можно будет повторить, передав в качестве параметра `nIconIndex` значение, возвращенное функцией `ExtractIconEx`, умноженное на номер вызова функции (начиная с нуля).

Можно также использовать динамический массив, предварительно установив его размер, вызвав функцию `ExtractIconEx` с параметром `nIconIndex`, равным `-1`. Установить значения параметров `piconLarge, piconSmall` в ноль (не меняя объявления функции) можно, объявив указатель на `HICON` (`^HICON`), присвоив ему значение `nil` и передав его в качестве упомянутых параметров в функцию.

На рис. 4.11 приводится внешний вид формы приложения после извлечения значков из файла `Explorer.exe`.

Обработчик нажатия кнопки `Загрузить значки` представленной на рис. 4.11 формы приводится в листинге 4.38.

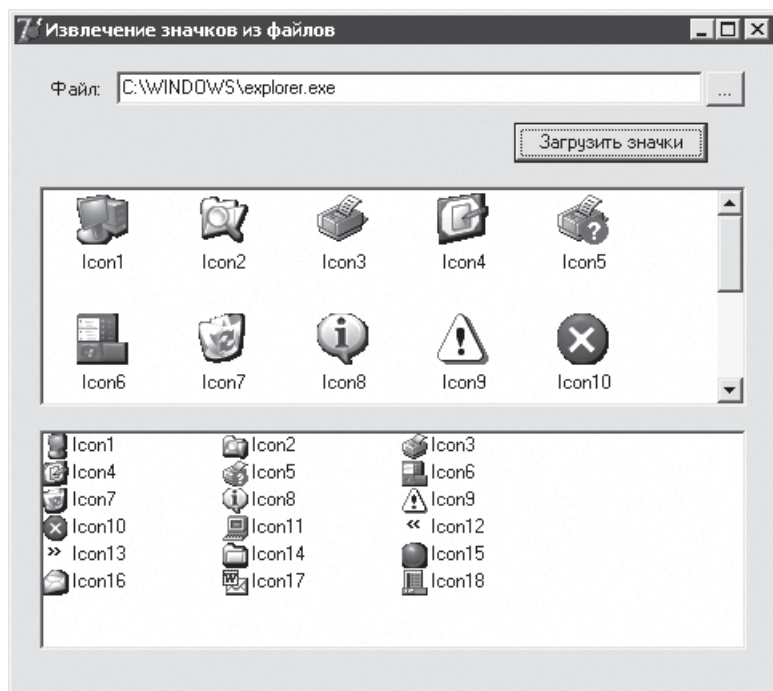


Рис. 4.11. Пример извлеченных из EXE-файла значков

Листинг 4.38. Составление списков значков и их отображение

```
procedure TForm1.cmbLoadIconClick(Sender: TObject);
var
  i: Integer;
  item: TListItem;
begin
  lvwIconsLg.Clear;
  lvwIconsSm.Clear;

  //Загрузка значков в ImageList
  ImageListLg.Clear;
  ImageListSm.Clear;
  LoadIcons(txtFile.Text, ImageListLg, ImageListSm);

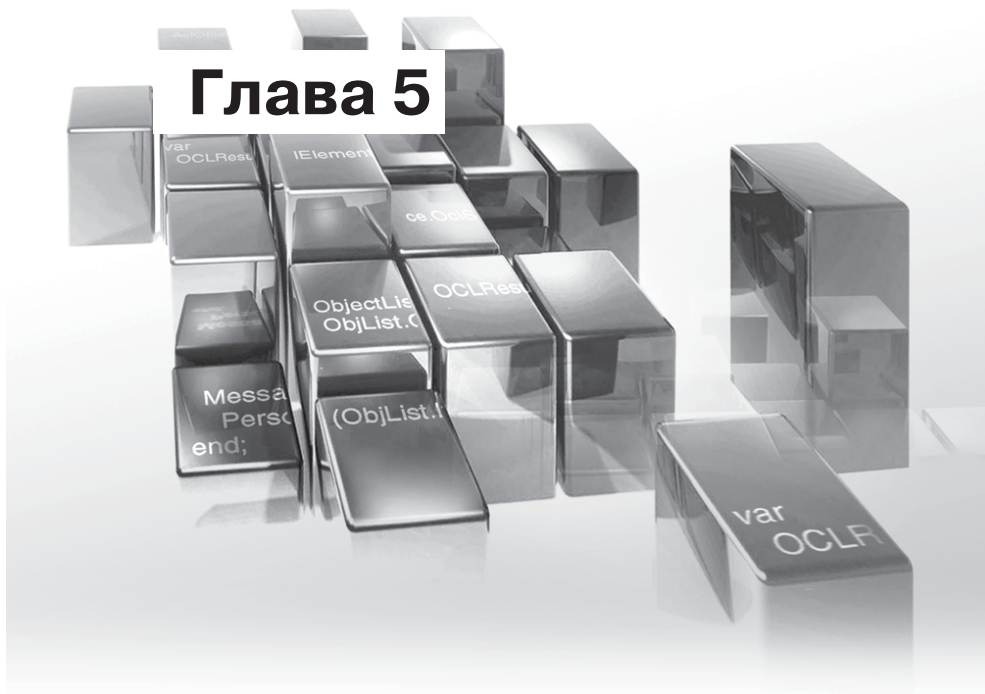
  //Создание элементов в ListView с большими и малыми значками
  for i := 0 to ImageListLg.Count - 1 do
  begin
    item := lvwIconsLg.Items.Add();
```

```
    item.Caption := 'Icon' + IntToStr(i+1);
    item.ImageIndex := i;

    item := lvwIconsSm.Items.Add();
    item.Caption := 'Icon' + IntToStr(i+1);
    item.ImageIndex := i;
end;
end;
```

Подразумевается, что имена элементов управления **ListView**: для отображения больших значков — **lvwIconLg** и для отображения малых **lvwIconSm**. На форме также расположены два элемента управления **ImageList**: **ImageListLg** для хранения больших и **ImageListSm** для хранения малых значков.

С помощью окна **Object Inspector** список **ImageListLg** назначен в качестве источника больших изображений (свойство **LargeImages**) для **lvwIconLg**. Соответственно, список **ImageListSm** назначен в качестве источника малых изображений (свойство **SmallImages**) для **lvwIconSm**.



Мультимедиа

- ☐ Воспроизведение звука с помощью системного динамика
- ☐ Использование компонента MediaPlayer
- ☐ Компонент Animate
- ☐ Разработка звукового проигрывателя
- ☐ Видеопроеигрыватель

Использование мультимедийных технологий позволяет повысить качество программ и придает им профессиональный вид, более привлекательный для пользователя. Среди разнообразных применений мультимедиа наиболее интересны аудио- и видеовозможности компьютера. Использование звуков и видео в программах позволяет иным образом взаимодействовать с пользователем: озвучивать его действия, информировать о некоторых событиях, просматривать видеоролики и т. п.

В рамках предложенной главы будут рассмотрены основные возможности мультимедийных средств и компонентов среды Delphi. Будут описаны компоненты **Animate** и **MediaPlayer**, использование API-функций для генерации звука системным динамиком и для воспроизведения звука из ресурсных файлов.

В отличие от языков Turbo Pascal и Borland Pascal, Delphi не содержит процедур типа **Sound** и **NoSound**, предназначенных для работы со звуком. Для использования мультимедийных возможностей компьютера в Delphi служат специальные компоненты **Animate** и **MediaPlayer**.

Компонент **MediaPlayer** является основным элементом воспроизведения аудио- и видеофайлов. Многофункциональный элемент **MediaPlayer** обладает рядом важных характеристик (свойств) и обеспечивает управление мультимедийными устройствами.

Для создания и воспроизведения простейшей анимации предназначен компонент **Animate**. Он позволяет воспроизводить файлы в формате AVI (Audio-Video Interleaved — Аудио- и видеосмесь).

5.1. Воспроизведение звука с помощью системного динамика

Звуковое сопровождение является важной частью большинства современных мультимедийных приложений. В простейших случаях генерации звукового сигнала удобно использовать процедуру **Beep** модуля **SysUtils**. В этом случае нет необходимости использовать вышеупомянутые мультимедийные компоненты языка, а звук создается встроенным системным динамиком. Процедура **Beep** осуществляет вызов одноименной API-функции, поэтому ее использование не составит большого труда (листинг 5.1).

Листинг 5.1. Генерация звукового сигнала посредством функции **Beep**

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Beep; //Генерация звукового сигнала  
    MessageDlg('Звуковой сигнал был подан', mtError, [mbOK], 0);  
end;
```

Наряду с **Beep** для получения звукового сигнала используется API-функция **MessageBeep(uType: UINT): Boolean**, генерирующая стандартный системный звук, тип которого указывает параметр **uType**. Параметр функции **MessageBeep** может задаваться двумя способами: в виде шестнадцатеричного числа или поименованной константы. Например, системный звук по умолчанию задается константой **MB_OK**,

а стандартный системный звук задается шестнадцатеричным числом `$FFFFFFFF`. Функция возвращает параметр типа `Boolean`, который в случае успешного выполнения (воспроизведения звука) равен `True`.

5.2. Использование компонента MediaPlayer

Мультимедийный проигрыватель `MediaPlayer` является многофункциональным управляющим элементом. Он представляет программисту набор свойств и методов, позволяющих манипулировать файлами и устройствами мультимедиа, поддерживать воспроизведение и перемещение между остальными фонограммами (дорожками, записями), а также идентифицировать подключенные устройства.

Компонент `MediaPlayer` содержит следующие кнопки (рис. 5.1, слева направо).

- Play — воспроизведение.
- Pause — пауза.
- Stop — остановка.
- Next — переход к следующей фонограмме (дорожке). Для случая одной фонограммы выполняется переход в ее конец.
- Prev — переход к предыдущей фонограмме. Для случая одной фонограммы выполняется переход в ее начало.
- Step — переход на несколько кадров вперед.
- Back — возврат на несколько кадров назад.
- Record — включение режима записи.
- Eject — извлечение носителя.

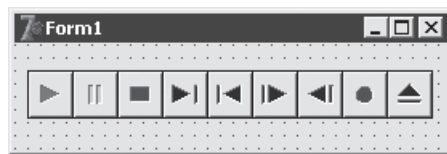


Рис. 5.1. Вид компонента `MediaPlayer`

Компонент `MediaPlayer` обладает рядом свойств, которые позволяют управлять воспроизведением файлов.

- `AutoOpen` — определяет, должно ли устройство автоматически открываться сразу после загрузки.
- `AutoRewind` — если равно `True`, то после завершения воспроизведения файла будет переход на его начало.
- `DeviceType` — определяет тип устройства, которым должен управлять объект `MediaPlayer`. Принимает одно из следующих значений:
 - `dtAVIVideo` — файл AVI;
 - `dtCDAudio` — аудио компакт-диски;

- `dtDAT` — цифровой кассетный аудиопроигрыватель;
 - `dtDigitalVideo` — цифровое видео (AVI, MPG, MOV-файлы или ММ-фильм);
 - `dtMMMovie` — формат multimedia movie;
 - `dtOther` — неопределенный формат;
 - `dtSequencer` — MIDI-файл;
 - `dtVCR` — видеомэгнитофон;
 - `dtVideodisc` — проигрыватель видеодисков;
 - `dtWaveAudio` — звуковой файл типа WAV;
 - `dtAutoSelect` — компонент выбирает устройство автоматически, устанавливается по умолчанию.
- `Display` — задает оконный элемент, в котором будет происходить воспроизведение видеоданных. Если свойство не задано, то будет открываться новое дополнительное окно.
- `DisplayRec` — задает прямоугольную область для воспроизведения данных.
- `EnableButtons` — определяет набор командных кнопок, которые можно использовать в компоненте.
- `StartPos` — определяет начальную позицию для воспроизводимых данных. Если не задано, то воспроизведение идет сначала.
- `EndPos` — определяет конечную позицию для воспроизведения данных. Если не задано, то воспроизведение идет до конца.
- `Position` — текущая позиция при воспроизведении.
- `Tracks` — определяет количество дорожек для компакт-дисков.
- `Frames` — определяет число кадров, на которое перемещается позиция устройства при вызове методов `Back` и `Next`.
- `Length` — длина файла (носителя).
- `TimeFormat` — устанавливает временной формат, используемый конкретным устройством.
- `Wait` — определяет, будет ли управление возвращено вызывающему приложению немедленно или после завершения воспроизведения.

Одним из важных свойств является `Capabilities` типа `TMPDevCapsSet`, которое позволяет определить возможности выбранного и открытого устройства. Это свойство может принимать следующие значения, устанавливающие доступность соответствующих операций:

- `mpCanEject` — извлечение носителя;
- `mpCanPlay` — воспроизведение;
- `mpCanRecord` — запись на носитель;
- `mpCanStep` — перемотка вперед или назад определенного количества кадров;
- `mpUsedWindow` — использование окна для вывода изображения.

Перед использованием устройства его нужно открыть, поскольку большинство методов, например `Play` и `StartRecording`, можно вызывать только после открытия устройства. Оно выполняется путем вызова метода `Open` (листинг 5.2). Если необходимо выполнить автоматическое открытие устройства, то свойству `AutoOpen` типа `Boolean` следует присвоить значение `True` (по умолчанию присвоено значение `False`). После открытия какого-либо устройства свойство `DeviceID` типа `Word` проигрывателя определяет идентификатор этого устройства. Если открытых устройств нет, то значение свойства `DeviceID` равно 0.

Листинг 5.2. Открытие проигрывателя компакт-дисков

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    //Задаем устройство воспроизведения
    MyMediaPlayer.DeviceType := dtCDAudio;
    //Открываем устройство
    MyMediaPlayer.Open;
end;
```

После завершения использования мультимедийного устройства его нужно закрыть, вызвав метод `Close`.

После открытия устройства с помощью свойства `Tracks` типа `Longint` можно получить информацию о количестве фонограмм (дорожек). Если устройство не поддерживает дорожки, то значение этого свойства неопределенно. Свойство `TrackLength` [`TrackNum: Integer`] типа `Longint` содержит длину фонограммы с индексом `TrackNum` (отсчет начинается с единицы). Длина дорожки указывается в формате времени, который определен свойством `TimeFormat`.

Свойство `TimeFormat` типа `TMPTimeFormats` задает формат значений свойств, которые связаны со временем. Оно влияет на способ интерпретации и отображение значений таких свойств, как `TrackLength`, `Length`, `StartPos`, `EndPos` и `Position`. Основными значениями свойства `TimeFormat` являются следующие.

- `tfMilliseconds` — целое четырехбайтовое число, счетчик миллисекунд.
- `tfHMS` — количество часов, минут и секунд, размещенных побайтно, начиная с младшего байта, в четырехбайтовом целом. Старший байт не учитывается.
- `tfMSF` — количество минут, секунд и кадров, размещенных побайтно, начиная с младшего байта, в четырехбайтовом целом. Старший байт не учитывается.
- `tfFrames` — целое четырехбайтовое число, содержащее количество кадров.

Теперь, когда мы ознакомились с основными свойствами мультимедиа-компонента `MediaPlayer`, можем приступить к непосредственному применению его на практике. Приведем пример исходного текста программы, при загрузке которой проигрывается звук (в формате WAV) (листинг 5.3).

Листинг 5.3. Воспроизведение звука при создании формы приложения

```
//Функция вызывается при создании формы
procedure TForm1.FormCreate(Sender: TObject);
begin
    //Скрываем компонент
    MyMediaPlayer.Visible := false;
    //Автоматически определяем устройство воспроизведения
    MyMediaPlayer.DeviceType := dtAutoSelect;
    //Загружаем файл воспроизведения
    MyMediaPlayer.FileName := 'start.wav';
    //Открываем устройство
    if not MyMediaPlayer.AutoOpen then MyMediaPlayer.Open;
    //Воспроизводим файл
    MyMediaPlayer.Play;
end;
```

При создании формы `Form1` воспроизводится звуковой файл `start.wav`.

В некоторых случаях удобно хранить данные (например, звуковые записи) и использовать их прямо в запускаемом модуле (EXE-файле). Такой метод предусматривает хранение звука в файлах ресурсов (RES). На этапе сборки программы файлы ресурсов прикрепляются к запускаемому модулю, тем самым увеличивая размер модуля. Но количество файлов, необходимых для корректной работы программы, уменьшается. Так, в предыдущем случае для нормальной работы программы (воспроизведение звука при загрузке) необходим файл `start.wav`. Следующий пример демонстрирует создание приложения, запускаемый модуль которого будет содержать все необходимые ресурсы, в нашем случае это звуковой файл.

Вначале необходимо создать файл ресурса, содержащий звуковую запись. Для этого понадобится компилятор ресурсов, который находится в каталоге `Borland\Delphi7\Bin\` и носит имя `brcc32.exe`. Далее создаем файл ресурса. Все ресурсы (значки, указатели, изображения, таблицы строк и т. п.), используемые приложением, описываются в специальном файле. Такое описание имеет фиксированный формат:

<имя> <тип> <параметры> <имя файла>

Имя — это уникальное имя ресурса, которое будет использоваться в процедурах работы с ресурсами. Имя файла — строка, содержащая путь к файлу. В нашем случае строка, описывающая ресурс:

```
LOADSOUND RCDATA LOADONCALL start.wav
```

Далее в командной строке записываем `brcc32.exe source.rc`, где `source.rc` — текстовый файл, содержащий описание ресурса.

После компиляции получаем готовый файл ресурса `source.RES`. Перемещаем его в каталог проекта. На этом этапе ресурс может использоваться.

Чтобы подключить файл ресурса, пишем в исходном тексте:

```
//Подключение ресурса
{$R SOURCE.RES}
```

Теперь, когда файл ресурса подключен и готов к использованию, необходимо создать функцию, которая будет доставать звуковой файл и воспроизводить его. Тело функции, выполняющей эти действия, выглядит следующим образом (листинг 5.4).

Листинг 5.4. Использование ресурсов для хранения звуковых записей

```
//Функция, которая воспроизводит звук, находящийся в ресурсе
procedure RetrieveLoadSound;
var
    hResource : THandle;
    pData      : Pointer;
begin
    //Загружаем файл ресурса и находим звук под именем 'LOADSOUND'
    hResource := LoadResource( hInstance, FindResource(hInstance,
'LOADSOUND', RT_RCDATA));
    try
        //Находим адрес загруженного ресурса
        pData := LockResource(hResource);
        if pData = nil then raise Exception.Create('Ошибка чтения
ресурса LOADSOUND');
        //Воспроизводим звуковой файл
        sndPlaySound(pData, SND_MEMORY);
    finally
        //Освобождаем ресурс
        FreeResource(hResource);
    end;
end;
```

Для работы функции `RetrieveLoadSound` понадобятся две следующие переменные: `hResource` (дескриптор ресурса) и `pData` (указатель на память, расположение ресурса). Перед использованием ресурса производится его загрузка (функция `LoadResource`). Но чтобы загрузить именно тот ресурс, который нам необходим (звук `LOADSOUND`), с помощью функции `FindResource` ищем его в ресурсах, подключенных к этому экземпляру приложения (`hInstance`). Далее получаем указатель на память, в которой находится звуковой файл, и записываем его в переменную `pData`. Если ресурс не найден, то программа выдаст сообщение об ошибке.

После того как был получен указатель на память, его можно использовать в функции `sndPlaySound` для воспроизведения звука. Параметр `SND_MEMORY` говорит о том, что воспроизведение будет осуществляться из памяти.

Функция `RetrieveLoadSound` может использоваться в любом месте программы для воспроизведения `start.wav`. В этом случае данные звукового файла будут находиться в запускаемом модуле, увеличивая его объем, но сокращая количество файлов приложения. Такой подход эффективен при создании небольших приложений, которые снабжаются короткими звуковыми сопровождениями.

В конце главы будет подробно описан процесс создания универсального проигрывателя, работа которого целиком построена на использовании компонента `MediaPlayer`. Далее рассмотрим следующий мультимедийный компонент Delphi — `Animate`, который позволяет воспроизводить как стандартную (встроенную в Windows), так и пользовательскую анимацию.

5.3. Компонент Animate

Видеоклип представляет собой файл в формате AVI, содержащий последовательность отдельных кадров, при отображении которых создается эффект движения. Наряду с изображением AVI-файлы могут содержать звук. Для воспроизведения видеоклипов можно использовать любой из компонентов — `Animate` или `MediaPlayer`.

Компонент `Animate` позволяет проигрывать AVI-файлы, а также отображать стандартную анимацию, используемую в Windows. AVI-файлы, воспроизводимые компонентом `Animate`, имеют следующие ограничения:

- они не должны содержать звука;
- информация в них не должна быть сжатой;
- размер файла не должен превышать 64 Кбайт.

Для задания воспроизводимого видеоклипа используются свойства `FileName` и `CommonAVI`. В один момент можно использовать только одно из этих свойств. Проигрываемый AVI-файл, существующий на диске, указывается путем задания свойства `FileName`, при этом свойству `CommonAVI` автоматически присваивается значение `aviNone`. Свойство `CommonAVI` позволяет выбрать один из стандартных клипов Windows и принимает следующие значения:

- `aviNone` — отсутствие стандартной анимации;
- `aviCopyFile` — копирование файла;
- `aviCopyFiles` — копирование файлов;
- `aviDeleteFile` — удаление файла;
- `aviEmptyRecycle` — очистка Корзины;
- `aviFindComputer` — поиск компьютера;
- `aviFindFile` — поиск файла;

- `aviFindFolder` — поиск папки;
- `aviRecycleFile` — перемещение файла в Корзину.

При присвоении свойству `CommonAVI` значения, отличного от `aviNone`, свойство `FileName` автоматически сбрасывается, принимая в качестве значения пустую строку.

Для задания видеоклипа также можно использовать `ResHandle` типа `THandle` и `ResID` типа `Integer`, которые составляют альтернативу свойствам `CommonAVI` и `FileName`. Значение `ResHandle` задает ссылку на модуль, в котором содержится изображение в виде ресурса, а значение свойства `ResID` в этом модуле указывает номер ресурса.

После выбора видеоклипа свойства `FrameCount`, `FrameHeight` и `FrameWidth` типа `Integer` определяют следующие параметры клипа: количество, высоту и ширину кадров (в пикселах) соответственно. Эти свойства являются свойствами времени выполнения, следовательно, доступны только для чтения.

По умолчанию размеры компонента `Animate` автоматически подстраиваются под размеры кадров видеоклипа, это определяет значение `True` свойства `AutoSize`. Если этому свойству присвоить значение `False`, то возможно отсечение части кадра изображения, если его размеры превышают размеры компонента `Animate`.

Воспроизведение видеоклипа начинается при установке свойству `Active` значения `True`. Начальный и конечный кадры задают диапазон воспроизведения и определяются соответственно значениями свойств `StartFrame` и `StopFrame` типа `SmallInt`. По умолчанию `StartFrame` указывает на первый кадр анимации, и его значение равно 1.

Свойство `Repetitions` типа `Integer` определяет количество повторений воспроизведения видеоклипа. По умолчанию его значение равно нулю. В этом случае видеоклип проигрывается до тех пор, пока процесс воспроизведения не будет остановлен.

Для запуска и остановки воспроизведения клипов можно использовать методы `Play`, `Stop` и `Reset`. Процедура `Play` (`FromFrame: Word`, `ToFrame: Word`, `Count: Integer`) проигрывает видеоклип, начиная с кадра, заданного параметром `FromFrame`, и заканчивая кадром, заданным параметром `ToFrame`. Параметр `Count` определяет количество повторений. Таким образом, эта процедура позволяет одновременно управлять `StartFrame`, `StopFrame` и `Repetitions`, задавая для них требуемые при воспроизведении значения, а также устанавливает свойству `Active` значение `True`.

Свойство `Open` типа `Boolean` доступно при выполнении программы и позволяет определить, готов ли компонент `Animate` к воспроизведению. Если выбор и загрузка видеоклипа проходят успешно, то свойству `Open` автоматически устанавливается значение `True`, компонент можно открыть и проиграть анимацию. При неуспешном завершении загрузки видеоклипа это свойство получает значение `False`. При необходимости программист может сам устанавливать свойству `Open` значение `False`, тем самым отключая компонент `Animate`.

Процедура `Stop` прерывает воспроизведение видеоклипа и устанавливает свойству `Active` значение `False`. Процедура `Reset`, кроме того, дополнительно сбрасывает свойства `StartFrame` и `StopFrame`, устанавливая значения по умолчанию.

В качестве примера, наглядно отражающего работу компонента `Animate`, рассмотрим приложение для просмотра стандартной анимации операционной системы Windows.

Стандартный видеоклип можно просмотреть, нажав кнопку `Просмотр`, предварительно выбрав анимацию в группе независимых переключателей. Клип воспроизводится непрерывное количество раз с первого до последнего кадра. Чтобы прервать воспроизведение, необходимо нажать кнопку `Стоп`. Окно приложения приведено на рис. 5.2.

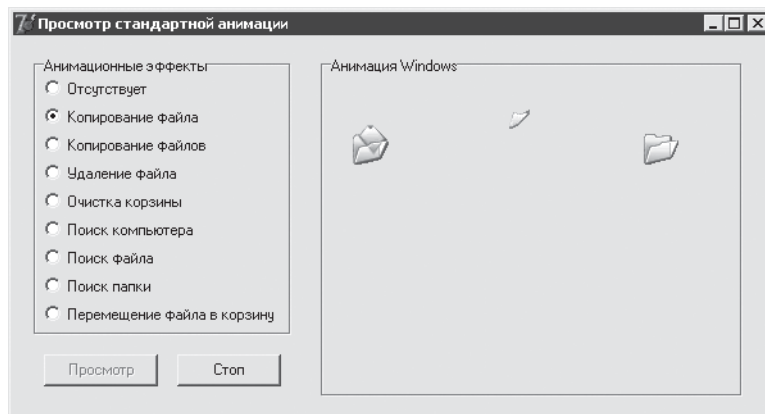


Рис. 5.2. Приложение для просмотра стандартной анимации

Рассмотрим исходный текст приложения подробно. Для работы программы необходим набор констант, значения которых может принимать свойство `CommonAVI`. Поэтому в начале программы объявляем константный массив `TypeofAVI` типа `TCommonAVI`, который и будет содержать необходимые значения:

```
const TypeofAVI: array[0..8] of TCommonAVI =
    (aviNone, aviCopyFile, aviCopyFiles,
     aviDeleteFile, aviEmptyRecycle,
     aviFindComputer, aviFindFile,
     aviFindFolder, aviRecycleFile);
```

При создании главного окна приложения устанавливаем положение переключателя в группе выбора анимации:

```
procedure TFormViewAnim.FormCreate(Sender: TObject);
begin
    //Стандартная анимация "Копирование файла"
    RadioGroupSelectAnimEffects.ItemIndex := 1;
end;
```

Создаем обработчик выбора группы независимых переключателей. При выборе анимации первым делом устанавливается доступность кнопок управления. Далее задается вид воспроизводимого ролика (например, *Копирование файлов*). В блоке `if` происходит проверка индекса выбранной анимации, и если она не выбрана (индекс равен 0), то блокируется кнопка *Просмотр*, так как в этом случае просмотр стандартной анимации невозможен (листинг 5.5).

Листинг 5.5. Обработчик выбора анимационных роликов

```
//Выбор стандартной анимации
procedure TFormViewAnim.RadioGroupSelectAnimEffectsClick(Sender:
TObject);
begin
    //Устанавливаем доступность кнопок управления
    bnStopView.Enabled := False;
    bnStartView.Enabled := True;
    //Устанавливаем значение свойства CommonAVI
    StandartAnimate.CommonAVI :=
        TypeofAVI[RadioGroupSelectAnimEffects.ItemIndex];
    //Если анимация не выбрана, делаем недоступной
    //кнопку старта показа
    if RadioGroupSelectAnimEffects.ItemIndex = 0
    then bnStartView.Enabled := False
    else bnStopView.Enabled := True;
end;
```

Значения индексов (`RadioGroupSelectAnimEffects.ItemIndex`) переключателей соответствуют порядковым номерам в массиве `TypeofAVI`, который содержит возможные значения свойства `CommonAVI`.

При нажатии кнопки начала показа происходит вызов метода `Play` компонента `Animate` и устанавливается доступность кнопок управления показом:

```
//Старт показа
procedure TFormViewAnim.bnStartViewClick(Sender: TObject);
begin
    //Начинаем показ выбранной анимации
    StandartAnimate.Play(1, StandartAnimate.FrameCount, 0);
    bnStartView.Enabled := False;
    bnStopView.Enabled := True;
end;
```

Обработчик кнопки *Стоп* основан на вызове метода `Stop` компонента `Animate` и выглядит следующим образом:

```
procedure TFormViewAnim.bnStopViewClick(Sender: TObject);  
begin  
    //Остановка показа анимации  
    StandartAnimate.Stop;  
    bnStartView.Enabled := True;  
    bnStopView.Enabled := False;  
end;
```

Зачастую компонент **Animate** используется при создании панелей инструментов для добавления в них анимационных пиктограмм, которые оживляют форму и служат для индикации того, что программа выполняет ту или иную обработку данных. Воспроизведение изображения может осуществляться, например, при нажатии кнопки в панели инструментов или по истечении заданного интервала времени.

Компонент **Animate** обеспечивает воспроизведение только простых AVI-файлов. С той же целью можно использовать компонент **MediaPlayer**, который с функциональной точки зрения значительно сложнее и обеспечивает много других мультимедийных возможностей.

5.4. Разработка звукового проигрывателя

Обладая достаточно большим багажом знаний о мультимедийных компонентах Delphi, мы вплотную подошли к созданию программы-проигрывателя. В рамках этой книги разработка многофункционального сложного проигрывателя не предусматривается, но создание легко реализуемого приложения с набором необходимых функций будет рассмотрено. Таким образом, приступим к проектированию проигрывателя. Для начала определим набор необходимых функций. В качестве базовых возможностей любого проигрывателя как видео-, так и аудиофайлов выделяют: непосредственно воспроизведение выбранного файла, возможность кратковременной остановки и возобновления воспроизведения (функция паузы), остановки, перемещение позиции воспроизведения (перемотка). Необходимыми также являются показ времени проигрывания и имя воспроизводимого файла. Как известно, компонент **MediaPlayer** поддерживает почти все эти функции, за исключением двух последних. Следовательно, **MediaPlayer** практически идеально подходит на роль основного элемента разрабатываемого проигрывателя.

Итак, создаем новый проект приложения. Соответствующим образом настраиваем свойства формы программы. Убираем кнопку максимизации, в данном случае она является лишней: устанавливаем значение **False** свойству **biMaximize**, которое находится на вкладке **BorderIcons**. Устанавливаем **BorderStyle** равным **bsSingle**. Это не позволит пользователю изменять размеры формы. Для удобства использования проигрыватель появляется в центре экрана, следовательно, свойство **Position** устанавливаем как **poScreenCenter**. Настраиваем цвета, в рассматриваемом случае **Color** равно **clInactiveCaptionText**.

Для отображения текстовой динамической информации удобным является использование компонента `Label` или меток. Время, позиция указателя воспроизведения в файле будут выводиться в специальный индикатор. Индикатор (в нашем случае `lbMainTime` типа `TLabel`) будет отображать текущее время проигрывания. Создаваемый проигрыватель должен обладать неплохим и удобным интерфейсом, поэтому настраиваем индикатор следующим образом: цвет фона `Color` устанавливаем как `clSkyBlue`, цвет и размер шрифта индикатора — `clMenuHighlight` и 28 соответственно. Другой индикатор (надпись с именем воспроизводимого файла) будет иметь свойства, установленные по умолчанию.

Управление воспроизведением будет осуществляться частично при помощи кнопок проигрывателя. Функции перемотки будут реализованы в обработчиках двух других дополнительных кнопок. Поэтому скрываем все кнопки компонента `MediaPlayer`, кроме кнопок воспроизведения, паузы и остановки. Делаем это при помощи присвоения свойству `VisibleButtons` массива значений `[btPlay, btPause, btStop]`. Кнопки управления перемоткой будут выглядеть стандартно. Нам также необходима кнопка открытия файла для выбора файла воспроизведения. Помещаем на форму стандартную кнопку и оставляем ее настройки по умолчанию.

Далее максимально эргономично размещаем на форме вышеперечисленные компоненты и можем переходить от создания дизайна к реализации функциональных возможностей. Для корректной работы индикатора времени его необходимо периодически обновлять. Для достижения этой цели нам понадобится таймер. Среда Delphi содержит компонент, который выполняет функции таймера `Timer` (вкладка `System`). На форму приложения также помещаем стандартный диалог открытия файлов. Находится этот компонент на вкладке `Dialogs`. Один из вариантов размещения компонентов интерфейса выглядит, как показано на рис. 5.3.

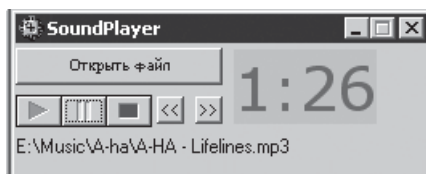


Рис. 5.3. Интерфейс проигрывателя

Начнем рассмотрение исходного текста приложения. В программе присутствует секция констант с единственной константой, необходимой для задания расстояния (положения указателя воспроизведения в файле), на которое будет осуществляться перемотка. В данном случае перемотка будет осуществляться на 10 секунд:

```
const
    //Константа для перемотки на 10 000 миллисекунд
    perem=10000;
```

Далее необходимо создать функцию, которая преобразует численные значения времени (миллисекунды) в более удобный для вывода строковый вариант с указанием минут и секунд (листинг 5.6).

Листинг 5.6. Функция преобразования времени

```
function TSoundPlayerForm.FileLangToStr(leng: longint): string;
var
    //Переменная результирующей строки
    strTime: string;
    sec: longint;
    min: longint;
begin
    //Получаем секунды и минуты из миллисекунд (leng)
    sec := trunc(leng/1000);
    min := trunc(sec/60);
    sec := sec - min*60;
    strTime := IntToStr(min);
    //Если секунд меньше десяти, то преобразуем результирующую
    //строку (участок минут), добавляя '0' спереди
    if sec < 10 then strTime := strTime + ':0' + IntToStr(sec)
    else strTime := strTime + ':' + IntToStr(sec);
    FileLangToStr := strTime;
end;
```

Находим количество секунд, затем минут, преобразуем эти данные в строковый вид (для вывода на индикатор времени). Если после нахождения количества минут секунд оказалось меньше десяти, то добавляем 0 в результирующую строку. К примеру, мы получили, что композиция занимает три минуты и пять секунд. В этом случае строка должна выглядеть как 3:05, а не 3:5.

Процедуру создания корректного формата времени мы разобрали. Теперь необходимо выяснить, как можно узнать время, которое прошло с момента начала воспроизведения файла. Для этого обратимся к свойствам компонента **MediaPlayer**, а именно к **Length** (длина загруженного файла) и **Position** (текущая позиция в нем). Зная позицию, можно при помощи ранее рассмотренной функции **FileLangToStr** найти время воспроизведения (листинг 5.7).

Листинг 5.7. Процедура вывода или обновления индикаторов

```
procedure TSoundPlayerForm.UpdateViewTime;
var
    //Длина файла и позиция в файле
    leng, posit: longint;
begin
    //Находим длину воспроизводимого файла
    leng := mdpSoundPlayer.Length;
```

```
//Находим позицию в воспроизводимом файле
posit := mdpSoundPlayer.Position;
//Преобразуем время в строку
lbMainTime.Caption := FileLangToStr(posit);
//Устанавливаем имя файла
lbFileName.Caption := mdpSoundPlayer.FileName;
end;
```

Как можно заметить из листинга 5.7, после получения позиции в файле и его имени данные о времени воспроизведения и путь к файлу попадают на индикаторы lbMainTime и lbFileName соответственно.

Открытие и загрузка файла в мультимедийный компонент происходит при выполнении кода из листинга 5.8. Кроме того, обработчик вызывает известную нам процедуру UpdateViewTime и включает таймер (tmTimer.Enabled := true).

Листинг 5.8. Открытие файла

```
procedure TSoundPlayerForm.bnOpenFileClick(Sender: TObject);
begin
if opdOpenDialog.Execute=true then
begin
//Открываем файл
mdpSoundPlayer.FileName := opdOpenDialog.FileName;
mdpSoundPlayer.Open;
//Устанавливаем значения в индикаторах
UpdateViewTime;
//Включаем таймер
tmTimer.Enabled := true;
end;
end;
```

Процедура обработки срабатывания таймера заключается в вызове функции обновления значений индикаторов (UpdateViewTime) (листинги 5.9 и 5.10).

Листинг 5.9. Событие таймера

```
procedure TSoundPlayerForm.tmTimerTimer(Sender: TObject);
begin
//Обновление значений экрана
UpdateViewTime;
end;
```

Листинг 5.10. Обработчик активизации формы

```
procedure TSoundPlayerForm.FormActivate(Sender: TObject);
begin
    //Временное выключение таймера
    tmTimer.Enabled := false;
    //Задание значений
    lbMainTime.Caption := '00:00';
    lbFileName.Caption := 'no file...';
    // Установка фильтров для диалога
    opdOpenDialog.Filter :=
        'MP3 music (*.mp3)|*.MP3|Wav files (*.wav)|*.WAV';
end;
```

Перемотка осуществляется при помощи двух кнопок. Для перемотки вперед на десять секунд необходимо нажать >>, назад — << (листинги 5.11 и 5.12).

Листинг 5.11. Перемотка вперед

```
procedure TSoundPlayerForm.bnNextStClick(Sender: TObject);
begin
    if mpCanPlay in mdpSoundPlayer.Capabilities then
        begin
            if (mdpSoundPlayer.Position+perem)<=mdpSoundPlayer.Length then
                mdpSoundPlayer.Position := mdpSoundPlayer.Position + perem
            else
                mdpSoundPlayer.Position := mdpSoundPlayer.Length;
            mdpSoundPlayer.Play;
        end;
end;
```

Листинг 5.12. Перемотка назад

```
procedure TSoundPlayerForm.bnPrevStClick(Sender: TObject);
begin
    if mpCanPlay in mdpSoundPlayer.Capabilities then
        begin
            if mdpSoundPlayer.Position>=perem then
                mdpSoundPlayer.Position := mdpSoundPlayer.Position - perem
            else
                mdpSoundPlayer.Position := 0;
```

```
mdpSoundPlayer.Play;  
end;  
end;
```

Таким образом, разработанный проигрыватель располагает набором минимальных функций и возможностей. Но он обладает важным преимуществом, а именно простотой реализации. Как вы могли заметить, созданная программа может проигрывать и MP3-файлы. Это становится возможным благодаря использованию специального программного обеспечения — кодеков, установленных в операционной системе. Современная и достаточно распространенная операционная система Windows XP содержит такие кодеки в комплекте базовой поставки. При использовании созданного проигрывателя в других операционных системах типа Windows, вероятно, понадобится самостоятельная установка кодеков.

На этом этапе принцип построения проигрывателя звуковых записей вам известен. Что касается просмотра видеозаписей, то благодаря универсальности компонента `MediaPlayer` он схож с воспроизведением звуковых файлов.

5.5. Видеопроеигрыватель

Не менее интересной задачей, рассмотренной в рамках этой главы, является разработка проигрывателя видеофайлов. Форматов видео присутствует достаточно большое количество, но самым распространенным из них, несомненно, является AVI. Учитывая этот факт, разработаем проигрыватель видеофайлов в AVI-формате.

Учитывая то, что среда Delphi предоставляет высокоуровневый доступ к мультимедийным возможностям компьютера, сам принцип построения проигрывателя не меняется. Как и в случае со звуковым проигрывателем, будет использоваться знакомый вам ранее компонент `MediaPlayer`. Особенностью воспроизведения видео является только вывод изображения на экран в дополнение к звуковому сопровождению. Таким образом, необходимо определить, какие именно компоненты могут служить в качестве контейнеров для воспроизведения в них видеопотока.

Приступим к созданию проигрывателя видео (рис. 5.4). Как и в случае звукового проигрывателя, нам понадобятся: компонент `MediaPlayer`, диалог для открытия файлов `OpenDialog`, компонент-контейнер для вывода изображения (используем `GroupBox`). Настраиваем форму приложения. Убираем кнопку максимизации, в данном случае она является лишней: присваиваем свойству `biMaximize`, которое находится на вкладке `BorderIcons`, значение `False`. Устанавливаем `BorderStyle` равным `bsSingle`. Это не позволит пользователю изменять размеры формы. Для удобства использования проигрыватель появляется в центре экрана, следовательно, свойство `Position` устанавливаем как `poScreenCenter`. В компоненте `MediaPlayer` оставляем видимыми только кнопки начала, паузы и остановки воспроизведения (аналогичным образом, как в проигрывателе звука). Помещаем на форму компонент `GroupBox`, свойство `Caption` устанавливаем пустой строкой, так как именно в этот компонент будет выводиться изображение.



Рис. 5.4. Вид видеопроеигрывателя

Рассмотрим некоторые особенности созданного видеопроеигрывателя. В качестве элемента-контейнера для динамического изображения использовался компонент `GroupBox`, поэтому его необходимо было назначить элементом вывода видео для `MediaPlayer`. Этот процесс сводится к присваиванию свойству `Display` компонента `MediaPlayer` экземпляра компонента `GroupBox`. Происходит это во время активизации формы (листинг 5.13).

Листинг 5.13. Назначение элемента вывода изображения

```
procedure TFormVideoPlayer.FormActivate(Sender: TObject);
begin
  //Устанавливаем область воспроизведения
  mpVideoPlayer.Display := gbViewVideo;
end;
```

В предложенном фрагменте текста программы переменная `gbViewVideo` является экземпляром компонента `GroupBox`.

В качестве доказательства простоты, удобства и гибкости использования компонента `MediaPlayer` приведем весь исходный текст приложения (листинг 5.14).

Листинг 5.14. Видеопроеигрыватель

```
unit video_player;

interface

uses
  //Подключаемые модули
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, MPlayer;
```

Type

```
//Главная форма приложения
TFormVideoPlayer = class(TForm)
    mpVideoPlayer: TMediaPlayer;
    FileOpenDlg: TOpenDialog;
    gbViewVideo: TGroupBox;
    bnOpenFile: TButton;
    //Нажатие кнопки открытия файла
    procedure bnOpenFileClick(Sender: TObject);
    //Активизация формы
    procedure FormActivate(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;
```

var

```
FormVideoPlayer: TFormVideoPlayer;
```

implementation

```
{ $R *.dfm }
```

```
procedure TFormVideoPlayer.bnOpenFileClick(Sender: TObject);
begin
    if FileOpenDlg.Execute = True then
    begin
        //Загружаем файл, выбранный в диалоге открытия файла
        mpVideoPlayer.FileName := FileOpenDlg.FileName;
        //Активизируем проигрыватель
        mpVideoPlayer.Open;
    end;
end;
```

```
procedure TFormVideoPlayer.FormActivate(Sender: TObject);
begin
```

```
//Устанавливаем область воспроизведения  
mpVideoPlayer.Display := gbViewVideo;  
end;  
  
end.
```

Из предложенного фрагмента видно, что, обладая минимальным объемом исходного текста, видеопроеигрыватель может выполнять все необходимые базовые функции.



Глава 6

Использование Windows GDI

- ☐ Графические объекты
- ☐ Аппаратно-независимый графический вывод
- ☐ Контекст устройства
- ☐ Графические режимы
- ☐ Работа со шрифтами
- ☐ Рисование примитивов
- ☐ Работа с текстом
- ☐ Работа с растровыми изображениями
- ☐ Альфа-смешивание

Операционная система Windows с самого начала создавалась прежде всего как графическая оболочка. И как следствие, в ней осуществляется графическое представление информации. Вполне естественным является то, что почти любое приложение использует экран для отображения данных, с которыми оно работает. По крайней мере, сама операционная система отображает на экране визуальные элементы приложений. Windows обеспечивает универсальность представления информации как на экране, так и на других устройствах вывода, например, на принтере. Стоит отметить, что для этого используются одни и те же примитивы отображения. Система самостоятельно определяет целевое устройство и активизирует соответствующий ему модуль. ОС Windows является многозадачной и предъявляет к приложениям ряд основных требований, исключающих конфликты при использовании функций вывода. Однако это вовсе не означает, что Windows обеспечивает приложения только набором функций вывода на экран или печать — система полностью управляет всем выводом.

Наверное, более правильно будет сказать, что приложения используют в качестве первичного вывода окно, а не непосредственно экран. Каждое устройство вывода в Windows характеризуется набором текущих параметров, с использованием которых происходит собственно вывод. Причем в каждый конкретный момент времени только одному приложению соответствует некоторое устройство вывода, что исключает одновременный доступ к последнему, изменение параметров одним приложением перед началом процесса вывода другим.

6.1. Графические объекты

Для управления выводом операционная система Windows предоставляет приложению набор графических объектов.

- Битовые массивы (bitmaps) — прямоугольные массивы точек, формирующие растровые изображения.
- Карандаши (pens) — используются для задания таких параметров рисования линий, как толщина, цвет и стиль (сплошная, прерывистая и т. п.).
- Кисти (brushes) — применяются для задания таких параметров заливки замкнутых контуров, как цвет и стиль.
- Шрифты (fonts) — позволяют задавать параметры вывода текста, включая имя шрифта, размер символов и т. д.
- Регионы (regions) — задают области окна, которые могут быть ограничены прямоугольником, многоугольником, эллипсом или их произвольной комбинацией, для выполнения операций заполнения, заливки, инверсии и т. д. Помимо этого, служат для определения местоположения указателя.
- Логические палитры (logical palettes) — осуществляют интерфейс между приложением и таким цветным устройством вывода, как дисплей, содержат список цветов, необходимых приложению.
- Контурные (paths) — используются для заполнения или выделения контура различных фигур.

6.2. Аппаратно-независимый графический вывод

Одна из главных особенностей Windows API — независимость графического вывода от устройства. Программное обеспечение, которое поддерживает независимость, содержится в двух динамически компокуемых библиотеках. Первая — `gdi.dll` — обеспечивает общий графический интерфейс устройства (Graphics Device Interface, GDI), а вторая является драйвером конкретного используемого устройства. В результате приложение использует тот интерфейс, который предоставляется первой библиотекой. Перед тем как произвести какую-либо операцию вывода на некоторое устройство, приложению необходимо запросить GDI о загрузке соответствующего драйвера (обычно это осуществляется автоматически и не требует дополнительных действий со стороны программиста). После загрузки соответствующего драйвера приложение может настроить ряд таких параметров вывода, как цвет линии и ее ширина, тип кисти и ее цветшрифт, область отсечения и т. д. Операционная система Windows обеспечивает хранение всех этих и других данных в специальной структуре, называемой контекстом устройства.

Стоит заметить, что GDI реализует интерфейс для рисования двухмерной графики. Это самый медленный способ отображения графики из существующих, однако самый простой для понимания основ. Используется он в основном для создания простых эффектов с минимальными усилиями.

6.3. Контекст устройства

Контекст устройства — структура, определяющая набор графических объектов и связанных с ними атрибутов и графических режимов, которые воздействуют на вывод. Графические объекты включают карандаши для рисования линий, кисти для закрашивания и заполнения, битовые образы для копирования или прокрутки части экрана, цветовые палитры для определения набора доступных цветов, области для отсечения и других операций, а также контуры для операций рисования и закрашивания.

Приложение не имеет прямого доступа к контексту устройства, и настройка параметров осуществляется посредством вызова соответствующих функций Win32 API.

Существуют четыре типа контекстов устройств:

- **экранный** — поддерживает операции рисования непосредственно на экране;
- **принтера** — поддерживает операции рисования непосредственно на принтере или плоттере;
- **памяти** — поддерживает операции рисования непосредственно в битовых масках;
- **информационный** — поддерживает получение данных об устройстве.

Приложение может осуществлять следующие операции над контекстом устройства:

- перечисление существующих графических объектов;
- выбор новых графических объектов;
- удаление существующих графических объектов;
- сохранение графических объектов, их атрибутов и параметров графических режимов;
- восстановление графических объектов, их атрибутов и параметров графических режимов.

Помимо всего прочего, приложение может использовать контекст устройства для определения процесса графического вывода, прерывания длительных графических операций, начатых другим потоком многопоточного приложения, а также может инициализировать принтер.

Экранный контекст устройства

Приложение получает контекст устройства экрана посредством вызова функций `BeginPaint`, `GetDC` или `GetDCEX`. Полученный контекст устройства идентифицирует окно, в которое будет непосредственно осуществляться вывод. Как правило, приложение получает контекст устройства экрана непосредственно перед тем, когда ему необходимо рисовать в клиентской области. Когда приложение завершает вывод, то оно обязано освободить контекст устройства, вызвав одну из соответствующих функций: `EndPaint` или `ReleaseDC`.

Win32 API позволяет получать три типа контекста устройства экрана: контекст класса, общий и частный контексты. Контекст класса и частные контексты устройства используются в приложениях, выполняющих многочисленные операции вывода. Например, программы автоматизированного проектирования, настольные издательские системы, то есть такие приложения, которые самостоятельно и постоянно осуществляют вывод, и соответственно время, затрачиваемое на эти операции, критично с точки зрения производительности. Общие контексты устройства используются в приложениях, выполняющих операции вывода лишь время от времени.

Контекст класса поддерживается только для совместимости с предыдущими версиями Windows. При создании Win32-приложения вместо контекстов класса следует использовать частные контексты.

Общий контекст устройства — контекст устройства экрана, который обрабатывается в специальном кэше системы. Такие контексты устройства используются в приложениях, осуществляющих операции вывода не очень часто. Перед тем как система возвращает описатель контекста устройства, она предварительно инициализирует общие контексты устройства значениями по умолчанию, которые можно менять по мере необходимости при помощи специальных функций. Любая операция вывода, выполняемая приложением, будет использовать значения по умолчанию до тех пор, пока не будет вызвана одна из функций GDI для выбора нового графического объекта, изменения атрибутов существующего объекта или

выбора нового режима. Поскольку может быть создано лишь определенное количество общих контекстов устройства, то приложение обязано освободить его после того, как осуществит операции вывода. Когда приложение освобождает общий контекст устройства, все произведенные в данных изменения по умолчанию будут отменены. В результате параметры необходимо устанавливать каждый раз заново.

Частный контекст устройства, в отличие от общего, сохраняет любые изменения для заданных по умолчанию данных. Этот контекст устройства не является частью системного кэша и поэтому не должен освобождаться. Система автоматически освободит его только после того, как последнее окно будет разрушено. Приложение создает частный контекст устройства (указав предварительно `CS_OWND` стиль окна) при заполнении структуры, описывающей класс окна, регистрируемого функцией `RegisterClass`. После создания окна с указанным стилем приложение может вызвать одну из функций (`GetDC`, `GetDCEX` или `BeginPaint`) для получения описателя, идентифицирующего частный контекст устройства. Приложение может использовать его до тех пор, пока не будет разрушено окно, созданное с этим классом. Любые изменения графических объектов и их атрибутов или графических режимов сохраняются системой, пока окно не удалено.

Контекст устройства принтера

Контекст устройства принтера может использоваться одинаково как для матричного, струйного и лазерного принтера, так и для плоттера. Приложение создает данный контекст устройства посредством вызова функции `CreateDC`. При этом задаются такие необходимые параметры, как имя драйвера принтера, имя принтера, файла или имени устройства для физической среды вывода и других параметров инициализации. Когда приложение завершает операцию печати, то требуется вызвать функцию `DeleteDC` для удаления созданного контекста. Заметьте, что созданный контекст устройства принтера должен быть удален посредством именно этой функции. Освобождение с помощью функции `ReleaseDC` невозможно.

Точно так же, как приложению требуется контекст устройства экрана прежде, чем оно сможет осуществлять операции вывода в клиентскую область окна, нужен контекст устройства принтера прежде, чем можно будет осуществлять операции вывода на принтер. Контекст устройства принтера, подобно контексту устройства экрана, содержит информацию о графических объектах и их атрибутах, а также о графических режимах, которые воздействуют на операции вывода. Графические объекты включают карандаш (для рисования линий), кисть (для заливки) и шрифт (для вывода текста).

В отличие от контекста устройства экрана, контексты устройства принтера не связаны с компонентом управления окна Win32 API и не могут быть получены посредством вызова функции `GetDC`. Вместо этого приложение обязано вызвать одну из функций: `CreateDC` или `PrintDlgEx`.

Если вы вызываете функцию `CreateDC`, то обязаны указать драйвер принтера и порт. Для получения этих данных можно воспользоваться одной из функций: `GetPrinter` или `EnumPrinters`.

Контекст устройства памяти

Чтобы дать возможность приложениям осуществлять операции вывода в память вместо работы с фактическим устройством, используется специальный контекст устройства для растровых операций, который называется контекстом устройства памяти. Возможность осуществлять операции вывода в памяти может понадобиться для улучшения характеристик вывода изображений. Контекст устройства памяти дает возможность системе обработать часть памяти как виртуальное устройство. Это массив битов в памяти, которые приложение может временно использовать для сохранения данных растрового изображения, созданного на нормальной поверхности для рисования. Поскольку точечный рисунок совместим с устройством, то иногда контекст устройства памяти упоминается как совместимый.

Контекст устройства памяти сохраняет растровые изображения для специфического устройства. Приложение может создать данный контекст посредством вызова функции `CreateCompatibleDC`. Результатом ее выполнения является растровое изображение, имеющее цветной формат, совместимый с форматом первоначального устройства.

Первоначально изображение в контексте устройства памяти имеет размер 1×1 пиксел. Прежде чем приложение сможет начать работать с изображением, оно должно установить битовый массив с соответствующей шириной и высотой в контекст устройства, вызывая функцию `SelectObject`.

Когда приложение передает описатель, который получен при помощи функции `CreateCompatibleDC` (одной из функций рисования), то запрашиваемая операция вывода не осуществляется на поверхности рисунка устройства. Вместо этого система сохраняет цветовую информацию для результирующей линии, кривой, текста или региона в битовом массиве. Приложение может копировать изображение, хранящееся в памяти, обратно на поверхность рисунка посредством вызова функции `BitBlt`, указывая в качестве источника контекст устройства памяти, а в качестве приемника — контекст устройства окна или экрана.

Информационный контекст устройства

Win32 API поддерживает информационный контекст устройства, используемый, чтобы восстановить или получить заданные по умолчанию параметры устройства. Для создания информационного контекста приложение должно вызвать функцию `CreateIC`. Для получения информации об объектах, заданных по умолчанию для интересующего устройства, используются функции `GetCurrentObject` и `GetObject`. Использование информационного контекста устройства более эффективно, чем контекстов других типов, потому как Win32 API работает с информационным контекстом на более низком уровне и не создает структур, необходимых для их работы. После завершения работы приложения с информационным контекстом устройства необходимо вызвать функцию `DeleteDC` для удаления созданного контекста.

6.4. Графические режимы

Операционная система Windows поддерживает пять различных графических режимов, которые позволяют приложениям определять тип смешивания цветов, место и параметры вывода и т. д.:

- настройки фона — определяет, как происходит смешивание цветов фона текстовых объектов и растровых изображений с цветом фона поля вывода;
- отображения — определяет, как происходит смешивание цвета карандашей, кистей, текстовых объектов и растровых изображений с цветом фона;
- масштабирования — определяет преобразование логических координат при графическом выводе в окна, на экран или принтер;
- заполнение контуров — определяет, каким образом будут применяться шаблоны кисти при заполнении контуров;
- сжатия — определяет, каким образом происходит преобразование цветов растровых изображений при их увеличении (уменьшении).

6.5. Работа со шрифтами

Приложение может использовать четыре различных вида технологий шрифта для отображения и печати текста:

- растровые;
- векторные;
- TrueType;
- OpenType.

Отличие между данными видами шрифтов заключается в способе хранения параметров начертания символов в специальных шрифтовых файлах. В случае растровых шрифтов каждый символ хранится в виде растра (битового массива). Векторные шрифты хранят для каждого символа относительные координаты концов отрезков, из которых состоит соответствующий символ. Шрифты TrueType и OpenType содержат информацию о линиях и командах изгиба, а также настроечную информацию для точного отображения символа, которая используется при уменьшении и увеличении масштаба отображения. Шрифты OpenType эквивалентны шрифтам TrueType, за исключением того, что они позволяют определять дополнительную информацию о символах.

Поскольку точечные рисунки для каждого символа в растровом шрифте предназначены для определенной разрешающей способности устройства, то, следовательно, качество их отображения зависит от устройства вывода. Напротив, векторные шрифты не зависят от устройства вывода, однако время, необходимое для их отображения, больше, чем у растровых или шрифтов TrueType. Последние обеспечивают приемлемую скорость вывода и могут быть промасштабированы с сохранением изначального вида символов.

Операционная система Windows предоставляет разработчикам широкий набор функций для использования шрифтового оформления своих приложений, начиная с того, что каждый контекст устройства имеет шрифт по умолчанию, и заканчивая предоставлением системного диалога для выбора шрифтов, который можно использовать в приложении.

6.6. Рисование примитивов

Теперь вы знаете хотя бы минимум теории, поэтому пора начинать практиковаться. Создадим простое приложение, которое будет рисовать на форме ряд примитивов. Для этого в новом приложении для формы сделаем обработку события `OnPaint` (листинг 6.1).

Листинг 6.1. Обработчик события формы `OnPaint`

```
procedure TfmShapes.FormPaint(Sender: TObject);
var
  hCurDC: HDC;
  hCurPen, hOldPen: HPEN;
  hCurBrush, hOldBrush: HBRUSH;
begin
  //получаем общий контекст устройства
  hCurDC := GetDC(Handle);
  //создаем графический объект Карандаш
  hCurPen := CreatePen(PS_SOLID, 2, RGB(255, 64, 0));
  //выбираем его для общего контекста устройства экрана
  //и запоминаем ранее выбранный
  hOldPen := SelectObject(hCurDC, hCurPen);
  //создаем графический объект Кисть
  hCurBrush := CreateSolidBrush(RGB(0, 128, 255));
  //выбираем ее для общего контекста устройства экрана
  //и запоминаем ранее выбранную
  hOldBrush := SelectObject(hCurDC, hCurBrush);
  //рисуем эллипс
  Ellipse(hCurDC, 10, 10, 100, 70);
  //рисуем прямоугольник
  Rectangle(hCurDC, 110, 10, 210, 70);
  //прямоугольник с округленными углами
  RoundRect(hCurDC, 10, 80, 100, 140, 10, 10);
  //прямоугольник в виде "бочки"
  RoundRect(hCurDC, 110, 80, 210, 140, 10, 100);
```

```
//рисуем прямую
MoveToEx(hCurDC, 10, 150, nil);
LineTo(hCurDC, 100, 220);
//рисуем дугу
Arc(hCurDC, 110, 150, 210, 220, 110, 150, 210, 220);
//восстанавливаем ранее выбранную кисть
SelectObject(hCurDC, hOldBrush);
//удаляем созданную кисть
DeleteObject(hCurBrush);
//восстанавливаем ранее выбранный карандаш
SelectObject(hCurDC, hOldPen);
//удаляем созданный карандаш
DeleteObject(hCurPen);
//освобождаем общий контекст устройства
ReleaseDC(Handle, hCurDC);
end;
```

Прежде чем начать рисовать, требуется получить контекст устройства нашей формы. Для этого мы используем функцию `GetDC`:

```
hCurDC := GetDC(Handle);
```

Она получает описатель контекста устройства экрана для клиентской области указанного окна или всего экрана. Функция имеет следующий формат заголовка:

```
Function GetDC(hWnd: HWND): HDC;
```

Здесь `hWnd` — дескриптор окна, для которого получается контекст устройства. Если это значение равно `nil`, то `GetDC` возвращает контекст устройства для всего экрана. В случае успешного выполнения функция возвращает контекст устройства. В противном случае ее результат равен `nil`.

Теперь мы должны изменить атрибуты контекста устройства по умолчанию на те, которые нам необходимы. Изменим цвет карандаша и его толщину, а также цвет кисти. Для этого создадим новый графический объект при помощи функции `CreatePen`.

```
hCurPen := CreatePen(PS_SOLID, 2, RGB(255, 64, 0));
```

Формат данной функции следующий:

```
Function CreatePen(fnPenStyle: Integer; nWidth: Integer;
                  crColor: COLORREF): HPEN;
```

Параметр `fnPenStyle` задает стиль карандаша. Возможные значения этого параметра приведены в табл. 6.1.

Таблица 6.1. Стили карандаша

Значение	Интерпретация
PS_SOLID	Сплошной
PS_DASH	Штриховой. Данный стиль можно применять, только если ширина карандаша равна 1 или меньше в единицах устройства
PS_DOT	Пунктирный. Стиль можно применять, только если ширина карандаша равна 1 или меньше в единицах устройства
PS_DASHDOT	Штрихпунктирный. Данный стиль можно применять, только если ширина карандаша равна 1 или меньше в единицах устройства
PS_DASHDOTDOT	Чередование черты и двойной точки. Стиль можно применять, только если ширина карандаша равна 1 или меньше в единицах устройства
PS_NULL	Невидимый
PS_INSIDEFRAME	Сплошной. В данном случае при рисовании границы прямоугольника контур будет внутри рисуемого прямоугольника

Параметр `nWidth` задает ширину карандаша в логических единицах. Если `nWidth` равен 0, то карандаш будет шириной в один пиксел независимо от текущей трансформации.

`CreatePen` возвращает карандаш с заданной шириной со стилем `PS_SOLID`, если вы указали ширину больше, чем 1, для одного из стилей: `PS_DASH`, `PS_DOT`, `PS_DASHDOT`, `PS_DASHDOTDOT`.

Параметр `crColor` задает цвет карандаша.

Если функция завершилась удачно, то она возвращает дескриптор логического карандаша. В противном случае она возвращает `nil`.

После того как карандаш создан, следует его выбрать для полученного контекста при помощи функции `SelectObject`:

```
hOldPen := SelectObject(hCurDC, hCurPen);
```

Данная функция имеет следующий формат:

```
Function SelectObject(hdc: HDC; hgdiobj: HGDIOBJ): HGDIOBJ;
```

- `hdc` — дескриптор контекста устройства;
- `hgdiobj` — дескриптор на выбираемый объект.

Если выбранный объект не регион и функция выполнялась успешно, то она возвращает дескриптор на объект, который был заменен. Если выбранный объект — регион и функция выполнялась успешно, то возвращаемое значение может быть одним из приведенных в табл. 6.2.

Таблица 6.2. Результат SelectObject для выбранного объекта регион

Значение	Интерпретация
SIMPLEREGION	Регион содержит прямоугольник
COMPLEXREGION	Регион содержит более одного прямоугольника
NULLREGION	Регион пуст

Если происходит ошибка и выбранный объект не регион, то возвращаемое значение — `nil`. Иначе — `HGDI_ERROR`.

Функция возвращает предыдущий выбранный объект указанного типа. Приложение должно всегда восстанавливать объект по умолчанию после того, как закончилось рисование с использованием нового объекта.

Приложение не может выбрать битовый массив более чем для одного контекста устройства одновременно.

После успешного выбора созданного нами карандаша и запоминания предыдущего выбранного необходимо создать и выбрать кисть. Для этого используем функцию `CreateSolidBrush`:

```
hCurBrush := CreateSolidBrush(RGB(0, 128, 255));
```

Данная функция имеет следующий формат:

```
Function CreateSolidBrush(crColor: COLORREF): HBRUSH;
```

Параметр `crColor` задает цвет кисти.

Если функция завершилась успешно, то она возвращает дескриптор логической кисти. В противном случае — `nil`.

После создания кисти выбираем ее с использованием той же самой функции `SelectObject` и запоминаем ранее выбранную.

```
hOldBrush := SelectObject(hCurDC, hCurBrush);
```

Далее рисуем примитивы с использованием полученного контекста устройства с новыми графическими объектами.

Чтобы нарисовать эллипс, используем функцию `Ellipse`:

```
Ellipse(hCurDC, 10, 10, 100, 70);
```

Функция имеет следующий формат:

```
Function Ellipse(hdc: HDC; nLeftRect, nTopLeft, nRightRect, nBottomRect: Integer): BOOL;
```

- `hdc` — дескриптор контекста устройства;
- `nLeftRect` — задает координату x (в логических единицах) верхнего левого угла описываемого прямоугольника;

- `nTopRect` — задает координату `y` (в логических единицах) верхнего левого угла;
- `nRightRect` — задает координату `x` (в логических единицах) правого нижнего угла;
- `nBottomRect` — задает координату `y` (в логических единицах) правого нижнего угла.

Если функция завершается успешно, то ее результат — ненулевое значение. В противном случае возвращается 0.

Для рисования прямоугольника используется функция `Rectangle`.

```
Rectangle(hCurDC, 110, 10, 210, 70);
```

У данной функции такой же формат, как и у `Ellipse`, но интерпретация последних четырех параметров немного иная. Они задают сам прямоугольник, а не прямоугольник, описываемый вокруг эллипса.

Далее мы рисуем прямоугольник с округленными углами при помощи функции `RoundRect`.

```
RoundRect(hCurDC, 10, 80, 100, 140, 10, 10);
```

У данной функции первые пять параметров идентичны параметрам предыдущей функции, а последние два задают ширину и высоту эллипса, при помощи которого происходит округление углов прямоугольника.

Следующим примитивом, который мы рисуем, является отрезок. Процесс рисования осуществляется в два этапа. Сначала при помощи функции `MoveToEx` устанавливается начальная точка отрезка. Затем используем функцию `MoveTo` с указанием конечной точки.

```
MoveToEx(hCurDC, 10, 150, nil);
```

```
LineTo(hCurDC, 100, 220);
```

Четвертый параметр в функции `MoveToEx` — это переменная типа `TPoint`, в которую помещается предыдущее положение карандаша.

И последней рисуется дуга при помощи функции `Arc`.

```
Arc(hCurDC, 110, 150, 210, 220, 110, 150, 210, 220);
```

В ней первые пять параметров соответствуют параметрам функции `Rectangle`, а последние четыре параметра задают начальную и конечную радиальные точки дуги.

После того как все операции вывода выполнены, требуется освободить все занятые ресурсы системы. Это осуществляется следующим образом:

```
SelectObject(hCurDC, hOldPen);
```

```
DeleteObject(hCurPen);
```

```
SelectObject(hCurDC, hOldPen);
```

```
DeleteObject(hCurPen);  
ReleaseDC(Handle, hCurDC);
```

Сначала восстанавливаются карандаш и кисть для контекста устройства и удаляются созданные нами, а после освобождается и сам контекст устройства. Результат выполнения приложения приведен на рис. 6.1.

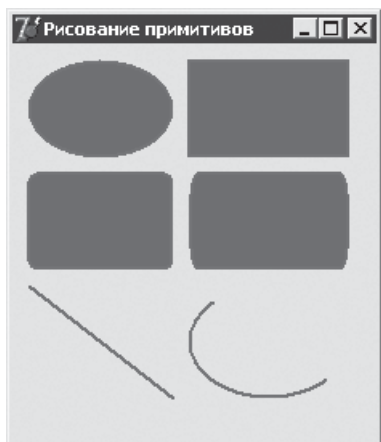


Рис. 6.1. Результат работы приложения «Рисование примитивов»

Здесь вы можете увидеть, что рисуется в итоге и как параметры функции влияют на это.

6.7. Работа с текстом

Теперь мы разработаем простое приложение, которое будет способно выводить текст под различным углом через определенный интервал времени. Для этого опять сделаем обработку события `OnPaint` нашей формы, в которой будем осуществлять вывод некоторого текста на поверхность формы. Исходный код данного обработчика приведен в листинге 6.2.

Листинг 6.2. Обработчик события формы `OnPaint`

```
procedure TfmText.FormPaint(Sender: TObject);  
var  
    hCurDC: HDC;  
    hCurFont, hOldFont: HFONT;  
    nOldMode: Integer;  
    sText: String;  
begin  
    //получаем общий контекст устройства  
    hCurDC := GetDC(Handle);
```

```
//создаем шрифт из шаблона
hCurFont := CreateFontIndirect(LogFontData);
//выбираем созданный шрифт
hOldFont := SelectObject(hCurDC, hCurFont);
//устанавливаем новый режим вывода
nOldMode := SetBkMode(hCurDC, TRANSPARENT);
//устанавливаем цвет текста
SetTextColor(hCurDC, RGB(0, 0, 255));
//задаем текстовую строку
sText := 'Текст примера';
//выводим текст на экран
TextOut(hCurDC, Width div 2, Height div 2, PAnsiChar(sText),
        Length(sText));
//восстанавливаем режим вывода
SetBkMode(hCurDC, nOldMode);
//восстанавливаем ранее выбранный шрифт
SelectObject(hCurDC, hOldFont);
//удаляем созданный шрифт
DeleteObject(hCurFont);
//освобождаем общий контекст устройства
ReleaseDC(Handle, hCurDC);
end;
```

Как можно легко заметить, обработчик события `OnPaint` работает по той же схеме, что и в предыдущем примере. Изначально получаем контекст устройства, потом создаем необходимый графический объект и выбираем его вместо установленного по умолчанию. После чего восстанавливаются все атрибуты контекста устройства, а затем он освобождается. Теперь перейдем от общего к частному. Мы создаем логический шрифт на основании указанных характеристик при помощи функции `CreateFontIndirect`.

```
hCurFont := CreateFontIndirect(LogFontData);
```

Данная функция имеет следующий формат заголовка:

```
Function CreateFontIndirect(const lf: LOGFONT): HFONT;
```

Параметр `lf` содержит описание характеристик логического шрифта. Если функция завершается успешно, то она возвращает дескриптор логического шрифта. В противном случае ее результатом является `nil`.

После создания шрифта выбираем его в контексте устройства.

```
hOldFont := SelectObject(hCurDC, hCurFont);
```

Далее устанавливаем режим прозрачности, то есть такой режим, при котором будет выводиться только текст без предварительной заливки фона определенным цветом.

```
nOldMode := SetBkMode(hCurDC, TRANSPARENT);
```

Функция `SetBkMode` служит для установки режима смешивания фона определенного контекста устройства. Этот режим используется для текста, штриховых кистей, а также для карандашей со стилем, отличным от сплошных линий.

Формат заголовка данной функции следующий:

```
Function SetBkMode(hdc: HDC; nBkMode: Integer): Integer;
```

- `hdc` — задает описатель контекста устройства, для которого устанавливается режим смешивания фона;
- `nBkMode` — определяет режим смешивания фона, может принимать одно из значений, указанных в табл. 6.3.

Таблица 6.3. Режимы смешивания фона

Значение	Интерпретация
OPAQUE	Фон заполняется текущим цветом фона прежде, чем текст, штриховая кисть или перо будут применены
TRANSPARENT	Фон остается нетронутым

Если функция завершается успешно, то она возвращает предыдущий установленный режим смешивания фона. В противном случае она возвращает ноль.

Стоит отметить, что данная функция оказывает эффект на стили линий, которые рисуются с использованием карандаша, созданного посредством функции `CreatePen`. Если карандаш создан при помощи функции `ExtCreatePen`, то никакого эффекта не будет.

Параметр `nBkMode` может быть установлен и в другие значения, отличные от указанных, которые специфичны для данного драйвера устройства. GDI передает драйверу устройства полученное специфическое значение.

Теперь необходимо установить определенный цвет текста при помощи функции `SetTextColor` для нашего контекста устройства.

```
SetTextColor(hCurDC, RGB(0, 0, 255));
```

Данная функция имеет следующий формат заголовка:

```
Function SetTextColor(hdc: HDC; crColor: COLORREF): COLORREF;
```

Первый параметр задает контекст устройства, для которого устанавливается цвет текста. Второй параметр задает сам цвет, который необходимо установить. В качестве результата функция возвращает предыдущий установленный цвет, но в случае неудачного завершения она возвращает `CLR_INVALID`.

Цвет текста используется при рисовании изображения каждого символа при помощи функций `TextOut` и `ExtTextOut`, а также для преобразования растрового изображения при конвертировании из цветного в монохромный режим.

Мы сделали все необходимые подготовки к выводу текста и теперь просто выводим его с центра нашей формы.

```
TextOut(hCurDC, Width div 2, Height div 2, PAnsiChar(sText),
Length(sText));
```

Но для нас недостаточно обработки лишь события `OnPaint`. Поэтому поместим на форму таймер и установим интервал его срабатывания равным 100. А в обработчике будем менять атрибуты текста, которые задают угол его наклона при выводе. После чего заставляем сработать обработчик события `OnPaint` нашей формы посредством вызова функции `Repaint` (листинг 6.3).

Листинг 6.3. Обработчик события таймера `OnTimer`

```
procedure TfmText.TurnTimerTimer(Sender: TObject);
begin
  with LogFontData do
    begin
      lfEscapement := lfEscapement + 60;
      lfOrientation := lfEscapement;
    end;
  Repaint;
end;
```

Переменная `LogFontData` объявлена следующим образом:

```
LogFontData: LOGFONT;
```

На основании ее мы создаем шрифт, которым выводится текст. Здесь мы изменяем только два ее поля, которые влияют на наклон текста при выводе. Все остальные параметры мы единожды заполняем при создании формы. Там же мы активизируем таймер (листинг 6.4).

Листинг 6.4. Обработчик события формы `OnCreate`

```
procedure TfmText.FormCreate(Sender: TObject);
begin
  with LogFontData do
    begin
      lfHeight      := 30; // высота шрифта
      lfWidth       := 0;  // средняя ширина символа
      lfEscapement  := 0;  // наклон строки относительно оси оХ
```

```
lfOrientation      := 0;    // наклон символа
                        // относительно оси OX

lfWeight           := FW_BOLD; // вес шрифта

lfItalic           := 0;

lfUnderline        := 0;

lfStrikeOut        := 0;

// кодовая страница по умолчанию

lfCharSet          := DEFAULT_CHARSET;

lfOutPrecision     := OUT_DEFAULT_PRECIS; // точность
                                           // вывода

lfClipPrecision    := CLIP_DEFAULT_PRECIS; // отсечение
                                           // вывода

lfQuality          := PROOF_QUALITY; // качество вывода

lfPitchAndFamily   := VARIABLE_PITCH or FF_DONTCARE;
                                           // семейство шрифта

lfFaceName         := 'Arial'; // название шрифта

end;

TurnTimer.Enabled := True;

end;
```

Результат работы приложения можно увидеть на рис. 6.2.



Рис. 6.2. Результат работы приложения «Работа с текстом»

6.8. Работа с растровыми изображениями

Вы можете использовать точечный рисунок, чтобы запомнить изображение, а потом сохранить его в памяти, отобразить в другом месте окна вашего приложения или вообще в другом окне.

В некоторых случаях вы можете захотеть, чтобы ваше приложение запоминало и хранило изображение только временно. Например, когда вам необходимо промасштабировать его в каком-нибудь приложении для рисования. Для этого необходимо временно запомнить нормальное представление изображения и показать измененное. После того как пользователь опять выберет нормальное представление изображения, приложение будет обязано заменить промасштабированное изображение копией нормального, которое временно сохранено.

Чтобы временно запомнить изображение, вашему приложению необходимо вызвать функцию `CreateCompatibleDC`, чтобы создать контекст устройства памяти, совместимый с контекстом устройства экрана текущего окна. После этого вы создаете точечный рисунок с соответствующими атрибутами посредством вызова функции `CreateCompatibleBitmap`, а затем выбираете его в контексте устройства памяти уже известным вам образом.

После того как создан совместимый контекст устройства и выбран соответствующий точечный рисунок, вы можете запоминать изображение. Функция `BitBlt` получает изображение, а также копирует данные из исходного точечного рисунка и помещает их в точечный рисунок приемника. Однако два параметра функции не являются описателями точечных рисунков. Вместо этого функция получает два описателя контекстов устройств и копирует растровые данные из точечного рисунка, выбранного в исходном контексте устройства, в точечный рисунок, выбранный в целевом контексте устройства. В этом случае целевой контекст устройства является совместимым контекстом устройства. Когда копирование растровых данных завершается, изображение помещается в память. Чтобы восстановить изображение, вызовите повторно `BitBlt`, указав теперь в качестве источника совместимый контекст устройства и в качестве приемника контекст устройства экрана (принтера и т. д.).

Следующий пример демонстрирует, как можно получать изображения всего Рабочего стола, а также как полученное изображение можно масштабировать. В данном приложении мы будем обрабатывать три события формы: `OnCreate`, `OnPaint`, `OnClose`, а также одно событие кнопки `OnClick`.

Рассмотрим исходный код обработчика события `OnCreate` (листинг 6.5).

Листинг 6.5. Обработчик события `OnCreate`

```
procedure TfmCaptureImage.FormCreate(Sender: TObject);
begin
    //создаем контекст устройства экрана
    hdcScreen := CreateDC('DISPLAY', nil, nil, nil);
```

```
//создаем совместимый контекст устройства памяти
hdcCompatible := CreateCompatibleDC(hdcScreen);
bmpWidth := GetDeviceCaps(hdcScreen, HORZRES);
bmpHeight := GetDeviceCaps(hdcScreen, VERTRES);
//создаем совместимый точечный рисунок для hdcScreen
hbmScreen := CreateCompatibleBitmap(hdcScreen, bmpWidth,
                                     bmpHeight);

if hbmScreen <> 0 then
    hOldBitmap := SelectObject(hdcCompatible, hbmScreen)
else
    hOldBitmap := 0;
    Captured := False;
end;
```

Здесь происходит создание контекста устройства Рабочего стола посредством вызова функции CreateDC.

```
hdcScreen := CreateDC('DISPLAY', nil, nil, nil);
```

После этого создается совместимый контекст устройства памяти для только что основанного контекста. Затем создается совместимый точечный рисунок.

```
bmpWidth := GetDeviceCaps(hdcScreen, HORZRES);
bmpHeight := GetDeviceCaps(hdcScreen, VERTRES);
//создаем совместимый точечный рисунок для hdcScreen
hbmScreen := CreateCompatibleBitmap(hdcScreen, bmpWidth,
                                     bmpHeight);
```

Если нам удалось создать совместимый точечный рисунок, то выбираем его в совместимом контексте устройства памяти. Еще мы вводим флаг, который указывает, сохранено ли в данный момент изображение. Все полученные данные сохраняются в полях формы, объявленных при описании ее класса.

```
hdcScreen, hdcCompatible: HDC;
hbmScreen, hOldBitmap: HBITMAP;
bmpWidth, bmpHeight: Integer;
Captured: LongBool;
```

Рассмотрим исходный код обработчика события OnPaint (листинг 6.6).

Листинг 6.6. Обработчик события OnPaint

```
procedure TfmCaptureImage.FormPaint(Sender: TObject);
var
    hCurDC: HDC;
begin
```

```
if Captured then
begin
    hCurDC := GetDC(Handle);
    StretchBlt(hCurDC, 0, 0, Width, Height, hdcCompatible,
               0, 0, bmpWidth, bmpHeight, SRCCOPY);
    ReleaseDC(Handle, hCurDC);
end;
end;
```

Проверяем, есть ли изображение, которое нам необходимо показывать. Если да, то получаем контекст устройства нашего окна и масштабируем на него полученное изображение при помощи функции `StretchBlt`.

Перед закрытием формы мы должны освободить занятые нами ресурсы системы. Поэтому мы обрабатываем событие `OnClose`, исходный код обработчика которого приведен ниже (листинг 6.7).

Листинг 6.7. Обработчик события `OnClose`

```
procedure TfmCaptureImage.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    //восстанавливаем точечный рисунок по умолчанию
    if hOldBitmap <> 0 then
        SelectObject(hdcCompatible, hOldBitmap);
    //удаляем совместимый точечный рисунок
    if hbmScreen <> 0 then
        DeleteObject(hbmScreen);
    //удаляем совместимый контекст устройства памяти
    if hdcCompatible <> 0 then
        DeleteDC(hdcCompatible);
    //удаляем контекст устройства экрана
    if hdcScreen <> 0 then
        DeleteDC(hdcScreen);
end;
```

Нам осталось рассмотреть последний обработчик события `OnClick` кнопки, помещенной на нашу форму. В нем мы прячем окно, сохраняем изображение экрана и затем показываем наше окно (листинг 6.8).

Листинг 6.8. Сохранение захваченного изображения

```
procedure TfmCaptureImage.btnCaptureClick(Sender: TObject);
var
    hdcForm: HDC;
```

```

begin
    //прячем наше окно
    Hide;
    //сохраняем текущее изображение экрана
    Captured := BitBlt(hdcCompatible, 0, 0, bmpWidth, bmpHeight,
hdcScreen, 0, 0, SRCCOPY);
    //показываем наше окно
    Show;
end;

```

В итоге мы создали довольно простое приложение, которое способно получать изображение всего Рабочего стола. Результат работы приложения приведен на рис. 6.3.

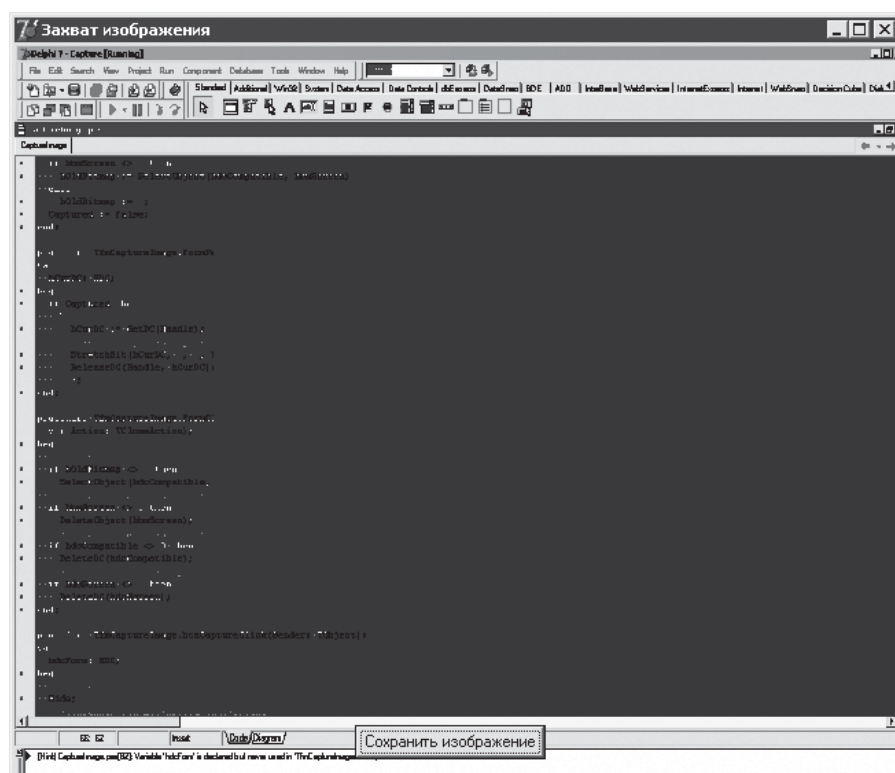


Рис. 6.3. Результат работы приложения «Захват изображения»

6.9. Альфа-смешивание

Здесь мы рассмотрим пример, иллюстрирующий, как осуществлять альфа-смешивание точечного рисунка. Мы создадим приложение, в котором окно делится

на три горизонтальные области. Затем создается точечный рисунок с альфа-смешиванием в каждой из областей окна следующим образом:

- в верхней области постоянная альфа = 50 %, но нет никакой исходной альфы;
- в средней области постоянная альфа = 100 % и исходная альфа = 0 %;
- в нижней области постоянная альфа = 75 % и исходная альфа переменная.

Добавим в описание нашей формы процедуру со следующим форматом заголовка:

```
procedure DrawAlphaBlend(hWnd: HWND; hdcwnd: HDC);
```

В самой процедуре объявим ряд переменных, которые нам понадобятся в процессе работы. Объявление приведено в листинге 6.9.

Листинг 6.9. Объявление переменных

```
var
  hCurDC: HDC;           //описатель контекста устройства,
                          //который мы создадим
  bf: BLENDFUNCTION;     //запись альфа-смешивания
  hbmp: HBITMAP;         //дескриптор точечного рисунка
  bmi: BITMAPINFO;       //заголовок точечного рисунка
  pvBits: Pointer;       //pointer to DIB section
  ulWindowWidth, ulWindowHeight: ULONG; //ширина/высота
                                      //клиентской области
  ulBitmapWidth, ulBitmapHeight: ULONG; //ширина/высота
                                      //точечного рисунка

  rt: TRect;             //используется для получения размера
                          //клиентской области
  x, y: Integer;         //циклические переменные
  ubAlpha: UCHAR;        //используется для создания
                          //прозрачного градиента
  ubRed: UCHAR;
  ubGreen: UCHAR;
  ubBlue: UCHAR;
  fAlphaFactor: Real;
  r, g, b: UCHAR;
```

В самом начале процедуры осуществляем подготовку необходимых данных для альфа-смешивания. Данные содержат информацию о требуемых размерах, а также необходимые данные точечного рисунка. Рассмотрите листинг 6.10 с необходимыми комментариями.

Листинг 6.10. Подготовка необходимых данных

```
//получаем размеры клиентской области
Windows.GetClientRect(hWnd, rt);
```

```
//рассчитываем ширину и высоту клиентской области
ulWindowWidth := rt.right - rt.left;
ulWindowHeight := rt.bottom - rt.top;
if (ulWindowWidth = 0) or (ulWindowHeight = 0) then
    Exit;
//делим окно на три горизонтальные области
ulWindowHeight := ulWindowHeight div 3;
//создаем контекст устройства для нашего точечного рисунка
hCurDC := CreateCompatibleDC(hdcwnd);
ZeroMemory(@bmi, sizeof(BITMAPINFO));
//Устанавливаем параметры точечного рисунка.
//Указываем ширину и высоту точечного рисунка для каждой
//из трех горизонтальных областей
//равными 60 % ширины и высоты главного окна.
//Смешивание в центре каждой из этих трех областей
with bmi.bmiHeader do
begin
    biSize := sizeof(BITMAPINFOHEADER);
    biWidth := ulWindowWidth - (ulWindowWidth div 5) * 2;
    ulBitmapWidth := biWidth;
    biHeight := ulWindowHeight - (ulWindowHeight div 5) * 2;
    ulBitmapHeight := biHeight;
    biPlanes := 1;
    biBitCount := 32; //четыре восьмибитных составляющих
    biCompression := BI_RGB;
    biSizeImage := ulBitmapWidth * ulBitmapHeight * 4;
end;

//создаем DIB секцию и выбираем точечный рисунок в контексте
устройства
hbmp := CreateDIBSection(hCurDC, bmi, DIB_RGB_COLORS, pvBits,
0, 0);
SelectObject(hCurDC, hbmp);
```

Далее осуществляем описанное ранее альфа-смешивание для каждой из областей. Для первой области в точечном рисунке мы устанавливаем синий цвет точки. Задаем необходимые параметры альфа-смешивания и выполняем его (листинг 6.11).

Листинг 6.11. Альфа-смешивание верхней области

```
//в верхней области постоянная альфа = 50 %,
//но исходная альфа отсутствует
```

```

//цветовой формат для каждого пиксела 0xaarrggb
//установим пиксели в синий цвет и альфу в ноль
for y := 0 to ulBitmapHeight - 1 do
  for x := 0 to ulBitmapWidth - 1 do
    PULONG(Integer(pvBits) +
      (x + y * ulBitmapWidth) * sizeof(ULONG))^ := $000000ff;

bf.BlendOp      := AC_SRC_OVER;
bf.BlendFlags   := 0;
bf.AlphaFormat  := 0; //игнорировать исходный альфа-канал
bf.SourceConstantAlpha := $7f; //половина $ff = 50 %
                        //прозрачности

if not Windows.AlphaBlend(hdcwnd, ulWindowWidth div 5,
                          ulWindowHeight div 5,
                          ulBitmapWidth, ulBitmapHeight,
                          hCurDC, 0, 0, ulBitmapWidth,
                          ulBitmapHeight, bf) then
begin
  DeleteObject(hbmp);
  DeleteDC(hCurDC);
  Exit;
end;

```

По аналогии выполняем необходимые действия со средней областью. В центре точечного рисунка прозрачность отсутствует, поэтому там будет только указанный цвет. Установим в центре красный цвет, а остальную часть сделаем синей. Далее опять задаем необходимые параметры альфа-смешивания и выполняем его (листинг 6.12).

Листинг 6.12. Альфа-смешивание средней области

```

//в средней области постоянная альфа=100 %, а исходная равна 0
for y := 0 to ulBitmapHeight - 1 do
  for x := 0 to ulBitmapWidth - 1 do
    if (x > Integer(ulBitmapWidth div 5)) and
      (x < (ulBitmapWidth - ulBitmapWidth div 5)) and
      (y > Integer(ulBitmapHeight div 5)) and
      (y < (ulBitmapHeight - ulBitmapHeight div 5)) then
      //в середине точечного рисунка альфа равна нулю,
      //это означает, что каждый цветной компонент умножается на 0.
      //Таким образом, после альфа-смешивания мы получим 0 * r,

```

```

//0x00 * g, 0x00 * b ($00000000)
//установим сейчас цвет пикселей в красный
PULONG(Integer(pvBits) +
  (x + y * ulBitmapWidth) * eof(ULONG))^ := $00ff0000
else
  //остальную часть точечного рисунка сделаем синей
  PULONG(Integer(pvBits) +
    (x + y * ulBitmapWidth) * sizeof(ULONG))^ := $000000ff;
bf.BlendOp      := AC_SRC_OVER;
bf.BlendFlags   := 0;
bf.AlphaFormat  := AC_SRC_ALPHA; //используем исходную альфа
bf.SourceConstantAlpha := $ff;  //непрозрачный

if not Windows.AlphaBlend(hdcwnd, ulWindowWidth div 5,
  ulWindowHeight div 5 + ulWindowHeight, ulBitmapWidth,
  ulBitmapHeight,
  hCurDC, 0, 0, ulBitmapWidth, ulBitmapHeight, bf) then
begin
  DeleteObject(hbmp);
  DeleteDC(hCurDC);
  Exit;
end;

```

В последней части происходит градиентное альфа-смешивание. Соответствующий код приведен в листинге 6.13.

Листинг 6.13. Альфа-смешивание нижней области

```

//нижняя область. Используем альфа = 75 % и переменную исходную альфу
//создаем градиентный эффект, используя исходную альфа
ubRed    := $00;
ubGreen  := $00;
ubBlue   := $ff;

for y := 0 to ulBitmapHeight - 1 do
  for x := 0 to ulBitmapWidth - 1 do
    begin
      ubAlpha := Trunc(x / ulBitmapWidth * 255) and $FF;
      fAlphaFactor := ubAlpha / $ff;

      r := (Round(ubRed * fAlphaFactor) * (1 shl 16)) and $FF;
      g := (Round(ubGreen * fAlphaFactor) * (1 shl 8)) and $FF;

```

```

b := Round(ubBlue * fAlphaFactor) and $FF;
PULONG(Integer(pvBits) +
          (x + y * ulBitmapWidth) * sizeof(ULONG))^ :=
  (ubAlpha shl 24) or //0xaa000000
  r or //0x00rr0000
  g or //0x0000gg00
  b; //0x000000bb
end;

bf.BlendOp      := AC_SRC_OVER;
bf.BlendFlags   := 0;
bf.AlphaFormat  := AC_SRC_ALPHA;
bf.SourceConstantAlpha := $bf;

Windows.AlphaBlend(hdcwnd, ulWindowWidth div 5,
                   ulWindowHeight div 5 + 2 * ulWindowHeight,
                   ulBitmapWidth, ulBitmapHeight, hCurDC, 0, 0,
                   ulBitmapWidth, ulBitmapHeight, bf);

DeleteObject(hbmp);
DeleteDC(hCurDC);

```

Обработчик события `OnPaint` нашей формы использует написанную функцию каждый раз, когда требуется ее обновить. Для этого он получает контекст устройства нашей формы, производит заливку фона темно-синим цветом, а после вызывает функцию альфа-смешивания трех областей. Соответствующий исходный код приведен в листинге 6.14.

Листинг 6.14. Обработчик события `OnPaint`

```

procedure TfmAlphaBlending.FormPaint(Sender: TObject);
var
  hCurDC: HDC;
  hCurBrush, hOldBrush: HBRUSH;
begin
  hCurDC := GetDC(Handle);
  hCurBrush := CreateSolidBrush(RGB(0, 0, 64));
  FillRect(hCurDC, Rect(0, 0, Width, Height), hCurBrush);
  DrawAlphaBlend(Handle, hCurDC);
  DeleteObject(hCurBrush);
  ReleaseDC(Handle, hCurDC);
end;

```

Теперь осталось только взглянуть на результат нашей работы, запустив приложение (рис. 6.4).

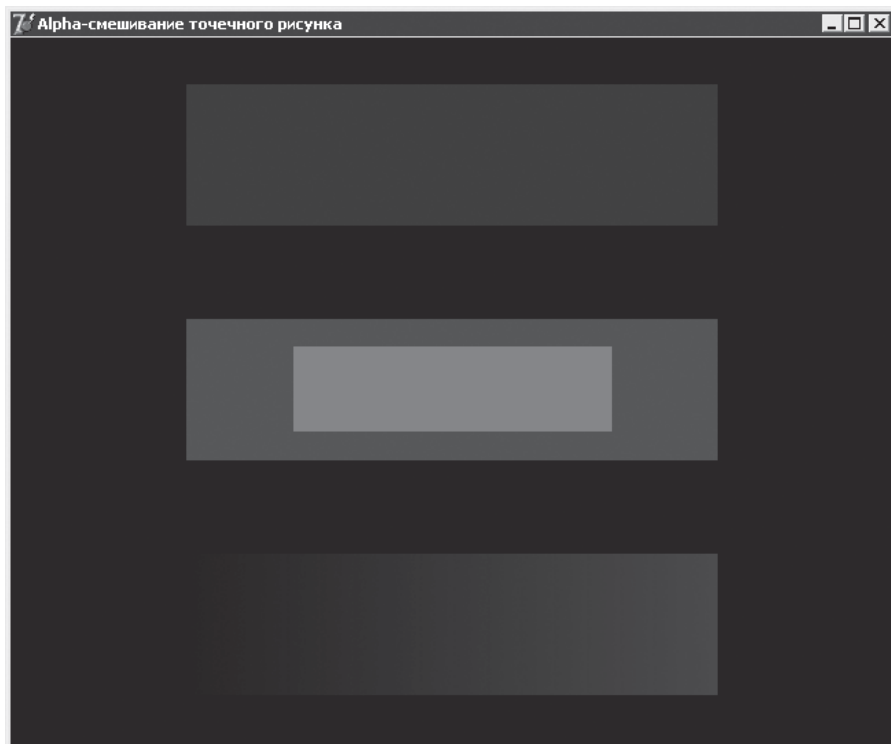


Рис. 6.4. Результат работы приложения «Alpha-смешивание точечного рисунка»

На этом закончим рассмотрение работы с графикой в Delphi.



Глава 7

Системная информация и реестр Windows

- ☐ Системная информация
- ☐ Системное время
- ☐ Реестр

Возникала ли у вас необходимость программно определить текущее состояние компьютера или узнать какие-нибудь сведения об операционной системе? Можно только удивляться, как близко — практически «под носом» у программиста — находятся средства для получения системной информации и как сложно о них узнать. Речь идет о средствах, которые всегда доступны при программировании для Windows — функции Windows API.

В данной главе мы рассмотрим некоторые способы, при помощи которых можно «добыть» информацию, касающуюся операционной системы. Это может пригодиться, например, если вы используете в своих приложениях возможности, отличающиеся в различных платформах Windows. Но и не только в этих случаях.

Рассмотренные в данной главе функции Windows API являются самыми обычными во всех смыслах этого слова. Просто они часто упоминаются вскользь либо вообще не упоминаются в книгах для программирования в таких средах, как Borland Delphi.

В примерах представленной вашему вниманию главы, кроме получения информации о самой Windows, некотором оборудовании компьютера, также рассмотрена работа с системным реестром Windows — этаким базой данных, в которой хранится много всего полезного и не очень: от параметров ОС и настроек приложений до сведений о работе компьютера в реальном времени. Правда, по определенным причинам последние сведения хранятся не в реальных, а в виртуальных ключах реестра. Но обо всем по порядку.

7.1. Системная информация

Начнем с несложных примеров, позволяющих получить информацию об операционной системе, установленном на компьютере оборудовании и такие сведения реального времени, как загрузка памяти компьютера, состояние питания и т. д.

Версия операционной системы

Получение сведений об операционной системе хотя и не является повседневной необходимостью, но все же в некоторых специфичных случаях может пригодиться. Например, когда ваша программа ведет себя по-разному при разных установленных обновлениях Windows. Либо когда вы самостоятельно пишете инсталлятор, который способен устанавливать версии программы, скомпилированные для Windows Me (95, 98) или Windows NT (2000, XP).

Одним из способов узнать версию Windows является использование API-функции `GetVersionEx`. Она принимает в качестве параметра структуру `OSVERSIONINFO` (или `OSVERSIONINFOEX`, но об этом позже), заполняет поля этой структуры и в случае удачной операции возвращает ненулевое значение.

Объявление ANSI-версии структуры `OSVERSIONINFO` в библиотеке Delphi 7 выглядит следующим образом:

```
OSVERSIONINFOA = record
    dwOSVersionInfoSize: DWORD; //Размер структуры
```

```
dwMajorVersion: DWORD;      //Старшая часть версии ОС Windows
dwMinorVersion: DWORD;      //Младшая часть версии
dwBuildNumber: DWORD;       //Номер сборки операционной системы
dwPlatformId: DWORD;        //Идентификатор платформы Windows
szCSDVersion: array[0..127] of AnsiChar; //Дополнительные
                                     //сведения, например, установленный пакет обновлений
end;
```

Не будем вдаваться в подробное описание возможных значений полей этой структуры: практически все будет ясно из приведенного далее примера. Напомним лишь, чтобы вы не забывали заполнять поле `dwOSVersionInfoSize` перед вызовом функции `GetVersionEx`.

Итак, пример обработки данных, помещаемых в структуру `OSVERSIONINFO`, приведен в листинге 7.1. При загрузке формы элемент управления `ListView` с именем `lvwVerInfo` заполняется сведениями о версии системы, представленными в читательской форме.

Листинг 7.1. Получение и отображение сведений о Windows

```
procedure TForm1.FormCreate(Sender: TObject);
var
  info: OSVERSIONINFO;
  item: TListItem;
begin
  //Получаем информацию о версии ОС
  info.dwOSVersionInfoSize := SizeOf(info);
  GetVersionEx(info);
  //Заполняем список информацией о ОС
  //..версия ОС
  item := lvwVerInfo.Items.Add();
  item.Caption := 'Версия системы';
  item.SubItems.Insert(0, IntToStr(Integer(info.dwMajorVersion)) +
    '.' + IntToStr(Integer(info.dwMinorVersion)));
  //..номер сборки
  item := lvwVerInfo.Items.Add();
  item.Caption := 'Сборка';
  item.SubItems.Insert(0, IntToStr(Integer(info.dwBuildNumber)));
  //..платформа
  item := lvwVerInfo.Items.Add();
  item.Caption := 'Платформа';
  case info.dwPlatformId of
    VER_PLATFORM_WIN32s:
```

```

    //Эмуляция Win32 или Win16
    item.SubItems.Insert(0, 'Win16');
VER_PLATFORM_WIN32_WINDOWS:
    //"Классическая" Win32: 95, 98 или Me
    item.SubItems.Insert(0, 'Win32');
VER_PLATFORM_WIN32_NT:
    //Ядро NT
    item.SubItems.Insert(0, 'WinNT');
end;
//..дополнительная информация (например, пакет обновлений)
item := lvwVerInfo.Items.Add();
item.Caption := 'Дополнительные сведения';
item.SubItems.Insert(0, info.szCSDVersion);
end;

```

Возможный результат работы программы (для Windows XP SP1) приводится на рис. 7.1.

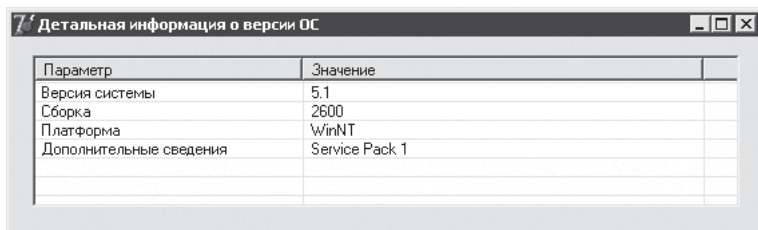


Рис. 7.1. Информация о версии Windows

Теперь снова обратимся к функции `GetVersionEx`, точнее говоря, к структуре `OSVERSIONINFOEX`, которая может также передаваться в качестве параметра в функцию. К сожалению, в библиотеке Delphi 7 эта структура не объявлена. Но это можно сделать самостоятельно:

```

OSVERSIONINFOEX = record
    dwOSVersionInfoSize: DWORD;
    dwMajorVersion: DWORD;
    dwMinorVersion: DWORD;
    dwBuildNumber: DWORD;
    dwPlatformId: DWORD;
    szCSDVersion: array[0..127] of AnsiChar;
    //Поля, которых нет в OSVERSIONINFO
    wServicePackMajor: WORD; //Старшая цифра версии пакета
                           //обновлений

```

```
wServicePackMinor: WORD; //Младшая цифра версии пакета
                        //обновлений
wSuiteMask: WORD;      //Комплектация системы
wProductType: BYTE;    //Дополнительная информации об ОС
wReserved: BYTE;
end;
```

Дополнительные (по сравнению с OSVERSIONINFO) поля структуры может заполнить ОС Windows NT 4.0 SP6 и более поздние версии Windows NT (в том числе 2000 и XP). Значения дополнительных полей структуры OSVERSIONINFOEX пояснены комментариями в объявлении структуры.

Значение поля wSuiteMask (является битовой маской) может быть составлено из значений следующих констант (увы, но их объявления также пришлось добавить самостоятельно).

```
VER_SUITE_BACKOFFICE = 4; //Установлена Microsoft Back Office
VER_SUITE_DATACENTER = 128; //Установлена Microsoft Data Center
VER_SUITE_ENTERPRISE = 2; //Установлена ОС Windows 2000
                        //Advanced Server
VER_SUITE_SMALLBUSINESS = 1; //Установлена Microsoft Small
                        //Business Server
VER_SUITE_SMALLBUSINESS_RESTRICTED = 32; //Установлена
                        //ограниченная версия Microsoft
                        //Small Business Server
VER_SUITE_TERMINAL = 16; //Установлены терминальные службы
VER_SUITE_PERSONAL = 512; //Персональная версия ОС (типичный
                        //набор функций меньше, чем
                        //в Professional)
```

Значение поля wProductType может быть одним из приведенных ниже (тип сетевой ОС и соответственно роль, которую компьютер с данной ОС может исполнять при подключении в сети):

```
VER_NT_WORKSTATION = 1; //Рабочая станция
VER_NT_DOMAIN_CONTROLLER = 2; //Контроллер домена
VER_NT_SERVER = 3; //Сервер
```

Чтобы можно было просто передавать в функцию `GetVersionEx` ссылку на структуру OSVERSIONINFOEX, а не OSVERSIONINFO, перегрузим эту функцию следующим образом:

```
function GetVersionEx(var lpVersionInformation: OSVERSIONINFOEX):
BOOL;
stdcall; external kernel32 name 'GetVersionExA';
```

Теперь определение полной информации о версии ОС для случая Windows на платформе NT (выше NT 4.0 SP6) может выглядеть следующим образом (листинг 7.2) (часть, одинаковая с листингом 7.1, опущена).

Листинг 7.2. Определение версии ОС (NT, 2000, XP)

```
procedure TForm1.FormCreate(Sender: TObject);
var
  info: OSVERSIONINFOEX;
  item: TListItem;
  suite, additional: String;
begin
  //Получаем информацию о версии ОС
  info.dwOSVersionInfoSize := SizeOf(info);
  GetVersionEx(info);
  //Заполняем список информацией об ОС
  //...
  //..версия о пакете обновлений
  item := lvwVerInfo.Items.Add();
  item.Caption := 'Версия ServicePack';
  item.SubItems.Insert
    (0, IntToStr(Integer(info.wServicePackMajor)) + '.' +
      IntToStr(Integer(info.wServicePackMinor)));
  //..компликация ОС
  suite := '';
  if info.wSuiteMask and VER_SUITE_BACKOFFICE <> 0 then
    suite := suite + '[Установлен Back Office] ';
  if info.wSuiteMask and VER_SUITE_DATACENTER <> 0 then
    suite := suite + '[Microsoft Data Center] ';
  if info.wSuiteMask and VER_SUITE_ENTERPRISE <> 0 then
    suite := suite + '[Windows 2000 Advanced Server] ';
  if info.wSuiteMask and VER_SUITE_SMALLBUSINESS <> 0 then
    suite := suite + '[Small Business Server] ';
  if info.wSuiteMask and VER_SUITE_SMALLBUSINESS_RESTRICTED <> 0
  then
    suite := suite + '[Small Business Server, ограниченная версия] ';
  if info.wSuiteMask and VER_SUITE_TERMINAL <> 0 then
    suite := suite + '[Terminal Service] ';
  if info.wSuiteMask and VER_SUITE_PERSONAL <> 0 then
    suite := suite + '[Workstation Personal (не Professional)] ';
  item := lvwVerInfo.Items.Add();
```

```
item.Caption := 'Комплектация';
item.SubItems.Add(suite);

//..дополнительные сведения
additional := '';
if info.wProductType and VER_NT_WORKSTATION <> 0 then
    additional := additional + '[Рабочая станция] ';
if info.wProductType and VER_NT_DOMAIN_CONTROLLER <> 0 then
    additional := additional + '[Контроллер домена] ';
if info.wProductType and VER_NT_SERVER <> 0 then
    additional := additional + '[Сервер] ';

item := lvwVerInfo.Items.Add();
item.Caption := 'Дополнительно';
item.SubItems.Add(additional);
end;
```

Имя компьютера

Следующий простой пример (листинг 7.3) показывает, как можно определить сетевое имя компьютера. Функция `ComputerName` скрывает «прелести» работы со строковым буфером, который нужно передавать в API-функцию `GetComputerName`.

Листинг 7.3. Определение сетевого имени компьютера

```
function ComputerName(): String;
var
    buffer: String;
    len: Cardinal;
begin
    len := MAX_COMPUTERNAME_LENGTH + 1;
    SetLength(buffer, len);
    if GetComputerName(PAnsiChar(buffer), len) <> False then
        ComputerName := Copy(buffer, 1, len)
    else
        ComputerName := '';
end;
```

Имя пользователя

Определить имя пользователя, от имени которого запущена программа (а точнее — вызывающий функцию поток), можно с использованием функции из листинга 7.4.

Если значения полей `BatteryLifePercent`, `BatteryLifeTime`, `BatteryFullLifeTime` предельно ясны, то извлечение информации из полей `ACLineStatus` и `BatteryFlag` можно посмотреть в листинге 7.5.

Листинг 7.5. Определение состояния системы питания

```
procedure TForm1.LoadPowerStatus();
var
    batFlags: String;
    status: TSystemPowerStatus;
    prof_info: THWProfileInfo;
begin
    lvwPowerStatus.Clear;
    //Получаем информацию о состоянии питания
    ZeroMemory(Addr(status), SizeOf(status));
    GetSystemPowerStatus(status);
    //Заполняем список информацией о состоянии питания
    //..подключение к сети
    case status.ACLineStatus of
        0: AddParam('Подключение к сети', 'Отключен');
        1: AddParam('Подключение к сети', 'Подключен');
    else AddParam('Подключение к сети', 'Неизвестно');
    end;
    //..заряд батареи (битовая маска)
    if status.BatteryFlag and 1 <> 0 then batFlags := 'Высокий ';
    if status.BatteryFlag and 2 <> 0 then batFlags := batFlags +
'Низкий ';
    if status.BatteryFlag and 4 <> 0 then
        batFlags := batFlags + 'Критический ';
    if status.BatteryFlag and 8 <> 0 then
        batFlags := batFlags + '(Идет зарядка)';
    if status.BatteryFlag and 128 <> 0 then
        batFlags := batFlags + 'Батарея не установлена';
    if status.BatteryFlag = 255 then batFlags := batFlags + 'Не-
известно';
    AddParam('Заряд батареи', batFlags);

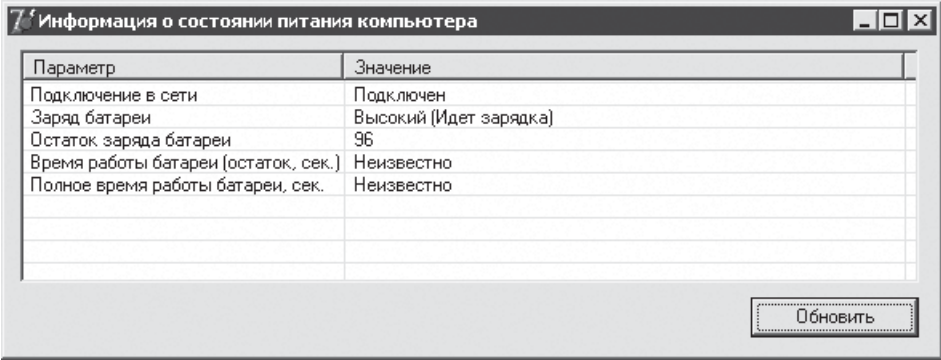
    //..численные характеристики батареи
    if status.BatteryLifePercent <> 255 then
        AddParam('Остаток заряда батареи',
```

```
        IntToStr(Integer(status.BatteryLifePercent)))
else
    AddParam('Остаток заряда батареи', 'Неизвестно');

if status.BatteryLifeTime <> Cardinal(-1) then
    AddParam('Время работы батареи (остаток, сек.)',
        IntToStr(Integer(status.BatteryLifeTime)))
else
    AddParam('Время работы батареи (остаток, сек.)', 'Неизвестно');

if status.BatteryFullLifeTime <> Cardinal(-1) then
    AddParam('Полное время работы батареи, сек.',
        IntToStr(Integer(status.BatteryFullLifeTime)))
else
    AddParam('Полное время работы батареи, сек.', 'Неизвестно');
end;
```

В листинге 7.5 для отображения каждого параметра системы питания вызывается процедура `AddParam`, добавляющая в элемент управления формы название параметра и его значение. Этим элементом управления может быть, например, `ListView`. Для такого случая возможный результат работы процедуры `LoadPowerStatus` показан на рис. 7.2.



Параметр	Значение
Подключение в сети	Подключен
Заряд батареи	Высокий (Идет зарядка)
Остаток заряда батареи	96
Время работы батареи (остаток, сек.)	Неизвестно
Полное время работы батареи, сек.	Неизвестно

Рис. 7.2. Собранная информация о системе питания

В нашем случае можно заключить, что программа испытывалась на компьютере, хоть и снабженном аккумулятором, но с явно недоделанной системой питания.

И последние несколько слов о том, когда рассмотренный пример может реально пригодиться. А пригодится он в случае, если ваше приложение оперирует большим объемом важных данных, на сохранение которых требуется длительное время и потеря которых может принести большие неприятности. Тогда при обнаружении

разрядки батареи приложение может сохранить (а точнее, длительное время сохранять) данные на диск до лучших времен, например, до тех пор, пока питание вновь не будет включено, а заряд батареи не достигнет требуемого значения.

Состояние памяти компьютера

Получение снимка текущего состояния памяти компьютера также является несложной задачей. Недаром эту информацию многие приложения, тот же Блокнот, выводят в окне **О программе**: заполнить форму чем-то надо, а сведения об объеме памяти кажутся довольно актуальными.

Итак, получить состояние памяти компьютера можно при помощи API-функции `GlobalMemoryStatus`. Данная функция принимает в качестве параметра структуру `TMemoryStatus`, заполняет ее поля значениями и в случае успеха возвращает отличное от нуля число. Объявление структуры `TMemoryStatus` с комментариями роли ее полей приводится ниже:

```
TMemoryStatus = record
    dwLength: DWORD;      //Размер структуры (байт)
    dwMemoryLoad: DWORD;  //Процент загрузки физической памяти
    dwTotalPhys: DWORD;   //Полный объем физической памяти
    dwAvailPhys: DWORD;   //Объем свободной оперативной памяти
    dwTotalPageFile: DWORD; //Полный объем файла подкачки
    dwAvailPageFile: DWORD; //Объем свободного пространства
                          //в файле подкачки
    dwTotalVirtual: DWORD; //Полный объем виртуальной памяти
    dwAvailVirtual: DWORD; //Объем свободной виртуальной памяти
end;
```

Два последние поля структуры `TMemoryStatus` относятся к приложению, вызывающему функцию `GlobalMemoryStatus`. Они рассмотрены чуть ниже. Пример использования функции `GlobalMemoryStatus` приведен в листинге 7.6.

Листинг 7.6. Определение состояния памяти

```
procedure TForm1.LoadMemoryInfo();
var
    memStat: TMemoryStatus;
begin
    memStat.dwLength := SizeOf(memStat);
    //Получение информации о загрузке памяти
    GlobalMemoryStatus(memStat);
    //Заполнение полей формы
    //...% использования памяти
```

```
pbMemUsage.Position := memStat.dwMemoryLoad;  
lblMemUsage.Caption := IntToStr(memStat.dwMemoryLoad) + '%';  
//..использование оперативной памяти  
txtMemTotal.Text := IntToStr(memStat.dwTotalPhys div 1024);  
txtMemAvail.Text := InttoStr(memStat.dwAvailPhys div 1024);  
//..использование файла подкачки  
txtPageTotal.Text := IntToStr(memStat.dwTotalPageFile div 1024);  
txtPageAvail.Text := InttoStr(memStat.dwAvailPageFile div 1024);  
//..использование виртуальной памяти  
txtVirtualTotal.Text := IntToStr(memStat.dwTotalVirtual div 1024);  
txtVirtualAvail.Text := InttoStr(memStat.dwAvailVirtual div 1024);  
end;
```

Внешний вид формы, элементы управления которой заполняются значениями в листинге 7.6, показан на рис. 7.3.

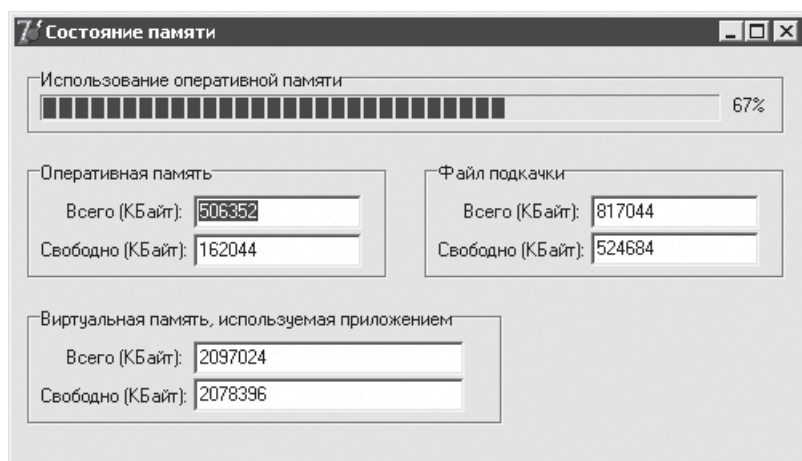


Рис. 7.3. Программа для определения состояния памяти компьютера

Напоследок рассмотрим (несколько упрощенно), что за результаты выводятся в текстовых полях формы, для тех, кто немного не в курсе, как организовано управление памятью в ОС Windows.

Итак, каждому процессу Windows предоставляет адресное пространство (виртуальное) размером чуть меньше 2 Гбайт. В отличие от 16-битных предшественниц, в 32-битных Windows адресные пространства различных процессов являются закрытыми: приложение использует память (а точнее, младшие 2 Гбайт адресного пространства) единолично и не может без дополнительных усилий манипулировать данными других процессов. Значения в двух последних полях структуры `TMemoryStatus` (и нижняя группа текстовых полей на форме рис. 7.3) как раз и показывают использование приложением предоставляемого ему адресного пространства.

Механизм виртуальной памяти является довольно удобной надстройкой, скрывающей ограниченность аппаратных ресурсов компьютера. Ограниченный объем оперативной памяти компенсируется использованием места на диске (файла подкачки, страничного файла). В этот файл записываются для временного хранения неиспользуемые страницы памяти (блоки данных по несколько Кбайт), давая возможность помещать другие данные, нужные приложению, в оперативную память.

Теперь вернемся к форме, показанной на рис. 7.3. Группа текстовых полей **Оперативная память** показывает полный и свободный объем реально установленной на компьютере оперативной памяти (за вычетом памяти, используемой для системных нужд). Использование этого вида памяти иллюстрирует индикатор **ProgressBar** на форме. Назначение правой группы текстовых полей (**Файл подкачки**) должно быть также очевидным.

Из цифр, выведенных в текстовые поля на форме (рис. 7.3), можно также определить, что общий объем памяти, доступной приложениям (всего было запущено 30 процессов), на испытуемом компьютере составлял около 1,26 Гбайт. Если представить, что память использовалась всеми процессами одинаково, то получается примерно 43 Мбайт на каждого, не считая памяти, резервируемой для самой ОС Windows.

7.2. Системное время

Этот раздел посвящен отнюдь не простому получению текущего времени или даты (благо эти функции можно найти и в библиотеке Borland). Здесь мы обратимся к несколько более интересной теме — использованию системных средств измерения малых промежутков времени.

Все рассмотренные далее способы измерения времени основаны на подсчете количества «тиков» таймера. Для сохранения показаний таймера система поддерживает соответствующие счетчики. Для определения временного интервала получаем показания счетчика в начале и в конце промежутка времени. Находим разность между полученными показаниями и, если период таймера не соответствует требуемой единице измерения (например, мс), делим разность на частоту таймера.

Давно ли запущена операционная система?

С момента своего запуска Windows начинает наращивание значения специального счетчика, показывающего количество «тиков» (в миллисекундах), прошедших с момента запуска системы.

Таким образом, этот системный счетчик «тиков» можно использовать как для определения времени работы системы, так и для измерения временных интервалов. Для доступа к нему можно использовать API-функцию `GetTickCount`. Она не имеет параметров и возвращает целочисленное 32-битное значение.

Приведенная в листинге 7.7. функция `GetSystemWorkTime` демонстрирует использование счетчика «тиков» для определения времени работы системы в часах, минутах и секундах.

Листинг 7.7. Определение времени работы системы

```
function GetSystemWorkTime(): String;
var
    ticks: DWORD;
    hh, mm, ss: Cardinal;
begin
    //Получаем количество миллисекунд с момента старта системы
    ticks := GetTickCount();
    //Переводим в секунды
    ticks := ticks div 1000;
    //Получаем количество часов, минут, секунд
    hh := ticks div 3600;
    Dec(ticks, hh * 3600);
    mm := ticks div 60;
    Dec(ticks, mm * 60);
    ss := ticks;

    GetSystemWorkTime := IntToStr(hh) + ':' +
                        IntToStr(mm) + ':' + IntToStr(ss);
end;
```

Из-за относительно малой разрядности значение счетчика обнуляется приблизительно каждые 49,7 суток, что следует учитывать при измерении длительных интервалов или если измерение времени начинается после длительной работы системы (например, начало измерения выпадает на 50-е сутки за час до обнуления счетчика).

Аппаратный таймер

Следующий рассматриваемый способ измерения времени основан на использовании таймера высокого разрешения (высокочастотного). Временной промежуток между «тиками» этого таймера может быть намного меньше 1 мс, что позволяет производить достаточно точные измерения. Для сохранения количества «тиков» аппаратного таймера используется 64-битный счетчик.

Пример получения значения счетчика аппаратного таймера приводится в листинге 7.8. Частота, возвращаемая функцией `hwTimerGetCounter`, измеряется в Гц (с^{-1}), то есть означает количество срабатываний таймера в 1 с.

Листинг 7.8. Получение значения счетчика аппаратного таймера

```
function hwTimerGetCounter(): Int64;
var
    freq: Int64;
```

```
begin
  if QueryPerformanceCounter(freq) <> False then
    hwTimerGetCounter := freq
  else
    hwTimerGetCounter := 0; //Ошибка
end;
```

Чтобы перевести количество «тиков» аппаратного таймера в привычные нам единицы измерения, нужно узнать его частоту. В этом нам поможет функция, приведенная в листинге 7.9.

Листинг 7.9. Определение частоты аппаратного таймера

```
function hwTimerGetFreq(): Int64;
var
  freq: Int64;
begin
  if QueryPerformanceFrequency(freq) <> False then
    hwTimerGetFreq := freq
  else
    hwTimerGetFreq := 0; //Ошибка
end;
```

Пусть нам известна разность между значения счетчика в начале и конце измерения. Перевести ее в секунды можно следующим образом:

```
time := counter div hwTimerGetFreq();
```

Пример, а точнее, результат определения характеристик аппаратного таймера приведен на рис. 7.4.

Аппаратный таймер	
Частота таймера (Гц):	3579545
Период (мс):	0.000279365114840015
Текущее значение счетчика:	689257063513
Время работы системы, рассчитанное по показаниям аппаратного таймера:	53:29:14

Рис. 7.4. Характеристики аппаратного таймера

Заполнение приведенных на рис. 7.4 текстовых полей осуществляется чрезвычайно просто, поэтому код, описывающий это, в тексте не приводится. При желании вы сможете найти его на диске, прилагаемом к книге, в папке с названием раздела.

Мультимедиа-таймер

Рассмотрим еще один способ измерения, основанный на использовании так называемого мультимедиа-таймера. Его использование удобно тем, что появляется возможность задания его точности. Группа API-функций работы с мультимедиа-таймером позволяет не только измерять временные интервалы, но и создавать программные таймеры (см. компонент `Timer`), срабатывающие через гораздо меньшие промежутки времени.

Для получения текущего значения счетчика мультимедийного таймера можно воспользоваться функцией `timeGetTime`. Вообще, она возвращает значения, аналогичные значениям, возвращаемым функцией `GetTickCount`. Счетчик также 32-битный, обнуляемый приблизительно каждые 49,7 суток. Прототип функции `timeGetTime` следующий:

```
function timeGetTime: DWORD; stdcall;
```

Пример использования этой функции приведен в листинге 7.12, а теперь несколько слов о том, как получить для рассматриваемого таймера значения минимальной и максимальной точности. Для получения этих данных можно использовать функцию `timeGetDevCaps`. Она принимает в качестве параметра структуру `TTimeCaps` и заполняет два ее поля соответствующими значениями. В листинге 7.10 приводится возможная реализация функций для определения характеристик мультимедийного таймера.

Листинг 7.10. Определение характеристик мультимедиа-таймера

```
//Получение максимального периода таймера (мс)
function timeGetMaxPeriod(): Cardinal;
var
    time: TTimeCaps;
begin
    timeGetDevCaps(Addr(time), SizeOf(time));
    timeGetMaxPeriod := time.wPeriodMax;
end;

//Получение минимального периода таймера (мс)
function timeGetMinPeriod(): DWORD;
var
    time: TTimeCaps;
begin
    timeGetDevCaps(Addr(time), SizeOf(time));
    timeGetMinPeriod := time.wPeriodMin;
end;
```

Итак, мы знаем, как получать параметры таймера. Но было сказано, что его точность можно регулировать. Делается это при помощи функций `timeBeginPeriod` и `timeEndPeriod`.

- Первая функция вызывается для установления минимальной точности таймера, которая устраивает приложение. Функция `timeBeginPeriod` принимает значение требуемой точности таймера в миллисекундах, возвращает `TIMERR_NOERROR` в случае успеха либо `TIMERR_NOCANDO`, если требуемая точность не может быть обеспечена.
- Вторая функция восстанавливает точность таймера такой, какой она была до вызова функции `timeBeginPeriod`. В функцию `timeEndPeriod` должно передаваться то же значение, что и в функцию `timeBeginPeriod`.

В листинге 7.11 показано использование функций `timeBeginPeriod`, а также `timeEndPeriod` (реализованы функции-оболочки). При использовании функций из листинга 7.11 нужно помнить, что после вызова `timeSetTimerPeriod` и проведения измерения обязательно должна быть вызвана `timeRestoreTimerPeriod`. Функция `timeSetTimerPeriod` сохраняет значение установленной точности таймера в глобальной переменной `lastPeriod`, чтобы можно было не заботиться о сохранении этого значения в коде, использующем таймер.

Листинг 7.11. Функции изменения точности таймера

```
Var lastPeriod: Cardinal;  
//Установка периода таймера (мс) перед началом измерения  
function timeSetTimerPeriod(period: Cardinal): Boolean;  
begin  
    if timeBeginPeriod(period) = TIMERR_NOERROR then  
        begin  
            //Сохраним значение для восстановления состояния таймера  
            lastPeriod := period;  
            timeSetTimerPeriod := True;  
        end  
    else  
        //Неудача  
        timeSetTimerPeriod := False;  
    end;  
//Восстановление периода таймера (обязательно)  
function timeRestoreTimerPeriod(): Boolean;  
begin  
    if timeEndPeriod(lastPeriod) = TIMERR_NOERROR then  
        timeRestoreTimerPeriod := True  
    else  
        timeRestoreTimerPeriod := False;  
    end;
```

Теперь, после долгого рассмотрения особенностей настройки мультимедиа-таймера, приведем пример его использования для измерения времени выполнения простейшего отрезка программы (листинг 7.12).

Листинг 7.12. Измерение времени выполнения отрезка программы

```
procedure TForm1.cmbTimeGoClick(Sender: TObject);
var
    summ, arg, maxVal: Int64;
    startTime, endTime: Cardinal;
begin
    txtTimeResult.Text := 'Измерение...';
    Refresh;
    maxVal := StrToInt(txtTimeMaxVal.Text);

    //Устанавливаем максимальную точность таймера
    timeSetTimerPeriod(timeGetMinPeriod());
    startTime := timeGetTime(); //Начальный момент времени

    //Суммируем 64-битные числа
    //(как раз и измеряем время его выполнения)
    summ := 0;
    arg := 1;
    while (arg <= maxVal) do
    begin
        Inc(summ, arg);
        Inc(arg);
    end;

    endTime := timeGetTime(); //Конечный момент времени
    //Восстанавливаем период таймера
    timeRestoreTimerPeriod();

    //Время выполнения операций (мс)
    txtTimeResult.Text := IntToStr(endTime - startTime);
end;
```

Создание программного таймера высокой точности

В самом начале рассмотрения возможностей мультимедиа-таймера было сказано, что в его API заложена возможность создания программных таймеров. Это действительно так. Причем максимальная точность такого таймера может получиться довольно большой: на современных компьютерах создание программного тай-

мера с периодом срабатывания 1 мс — не проблема. Правда, использовать максимальную частоту таймера вряд ли стоит: слишком велика вероятность ошибки как минимум на 1 мс.

Теперь уясним, что же за программный таймер мы создаем и чем он отличается от компонента `Timer`, помещаемого на форму. А отличается наш таймер, кроме высокой точности, тем, что его не нужно привязывать к окну (форме): при срабатывании стандартного компонента `Timer` окну, за которым он закреплен, посылается сообщение `WM_TIMER`. Создаваемый же нами таймер работает по-другому, что удобнее рассмотреть на примере.

```
timerID := timeSetEvent
(
  StrToInt(txtTimeInterval.Text), //Интервал между
                                //срабатываниями таймера
  timeGetMinPeriod(), //Точность таймера
  TimerProc,           //Адрес процедуры, вызываемой при каждом
                        //срабатывании таймера
  0,                   //Параметр, передаваемый в процедуру
                        //обратного вызова
  TIME_CALLBACK_FUNCTION or TIME_PERIODIC //Тип таймера
);
```

В приведенном выше отрывке программы с помощью функции `timeSetEvent` происходит регистрация и запоминание адреса процедуры `TimerProc`, вызываемой периодически при срабатываниях таймера. При успешном создании таймера функция `timeSetEvent` возвращает ненулевое значение — идентификатор созданного таймера. Оно может использоваться в дальнейшем для определения, какой именно таймер работал. Значение, возвращенное функцией `timeSetEvent`, также необходимо при удалении таймера:

```
timeKillEvent(timerID);
```

Функция `timeKillEvent` возвращает целочисленное значение:

- `TIMERR_NOERROR` — если ее вызов завершился успешно;
- `MMSYSERR_INVALIDPARAM` — если таймера, заданного параметром функции, не существует.

Теперь о процедуре, адрес которой мы передаем в функцию `timeSetEvent`. В нашем примере она выглядит следующим образом (листинг 7.13).

Листинг 7.13. Процедура, вызываемая при срабатывании таймера

```
procedure TimerProc(uTimerID, uMessage: UINT; dwUser, dw1, dw2:
DWORD) stdcall;
begin
```

```
//Добавляем текущее значение времени в список (чтобы была  
//видна разница между моментами вызова этой процедуры)  
Form1.lstTimes.Items.Add(IntToStr(timeGetTime()));
```

```
end;
```

Естественно, действия, выполняемые процедурой `TimerProc`, могут быть самыми различными. В нашем случае происходит заполнение списка (List) значениями счетчика «тиков» таймера на момент вызова процедуры (рис. 7.5).

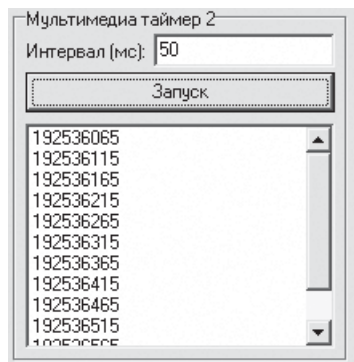


Рис. 7.5. Результат работы таймера

В завершение вновь обратимся к функции `timeSetEvent`: кратко перечислим предоставляемые ею возможности, которыми мы не воспользовались в приведенном выше примере.

Как вы могли заметить, последний параметр функции `timeSetEvent` является битовой маской. Флаги этой маски задают два аспекта поведения таймера: количество срабатываний таймера и тип действия, которое требуется выполнять при срабатывании таймера.

Количество срабатываний таймера определяется двумя значениями.

- `TIME_ONESHOT` — таймер срабатывает один раз. Для таких таймеров вызывать `timeKillEvent` после срабатывания не нужно.
- `TIME_PERIODIC` — таймер срабатывает периодически через заданные промежутки времени.

Тип действия, выполняемого таймером, задается при помощи следующих констант:

- `TIME_CALLBACK_FUNCTION` — при срабатывании таймера вызывается процедура, адрес которой был передан третьим параметром;
- `TIME_CALLBACK_EVENT_SET` — вызывает `SetEvent` для объекта синхронизации «событие», дескриптор которого передан третьим параметром;
- `TIME_CALLBACK_EVENT_PULSE` — вызывается `PulseEvent` для объекта синхронизации «событие», дескриптор которого передан третьим параметром.

К сожалению, использование объектов синхронизации хоть и является темой для интересного разговора, но все же выходит за рамки этой главы. Потому, упомянув о соответствующих возможностях таймера, больше не будем распространяться на эту тему.

7.3. Реестр

Далее будет рассмотрено несколько примеров использования в программах на Delphi одного из важнейших хранилищ информации Windows — системного реестра.

Краткие сведения о реестре Windows

Что же представляет собой системный реестр и для чего он предназначен? Реестр состоит из нескольких файлов с довольно сложной организацией записей, формирующих иерархическую структуру (родитель—потомки), а точнее, несколько веток структуры. Благодаря наличию специальных функций мы можем работать с реестром именно как с иерархической структурой, а не как с набором записей в файле.

Реестр Windows является отличным примером организации централизованного хранения данных, в основном, настроек программ. Реестр является хорошей альтернативой большим INI-файлам, доставшимся в наследство от 16-разрядных версий Windows, главным образом из-за возможности лучше структурировать информацию (ведь секции разделов в реестре могут быть много раз вложенными). В реестре хранятся и данные, которые могут пригодиться сразу многим программам: например, расположения СОМ-серверов, пути приложений, ассоциированных с различными типами файлов.

В реестре могут быть объекты двух типов: разделы (во многом аналогичны папкам файловой системы) и параметры (имеют имя, тип и значение).

Данные реестра сгруппированы в несколько ветвей (рис. 7.6). Для запуска показанной на рис. 7.6 программы Редактор реестра достаточно набрать в командной строке `Regedit` либо отыскать файл `Regedit.exe` в каталоге Windows.

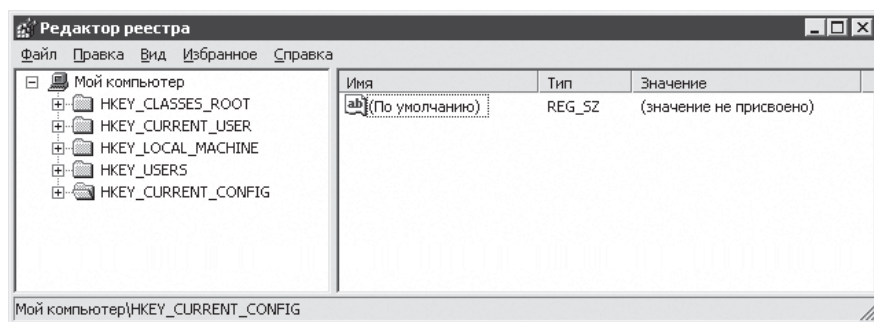


Рис. 7.6. Корневые разделы реестра

Информация, помещаемая в различных разделах реестра, группируется по следующим признакам.

- `HKEY_CURRENT_USER` — в этом разделе хранится информация, используемая для текущего пользователя, осуществившего вход в систему. Этой информацией могут быть, например, значения переменных окружения, фон Рабочего стола, вид меню Пуск.
- `HKEY_USERS` — содержит настройки системы для различных пользователей, а также настройки, используемые по умолчанию для нового пользователя.
- `HKEY_LOCAL_MACHINE` — самая большая и главная ветвь реестра, содержащая параметры Windows, приложений, оборудования, ассоциации расширений файлов, расположение COM-серверов и еще много чего полезного.
- `HKEY_CURRENT_CONFIG` — в этом разделе хранятся значения параметров Windows, отличающихся от стандартных. Он является псевдонимом для ветви `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Hardware Profiles\Current`.
- `HKEY_CLASSES_ROOT` — в системах Windows 95/98/NT 4.0 и более ранних этот раздел является псевдонимом для ветви `HKEY_LOCAL_MACHINE\SOFTWARE\Classes`. В Windows 2000/XP содержимое этого раздела составляется из содержимого разделов `HKEY_LOCAL_MACHINE\SOFTWARE\Classes` и `HKEY_CURRENT_USER\Software\Classes`.

Доступ к разделам реестра происходит по дескрипторам. Дескриптор раздела можно получить при его создании или открытии, указав дескриптор одной из рассмотренных выше корневых ветвей, а также путь требуемого раздела. Для хранения дескрипторов корневых ветвей реестра определены одноименные константы.

Средства работы с реестром

Для работы с реестром предусмотрена целая группа API-функций. Однако зачем изобретать велосипед, испытывая на себе «удобство» работы с этими функциями? Ведь Borland предоставила нам в распоряжение замечательный по своей простоте класс `TRegistry`. Использованию этого класса как раз и посвящено несколько следующих абзацев.

Итак, класс `TRegistry` находится в модуле `Registry`. Если кому-то все же станет интересно использование API для работы с реестром, то можете заглянуть в этот модуль и там посмотреть, как реализованы методы класса `TRegistry`.



ПРИМЕЧАНИЕ

Помимо `TRegistry`, в модуле `Registry` можно найти такие классы, как `TRegIniFile` и `TRegistryIniFile`, позволяющие работать с реестром, как будто бы это INI-файл. В ряде случаев использование этих классов вместо `TRegistry` позволит сократить размер программы, да и значительно ее упростить.

В табл. 7.1 приведены свойства класса TRegistry.

Таблица 7.1. Свойства класса TRegistry

Свойство	Тип	Назначение
CurrentKey	HKEY, только чтение	Дескриптор текущего раздела реестра
CurrentPath	String, только чтение	Путь текущего раздела
RootKey	HKEY	Дескриптор корневого раздела иерархии. Устанавливается до открытия, создания раздела. По умолчанию имеет значение HKEY_CURRENT_USER
LazyWrite	Boolean	Если False (по умолчанию — True), то при закрытии текущего раздела происходит немедленная запись изменений в файл реестра на диске. Использование немедленной записи данных реестра на диск является примером довольно расточительного использования ресурсов компьютера, но все же применимо, когда нужно гарантировать сохранение внесенных в реестр сведений. Если включен «ленивый» режим записи, то сохранение данных может произойти позже, например, когда система будет простаивать, либо перед выключением или перезапуском компьютера
Access	LongWord	Битовая маска, хранящая режим доступа к реестру. По умолчанию имеет значение KEY_ALL_ACCESS

Список констант, которые могут объединяться операцией `or` для формирования значения свойства Access:

- KEY_QUERY_VALUE — получение значений параметров раздела;
- KEY_ENUMERATE_SUB_KEYS — возможность составления списка подразделов;
- KEY_SET_VALUE — создания параметров в разделе, задание их значений;
- KEY_CREATE_SUB_KEY — создание подразделов;
- KEY_CREATE_LINK — создание символических ссылок (здесь не рассматривается);
- KEY_NOTIFY — право на уведомление об изменении раздела и его подразделов (здесь не рассматривается);
- KEY_READ — комбинация значений KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS и KEY_NOTIFY;
- KEY_WRITE — комбинация значений KEY_SET_VALUE и KEY_CREATE_SUB_KEY;
- KEY_ALL_ACCESS — комбинация значений KEY_READ, KEY_WRITE и KEY_CREATE_LINK.

Приводить список всех методов класса TRegistry в книге не рационально, да и не зачем. Благо, названия методов говорят сами за себя, к тому же Delphi поставляется с неплохой справочной системой. Здесь же мы остановимся на рассмотрении некоторых особенностей работы с методами класса TRegistry.

Итак, работая с разделами реестра, важно (в общем случае) соблюдать следующую последовательность.

1. Установить значение свойства `RootKey`, если корневой раздел отличен от `HKEY_CURRENT_USER`. Установить значение свойства `Access`, если не нужен полный доступ.
2. Открыть методом `OpenKey` или создать методом `CreateKey` раздел реестра. Если использовать `OpenKeyReadOnly`, то задавать значение свойства `Access`, как сказано в пункте 1, не имеет смысла.
3. Произвести нужные операции с элементами раздела.
4. Не забыть закрыть раздел, по крайней мере, если вы собираетесь использовать один и тот же объект `TRegistry` для последовательной работы с несколькими разделами (метод `OpenKey` не закрывает ранее открытый раздел).

Теперь несколько слов о проверке успешности работы методов класса `TRegistry`. Итак, большинство методов этого класса, осуществляющих доступ к разделам реестра, реализованы как функции, возвращающие `True` в случае успеха и `False` при возникновении ошибки. Вероятно, по каким-то чрезвычайно сложным соображениям разработчики класса `TRegistry` реализовали-таки функцию (!) `CreateKey` генерирующей исключение `ERegistryException` в случае неудачи, а не возвращающей значение `True` или `False`.

Для чтения/записи параметров разного типа в классе `TRegistry` предусмотрены пары `Read`- и `Write`-методов. Использовать их крайне просто, в чем вы убедитесь далее. Главное, при использовании этих методов не забывать определить тип значений параметров, если он заранее вам точно не известен, например с помощью функции `GetDataType`. Следует также помнить, что методы работы с параметрами генерируют исключение `ERegistryException` при возникновении ошибок.

И напоследок о параметре (По умолчанию) — он может присутствовать в каждом разделе. Для обращения к этому параметру используйте пустую строку в качестве имени раздела. Только нужно учитывать, что, в отличие от более ранних версий Windows, в Windows 2000/XP этот параметр автоматически не создается.

Хранение настроек программы в реестре

Первый простой пример демонстрирует, как можно использовать реестр для сохранения небольшого объема данных между запусками приложения.

Пусть нужно, чтобы формы приложения запоминали свое расположение, размер, введенные и выбранные в элементах управления данные. В таком случае необходимость в сотый раз перетаскивать часто открываемую форму на удобное место не будет раздражать пользователя. Если же форма требует постоянного ввода похожих данных, то восстановление выбранных и введенных в прошлый раз значений будет только плюсом.

Теперь о деле: есть форма для фильтрации запроса к базе данных, она показанна на рис. 7.7.

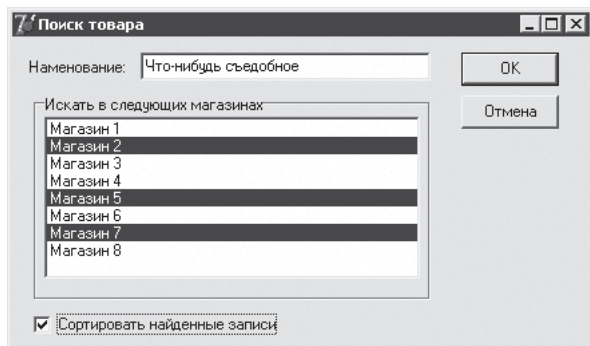


Рис. 7.7. Форма фильтра для поиска товара

Содержимое формы не суть важно, а важно то, что при нажатии кнопки **OK** положение, размер формы, а также данные, введенные пользователем, будут сохранены в реестре при помощи процедуры `SaveFilter` (листинг 7.14).

Листинг 7.14. Сохранение параметров формы в реестре

```
procedure TForm1.OKClick(Sender: TObject);
begin
    SaveFilter();
    //Выполняем требуемые действия...
end;
//Процедура сохраняет параметры в реестр
procedure TForm1.SaveFilter();
var
    reg: TRegistry; //По умолчанию: RootKey = HKEY_CURRENT_USER
    strShops: String;
    i: Integer;
begin
    reg := TRegistry.Create();
    try
        //Открываем или создаем раздел, в котором будут
        //сохранены параметры формы
        reg.OpenKey(strBaseKey + '\Form1', True);
        //Сохранение параметров
        //1. Размер и положение формы
        reg.WriteInteger('Width', Width);
        reg.WriteInteger('Height', Height);
        reg.WriteInteger('Top', Top);
        reg.WriteInteger('Left', Left);
```

```
//2. Последнее введенное наименование
reg.WriteString('txtName.Text', txtName.Text);
//3. Выбранные магазины
strShops := '';
for i := 0 to lstShops.Count-1 do
    if lstShops.Selected[i] then
        strShops := strShops + lstShops.Items[i] + ',';
reg.WriteString('lstShops.Selection', strShops);
//4. Применение сортировки
reg.WriteBool('chkSort.Checked', chkSort.Checked);

except
    on ERegistryException do
        MessageBox(Handle, 'Ошибка при сохранении фильтра',
            'Поиск товара', MB_ICONEXCLAMATION)
;
end;
reg.CloseKey();
reg.Free();
end;
```

В рассматриваемом примере константа `strBaseKey`, определяющая положение раздела для сохранения настроек, задана следующим образом:

```
const
    strBasekey = 'Software\Delphi. Трюки и эффекты\Настройки программы';
```

Открыв Редактор реестра, можно удостовериться в правильном сохранении требуемых нам параметров (рис. 7.8).

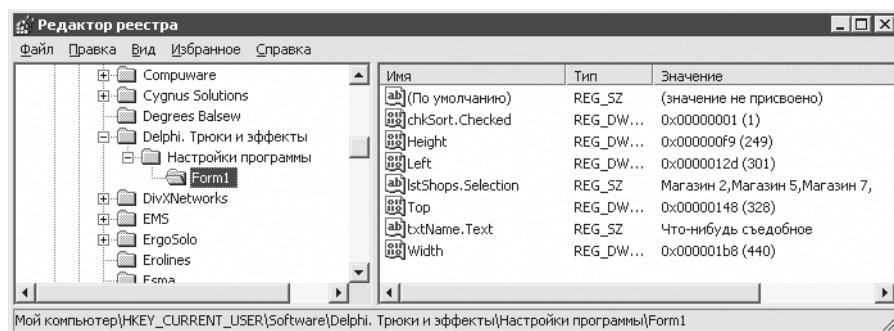


Рис. 7.8. Параметры формы, записанные в реестр

Считывание параметров формы можно производить, например, при ее создании. Тогда в обработчике события `Create` достаточно поместить вызов процедуры `LoadFilter` (листинг 7.15).

Листинг 7.15. Загрузка параметров формы из реестра

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    LoadFilter();
end;
//Процедура загружает параметры из реестра
procedure TForm1.LoadFilter();
var
    reg: TRegistry; //По умолчанию: RootKey = HKEY_CURRENT_USER
    strShops: String;
    shopStart, shopEnd: Integer;
begin
    reg := TRegistry.Create();
    try
        //Открываем раздел, в котором сохранены параметры формы
        reg.OpenKey(strBaseKey + '\Form1', False);
        //Загрузка сохраненных ранее параметров
        //1. Размер и положение формы
        Width := reg.ReadInteger('Width');
        Height := reg.ReadInteger('Height');
        Top := reg.ReadInteger('Top');
        Left := reg.ReadInteger('Left');
        //2. Последнее введенное наименование
        txtName.Text := reg.ReadString('txtName.Text');
        //3. Выбранные в прошлый раз магазины
        strShops := reg.ReadString('lstShops.Selection');
        shopStart := 1;
        for shopEnd := 0 to Length(strShops) do
            if strShops[shopEnd] = ',' then
                begin
                    //Получение имени магазина и выделение его в списке
                    SelectShop(Copy(strShops, shopStart,
                                    shopEnd - shopStart));
                    shopStart := shopEnd + 1;
                end;
            end;
```

```
//4. Применение сортировки
chkSort.Checked := reg.ReadBool('chkSort.Checked');
except
  on ERegistryException do
    //Игнорируем ошибки (просто не
    //будут зачитаны данные из реестра,
    //например, при первом запуске программы)
    ;
  end;
reg.CloseKey();
reg.Free();
end;
//Процедура выделяет магазин с заданным названием
//(если он есть в списке)
procedure TForm1.SelectShop(strShopName: String);
var
  i: Integer;
begin
  for i := 0 to lstShops.Count-1 do
    if lstShops.Items[i] = strShopName then
      begin
        lstShops.Selected[i] := True;
        Exit;
      end;
  end;
end;
```

Некоторая сложность алгоритма загрузки списка выбранных магазинов обусловлена желанием добиться того, чтобы при изменении списка не выделялись ранее не выбранные магазины (иначе можно было бы просто сохранять индексы).



ПРИМЕЧАНИЕ

Чтобы при первом запуске процедуры LoadFilter не появлялись сообщения об исключениях (при работе в отладчике Delphi), снимите флажок Stop on Delphi Exceptions на вкладке Language Exceptions диалогового окна Debugger Options (меню Tools ▶ Debugger Options).

Автозапуск программ

Так уж повелось, что, рассматривая работу с реестром, редко удается удержаться от рассказа, как можно организовать автоматический запуск приложений, минуя

пресловутое меню **Автозагрузка**. Коснемся этой темы и мы: рассмотрим наиболее простые способы автоматического запуска не сервисных (!) программ.

Итак, в ветвях реестра `HKEY_CURRENT_USER` и `HKEY_LOCAL_MACHINE` находятся разделы `Software\Microsoft\Windows\CurrentVersion\Run` и `Software\Microsoft\Windows\CurrentVersion\RunOnce`. В первом (`Run`) сохраняются пути приложений, запускаемых при каждой загрузке `Windows`. В разделе же `RunOnce` обычно регистрируются приложения типа инсталляторов, которые запускаются при первой с момента регистрации перезагрузке `Windows`, но до запуска программы **Проводник**. При запуске приложения, зарегистрированного в ключе `RunOnce`, соответствующая запись из этого раздела автоматически удаляется.

От выбора ветви реестра (`HKEY_LOCAL_MACHINE` или `HKEY_CURRENT_USER`) зависит, в сеансе всех ли пользователей будет запускаться приложение.

Рассмотрим создание простейшей программы, способной определить, запускается ли она автоматически, а если запускается, то каким образом. Программа также будет уметь создавать и удалять параметры в нужных разделах реестра для задания нужного режима запуска.

Пусть на форме приложения расположены три переключателя (рис. 7.9). Процедура, приведенная в листинге 7.16, устанавливает состояния переключателей в зависимости от того, в каком разделе ветви `HKEY_LOCAL_MACHINE` расположен параметр с именем, совпадающим с именем программы (это условность, которая нужна для работы нашего примера).

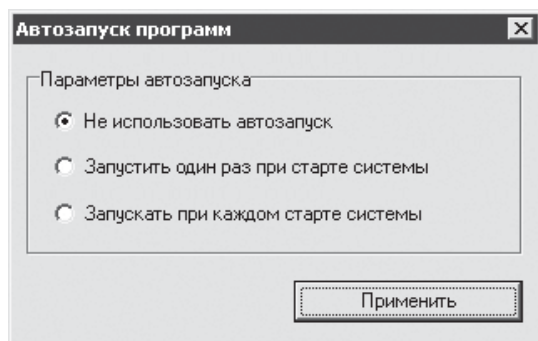


Рис. 7.9. Форма после определения варианта режима запуска приложения

Листинг 7.16. Определение режима запуска приложения

```
procedure TForm1.GetRunMode();
var
  reg: TRegistry;
begin
  reg := TRegistry.Create();
  reg.RootKey := HKEY_LOCAL_MACHINE;
```

```
//Определение, как запускается программа (по наличию значений
//в соответствующих разделах)
if reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\Run',
False)
then
begin
    if reg.ValueExists(Application.Title) then
    begin
        //Программа есть в разделе Run –
        //запускается при каждой загрузке Windows
        optAutoRun.Checked := True;
        reg.CloseKey();
        Exit;
    end;
    reg.CloseKey();
end;
if reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\
RunOnce', False)
then
begin
    if reg.ValueExists(Application.Title) then
    begin
        //Программа есть в разделе RunOnce –
        //запускается один раз при старте Windows
        optRunOnce.Checked := True;
        reg.CloseKey();
        Exit;
    end;
    reg.CloseKey();
end;
//Автозапуск программы (рассматриваемым способом) не включен
optRunNone.Checked := True;
reg.Free();
end;
```

Параметры запуска изменяются (в рассматриваемом приложении) при нажатии кнопки Применить (листинг 7.17).

Листинг 7.17. Применение режима запуска

```
procedure TForm1.cmbApplyClick(Sender: TObject);
var
    reg: TRegistry;
```

```
begin
  reg := TRegistry.Create();
  reg.RootKey := HKEY_LOCAL_MACHINE;
  //Отмена прошлого режима
  //..удаление параметра из раздела Run
  if not optAutoRun.Checked then
    if reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\
Run', False)
    then
      begin
        reg.DeleteValue( Application.Title );
        reg.CloseKey();
      end;
    //..удаление параметра из раздела RunOnce
    if not optRunOnce.Checked then
      if reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\
RunOnce', False)
      then
        begin
          reg.DeleteValue( Application.Title );
          reg.CloseKey();
        end;
      //Установка нового режима (создание параметра в соответствующем
      //разделе)
      if optAutoRun.Checked then
        //..добавление параметра в раздел Run
        if reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\
Run', True)
        then
          begin
            reg.WriteString( Application.Title, Application.ExeName);
            reg.CloseKey();
          end;
        if optRunOnce.Checked then
          //..добавление параметра в раздел RunOnce
          if reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\
RunOnce', True)
          then
            begin
              reg.WriteString( Application.Title, Application.ExeName);
              reg.CloseKey();
            end;
          end;
```

```
//Для верности обновим показания на форме по данным из реестра
GetRunMode();
reg.Free();
end;
```

При желании вы можете изменить ветвь реестра на `HKEY_CURRENT_USER`, если приложение (которое вы будете делать) запускалось только для определенных пользователей.

Запуск приложения из командной строки

Сразу оговоримся, что из командной строки (например, из окна **Запуск программы**, открываемого командой **Пуск ▶ Выполнить**) можно запустить любое приложение: достаточно только ввести его полный или относительный (относительно рабочей папки) путь. Однако, возможно, вы замечали, что некоторые приложения можно запускать, просто вводя в командной строке имя приложения, например `msaccess` или `winword`. Займемся обеспечением возможности запуска приложения таким ускоренным способом.

Чтобы зарегистрировать приложение для быстрого запуска, можно поместить его путь в ветвь реестра `SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths` корневого раздела `HKEY_CURRENT_USER` или `HKEY_LOCAL_MACHINE`. Путь EXE-файла приложения должен быть записан в параметр (По умолчанию) подраздела, имеющего такое же имя, как и EXE-файл приложения (включая расширение).

Пример процедуры, регистрирующей приложение для быстрого запуска, приведен в листинге 7.18.

Листинг 7.18. Регистрация приложения для запуска из командной строки

```
procedure RegisterQuickStart();
var
  reg: TRegistry;
begin
  reg := TRegistry.Create();
  reg.RootKey := HKEY_LOCAL_MACHINE;
  //Регистрируем программу для запуска по имени из
  //командной строки
  if reg.OpenKey(paths + '\' + Application.Title + '.exe', True)
  then
    begin
      reg.WriteString('', Application.ExeName);
      reg.CloseKey();
    end;
  reg.Free();
end;
```

Для отмены быстрого запуска приложения из командной строки можно воспользоваться процедурой, приведенной в листинге 7.19.

Листинг 7.19. Отмена быстрого запуска приложения

```
procedure UnregisterQuickStart();
var
  reg: TRegistry;
begin
  reg := TRegistry.Create();
  reg.RootKey := HKEY_LOCAL_MACHINE;
  //Удаляем сведения о программе из реестра
  reg.DeleteKey(paths + '\' + Application.Title + '.exe');
  reg.Free();
end;
```

В приведенных выше листингах значение константы `paths` равно:

```
const paths = 'SOFTWARE\Microsoft\Windows\CurrentVersion\App
Paths';
```

Регистрация типов файлов

Теперь рассмотрим вопрос, нередко интересующий программистов, приложения которых должны уметь сохранять и загружать данные из файлов. Логично задать всем таким файлам одно расширение: получается тип файлов приложения.

Открытие файлов (документов) приложения из самого приложения организовать несложно: достаточно применить диалог открытия файла. Но как заставить, например, Проводник автоматически запускать наше приложение при выборе соответствующего файла? Сделать это тоже несложно: достаточно внести небольшие изменения в раздел реестра `HKEY_CLASSES_ROOT`.

Итак, перечень операций, которые нужно произвести для регистрации собственного типа файла (пусть, `MYDOC`).

1. Создать раздел `HKEY_CLASSES_ROOT\.mydoc`, в параметр (По умолчанию) которого записать имя типа файла, например `TricksDelphi.DocumentSample`.
2. Создать раздел `HKEY_CLASSES_ROOT\<имя_типа>`, например `HKEY_CLASSES_ROOT\TricksDelphi.DocumentSample`. Если в параметр (По умолчанию) этого раздела записать строку, то она будет отображаться в качестве описания типа файла.
3. Если нужно, чтобы для документа использовался определенный значок, необходимо создать раздел `HKEY_CLASSES_ROOT\<имя_типа>\DefaultIcon`, в параметр (По умолчанию) которого записать полный путь EXE- или DLL-файла, из которого брать значок, и через запятую — номер значка (см. гл. 4).
4. Наконец, для автоматического запуска приложения при выборе файла заданного типа создаем раздел `HKEY_CLASSES_ROOT\<имя_типа>\Shell\Open\Command`,

в параметр (По умолчанию) которого записываем строку вида <путь_приложения> %1 для передачи имени документа в командной строке.

Пример процедуры, которая производит все вышеперечисленные манипуляции, приводится в листинге 7.20.

Листинг 7.20. Регистрация типа файла

```
procedure RegisterAppDocuments();
var
    reg: TRegistry;
begin
    reg := TRegistry.Create();
    reg.RootKey := HKEY_CLASSES_ROOT;
    //Вносим информацию о нашем типе файла в реестр
    //..само расширение
    if reg.OpenKey('.mydoc', True) then
    begin
        reg.WriteString('', 'TricksDelphi.DocumentSample');
        reg.CloseKey();
    end;
    //..описание типа файла
    if reg.OpenKey('TricksDelphi.DocumentSample', True) then
    begin
        reg.WriteString('', 'Документ TricksDelphi.DocumentSample');
        reg.CloseKey();
    end;
    //..значок для файлов MYDOC-типа
    if reg.OpenKey('TricksDelphi.DocumentSample\DefaultIcon', True)
    then
    begin
        reg.WriteString('', Application.ExeName + ', 1');
        reg.CloseKey();
    end;
    //..приложение, открывающее MYDOC-документ
    if reg.OpenKey('TricksDelphi.DocumentSample\Shell\Open\Command',
    True)
    then
    begin
        reg.WriteString('', Application.ExeName + ' %1');
        reg.CloseKey();
    end;
```

```

end;
reg.Free();
end;

```

Результат работы этой процедуры показан на рис. 7.10.

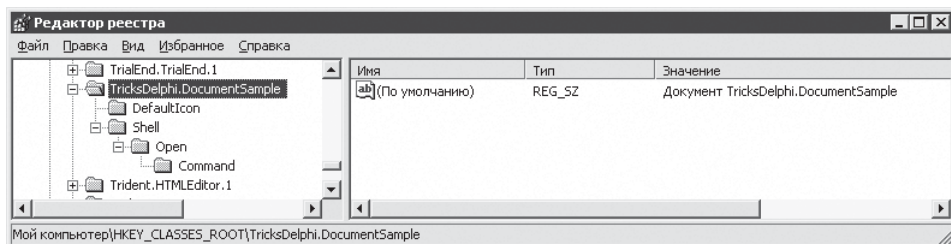


Рис. 7.10. Результат регистрации типа файла

Теперь при выборе в файловой оболочке наше приложение запускается с путем выбранного файла (правда, в формате 8.3) в качестве аргумента командной строки. Как перевести путь из короткой формы в длинную (если это вообще надо), рассказано в разд. 4.2. Если вы не знакомы с тем, как получать доступ к аргументам командной строки, можете взглянуть на листинг 7.21 (тут происходит отображение имени открываемого файла в текстовом поле на форме).

Листинг 7.21. Определение имени открываемого файла

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    if ParamCount() > 0 then
    begin
        //Обрабатываем данные командной строки...
        txtDoc.Text := 'Имя открываемого файла: ' + ParamStr(1);
    end;
end;

```

Уничтожение сведений о типе файла возможно путем простого удаления созданных ранее разделов, например так, как сделано в листинге 7.22.

Листинг 7.22. Удаление из реестра сведений о типе файла

```

procedure UnregisterAppDocuments();
var
    reg: TRegistry;
begin
    reg := TRegistry.Create();
    reg.RootKey := HKEY_CLASSES_ROOT;

```

```
//Удаление из реестра информации о типе файла  
reg.DeleteKey('.mydoc');  
reg.DeleteKey('TricksDelphi.DocumentSample');  
reg.Free();  
end;
```

Программа для просмотра реестра

Для демонстрации некоторых других приемов работы с реестром, например перемещение по иерархии разделов реестра, определение списка параметров, их типа и значений), рассмотрим реализацию приложения, предоставляющего соответствующие возможности.

В итоге у нас получится такая альтернатива программе Редактор реестра, правда, пригодная только для просмотра, но не для редактирования реестра. Главная форма программы выглядит так, как показано на рис. 7.11.

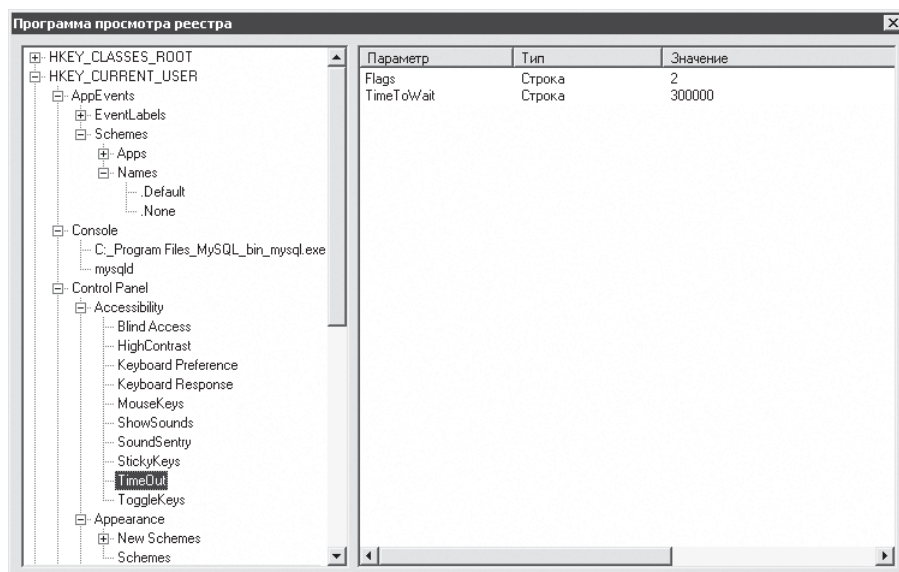


Рис. 7.11. Программа для просмотра реестра

Рассмотрим функции и процедуры, формирующие основу этого приложения, в порядке их использования. Итак, при запуске формы составляется список корневых разделов реестра (листинг 7.23).

Листинг 7.23. Первоначальная инициализация дерева разделов реестра

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
    item: TTreeNode;
```

```
begin
  //Формирование списка корневых разделов реестра
  item := keys.Items.AddChild(nil, 'HKEY_CLASSES_ROOT');
  item.Data := Pointer(HKEY_CLASSES_ROOT);
  CheckSubKeys(item);

  item := keys.Items.AddChild(nil, 'HKEY_CURRENT_USER');
  item.Data := Pointer(HKEY_CURRENT_USER);
  CheckSubKeys(item);

  item := keys.Items.AddChild(nil, 'HKEY_LOCAL_MACHINE');
  item.Data := Pointer(HKEY_LOCAL_MACHINE);
  CheckSubKeys(item);

  item := keys.Items.AddChild(nil, 'HKEY_USERS');
  item.Data := Pointer(HKEY_USERS);
  CheckSubKeys(item);

  item := keys.Items.AddChild(nil, 'HKEY_CURRENT_CONFIG');
  item.Data := Pointer(HKEY_CURRENT_CONFIG);
  CheckSubKeys(item);
end;
```

Процедура CheckSubKeys, вызываемая для каждого нового элемента дерева (листинг 7.23), реализована следующим образом (листинг 7.24).

Листинг 7.24. Оформление элемента дерева в зависимости от наличия вложенных разделов

```
procedure TForm1.CheckSubKeys(item: TTreeNode);
var
  reg: TRegistry;
begin
  reg := TRegistry.Create();
  //Проверка, есть ли в разделе реестра вложенные подразделы
  reg.RootKey := GetRootKey(item);
  if reg.OpenKeyReadOnly(GetKeyPath(item)) then
  begin
    if reg.HasSubKeys() then
    begin
```

```
//Добавляем фиктивный элемент (чтобы показывался "+" для
//разворачивания раздела). Одновременно помечаем
//фиктивный элемент
keys.Items.AddChild(item, '').Data := Pointer(-1);
end;
reg.CloseKey();
end;
reg.Free();
end;
```

По сравнению с примером (дерево каталогов), рассмотренным в подразд. «Построение дерева каталогов» разд. 4.2, определение наличия дочерних разделов реестра — относительно легковесная операция, поэтому эту проверку производим сразу при составлении списка подразделов. Как и в только что упомянутом примере из гл. 4, мы добавляем в дерево фиктивный дочерний элемент для тех элементов дерева, для которых соответствующие им разделы реестра содержат подразделы.

Важно то, что фиктивный элемент помечается значением `-1`. Как раз по наличию дочернего элемента с полем `Data`, равным `-1`, можно определить, зачитывалось ли содержимое раздела, соответствующего определенному элементу дерева. Содержимое раздела читается при разворачивании элемента дерева (листинг 7.25).

Листинг 7.25. Составление списка дочерних разделов

```
procedure TForm1.keysExpanding(Sender: TObject; Node: TTreeNode;
  var AllowExpansion: Boolean);
var
  reg: TRegistry;
  subkeys: TStringList;
  i: Integer;
begin
  if Integer(Node.getFirstChild.Data) <> -1 then
    //Список подразделов был зачитан ранее
    Exit;
  Node.DeleteChildren(); //Удаление фиктивного элемента дерева
  reg := TRegistry.Create();
  //Загрузка списка подразделов выбранного раздела
  reg.RootKey := GetRootKey(Node);
  if reg.OpenKey(GetKeyPath(Node), False) then
    begin
      //Получение списка подразделов
      subkeys := TStringList.Create();
      reg.GetKeyNames(subkeys);
      for i := 0 to subkeys.Count - 1 do
```

```

begin
    //Добавление элемента для дочернего раздела (не забываем
    //проверять подразделы у каждого дочернего раздела)
    CheckSubKeys(keys.Items.AddChild(Node, subkeys[i]));
end;
subkeys.Free();
reg.CloseKey();
end;
reg.Free();
end;

```

В листинге 7.25 используются две дополнительные функции: для определения полного пути раздела, соответствующего элементу дерева (без имени корневого раздела), и для получения дескриптора корневого раздела (хранится в поле `Data` корневого элемента каждой ветви дерева).

Путь раздела определить несложно: просто поднимаемся к корню соответствующей верши дерева, собирая по ходу имена элементов дерева (листинг 7.26).

Листинг 7.26. Определение пути раздела в дереве

```

function GetKeyPath(item: TTreeNode): String;
var
    temp: TTreeNode;
    path: String;
begin
    temp := item;
    while temp.Parent <> nil do
    begin
        path := temp.Text + '\' + path;
        temp := temp.Parent;
    end;
    GetKeyPath := path;
end;

```

Аналогичным образом, даже проще, определяется дескриптор корневого раздела определенной ветви реестра: для этого нужно просто добраться до корня ветви дерева и прочитать значение поля `Data` корневого элемента (листинг 7.27).

Листинг 7.27. Определение дескриптора корневого раздела ветви

```

function GetRootKey(item: TTreeNode): HKEY;
var
    temp: TTreeNode;
begin

```

```
temp := item;
while temp.Parent <> nil do
    temp := temp.Parent;
    GetRootKey := HKEY(temp.Data);
end;
```

При выделении элемента дерева происходит отображение параметров соответствующего раздела в списке в правой части формы. Как заполнять список, представлено в листинге 7.28.

Листинг 7.28. Составление списка параметров раздела реестра

```

procedure TForm1.KeysChange(Sender: TObject; Node: TTreeNode);
var
    reg: TRegistry;
    valueItem: TListItem;
    item: TTreeNode;
    valueNames: TStringList;
    i: Integer;
begin
    item := keys.Selected;
    if item <> nil then
    begin
        //Зачитаем содержимое выбранного раздела в ListView (values)
        values.Clear;
        reg := TRegistry.Create();
        reg.RootKey := GetRootKey(item);
        if reg.OpenKeyReadOnly(GetKeyPath(item)) then
        begin
            valueNames := TStringList.Create();
            //Получение списка названий параметров
            reg.GetValueNames(valueNames);
            //Добавление каждого параметра в список
            for i := 0 to valueNames.Count - 1 do
            begin
                valueItem := values.Items.Add();
                if valueNames[i] = '' then
                    valueItem.Caption := '<По умолчанию>'
                else
                    valueItem.Caption := valueNames[i];
            end
        end
    end
end

```

```
//Получение типа и значения параметра
case reg.GetDataType(valueNames[i]) of
  rdUnknown:
    valueItem.SubItems.Add('Неизвестно');
  rdString, rdExpandString:
    begin
      valueItem.SubItems.Add('Строка');
      valueItem.SubItems.Add(reg.ReadString(valueNames[i]));
    end;
  rdInteger:
    begin
      valueItem.SubItems.Add('Число');
      valueItem.SubItems.Add(IntToStr(
        reg.ReadInteger(valueNames[i])));
    end;
  rdBinary:
    valueItem.SubItems.Add('Двоичные данные');
end;
end;
valueNames.Free();
reg.CloseKey();
end;

reg.Free();
end;
end;
```

Процедура, приведенная в листинге 7.28, не считывает значения двоичных параметров. Так сделано для упрощения этого и так громоздкого фрагмента кода. В считывании значений двоичных параметров на самом деле нет ничего сложного: нужно лишь заранее определить размер данных (метод `GetDataSize`) и создать буфер соответствующего размера.



Глава 8

Обмен данными между приложениями

- ☐ Сообщение WM_COPYDATA
- ☐ Использование буфера обмена
- ☐ Проецируемые в память файлы

Организация обмена данными между приложениями, а именно между процессами этих приложений, является достаточно трудоемкой задачей. Архитектура Win32 подразумевает максимальную изоляцию выполняющихся приложений друг от друга. Каждое приложение выполняется в своем виртуальном адресном пространстве, которое изолировано и не имеет доступа к памяти других процессов приложений. Но довольно часто возникает необходимость передачи данных из одного выполняющегося процесса в другой. Это вызвано тем, что функциональные приложения и пакеты программ исполняются не в одном процессе, поэтому для нормальной работы используются основные возможности межпроцессного взаимодействия. Наиболее простым, понятным, но не всегда удобным является передача данных с использованием сообщения WM_COPYDATA. Также для передачи данных между приложениями широко используются проецируемые в память файлы (Mapping Files). Существуют и такие высокоуровневые средства, как буфер обмена или уже рассмотренная технология СОМ. Перечисленные способы будут подробно рассматриваться в этой главе. За рамки этой книги выходит рассмотрение способа передачи данных через каналы (трубы, или Pipe), который считается устаревшим и по этой причине не вызывает интереса.

8.1. Сообщение WM_COPYDATA

Сообщение WM_COPYDATA позволяет приложениям копировать данные между их адресными пространствами. Для передачи сообщения должна использоваться функция синхронной отправки сообщения SendMessage, а не PostMessage, которая асинхронным образом передает сообщение. Данные, предназначенные для передачи, не должны содержать указателей или других ссылок на объекты, недоступные для программы, принимающей эти данные. Рассмотрим параметры, передаваемые с сообщением WM_COPYDATA:

```
//дескриптор передающего окна  
wParam = (WPARAM) (HWND) hwnd;  
//указатель на структуру с данными  
lParam = (LPARAM) (PCOPYDATASTRUCT) pcds;
```

На использование сообщения налагаются следующие ограничения:

- данные, которые будут приняты, должны быть только для чтения, так как изменение структуры с данными может привести к непредсказуемым последствиям;
- если приложению, получающему данные, требуется использовать их после возврата из обработчика WM_COPYDATA, оно должно скопировать их в локальный буфер.

Итак, приступим к созданию приложения, демонстрирующего работу WM_COPYDATA. Для создания хорошего примера потребуется создать два приложения. Первое будет отправлять данные (например, строку текста), другое приложение будет их получать. На главной форме первого приложения помещаем элемент управления TextBox, в который будет записываться передаваемая строка, и кнопку, нажатие которой инициирует передачу данных. Для второго приложения достаточно элемента для отображения

текстовой информации типа `Label`. Перейдем к рассмотрению исходных текстов созданных приложений.

Мы будем посылать сообщение окну, и сообщений может быть различное количество, поэтому для уникальной идентификации операции введем специальную константу:

```
const
```

```
    CMD_SETLABELTEXT = 1; // Задаем ID команды
```

На форме находится кнопка отправки данных другому приложению, ее обработчик выглядит следующим образом (листинг 8.1).

Листинг 8.1. Отправка данных другому приложению

```
procedure TDataSender.bnSendClick(Sender: TObject);
var
    CDS: TCopyDataStruct;
begin
    //Устанавливаем тип команды
    CDS.dwData := CMD_SETLABELTEXT;
    //Устанавливаем длину передаваемых данных
    CDS.cbData := Length(StringEdit.Text) + 1;
    //Выделяем память буфера для передачи данных
    GetMem(CDS.lpData, CDS.cbData);
    try
        //Копируем данные в буфер
        StrPCopy(CDS.lpData, StringEdit.Text);
        // Отсылаем сообщение в окно с заголовком StringReciever
        SendMessage(FindWindow(NIL, 'StringReciever'),
            WM_COPYDATA, Handle, Integer(@CDS));
    finally
        //Высвобождаем буфер
        FreeMem(CDS.lpData, CDS.cbData);
    end;
end;
```

Подробного комментария данный листинг не требует. Обратите лишь внимание на вызов функции `SendMessage`, которая использует `FindWindow` для задания одного из своих параметров. Процедура `FindWindow` в случае успешного выполнения возвращает `HWND` окна, заголовок которого задается в параметре этой функции (строка `StringReciever` из предыдущего примера). Синхронная отправка сообщения `WM_COPYDATA` с набором данных, которые помещены в структуру `CDS`, осуществляется вызовом `SendMessage`.

Рассмотрим второе приложение, которое принимает строку и отображает ее в надписи (Label). Для начала в блок объявления помещаем обработчик сообщения и объявляем само сообщение WM_COPYDATA:

```
type
  TStringReciever = class(TForm)
    LabelStr: TLabel;
private
  //Обработчик сообщения WM_COPYDATA
  procedure WMCopyData(var MessageData: TWMCopyData);
  message WM_COPYDATA;
```

Как и в случае первого приложения, нам необходима константа, которая будет идентифицировать тип операции:

```
const
  CMD_SETLABELTEXT = 1;
```

Далее рассмотрим тело функции обработчика сообщения WM_COPYDATA (листинг 8.2).

Листинг 8.2. Обработка сообщения WM_COPYDATA

```
procedure TStringReciever.WMCopyData(var MessageData: TWMCopyData);
begin
  //Устанавливаем свойства метки, если заданная команда совпадает
  if MessageData.CopyDataStruct.dwData = CMD_SETLABELTEXT then
  begin
    //Устанавливаем текст из полученных данных
    LabelStr.Caption := PChar(MessageData.CopyDataStruct.lpData);
    MessageData.Result := 1;
  end else
    MessageData.Result := 0;
end;
```

Если окну второго приложения, которое носит название **StringReciver** (получатель строки), приходит сообщение WM_COPYDATA, то происходит вызов WMCopyData. В качестве параметра эта процедура получает структуру данных MessageData типа TWMCopyData, содержащую идентификатор операции и данные (передаваемую строку). После проверки типа операции в случае совпадения его с константой CMD_SETLABELTEXT полученные данные преобразуются в строку. Преобразование происходит при помощи функции PChar. Полученная строка устанавливается в качестве заголовка для метки с именем LabelStr. Затем полю Result структуры MessageData присваивается значение 1 или 0, в зависимости от успеха операции.

Таким образом, для передачи данных (строки) записываем передаваемую строку в текстовое поле первой формы и нажимаем кнопку **Отправить**. Результат работы приложений можно увидеть на рис. 8.1.

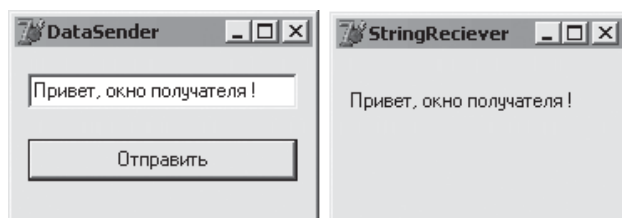


Рис. 8.1. Вид приложений отправки и получения строки

Необходимо добавить, что передача данных посредством сообщения `WM_COPYDATA` является удобным и простым способом. Но `WM_COPYDATA` можно передавать только функцией `SendMessage`, и это является существенным недостатком такого метода. `SendMessage` «замораживает» работу приложения-отправителя, поэтому такой способ применяется для передачи небольших объемов данных, которые не требуют сложной обработки на стороне программы-приемника. К тому же на использование `WM_COPYDATA` налагаются некоторые существенные ограничения, о которых говорилось выше.

8.2. Использование буфера обмена

Буфер обмена представляет собой область оперативной памяти, которая используется операционной системой для временного хранения данных. Он выступает в роли общего хранилища данных для всех приложений системы, фактически любая программа может записывать данные в буфер обмена и считывать их оттуда. Он способен хранить данные различных типов и, кроме данных, содержит сведения о типе хранимой информации. Буфер обмена является неотъемлемым компонентом операционной системы типа Windows.

Буфер обеспечивает простейший обмен данными между приложениями. Одно приложение помещает туда данные, другое — считывает данные из буфера. Как правило, эти действия (чтение и запись) выполняются при непосредственном участии пользователя. В использовании буфера может участвовать и одно приложение, в этом случае проходит обмен данными внутри его.

Для выполнения операции обмена данными через буфер в Delphi предназначен специальный класс `TClipboard`. В Delphi также имеется глобальный объект `Clipboard`, который является экземпляром класса `TClipboard` и представляет буфер обмена Windows.

При помощи свойств и методов объекта `Clipboard` возможно осуществление различных операций обмена или анализа хранимых данных. Для доступа к объекту буфера в разделе `uses` модуля, в котором выполняются операции с объектом буфера обмена, указывается модуль `Clipboard`.

Общее количество форматов, поддерживаемых буфером обмена, содержится в свойстве `FormatCount` типа `Integer`. Для отображения количества форматов, которые распознает буфер, можно использовать следующий листинг:

```
//В разделе uses указываем модуль Clipboard  
InformationClipLabel.Caption := IntToString(Clipboard.FormatCount);
```

Буфер обмена поддерживает самые разнообразные типы данных. Приведем список поименованных констант некоторых форматов.

- `CF_TEXT` — обычный текст (коды ANSI). Символ окончания строки — `#10` и `#13`, окончание текста — `#0`.
- `CF_BITMAP` — рисунок BMP-формата.
- `CF_MetaFilePic` — рисунок WMF-формата.
- `CF_TIFF` — рисунок TIFF-формата.
- `CF_OEMTEXT` — текст.
- `CF_DIB` — рисунок DIB-формата.
- `CF_Wave` — звук.
- `CF_UnicodeText` — текст (коды Unicode).
- `CF_Picture` — объект типа `TPicture`.

При необходимости можно создать и зарегистрировать свои форматы данных в дополнение к имеющимся базовым.

При использовании нестандартных форматов данных, помещаемых в буфер обмена и извлекаемых оттуда, программы должны соблюдать устанавливаемые разработчиками соглашения об обмене такими данными.

В листинге 8.3 приводится обработчик нажатия кнопки, загружающий в список `ListBoxInfo` значения констант, идентифицирующих каждый формат данных буфера обмена.

Листинг 8.3. Отображение значений форматов буфера

```
procedure TFormClipboard.bnInfoCipClick(Sender: TObject);  
var i: integer;  
begin  
    ListBoxInfo.Clear;  
    for i := 0 to Clipboard.FormatCount - 1 do  
        ListBoxInfo.Items.Add(IntToStr(Clipboard.Formats[i]));  
end;
```

Приложение может помещать информацию в буфер обмена и извлекать ее только в тех форматах, которые будет поддерживать буфер. Список поддерживаемых форматов создается при инициализации приложения.

Перед доступом к данным, содержащимся объектом `Clipboard`, может потребоваться анализ формата данных, для этого служит метод `HasFormat`. Процедура `HasFormat (Format: Word): Boolean` используется для запроса к буферу обмена и позволяет узнать, можно ли извлечь хранимые в нем данные в заданном формате, указанном параметром `Format`. При положительном ответе возвращаемое значение равно `True`, в противном случае — `False`.

Как правило, различные приложения используют буфер обмена. Но в случае, когда необходимо получить монополярный доступ к буферу, приложение должно открыть его для себя в специальном режиме. Для этого вызывается метод `Open`, позволяющий программе получить полный (исключительный) доступ к общей области обмена. После вызова метода `Open` содержимое буфера не может быть изменено другими приложениями, поэтому после окончания монополярного использования приложение должно вызвать метод `Close` объекта `Clipboard`. Если открытый буфер не был закрыт с помощью метода `Close`, то он будет автоматически закрыт системой после завершения программы, открывшей буфер обмена.

Для очистки содержимого буфера обмена используется метод `Clear`. Он вызывается автоматически при изменении содержимого в буфере. Перед записью новых данных, помещаемых в буфер, старая информация удаляется.

Класс `TClipboard` используется многими другими классами и компонентами, которые поддерживают обмен данными через буфер обмена. К примеру, компоненты `Мето` и `Edit` имеют специальные методы для обмена текстовой информацией посредством буфера. Методы `CopyToClipboard` и `CutToClipboard` помещают текстовые данные в буфер обмена, копируя и вырезая их из источника (компонента) соответственно, а метод `PasteFromClipboard` вставляет текстовый фрагмент из буфера в текстовое поле.

Только при использовании подобных методов лучше проверять, является ли содержимое буфера обмена текстовой информацией. В листинге 8.4 показан пример копирования в буфер обмена всего текста, введенного в текстовое поле.

Листинг 8.4. Копирование текста из поля редактора Мето в буфер обмена

```
procedure TFormClipboard.bnCopyTextClick(Sender: TObject);
begin
    //Выделяем весь текст в поле редактора
    MemoText.SelectAll;
    //Копируем текст
    MemoText.CopyToClipboard;
end;
```

Буфер обмена часто используется для хранения текста, поэтому объект `Clipboard` имеет специальное свойство `AsText` типа `String`, предназначенное для обработки содержимого буфера как данных текстового формата. Свойство `AsText` предназначено как для чтения, так и для записи. При чтении свойства данные извлекаются из буфера, а при записи — заносятся в буфер обмена (листинг 8.5).

Листинг 8.5. Копирование текстовой информации

```
procedure TFormClipboard.bnCopyTextAsTextClick(Sender: TObject);
begin
    //Если в буфере текст, то выводим его в поле редактора
    if Clipboard.HasFormat(CF_Text)
        then MemoText := Clipboard.AsText
    end;
```

При работе с графическими компонентами для операций, связанных с обменом информацией через общую область, удобно использовать метод `Assign`. Процедура `Assign (Source:TPersistent)` присваивает буферу обмена объект, указанный параметром `Source`. Если объект является изображением и принадлежит таким графическим классам, как `TBitmap`, `TPicture` или `TMetafile`, то в буфер обмена копируется изображение установленного формата. Для извлечения изображения также может использоваться метод `Assign`.

Пример использования буфера обмена для копирования изображений проводится в листинге 8.6.

Листинг 8.6. Обмен изображением через буфер обмена

```
procedure TFormClipboard.bnCopyImageClick(Sender: TObject);
begin
    //Открываем монопольный доступ
    Clipboard.Open;
    //Заносим изображение в буфер
    Clipboard.Assign(ImageMyPic1.Picture);
    //Проверяем формат находящихся в буфере данных
    if Clipboard.HasFormat(CF_Picture)
        then ImageMyPic2.Picture.Assign(Clipboard);
    //Закрываем монопольный доступ к буферу
    Clipboard.Close;
end;
```

Изображение, находящееся в образе `ImageMyPic1`, помещается в буфер обмена, откуда затем копируется в образ `ImageMyPic2`. Для выполнения этих операций устанавливается монопольный доступ к объекту `Clipboard`.

Таким образом, использование объекта `Clipboard` находит широкое применение в программировании приложений, которым необходим обмен данными с другими программами. Необходимо отметить, что буфер обмена ориентирован на работу с пользователем (пользователь инициирует обмен данными между приложениями), поэтому такой способ обмена данными наиболее удобен с точки зрения пользователя. К тому же буфер обмена поддерживает множество форматов представления информации, что позволяет сделать обмен данными более гибким и эффективным.

8.3. Проецируемые в память файлы

Не менее мощным и гибким методом организации обмена данными между приложениями является метод, который базируется на проецируемых в память файлах (Files Mapping). Главная идея этого механизма основывается на использовании динамической разделяемой памяти системы для хранения в ней данных. Как известно, каждый процесс имеет свой участок памяти, называемый виртуальным адресным пространством. При использовании механизма проецируемых в память файлов данные становятся доступны из любого процесса, который использует этот файл. В этом случае говорят, что файл отображается в виртуальное адресное пространство процесса, поэтому данные, хранимые в файле, доступны процессу, который этот файл открыл. Механизм проецирования файлов в память используется, например, для исполняемых файлов приложений, а также для DLL.

Для работы с проецируемыми в память файлами существует целый ряд API-функций. Но прежде чем их рассматривать, разберемся в процессе организации обмена данными через проецируемые файлы. На первом этапе необходимо создать объект (файл, отображаемый в память), затем «отобразить» созданный объект в адресное пространство процесса приложения, получая возможность записи и чтения данных из этого файла. При отображении файла на определенный участок памяти (адресного пространства процесса) манипуляции с данными этого участка памяти отражаются на содержимом файла. После произведенных над объектом манипуляций необходимо закрыть доступ к данным файла (удалить проекцию и закрыть файл).

Рассмотрим некоторые функции для работы с проецируемым в память файлом. Для того чтобы создать объект файла, проецируемого в память, можно использовать функцию `CreateFileMapping`. Ее синтаксис выглядит следующим образом:

```
function CreateFileMapping(hFile: THandle;  
    lpFileMappingAttributes: PSecurityAttributes;  
    flProtect, dwMaximumSizeHigh, dwMaximumSizeLow: DWORD;  
    lpName: PChar ): THandle;
```

Подробнее рассмотрим параметры функции.

- `hFile` — идентификатор файла. В результате присвоения этому аргументу значения константы `INVALID_HANDLE_VALUE` мы свяжем создаваемый объект файлового отображения со страничным swap-файлом (системным файлом подкачки).
- `lpFileMappingAttributes` — указатель на структуру типа `TSecurityAttributes`. Структура содержит параметры безопасности создаваемого файла.
- `flProtect` — параметр, задающий способ совместного использования создаваемого объекта, в случае доступа на чтение и запись принимает значение `PAGE_READWRITE`.
- `dwMaximumSizeHigh` — старший разряд 64-битного значения размера выделяемого объема памяти.

- `dwMaximumSizeLow` — младший разряд 64-битного значения размера выделяемого объема памяти.
- `lpName` — имя объекта проецируемого файла (может быть `nil` для создания безымянной проекции файла).

Функция возвращает глобальный дескриптор (`THandle`). Если проецируемый файл не создан, то функция `CreateFileMapping` возвращает нулевое значение.

После того как проецируемый файл был создан, необходимо отобразить его в адресное пространство процесса. Для этого предназначена функция `MapViewOfFile`, имеющая следующий синтаксис:

```
function MapViewOfFile(hFileMappingObject: THandle;  
                      dwDesiredAccess: DWORD;  
                      dwFileOffsetHigh, dwFileOffsetLow,  
                      dwNumberOfBytesToMap: DWORD ): Pointer;
```

Функция имеет следующие параметры.

- `hFileMappingObject` — описатель созданного объекта файлового отображения.
- `dwDesiredAccess` — параметр доступа к полученным данным, в случае чтения и записи принимает значение `FILE_MAP_WRITE`.
- `dwFileOffsetHigh, dwFileOffsetLow` — 64-битное смещение от начала файла.
- `dwNumberOfBytesToMap` — указывает, сколько байт будет отображено. Если этот аргумент имеет значение `0`, то на область памяти будет отображен весь файл.

В результате успешного выполнения функции `MapViewOfFile` будет получен указатель (тип `Pointer`) на начальный адрес данных объекта. Указатель будет использоваться в дальнейшем для записи или чтения файла.

Следующей функцией, противоположной по производимым действиям функции `MapViewOfFile`, является `UnMapViewOfFile`. Она отключает проецируемый файл от текущего процесса:

```
function UnMapViewOfFile(lpBaseAddress: Pointer): Boolean;
```

Функция принимает указатель, возвращаемый `MapViewOfFile`, и использует его для отмены проекции файла на адресное пространство процесса. В случае успешной выгрузки функция возвращает `True`, в противном случае — `False`.

И последняя функция, которую необходимо рассмотреть, — это `CloseHandle`. Она используется для закрытия дескриптора (многих системных объектов, а не только проекции файла).

```
function CloseHandle(hFileMapObj:THandle):Boolean;
```

Как видно из синтаксиса функции, она принимает описатель объекта файлового отображения, полученный в результате выполнения функции `CreateFileMapping` и освобождает его. Для правильного завершения работы с объектом файлового отображения сначала следует применить функцию `UnMapViewOfFile`, а затем `CloseHandle`.

Сама проекция файла будет удалена только после того, как будут закрыты все дескрипторы во всех использующих эту проекцию процессах.

Для демонстрации работы проецируемых в память файлов создадим приложение, которое будет записывать в такой файл строку и спустя некоторое время считывать ее оттуда. Для этого нам понадобится стандартный `TextBox`, кнопка, метка и таймер. Программа будет работать следующим образом: строка, записанная в поле редактора, после нажатия кнопки помещается в проецируемый файл. Далее, спустя некоторое время (задается таймером), содержимое файла считывается и задается в качестве заголовка метки (рис. 8.2).

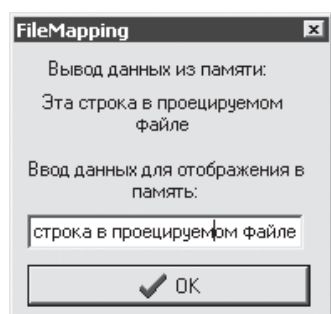


Рис. 8.2. Вид приложения, использующего проецируемый файл

В секцию описания переменных программы помещаем следующие объявления:

```
var
    FormMappingFile: TFormMappingFile;
    //Глобальные переменные
    //Описатель объекта проецируемого файла
    hFileMapObj:THandle;
    //Указатель на начальный адрес данных
    lpBaseAddress:PChar;
```

Далее рассмотрим, какие действия выполняются при загрузке формы. Создание проецируемого файла и его отображение в адресное пространство процесса выполняется в момент создания формы (листинг 8.7).

Листинг 8.7. Создание формы приложения

```
procedure TMappingFile.FormCreate(Sender: TObject);
begin
```

```
//Создаем проецируемый файл с именем FileMemory
//и передаем полученный в результате описатель
//в глобальную переменную hFileMapObj
hFileMapObj := CreateFileMapping (MAXDWORD, Nil, PAGE_READWRITE,
0, 4, 'FileMemory');
If (hFileMapObj = 0) Then
    ShowMessage('Не могу создать проецируемый файл!')
Else
    //Подключаем файл к адресному пространству
    //и получаем начальный адрес данных
    lpBaseAddress := MapViewOfFile(hFileMapObj, FILE_MAP_WRITE,
0, 0, 0);
    If lpBaseAddress = Nil Then
        ShowMessage('Не могу подключить проецируемый файл!');
end;
```

После инициализации файла его можно использовать. Приведем листинг обработчика, копирующего данные в проецируемый файл (листинг 8.8).

Листинг 8.8. Копирование данных в проецируемый файл

```
procedure TMappingFile.bnOKClick(Sender: TObject);
begin
    //Считываем данные в проецируемый файл
    StrPCopy(lpBaseAddress, edVariable.Text);
end;
```

После того как будет нажата кнопка, данные помещаются в проецируемый файл. По истечении некоторого времени, заданного таймером, строка устанавливается в качестве текста метки (листинг 8.9).

Листинг 8.9. Считывание данных из проекции файла

```
procedure TMappingFile.TimerMFTimer(Sender: TObject);
begin
    lbVariable.Caption := PChar(lpBaseAddress);
end;
```

В момент завершения приложения необходимо отключить проецируемый файл от адресного пространства процесса и закрыть объект файла. Эти действия можно выполнять в момент уничтожения формы (листинг 8.10).

Листинг 8.10. Уничтожение формы приложения

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

```
begin
    //Отключим файл от адресного пространства
    UnMapViewOfFile(lpBaseAddress);
    //Освобождаем объект файла
    CloseHandle(hFileMapObj);
    //Закрываем форму
    Action := caFree;
end;
```

Здесь рассмотрен простой пример работы с проекцией файла в рамках одного приложения. Более же интересный и реальный пример вы увидите в разд. 10.2 при рассмотрении программы «Оконный шпион»: там проекция файла в память используется для передачи данных из функции DLL, работающей в памяти другого процесса.



Глава 9

Возможности COM в Microsoft Word и Microsoft Excel

- ☐ Технология OLE
- ☐ Технология COM
- ☐ Использование OLE в Delphi
- ☐ Управление Microsoft Word и Microsoft Excel

Технология COM/DCOM является одной из важных и широко используемых современных технологий. Охватить все аспекты технологии COM/DCOM очень сложно, и в рамках данной книги в этом нет необходимости. В этой главе будут рассмотрены основные возможности COM и их практическое применение. Примеры, разобранные в главе, демонстрируют управление приложениями, снабжаемыми COM-объектами. К таким приложениям можно отнести все программы из пакета Microsoft Office (Microsoft Word, Microsoft Excel и т. д.).

9.1. Технология OLE

В Windows 3.1 и более ранних версиях основным средством обмена данными между программами была технология DDE — Dynamic Data Exchange (динамический обмен данными). На этой технологии основывалась технология OLE — Object Linking and Embedding (связывание и внедрение объектов). OLE позволяет делать документы одного приложения частью документов другого приложения. Таким образом, пользователь получил возможность применять функции различных программ для редактирования одного документа.

В основе DDE лежит обмен сообщениями между окнами операционной системы. Подобный механизм затрудняет распараллеливание процессов и обмен данными через сеть между приложениями, работающими на разных компьютерах. Это привело к созданию расширения DDE — NetDDE, но эта технология работает медленно и неустойчиво.

Начиная с Windows NT 3.51, внедряется технология OLE 2 — дальнейшее развитие OLE. OLE 2 дополнительно включает в себя технологию ActiveX. Позже термин OLE 2 изменили на OLE.

Технология DDE была недостаточной для поддержки OLE 2, поэтому специально для нее была создана технология взаимодействия между программами — COM (Component Object Model, модель компонентных объектов). COM оказалась очень удачной технологией, поэтому, начиная с Windows 95, DDE была объявлена устаревшей, а основной технологией обмена данными в системе стала технология COM.

9.2. Технология COM

Модель COM построена по принципу архитектуры «клиент — сервер». Сервер предоставляет список возможных действий (функций), которые могут использоваться клиентским процессом. Таким образом, серверный процесс позволяет обрабатывать запросы клиента, выполняя некоторые действия. Когда взаимодействие между клиентом и сервером подразумевает обмен данными, эти данные передаются в качестве параметров функций. При необходимости клиент также может экспортировать функции, которые могут быть вызваны сервером.

В основе COM лежат ключевые понятия, которые характерны и для объектно-ориентированного программирования: инкапсуляция, наследование и полиморфизм. Рассмотрим их применительно к объектам COM.

Инкапсуляция позволяет скрыть методы (функции) и данные от использования другими объектами. Этот механизм необходим для обеспечения безопасности и надежности конечной системы. Термин «метод» использован не случайно, объекты COM строятся по принципу классов в программировании (класс имеет название `CoClass`, приставка `Co` говорит о том, что это класс COM).

Наследование позволяет многократно использовать готовые решения. Создавая объект и наследуя некоторые свойства (данные) и методы (функции), мы можем использовать их в дальнейшем. Механизм наследования в связке с принципом полиморфизма позволяет создавать иерархии COM-классов для эффективного решения любых задач. Кроме наследования, часто используется и агрегация — внедрение ранее реализованных объектов внутрь вновь разрабатываемых.

Полиморфизм позволяет переопределять реализацию (поведение) унаследованных функций и данных. Это дает возможность более гибко строить иерархию классов, снижая тем самым сложность реализации программ.

Изначально технология COM обеспечивала межпроцессное взаимодействие только на локальном компьютере. Эволюция COM привела к созданию DCOM (Distributed COM, распределенная COM), позволяющей работать с объектами, которые расположены на различных и удаленных друг от друга компьютерах.

На данный момент DCOM является межплатформенной технологией. Существуют средства для поддержки DCOM в различных UNIX-системах (в том числе Linux), Solaris, MacOS, VxWorks.

9.3. Использование OLE в Delphi

Как и многие современные среды программирования, Delphi поддерживает возможность автоматизированной разработки приложений, работающих с различными COM-сервисами или серверами. Для более глубокого понимания принципов работы приложений, создаваемых на основе COM-технологии, проведем краткий обзор наиболее доступных COM-серверов — приложений пакета Microsoft Office.

Microsoft Office с точки зрения COM

Microsoft Office является средой, в которой большая часть задач решается без использования программирования. Но ценность приложений Microsoft Office заключается в том, что все задачи решаются как традиционным способом (ручное редактирование), так и посредством применения программирования на известном языке VBA (Visual Basic for Application). Кроме того, приложения пакета снабжаются серверами COM, которые предоставляют интерфейс доступа к приложению и его объектам. Благодаря этому разработчик в среде Delphi имеет возможность, создав контроллер автоматизации, управлять сервером. Приложение Microsoft Office можно рассматривать как совокупность объектов с их методами и свойствами (они организуют основу программы). Как правило, в каждом приложении существует так называемый корневой объект, который носит название `Application`. Каждое приложение Microsoft Office имеет собственный корневой

объект — `Word.Application`, `Excel.Application`. Приложение само является корневым объектом, несмотря на это, в объект `Application` встраиваются все остальные объекты (участники), которые являются свойствами главного объекта. Документ, созданный на базе COM, предоставляет большое количество разнообразных методов, но имеются и одинаковые методы в различных приложениях Microsoft Office. Например, `Run`, `Quit`, `Activate`.

При открытии любого приложения из пакета автоматически создается каркас нового документа, который представляет собой набор библиотек с классами. Объекты этих классов будут доступны в открытом документе. Задача разработчика клиента (контроллера автоматизации) — получить доступ к корневому объекту сервера, выстроить цепочку доступа к объектам-участникам (встроенным объектам), правильно передать параметры. Таким образом, получив доступ к объектам документа, можно проводить с ним различные манипуляции, редактировать его и т. д.

Объект Application

Безусловно, самым важным объектом в приложениях Microsoft Office является объект `Application`. Небольшой пример использования данного объекта продемонстрирует простоту программирования с применением COM. Для решения часто встречающихся задач используются уже хорошо нам известные компоненты среды Delphi. В случае технологии COM это не является исключением. Запустим сервер приложения Microsoft Word. Для этого выполним следующее.

1. Создаем новый проект.
2. На главную форму приложения помещаем компонент `WordApplication` вкладки `Servers`.
3. Задаем свойства компонента `AutoConnect` и `AutoQuit` значением `True`.
4. Запускаем созданное приложение.

На первый взгляд, ничего существенного не происходит, но результат работы программы можно заметить путем просмотра запущенных процессов (не путать с задачами). В приложении Диспетчер задач среди процессов различных приложений можно увидеть `WordCOM.exe`. Этот факт говорит нам о том, что созданное нами посредством COM приложение подключилось к серверу Microsoft Word и запустило его. Фактически в системе произошло следующее. В реестре был найден зарегистрированный ранее COM-сервер приложения Microsoft Word. Используя все тот же реестр, был найден путь к программе и произведен ее запуск. Вследствие этого мы заметили появление процесса, отвечающего за работу редактора Microsoft Word.

Но для того чтобы лучше понять запуск приложения Microsoft Word, приведем фрагмент исходного текста, результат работы которого аналогичен (листинг 9.1).

Листинг 9.1. Запуск Microsoft Word

```
procedure TFormStartWord.ButtonStartClick(Sender: TObject);  
var
```

```
//Переменная, интерфейс к объекту
Wordvar : OleVariant;
file_Name : string;
begin
  //Начало блока перехвата исключения
  try
    file_Name := ExtractFilePath(Application.EXEName) +
'worddoc.DOC';
    //Инициализируем объект интерфейса
    //для доступа к серверу COM Microsoft Word
    Wordvar := CreateOleObject('Word.Application');
    //Добавление документа
    wordvar.application.documents.add;
    wordvar.application.activedocument.range.insertAfter(now);
    //Сохранение документа (аналог действиям: "Сохранить как...",
    //с указанием имени файла)
    wordvar.application.activedocument.saveas(fileName);
    //Завершение работы с приложением и выгрузка COM-сервера
    wordvar.application.quit(true,0);
    ...
end;
```

Предложенный исходный текст демонстрирует подключение к серверу без помощи компонента среды разработки. Для корректной работы необходимо в раздел `uses` включить `COMOBJ` — модуль работы с объектом COM. Важно отметить, что наличие функций, вызываемых для объекта `wordvar`, определяется в период выполнения. Это значит, что ошибка может обнаружиться только в период выполнения программы, поэтому весь код работы с объектом помещен в блок `try`.

Класс TOLEServer

На вкладке `Servers` находится набор компонентов для доступа к серверам автоматизации. Не все компоненты возвращают ссылку на объект `Application`, то есть могут быть получены интерфейсы для доступа к таким вложенным объектам, как документ Microsoft Word или рабочая книга Microsoft Excel. Все компоненты унаследованы от класса `TOLEServer`, который наследует свойства класса `Tcomponent`. `TOLEServer` является базовым классом всех COM-серверов. Кроме этого, данный класс имеет еще несколько свойств и методов для управления связью с COM-сервером. Среди таких уже знакомое нам свойство `AutoConnect`, которое автоматически запускает COM-сервер и извлекает из него интерфейс, обеспечивающий связь с контроллером. Еще одно важное свойство класса `TOLEServer` — это `ConnectKind`, указывающее тип процесса, с которым устанавливается связь. Свойство используется методом

Connect, который вызывается автоматически, если свойство `AutoConnect` истинно. В табл. 9.1 описаны значения, которые может принимать `ConnectKind`.

Таблица 9.1. Значение свойства `ConnectKind`

Значение свойства	Характеристика
<code>CkRunningOrNew</code>	Контроллер позволяет производить подключение к уже существующему процессу. Если процесс не запущен, то создается новый процесс, к которому происходит дальнейшее подключение. Этот вид взаимодействия между COM-сервером и контроллером наиболее часто применяется на практике. Это значение установлено по умолчанию
<code>CkNewInstance</code>	При соединении с сервером каждый раз создается новый экземпляр процесса
<code>CkRunningInstance</code>	Соединение устанавливается с уже запущенным COM-сервером. В случае отсутствия процесса сервера возникает ошибка
<code>CkRemote</code>	Это значение используется совместно с <code>RemoteMachineName</code> , если необходимо подключиться к серверу на удаленном компьютере
<code>ckAttachToInterface</code>	При установке этого значения интерфейс не создается, и соответственно нельзя указывать значение <code>True</code> для свойства <code>AutoConnect</code> . Соединение с сервером производится с помощью метода <code>ConnectTo</code>

Более подробно следует рассмотреть значение свойства `ConnectKind`, равное `ckAttachToInterface`. Соединение с сервером производится посредством использования главного интерфейса `Application`, но, например, возникает необходимость подключить к нашему проекту такие компоненты как `WordDocument` или `WordParagraphFormat`. В этом случае мы просто подключаемся к уже существующему интерфейсу, а не создаем его заново. Также это может быть необходимо, когда контроллер должен отслеживать события, происходящие в COM-сервере.

9.4. Управление Microsoft Word и Microsoft Excel

Трюки в Microsoft Word

В этом разделе мы более подробно остановимся на рассмотрении практических примеров использования COM-сервера редактора Microsoft Word. Достаточно популярный редактор обладает обширным набором возможностей, которые можно использовать вручную (традиционное создание и редактирование документов) и с применением технологии COM. Основное удобство последнего метода заключается в автоматизации рутинной работы, например составления отчетов. Следующий пример поможет нам разобраться в принципах построения контроллеров автоматизации, которые ранее уже упоминались. Контроллер автоматизации с точки зрения COM представляет собой приложение, которое посредством вызова проце-

дур сервера проводит различные манипуляции над документом. В Microsoft Word это может быть написание текста в установленном формате и т. д.

Рассмотрим пример приложения, которое будет создавать новый документ Microsoft Word, записывать в него некоторый текст, добавлять таблицу и сохранять полученный документ в файл. В целях наилучшего понимания принципов использования объектов СОМ первый пример не будет использовать компонент среды разработки. Итак, приступим к созданию приложения. Для начала создаем новый проект и помещаем на форму следующие кнопки:

- открытия приложения Microsoft Word;
- вывода текста;
- добавления таблицы;
- сохранения документа;
- завершения работы Microsoft Word.

Мы не будем использовать компоненты, поэтому добавляем в секцию `uses` модуль `ComObj`. Для работы с СОМ-сервером редактора нам понадобится объект OLE. Добавляем переменную типа `OleVariant`:

```
var
    //Объект OLE
    Wrd: OleVariant;
```

Обработчик кнопки запуска редактора имеет следующий вид (листинг 9.2).

Листинг 9.2. Запуск редактора Microsoft Word

```
procedure TFormWord.bnOpenWordClick(Sender: TObject);
begin
    //Создаем объект
    Wrd := CreateOleObject('Word.Application');
    //Делаем видимым приложение
    Wrd.Visible := true;
    //Добавляем новый документ
    Wrd.Documents.Add;
end;
```

После инициализации объекта создаем новый документ, предварительно активизировав (отобразив на экране) приложение. После того как Microsoft Word запущен и в нем создан новый документ, можно записывать текст. Для этого определяем обработчик кнопки вывода текста (листинг 9.3).

Листинг 9.3. Вывод текста в Microsoft Word

```
procedure TFormWord.bnSetTextClick(Sender: TObject);
begin
```

```
//Процедура записи текста
//Устанавливаем шрифт
Wrd.Selection.Font.Size := 20;
Wrd.Selection.Font.Bold := true;
//Пишем текст
Wrd.Selection.TypeText('Технология COM является одной из со-
временных');
Wrd.Selection.TypeText('технологий организации межпроцессного
взаимодействия'#13#10#13#10);
//Задаем новые параметры шрифта
Wrd.Selection.Font.Size := 12;
Wrd.Selection.Font.Bold := false;
Wrd.Selection.Font.Italic := true;
Wrd.Selection.TypeText('Подпись: ');
Wrd.Selection.Font.Bold := true;
Wrd.Selection.TypeText('Delphi'#13#10#13#10);
end;
```

Особой сложности данный фрагмент вызывать не должен, так как настройка шрифта и вывод текста производятся посредством интуитивно понятных функций и заданием соответствующих свойств. Но надо пояснить, что набор символов #13#10 эквивалентен переходу на новую строку.

Процедура добавления таблицы является достаточно простой и выглядит следующим образом (листинг 9.4).

Листинг 9.4. Добавление таблицы

```
procedure TFormWord.bnAddTableClick(Sender: TObject);
begin
    //Процедура добавления новой таблицы
    Wrd.ActiveDocument.Tables.Add(Wrd.Selection.Range, 3, 3);
end;
```

Таблица содержит три столбца и столько же строк. Далее следует пояснить обработчик нажатия кнопки сохранения документа (листинг 9.5).

Листинг 9.5. Сохранение документа Microsoft Word

```
procedure TFormWord.bnSaveClick(Sender: TObject);
begin
    //Сохранение документа
    Wrd.ActiveDocument.SaveAs(ExtractFilePath(Application.EXEName) +
                               '_result.DOC');
end;
```

Сохранение осуществляется путем вызова метода `SaveAs` объекта `ActiveDocument`, который в качестве параметра принимает путь к файлу. После нажатия кнопки сохранения документ с текстом будет записан в файл (`_result.doc`) каталога, из которого была запущена программа.

Процедура завершения работы основана на вызове метода `Quit` (листинг 9.6).

Листинг 9.6. Завершение работы с Microsoft Word

```
procedure TFormWord.bnExitWordClick(Sender: TObject);
begin
    //Завершение приложения
    Wrd.Quit;
end;
```

Рассмотренное приложение является примитивным контроллером автоматизации и может служить отправной точкой создания более сложных и функциональных программ автоматического составления отчетов и т. п.

Далее приступим к созданию приложения, которое будет подключаться к серверу COM Microsoft Word и выводить текст, дату и время вывода этого текста в активный документ при его смене (переключении между документами). На этот раз мы воспользуемся компонентами `WordDocument` и `WordApplication` с вкладки **Servers**.

Создаем новый проект и на главную форму приложения помещаем компоненты `WordDocument` и `WordApplication`. Далее устанавливаем свойство `ConnectKind` компонента `WordApplication` в `ckRunningInstance`, а также значение свойства `AutoConnect` в `True`. В данном случае приложение Microsoft Word создаваться не будет, а программа подключится к уже существующему серверу. Основную практическую ценность для нас представляет механизм определения активного документа и добавление в него текста, даты и времени (листинг 9.7).

Листинг 9.7. Реакция на смену активного документа

```
procedure TFormActiveWord.WordApplicationActiveDocumentChange
(Sender: TObject);
begin
    //Подключаемся к текущему документу
    WordDocumentNew.ConnectTo( WordApplicationActive.ActiveDocument);
    //Контроллер добавляет новую строку в текущий документ
    WordDocumentNew.Range.InsertAfter(#13#10+'Переход к доку-
менту'+#13#10+
    WordApplicationActive.ActiveDocument.Get_FullName+' произ-
веден :'+ DateTimeToStr(Now));
end;
```

Как вы заметили, подключение к уже существующему серверу происходит каждый раз после смены активного документа. В этот момент в содержимое документа

записывается информация: текстовая строка, дата и время перехода к этому документу.

Чтобы просмотреть работу этого приложения, запустите Microsoft Word и создайте в нем два документа. Запустите созданный пример и поочередно активизируйте документы (щелчком кнопкой мыши на Панели задач).

Трюки в Microsoft Excel

Не менее популярным и функциональным приложением из пакета Microsoft Office является Microsoft Excel. Это программа для работы с электронными таблицами. Как и уже знакомое нам приложение Microsoft Word, Microsoft Excel также обладает возможностью создания и редактирования документов (в данном случае таблиц) посредством COM. Преимущества использования Microsoft Excel из других программ очевидны, так как она предоставляет широкий спектр возможностей по построению диаграмм, графиков, произведению различных расчетов и пр. Поэтому в качестве примера создадим приложение, которое будет выполнять запуск Microsoft Excel, добавление новой книги, создание листа и помещение в его ячейки текста и формул.

Как и в случае с Microsoft Word, будет использоваться объект типа `OleVariant`. Но методы и свойства COM-сервера поменяются. Рассмотрим исходный текст приложения для выполнения несложных операций с сервером Microsoft Excel (листинг 9.8).

Листинг 9.8. Работа с Microsoft Excel

```
unit COMinExcel;
interface
uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms,
    Dialogs,
    //Включаем модуль работы с COM-объектами
    ComObj;

type
    TFormCOMExcel = class(TForm)
    //Процедура, вызываемая при создании формы
    procedure FormCreate(Sender: TObject);
    //Процедура, вызываемая при завершении работы приложения
    procedure FormDestroy(Sender: TObject);
private
    { Private declarations }
```

```
public
  { Public declarations }
end;

var
  FormCOMExcel: TFormCOMExcel;
  //Объявление объекта OleVariant с именем Microsoft Excel
  Excel: OleVariant;

implementation

{$R *.dfm}

procedure TFormCOMExcel.FormCreate(Sender: TObject);
begin
  //Инициализируем объект
  Excel := CreateOleObject('Excel.Application');
  //Устанавливаем видимым окно приложения Microsoft Excel
  Excel.Visible := true;
  //Добавляем новую книгу
  Excel.Application.Workbooks.Add;
  //Вводим текст в ячейку с индексом E5
  Excel.Application.Worksheets.Item['Лист1'].
  Cells.Item[5,5].FormulaR1C1 := '! ТЕКСТ !';
  //Задаем характеристики шрифта
  Excel.Application.Worksheets.Item['Лист1'].
  Cells.Item[1,1].Font.Bold := true;
  //В ячейку с индексом A1 записываем формулу
  Excel.Application.Worksheets.Item['Лист1'].
  Cells.Item[1,1].FormulaR1C1 := '=18*2';
end;

procedure TFormCOMExcel.FormDestroy(Sender: TObject);
begin
  //Закрываем приложение Microsoft Excel
  Excel.Quit;
end;
end.
```

Предложенный листинг демонстрирует основы удаленного управления приложением Microsoft Excel. Запуск Microsoft Excel, заполнение ячеек новой таблицы происходит в функции FormCreate.

Во время создания главной формы приложения-примера на экране появится окно программы Microsoft Excel с числом в ячейке с индексом A1 и текстом в ячейке с индексом E5. Хотя в ячейку с индексом A1 мы записывали $=18*2$, на экране в этой ячейке будет отображаться 32, так как Microsoft Excel автоматически преобразует выражения в ячейках.



Глава 10

Окна других приложений

- ☐ Ловушки Windows
- ☐ Программа «Оконный шпион»

Здесь мы будем использовать сведения, приведенные в предыдущих главах (а точнее, в главах 1, 2 и 8), для построения программы, позволяющей проводить различные операции с окнами приложений. Вы также дополнительно познакомитесь с техникой применения ловушек (hook) в Windows и увидите пример реального использования проецирования файла в память для обмена данными между несколькими приложениями. Причем второе в нашем примере обусловлено особенностью работы ловушек, следящих за работой других приложений. Вы также узнаете, как перечислять все открытые окна и, соответственно, получать к ним доступ. Но обо всем по порядку.

10.1. Ловушки Windows

Из предыдущих глав вам должен быть понятен или, по крайней мере, известен механизм, который используется Windows для управления окнами приложений, — сообщения. Вероятно, большая мощь этого механизма и в то же время его уязвимость состоят в возможности посылки сообщений любым окнам (окнам одного процесса или окнам других процессов).

В Windows также предусмотрен мощный механизм, позволяющий следить за некоторыми важными событиями в системе и, конечно, производить мониторинг сообщений, получаемых различными окнами. Речь идет об установке так называемых ловушек. Ловушка представляет собой функцию, вызываемую при возникновении определенного события, например перед получением каким-либо окном нового сообщения, при нажатии клавиши, записи события в системный журнал и т. д.: все зависит от того, для каких событий разработчики предусмотрели ловушки. Интересен тот факт, что в Windows предусмотрены даже ловушки для отладки других ловушек.

Мы рассмотрим некоторые наиболее простые виды ловушек, перехватывающих сообщения окон. По глобальности действия рассмотренные нами ловушки являются устанавливаемыми на отдельный поток: при ошибке в функции-ловушке это безопаснее для системы.

Начинается создание ловушки с написания собственно функции-ловушки, имеющей следующий прототип:

```
function HookProc(code: Integer; wParam: WPARAM; lParam: LPARAM):  
    HRESULT stdcall;
```

Параметр `code` используется для обозначения тех случаев, когда функция ловушки должна вызвать специальную API-функцию `CallNextHookEx` и вернуть значение, возвращенное ею. Назначения параметров `wParam` и `lParam` этой функции сильно зависят от того, для реакции на какое именно событие ловушка используется.

Для регистрации ловушки используется API-функция `SetWindowsHookEx`, имеющая следующий прототип:

```
function SetWindowsHookEx(idHook: Integer; //Тип ловушки
    lpfn: TFNHookProc; //Адрес функции-ловушки
    hmod: HINST; //Используемый модуль, в котором
        //расположена функция ловушки
    dwThreadId: DWORD //Идентификатор потока, для
        //которого создается ловушка
): HHOOK; stdcall;
```

В случае успешного создания ловушки функция `SetWindowsHookEx` возвращает дескриптор новой ловушки (ненулевое значение).

Для удаления ловушки используется функция `UnhookWindowsHookEx`, принимающая единственный параметр — дескриптор ловушки, возвращенный функцией `SetWindowsHookEx`. Причем удаление ловушки нужно производить обязательно, поэтому по крайней мере при закрытии приложения не следует забывать вызывать функцию `UnhookWindowsHookEx`.

Теперь несколько слов о функции `CallNextHookEx`. Ее объявление имеет следующий вид:

```
function CallNextHookEx(hhk: HHOOK; nCode: Integer;
    wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
```

В чем важность этой функции? Она предназначена для продолжения передачи сообщения по цепочке ловушек (ведь одновременно несколько приложений могут создать несколько ловушек). Вызов этой функции настоятельно рекомендуется осуществлять в любом случае (независимо от значения параметра `code` функции-ловушки), только если целью не стоит блокирование других ловушек.

Виды ловушек

Приведем список некоторых простых типов ловушек, а именно констант из модуля `Windows`, их обозначающих и передаваемых в функцию `SetWindowsHookEx`:

- `WH_CALLWNDPROC` — функция ловушки вызывается каждый раз до вызова функции обработки сообщений окон, созданных наблюдаемым потоком;
- `WH_CALLWNDPROCRET` — вызывается каждый раз при возврате из функции обработки сообщений окон наблюдаемого потока;
- `WH_KEYBOARD` — функция ловушки вызывается перед обработкой сообщений `WM_KEYDOWN` и `WM_KEYUP` оконной функцией исследуемого потока;
- `WH_MOUSE` — вызывается перед обработкой оконной функцией наблюдаемого потока сообщений от манипулятора «мышь».

Рассмотрим, какое значение имеют параметры `lParam` и `wParam` функции-ловушки в каждом из перечисленных случаев.

Перехват вызова оконной функции

Итак, для ловушки `WH_CALLWNDPROC`, которая, кстати, используется в рассматриваемом далее приложении, два последних параметра функции-ловушки трактуются следующим образом:

- `wParam` — равен нулю, если сообщение послано в окно тем же потоком, в котором выполняется функция ловушки, и не равен нулю, если сообщение послано другим потоком;
- `lParam` — указатель на структуру `TCWPStruct`, содержащую информацию о сообщении, которое передано окну (и будет передано в оконную функцию).

Объявление структуры `TCWPStruct` с описанием ее полей выглядит следующим образом:

```
type TCWPStruct = packed record
    lParam: LPARAM; //Параметр сообщения
    wParam: WPARAM; //Параметр сообщения
    message: UINT; //Код сообщения
    hwnd: HWND; //Окно, которому адресовано сообщение
end;
```

Ниже приводится пример преобразования параметра `lParam` функции ловушки к указателю на структуру с последующей проверкой кода сообщения (фрагмент программы):

```
var hook_data : hook_data: ^TCWPStruct;
begin
    hook_data := Pointer(lParam);
    if hook_data^.message = WM_SIZE then
    begin
        //Реагируем на изменение размера окна
    end;
end;
```

Получение доступа к данным, передаваемым в остальные функции ловушки (а именно, несложное преобразование типов у операции с указателем), осуществляется аналогичным образом, поэтому более демонстрироваться не будет.

Перехват возврата из оконной процедуры

Для ловушки `WH_CALLWNDPROCRET` параметры `wParam` и `lParam` функции-ловушки следует трактовать следующим образом:

- `wParam` — равен нулю, если сообщение послано другим процессом, и не равен нулю в противном случае;

- `lParam` — указатель на структуру `TCWPRetStruct`, содержащую информацию о сообщении, которое передано окну (и будет передано в оконную функцию).

Объявление структуры `TCWPRetStruct` с описанием ее полей выглядит следующим образом:

```
type TCWPRetStruct = packed record
    lResult: LRESULT; //Значение, возвращенное оконной функцией
    lParam: LPARAM;   //Параметр сообщения
    wParam: WPARAM;   //Параметр сообщения
    message: UINT;     //Код сообщения
    hwnd: HWND;        //Дескриптор окна-получателя
end;
```

Перехват сообщений клавиатурного ввода

Для ловушки `WH_KEYBOARD` параметры `wParam` и `lParam` функции-ловушки следует трактовать следующим образом:

- `wParam` — код нажатой клавиши;
- `lParam` — первые 16 бит этого параметра означают количество повторений нажатия; старшие 16 бит используются для дополнительного описания состояния клавиатуры в момент нажатия клавиши.

Параметры `wParam` и `lParam` полностью аналогичны параметрам сообщений `WM_KEYDOWN` и `WM_KEYUP`.

Перехват сообщений от мыши

В ловушку `WH_KEYBOARD` в параметрах `wParam` и `lParam` передаются следующие значения:

- `wParam` — код сообщения мыши;
- `lParam` — указатель на структуру `TMouseHookStruct`.

Объявление структуры `TMouseHookStruct` с описанием полей выглядит следующим образом:

```
type TMouseHookStruct = packed record
    pt: TPoint; //Экранные координаты указателя мыши
    hwnd: HWND; //Дескриптор окна-получателя сообщения
    wParam: WPARAM; //Код, возвращенный оконной функцией
    lParam: LPARAM; //в ответ на сообщение WM_NCHITTEST
    dwExtraInfo: DWORD; //Дополнительные данные
end;
```

Если вы забыли, какое значение имеет для окна сообщение `WM_NCHITTEST`, то можете вновь обратиться к гл. 1.

Расположение функции-ловушки и DLL

Теперь поговорим немного о расположении функции-ловушки.

Казалось бы, что здесь может быть такого: написал функцию в модуле, строго соответствующую приведенному ранее прототипу, передал ее адрес в функцию `SetWindowsHookEx` и используй ловушку. Но не так все просто. Функция ловушки может находиться в исполняемом файле только в том случае, если предполагается использовать ее для перехвата сообщений потока (потоков) того же процесса. Тогда в функцию создания ловушки в качестве параметра `hmod` следует передавать нулевое значение.

Если же предполагается слежение за другими приложениями (за потоками других процессов), то функция ловушки должна быть экспортируемой функцией DLL. Тогда в функцию `SetWindowsHookEx` передается дескриптор модуля DLL (похоже, что это адрес в адресном пространстве процесса, куда спроецирован файл DLL). Библиотека (DLL) может загружаться как при запуске приложения (если используется так называемое *load-time* связывание), так и динамически при помощи API-функции `LoadLibrary`:

```
function LoadLibrary(lpLibFileName: PChar): HMODULE; stdcall;
```

Функция принимает в качестве параметра путь DLL и возвращает дескриптор загруженного модуля (или 0 в случае ошибки). Если библиотека больше не нужна, то можно вызвать функцию `FreeLibrary`, передав в качестве единственного параметра возвращенный ранее функцией `LoadLibrary` дескриптор модуля DLL.

Возвращаясь к теме расположения ловушки зададимся вопросом: почему именно DLL? Чем плохо расположение ловушки в EXE-модуле приложения? Самое время вспомнить о том, что каждый процесс в Windows выполняется в своем собственном адресном пространстве. Поэтому адрес функции в исполняемом файле одного процесса вполне может быть адресом структуры данных где-то внутри другого процесса (рис. 10.1).

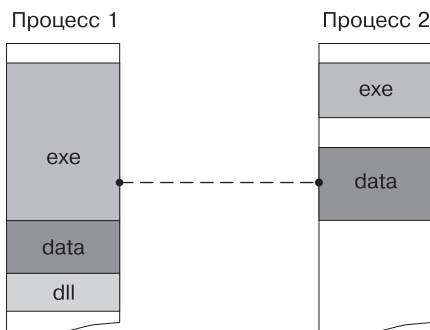


Рис. 10.1. Пример адресного пространства разных процессов

В отличие от EXE-файлов, файлы библиотек легко проецируются в адресное пространство использующего их процесса. Разместив функцию ловушки в DLL и ука-

зав дескриптор модуля этой DLL, мы предоставляем системе полную информацию для того, чтобы она смогла:

- спроецировать библиотеку с ловушкой в адресное пространство исследуемого процесса;
- однозначно определить положение (адрес) функции-ловушки в адресном пространстве исследуемого процесса.

Описанные выше манипуляции с DLL проиллюстрированы на рис. 10.2 (Процесс 2 на рисунке — процесс, в который внедряется ловушка).

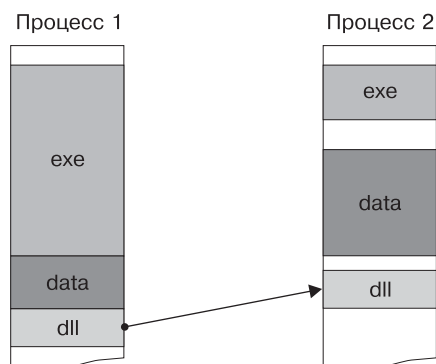


Рис. 10.2. Загрузка DLL с ловушкой в адресное пространство исследуемого процесса

Теперь нет никаких препятствий в вызове функции-ловушки при наблюдении за другим процессом.



ПРИМЕЧАНИЕ

В каждой библиотеке реализуется функция `DllMain`, вызываемая (упрощенно) при загрузке/выгрузке библиотеки. Жаль только, что при описанном способе подгрузки DLL эта функция не вызывается. Это приводит к усложнению кода ловушки, в чем вы сможете вскоре убедиться.

10.2. Программа «Оконный шпион»

Перейдем, наконец, к практической части главы: рассмотрим создание программы, позволяющей составлять список (а точнее, дерево) всех окон, просматривать и изменять их свойства, а также осуществлять перехват сообщений выбранного окна (недаром мы столько времени потратили на рассмотрение ловушек Windows).

Несмотря на свое «противозаконное» название, рассматриваемая программа может весьма пригодиться при отладке приложений, а отнюдь не при их взломе и шпионаже (хотя многое зависит от добросовестности лица, использующего программу). В частности, с помощью этой программы была найдена ошибка, на долгое время закрывшаяся в один из примеров гл. 2: из-за неправильной установки стилей при

ручном создании главного окна программы не удалось добиться правильной перерисовки элемента управления «рамка».

Составление списка открытых окон

Список (а точнее, дерево) окон, открытых в момент запуска программы, показан на рис. 10.3.

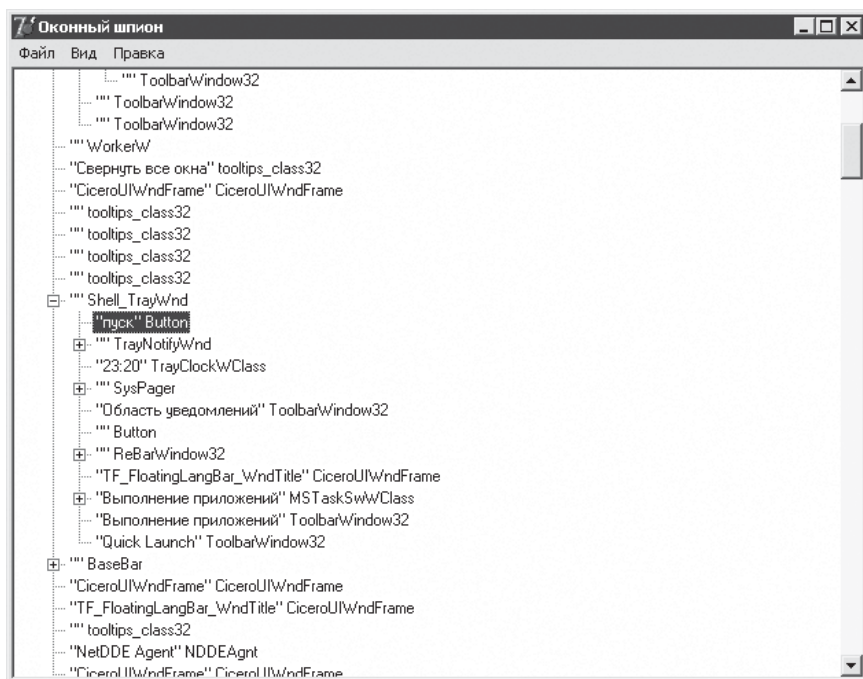


Рис. 10.3. Дерево открытых окон

Форма, показанная на рис. 10.3, имеет имя `frmMain`. Элемент управления `TreeView` имеет имя `tree`. Часть программы, отвечающая за построение дерева, относительно проста. Она использует вскользь рассмотренный в гл.2 механизм перечисления окон.

Составление дерева окон начинается с процедуры `LoadWindowsTree`, которая и запускает перечисление окон верхнего уровня, то есть окон, родителем которых формально является окно Рабочего стола (листинг 10.1).

Листинг 10.1. Начало составления дерева окон

```
procedure TfrmMain.LoadWindowsTree();
var
    desktop: TTreeNode;
    //enInfo: TEnumInfo;
```

```

begin
    tree.Items.Clear;
    //Добавление узла для Рабочего стола
    desktop := tree.Items.Add(tree.Items.GetFirstNode, 'Рабочий
стол');
    //Перечисление окон
    enInfo.tree := tree;
    enInfo.parent := desktop;
    EnumWindows(Addr(NewWindow), Integer(Addr(enInfo)));
end;

```

Сразу следует привести объявление структуры, интенсивно используемой (далее это будет видно) при составлении дерева:

```

Type
    TEnumInfo = Record
        tree: TTreeView;    //Компонент TTreeView
        parent: TTreeNode; //Элемент дерева, соответствующий
                           //текущему окну, дочерние
                           //окна которого перечисляются
    end;

```

При нахождении каждого нового окна вызывается функция `NewWindow` (ее адрес передан в API-функцию `EnumWindows`). Функция `NewWindow` (листинг 10.2) решает две задачи. Во-первых, она добавляет в дерево элемент, соответствующий найденному окну. Во-вторых, запускает поиск дочерних окон относительно найденного окна, что позволяет перечислить все окна (от главной формы приложения до кнопок, надписей и т. д.).

Листинг 10.2. Добавление в дерево информации об окне и поиск дочерних окон

```

function NewWindow(wnd: HWND; param: LPARAM):BOOL; stdcall;
var
    wndNode, parentNode: TTreeNode;
begin
    wndNode := AddWindowToTree(wnd); //Добавление информации об
окне в дерево

    //Перечисление дочерних окон
    parentNode := enInfo.parent;
    enInfo.parent := wndNode;
    EnumChildWindows(wnd, Addr(NewWindow), param);
    enInfo.parent := parentNode;

```

```
//Продолжать перечисление (после перечисления
//всех дочерних окон)
NewWindow := True;
end;
```

Используемая в листинге 10.3 функция `AddWindowToTree` добавляет элемент, соответствующий найденному окну, в дерево (определяет текст заголовка окна и имя оконного класса):

Листинг 10.3. Добавление элемента, соответствующего окну, в дерево

```
function AddWindowToTree(wnd: HWND): TTreeNode;
var
    caption, classname: String;
    text: String;
    node: TTreeNode;
begin
    //Получение текста окна
    SetLength(caption, SendMessage(wnd, WM_GETTEXTLENGTH, 0, 0) + 1);
    SetLength(caption, SendMessage(wnd, WM_GETTEXT, Length(caption),
        Integer(PAnsiChar(caption))));
    //Имя класса окна
    SetLength(classname, 1024);
    SetLength(classname, GetClassName(wnd, PAnsiChar(classname),
        100));

    //Формирование текста для элемента и добавление его в дерево
    text := '"' + caption + '" ' + classname;
    node := enInfo.tree.Items.AddChild( enInfo.parent, text );
    node.Data := Pointer(wnd); //Не забываем запомнить
                                //декриптор окна
    AddWindowToTree := node;
end;
```

Вот, собственно, и все, что требуется для построения полного дерева окон, показанного на рис. 10.3.

Получение информации об окне

Следующей функцией «оконного шпиона» является определение более-менее полной информации об окне, выбранном в дереве. Форма с информацией о выделенном в дереве окне (в данном случае это пресловутая кнопка **Пуск**) показана на рис. 10.4.

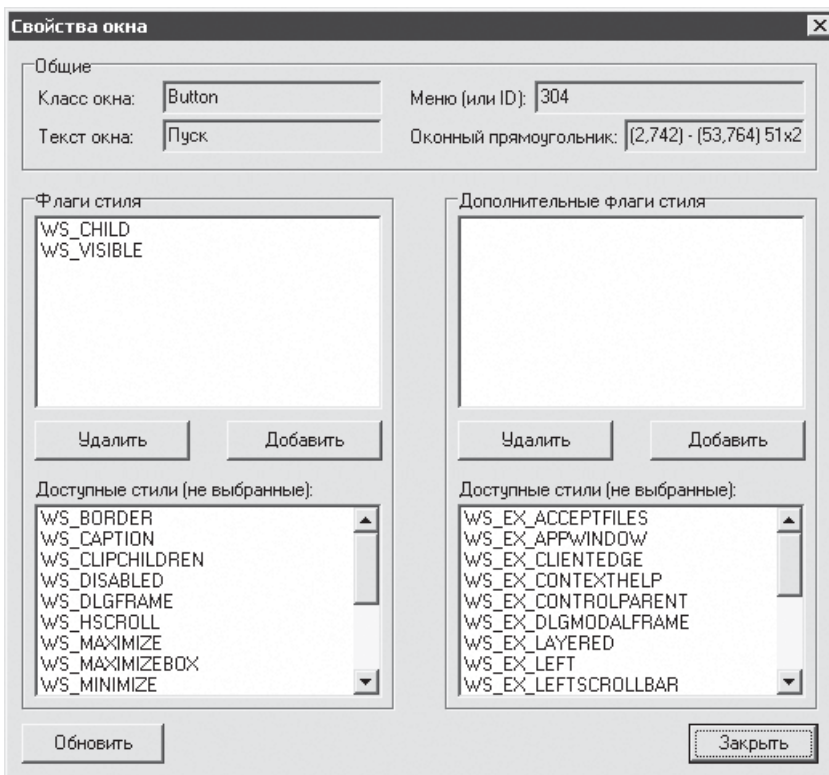


Рис. 10.4. Форма свойств окна

Начинается все с того, что по команде меню **Правка ► Свойства** вызывается метод `ShowWindowProp` созданного при запуске программы объекта `frmWindowProp`. Этот метод принимает в качестве параметра дескриптор окна, информацию о котором нужно отобразить (дескриптор сохраняется в поле `Data` каждого элемента при построении дерева) (листинг 10.4).

Листинг 10.4. Подготовка формы свойств выбранного окна

```
procedure TfrmWindowProp.ShowWindowProp(window: HWND);
begin
    wnd := window;
    LoadWindowInfo();
    ShowModal(); //Не забываем показать сами себя
end;
```

Переменная `wnd`, в которой сохраняется переданный `vShowWindowProp` дескриптор окна, является членом класса `TfrmWindowProp`. Она нужна для того, чтобы другие методы формы `TfrmWindowProp` могли получать доступ к дескриптору окна.

Определение заголовка, имени класса, идентификатора окна, а также области экрана, занимаемой окном, осуществляется в процедуре `LoadWindowInfo` (листинг 10.5).

Листинг 10.5. Определение общей информации об окне

```
procedure TfrmWindowProp.LoadWindowInfo();
var
    rect: TRect;
    buffer: String;
begin
    //Сбор сведений об окне
    //..имя класса
    SetLength(buffer, 1024);
    SetLength(buffer, GetClassName(wnd, PAnsiChar(buffer), 1024));
    txtClassName.Text := buffer;
    //..имя (заголовок) окна
    SetLength(buffer, SendMessage(wnd, WM_GETTEXTLENGTH, 0, 0) + 1);
    SendMessage(wnd, WM_GETTEXT, Length(buffer),
        Integer(PAnsiChar(buffer)));
    txtWindowName.Text := buffer;
    //..идентификатор (или дескриптор меню) окна
    txtId.Text := IntToStr(GetWindowLong(wnd, GWL_ID));
    //..оконный прямоугольник
    GetWindowRect(wnd, rect);
    txtWindowRect.Text :=
        '(' + IntToStr(rect.Left) + ',' + IntToStr(rect.Top) + ')' +
        ' - ' +
        '(' + IntToStr(rect.Right) + ',' + IntToStr(rect.Bottom) + ')' +
        IntToStr(rect.Right-rect.Left) + 'x' + IntToStr(rect.Bottom -
            rect.Top);

    //Определение стиля окна
    LoadWindowStyle();
    LoadWindowExStyle();
end;
```

Если вы внимательно просмотрели листинг 10.5, то могли заметить вызовы двух процедур в двух последних строках кода. Процедура `LoadWindowStyle` заполняет списки используемых и доступных оконных стилей (см. рис. 10.4), а процедура `LoadWindowExStyle` соответственно заполняет списки используемых и доступных дополнительных (или расширенных) стилей окна.

Реализация процедуры LoadWindowState приводится в листинге 10.6.

Листинг 10.6. Заполнение списков оконных стилей

```
procedure TfrmWindowProp.LoadWindowState();
var
  i: Integer;
  style: DWORD;
begin
  style := GetWindowLong(wnd, GWL_STYLE);
  lstStyle.Clear();
  lstAvailStyle.Clear();
  //Выделение из 32-битного значения составляющих стиля окна
  for i := 0 to 17 do
    if styles[i].value and style <> 0 then
      begin
        //Стиль используется
        lstStyle.Items.Add(styles[i].name);
        styles[i].used := True;
      end
    else
      begin
        //Стиль не используется
        lstAvailStyle.Items.Add(styles[i].name);
        styles[i].used := False;
      end;
  end;
end;
```

Вместо громоздкой проверки наличия в значении, возвращенном API-функцией GetWindowLong, битов каждого возможного стиля при помощи, например, case здесь используется глобальный массив styles структур StyleInfo. Объявление типа структуры (записи) StyleInfo выглядит следующим образом:

```
type
  StyleInfo = record
    value: DWORD; //Код стиля
    name: String; //Текстовое обозначение стиля
    used: Boolean; //Служебное поле
  end;
```

Каждый элемент массива styles хранит информацию об определенном оконном стиле. Объявление этого массива, так же, как структуры StyleInfo и прочих

рассмотренных в этом разделе типов данных, находится в модуле `WindowData`, расположенном на диске в папке с номером главы.

Ниже приведено объявление массива `styles` (флаги стиля, являющиеся комбинацией других флагов, в массив не попали) (листинг 10.7).

Листинг 10.7. Массив с информацией об оконных стилях

```
styles: array [0..17] of StyleInfo =  
(  
    (value: WS_BORDER;          name: 'WS_BORDER'),  
    (value: WS_CAPTION;        name: 'WS_CAPTION'),  
    (value: WS_CHILD;          name: 'WS_CHILD'),  
    (value: WS_CLIPCHILDREN;   name: 'WS_CLIPCHILDREN'),  
    (value: WS_DISABLED;       name: 'WS_DISABLED'),  
    (value: WS_DLGFRAME;       name: 'WS_DLGFRAME'),  
    (value: WS_HSCROLL;        name: 'WS_HSCROLL'),  
    (value: WS_MAXIMIZE;       name: 'WS_MAXIMIZE'),  
    (value: WS_MAXIMIZEBOX;    name: 'WS_MAXIMIZEBOX'),  
    (value: WS_MINIMIZE;       name: 'WS_MINIMIZE'),  
    (value: WS_MINIMIZEBOX;    name: 'WS_MINIMIZEBOX'),  
    (value: WS_OVERLAPPED;     name: 'WS_OVERLAPPED'),  
    (value: WS_POPUP;          name: 'WS_POPUP'),  
    (value: WS_SYSMENU;        name: 'WS_SYSMENU'),  
    (value: WS_TABSTOP;        name: 'WS_TABSTOP'),  
    (value: WS_THICKFRAME;     name: 'WS_THICKFRAME'),  
    (value: WS_VISIBLE;        name: 'WS_VISIBLE'),  
    (value: WS_VSCROLL;        name: 'WS_VSCROLL')  
);
```

Процедура `LoadWindowExStyle` реализована практически так же, как и процедура `LoadWindowStyle`. Только она заполняет списки `lstExStyle` и `lstAvailExStyle` и обращается к массиву `exstyles`, а не `styles`. Поэтому приведем объявление только массива `exstyles` (листинг 10.8).

Листинг 10.8. Массив с информацией о дополнительных оконных стилях

```
exstyles: array [0..18] of StyleInfo =  
(  
    (value: WS_EX_ACCEPTFILES; name: 'WS_EX_ACCEPTFILES'),  
    (value: WS_EX_APPWINDOW;   name: 'WS_EX_APPWINDOW'),  
    (value: WS_EX_CLIENTEDGE;  name: 'WS_EX_CLIENTEDGE'),  
    (value: WS_EX_CONTEXTHELP; name: 'WS_EX_CONTEXTHELP'),
```

```

    (value: WS_EX_CONTROLPARENT;    name: 'WS_EX_CONTROLPARENT'),
    (value: WS_EX_DLGMODALFRAME;    name: 'WS_EX_DLGMODALFRAME'),
    (value: WS_EX_LAYERED;          name: 'WS_EX_LAYERED'),
    (value: WS_EX_LEFT;             name: 'WS_EX_LEFT'),
    (value: WS_EX_LEFTSCROLLBAR;    name: 'WS_EX_LEFTSCROLLBAR'),
    (value: WS_EX_MDICHILD;         name: 'WS_EX_MDICHILD'),
    (value: WS_EX_NOACTIVATE;       name: 'WS_EX_NOACTIVATE'),
    (value: WS_EX_NOINHERITLAYOUT;  name: 'WS_EX_NOINHERITLAYOUT'),
    (value: WS_EX_NOPARENTNOTIFY;   name: 'WS_EX_NOPARENTNOTIFY'),
    (value: WS_EX_RIGHTSCROLLBAR;   name: 'WS_EX_RIGHTSCROLLBAR'),
    (value: WS_EX_STATICEDGE;       name: 'WS_EX_STATICEDGE'),
    (value: WS_EX_TOOLWINDOW;       name: 'WS_EX_TOOLWINDOW'),
    (value: WS_EX_TOPMOST;          name: 'WS_EX_TOPMOST'),
    (value: WS_EX_TRANSPARENT;      name: 'WS_EX_TRANSPARENT'),
    (value: WS_EX_WINDOWEDGE;       name: 'WS_EX_WINDOWEDGE')
);

```

Изменение оконных стилей

Изменение стилей окна «на лету» производится не сложнее, чем их определение: с помощью API-функций `GetWindowLong` и `SetWindowLong`. Пример добавления флага, обозначение которого выбрано в списке доступных стилей, приводится в листинге 10.9.

Листинг 10.9. Добавление оконного стиля

```

procedure TfrmWindowProp.cmbAddStyleClick(Sender: TObject);
var
    style: DWORD;
    addstyle: DWORD;
begin
    if lstAvailStyle.ItemIndex = -1 then Exit;
    //Удаление выбранного стиля окна
    //..определяем, какой стиль удалить
    addstyle := styles[GetStyleIndex(lstAvailStyle.ItemIndex,
    False)].value;
    //..вычисляем и устанавливаем новое значение стиля окна
    style := GetWindowLong(wnd, GWL_STYLE);
    style := style or addstyle;
    SetWindowLong(wnd, GWL_STYLE, style);
    //..перерисуем все окна

```

```

    InvalidateRect(0, nil, True);
    //Обновим список стилей окна
    LoadWindowStyle();
end;
```

Удаление флага стиля производится аналогично добавлению, просто над битами стиля окна выполняется другая операция (листинг 10.10).

Листинг 10.10. Удаление оконного стиля

```

procedure TfrmWindowProp.cmbDelStyleClick(Sender: TObject);
var
    style: DWORD;
    delstyle: DWORD;
begin
    if lstStyle.ItemIndex = -1 then Exit;
    //Удаление выбранного стиля окна
    //..определяем, какой стиль удалить
    delstyle := styles[GetStyleIndex(lstStyle.ItemIndex, True)].value;
    //..вычисляем и устанавливаем новое значение стиля окна
    style := GetWindowLong(wnd, GWL_STYLE);
    style := style and not delstyle;
    SetWindowLong(wnd, GWL_STYLE, style);
    //..перерисовываем все окна
    InvalidateRect(0, nil, True);
    //Обновим список стилей окна
    LoadWindowStyle();
end;
```

После удаления или добавления оконного стиля вызывается перерисовка всех окон, чтобы проявился результат проведенной операции. Удаление и добавление дополнительных (расширенных) оконных стилей осуществляется аналогично. Только при этом используются массив `exstyles`, функция `GetExStyleIndex` и константа `GWL_EXSTYLE`, передаваемая в функции `GetWindowLong` и `SetWindowLong`.

Что же за функция `GetStyleIndex` используется в листинге 10.10? Она позволяет определить положение в массиве `styles` стиля, выбранного в списке доступных или используемых стилей (верхний список) (листинг 10.11).

Листинг 10.11. Определение положения записи о нужном стиле

```

function TfrmWindowProp.GetStyleIndex(listIndex: Integer;
                                       used: Boolean): Integer;
var
    i, count: Integer;
```

```
begin
  count := 0;
  for i := 0 to 17 do
    if styles[i].used = used then
      begin
        if count = listIndex then
          begin
            //Нашли
            GetStyleIndex := i;
            Exit;
          end;
          Inc(count);
        end;
        GetStyleIndex := 0;
      end;
end;
```

Функция `GetStyleIndex` принимает в качестве параметров номер строки в соответствующем списке и логическое значение, от истинности или ложности которого зависит, используемые или неиспользуемые стили будут подсчитываться внутри функции.

Применение функции `GetStyleIndex` и введение в структуру `StyleInfo` поля `used` несколько усложняет алгоритм работы с массивом стилей, но зато позволяет избавиться от постоянного перемещения данных, например, из массива доступных стилей в массив используемых стилей. К тому же пришлось бы использовать по два массива для обычных и дополнительных оконных стилей.

Перехват сообщений

Теперь рассмотрим самую сложную часть программы, отвечающую за перехват сообщений выбранного окна. Форма, ведущая статистику перехваченных сообщений, приведена на рис. 10.5.

Показанная на рис. 10.5 форма имеет имя `frmMessages`.

Перехватчик сообщений состоит из двух частей: части программы (EXE), отвечающей за построение фильтра сообщений, а также обрабатывающей перехваченные сообщения, и ловушки, заключенной в DLL (`hook\hook.dll`).

Взаимодействие ловушки и EXE-файла построено по следующей схеме.

1. Из приложения вызываются функции создания и удаления ловушки (расположенные в DLL).
2. При перехвате каждого сообщения функция-ловушка посылает окну (форме) `frmMessages` сообщение `WM_SPY_NOTIFY` (определенное пользователем, точнее, программистом сообщение, листинг 10.12).

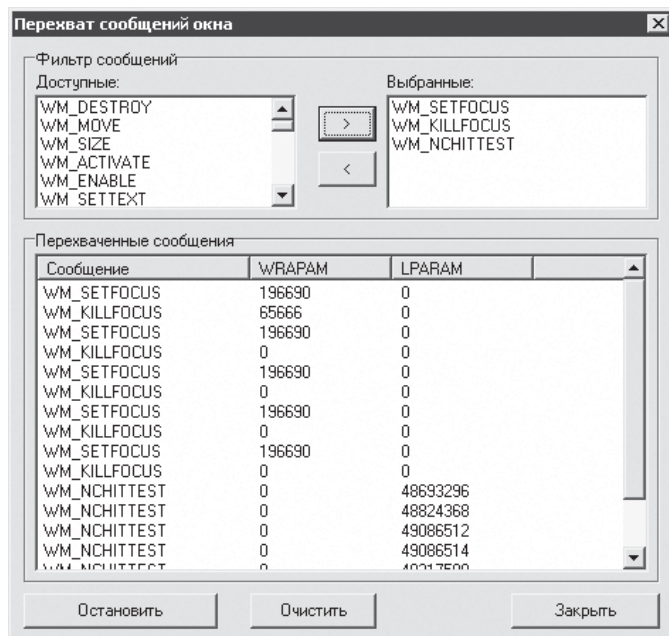


Рис. 10.5. Форма перехвата сообщений

Но ведь ловушка предназначена для работы в другом процессе, а если так, то как ей дать знать, какому именно окну посылать сообщения? Для этого и используется именованная проекция файла в память, в которой сохраняются данные, необходимые для ловушки. В проекции файла ловушка также сохраняет информацию о перехваченном сообщении (код и параметры сообщения). Эта информация используется приложением, ведущим слежение.

Данные в проекции файла хранятся в виде записи `THookInfo`, объявленной в модуле `HookData`. В этом же модуле объявлены константа с именем проекции файла, код сообщения `WM_SPY_NOTIFY` (листинг 10.12) и две служебные переменные, использование которых будет пояснено далее.

Листинг 10.12. Содержимое файла `HookData.pas`

type

```
//Структура (запись), которая хранится в разделяемом файле
//и используется для передачи данных между процессами
THookInfo = record
    wnd: HWND;           //Окно, за которым ведется наблюдение
    hook_handle: HHOOK;  //Дескриптор ловушки
    spy_wnd: HWND;       //Окно, уведомляемое о перехвате сообщения
    //Следующие поля заполняются при перехвате сообщения
    mess: UINT;
```

```
wParam: WPARAM;  
lParam: LPARAM;  
end;
```

```
var
```

```
//Указатель на разделяемую область памяти  
hook_info: ^THookInfo;  
//Дескриптор проекции файла в память  
hFile: THandle;
```

```
const
```

```
//Имя проекции файла  
strFileMapName = 'TricksDelphi_WinSpy_Mapping';  
//Сообщение для уведомления окна-шпиона  
WM_SPY_NOTIFY = WM_USER + 1;
```

Построение фильтра и обработка перехваченных сообщений

Теперь вернемся к приложению-шпиону, а точнее, к той его части, которая отвечает за работу формы, показанной на рис. 10.5.

Начнем с самого простого — управления фильтром сообщений. Он построен по тому же принципу, что управление списками оконных стилей (форма свойств окна, рассмотренная ранее).

Итак, структура, хранящая информацию о сообщении, выглядит следующим образом:

```
type MessageInfo = record  
    value: DWORD; //Код сообщения  
    name: String; //Название сообщения  
    used: Boolean; //Служебное поле  
end;
```

При написании программы не стояла цель поместить в фильтр все возможные сообщения, поэтому массив `messages_list` (листинг 10.13) содержит только 16 элементов. При необходимости вы можете добавить нужные сообщения самостоятельно, взяв их обозначения из модуля `Windows`.

Листинг 10.13. Сообщения, поддерживаемые программой

```
const  
    mess_first = 0;  
    mess_last = 15;
```

```

var
  messages_list: array [mess_first..mess_last] of MessageInfo =
  (
    (value: WM_DESTROY;      name: 'WM_DESTROY';      used: False),
    (value: WM_MOVE;         name: 'WM_MOVE';          used: False),
    (value: WM_SIZE;         name: 'WM_SIZE';           used: False),
    (value: WM_ACTIVATE;     name: 'WM_ACTIVATE';       used: False),
    (value: WM_SETFOCUS;     name: 'WM_SETFOCUS';       used: False),
    (value: WM_KILLFOCUS;    name: 'WM_KILLFOCUS';      used: False),
    (value: WM_ENABLE;       name: 'WM_ENABLE';         used: False),
    (value: WM_SETTEXT;      name: 'WM_SETTEXT';        used: False),
    (value: WM_GETTEXT;      name: 'WM_GETTEXT';        used: False),
    (value: WM_PAINT;        name: 'WM_PAINT';          used: False),
    (value: WM_CLOSE;        name: 'WM_CLOSE';          used: False),
    (value: WM_QUIT;         name: 'WM_QUIT';           used: False),
    (value: WM_SIZING;       name: 'WM_SIZING';         used: False),
    (value: WM_MOVING;       name: 'WM_MOVING';         used: False),
    (value: WM_NOTIFY;       name: 'WM_NOTIFY';         used: False),
    (value: WM_NCHITTEST;    name: 'WM_NCHITTEST';      used: False)
  );

```

Загрузка фильтра (выбранных и невыбранных сообщений в соответствующие списки) производится очень просто (листинг 10.14).

Листинг 10.14. Загрузка фильтра сообщений

```

procedure TfrmMessages.LoadFilter();
var
  i: Integer;
begin
  //Загрузка фильтра сообщений
  lstAvailMessages.Clear();
  lstSelMessages.Clear();
  for i := mess_first to mess_last do
    if messages_list[i].used then
      //Сообщение перехватывается
      lstSelMessages.Items.Add(messages_list[i].name)
    else
      lstAvailMessages.Items.Add(messages_list[i].name);
  end;
end;

```

При обращении к форме `frmMessages`, кроме загрузки фильтра, нужно произвести некоторые дополнительные действия. Поэтому работа с этой формой начинается так же, как и в случае формы свойств окна, с вызова ее специального метода (листинг 10.15).

Листинг 10.15. Инициализация формы

```
procedure TfrmMessages.ShowMessages(wnd: HWND);
begin
    self.wnd := wnd;
    LoadFilter();
    ShowModal();
end;
```

При нажатии кнопок > (выбрать) и < (отменить выбор) происходит перемещение сообщений между списками фильтра (листинг 10.16).

Листинг 10.16. Перемещение сообщений между списками выбранных и доступных сообщений

```
procedure TfrmMessages.cmbAddMessageClick(Sender: TObject);
var
    i: Integer;
begin
    if lstAvailMessages.SelCount = 0 then Exit;
    //Включение выбранных сообщений в список перехватываемых
    for i := lstAvailMessages.Count - 1 downto 0 do
        if lstAvailMessages.Selected[i] then
            messages_list[GetMessageIndex(i, False)].used := True;
    //Отобразим изменения в списках
    LoadFilter();
end;

procedure TfrmMessages.cmDelMessageClick(Sender: TObject);
var
    i: Integer;
begin
    if lstSelMessages.SelCount = 0 then Exit;
    //Исключение выбранных сообщений из списка перехватываемых
    for i := lstSelMessages.Count - 1 downto 0 do
        if lstSelMessages.Selected[i] then
            messages_list[GetMessageIndex(i, True)].used := False;
```

```
//Отообразим изменения в списках  
LoadFilter();  
end;
```

Функция `GetMessageIndex`, используемая в листинге 10.16, реализована следующим образом (листинг 10.17).

Листинг 10.17. Преобразование номера сообщения в списке в номер сообщения в массиве `messages_list`

```
function TfrmMessages.GetMessageIndex(listIndex: Integer;  
                                     used: Boolean):Integer;  
var  
    i, count: Integer;  
begin  
    count := 0;  
    for i := mess_first to mess_last do  
        if messages_list[i].used = used then  
            begin  
                if count = listIndex then  
                    begin  
                        //Нашли  
                        GetMessageIndex := i;  
                        Exit;  
                    end;  
                    Inc(count);  
                end;  
            end  
            GetMessageIndex := 0;  
        end;  
end;
```

Теперь обратимся к реализации главной функции, выполняемой формой: использованию ловушки. Итак, слежение за выбранным в дереве окном (дескриптор его сохранен в поле `wnd` при инициализации формы) начинается и заканчивается при нажатии кнопки `cmbStart`. Обработчик нажатия этой кнопки приведен в листинге 10.18.

Листинг 10.18. Запуск/остановка перехвата сообщений

```
procedure TfrmMessages.cmbStartClick(Sender: TObject);  
begin  
    if cmbStart.Caption <> 'Остановить' then  
        begin  
            //Начинаем слежение  
            lvwMessages.Clear;
```

```
//Создаем проекцию файла
hFile := CreateFileMapping(INVALID_HANDLE_VALUE, nil,
                           PAGE_READWRITE,
                           0, SizeOf(THookInfo),
                           strFileMapName);

hook_info := MapViewOfFile(hFile, FILE_MAP_WRITE, 0, 0,
                           SizeOf(THookInfo));

//Создание ловушки
if InstallHook(wnd, frmMessages.Handle) then
    cmbStart.Caption := 'Остановить'
else
begin
    //При ошибке удалим проекцию файла
    UnMapViewOfFile(hook_info);
    hook_info := nil;
    CloseHandle(hFile);
    hFile := 0;
    MessageBox(Handle, 'Ошибка при создании ловушки',
                PAnsiChar(Application.Title), MB_ICONEXCLAMATION);
end;
end
else
begin
    //Заканчиваем слежение (удаляем ловушку и проекцию файла)
    RemoveHook();
    UnMapViewOfFile(hook_info);
    hook_info := nil;
    CloseHandle(hFile);
    hFile := 0;
    cmbStart.Caption := 'Начать слежение';
end;
end;
```

Как можно увидеть, вся сложность на стороне приложения-шпиона состоит в создании/удалении проекции файла и в вызове двух экспортируемых из библиотеки `hook.dll` функций. Они подключаются следующим объявлением:

```
function InstallHook(wnd: HWND; spy: HWND): Boolean stdcall;
external 'hook\hook.dll' name 'InstallHook';
function RemoveHook(): Boolean stdcall;
external 'hook\hook.dll' name 'RemoveHook';
```

Для обработки сообщения WM_SPY_NOTIFY, посылаемого ловушкой, переопределена оконная процедура формы frmMessages (листинг 10.19).

Листинг 10.19. Обработка сообщения WM_SPY_NOTIFY

```
procedure TfrmMessages.WndProc(var Message: TMessage);
var
    item: TListItem;
    i: Integer;
begin
    if (Message.Msg = WM_SPY_NOTIFY) and (hook_info <> nil) then
    begin
        //Обрабатываем уведомление о приходе сообщения в наблюдае-
        мое окно
        for i := mess_first to mess_last do
            if (messages_list[i].value = hook_info^.mess) and
                messages_list[i].used then
            begin
                //Сообщение выбрано в фильтре — добавим запись в список
                item := lvwMessages.Items.Add();
                item.Caption := messages_list[i].name;
                item.SubItems.Add(IntToStr(hook_info^.wParam));
                item.SubItems.Add(IntToStr(hook_info^.lParam));
            end;
        end
    else
        inherited WndProc(Message);
    end;
```

Ловушка

Теперь обратимся к реализации самой ловушки. По рассмотренным ранее причинам ловушка размещена в отдельной DLL (hook\hook.dll на прилагаемом к книге диске в папке с номером главы). На случай, если вы не знакомы с созданием DLL средствами Delphi, приведем краткие сведения.

Среда программирования Delphi замечательна тем, что позволяет просто делать довольно сложные вещи. Хотя и при использовании сред разработки, скрывающих меньшее количество сложных деталей, например Visual C++, создание DLL не является очень сложной задачей. Итак, для создания DLL в простейшем, то есть нашем, случае достаточно выполнить следующие действия.

1. Создать соответствующий проект (с помощью команды меню File ► New ► Other, тип проекта — DLL Wizard) (рис. 10.6).

2. В DPR-файле получившегося проекта реализуем функции, которые предполагается экспортировать.
3. Объявляем, какие функции нужно экспортировать с помощью ключевого слова `exports` (листинг 10.20).

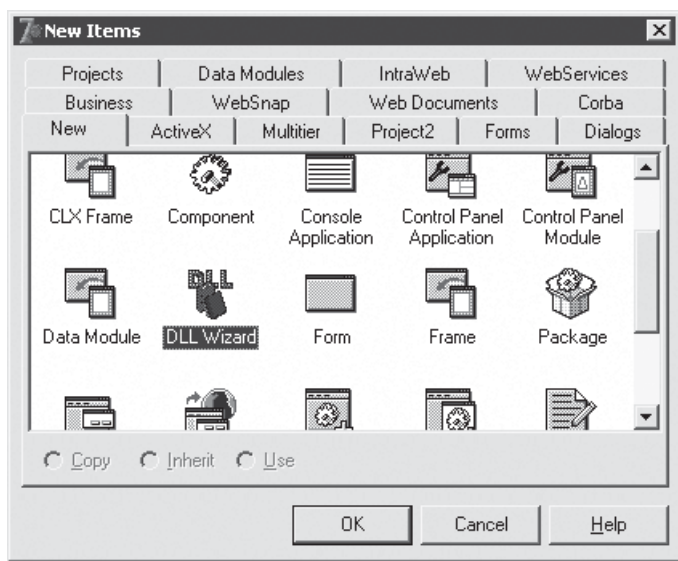


Рис. 10.6. Создание проекта DLL

Структура DLL ловушки, реализованной в нашем примере, приведена в листинге 10.20.

Листинг 10.20. DLL ловушки без реализации функций

```
library hook;
uses
  Windows,
  HookData;
//*****
//Экспортируемые функции
function InstallHook(wnd: HWND; spy: HWND): Boolean stdcall;
forward;
function RemoveHook(): Boolean stdcall; forward;
exports
  InstallHook,
  RemoveHook;
//*****
```

```
...
begin
    hook_info := nil;
    hFile := 0;
end.
```

Код после `begin` является кодом инициализации библиотеки (выполняется при загрузке DLL в память процесса). Правда, как показали многочисленные эксперименты, проведенные во время написания и отладки ловушки, код этот не выполняется при загрузке DLL ловушки в адресное пространство другого процесса.

Теперь обратимся к реализации экспортируемых функций `InstallHook`, а также `RemoveHook`. Как вы помните, только эти две функции вызываются из программы-шпиона. Начнем с функции установки ловушки (листинг 10.21).

Листинг 10.21. Установка (создание) ловушки

```
function InstallHook(wnd: HWND; spy: HWND): Boolean stdcall;
begin
    //Открываем проекцию файла (области файла подкачки)
    if not GetFileMapping() then
    begin
        //Не удалось спроецировать файл в память
        InstallHook := False;
        Exit;
    end;

    //Сохраняем данные, необходимые для работы ловушки
    hook_info^.wnd := wnd;
    hook_info^.spy_wnd := spy;

    //Создаем ловушку
    if (GetWindowThreadProcessId(wnd) <> 0)
    then
        hook_info^.hook_handle :=
            SetWindowsHookEx(WH_CALLWNDPROC, WndProcHook,
                             hInstance, GetWindowThreadProcessId(wnd))
    else
        //Создание ловушки для потоков нашего приложения
        //было бы фатальным
        hook_info^.hook_handle := 0;

    InstallHook := hook_info^.hook_handle <> 0;
    //Освободим проекцию файла
    ReleaseFileMapping();
```

```
end;
```

Функция `InstallHook` использует глобальную переменную-указатель `hook_info`, которая объявлена в модуле `HookData`. Функция `GetFileMapping`, также используемая в листинге 10.21, связывает указатель `hook_info` с областью памяти, на которую проецируется файл. Соответственно, процедура `ReleaseFileMapping` отменяет проецирование файла в память (после этого использовать указатель `hook_info` нельзя).

API-функция `GetWindowThreadProcessId` используется для определения идентификатора потока, создавшего наблюдаемое окно. Проверка неравенства значения, возвращенного этой функцией, нулю используется для того, чтобы в случае закрытия интересующего нас окна до запуска ловушки мы не начали следить за окнами приложения-шпиона.

Работу с проецируемым файлом в ловушке рассмотрим чуть позже. Сейчас же обратимся к функции удаления ловушки, реализация которой приводится в листинге 10.22.

Листинг 10.22. Удаление ловушки

```
function RemoveHook(): Boolean stdcall;  
begin  
  if GetFileMapping() then  
    begin  
      if hook_info^.hook_handle <> 0 then  
        //Удаляем ловушку  
        UnhookWindowsHookEx(hook_info^.hook_handle);  
        //Закрываем проекцию файла  
        ReleaseFileMapping();  
        RemoveHook := True;  
      end  
    else  
      RemoveHook := False;  
    end;  
end;
```

Тут все просто и не требует подробного пояснения. Теперь же рассмотрим так часто используемые функцию и процедуру, работающие с проекцией файла в память. Функция `GetFileMapping`, приведенная в листинге 10.23, открывает проекцию файла в память и связывает указатель `hook_info` с областью памяти, отведенной для проекции файла.

Листинг 10.23. Открытие проекции файла

```
function GetFileMapping(): Boolean;  
begin  
  //Пытаемся открыть проекцию файла  
  hFile := OpenFileMapping(FILE_MAP_WRITE, False,
```

```
        PAnsiChar(strFileMapName));  
//Получаем адрес разделяемой памяти  
hook_info := MapViewOfFile(hFile, FILE_MAP_WRITE, 0, 0,  
                           sizeof(THookInfo));  
GetFileMapping := hook_info <> nil;  
end;
```

Процедура `ReleaseFileMapping`, симметричная по своему назначению функции `GetFileMapping`, реализована так, как показано в листинге 10.24.

Листинг 10.24. Освобождение проекции файла

```
procedure ReleaseFileMapping();  
begin  
    UnmapViewOfFile(hook_info);  
    hook_info := nil;  
    CloseHandle(hFile);  
    hFile := 0;  
end;
```

Функция `GetFileMapping` и процедура `ReleaseFileMapping` используют дополнительно глобальную переменную `hFile` (тип `THandle`), объявленную в модуле `HookData`.

Наконец пришла очередь функции-ловушки. Ее реализация приведена в листинге 10.25.

Листинг 10.25. Функция-ловушка

```
function WndProcHook(code: Integer; wparam: WPARAM;  
                    lparam: LPARAM): LRESULT stdcall;  
var  
    hook_data: ^TCWPStruct;  
begin  
    //Получим доступ к проекции файла  
    if not GetFileMapping() then  
        begin  
            //Не удалось получить доступ к проекции файла. Ценой потери  
            //сообщений не дадим возникнуть ошибкам доступа к памяти  
            WndProcHook := 0;  
            Exit;  
        end;  
end;
```

```
if code < 0 then
begin
    WndProcHook := CallNextHookEx(hook_info^.hook_handle, code,
                                   wParam, lParam);

    //Освободим проекцию файла
    ReleaseFileMapping();
    Exit;
end;

//Можно обрабатывать сообщение
hook_data := Pointer(lParam);
//Обрабатываем только сообщения нужного окна
if hook_data^.hwnd = hook_info^.wnd then
begin
    //Заполняем поля структуры в общей области памяти и посылаем
    //сообщение окну-шпиону
    hook_info^.mess := hook_data^.message;
    hook_info^.wParam := hook_data^.wParam;
    hook_info^.lParam := hook_data^.lParam;
    PostMessage(hook_info^.spy_wnd, WM_SPY_NOTIFY, 0, 0);
end;

//Передаем сообщение для дальнейшей обработки
WndProcHook := CallNextHookEx(hook_info^.hook_handle, code,
                               wParam, lParam);

//Освободим проекцию файла
ReleaseFileMapping();
end;
```

Код функции `WndProc` достаточно прост, поэтому не будем подробно его описывать. Поясним лишь, для чего все-таки `GetFileMapping` и `ReleaseFileMapping` вызываются при обработке каждого перехваченного сообщения.

Дело в том, что загрузка DLL в адресное пространство другого процесса отличается от штатной загрузки библиотеки, например, при помощи функции `LoadLibrary`: не вызывается код инициализации. Следовательно, мы не можем, например, обнулить указатель `hook_info` или установить еще какой-либо признак того, была ли открыта проекция файла. Велика вероятность того, что без отсутствия ручной иници-

циализации указатель `hook_info` не будет равен нулю. Как тогда определить, связан ли этот указатель с областью памяти, куда спроецирован файл?

Можно было бы, конечно, завести 64-битную или более переменную, которой присваивалось бы «магическое» число при первой инициализации указателя `hook_info`. Но в таком случае работоспособность нашей программы носила бы вероятностный характер.

Речь не идет о том, что в приведенном примере ловушка реализована самым оптимальным образом, просто альтернатива `GetFileMapping` и `ReleaseFileMapping` при написании программы показалась наиболее простой и легко поддающейся объяснению.



Глава 11

Сетевое взаимодействие

- ☐ Краткое описание сетевых компонентов
- ☐ Простой обмен данными
- ☐ Слежение за компьютером по сети
- ☐ Многопользовательский разговорник

Организация надежного сетевого взаимодействия между приложениями или компонентами одного приложения зачастую является задачей довольно сложной даже для программиста со значительным опытом работы. Это правда, если пытаться самостоятельно использовать API сетевого взаимодействия, предоставляемый операционной системой (в нашем случае — Windows). Однако с использованием компонентов Delphi, в которых уже реализованы рутинные операции по созданию соединений, пересылке данных, контролю ошибок и т. д., программирование сетевых приложений становится не только простым, но и увлекательным занятием.

В данной главе мы рассмотрим несколько примеров создания несложных сетевых приложений, построенных с использованием архитектуры «клиент — сервер».

11.1. Краткое описание сетевых компонентов

В Delphi 7 количество компонентов для программирования самых различных сетевых приложений просто радует глаз (см. вкладки `IndyClients` и `IndyServers`). Мы рассмотрим построение приложения на базе только `IdTCPServer` и `IdTCPClient` (написание клиент-серверных приложений с использованием всех сетевых компонентов могло бы занять всю книгу).

Итак, сначала о компоненте сервера `IdTCPServer`. Для использования возможностей сервера этот компонент нужно поместить на форму (компонент неотображаемый). При настройке компонента полезными являются следующие его свойства:

- `Active` — активизирует или деактивизирует сервер (по умолчанию `False`);
- `Bindings` — настраивает серверные сокеты (присоединяет их к определенному порту компьютера, позволяет задавать диапазон IP-адресов и портов клиентов) при помощи диалогового окна `Binding Editor`;
- `ListenQueue` — численное значение, ограничивающее максимальное количество запросов на установление соединения от клиентов в очереди;
- `MaxConnections` — позволяет ограничить максимальное количество клиентов, присоединенных к серверу;
- `MaxConnectionReply` — позволяет настроить сообщение, посылаемое сервером новым клиентам, когда их количество достигает `MaxConnections`.

Рассмотрим несколько подробнее настройку серверных гнезд с использованием свойства `Bindings`. Так, на рис. 11.1 показано, как при помощи диалогового окна `Binding Editor` настроить сервер на обслуживание клиентов с любыми IP-адресами, при этом серверный сокет присоединяется к порту 12340.

Для более детальной настройки каждого серверного сокета можно использовать окна `Object TreeView` и `Object Inspector` так, как показано на рис. 11.2.

На этом настройку сервера можно и завершить (хотя здесь используются далеко не все возможности компонента `IdTCPServer`). Основная же работа сервера при обработке

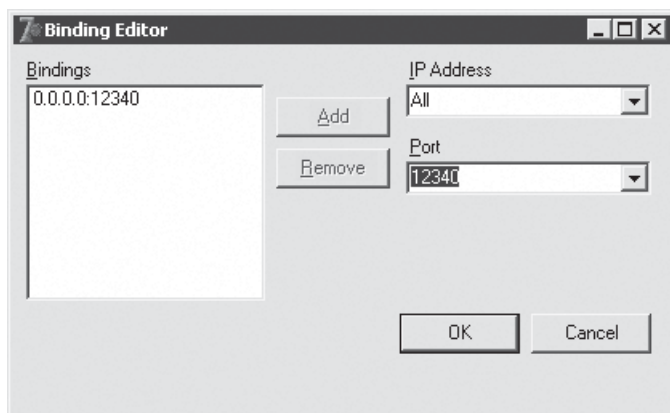


Рис. 11.1. Использование окна Binding Editor

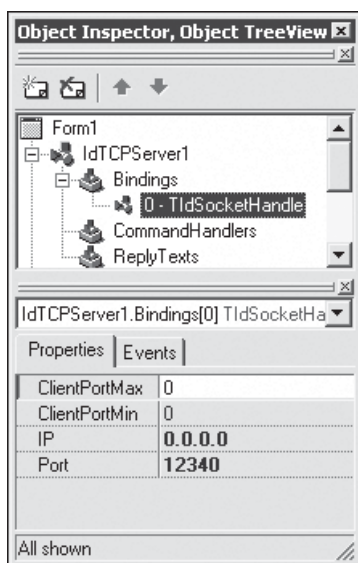


Рис. 11.2. Настройка серверного гнезда

запросов клиентов может реализоваться в обработчике события OnExecute. В этот обработчик передается ссылка на объект TIdPeerThread — поток, ассоциированный с клиентом, присоединенным к серверу. Посредством этого объекта (а точнее, его свойства Connection) можно получать и отправлять данные, а также получать и устанавливать множество полезных параметров соединения. Первый пример использования объекта TIdPeerThread при обработке запроса клиента приведен в листинге 11.1.

Теперь рассмотрим, как сконфигурировать клиент (IdTCPClient), чтобы он был способен взаимодействовать с нашим сервером. Чтобы использовать компонент TSP-клиента, достаточно поместить его на форму (компонент также неотображаемый).

После этого как минимум нужно настроить следующие его свойства (остальные упоминаются по мере необходимости в приведенных далее примерах):

- `Host` — имя или IP-адрес компьютера, на котором запущен сервер;
- `Port` — номер порта, к которому присоединен серверный сокет.

Вообще, даже эти свойства на этапе разработки формы настраивать не обязательно. Приложение получается гораздо более гибким, если давать, например, пользователю возможность выбрать (или ввести) имя или адрес сервера.

11.2. Простой обмен данными

В начале работы с описанными в предыдущем разделе компонентами `IdTCPServer` и `IdTCPClient` рассмотрим создание несложного клиент-серверного приложения, клиентская и серверная части которого выполняют следующие функции.

- Клиентское приложение соединяется с сервером и отправляет ему введенную пользователем строку, ждет ответа, выводит полученный от сервера текст, отсоединяется от сервера.
- Серверное приложение принимает строку от клиентского приложения и посылает ответ (также текстовый), после чего разрывает соединение. Плюс к этому ведется подсчет количества обслуженных клиентов и запоминается IP-адрес компьютера, с которого пришел последний запрос.

Реализация как серверного, так и клиентского приложений в нашем случае предельно проста. Проект серверного приложения называется `SimpleServer`. Внешний вид формы сервера (во время работы приложения) представлен на рис. 11.3.

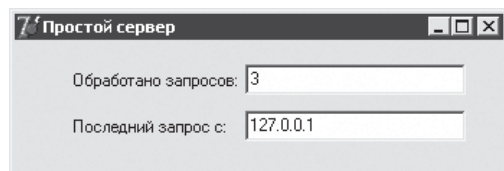


Рис. 11.3. Внешний вид простого сервера

Текстовое поле (Edit) с количеством обработанных запросов имеет имя `txtCount`, а текстовое поле с адресом последнего обслуженного компьютера названо `txtFrom`. Вся работа сервера заключается в обработке события `Execute` для компонента `IdTCPServer`, помещенного на форму (присоедините этот компонент к порту 12340 и установите значение свойства `Active = True`) (листинг 11.1).

Листинг 11.1. Реализация простого сервера

```
procedure TForm1.IdTCPServer1Execute(AThread: TIdPeerThread);  
var  
    strText: string;
```

```
begin
    //Принимаем от клиента строку
    strText := AThread.Connection.ReadLn;
    //Отвечаем
    AThread.Connection.WriteLine('Принял строку:' + strText);
    //Обновим сведения на форме сервера (сервер многопоточный,
    //поэтому используем синхронизацию)
    section.Enter;
    Inc(processed,1);
    txtCount.Text := IntToStr(processed);
    txtFrom.Text := AThread.Connection.Socket.Binding.PeerIP;
    section.Leave;
    //Закрываем соединение с пользователем
    AThread.Connection.Disconnect;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
    section := TCriticalSection.Create;
end;
```

При ответе клиенту сервер только повторяет принятую от него строку с добавлением текста 'Принял: ' в начало строки.

Анализируя листинг 11.1, можно заметить, что даже в рассматриваемом простейшем сервере пришлось применить синхронизацию при обновлении внешнего вида формы при помощи критической секции (необходимо дополнительно добавить имя модуля `SyncObjs` в секцию `uses`).

Теперь рассмотрим реализацию клиентской части (проект `SimpleClient`). Внешний вид клиентского приложения приведен на рис. 11.4.

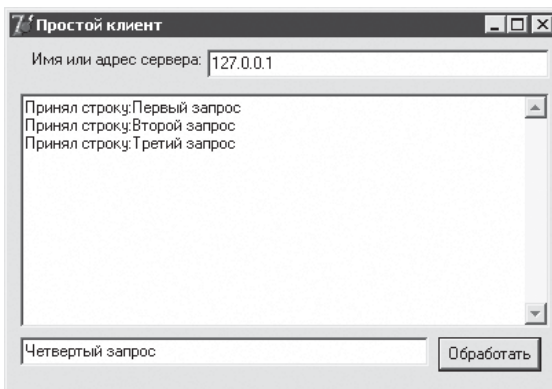


Рис. 11.4. Внешний вид клиента

Естественно, что для работы клиентского приложения на форму помещен экземпляр компонента `IdTCPClient` (его имя — `IdTCPClient1`). Свойству `Port` этого компонента нужно присвоить значение 12340. Текстовое поле (Edit) для ввода строки, подлежащей отправке на сервер, имеет имя `txtMessage`. Текстовое поле (Edit), в которое вводится имя или адрес сервера, названо `txtServer`. Поле со строками ответов (Мемо) имеет имя `txtResults`.

Вся работа клиентского приложения выполняется при нажатии кнопки **Обработать**. Текст соответствующего обработчика приведен в листинге 11.2.

Листинг 11.2. Реализация простого клиента

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    //Соединяемся с сервером и посылаем ему введенную строку
    IdTCPClient1.Host := txtServer.Text;
    IdTCPClient1.Connect;
    IdTCPClient1.WriteLine(txtMessage.Text);
    txtMessage.Text := '';
    //Ожидаем ответ и закрываем соединение
    txtResults.Lines.Append(IdTCPClient1.ReadLn);
    IdTCPClient1.Disconnect;
end;
```



ПРИМЕЧАНИЕ

Для простоты в реализации клиентского приложения не производится обработка исключений, генерация которых возможна, например, при неправильном указании компьютера, на котором запущено серверное приложение. В более сложных примерах, с которыми вы познакомитесь далее в этой главе, обработка указанных исключений реализована.

Все, теперь можно запускать сервер и клиенты (на произвольном количестве компьютеров) и понаблюдать за результатами их работы. Только не забудьте запустить сервер до того, как будете обращаться к нему с помощью программы-клиента.

11.3. Слежение за компьютером по сети

Теперь рассмотрим более интересный пример использования сетевых компонентов `IdTCPServer` и `IdTCPClient`, который может быть полезен для людей, имеющих отношение к администрированию компьютеров сети.

Серверная программа предварительно запускается на наблюдаемом компьютере. В этом примере программа-сервер позволяет клиентской программе получать следующие сведения о компьютере, на котором она (программа-сервер) запущена:

- разрешение монитора;
- глубину цвета для монитора;
- полноразмерную копию экрана;
- копию экрана, уменьшенную (или увеличенную) до заданных размеров.

Для получения указанных сведений программа-клиент должна послать серверу следующие строковые значения:

- `get_screen_width` — для получения ширины и `get_screen_height` — для получения высоты экрана в пикселах;
- `get_screen_colors` — для получения значения установленной для монитора глубины цвета (бит на точку);
- `get_screen` — для получения полноразмерной копии экрана;
- `get_screen:X,Y` — для получения копии экрана, приведенной к размеру X×Y.

Сначала рассмотрим реализацию сервера (проект SpyServer). Весь код, обеспечивающий работу сервера, помещен в модуле `Unit1.pas` формы `Form1`. Обработчик запросов клиентов — главная процедура для сервера — приводится в листинге 11.3.

Листинг 11.3. Обработчик клиентских запросов

```
procedure TForm1.IdTCPServer1Execute(AThread: TIdPeerThread);
var
    strText: string;
    width, height, i: Integer;
    dc: HDC;
begin
    //Принимаем от клиента строку
    strText := AThread.Connection.ReadLn;
    //Определяем, что нужно выполнить
    if (strText = 'get_screen_height') then
        //Возвратим высоту экрана
        AThread.Connection.WriteInteger(Screen.Height)
    else if (strText = 'get_screen_width') then
        //Возвратим ширину экрана
        AThread.Connection.WriteInteger(Screen.Width)
    else if (strText = 'get_screen_colors') then
        begin
            //Возвратим количество бит на точку
            dc := GetDC(0);
            AThread.Connection.WriteInteger(GetDeviceCaps(dc,
                BITSPIXEL));
```

```

    ReleaseDC(0, dc);
end
else if (strText = 'get_screen') then
    //Возвратим полноразмерную копию экрана
    SendScreen(Screen.Width, Screen.Height, AThread.Connection)
else begin //строка вида 'get_screen:x,y'
    //Определим значения высоты и ширины,
    //переданные пользователем
    strText := Copy(strText, 12, Length(strText)-11);
    i := Pos(',', strText); //Положение запятой
    width := StrToInt(Copy(strText, 1, i-1));
    height := StrToInt(Copy(strText, i+1, Length(strText)-i));
    //Возвратим копию экрана
    SendScreen(width, height, AThread.Connection);
end;
end;
end;

```

Используемая в листинге 11.3 процедура `SendScreen`, отправляющая клиенту копию экрана, приведена в листинге 11.4.

Листинг 11.4. Снятие копии экрана

```

//Процедура снимает копию экрана, приводит полученное
//изображение к заданному размеру и отправляет
//преобразованное изображение клиентской программе
procedure SendScreen(width: Integer; height: Integer;
                    Connection: TIdTCPServerConnection);
var
    ScreenCopy: TCanvas;
    gr: TBitmap;
    stream: TMemoryStream;
    rcDest, rcSource: TRect;
begin
    rcDest := Rect(0, 0, width, height); //Конечный размер
                                         //изображения
    rcSource := Screen.DesktopRect;      //Исходный размер
                                         //изображения

    //Создаем канву и присоединяем ее к контексту Рабочего стола
    ScreenCopy := TCanvas.Create;
    ScreenCopy.Handle := GetDC(0);
    //Создаем объект для хранения копии экрана
    //и копируем изображение

```

```
gr := TBitmap.Create;  
gr.Height := height;  
gr.Width := width;  
gr.Canvas.CopyRect(rcDest, ScreenCopy, rcSource);  
ReleaseDC(0, ScreenCopy.Handle);  
//Сохраняем изображение в поток данных  
stream := TMemoryStream.Create;  
gr.SaveToStream(stream);  
//Отправляем изображение клиенту  
Connection.WriteStream(stream, True, True);  
  
stream.Clear;  
stream.Free;  
gr.Free;  
end;
```

Как можно увидеть, даже самая сложная операция рассматриваемого сервера — копирование изображения — реализуется довольно просто благодаря наличию такого стандартного класса, как `TMemoryStream`.

При реализации сервера использован таймер. Он применен для скрывтия формы сервера сразу при запуске приложения (не забудьте установить значения его свойств `Enabled = True` и `Interval = 50`). Компонент `IdTCPServer` (с именем `IdTCPServer1`) в этом примере присоединен к порту 12341 (не забудьте также установить свойство `Active = True`).

Теперь о реализации клиентского приложения (проект `SpyClient`). Внешний вид формы (Form1) клиента во время работы приводится на рис. 11.5 (видно, что пользователь наблюдаемого компьютера только что проиграл в игру `Сапер`).

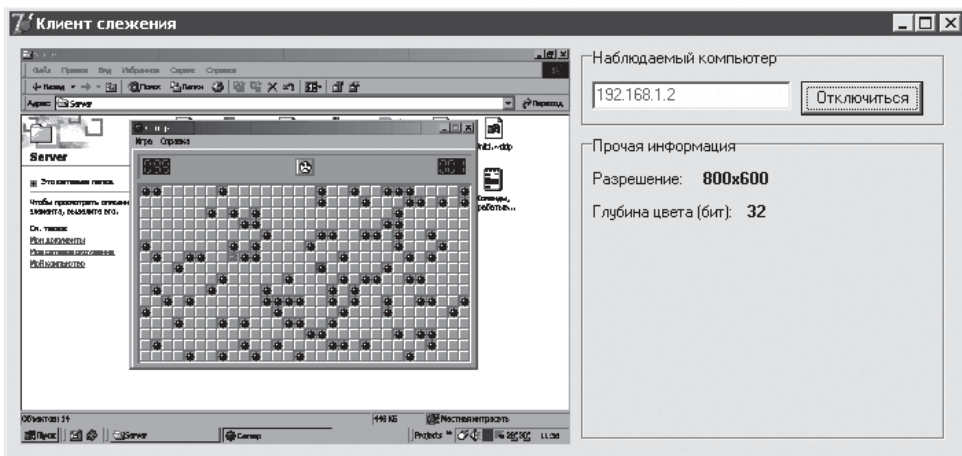


Рис. 11.5. Внешний вид клиента слежения

Описания, имена и значения настроенных вручную свойств самых важных компонентов формы клиента приведены в табл. 11.1.

Таблица 11.1. Основные компоненты формы клиента слежения и их свойства

Название (свойство Name)	Описание	Измененные свойства и их новые значения
imgScreen	Компонент Image для хранения изображения, полученного с сервера	Stretch = True Пропорции: ширина:высота = 4:3, например, 400 × 300
txtServer	Текстовое поле (Edit) для ввода имени или адреса компьютера сервера	Отсутствует
cmbConnect	Кнопка (Button) для начала или завершения наблюдения за компьютером	Caption = «Подключиться»
lblResolution	Надпись (Label) со значением разрешения монитора наблюдаемого компьютера	Font.Style = [fsBold]
lblColors	Надпись (Label) со значением глубины цвета монитора наблюдаемого компьютера	Font.Style = [fsBold]
IdTCPClient1	Компонент (IdTCPClient) для связи с сервером	Port = 12341
Timer1	При каждом событии от этого таймера (Timer) выполняется запрос информации с сервера	Interval = 5000 Enabled = False

Работа клиентского приложения начинается с соединения с сервером. Код, отвечающий за эту операцию, приведен в листинге 11.5.

Листинг 11.5. Соединение с сервером

```
procedure TForm1.cmbConnectClick(Sender: TObject);
begin
    if (cmbConnect.Caption = 'Подключиться') then
    begin
        if (txtServer.Text = '') then
            //Не введено имя сервера
            MessageDlg('Введите имя машины-сервера в текстовое поле',
                mtInformation, [mbOK], 0)
        else begin
            //Подключаемся к серверу
            IdTCPClient1.Host := txtServer.Text;
```

```
    try
        IdTCPClient1.Connect;
    except
        MessageDlg('Не удастся соединиться с указанным сервером',
            mtError, [mbOK], 0);
        Exit;
    end;
end
end
else begin
    //Отключаемся от сервера
    IdTCPClient1.Disconnect;
end;
end;
```

Если соединение с сервером произошло успешно, то выполняется обработчик `TForm1.IdTCPClient1Connected`, подготавливающий приложение-клиент к периодическим запросам данных с сервера (листинг 11.6).

Листинг 11.6. Действия, выполняемые при соединении с сервером

```
procedure TForm1.IdTCPClient1Connected(Sender: TObject);
begin
    txtServer.Enabled := False;
    cmbConnect.Caption := 'Отключиться';
    //Начинаем периодически запрашивать данные с сервера
    Timer1.Enabled := True;
    //Выполним первый запрос, не дожидаясь срабатывания таймера
    Timer1Timer (Nil);
end;
```

При отсоединении от сервера также выполняются действия, прекращающие периодические запросы данных и переводящие клиент в состояние ожидания подключения (первоначальное состояние программы) (листинг 11.7).

Листинг 11.7. Действия при отсоединении от сервера

```
procedure TForm1.IdTCPClient1Disconnected(Sender: TObject);
begin
    txtServer.Enabled := True;
    cmbConnect.Caption := 'Подключиться';
    Timer1.Enabled := False;
end;
```

Самой сложной частью клиентского приложения является обработка данных, присылаемых сервером. Клиентское приложение запрашивает данные по таймеру и обрабатывает полученные данные так, как показано в листинге 11.8.

Листинг 11.8. Запрос и обработка данных, полученных с сервера

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
    stream: TMemoryStream;
begin
    //Запрашиваем у сервера данные о наблюдаемом компьютере
    with (IdTCPClient1) do
    begin
        //...разрешение
        WriteLn('get_screen_width');
        WriteLn('get_screen_height');
        lblResolution.Caption := IntToStr(ReadInteger) + 'x' +
                                IntToStr(ReadInteger);

        //...глубина цвета
        WriteLn('get_screen_colors');
        lblColors.Caption := IntToStr(ReadInteger);

        //...копия экрана
        //.....первый вариант – копирование экрана без сжатия
        WriteLn('get_screen');
        //.....второй вариант – сжатие на стороне сервера
        WriteLn('get_screen:' + IntToStr(imgScreen.Width) + ',' +
                IntToStr(imgScreen.Height));

        //....получаем данные
        stream := TMemoryStream.Create;
        ReadStream(stream);
        stream.Position := 0;
        //....формируем изображение
        imgScreen.Picture.Bitmap.LoadFromStream(stream);
        stream.Clear;
        stream.Free;
    end;
end;
```

В тексте листинга 11.8 создано большое количество комментариев, поэтому дополнительно пояснять его нет смысла. Остановимся лишь на том, зачем в процедуре `TForm1.Timer1Timer` предусмотрено два варианта получения изображения с сервера.

Все дело в том, что сжатие (в нашем примере разрешение экрана наблюдаемого компьютера больше размера компонента `imgScreen`) на стороне сервера требует от компьютера, на котором запущено серверное приложение, большего процессорного времени на снятие копии экрана. Это снижает нагрузку на сеть при передаче изображения, а также экономит ресурсы компьютера-клиента. Но качество сжатого изображения в этом случае получается несколько хуже, чем когда мы предоставляем компоненту `Image` возможность масштабировать изображение самостоятельно.

Если же не использовать сжатие изображения на сервере, возрастает нагрузка на сеть при передаче полноразмерной копии экрана, а вся работа по сжатию изображения возлагается на компонент `imgScreen` (то есть дополнительно тратится процессорное время на компьютере клиента). При большом разрешении экрана наблюдаемого компьютера (или при наблюдении сразу за несколькими компьютерами) машина клиента, если она недостаточно мощная, может начать весьма ощутимо «тормозить». Качество сжатого изображения при этом получается более высоким.

В качестве более-менее эффективного решения можно предложить использовать большие промежутки времени между запросами данных с сервера слежения с масштабированием изображения на серверной стороне (если только машина сервера не является очень маломощной).

11.4. Многопользовательский разговорник

В завершение знакомства с компонентами `IdTCPClient` и `IdTCPServer` для организации сетевого взаимодействия рассмотрим создание полноценного клиент-серверного приложения — многопользовательского разговорника. Как можно догадаться из названия, это приложение будет позволять обмениваться сообщениями большому количеству пользователей (наподобие чата).

Поскольку этот пример будет несколько сложнее (в плане организации сетевого взаимодействия) предыдущих примеров главы, то рассмотрим подробно основные этапы его проектирования, разработки и реализации (начиная с требований и поведения клиента и сервера и заканчивая нюансами реализации приложений).

Требования к клиентскому и серверному приложениям

Пользователи при работе с клиентскими приложениями должны иметь следующие возможности:

- видеть полный текст разговора с момента их подключения к серверу;
- отправлять сообщения как всем, так и только определенным пользователям;

- видеть список пользователей, участвующих в разговоре (при этом список должен автоматически обновляться при отключении или присоединении новых пользователей);
- получать уведомления об отключении или присоединении новых пользователей (прямо в тексте разговора).

Серверное приложение, кроме управления подключением, отключением пользователей, а также доставки сообщений, должно обеспечивать протоколирование событий (подключение, отключение пользователей, от кого и кому послано то или иное сообщение).

При реализации серверного приложения нужно преодолеть некоторые сложности, связанные с тем, что к серверу будут постоянно подключены сразу несколько пользователей, причем информация о каждом пользователе будет постоянно храниться и использоваться сервером. Нужно также обеспечить надежную работу клиентского, а главное — серверного приложения при проблемах, связанных с неисправностями сети.

И, наконец, нужно обеспечить автоматическую рассылку клиентским приложениям следующей информации (клиенты эту информацию специально с сервера не запрашивают):

- текста сообщений;
- уведомлений о присоединении или отсоединении пользователей.

Формат сообщений клиента и сервера

Клиент и сервер обмениваются только текстовыми сообщениями (не путать с сообщениями, которыми обмениваются пользователи в ходе разговора). Строка любого сообщения состоит из двух частей: префикса и текста сообщения. Префикс отделяется от текста сообщения символом `:` (двоеточие). По префиксу можно определить, что делать с полученным сообщением.

Возможны следующие сообщения от клиента серверу:

- `name:имя_пользователя` — при помощи этого сообщения клиентская программа сообщает серверу, под каким именем зарегистрировать пользователя (это имя будут видеть другие пользователи);
- `text:текст` — при получении этого сообщения сервер должен разослать текст всем участникам разговора (включая отправителя);
- `имя_адресата:текст` — при получении этого сообщения сервер должен отправить текст только заданному префиксом пользователю `имя_адресата`, а также должен отправить копию автору сообщения.

К сообщениям третьего типа относятся все сообщения, принимаемые сервером и не начинающиеся с `text:` или `name:`.

В свою очередь, сервер может посылать клиентской программе сообщения следующего вида:

- `ok:` — означает, что пользователь зарегистрирован и может вступать в разговор;
- `error:сообщение_об_ошибке` — означает, что по каким-то причинам пользователь не может участвовать в разговоре. При получении этого сообщения клиентская программа должна показать окно с текстом `сообщение_об_ошибке` и разорвать соединение с сервером;
- `adduser:имя_пользователя` — при получении такого сообщения клиентская программа должна добавить строку `имя_пользователя` в список участников разговора;
- `deluser:имя_пользователя` — при получении такого сообщения клиентская программа должна удалить строку `имя_пользователя` из списка участников разговора;
- `text:текст` — клиентская программа должна добавить `текст` к тексту разговора.

Перед рассмотрением реализации клиентской и серверной частей скажем несколько слов об использовании специальных сообщений клиента (`name:имя_пользователя`) и сервера (`ok:` и `error:сообщение_об_ошибке`). Дело в том, что в предлагаемой реализации сервера присоединение нового пользователя к разговору происходит следующим образом.

1. Клиентское приложение присоединяется к серверу (количество пользователей ограничено, поэтому сервер может послать лишнему пользователю сообщение `error:` с соответствующим текстом, описывающим ошибку, и тут же разорвать установленное соединение).
2. Клиентское приложение посылает серверу сообщение с именем пользователя (префикс `name:`).
3. Если имя, под которым хочет зарегистрироваться новый пользователь, используется, то клиентскому приложению отправляется сообщение `error:` с пояснением ошибки.
4. Если имя свободно, то сервер сохраняет его (и рассылает его всем остальным клиентским приложениям), а также посылает приложению присоединенного пользователя список всех остальных пользователей, и только после этого дает новому пользователю возможность участвовать в разговоре (сообщение `ok:`).

Остальные нюансы будут рассмотрены при описании исходного кода клиентского и серверного приложений.

Реализация сервера

Серверное приложение реализовано с оконным интерфейсом. Форма `frmServer` приложения во время работы представлена на рис. 11.6.

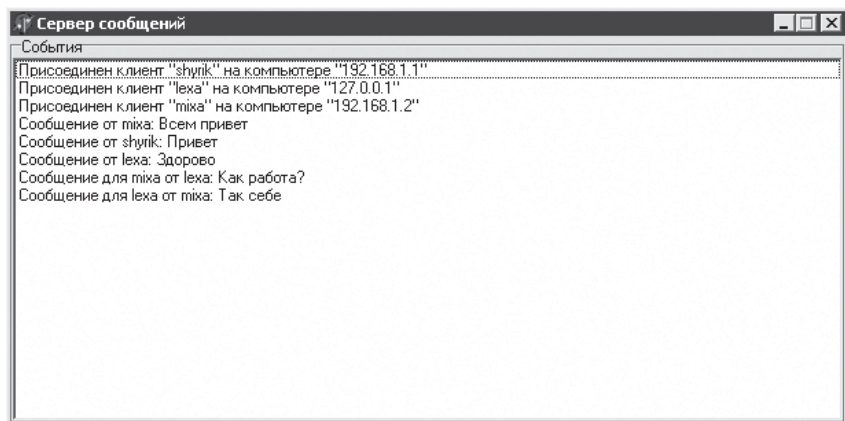


Рис. 11.6. Форма сервера сообщений

Элемент управления `ListBox` (имя `lstEvents`), который можно увидеть на форме, предназначен для вывода списка событий (присоединение, отсоединение клиентов, передача сообщений). Список помещается в рамку `GroupBox1`. Для списка и рамки задано значение свойства `align = client`.

Кроме перечисленных элементов управления, на форму также помещены компоненты `IdTCPServer` (имя `TCPServer`) и `Timer` (имя `Timer1`). Для таймера задаются значения свойств `Enabled = True` и `Interval = 50`. Компонент `TCPServer` настраиваем на прослушивание порта 12345, а также устанавливаем значение свойства `Active = True`.

При реализации сервера основной программный код помещен в файле формы (`Unit1.pas`). Этот модуль условно можно разделить на две части: в первой реализованы специальные функции и процедуры (регистрации пользователей, пересылки текстовых сообщений между пользователями и т. д.), во второй части следуют процедуры-обработчики событий (методы класса `TfrmServer`).

Сначала рассмотрим процедуры обработки событий, так как они значительно проще, чем остальные функции и процедуры, и их рассмотрение вначале позволит лучше представить функционирование приложения (листинг 11.9).

Листинг 11.9. Процедуры обработки событий серверного приложения (`Unit1.pas`)

```
procedure TfrmServer.Timer1Timer(Sender: TObject);
begin
    //Если нужно, то скроем окно сервера
    if (not SERVERVISIBLE) then
    begin
        frmServer.Visible := False;
        ShowWindow(Application.Handle, SW_HIDE);
    end;
```

```
//Таймер больше не нужен
Timer1.Enabled := False;
end;

procedure TfrmServer.TCPServerExecute(AThread: TIdPeerThread);
begin
    //Обрабатываем сообщение, пришедшее от клиента
    ProcessMessage(AThread.Connection, AThread.Connection.ReadLn);
end;

procedure TfrmServer.TCPServerConnect(AThread: TIdPeerThread);
begin
    //Попытаемся добавить нового пользователя
    if (AddClient(AThread.Connection)) then
        //Пользователь должен прислать свое имя
        ProcessMessage(AThread.Connection, AThread.Connection.ReadLn)
    else
        begin
            //Нет места для нового пользователя
            AThread.Connection.WriteLine('error:Достигнуто максимальное
количество ' + 'пользователей. Извините, невозможно принять вас
в разговор.');
```

АThread.Connection.Socket.Close;

```
        end;
    end;

procedure TfrmServer.TCPServerDisconnect(AThread: TIdPeerThread);
var clDisconnected: client; //Структура с информацией об
                            //отсоединенном клиенте (заполнены
                            //только поля strName и strIP)
begin
    //Удалим информацию об отсоединенном клиенте
    clDisconnected := DeleteClient(AThread.Connection);
    if (clDisconnected.strName <> '') then
        begin
            //Сообщим о событии остальным клиентам
            SendAll('deluser:' + clDisconnected.strName);
            SendAll('Нас покинул "' + clDisconnected.strName + '".');
```

```
//Добавим событие в журнал
if (REPORT) then AddEvent('Отсоединился клиент ' +
    clDisconnected.strName + ' на компьютере ' +
    clDisconnected.strIP + '');
end;
end;

procedure TfrmServer.FormCreate(Sender: TObject);
begin
    //Создаем критическую секцию
    section := TCriticalSection.Create;
end;
```

Первая и последняя из приведенных в листинге 11.9 процедур не имеют непосредственного отношения к работе TCP-сервера. Процедура `TfrmServer.Timer1Timer` вызывается только один раз при первом срабатывании таймера `Timer1`. В ней, исходя из заданного значения глобальной переменной `SERVERVISIBLE`, происходит (или не происходит) скрытие окна сервера. Значение глобальной переменной `SERVERVISIBLE` (и переменной `REPORT`) определяется в момент запуска сервера.

Процедура `TfrmServer.FormCreate` создает объект синхронизации, используемый остальными функциями и процедурами для предотвращения одновременного доступа к общим данным нескольких потоков (ведь сервер-то у нас многопоточный).

Остальные три процедуры используются непосредственно для организации взаимодействия сервера с клиентами. Как было сказано ранее, сервер хранит информацию о присоединенных к нему клиентах. Хранилищем этой информации является массив структур (подробно он будет рассмотрен немного ниже). Здесь же необходимо сказать, что при присоединении к серверу нового клиента (процедура `TfrmServer.TCPServerConnect`) предпринимается попытка найти для информации о новом пользователе место в указанном массиве (вызов функции `AddClient`). Если место нашлось, то функция `AddClient` возвращает `True`, и сервер переходит в режим регистрации пользователя. Для регистрации клиентская программа должна передать серверу имя пользователя (сообщение с префиксом `name:`).

Особенностью реакции сервера на отключение клиентской программы (процедура `TfrmServer.TCPServerDisconnect`) является то, что, помимо удаления информации об отсоединившемся клиенте (вызов функции `DeleteClient`), все остальные пользователи уведомляются об отсоединении собеседника (вызовы функции `SendAll`).

При получении сообщения от клиента (процедура `TfrmServer.Execute`) происходит всего лишь передача полученной строки функции `ProcessMessage`, которая и занимается анализом текста сообщения и определением действий, которые сервер должен выполнять.

Теперь рассмотрим функции и процедуры, которые прямо или косвенно используются описанными выше обработчиками событий и на которых по большей части и основывается работа серверного приложения. Часть файла `Unit1.pas`, содержащая объявление типов данных, переменных и подключения модулей (добавленные вручную), которые нужны для работы сервера, приведена в листинге 11.10.

Листинг 11.10. Типы данных и переменные серверного приложения (`Unit1.pas`)

```
unit Unit1;
interface
uses
    ..., SyncObjs;
type
    TfrmServer = class(TForm)
        lstEvents: TListBox;           //Список событий
        ...
    end;
var
    frmServer: TfrmServer;
    REPORT: Boolean;                  //Если = True, то все события
                                      //записываются в ListBox
                                      //окна сервера
    SERVERVISIBLE: Boolean; //Если = True, то окно показывается
                              //на экране и приложение есть
                              //на Панели задач

implementation
//Следующая структура используется для хранения информации
//о пользователе, подключившемся к серверу
type
    client = record
        fUsed: Boolean; {Ячейка занята}
        fNamed: Boolean; {Клиент сообщил свое имя}
        strName: string; {Имя пользователя}
        strIP: string; {IP-адрес клиента}
        Connection: TIdTCPServerConnection; {Соединение клиента
                                              с сервером}
    end;
const
    MAX_CLIENT = 100; //Максимальное количество кlients
```

```
var
  clients: array [1..MAX_CLIENT] of client; //Массив со
                                              //сведениями о клиентах
  section: TCriticalSection; //Критическая секция
                               //для синхронизации потоков
```

Процедура, записывающая событие в журнал (ListBox на форме сервера), приведена в листинге 11.11.

Листинг 11.11. Добавление события в журнал сервера

```
procedure AddEvent(strEvent: string);
begin
  section.Enter;
  frmServer.lstEvents.Items.Append(strEvent);
  section.Leave;
end;
```

В листинге 11.12 приводится процедура, рассылающая текстовое сообщение всем присоединенным к серверу клиентам.

Листинг 11.12. Рассылка сообщения всем клиентам

```
procedure SendAll(strMessage: string);
var
  i: Integer;
begin
  for i:=1 to MAX_CLIENT do
    if (clients[i].fNamed) then
      begin
        try
          clients[i].Connection.WriteLine(strMessage);
        except
          //При возникновении ошибки отключим клиента
          //и продолжим рассылку
          ErrorCloseConnection(clients[i].Connection);
        end;
      end;
  end;
end;
```

Далее, в листинге 11.13, приведена процедура, посылающая текстовое сообщение strMessage клиенту с заданным именем strName.

Листинг 11.13. Посылка сообщения клиенту с заданным именем

```
procedure SendTo(strMessage: string; strName: string);
var
    i: Integer;
begin
    for i:=1 to MAX_CLIENT do
        if (clients[i].fNamed) then
            if (clients[i].strName = strName) then
                //Нашли клиента с заданным именем
                try
                    clients[i].Connection.WriteLine(strMessage);
                except
                    //При возникновении ошибки отключим клиента
                    //и продолжим рассылку
                    ErrorCloseConnection(clients[i].Connection);
                end;
            end;
    end;
```

Процедура, приведенная в листинге 11.14, находит и помечает как занятую для нового пользователя запись в массиве `clients`. Если свободных записей в массиве не осталось, то достигнуто максимальное количество пользователей.

Листинг 11.14. Добавление информации о новом клиенте

```
function AddClient(Connection: TIdTCPServerConnection): Boolean;
var
    i: Integer;
begin
    section.Enter;
    for i:=1 to MAX_CLIENT do
        begin
            if (not clients[i].fUsed) then
                begin
                    //Нашли свободную запись — заполним ее
                    //(клиент пока безымянный)
                    clients[i].fUsed := True;
                    clients[i].Connection := Connection;
                    clients[i].strIP := Connection.Socket.Binding.PeerIP;
                    AddClient := True;
                end;
            end;
        end;
```

```
        section.Leave;  
        Exit;  
    end;  
end;  
section.Leave;  
AddClient := False;  
end;
```

Процедура `DeleteClient`, приведенная в листинге 11.15, освобождает запись заданного пользователя в массиве `clients`.

Листинг 11.15. Удаление информации о клиенте

```
function DeleteClient(Connection: TIdTCPServerConnection):client;  
var  
    i: Integer;  
begin  
    section.Enter;  
    for i:=1 to MAX_CLIENT do  
        if (clients[i].fUsed) then  
            if (clients[i].Connection = Connection) then  
                begin  
                    //Вот она — запись о нужном клиенте  
                    clients[i].fUsed := False;  
                    clients[i].fNamed := False;  
                    clients[i].Connection := Nil;  
                    DeleteClient := clients[i];  
                    clients[i].strName := '';  
                    clients[i].strIP := '';  
  
                    section.Leave;  
                    Exit;  
                end;  
            end;  
        end;  
    end;  
end;
```

Процедура `SendClientList`, приведенная в листинге 11.16, отправляет клиентской программе заданного пользователя (только что зарегистрировавшегося) сообщения `addclient`: с именем каждого зарегистрированного ранее пользователя.

Листинг 11.16. Посылка списка всех присоединенных клиентов

```
procedure SendClientList(Connection: TIdTCPServerConnection);  
var  
    i: Integer;
```

```

begin
  for i:= 1 to MAX_CLIENT do
    if (clients[i].fNamed) then
      if (clients[i].Connection <> Connection) then
        try
          //Сообщим имя очередного найденного пользователя
          Connection.WriteLine('adduser:' + clients[i].strName);
        except
          //При возникновении ошибки отключим клиента
          //и продолжим рассылку
          ErrorCloseConnection(clients[i].Connection);
        end;
      end;
    end;
  end;
end;

```

Процедура `ErrorCloseConnection` (листинг 11.17) вызывается при ошибке отправки сообщений пользователям (например, при нарушении сетевого соединения). Она отключает пользователя, соединение с которым работает с ошибками, и сообщает об этом другим пользователям.

Листинг 11.17. Закрытие соединения с клиентом (при возникновении ошибки)

```

procedure ErrorCloseConnection(Connection: TIdTCPServerConnection);
var
  clError: client; //Информация о пользователе, соединение
                  //с которым прервалось (только имя и IP)
begin
  //Отключим соединение, работающее с ошибками
  clError := DeleteClient(Connection);
  //Сообщим об отключении остальным пользователям
  SendAll('deluser:' + clError.strName);
  SendAll('Нас покинул "' + clError.strName + '"');

  //Добавим событие в журнал
  if (REPORT) then AddEvent('Из-за ошибки отсоединен клиент "' +
    clError.strName + '" на компьютере "' + clError.strIP + '"');
end;

```

Процедура `RegisterClient`, приведенная в листинге 11.18, регистрирует пользователя под указанным в сообщении `name`: именем (ранее выполнялась функция `AddClient`, которая нашла для записи этого пользователя место в массиве `clients`). Если имя, под которым хочет зарегистрироваться пользователь, уже используется, то клиентской программе посылается соответствующее уведомление, после чего соединение разрывается.

Листинг 11.18. Регистрация нового клиента

```
procedure RegisterClient(Connection: TIdTCPServerConnection;
                        strName: string);

var
    i: Integer;
begin
    //Проверим, чтобы имя клиента еще не использовалось
    for i:=1 to MAX_CLIENT do
    begin
        if (clients[i].fNamed) then
            if (clients[i].strName = strName) then
            begin
                //Дублирование имени – придется разрывать соединение
                Connection.WriteLine('error:Пользователь с именем "' +
                                    strName + '" уже участвует в разговоре.');
```

DeleteClient(Connection);

Connection.Socket.Close;

Exit;

end;

end;

//Поиск записи о нужном клиенте и присвоение ему имени

for i:=1 to MAX_CLIENT do

begin

if (not clients[i].fNamed and clients[i].fUsed) then

if (clients[i].Connection = Connection) then

begin

//Вот он, наш клиент...

clients[i].fNamed := True;

clients[i].strName := strName;

//Сообщим другим о появлении нового участника

SendAll('adduser:' + strName);

SendAll('text:К нам присоединился "' + strName +

'" . Поприветствуем!');

//Отсылаем новому клиенту список остальных участников

//разговора

SendClientList(Connection);

```

        //Разрешим новому клиенту отсылать сообщения
        Connection.WriteLine('ok:');

        //Если нужно, то добавим событие в список
        if (REPORT) then AddEvent('Присоединен клиент "' +
            strName + '" на компьютере "' +
            Connection.Socket.Binding.PeerIP + '"');
    end;
end;
end;

```

В листинге 11.19 приведена служебная функция, возвращающая имя пользователя по ссылке на объект `TIdTCPServerConnection`, соответствующий этому клиенту.

Листинг 11.19. Определение имени клиента по его соединению с сервером

```

function GetClientName(Connection: TIdTCPServerConnection):string;
var
    i: Integer;
begin
    for i:=1 to MAX_CLIENT do
        if (clients[i].fNamed) then
            if (clients[i].Connection.Socket.Binding.Handle =
                Connection.Socket.Binding.Handle) then
                begin
                    GetClientName := clients[i].strName;
                    Exit;
                end;
    end;
end;

```

И, наконец, в листинге 11.20 приводится главная процедура серверного приложения, обрабатывающая сообщения, полученные от клиентов.

Листинг 11.20. Обработка сообщения от клиента

```

procedure ProcessMessage(Connection: TIdTCPServerConnection;
    strMessage: string);
var
    strName: string;    //Имя отправителя сообщения
    strAction: string;  //Строка с обозначением действия (префикс)
    len: Integer;       //Длина строки strAction

```

```

begin
    //Определим действие, которое хочет выполнить клиент
    len := Pos(':', strMessage);
    strAction := Copy(strMessage, 1, len-1);
    Delete(strMessage, 1, len);
    if (strAction = 'name') then
    begin
        //Клиент сообщает свое имя — попытаемся его зарегистрировать
        RegisterClient(Connection, strMessage);
    end
    else if (strAction = 'text') then
    begin
        //Клиент передает сообщение всем — подпишем сообщение и отошлем
        strMessage := GetClientName(Connection) + ': ' + strMessage;
        SendAll('text:' + strMessage);
        //Если надо, то сохраняем сообщение в списке событий
        if (REPORT) then AddEvent('Сообщение от ' + strMessage);
    end
    else
    begin
        //Клиент передает сообщение определенному собеседнику
        //(строка strAction содержит имя собеседника)
        strName := GetClientName(Connection);
        SendTo('text:' + strName + ': ' + strMessage, strAction);
        if (strName <> strAction) then
            //Передадим копию сообщения отправителю
            Connection.WriteLine('text:' + strName + ' для ' +
                                  strAction + ': ' + strMessage);

            //Если надо, то сохраняем сообщение в списке событий
            if (REPORT) then AddEvent('Сообщение для ' + strAction +
                                      ' от ' + strName + ': ' + strMessage);
        end;
    end;
end;

Информация о каждом пользователе (участнике разговора) хранится в отдельной
структуре client:

type
    client = record

```

```
fUsed: Boolean; {Ячейка занята}
fNamed: Boolean; {Клиент сообщил свое имя}
strName: string; {Имя пользователя}
strIP: string; {IP-адрес клиента}
Connection: TIdTCPServerConnection; {Соединение клиента
                                     с сервером}

end;
```

Непосредственно к пользователю имеют отношение три последних поля структуры. Самым полезным из них является ссылка на объект `TIdTCPServerConnection`, с помощью которой сервер может в любое время отправить данные определенному пользователю.

Информация обо всех пользователях хранится в массиве `clients`. Его размер ограничен (константа `MAXCLIENT`) и определяет максимальное количество пользователей — участников разговора. Так как используется массив с постоянным количеством элементов, то можно применять специальный флаг (поле `fUsed`) для индикации того, что ячейка массива занята (значение `True`) или свободна (значение `False`). Поле `fName` структуры `client` используется для фиксации факта сообщения клиентской программой имени пользователя (клиентские программы незарегистрированных пользователей сообщения не получают). Изначально значение поля `fNamed` равно `False` и устанавливается в `True`, только если имя пользователя сообщено серверу и не используется одним из участников разговора.

Одним из самых сложных моментов работы рассматриваемого сервера является обеспечение синхронизации доступа к массиву `clients`. Для этого используется критическая секция. Она также применяется для синхронизации добавления событий в список `lstEvents` сервера.

И, наконец, последний момент в реализации сервера. Чтобы сервер можно было запускать с отключенным протоколированием событий, а также чтобы окно сервера не мешало пользователю, можно хранить значения переменных `REPORT` и `SERVERVISIBLE` в INI-файле. Так, собственно, и сделано: значения этих переменных хранятся в секции `[Common]` файла `Server.ini`. Для считывания значений из INI-файла при запуске сервера код в модуле `Server` (файл `Server.dpr`) изменен следующим образом (листинг 11.21).

Листинг 11.21. Изменения в модуле `Server`

```
program Server;
uses
  Forms,
  Unit1 in 'Unit1.pas' {frmServer},
  IniFiles, Dialogs;
{$R *.res}
```

```
var
  {Переменные из INI-файла}
  config: TIniFile;
  strPath: string;
begin
  //Грузим информацию из INI-файла
  strPath :=
    Copy(Application.ExeName,1,Length(Application.ExeName)-3) +
    'ini';
  config := TIniFile.Create(strPath);
  SERVERVISIBLE := config.ReadBool('Common', 'ServerVisible',
                                   False);
  REPORT := config.ReadBool('Common','EventReport', False);
  config.Free ;
  try
    //Запуск сервера
    Application.Initialize;
    Application.CreateForm(TfrmServer, frmServer);
    Application.Run;
  except
    MessageDlg('Не удастся запустить сервер сообщений. ' +
              'Возможно, он был запущен ранее.', mtError, [mbOK], 0);
  end;
end.
```

В приведенном листинге код создания формы помещен в блок `try`. Сделано это только для того, чтобы сервер не «падал» с выдачей всем прекрасно знакомого окна о критической ошибке при попытке ошибочного запуска своей копии.

Соответственно, INI-файл для запуска сервера с видимым окном и включенным протоколированием имеет следующий вид:

```
[Common]
ServerVisible=1
EventReport=1
```

Реализация клиентского приложения

Проект клиентской программы имеет имя `Client`. Внешний вид формы клиентского приложения во время его работы представлен на рис. 11.7.

Приведенная на рис. 11.7 форма имеет имя `frmClient`. Свойства (только существенные для работы приложения) основных элементов управления, помещенных на форму, приведены в табл. 11.2.

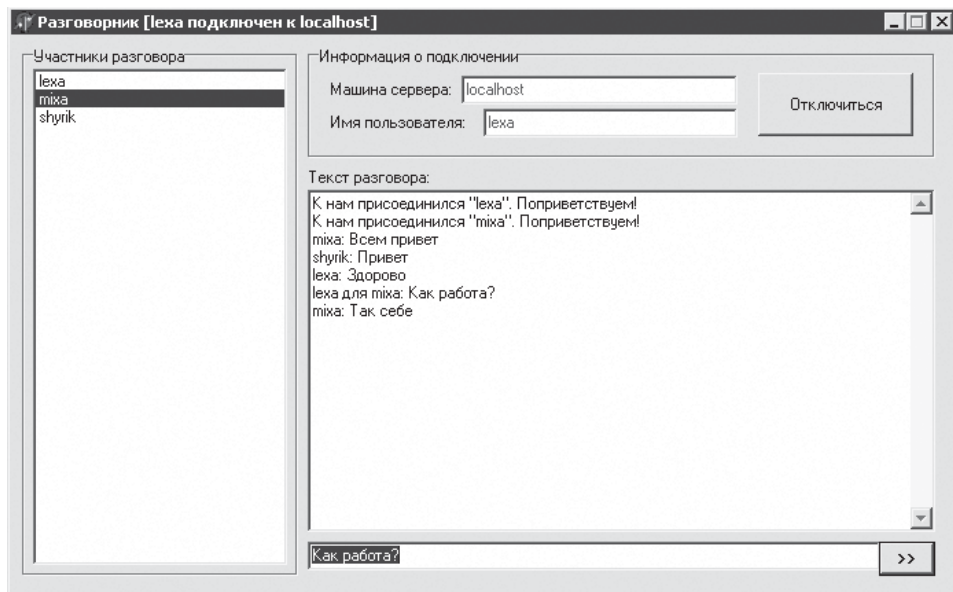


Рис. 11.7. Форма клиента при ведении разговора

Таблица 11.2. Свойства элементов управления формы frmClient

Название (свойство Name)	Описание	Измененные свойства и их новые значения
TCPClient	Элемент управления IdTCPClient	MaxLineAction = maSplit Port = 12345 ReadTimeout = 1
Timer1	Таймер (Timer) — используется для проверки прихода уведомления от сервера	Enabled = True Interval = False
lstUsers	Список участников разговора (ListBox) в левой части формы	Enabled = False Sorted = True
txtChat	Элемент Мемо с текстом разговора	Enabled = False ReadOnly = True ScrollBars = ssVertical
txtMessage	Текстовое поле (Edit) для ввода и редактирования сообщений внизу формы	Enabled = False
cmbSend	Кнопка (Button) отправки сообщения, введенного в текстовое поле txtMessage	Caption = >> Default = True Enabled = False Font.Style = [fsBold]

Продолжение ➤

Таблица 11.2 (продолжение)

Название (свойство Name)	Описание	Измененные свойства и их новые значения
txtServer	Текстовое поле (Edit) для указания имени или IP-адреса сервера	Отсутствует
txtUser	Текстовое поле (Edit) для указания имени пользователя	Отсутствует
cmbConnect	Кнопка (Button) подключения к указанному в txtServer серверу	Caption = Подключиться

Далее приведены функции и процедуры, не являющиеся обработчиками событий, но имеющие большое значение для работы клиентского приложения.

Приведенная в листинге 11.22 процедура обновляет форму при удачном подключении к серверу.

Листинг 11.22. Обновление формы при присоединении к серверу

```
procedure Connect();
begin
    with frmClient do
    begin
        cmbConnect.Caption := 'Отключиться';
        txtUser.Enabled := False;
        txtServer.Enabled := False;
        Caption := 'Разговорник [' + txtUser.Text + ' подключен к ' +
            txtServer.Text + ']';
        lstUsers.Enabled := True;
        cmbSend.Enabled := True;
        txtMessage.Enabled := True;
        txtChat.Enabled := True;
    end;
end;
```

Процедура Disconnect, приведенная в листинге 11.23, обновляет форму при отключении от сервера (в таком виде форма frmClient предстает первоначально).

Листинг 11.23. Обновление формы при отсоединении от сервера

```
procedure Disconnect();
begin
    with frmClient do
    begin
        cmbConnect.Caption := 'Подключиться';
        txtUser.Enabled := True;
```

```
txtServer.Enabled := True;
Caption := 'Разговорник';
lstUsers.Enabled := False;
lstUsers.Clear;
cmbSend.Enabled := False;
txtMessage.Enabled := False;
txtChat.Enabled := False;
end;
end;
```

Процедура `ProcessMessage` (листинг 11.24) обрабатывает сообщение, полученное от сервера, аналогично такой же процедуре в серверном приложении (естественно, сообщения и реакция на них отличны от серверных).

Листинг 11.24. Обработка строки, полученной от сервера

```
procedure ProcessMessage(strMessage: string);
var
  strAction: string; //Тип сообщения (префикс сообщения)
  len: Integer;      //Длина строки strAction
begin
  //Определим тип сообщения и выполним соответствующие действия
  len := Pos(':', strMessage);
  strAction := Copy(strMessage, 1, len-1);
  Delete(strMessage, 1, len);
  if (strAction = 'ok') then
  begin
    //Регистрация пользователя завершена – можно отправлять
    //сообщения
    Connect;
  end
  else if (strAction = 'error') then
  begin
    //Ошибка!!!
    frmClient.TCPClient.Disconnect;
    Disconnect;
    MessageDlg(strMessage, mtError, [mbOK], 0);
  end
  else if (strAction = 'adduser') then
  begin
    //К разговору присоединился новый пользователь
    frmClient.lstUsers.Items.Add(strMessage);
  end
end
```

```
else if (strAction = 'deluser') then
begin
    //Какой-то пользователь отсоединился
    frmClient.lstUsers.Items.Delete(
        frmClient.lstUsers.Items.IndexOf(strMessage));
end
else begin
    //Покажем принятое сообщение
    frmClient.txtChat.Lines.Add(strMessage);
end;
end;
```

Далее приводятся обработки событий, на которых, собственно, и основана работа клиентской программы. Обработчик нажатия кнопки `cmbConnect`, приведенный в листинге 11.25, пытается присоединиться к серверу. Если клиент присоединен к серверу, то эта же кнопка используется для его отсоединения.

Листинг 11.25. Присоединение/отсоединение от сервера

```
procedure TfrmClient.cmbConnectClick(Sender: TObject);
begin
    if (cmbConnect.Caption = 'Подключиться') then
    begin
        //Проверим, чтобы были введены имя сервера
        //и имя пользователя
        if (txtServer.Text = '') then
        begin
            MessageDlg('Введите имя сервера в текстовое поле.',
                mtInformation, [mbOK], 0);
            Exit;
        end
        else if (txtUser.Text = '') then
        begin
            MessageDlg('Введите имя пользователя в текстовое поле.',
                mtInformation, [mbOK], 0);
            Exit;
        end;
        //Пытаемся подключиться к серверу
        try
            TCPClient.Host := txtServer.Text;
            TCPClient.Connect;
```

```
except
    MessageDlg('Не удастся соединиться с сервером', mtError,
[mbOK], 0);
end;
end
else
    //Отключаемся от сервера
    TCPClient.Disconnect;
end;
```

Обработчик нажатия кнопки `cmbSend` (листинг 11.26) отправляет сообщение, которое могут прочесть все пользователи, присоединенные к серверу.

Листинг 11.26. Отправка сообщения всем собеседникам

```
procedure TfrmClient.cmbSendClick(Sender: TObject);
begin
    if (txtMessage.Text <> '') then
    begin
        //Отправка сообщения всем собеседникам
        TCPClient.WriteLine('text:' + txtMessage.Text);
        txtMessage.Text := '';
        txtMessage.SetFocus;
    end;
end;
```

При двойном щелчке кнопкой мыши на имени в списке пользователей отправляется сообщение, которое получает только выделенный в списке пользователь (листинг 11.27).

Листинг 11.27. Отправка сообщения заданному собеседнику

```
procedure TfrmClient.lstUsersDblClick(Sender: TObject);
begin
    if ((lstUsers.ItemIndex >= 0) and (txtMessage.Text <> ''))
then
    begin
        //Отправим сообщение только для выбранного собеседника
        //(сообщение вида "имя_собеседника:текст_сообщения")
        TCPClient.WriteLine(lstUsers.Items.Strings[lstUsers.ItemIndex] +
            ':' + txtMessage.Text);
        txtMessage.SetFocus;
    end;
end;
```

Сразу после соединения с сервером, то есть в обработчике `TfrmClient.TCPClientConnected`, приведенном в листинге 11.28, клиентская программа отправляет имя пользователя серверу. При отсоединении от сервера (тот же листинг 11.28) происходит соответствующее оформление внешнего вида формы `frmClient`.

Листинг 11.28. Обработка присоединения/отсоединения от сервера

```
procedure TfrmClient.TCPClientConnected(Sender: TObject);
begin
    //Отправляем на сервер имя пользователя
    TCPClient.WriteLine('name:' + txtUser.Text);
end;
procedure TfrmClient.TCPClientDisconnected(Sender: TObject);
begin
    //Оформим форму для отсоединенного от сервера состояния
    Disconnect;
end;
```

Ключевой обработчик (именно по таймеру проверяется факт прихода сообщения от сервера) приведен в листинге 11.29. Для элемента управления `TCPClient` значение тайм-аута установлено для того, чтобы при отсутствии принятых данных клиентская программа не переходила надолго в состояние ожидания, а генерировалось исключение, по которому и можно судить, что данных еще нет (см. блок `try` в этом обработчике).

Листинг 11.29. Проверка, есть ли данные от сервера

```
procedure TfrmClient.Timer1Timer(Sender: TObject);
var strMessage: string;
begin
    //Проверим, нет ли для нас сообщения
    if (TCPClient.Connected) then
        begin
            try
                strMessage := TCPClient.ReadLn;
                if (strMessage <> '') then
                    ProcessMessage(strMessage);
            except
                on EIdReadTimeout do ; //Ошибки тайм-аута игнорируем
                else
                    //При остальных ошибках отсоединяемся от сервера
                    TCPClient.Disconnect;
            end;
        end;
```

```
end;  
end;  
end.
```

**ПРИМЕЧАНИЕ**

Чтобы при запуске клиентского приложения из среды Delphi постоянно не появлялись сообщения об исключениях (возникают при истечении тайм-аута в `TfrmClient.Timer1Timer`), снимите флажок `Stop on Delphi Exceptions` на вкладке `Language Exceptions` диалогового окна `Debugger Options` (меню `Tools ▶ Debugger Options`).

На этом рассмотрение сетевого взаимодействия средствами Delphi в рамках этой книги завершается. Конечно, в главе перечислены далеко не все типы соединений и служб, поддерживаемых хотя бы компонентами, поставляемыми вместе с Delphi. Для рассмотрения работы со всеми имеющимися компонентами понадобилось бы написать целую книгу. Тем не менее хочется надеяться, что приведенные в главе примеры помогут вам в освоении механизмов программного взаимодействия между частями компьютерной сети.



Глава 12

Шифрование

- ☐ Основы криптографии
- ☐ Шифр простой подстановки
- ☐ Транспозиция
- ☐ Шифр Виженера и его варианты
- ☐ Шифр с автоключом
- ☐ Взлом

По той или иной причине часто бывает необходимо сообщить определенную информацию конкретному кругу людей так, чтобы она оставалась тайной для других. Возникает вполне очевидный вопрос: что для этого нужно сделать? Скорее всего, многие читатели в разные моменты времени и с различными целями пытались решить для себя задачу о секретной передаче. В результате выбиралось приемлемое решение, наверняка повторяющее изобретение одного из существующих способов скрытой передачи информации, история которого насчитывает не одну тысячу лет.

В отношении задачи о секретной передаче нетрудно прийти к выводу, что есть три способа ее реализации в компьютерных системах:

- создание абсолютно надежного канала связи, к которому есть доступ только у отправителя и адресата;
- использование общедоступного канала связи, но скрытие самого факта передачи информации;
- использование общедоступного канала связи с передачей по нему нужной информации, определенным образом преобразованной так, что восстановить ее может только адресат.

Что касается первого способа, то при современном уровне развития науки и техники сделать такой канал связи между удаленными абонентами для неоднократной передачи больших объемов информации практически нереально.

Второй способ является предметом изучения стеганографии. В область этой науки входит разработка средств и методов скрытия факта передачи сообщения.

Первые следы стеганографических методов теряются в глубокой древности. Например, известен такой способ скрытия письменного сообщения: голову раба брили, на коже головы писали сообщение и после отрастания волос раба отправляли к адресату.

Из детективных произведений хорошо известны различные способы тайнописи между строк обычного, незащищаемого текста: от молока до сложных химических реактивов с последующей обработкой.

Оттуда же известен метод «микроточки»: сообщение записывается с помощью современной техники на очень маленький носитель (микроточку), который пересылается с обычным письмом, например, под маркой или где-нибудь в другом, заранее обусловленном месте.

В настоящее время в связи с широким распространением компьютеров известно много тонких методов «запрятывания» защищаемых данных внутри больших объемов информации, хранящейся на компьютере.

Третий способ является предметом изучения криптографии. В ее область входит разработка методов преобразования (шифрования) информации с целью защиты от незаконных пользователей. Такие методы и способы преобразования информации называются шифрами.

12.1. Основы криптографии

Американский математик Клод Шеннон написал работу «Теория связи в секретных системах», в которой он обобщил накопленный до него опыт разработки шифров. В этой работе указано на то, что даже в самых сложных шифрах в качестве типичных компонентов можно выделить шифры замены, шифры перестановки или их сочетание.

Для начала рассмотрим эти шифры, а позже реализуем их. Начнем, пожалуй, с шифра замены как с самого простого и наиболее популярного. Примерами самых распространенных из известных шифров замены могут служить шифр Цезаря, «цифирная азбука» Петра Великого и «пляшущие человечки» А. Конан Дойла. Из самого названия видно, что шифр замены осуществляет преобразование заменой букв или других «частей» открытого текста на аналогичные «части» шифрованного текста. Легко дать математическое описание шифра замены. Пусть X и Y — два алфавита (открытого и шифрованного текстов соответственно), состоящие из одинакового количества символов. Пусть также $g: X \rightarrow Y$ — взаимнооднозначное отображение X в Y . Тогда шифр замены действует так: открытый текст $x_1x_2...x_n$ преобразуется в шифрованный текст $g(x_1)g(x_2)...g(x_n)$.

Шифр перестановки, как видно из названия, осуществляет преобразование перестановки букв в открытом тексте. Примером одного из известных шифров перестановкой может служить шифр «Считала». Обычно открытый текст разбивается на отрезки равной длины, и каждый отрезок шифруется независимо. Пусть, например, длина отрезков равна n и g — взаимнооднозначное отображение множества $\{1, 2, ..., n\}$ в себя. Тогда шифр перестановки действует так: отрезок открытого текста $x_1x_2...x_n$ преобразуется в отрезок шифрованного текста $xg(1)xg(2)...xg(n)$.

Важнейшим для развития криптографии был вывод К. Шеннона о существовании и единственности абсолютно стойкого шифра. Единственным таким шифром является какая-нибудь форма так называемой ленты однократного использования, в которой открытый текст «объединяется» с полностью случайным ключом такой же длины.

Этот вывод был доказан К. Шенноном с помощью разработанного им теоретико-информационного метода исследования шифров. Мы не будем здесь останавливаться на этом подробно, заинтересованному читателю рекомендуем изучить работу К. Шеннона.

Проясним для читателя один очень важный момент по поводу единственного абсолютно стойкого шифра. Чтобы шифр являлся таковым, должны выполняться три условия:

- полная случайность (равновероятность) ключа (это, в частности, означает, что ключ нельзя выработать с помощью какого-либо детерминированного устройства);
- равенство длины ключа и длины открытого текста;
- однократность использования ключа.

В случае нарушения хотя бы одного из этих условий шифр перестает быть абсолютно стойким и появляются принципиальные возможности для его вскрытия (хотя реализовать их может быть чрезвычайно сложно).

Но, оказывается, именно эти условия и делают абсолютно стойкий шифр очень дорогим и непрактичным. Прежде чем пользоваться таким шифром, мы должны обеспечить всех законных пользователей достаточным запасом случайных ключей и исключить возможность их повторного применения. А это сделать очень трудно и дорого.

В силу указанных причин абсолютно стойкие шифры применяются только в сетях связи с небольшим объемом передаваемой информации, обычно это сети для передачи особо важной государственной информации.

Теперь уже понятно, что чаще всего для защиты своей информации законные пользователи вынуждены применять не абсолютно стойкие шифры. Такие шифры могут быть вскрыты (по крайней мере, теоретически). Вопрос только в том, хватит ли у противника сил, средств и времени для разработки и реализации соответствующих алгоритмов. Обычно эту мысль выражают так: противник с неограниченными ресурсами может вскрыть любой не абсолютно стойкий шифр.

Как же должен действовать в этой ситуации законный пользователь, выбирая для себя шифр? Лучше всего, конечно, было бы доказать, что никакой противник не может вскрыть выбранный шифр, скажем, за десять лет и тем самым получить теоретическую оценку стойкости. К сожалению, математическая теория еще не дает нужных теорем — они относятся к нерешенной проблеме нижних оценок вычислительной сложности задач.

У пользователя остается единственный путь — получение практических оценок стойкости. Этот путь состоит из следующих этапов.

1. Понять и четко сформулировать, от какого противника мы собираемся защищать информацию. Необходимо уяснить, что именно противник знает или сможет узнать о системе шифра, а также какие силы и средства он сможет применить для его вскрытия.
2. Мысленно стать в положение противника и пытаться с его позиций атаковать шифр, то есть разрабатывать различные алгоритмы вскрытия шифра. При этом необходимо в максимальной мере обеспечить моделирование сил, средств и возможностей противника.
3. Наилучший из разработанных алгоритмов использовать для практической оценки стойкости шифра.

Полезно будет упомянуть о двух простейших методах вскрытия шифра: случайное угадывание ключа (он срабатывает с малой вероятностью, зато имеет небольшую сложность) и перебор всех подряд ключей вплоть до нахождения истинного (он срабатывает всегда, зато имеет очень большую сложность). Отметим также, что не всегда нужна атака на ключ: для некоторых шифров можно сразу, даже не зная ключа, восстанавливать открытый текст по шифрованному.

Теперь мы перейдем не только к столь необходимой теоретической части, но и к практической реализации различных криптосистем. Существует много их классификаций. Принципы классификации относятся не к качеству рассматриваемых криптосистем, а к присущим им свойствам.

12.2. Шифр простой подстановки

В шифре простой подстановки производится замена каждой буквы сообщения некоторым заранее определенным символом (обычно это также буква). В результате сообщение, имеющее вид $M = m_1m_2m_3m_4\dots$, где m_1, m_2, \dots — последовательность букв, переходит в сообщение вида $E = e_1e_2e_3e_4\dots = f(m_1)f(m_2)f(m_3)f(m_4)\dots$, причем функция $f(m)$ имеет обратную функцию g , для которой верно $g(f(m)) = m$, при всех возможных значениях m . В данном шифре ключом является просто перестановка алфавита (это верно в том случае, если буквы заменяются буквами). Например, подобная перестановка: ЛРЭИБПВЪДЁЗЦЙГХМЦАУОСЖТЯФКЕШНЫЬЧЮ. Она используется следующим образом:

- буква А открытого текста заменяется буквой Л;
- Б заменяется Р;
- В заменяется Э и т. д.

Как можно понять из определения, данный шифр является довольно простым. Перейдем к примеру, показывающему одну из возможных его реализаций. Для этого нам понадобится создать новое приложение, а на форму поместить следующие компоненты: по два компонента классов `TMemo` и `TLabel` с соответствующими именами `mmDecryptMessage`, `mmEncryptMessage`, `lbDecryptMessage`, `lbEncryptMessage`, три компонента класса `TButton` — `btnEncryptMessage`, `btnDecryptMessage`, `btnGenRearrangement`, а также один компонент класса `TValueListEditor` — `vleSubst`. По умолчанию все перечисленные компоненты находятся на вкладке **Standard**, кроме компонента класса `TValueListEditor`, который расположен на вкладке **Additional**. Когда вы закончите создание интерфейса программы, то у вас получится нечто подобное тому, что изображено на рис. 12.1.

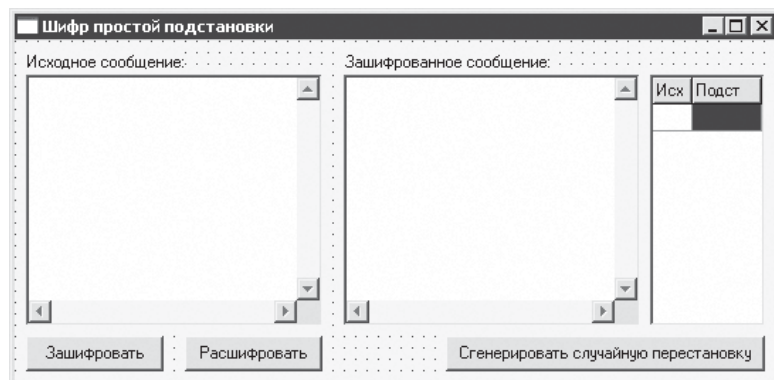


Рис. 12.1. Интерфейс программы «Шифр простой подстановки»

Текстовый редактор `mmDecryptMessage` будет служить для ввода и отображения открытого текста нашего сообщения, `mmEncryptMessage` — для текста, преобразованного при помощи шифра. Редактор значений `vleSubst` мы будем использовать для задания перестановки алфавита, при помощи которой будет шифроваться и дешифроваться текст сообщения. Кнопка `btnEncryptMessage` будет отвечать за шифрование сообщения из текстового редактора `mmDecryptMessage` и помещение результата в `mmEncryptMessage`. Кнопка `btnDecryptMessage` предназначена для противоположных действий. Последняя кнопка `btnGenRearrangement` будет служить для генерации случайной перестановки алфавита, чтобы не утруждать себя ее вводом вручную. Необходимо добавить обработчики событий `OnClick` для каждой из кнопок и обработчик события `OnCreate` для формы (он нужен для инициализации редактора значений `vleSubst`).

Теперь стоит оговориться, что программа будет шифровать и дешифровать только русский текст, оставляя неизменным все остальное. Далее рассмотрим исходный код нашей программы.

Первым делом нужно ввести необходимые типы для лучшего понимания написанного кода, а также следует соответствующим образом объявить класс формы. Как это сделать, показано в листинге 12.1.

Листинг 12.1. Объявление типов и класса нашей формы

type

```
TRusDstAlphabet = array [Char] of Char;
```

```
TfmSubstitution = class(TForm)
```

```
    mmDecryptMessage: TMemo;
```

```
    mmEncryptMessage: TMemo;
```

```
    lbDecryptMessage: TLabel;
```

```
    lbEncryptMessage: TLabel;
```

```
    btnEncryptMessage: TButton;
```

```
    btnDecryptMessage: TButton;
```

```
    btnGenRearrangement: TButton;
```

```
    vleSubst: TValueListEditor;
```

```
    procedure FormCreate(Sender: TObject);
```

```
    procedure btnGenRearrangementClick(Sender: TObject);
```

```
    procedure btnEncryptMessageClick(Sender: TObject);
```

```
    procedure btnDecryptMessageClick(Sender: TObject);
```

```
private
```

```
    { Private declarations }
```

```
    RusDstAlphabet: TRusDstAlphabet;
```

```
    procedure GenRearrangement;
```

```
    function ValidateRearrangement: Boolean;
```

```
function UpCaseRus(Ch: Char): Char;  
function LowCaseRus(Ch: Char): Char;  
procedure RecalcAlphabet(nKey: Integer);  
function EncryptDecryptString(strMsg: String): String;  
public  
  { Public declarations }  
end;
```

Каждую задачу следует рассматривать детально и выделять необходимые подзадачи, решение которых позволит облегчить и упростить общее решение. Помимо ряда стандартных обработчиков событий, мы добавили несколько собственных методов для лучшей структуризации кода и повышения его читабельности, что является немаловажным фактором при разработке приложений. Рассмотрим каждый метод и поясним их работу.

В нашем приложении для удобства и простоты работы будет реализована возможность задания случайной автоматической перестановки. Первым рассматриваемым методом является функция, реализующая алгоритм генерации случайной перестановки заданной длины из букв русского алфавита. Принцип ее работы заключается в следующем. Сначала считается, что в перестановке нет ни единого символа, о чем свидетельствует установка всех элементов массива WasGen в значение False. Далее в цикле случайным образом генерируются буквы русского алфавита. На очередном шаге цикла буква генерируется до тех пор, пока она будет присутствовать среди уже сгенерированных. Как только такая буква получена, то соответствующий элемент массива WasGen устанавливается в значение True, которое свидетельствует о том, что буква больше не может быть сгенерирована. Мы также не забываем добавить ее в перестановку. Код, соответствующий данному описанию, приведен в листинге 12.2.

Листинг 12.2. Реализация метода генерации случайной перестановки

```
procedure TfmSubstitution.GenRearrangment;  
var  
  Ch, c: char;  
  //нужен для определения, встречался ли символ ранее  
  WasGen: array [Char] of Boolean;  
begin  
  //заполняем массив значением False  
  FillChar(WasGen, SizeOf(WasGen), False);  
  for Ch := 'А' to 'Я' do  
    begin  
      //генерируем случайный символ до тех пор, пока  
      //не будет получен еще не сгенерированный  
      repeat  
        c := Chr(Ord('А') + random(32));
```

```

until not WasGen[c];
//помечаем, что символ сгенерирован
WasGen[c] := True;
vleSubst.Values[Ch] := c;
end;
end;

```

В нашем приложении пользователь может сам задавать необходимую перестановку букв алфавита, поэтому стоит учесть тот факт, что он может ошибиться при ее вводе. Для решения данной проблемы реализуем функцию, которая будет отвечать на вопрос о том, является ли введенная перестановка корректной. Определимся с тем, каким критериям должна отвечать перестановка, чтобы считаться допустимой. Во-первых, в каждой ячейке ввода должна присутствовать лишь одна буква — ни больше, ни меньше. Во-вторых, каждая введенная буква обязана принадлежать множеству букв русского алфавита. И в-третьих, ни одна введенная буква не должна ни разу повторяться. Проверка первого критерия довольно проста. Для этого достаточно лишь проверить длину строки, введенной в каждой ячейке. Второй критерий также проверяется довольно простой конструкцией принадлежности заданному множеству. Третий критерий проверяется подобно тому, как в предыдущем реализованном методе проверялось, сгенерирована данная буква или нет. Следующий исходный код, представленный в листинге 12.3, показывает, как эта проверка осуществляется.

Листинг 12.3. Реализация метода проверки допустимости перестановки

```

function TfmSubstitution.ValidateRearrangement: Boolean;
var
  i: Integer;
  s: String;
  Used: array [Char] of Boolean;
begin
  Result := False;
  FillChar(Used, SizeOf(Used), False);
  for i := 1 to vleSubst.RowCount - 1 do
    Begin
      //символ единственный в строке?
      s := vleSubst.Cells[1, i];
      if (Length(s) <> 1) then
        Exit;
      //символ — буква русского языка?
      s[1] := UpCaseRus(s[1]);
      if not (s[1] in ['А'..'Я']) then
        Exit;
    End;
  End;
end;

```

```

    //уже встречался ранее?
    if Used[s[1]] then Exit;
    Used[s[1]] := True;
End;
Result := True;
end;

```

Далее мы реализуем две вспомогательные функции, которые позволят преобразовать буквы верхнего регистра к нижнему и наоборот. Их реализация немного специфична и основывается на используемой кодировке. Отдельная проверка буквы «Ё» производится на основании иного расположения в таблице кодировки, чем у остальных букв. Буквы русского алфавита верхнего регистра расположены начиная с «А» по порядку следования в алфавите, а сразу после них аналогично расположены буквы нижнего регистра. Этим объясняется увеличение кода буквы на фиксированное число. Реализация данных вспомогательных функций приведена в листинге 12.4.

Листинг 12.4. Вспомогательные функции преобразования регистра букв

```

function TfmSubstitution.UpCaseRus(Ch: Char): Char;
begin
    if Ch = 'ё' then Ch := 'Е';
    if Ch in ['а'..'я'] then Dec(Ch, 32);
    Result := Ch;
end;

function TfmSubstitution.LowCaseRus(Ch: Char): Char;
begin
    if Ch = 'Ё' then Ch := 'е';
    if Ch in ['А'..'Я'] then Inc(Ch, 32);
    Result := Ch;
end;

```

Теперь рассмотрим работу обработчика события формы OnCreate и обработчика события кнопки OnClick. Первый сначала инициализирует редактор значений полями, для которых будут задаваться данные. После того как все поля созданы, вызывается функция генерации случайной перестановки, которая, в свою очередь, заполняет все поля редактора значений необходимыми данными. Второй же обработчик только вызывает функцию генерации случайной перестановки. В листинге 12.5 приведен исходный код данных методов.

Листинг 12.5. Использование генерации случайной перестановки

```

procedure TfmSubstitution.FormCreate(Sender: TObject);
var
    Ch: char;

```

```

begin
    Randomize;
    //инициализация редактора значений
    for Ch := 'A' to 'Я' do
        vleSubst.InsertRow(Ch, '', True);
    //генерация случайной перестановки
    GenRearrangment;
end;

procedure TfmSubstitution.btnGenRearrangementClick(Sender:
                                                    TObject);
begin
    GenRearrangment;
end;

```

Следующим объектом нашего рассмотрения является функция предварительной подготовки алфавита преобразования для шифрования либо дешифрования сообщения. У метода `RecalcAlphabet` есть параметр `nKey`, который в зависимости от своего значения показывает, что является ключом. Возможными значениями `nKey` являются 0 и 1. Значение 0 указывает на то, что будет производиться шифрование сообщения и требуется поставить в соответствие буквам открытого текста буквы перестановки. Значение 1, напротив, указывает на то, что будет производиться дешифрование сообщения и требуется поставить в соответствие буквам перестановки буквы открытого текста. Для этого массив сопоставления символов изначально заполняется таким образом, чтобы каждый символ соответствовал самому себе. Это происходит в следующих строках метода:

```

for Ch := Low(RusDstAlphabet) to High(RusDstAlphabet) do
    RusDstAlphabet[Ch] := Ch;

```

После чего требуется подкорректировать данный массив таким образом, чтобы выполнялось требуемое соответствие. Для этого мы проходим по всем элементам редактора значений `vleSubst` и поправляем массив, указывая в качестве индекса элемента то, чему ставится соответствие, а в качестве значения элемента массива — то, что является соответствием.

```

for i := 1 to vleSubst.RowCount - 1 do
    RusDstAlphabet[vleSubst.Cells[nKey, i][1]] :=
        vleSubst.Cells[1 - nKey, i][1];

```

Редактор значений `vleSubst` предназначен для сопоставления букв верхнего регистра. Нам же требуется избавиться от различия между буквами верхнего и нижнего регистров. Для этого мы дополнительно производим следующие действия:

```

for i := 1 to vleSubst.RowCount - 1 do
    RusDstAlphabet[LowerCaseRus(vleSubst.Cells[nKey, i][1])] :=
        LowerCaseRus(vleSubst.Cells[1 - nKey, i][1]);

```

Мы рассмотрели работу данного метода по частям. Его полный код приведен в листинге 12.6. Как видите, все относительно просто. Здесь мы используем вспомогательную функцию `LowerCaseRus`.

Листинг 12.6. Функция предварительной подготовки алфавита преобразования

```
procedure TfmSubstitution.RecalcAlphabet(nKey: Integer);
var
  Ch: Char;
  i: Integer;
begin
  //предварительно все символы в алфавите шифрования
  //соответствуют символам из незашифрованного алфавита
  for Ch := Low(RusDstAlphabet) to High(RusDstAlphabet) do
    RusDstAlphabet[Ch] := Ch;
  //формируем алфавит отдельно для каждого из регистров букв
  //здесь для верхнего
  for i := 1 to vleSubst.RowCount - 1 do
    RusDstAlphabet[vleSubst.Cells[nKey, i][1]] :=
      vleSubst.Cells[1 - nKey, i][1];
  //здесь для нижнего
  for i := 1 to vleSubst.RowCount - 1 do
    RusDstAlphabet[LowerCaseRus(vleSubst.Cells[nKey, i][1])] :=
      LowerCaseRus(vleSubst.Cells[1 - nKey, i][1]);
end;
```

Еще одной вспомогательной функцией является функция преобразования строки символов с помощью алфавита преобразования в соответствии с указанной операцией. Работа ее довольно проста. В цикле осуществляется прямой проход по строке, и каждый символ, принадлежащий ей, заменяется соответствующим символом алфавита преобразования. В итоге мы получаем зашифрованную либо дешифрованную строку. Посмотреть исходный код данного метода можно в листинге 12.7.

Листинг 12.7. Преобразование строки при помощи массива сопоставления

```
function TfmSubstitution.EncryptDecryptString(strMsg: String):
String;
var
  i: Integer;
begin
  //преобразуем строку посимвольно
  for i := 1 to Length(strMsg) do
    strMsg[i] := RusDstAlphabet[strMsg[i]];
```

```
Result := strMsg;  
end;
```

Теперь, используя все описанные функции, мы без труда можем зашифровать либо дешифровать сообщение. Например, чтобы зашифровать его, мы подготавливаем массив соответствия букв вызовом функции `RecalcAlphabet` с параметром 0. После чего для каждой строки открытого текста вызываем функцию `EncryptDecryptString` и в качестве результата получаем зашифрованную строку. Обработчики событий `OnClick` соответствующих кнопок шифруют либо дешифруют весь текст. Основная идея каждого из методов заключается в том, чтобы проверить корректность заданной перестановки, после чего производится предварительная подготовка алфавита сопоставления, и далее сообщение преобразуется (листинг 12.8).

Листинг 12.8. Шифрование/дешифрование сообщения

```
procedure TfmSubstitution.btnEncryptMessageClick(Sender:  
                                                    TObject);  
var  
    i: Integer;  
begin  
    //проверяем корректность ввода перестановки  
    if ValidateRearrangement then  
        begin  
            //создаем алфавит преобразования открытого текста  
            RecalcAlphabet(0);  
            //предотвращаем перерисовку компонента до тех пор,  
            //пока не зашифруем все строки сообщения  
            mmEncryptMessage.Lines.BeginUpdate;  
            //очищаем текстовый редактор  
            mmEncryptMessage.Clear;  
            //шифруем открытый текст построчно  
            for i := 0 to mmDecryptMessage.Lines.Count - 1 do  
                mmEncryptMessage.Lines.Add(EncryptDecryptString  
                    (mmDecryptMessage.Lines[i]));  
            //разрешаем перерисовку компонента  
            mmEncryptMessage.Lines.EndUpdate;  
        end  
    else  
        MessageDlg('Ошибка: символы подстановки заданы неверно',  
                    mtError, [mbOk], 0);  
end;
```

```

procedure TfmSubstitution.btnDecryptMessageClick(Sender:
                                                    TObject);
var
  i: Integer;
begin
  //проверяем корректность ввода перестановки
  if ValidateRearrangement then
    begin
      //создаем алфавит преобразования шифрованного текста
      RecalcAlphabet(1);
      mmDecryptMessage.Lines.BeginUpdate;
      mmDecryptMessage.Clear;
      //дешифруем шифрованный текст построчно
      for i := 0 to mmEncryptMessage.Lines.Count - 1 do
        mmDecryptMessage.Lines.Add(EncryptDecryptString
                                   (mmEncryptMessage.Lines[i]));
      mmDecryptMessage.Lines.EndUpdate;
    end
  else
    MessageDlg('Ошибка: символы подстановки заданы неверно',
               mtError, [mbOk], 0);
end;

```

В итоге мы получили вполне рабочий вариант приложения, способного без особого труда шифровать и дешифровать сообщения. На рис. 12.2 представлен результат работы данного приложения.

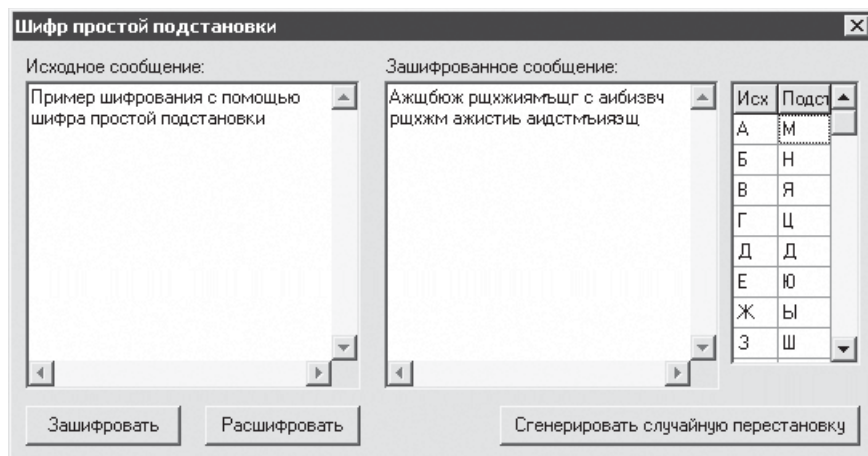


Рис. 12.2. Результат работы приложения «Шифр простой подстановки»

12.3. Транспозиция

Следующий шифр, который мы будем рассматривать, называется транспозицией с фиксированным периодом d . В этом случае сообщение делится на группы символов длины d и к каждой группе применяется одна и та же перестановка. Эта перестановка является ключом и может быть задана некоторой перестановкой первых d целых чисел.

Таким образом, для $d = 5$ в качестве перестановки можно взять 23154. Это будет означать, что $m_1m_2m_3m_4m_5m_6m_7m_8m_9m_{10}...$ переходит в $m_2m_3m_1m_5m_4m_7m_8m_6m_{10}m_9...$ Последовательное применение двух или более транспозиций будет называться составной транспозицией. Если периоды этих транспозиций $d_1, ..., d_s$, то, очевидно, в результате получится транспозиция периода d , где d — наименьшее общее кратное $d_1, ..., d_s$.

Теперь, зная определение данного шифра, можно перейти к примеру одной из возможных его реализаций. Для этого, как и в предыдущем случае, создадим новое приложение, а на форму поместим те же самые компоненты, за исключением редактора значений и кнопки для генерации перестановки. Вместо них используем следующие компоненты: текстовое поле класса TEdit и еще один компонент класса TLabel с соответствующими именами edRearrangement и lbRearrangement. Когда вы закончите, то в результате должно получиться нечто подобное изображенному на рис. 12.3.

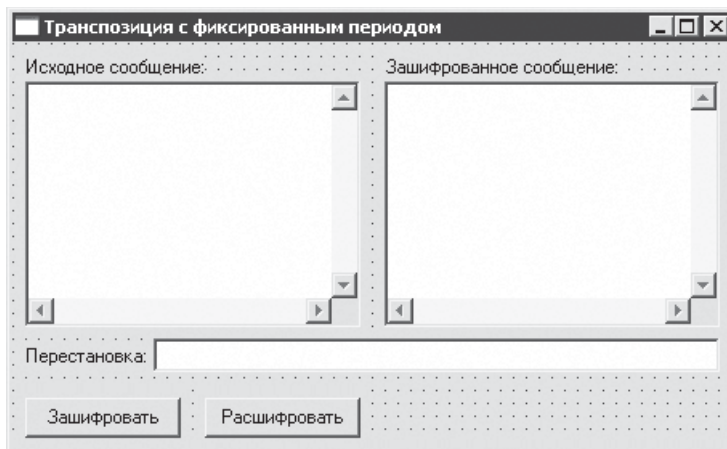


Рис. 12.3. Интерфейс программы «Транспозиция с фиксированным периодом»

Текстовое поле edRearrangement предназначено для задания перестановки, которая будет использоваться при шифровании. Перестановка будет задаваться числами, разделенными пробелом, а их количество задаст период транспозиции. По остальному интерфейсу наше приложение аналогично предыдущему.

Стоит отметить одну неприятную особенность данного шифра. Поскольку период фиксирован, то на текст накладывается определенное ограничение. Оно заключается

в том, что длина текста должна быть кратна периоду. Существует несколько вариантов решения данной проблемы. Можно дополнять открытый текст какими-либо символами. И тогда зашифровать сообщение не составит труда. Если эти символы заранее определены, то это облегчит задачу противника по вскрытию шифра. Другой вариант — переписать сообщение, используя, например, синонимы, либо удалив часть сообщения, которую легко восстановить из контекста, таким образом, чтобы длина текста стала кратной периоду.

Теперь перейдем к рассмотрению исходного кода нашего приложения. Как и в прошлый раз, начнем с объявления класса необходимых нам типов, а также класса формы. Соответствующий программный код показан в листинге 12.9. Здесь мы ввели целочисленную константу, ограничивающую длину задаваемого периода. В данном случае она равна 100. Нам понадобится помнить саму перестановку, при помощи которой будет осуществляться транспозиция сообщения, поэтому вводится соответствующий тип.

Листинг 12.9. Объявление типов и класса нашей формы

```
const
    MaxTerm = 100;
type
    TRearrangement = array [0..MaxTerm] of Integer;

    TfmTransposition = class(TForm)
        mmDecryptMessage: TMemo;
        mmEncryptMessage: TMemo;
        lbDecryptMessage: TLabel;
        lbEncryptMessage: TLabel;
        lbRearrangement: TLabel;
        edRearrangement: TEdit;
        btnEncryptMessage: TButton;
        btnDecryptMessage: TButton;
        procedure btnEncryptMessageClick(Sender: TObject);
        procedure btnDecryptMessageClick(Sender: TObject);
    private
        { Private declarations }
        Rear: TRearrangement;
        function RecalcRearrangement(nKey: Integer): Boolean;
        function GetLine(Lines: TStrings;
            nRow, nInd: Integer): String;
        procedure EncryptDecrypt(SrcLines, DstLines: TStrings;
            nKey: Integer);
    public
        { Public declarations }
    end;
```

Теперь перейдем к рассмотрению исходного кода решаемых в данном случае подзадач. Первой функцией, с которой мы начнем, будет функция разбора введенной строки, выделяющая перестановку из нее и проверяющая, является ли она допустимой.

Функция `RecalcRearrangement` подготавливает перестановку требуемым образом для шифрования либо дешифрования в зависимости от параметра `nKey`, который принимает два значения: 0 и 1. Значение 0 указывает на то, что будет производиться шифрование сообщения и дополнительных действий по подготовке перестановки не требуется, за исключением проверки ее корректности. Значение 1, напротив, указывает на то, что будет производиться дешифрование сообщения и требуется еще дополнительно преобразовать перестановку так, чтобы она была симметрична исходной, в этом случае процесс дешифрования ничем не будет отличаться от процесса шифрования.

Чтобы введенная перестановка считалась корректной, необходимо и достаточно выполнить три следующих требования:

- введены только числа через пробел;
- все числа не повторяются;
- числа находятся в диапазоне от 1 до их общего количества.

Проверка первого условия осуществляется следующим образом. Изначально считается, что в строке идут пробелы. Как только пробелы заканчиваются, предполагается, что началось число, и до тех пор, пока мы опять не встретим пробел, выделяем это число. Как только встретили пробел, пытаемся преобразовать выделенную часть из строкового представления в численное. После этого добавляем полученное число к итоговой перестановке. Когда фрагмент кода, в котором находится первый цикл с условием после него, отработает, в массиве `Rear` будет храниться введенная перестановка (в `Rear[0]` хранится количество чисел в полученной перестановке). Сразу за первой проверкой осуществляется совместно вторая и третья, то есть проверяется допустимость самих введенных чисел, а также их уникальность. После всех проверок при необходимости осуществляется преобразование исходной перестановки к симметричной.

Для получения симметричной перестановки стоит выполнить нехитрое действие по обмену местами индексов чисел и сами x чисел, то есть если имеется перестановка 3 1 2, то она преобразуется в 2 3 1, так как 1 стоит на втором месте, 2 — на 3, 3 — на 1.

Исходный код данной функции приведен в листинге 12.10.

Листинг 12.10. Функция разбора строки и проверки допустимости перестановки

```
function TfmTransposition.RecalcRearrangement(nKey: Integer): Boolean;
var
  i: Integer;
  s: String;
  Space: Boolean;
  Used: array [1..MaxTerm] of Boolean;
```

```
ExRear: TRearrangement;
begin
  Result := False;
  Rear[0] := 0;
  Space := True;
  //выделяем каждое слово, разделенное пробелом,
  //и преобразуем его к числу
  for i := 1 to Length(edRearrangement.Text) do
    if (edRearrangement.Text[i] = ' ') and (not Space) then
      begin
        Inc(Rear[0]);
        Rear[Rear[0]] := StrToInt(s);
        Space := True;
      end
    else
      if (edRearrangement.Text[i] <> ' ') then
        begin
          if Space then
            begin
              Space := False;
              s := '';
            end;
          s := s + edRearrangement.Text[i];
        end;
      if not Space then
        begin
          Inc(Rear[0]);
          Rear[Rear[0]] := StrToInt(s);
        end;
      //проверяем допустимость полученных чисел
      FillChar(Used, SizeOf(Used), False);
      for i := 1 to Rear[0] do
        if (0 < Rear[i]) and (Rear[i] <= Rear[0])
            and not Used[Rear[i]] then
          Used[Rear[i]] := True
        else
          Exit;
```

```
//преобразуем перестановку к шифровке, обратной
//для симметричности процесса дешифровки
if nKey = 1 then
begin
    ExRear[0] := Rear[0];
    for i := 1 to Rear[0] do
        ExRear[Rear[i]] := i;
    Rear := ExRear;
end;
Result := Rear[0] > 1;
end;
```

Еще для упрощения алгоритма шифрования необходимо уметь получать часть текста заданной длины, начиная с указанной позиции, в виде одной строки, пропуская все переводы строк. Это действие выполняет следующая описываемая функция. Алгоритм ее работы довольно прост. Изначально в результирующей строке нет ни единого символа. Далее осуществляется двойной вложенный цикл. Цикл верхнего уровня осуществляет изменение значения переменной, начиная с указанной строки до самой последней. Вложенный цикл, в свою очередь, изменяет значение переменной, первый раз начиная с указанной позиции в строке, а в остальных случаях всегда с 1, до длины текущей обрабатываемой строки. Каждый очередной символ добавляется к результирующей строке до тех пор, пока не будет достигнута заданная длина строки, равная периоду транспозиции. Соответствующий код приведен в листинге 12.11.

Листинг 12.11. Функция получения части текста заданной длины, начиная с указанной позиции, в виде одной строки

```
function TfmTransposition.GetLine(Lines: TStrings;
                                   nRow, nInd: Integer): String;
var
    i, j, k: Integer;
    s: String;
begin
    Result := '';
    s := '';
    k := nInd;
    for i := nRow to Lines.Count - 1 do
        begin
            for j := k to Length(Lines[i]) do
                begin
                    s := s + Lines[i][j];
                end;
            k := k + 1;
        end;
    end;
```

```

        if Length(s) = Rear[0] then
            begin
                Result := s;
                Exit;
            end;
        end;
        k := 1;
    end;
end;

```

Подготовительный этап мы рассмотрели, теперь остается рассмотреть основной код программы. Обработчики кнопок `OnClick` вызывают один и тот же метод и указывают необходимые параметры, чтобы зашифровать либо дешифровать текст сообщения. Процедура `EncryptDecrypt` в качестве параметров принимает источник текста сообщения, с которым нужно проделать необходимые действия, приемник преобразованного текста сообщения и тип преобразования. Последний параметр принимает одно из двух значений: 0 или 1. Значение 0 указывает на то, что будет производиться шифрование сообщения. Значение 1 указывает на то, что будет производиться дешифрование сообщения. Процедура `EncryptDecrypt` выполняет следующие действия. Сначала она пытается подготовить необходимую перестановку и, только если все прошло успешно, переходит к попытке преобразования текста сообщения, но предварительно делает еще одну проверку. Эта проверка заключается в следующем: нужно удостовериться в соответствии общей длины текста накладываемому на нее ограничению, то есть длина обязана быть кратна периоду транспозиции. Если все хорошо, то далее следует код преобразования текста сообщения с использованием подготовленной транспозиции. Для начала приведем исходный код, который находится в листинге 12.12.

Листинг 12.12. Шифрование/дешифрование текста сообщения

```

procedure TfmTransposition.btnEncryptMessageClick (Sender:
                                                    TObject);
begin
    EncryptDecrypt (mmDecryptMessage.Lines,
                    mmEncryptMessage.Lines, 0);
end;

procedure TfmTransposition.btnDecpyptMessageClick (Sender:
                                                    TObject);
begin
    EncryptDecrypt (mmEncryptMessage.Lines,
                    mmDecryptMessage.Lines, 1);
end;

```

```
procedure TfmTransposition.EncryptDecrypt(SrcLines,
                                           DstLines: TStrings;
                                           nKey: Integer);

var
  i, j, Cnt: Integer;
  s, EncryptMsg: String;
begin
  if RecalcRearrangement(nKey) then
    begin
      //вычисляем общую длину текста
      Cnt := 0;
      for i := 0 to SrcLines.Count - 1 do
        Inc(Cnt, Length(SrcLines[i]));
      //проверяем кратность общей длины длине перестановки
      if Cnt mod Rear[0] <> 0 then
        begin
          MessageDlg('Ошибка: текст сообщения не кратен длине
                     перестановки', mtError, [mbOk], 0);
          Exit;
        end;
      //преобразуем сообщение
      Cnt := Rear[0];
      DstLines.BeginUpdate;
      DstLines.Clear;
      for i := 0 to SrcLines.Count - 1 do
        begin
          EncryptMsg := '';
          for j := 1 to Length(SrcLines[i]) do
            begin
              if Cnt = Rear[0] then
                begin
                  s := GetLine(SrcLines, i, j);
                  Cnt := 0;
                end;
              Inc(Cnt);
              EncryptMsg := EncryptMsg + s[Rear[Cnt]];
            end;
          DstLines.Add(EncryptMsg);
        end;
      end;
```

```
        DstLines.EndUpdate;  
    end  
else  
    MsgBox('Ошибка: перестановка задана неверно', mtError,  
        [mbOk], 0);  
end;
```

С подготовительным этапом мы разобрались, а теперь рассмотрим непосредственно сам процесс преобразования текста сообщения. Здесь переменная `Cnt` отвечает за то, какую часть очередной группы букв уже обработали. Если она равна количеству чисел в перестановке, то происходит переход к очередной группе букв сообщения. Алгоритм преобразования усложняется тем, что строки текста не обязательно кратны количеству чисел в перестановке. Поэтому для удобства мы написали функцию `GetLine`, получающую часть сообщения с указанной позиции в виде одной строки определенной длины, которая при необходимости склеена из нескольких подряд идущих строк. Теперь нам ничего не мешает заменить очередную букву сообщения соответствующей буквой из полученной строки. Результат работы приложения приведен на рис. 12.4.

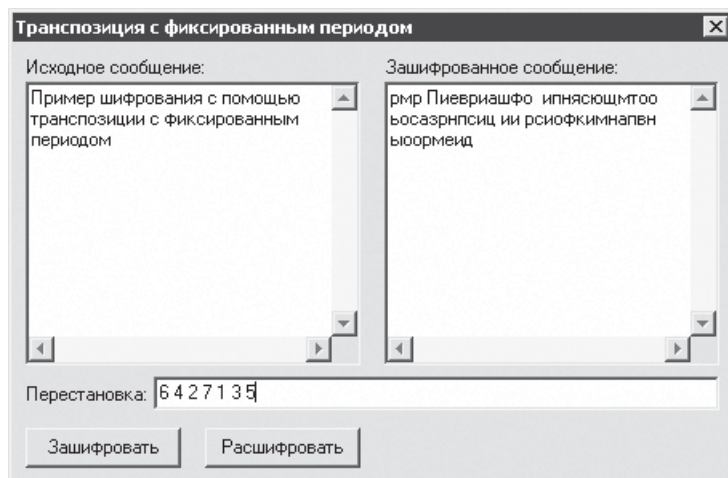


Рис. 12.4. Результат работы приложения «Транспозиция с фиксированным периодом»

12.4. Шифр Виженера и его варианты

Ключ в шифре Виженера задается набором из n букв. Такие наборы подписываются с повторением под текстом сообщения, и полученные две последовательности складываются по модулю m , где m — количество букв в рассматриваемом алфавите (например, для русского алфавита каждая буква нумеруется от 0 (А) до 32 (Я) и $m = 33$). В результате получаем правило преобразования открытого текста $li = xi + yi \pmod{m}$, где xi — буква в открытом тексте с номером i , yi — буква ключа, полученная сокращением числа i по модулю n . В табл. 12.1 приведен пример использования ключа ПБЕ.

Таблица 12.1. Шифр Виженера с ключом ПБЕ

Сообщение	Ключ	Криптограмма
Ш	П	Ж
И	Б	Й
Ф	Е	Щ
Р	П	Я
О	Б	П
В	Е	З
К	П	Щ
А	Б	Б

Шифр Виженера с периодом 1 называется шифром Цезаря. По сути, он представляет собой простую подстановку, в которой каждая буква некоторого сообщения M сдвигается циклически вперед на фиксированное количество мест по алфавиту. Именно это количество является ключом. Оно может принимать любое значение в диапазоне от 0 до $m - 1$. Повторное применение двух или более шифров Виженера будет называться составным шифром Виженера. Он имеет уравнение $li = xi + yi + \dots + zi \pmod{m}$, где $xi + yi + \dots + zi$ имеют различные периоды. Период их суммы, как и в составной транспозиции, будет наименьшим общим кратным отдельных периодов.

Если используется шифр Виженера с неограниченным неповторяющимся ключом, то мы имеем шифр Вернама, в котором $li = xi + yi \pmod{m}$ и yi выбираются случайно и независимо среди чисел $0, 1, \dots, m - 1$. Если ключом служит текст, имеющий смысл, то имеем шифр «бегущего ключа».

Теперь перейдем к примеру. Рассмотрим одну из возможных реализаций шифра Цезаря. Как обычно, создадим новое приложение и, по аналогии с предыдущим примером, разместим на форме такие же компоненты. У вас получится приблизительно следующее приложение (рис. 12.5).

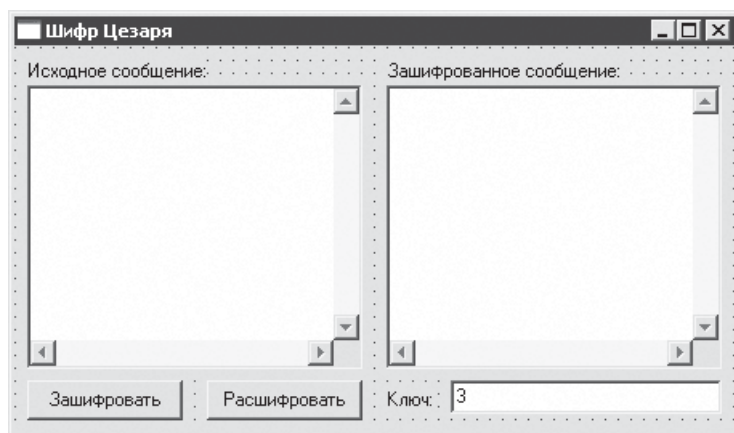


Рис. 12.5. Интерфейс приложения «Шифр Цезаря»

Текстовое поле имеет имя `edKey` и предназначено для задания ключа, при помощи которого будет происходить процесс шифрования или дешифрования. Остальная часть интерфейса программы нам знакома, поэтому останавливаться на ней повторно не имеет смысла.

Перейдем к рассмотрению исходного кода программы. Объявление необходимых типов, описание классов и переменных приведено в листинге 12.13.

Листинг 12.13. Объявление типов и класса нашей формы

```
type
  //исходный алфавит русского языка
  TRusSrcAlphabet = array [0..65] of Char;
  //сопоставление букв алфавита открытого текста и зашифрованного
  TRusDstAlphabet = array [Char] of Char;

  TfmCryptography = class(TForm)
    mmDecryptMessage: TMemo;
    mmEncryptMessage: TMemo;
    lbDecryptMessage: TLabel;
    lbEncryptMessage: TLabel;
    btnEncryptMessage: TButton;
    btnDecryptMessage: TButton;
    edKey: TEdit;
    lbKey: TLabel;
    procedure btnEncryptMessageClick(Sender: TObject);
    procedure btnDecryptMessageClick(Sender: TObject);
  private
    { Private declarations }
    RusDstAlphabet: TRusDstAlphabet;
    function GetKey: Integer;
    procedure RecalcAlphabet(nKey: Integer);
    function EncryptDecryptString(strMsg: String;
                                   nKey: Integer): String;
  public
    { Public declarations }
  end;

var
  RusSrcAlphabet: TRusSrcAlphabet =
    'АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ' +
    'абвгдеёжзийклмнопрстуфхцчшщъыьэюя';
  fmCryptography: TfmCryptography;
```

Далее приведем описание работы методов, решающих определенные подзадачи, которые возникают в процессе решения основной проблемы. Итак, начнем рассмотрение с функции получения введенного пользователем ключа. Ее работа заключается в следующем. Сначала текстовое представление ключа преобразуется в численное представление. Далее проверяется, успешно ли прошло преобразование. Если все отлично, то возвращается полученное значение. В противном случае результатом функции будет `-1`, что свидетельствует о некорректном вводе пользователем ключа. Исходный код данной функции приведен в листинге 12.14.

Листинг 12.14. Функция получения ключа

```
function TfmCryptography.GetKey: Integer;
var
    key, code: Integer;
begin
    Result := -1;
    //получаем текст элемента управления текстовая строка
    Val(edKey.Text, key, code);
    //ошибка во время преобразования к целому числу?
    //или ключ имеет отрицательное значение?
    if (code = 0) and (0 < key) then
        Result := key;
end;
```

Процедура `RecalcAlphabet` имеет один параметр `nKey`, который принимает любое целое значение. Он показывает, на сколько требуется сдвинуть алфавит циклически вперед, то есть если имеется алфавит АБВГД, а `nKey=3`, то результатом будет ВГДАБ. Первым делом алфавит соответствия заполняется один к одному, то есть каждый символ соответствует сам себе. После этого циклом проходимся по строке, содержащей весь необходимый алфавит, подлежащий сдвигу, и переназначаем соответствие этих букв смещенным. Как это делается, можно посмотреть в листинге 12.15.

Листинг 12.15. Функция пересчета алфавита преобразования

```
procedure TfmCryptography.RecalcAlphabet(nKey: Integer);
var
    Ch: Char;
    i: Integer;
    LetCnt: Integer;
begin
    //предварительно все символы в алфавите шифрования
    //соответствуют символам из незашифрованного алфавита
    for Ch := Low(RusDstAlphabet) to High(RusDstAlphabet) do
        RusDstAlphabet[Ch] := Ch;
```

```
//количество символов в алфавите
LetCnt := SizeOf(TRusSrcAlphabet);
//смещаем эталонный алфавит циклически влево на значение,
//заданное ключом nKey
for i := 0 to LetCnt - 1 do
    RusDstAlphabet[RusSrcAlphabet[(i - nKey + LetCnt)
        mod LetCnt]] := RusSrcAlphabet[i];
end;
```

Процедура `RecalcAlphabet` производит необходимую подготовку перед шифрованием или дешифрованием. Результаты процедуры используются в функции `EncryptDecryptString`, где каждая буква открытого текста заменяется соответствующей ей буквой из смещенного алфавита. Это преобразование осуществляется простым проходом по всей строке и выполнением операции замены символа соответствующим ему. Стоит заметить, что для дешифровки сообщения по заданному ключу вычисляется симметричный ему ключ. В результате процесс дешифровки текста сообщения ничем не отличается от процесса его шифровки (листинг 12.16).

Листинг 12.16. Шифрование/дешифрование строки

```
function TfmCryptography.EncryptDecryptString(strMsg: String;
        nKey: Integer): String;
var
    i: Integer;
begin
    //каждый символ строки заменяется соответствующим символом
    //алфавита шифрования
    for i := 1 to Length(strMsg) do
        strMsg[i] := RusDstAlphabet[strMsg[i]];
    Result := strMsg;
end;
```

Теперь у нас есть все, чтобы перейти к решению основной задачи. Процесс шифрования аналогичен процессу дешифрования текста сообщения. Для начала нужно попытаться получить ключ, который ввел пользователь, что мы и делаем. После проверяем значение ключа. Если он равен `-1`, то это значит, что ключ введен неверно и преобразование текста невозможно. Когда все отлично, перед преобразованием текста мы вызываем метод подготовки алфавита с полученным ключом. Стоит отметить, что, когда происходит процесс дешифрования, вычисляется обратный ключ. С его помощью можно получить алфавит, используя который аналогично процессу шифрования получаем открытый текст сообщения. Далее просто: для каждой строки текста сообщения вызывается функция преобразования. На этом каждый метод заканчивает свою работу. Исходный код, соответствующий приведенному выше описанию, показан в листинге 12.17.

Листинг 12.17. Шифрование/дешифрование текста сообщения

```
procedure TfmCryptography.btnEncryptMessageClick(Sender: TObject);
var
  i: Integer;
  nKey: Integer;
begin
  //получаем ключ, с помощью которого будет
  //шифроваться сообщение
  nKey := GetKey;
  //ключ задан верно?
  if nKey = -1 then
    Begin
      MessageDlg('Ошибка: ключ задан неверно', mtError, [mbOk], 0);
      Exit;
    End;
  //получаем алфавит, с помощью которого будет
  //происходить шифрование
  RecalcAlphabet(nKey);
  //предотвращаем перерисовку компонента до тех пор, пока не
  //зашифруем все строки сообщения
  mmEncryptMessage.Lines.BeginUpdate;
  //освобождаем список от любых старых значений
  mmEncryptMessage.Clear;
  //шифруем сообщение построчно
  for i := 0 to mmDecryptMessage.Lines.Count - 1 do
    mmEncryptMessage.Lines.Add(
      EncryptDecryptString(mmDecryptMessage.Lines[i], nKey));
  //заново разрешаем перерисовку компонента
  mmEncryptMessage.Lines.EndUpdate;
end;

procedure TfmCryptography.btnDecpyptMessageClick(Sender: TObject);
var
  i: Integer;
  nKey: Integer;
begin
  nKey := GetKey;
  if nKey = -1 then
```

```
Begin
    MessageDlg('Ошибка: ключ задан неверно', mtError, [mbOk], 0);
    Exit;
End;
//получаем алфавит, с помощью которого будет происходить
//дешифрование
RecalcAlphabet(SizeOf(TRusSrcAlphabet) - nKey
               mod SizeOf(TRusSrcAlphabet));
mmDecryptMessage.Lines.BeginUpdate;
mmDecryptMessage.Clear;
for i := 0 to mmEncryptMessage.Lines.Count - 1 do
    mmDecryptMessage.Lines.Add(
        EncryptDecryptString(mmEncryptMessage.Lines[i], nKey));
mmDecryptMessage.Lines.EndUpdate;
end;
```

Первое, что бросается в глаза при рассмотрении всего текста приложения, — это практически полная идентичность интерфейса и основной части исходного кода. На самом деле это совсем не случайно. Достаточно часто программы пишутся универсально (даже более универсально, чем здесь!). Это основывается на очень простом предположении, что код должен быть многократно, то есть его можно повторно использовать в других приложениях. В результате у вас получается некий шаблон, который позволяет решать целый класс задач. Для этого нужно выполнить несколько маленьких изменений и потом просто можно забыть об этом. Результат выполнения итогового приложения можно увидеть на рис. 12.6.

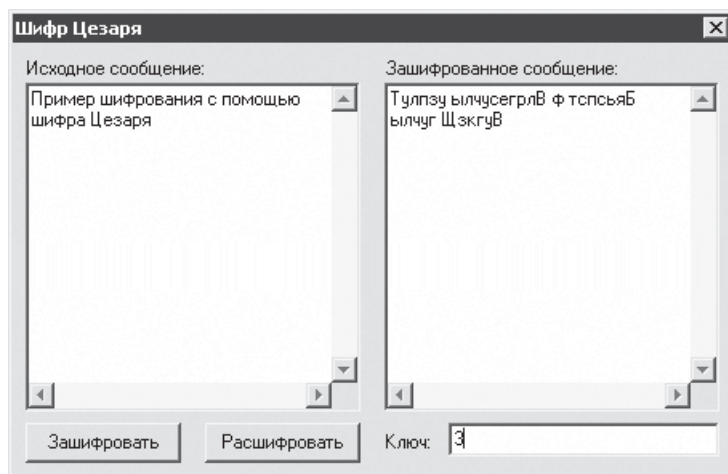


Рис. 12.6. Результат работы приложения «Шифр Цезаря»

12.5. Шифр с автоключом

Шифр, основывающийся на шифре Виженера, в котором или само сообщение, или результирующая криптограмма используются в качестве ключа, называется шифром с автоключом. Шифрование начинается с помощью «первичного ключа» (который является настоящим ключом в нашем смысле) и продолжается с помощью сообщения или криптограммы, смещенной на длину первичного ключа. Рассмотрим пример, в котором первичным ключом является набор букв ЗЕБРА. В табл. 12.2 приведено шифрование, когда в качестве ключа используется сообщение.

Таблица 12.2. Шифр с автоключом (ключ — сообщение)

Сообщение	Ключ	Криптограмма
Ш	З	Я
и	Е	н
ф	Б	х
р	Р	А
с	А	с
а	Ш	Ш
в	и	К
т	ф	ж
о	р	Ю
к	с	Ы
л	а	Л
ю	в	а
ч	т	й
о	о	Ь
м	к	Ц
...

Если же в качестве ключа использовать криптограмму, то получится шифрование, как в табл. 12.3.

Таблица 12.3. Шифр с автоключом (ключ — криптограмма)

Сообщение	Ключ	Криптограмма
Ш	З	Я
и	Е	н
ф	Б	х
р	Р	А
с	А	с

Продолжение ➤

Таблица 12.3. (продолжение)

Сообщение	Ключ	Криптограмма
а	Я	я
в	н	П
т	х	з
о	А	о
к	с	Ы
л	я	к
ю	П	Н
ч	з	Ю
о	о	Ь
м	Ы	З

Теперь, когда понятно, как работает данный шифр, реализуем второй вариант как чуть более сложный, чем первый. В интерфейсе программы менять ничего не станем, поэтому он будет выглядеть как в предыдущем примере (см. рис. 12.5). Только поменяем назначение текстового поля. Теперь оно будет содержать ключ уже не в виде целого числа, а в виде произвольной строки, полностью состоящей из русских букв верхнего и нижнего регистров, за исключением буквы «ё» обоим регистров.

Как обычно, сначала приведем код с объявлением необходимых типов, констант и переменных, а также объявление класса нашей формы. Все это содержится в листинге 12.18.

Листинг 12.18. Объявление типов и класса нашей формы

```
type
```

```
TRusLetters = set of Char;
```

```
TfmEncryptingAutoKey = class(TForm)
```

```
  mmDecryptMessage: TMemo;
```

```
  mmEncryptMessage: TMemo;
```

```
  lbDecryptMessage: TLabel;
```

```
  lbEncryptMessage: TLabel;
```

```
  btnEncryptMessage: TButton;
```

```
  btnDecpyptMessage: TButton;
```

```
  edKey: TEdit;
```

```
  lbKey: TLabel;
```

```
  procedure btnEncryptMessageClick(Sender: TObject);
```

```
  procedure btnDecpyptMessageClick(Sender: TObject);
```

```

private
  { Private declarations }
  function GetKey: String;
  function EncryptString(strEncryptMsg: String;
                        var strKey: String): String;
  function DecryptString(strDecryptMsg: String;
                        var strKey: String): String;
  procedure EncryptDecrypt(SrcLines, DstLines: TStrings;
                          bEncrypt: Boolean);

public
  { Public declarations }
end;

const
  RusLetters: TRusLetters = ['А'..'Я'];
var
  fmEncryptingAutoKey: TfmEncryptingAutoKey;

```

Начнем рассмотрение, как и в предыдущем примере, с функции получения введенного пользователем ключа. Ее работа заключается в следующем. Сначала каждый символ ключа проверяется на принадлежность алфавиту русского языка. Если найден посторонний символ, то результатом работы функции будет пустая строка, что свидетельствует об ошибке ввода ключа пользователем. В случае успешного завершения функции она возвращает исходную строку ключа. Код этой функции приведен в листинге 12.19.

Листинг 12.19. Функция получения ключа

```

function TfmEncryptingAutoKey.GetKey: String;
var
  i: Integer;
begin
  Result := '';
  for i := 1 to Length(edKey.Text) do
    if not (edKey.Text[i] in RusLetters) then
      Exit;
  Result := edKey.Text;
end;

```

Рассмотрим работу функций `EncryptString` и `DecryptString`. На входе они получают строку, которую требуется преобразовать, и первичный ключ. Внешне они очень похожи, но все же отличаются, и эти отличия существенны. Функция шифрования выполняет следующие действия. В цикле осуществляется проход

по строке и проверяется, является ли очередной символ буквой русского алфавита. В случае положительного ответа этот символ преобразуется при помощи очередного символа ключа и добавляется в его конец. Преобразование осуществляется по правилу, которое мы указывали при рассмотрении шифра Виженера: $li = xi + yi \pmod{m}$, то есть символ открытого текста и символ ключа складываются с последующим сокращением этой суммы по модулю m , где m — общее количество букв в алфавите (листинг 12.20).

Листинг 12.20. Функция шифрования строки с помощью ключа и криптограммы

```
function TfmEncryptingAutoKey.EncryptString(strEncryptMsg: String;
    var strKey: String): String;
var
    i: Integer;
begin
    for i := 1 to Length(strEncryptMsg) do
        if strEncryptMsg[i] in RusLetters then
            begin
                strEncryptMsg[i] := Chr(((Ord(strEncryptMsg[i]) -
                    Ord('A')) + (Ord(strKey[1]) - Ord('A')) mod 64 + Ord('A')));
                Delete(strKey, 1, 1);
                strKey := strKey + strEncryptMsg[i];
            end;
        Result := strEncryptMsg;
    end;
```

Функция дешифрования строки с помощью ключа и криптограммы делает следующее. Как и в предыдущей функции, в цикле осуществляется проход по строке и проверяется, является ли очередной символ буквой русского алфавита. При положительном ответе данный символ сначала добавляется в конец ключа, а потом только осуществляется его преобразование. Обратное преобразование символа проходит по следующему правилу: $li = xi - yi \pmod{m}$, то есть из символа преобразованного текста вычитается символ ключа с последующим сокращением этой разности по модулю m , где m — общее количество букв в алфавите. Если результат отрицателен, то происходит дополнение до положительного числа значением m . Как это реализовано, показано в листинге 12.21.

Листинг 12.21. Функция дешифрования строки с помощью ключа и криптограммы

```
function TfmEncryptingAutoKey.DecryptString(strDecryptMsg: String;
    var strKey: String): String;
var
    i: Integer;
```

```

begin
  for i := 1 to Length(strDecryptMsg) do
    if strDecryptMsg[i] in RusLetters then
      begin
        strKey := strKey + strDecryptMsg[i];
        strDecryptMsg[i] := Chr((((Ord(strDecryptMsg[i]) -
Ord('A')) - (Ord(strKey[1]) - Ord('A')) + 64) mod 64 +
Ord('A')));
        Delete(strKey, 1, 1);
      end;
    Result := strDecryptMsg;
  end;
end;

```

Обработчики событий `OnClick` вызывают функцию `EncryptDecrypt` с необходимыми параметрами. У этой функции всего три параметра. Первый указывает на источник текста сообщения, требующего преобразования, второй указывает на приемник преобразованного текста сообщения. Последний параметр определяет тип преобразования текста сообщения. Если он равен `True`, то текст сообщения шифруется и помещается в приемник. В противном случае текст сообщения дешифруется и также помещается в приемник. Это происходит следующим образом. Сначала получается ключ, при помощи которого будет осуществляться преобразование текста сообщения. Если ключ некорректен, то выдаем соответствующее предупреждение и больше ничего не делаем. Если ключ корректен, то в зависимости от последнего параметра вызываем соответствующую функцию преобразования для каждой строки источника текста сообщения и добавляем результат в приемник (листинг 12.22).

Листинг 12.22. Функция шифрования/дешифрования текста сообщения

```

//bEncrypt = True — шифровать
//bEncrypt = False — дешифровать
procedure TfmEncryptingAutoKey.EncryptDecrypt (SrcLines,
                                                DstLines: TStrings; bEncrypt: Boolean);
var
  i: Integer;
  strKey: String;
begin
  strKey := GetKey;
  if strKey <> '' then
    begin
      DstLines.BeginUpdate;
      DstLines.Clear;
      if bEncrypt then

```

```
for i := 0 to SrcLines.Count - 1 do
    DstLines.Add(EncryptString(SrcLines[i], strKey))
else
    for i := 0 to SrcLines.Count - 1 do
        DstLines.Add(DecryptString(SrcLines[i], strKey));
    DstLines.EndUpdate;
end
else
    MessageDlg('Ошибка: ключ задан неверно', mtError, [mbOk], 0);
end;

procedure TfmEncryptingAutoKey.btnEncryptMessageClick(Sender:
                                                    TObject);
begin
    EncryptDecrypt(mmDecryptMessage.Lines,
                  mmEncryptMessage.Lines, True);
end;

procedure TfmEncryptingAutoKey.btnDecryptMessageClick(Sender:
                                                    TObject);
begin
    EncryptDecrypt(mmEncryptMessage.Lines,
                  mmDecryptMessage.Lines, False);
end;

end.
```

Пример того, как работает полученное нами приложение, показан на рис. 12.7.

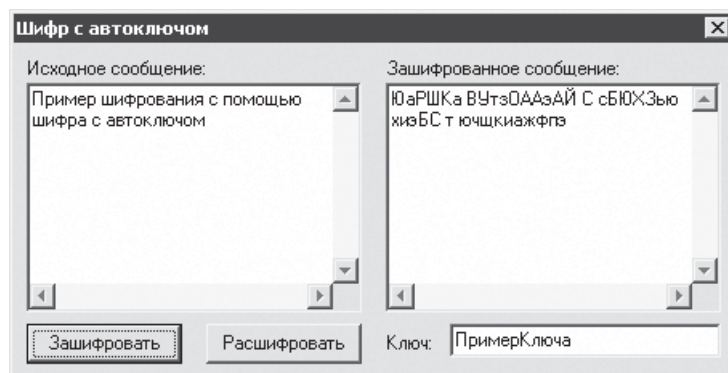


Рис. 12.7. Результат работы приложения «Шифр с автоключом»

12.6. Взлом

В заключение мы рассмотрим один из методов вскрытия шифров. Здесь мы попытаемся реализовать приложение, которое будет способно взломать шифр Цезаря. Оно будет основываться на одном довольно распространенном методе криптоанализа, который называется частотным анализом. Суть его заключается в том, что в большинстве осмысленных текстов есть определенная закономерность относительно того, как часто встречаются те или иные буквы. Следовательно, если мы будем знать, как часто встречается та или иная буква в языке, на котором написано сообщение, мы сможем сделать предположение о том, какие буквы зашифрованы в данной криптограмме. Таким образом, нам требуется подсчитать частоту встречи каждой буквы в криптограмме и после этого сопоставить их с частотами букв, которые известны относительно алфавита заданного языка.

Абсолютная частота буквы есть количество раз, которое она встречается в тексте. Относительная частота — это отношение абсолютной частоты символов к общему количеству символов в сообщении. Теперь оговоримся, что наша программа будет взламывать русскоязычные тексты. Поэтому приведем здесь относительные частоты букв русского языка (табл. 12.4).

Таблица 12.4. Относительные частоты букв русского языка

Буква	Частота	Буква	Частота	Буква	Частота
А	0,063	К	0,028	Х	0,009
Б	0,014	Л	0,035	Ц	0,004
В	0,038	М	0,026	Ч	0,012
Г	0,013	Н	0,052	Ш	0,005
Д	0,025	О	0,090	Щ	0,003
Е	0,072	П	0,023	Ъ	0,015
Ё	0,072	Р	0,040	Ы	0,017
Ж	0,007	С	0,045	Ь	0,015
З	0,016	Т	0,053	Э	0,002
И	0,062	У	0,021	Ю	0,006
Й	0,010	Ф	0,001	Я	0,018

Теоретическая основа для нашей программы имеется, поэтому перейдем к реализации задуманного. Создадим новое приложение. На форму поместим два компонента классов `TMemo` с соответствующими именами `mmDecryptMessage` и `mmEncryptMessage`, три `TLabel`, а также по одному компоненту классов `TEdit` и `TButton` — `edKey` и `btnHackEncrypting` соответственно. Текстовый редактор `mmDecryptMessage` и текстовое поле `edKey` сделаем доступными только для чтения, поскольку мы будем вводить лишь зашифрованное сообщение, а ключ и соответствующий открытый текст будет определяться нашей программой. Результат разработки интерфейса программы показан на рис. 12.8.



Рис. 12.8. Интерфейс программы «Шифр Цезаря — взлом»

Осталось лишь реализовать алгоритм по вскрытию криптограммы. Процесс вскрытия шифра часто оказывается задачей трудоемкой и требующей больше усилий, чем при написании приложений, которые шифруют и дешифруют текст сообщения, используя известный ключ. Приведем исходный код приложения, в котором осуществляется объявление необходимых типов, констант и переменных, а также описание формы приложения (листинг 12.23).

Листинг 12.23. Объявление типов и класса нашей формы

```
type
    //множество всех русских букв
    TRusLetters      = set of Char;
    //исходный алфавит русского языка
    TRusSrcAlphabet  = array [0..65] of Char;
    //относительные частоты русских букв
    TRusFrequency    = array [0..32] of Real;
    TFrequency       = array [Char] of Real;
    TRusDstAlphabet  = array [Char] of Char;

    TfmHackEncrypting = class(TForm)
        mmDecryptMessage: TMemo;
        mmEncryptMessage: TMemo;
        lbDecryptMessage: TLabel;
        lbEncryptMessage: TLabel;
        btnHackEncrypting: TButton;
        edKey: TEdit;
        lbKey: TLabel;
        procedure FormCreate(Sender: TObject);
        procedure btnHackEncryptingClick(Sender: TObject);
```

```

private
  { Private declarations }
  //значение ключа, вычисляемого на основании частотного
  //анализа
  nHackKey: Integer;
  //количество букв русского алфавита в закодированном
  //сообщении
  nCount: LongInt;
  //абсолютная частота букв русского алфавита
  //(то есть количество каждой буквы по отдельности)
  //в зашифрованном сообщении
  AbsFrequency: TFrequency;
  //относительная частота букв русского алфавита в шифровке
  RelFreqInMsg: TFrequency;
  //относительная частота букв русского алфавита
  //в русском языке
  RelFreqInLang: TFrequency;
  RusDstAlphabet: TRusDstAlphabet;
  function UpCaseRus(Ch: Char): Char;
  procedure RecalcAlphabet(nKey: Integer);
  function DecryptString(strDecryptMsg: String;
                        nKey: Integer): String;

public
  { Public declarations }
end;

const
  RusLetters: TRusLetters = ['Ё', 'ё', 'А'..'я'];
  RusSrcAlphabet: TRusSrcAlphabet =
    'АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ' +
    'абвгдеёжзийклмнопрстуфхцчшщъыьэюя';
  //частоты в соответствии с порядком букв в русском алфавите
  RusFrequency: TRusFrequency = (
    0.063, 0.014, 0.038, 0.013, 0.025, 0.072, 0.072, 0.007,
    0.016, 0.062, 0.010, 0.028, 0.035, 0.026, 0.052, 0.090,
    0.023, 0.040, 0.045, 0.053, 0.021, 0.001, 0.009, 0.004,
    0.012, 0.005, 0.003, 0.015, 0.017, 0.015, 0.002, 0.006,
    0.018);

var
  fmHackEncrypting: TfmHackEncrypting;

```

Теперь рассмотрим инициализацию формы приложения. Та таблица, которую мы объявили в виде константы, не очень удобна, поэтому сразу преобразуем ее в другой вид. В новой таблице можно будет, зная только сам символ, получить его относительную частоту для русскоязычных текстов. Как это происходит, показано в исходном коде листинга 12.24.

Листинг 12.24. Обработчик события формы OnCreate

```
procedure TfmHackEncrypting.FormCreate(Sender: TObject);
var
    i, h: Integer;
begin
    h := High(RusSrcAlphabet) div 2;
    for i := Low(RusSrcAlphabet) to High(RusSrcAlphabet) do
        RelFreqInLang[RusSrcAlphabet[i]] := RusFrequency[i mod h];
    end;
```

Вспомогательные методы UpCaseRus, RecalcAlphabet и DecryptString нам уже знакомы. Они выполняют стандартные действия из предыдущих примеров. Поэтому мы только приведем их реализацию для данного случая (листинг 12.25).

Листинг 12.25. Вспомогательные функции

```
function TfmHackEncrypting.UpCaseRus(Ch: Char): Char;
begin
    if Ch = 'ё' then Ch := 'Ё';
    if Ch in ['а'..'я'] then Dec(Ch, 32);
    Result := Ch;
end;

procedure TfmHackEncrypting.RecalcAlphabet(nKey: Integer);
var
    Ch: Char;
    i: Integer;
    LetCnt: Integer;
begin
    for Ch := #0 to #255 do
        RusDstAlphabet[Ch] := Ch;
    LetCnt := SizeOf(TRusSrcAlphabet);
    for i := 0 to LetCnt - 1 do
        RusDstAlphabet[RusSrcAlphabet[(i - nKey + LetCnt)
            mod LetCnt]] := RusSrcAlphabet[i];
    end;
```

```
function TfmHackEncrypting.DecryptString(strDecryptMsg: String;
    nKey: Integer): String;
var
    i: Integer;
begin
    for i := 1 to Length(strDecryptMsg) do
        strDecryptMsg[i] := RusDstAlphabet[strDecryptMsg[i]];
    Result := strDecryptMsg;
end;
```

Основные действия по вскрытию шифра осуществляются в обработчике события `OnClick` кнопки `btnHackEncrypting`. Первым делом подсчитываются абсолютные частоты букв и их общее количество в криптограмме. После этого на основании полученных данных производится расчет относительных частот для каждой из букв. На этом подготовительный этап заканчивается, и начинается процесс вскрытия шифра. Далее проверяется каждый допустимый ключ, сокращенный по модулю количества букв алфавита, без повторения. И для каждого из них вычисляется сумма модуля разности относительных частот, вычисленных для данной криптограммы, и относительных частот для русского языка. Из всех таких сумм выбирается наименьшая как та, при которой относительные частоты букв практически совпадают, а следовательно, наиболее вероятно, что в данном случае ключ, который соответствует этой сумме, и есть искомый. Стоит отметить, что подобные методы вскрытия очень зависимы от сделанного в самом начале предположения. И если тот, кто передавал зашифрованное сообщение, подумал о возможности такого же предположения, то он мог специально сделать все, чтобы метод вскрытия, построенный на нем, не сработал. Например, можно предварительно заархивировать весь текст сообщения. В результате вы получите некий текст с довольно близкими значениями частот для разных букв. В этом случае метод вскрытия по такому алгоритму может оказаться неэффективным. Исходный код приведен в листинге 12.26.

Листинг 12.26. Обработчик события кнопки `OnClick`

```
procedure TfmHackEncrypting.btnHackEncryptingClick(Sender:
    TObject);
var
    Ch: Char;
    i, j, h: Integer;
    Delta, MinDelta: Real;
begin
    //обнуляем счетчик русских букв в закодированном сообщении
    nCount := 0;
    FillChar(AbsFrequency, SizeOf(AbsFrequency), 0);
    for i := 0 to mmEncryptMessage.Lines.Count - 1 do
```

```

for j := 1 to Length(mmEncryptMessage.Lines[i]) do
begin
    //очередной символ сообщения
    Ch := mmEncryptMessage.Lines[i][j];
    //проверяем, принадлежит ли символ
    //множеству русских букв
    if Ch in RusLetters then
    begin
        //подсчитываем количество данной буквы в отдельности
        //и в совокупности со всеми русскими буквами
        AbsFrequency[UpCaseRus(Ch)] :=
            AbsFrequency[UpCaseRus(Ch)] + 1;

        Inc(nCount);
    end;
end;
if nCount = 0 then
begin
    MessageDlg('Дешифровать сообщение нельзя, так как' +
        ' отсутствует русский текст', mtError, [mbOk], 0);
    Exit;
end;
//вычисляем относительные частоты букв в закодированном
//сообщении
FillChar(RelFreqInMsg, SizeOf(RelFreqInMsg), 0);
for i := Low(RusSrcAlphabet) to High(RusSrcAlphabet) div 2 do
    RelFreqInMsg[RusSrcAlphabet[i]] :=
        AbsFrequency[RusSrcAlphabet[i]] / nCount;
//перебираем все возможные ключи и выбираем тот, при
//использовании которого частоты появления русских букв
//в закодированном сообщении наиболее близки к частотам
//появления русских букв в русском языке, то есть сумма
//абсолютных разностей частот букв должна быть наименьшей
h := High(RusSrcAlphabet) div 2 + 1;
MinDelta := h;
for i := 1 to h - 1 do
begin
    Delta := 0;
    for j := 0 to h - 1 do
        Delta := Delta + Abs(RelFreqInLang[RusSrcAlphabet[j]] -
            RelFreqInMsg[RusSrcAlphabet[(i + j + h) mod h]]);
    end;
end;

```

```
//очередная сумма разностей меньше всех предыдущих?  
if MinDelta > Delta then  
begin  
    //запоминаем ее...  
    MinDelta := Delta;  
    //... и запоминаем ключ, при котором получено  
    //данное значение  
    nHackKey := i;  
end;  
end;  
edKey.Text := IntToStr(nHackKey);  
h := High(RusSrcAlphabet) + 1;  
RecalcAlphabet(h - nHackKey mod h);  
mmDecryptMessage.Lines.BeginUpdate;  
mmDecryptMessage.Clear;  
for i := 0 to mmEncryptMessage.Lines.Count - 1 do  
    mmDecryptMessage.Lines.Add(DecryptString(  
        mmEncryptMessage.Lines[i], nHackKey));  
mmDecryptMessage.Lines.EndUpdate;  
end;
```

Итог работы написанного приложения показан на рис. 12.9. Как видите, у нас все получилось!

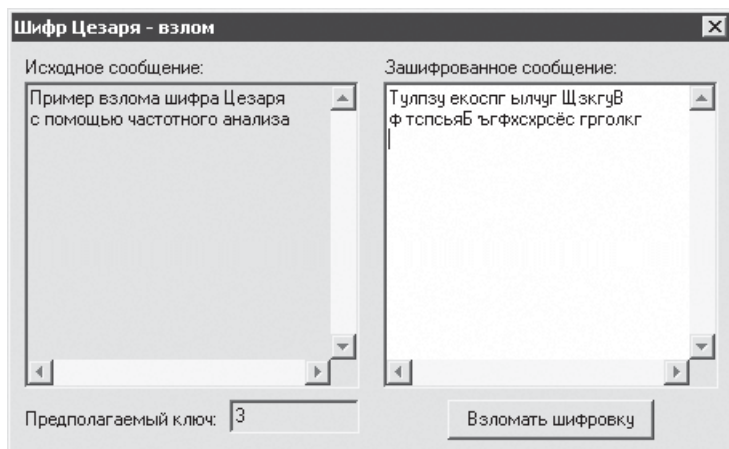


Рис. 12.9. Результат работы приложения «Шифр Цезаря — взлом»

Хочется отметить, что частотный анализ производится не только по частоте использования букв, но и по частоте употребления определенных слов и даже фраз. Например, если ведется переписка между Димой и Николаем, то вероятность, что

Дима начнет свое обращение со слов «Дорогой Николай» больше, чем то, что он начнет его произвольным набором символов «ЫКр2!». Поэтому, когда вы сами попытаетесь вскрыть чей-то шифр, помните о такой возможности, но не забывайте, что существуют и значительно более сложные шифры, чем рассмотренные здесь. Часто для улучшения стойкости этих шифров могут применяться различные методики сжатия информации, чтобы было сложнее воспользоваться частотным анализом, так как в этом случае частоты будут почти одинаковы.

Заключение

Вот и закончилась эта книга. К сожалению, рассмотреть абсолютно все нюансы и интересные подробности программирования в Windows практически невозможно (особенно в книге такого объема). Но мы надеемся, что описанные приемы, алгоритмы и примеры использования возможностей как библиотеки Delphi, так и Windows API хотя бы пролили свет и на некоторые механизмы работы этой ОС, и на другие области, в которых программирование применяется весьма успешно (речь о криптографии).

При написании книги мы старались минимизировать количество примеров, которым невозможно найти применение на практике. Насколько это нам удалось, судить только вам. Нам лишь остается пожелать вам успехов, уважаемый читатель, в программистской практике (неважно, с использованием Delphi или других языков и сред программирования).

Приложение 1



Коды и обозначения основных клавиш

В табл. П1.1 приведены коды, обозначения целочисленных констант и описания основных клавиш.

Таблица П1.1. Коды, обозначения и описания клавиш

Обозначение	Код (десятичный)	Описание
VK_BACK	8	Backspace
VK_TAB	9	Tab
VK_RETURN	13	Enter
VK_SHIFT	16	Shift (левая или правая)
VK_CONTROL	17	Ctrl (левая или правая)
VK_MENU	18	Alt (левая или правая)
VK_PAUSE	19	Pause Break
VK_CAPITAL	20	Caps Lock
VK_ESCAPE	27	Esc
VK_SPACE	32	Пробел
VK_PRIOR	33	Page Up
VK_NEXT	34	Page Down
VK_END	35	End
VK_HOME	36	Home
VK_LEFT	37	←
VK_UP	38	↑
VK_RIGHT	39	→
VK_DOWN	40	↓
VK_SNAPSHOT	44	Print Screen (снятие копии экрана)
VK_INSERT	45	Insert
VK_DELETE	46	Delete
VK_LWIN	91	Windows (левая)
VK_RWIN	92	Windows (правая)
VK_APPS	93	Клавиша для вызова контекстного меню (обычно рядом с правой Windows на клавиатуре для правши)
VK_NUMPAD0	96	0 на цифровой клавиатуре
VK_NUMPAD1	97	1 на цифровой клавиатуре
VK_NUMPAD2	98	2 на цифровой клавиатуре
VK_NUMPAD3	99	3 на цифровой клавиатуре
VK_NUMPAD4	100	4 на цифровой клавиатуре
VK_NUMPAD5	101	5 на цифровой клавиатуре

Продолжение ➤

Таблица П1.1 (продолжение)

Обозначение	Код (десятичный)	Описание
VK_NUMPAD6	102	6 на цифровой клавиатуре
VK_NUMPAD7	103	7 на цифровой клавиатуре
VK_NUMPAD8	104	8 на цифровой клавиатуре
VK_NUMPAD9	105	9 на цифровой клавиатуре
VK_MULTIPLY	106	* на цифровой клавиатуре
VK_ADD	107	+ на цифровой клавиатуре
VK_SUBTRACT	109	– на цифровой клавиатуре
VK_DECIMAL	110	Разделитель дробной части на цифровой клавиатуре
VK_DIVIDE	111	/ на цифровой клавиатуре
VK_F1	112	F1
VK_F2	113	F2
VK_F3	114	F3
VK_F4	115	F4
VK_F5	116	F5
VK_F6	117	F6
VK_F7	118	F7
VK_F8	119	F8
VK_F9	120	F9
VK_F10	121	F10
VK_F11	122	F11
VK_F12	123	F12
VK_F13	124	F13
VK_F14	125	F14
VK_F15	126	F15
VK_F16	127	F16
VK_F17	128	F17
VK_F18	129	F18
VK_F19	130	F19
VK_F20	131	F20
VK_F21	132	F21
VK_F22	133	F22
VK_F23	134	F23
VK_F24	135	F24
VK_NUMLOCK	144	Num Lock

Обозначение	Код (десятичный)	Описание
VK_SCROLL	145	Scroll Lock
VK_LSHIFT	160	Shift (левая)
VK_RSHIFT	161	Shift (правая)
VK_LCONTROL	162	Ctrl (левая)
VK_RCONTROL	163	Ctrl (правая)
VK_LMENU	164	Alt (левая)
VK_RMENU	165	Alt (правая)

Приложение 2



Оконные стили

В приложении представлены таблицы, описывающие следующие оконные стили: общие (табл. П2.1), дополнительные (табл. П2.2), стили кнопок (табл. П2.3), статических надписей (табл. П2.4), текстовых полей (табл. П2.5), списков (табл. П2.6) и стили раскрывающихся списков (табл. П2.7).

Таблица П2.1. Общие оконные стили

Константа	Описание стиля
WS_BORDER	Окно с рамкой
WS_CAPTION	Окно со строкой заголовка
WS_CHILD	Дочернее окно, нельзя использовать вместе со стилем WS_POPUP
WS_CLIPCHILDREN	Исключает из области рисования родительского окна области, занятые дочерними окнами
WS_CLIPSIBLINGS	Используется только со стилем WS_CHILD, не дает при перерисовке дочернего окна рисовать на поверхности соседних дочерних окон
WS_DISABLED	Деактивизированное окно (в него нельзя установить фокус ввода, специальную подсветку текста и т. д.)
WS_DLGFRAME заголовок	Окно с двойной рамкой (как у окна диалога), но без строки
WS_GROUP	Применяется для обозначения элемента управления в группе (удобнее представить это на примере группы переключателей). После того как создано окно со стилем WS_GROUP, все создающиеся далее окна без этого стиля считаются принадлежащими группе, следующее же окно со стилем WS_GROUP считается началом следующей группы и т. д.
WS_HSCROLL	Помещает в окно горизонтальную полосу прокрутки
WS_ICONIC	То же, что и WS_MINIMIZE
WS_MAXIMIZE	Создаваемое окно изначально развернуто
WS_MAXIMIZEBOX	Создаваемое окно имеет кнопку максимизации
WS_MINIMIZE	Создаваемое окно изначально свернуто
WS_MINIMIZEBOX	Создаваемое окно имеет кнопку минимизации
WS_OVERLAPPED	Создает перекрывающееся окно (обычно с рамкой и строкой заголовка)
WS_OVERLAPPEDWINDOW	Сочетание стилей WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX и WS_MAXIMIZEBOX
WS_POPUP	Создает всплывающее окно (не применяется совместно со стилем WS_CHILD)
WS_POPUPWINDOW	Сочетание WS_BORDER, WS_POPUP и WS_SYSMENU
WS_SYSMENU	Создает окно с возможностью открытия системного меню (обычно открывается щелчком кнопкой мыши на значке окна). Применим только для окон со строкой заголовка
WS_TABSTOP	При наличии этого стиля в окно можно установить фокус ввода при помощи клавиши Tab

Продолжение ➤

Таблица П2.1 (продолжение)

Константа	Описание стиля
WS_THICKFRAME	Создает окно с тонкой рамкой, которая может использоваться для изменения размера окна
WS_VISIBLE	Окно первоначально является видимым
WS_VSCROLL	Помещает в окно вертикальную полосу прокрутки

Таблица П2.2. Дополнительные оконные стили

Константа	Описание стиля
WS_EX_ACCEPTFILES	Окно поддерживает перетаскивание в него файлов (например, из Проводника). При перемещении указателя мыши над таким окном он имеет соответствующую форму, а при завершении перетаскивания окно получает сообщение WM_DROPFILES
WS_EX_APPWINDOW	Показывает значок и текст заголовка окна в списке задач
WS_EX_CLIENTEDGE	Создает окно с «вдавленной» рамкой
WS_EX_CONTEXTHELP	Помещает в строку заголовка окна кнопку вызова контекстной правки. При нажатии этой кнопки окно получает сообщение WM_HELP. Стил не может использоваться совместно с WS_MAXIMIZEBOX или WS_MINIMIZEBOX
WS_EX_CONTROLPARENT	Если этот стиль установлен для дочернего окна, то элементы управления этого окна включаются в цепочку при перемещении с использованием клавиши Tab и указателей
WS_EX_DLGMODALFRAME	Создает окно с двойной (по толщине) рамкой. Обычно используется для создания диалоговых окон совместно со стилем WS_CAPTION
WS_EX_LAYERED	Создает окно, прозрачность которого можно регулировать (Windows 2000/XP)
WS_EX_LAYOUTRTL	Отсчет горизонтальных координат шкалы такого окна ведется справа (Windows 2000/XP и версии Windows 98/Me, поддерживающие языки с порядком письма справа налево)
WS_EX_LEFT	Используется по умолчанию. Параметры окна отсчитываются слева направо
WS_EX_LEFTSCROLLBAR	Перемещает полосу прокрутки в левую часть окна
WS_EX_LTRREADING	Используется по умолчанию. Задает порядок вывода текста слева направо
WS_EX_MDICHILD	Этим стилем отмечается MDI-дочернее окно
WS_EX_NOACTIVATE	Окно не активизируется щелчком кнопкой мыши (Windows 2000/XP). Удобно, например, для окон панелей инструментов
WS_EX_NOINHERITLAYOUT	Дочерние окна такого окна не наследуют параметры его шкалы (Windows 2000/XP)
WS_EX_NOPARENTNOTIFY	Созданное с таким стилем окно не посылает родителю сообщения WM_PARENTNOTIFY при своем создании или уничтожении
WS_EX_OVERLAPPEDWINDOW	Комбинация стилей WS_EX_CLIENTEDGE и WS_EX_WINDOWEDGE

Константа	Описание стиля
WS_EX_PALETTEWINDOW	Комбинация стилей WS_EX_WINDOWEDGE, WS_EX_TOOLWINDOW и WS_EX_TOPMOST
WS_EX_RIGHT	Параметры окна отсчитываются справа налево
WS_EX_RIGHTSCROLLBAR	Используется по умолчанию. Помещает полосу прокрутки в правую часть окна
WS_EX_RTLREADING	Определяет порядок вывода текста справа налево (для арабских и прочих языков с таким направлением письма)
WS_EX_STATICEDGE	Создает «вдавленную» трехмерную рамку. Часто используется для статических элементов управления, чтобы они отображались как нередатируемые текстовые поля
WS_EX_TOOLWINDOW	Создает окно для панели инструментов. Такие окна имеют тонкую строку заголовка, тонкую рамку для экономии места
WS_EX_TOPMOST	Создает окно, всегда отображаемое поверх других обычных (без этого стиля) окон

Таблица П2.3. Стили кнопок

Константа	Описание стиля
BS_3STATE	Создает флажок, который может находиться в трех состояниях: установленный, снятый и неактивный (установленный и неактивный)
BS_AUTO3STATE	Создает флажок с тремя состояниями, переключения между которыми и происходят автоматически в следующем порядке: установленный, неактивный, снятый
BS_AUTOCHECKBOX	Создает флажок с двумя состояниями (установленный и снятый), переключение между которыми происходит автоматически
BS_AUTORADIOBUTTON	Создает переключатель, состояние которого контролируется автоматически (может быть установлен только один переключатель)
BS_CHECKBOX	Создает флажок. Его состояние автоматически не изменяется при выборе: соответствующий код нужно писать самостоятельно
BS_DEFPUSHBUTTON	Создает кнопку по умолчанию. Такая кнопка рисуется с более темной рамкой. Нажатие клавиши Enter приводит к нажатию этой кнопки (если фокус не у элемента управления, способного обрабатывать нажатие клавиши Enter, например, многострочного текстового поля). В остальном аналогичен BS_PUSHBUTTON
BS_GROUPBOX	Создает рамку с подписью. Используется для группировки элементов управления
BS_LEFTTEXT	Помещает подписи слева от флажков, переключателей. Аналогичен стилю BS_RIGHTBUTTON
BS_OWNERDRAW	Создает элемент управления, рисование которого предоставляется приложению. При перерисовке элемента управления приложение получает сообщение WM_DRAWITEM. Не может комбинироваться с другими стилями
BS_PUSHBUTTON	Создает обычную кнопку. При нажатии такой кнопки родительскому окну посылается сообщение WM_COMMAND

Продолжение ➤

Таблица П2.3 (продолжение)

Константа	Описание стиля
BS_RADIOBUTTON	Создает переключатель (с подписью)
BS_BITMAP	На кнопку, созданную с таким стилем, может помещаться растровое изображение
BS_BOTTOM	Задаёт вертикальное выравнивание подписи по нижнему краю
BS_CENTER	Задаёт горизонтальное выравнивание подписи по центру
BS_ICON	На кнопку, созданную с таким стилем, может помещаться значок
BS_FLAT	Отменяет рисование трехмерной границы (вдавленностей или выпуклостей) элемента управления
BS_LEFT	Задаёт горизонтальное выравнивание подписи по левому краю
BS_MULTILINE	Включает перенос на новую строку непомещающегося текста
BS_NOTIFY	При наличии этого стиля родительское окно дополнительно получает уведомления BN_KILLFOCUS и BN_SETFOCUS
BS_PUSHLIKE	Создает флажок или переключатель, который выглядит как обычная кнопка. Состояния флажка или переключателя иллюстрируются «залипанием» состояния кнопки. Кнопка нажата в установленном состоянии и не нажата в снятом состоянии
BS_RIGHT	Задаёт горизонтальное выравнивание подписи по правому краю
BS_RIGHTBUTTON	Помещает флажок или переключатель справа от подписи. Аналогичен стилю BS_LEFTTEXT
BS_TEXT	Элемент управления отображает текст
BS_TOP	Задаёт вертикальное выравнивание подписи по верхнему краю
BS_VCENTER	Задаёт вертикальное выравнивание подписи по центру

Таблица П2.4. Стили статических надписей

Константа	Описание стиля
SS_BITMAP	Созданный с таким стилем элемент управления может отображать растровое изображение
SS_BLACKFRAME	Создаёт элемент управления, рамка которого рисуется таким же цветом, как и рамки окон (по умолчанию черным)
SS_BLACKRECT	Закрашивает область элемента управления цветом, используемым для рисования рамок окон (по умолчанию черным)
SS_CENTER	Задаёт горизонтальное выравнивание текста надписи по центру. Осуществляется автоматический перенос по словам и разрыв непомещающихся слов
SS_CENTERIMAGE	Если изображение, отображаемое в элементе управления, меньше, чем размер самого элемента управления, то оно показывается ровно в центре. Если в элементе управления отображается строка текста, то она выравнивается по центру (по горизонтали и вертикали)

Константа	Описание стиля
SS_ENHMETAFILE	Созданный с таким стилем элемент управления может отображать метафайл
SS_ETCHEDFRAME	Рамка элемента управления, созданного с таким стилем, рисуется выпуклой по сравнению с клиентской областью как элемента управления, так и родительского окна
SS_ETCHEDHORZ	Аналогично SS_ETCHEDFRAM, но только для горизонтальных линий границы
SS_ETCHEDVERT	Аналогично SS_ETCHEDFRAM, но только для вертикальных линий границы
SS_GRAYFRAME	Создает элемент управления, рамка которого рисуется таким же цветом, который используется для закрашивания поверхности Рабочего стола (когда нет фонового рисунка)
SS_GRAYRECT	Закрашивает область элемента управления цветом, используемым для закрашивания поверхности Рабочего стола
SS_ICON	Созданный с таким стилем элемент управления может отображать значок
SS_LEFT	Задаёт горизонтальное выравнивание текста надписи по левому краю. Осуществляется автоматический перенос по словам и разрыв непомещающихся слов
SS_LEFTNOWORDWRAP	Задаёт горизонтальное выравнивание текста надписи по левому краю. При выводе текста учитываются символы табуляции, но перенос текста не производится
SS_NOPREFIX	Задаёт не использовать символ & для обозначения клавиш быстрого перехода к элементам управления, подписанным статической надписью. Без этого стиля следующий после & символ подчеркивается и обозначает, какую клавишу в сочетании с Alt нужно нажать для перехода к соответствующему элементу управления
SS_NOTIFY	При установке этого стиля родительское окно получает от элемента управления уведомления STN_CLICKED, STN_DBLCLK, STN_DISABLE и STN_ENABLE
SS_OWNERDRAW	Создаёт элемент управления, рисование которого предоставляется приложению. При перерисовке элемента управления приложение получает сообщение WM_DRAWITEM. Не может комбинироваться с другими стилями
SS_REALSIZEIMAGE	Для элементов управления, отображающих растровые изображения или значки, предотвращает изменение размера элемента в случаях, когда размер изображения не соответствует размеру элемента управления
SS_RIGHT	Задаёт горизонтальное выравнивание текста надписи по правому краю. Осуществляется автоматический перенос по словам и разрыв непомещающихся слов
SS_RIGHTJUST	Для элементов управления, отображающих растровое изображение или значок, делает неподвижным правый нижний угол при изменении размера (по умолчанию на месте остается левый верхний угол)
SS_SIMPLE	Создаёт простую надпись: прямоугольник с одной строкой текста, выровненного по левому краю

Таблица П2.4 (продолжение)

Константа	Описание стиля
SS_SUNKEN	Задаёт рисование «вдавленной» рамки элемента управления
SS_WHITEFRAME	Создаёт элемент управления, рамка которого рисуется таким же цветом, как и фон окна (по умолчанию белым)
SS_WHITERECT	Закрашивает область элемента управления цветом, используемым для рисования фона окон (по умолчанию белым)

Таблица П2.5. Стили текстовых полей

Константа	Описание стиля
ES_AUTOHSCROLL	Задаёт автоматический сдвиг текста по горизонтали, если пользователь вводит или перемещает курсор по строке текста, не помещающейся по ширине в прямоугольнике текстового поля. Если этот стиль не задан, то разрешается вводить только текст, по ширине помещающийся в текстовом поле
ES_AUTOVSCROLL	Задаёт автоматическую прокрутку текста по вертикали (в многострочном текстовом поле)
ES_CENTER	Задаёт горизонтальное выравнивание по центру
ES_LEFT	Задаёт горизонтальное выравнивание по левому краю
ES_LOWERCASE	Включает автоматический перевод вводимых символов в нижний регистр
ES_MULTILINE	Создаёт многострочное текстовое поле
ES_NOHIDESEL	Текстовое поле будет закрашивать выделенный текст даже тогда, когда потеряет фокус ввода
ES_NUMBER	В текстовое поле, созданное с таким стилем, можно ввести только числа (это не распространяется на вставку текста из буфера обмена)
ES_OEMCONVERT	Задаёт автоматическое преобразование вводимого текста в OEM-кодировку, а потом обратно в Windows-кодировку. Применяется для ввода имен файлов (чтобы убедиться в корректности вводимого имени)
ES_PASSWORD	Задаёт отображение заданного символа (обычно *) вместо символов вводимого текста. Используется для ввода паролей
ES_READONLY	Запрещает редактирование текста пользователем
ES_RIGHT	Задаёт горизонтальное выравнивание по левому краю
ES_UPPERCASE	Включает автоматический перевод вводимых символов в верхний регистр
ES_WANTRETURN	Заставляет многострочное текстовое поле обрабатывать нажатие клавиши Enter (выполнять вставку строки). Если этого стиля нет, то при нажатии клавиши Enter выполняется действие, установленное для нее по умолчанию, например нажатие кнопки со стилем BS_DEFPUSHBUTTON

Таблица П2.6. Стили списков (ListBox)

Константа	Описание стиля
LBS_DISABLENOSCROLL	Задаёт отображение полосы прокрутки даже тогда, когда она не нужна (в этом случае полоса неактивна)
LBS_EXTENDEDSEL	Позволяет выбирать несколько элементов при помощи клавиш Shift, Ctrl и мыши

Константа	Описание стиля
LBS_HASSTRINGS	Указывает, что в списке хранятся строки. Список сам занимается выделением и перемещением памяти для корректной работы со строками. По умолчанию этот стиль имеют все списки, кроме OWNERDRAW-списков
LBS_MULTICOLUMN	Создает список, поддерживающий несколько столбцов
LBS_MULTIPLESEL	Включает режим множественного выделения строк (для выделения нескольких строк не нужно использовать Shift или Ctrl)
LBS_NODATA	Используется при большом количестве элементов списка совместно со стилем LBS_OWNERDRAWFIXED. У такого списка не должно быть стилей LBS_SORT и LBS_HASSTRINGS
LBS_NOINTEGRALHEIGHT	Отменяет автоматическое изменение размера списка. Без этого флага размер (высота) списка изменяется так, чтобы в нем не было строк, показанных частично
LBS_NOREDRAW	Запрещает перерисовку списка при изменениях (например, при добавлении или удалении элемента). Полезно при программных манипуляциях с большим количеством элементов списка
LBS_NOSEL	Запрещает выделять элементы списка
LBS_NOTIFY	При наличии этого флага родительское окно получает уведомления при щелчке или двойном щелчке кнопкой мыши на элементах списка
LBS_OWNERDRAWFIXED	Указывает, что рисованием элементов списка занимается приложение (получает сообщение WM_MEASUREITEM при добавлении и WM_DRAWITEM при необходимости отрисовки определенного элемента списка). При этом все элементы списка имеют одинаковую высоту
LBS_OWNERDRAWVARIABLE	То же, что и LBS_OWNERDRAWFIXED, но при этом элементы списка могут иметь разную высоту
LBS_SORT	Включает алфавитную сортировку строк в списке
LBS_STANDARD	Создает список с включенной сортировкой, стилем LBS_NOTIFY и рамкой
LBS_USETABSTOPS	Указывает учитывать при отображении элементов списка символы табуляции
LBS_WANTKEYBOARDINPUT	При наличии этого флага родительское окно получает сообщение WM_VKEYTOITEM при вводе символа с клавиатуры, когда список имеет фокус ввода. Может использоваться для быстрого поиска элементов в списке

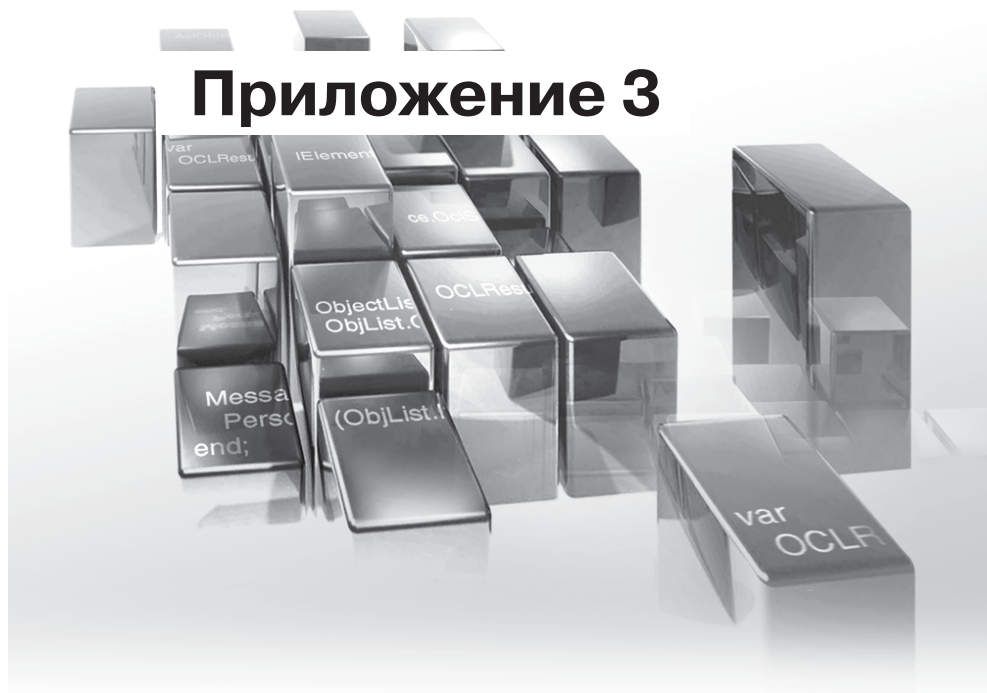
Таблица П2.7. Стили раскрывающихся списков (ComboBox)

Константа	Описание стиля
CBS_AUTOHSCROLL	Задаёт автоматический сдвиг текста по горизонтали, если пользователь вводит или перемещает курсор по строке текста, не помещающейся по ширине в прямоугольнике текстового поля списка. Если этот стиль не задан, то разрешается вводить только текст, помещающийся по ширине в текстовом поле
CBS_DISABLENOSCROLL	Задаёт отображение полосы прокрутки даже тогда, когда она не нужна (в этом случае полоса неактивна)

Продолжение ➤

Таблица П2.7 (продолжение)

Константа	Описание стиля
CBS_DROPDOWN	Отображает (раскрывает) список только тогда, когда пользователь нажимает предназначенную для этого кнопку. Текст в поле можно редактировать
CBS_DROPDOWNLIST	Аналогично CBS_DROPDOWN, только текстовое поле заменяется статической надписью, то есть можно только выбирать элементы из списка
CBS_HASSTRINGS	Указывает, что в списке хранятся строки. Список сам занимается выделением и перемещением памяти для корректной работы со строками. По умолчанию этот стиль имеют все списки, кроме OWNERDRAW-списков
CBS_LOWERCASE	Включает автоматический перевод в нижний регистр вводимых в текстовое поле символов
CBS_NOINTEGRALHEIGHT	Отменяет автоматическое изменение размера списка. Без этого флага размер (высота) списка изменяется так, чтобы в нем не было строк, показанных частично
CBS_OEMCONVERT	Задаёт автоматическое преобразование вводимого в поле текста в OEM-кодировку, а потом обратно в Windows-кодировку
CBS_OWNERDRAWFIXED	Указывает, что рисованием элементов списка занимается приложение (получает сообщение WM_MEASUREITEM при добавлении и WM_DRAWITEM при необходимости отрисовки определенного элемента списка). При этом все элементы списка имеют одинаковую высоту
CBS_OWNERDRAWVARIABLE	То же, что и CBS_OWNERDRAWFIXED, но при этом элементы списка могут иметь разную высоту
CBS_SIMPLE	Задаёт постоянное отображение списка. Текст выделенного в списке элемента показывается в текстовом поле
CBS_SORT	Включает алфавитную сортировку строк в списке
CBS_UPPERCASE	Включает автоматический перевод вводимых в текстовое поле символов в верхний регистр



Приложение 3

Сообщения

В таблицах данного приложения приводятся обозначения констант, описания сообщений, а также назначение параметров `wParam` и `lParam` сообщений. Часто параметры `wParam` или `lParam` являются указателями на структуры. Для экономии места объявления этих структур не приводятся: их можно найти в модуле `Windows`.

Сообщения типа `WM_SETTEXT`, `WM_SETFONT` и подобных им могут как получаться, так и отправляться, то есть могут использоваться для управления окнами. Для большинства сообщений, обозначения которых начинаются с `GET`, требуемое значение возвращается функцией отправки сообщения.

Итак, в приложении представлены таблицы с перечислением некоторых часто используемых сообщений (табл. ПЗ.1), уведомлений от элементов управления (табл. ПЗ.2), сообщений для управления кнопками (табл. ПЗ.3), статическими надписями (табл. ПЗ.4), текстовым полем (табл. ПЗ.5), списком (табл. ПЗ.6) и сообщений для управления раскрывающимся списком (табл. ПЗ.7).

Таблица ПЗ.1. Некоторые часто используемые сообщения

Константа	Описание	wParam	lParam
<code>WM_CREATE</code>	Создание окна	Нет	Адрес структуры <code>TCreateStruct</code>
<code>WM_DESTROY</code>	Уничтожение окна	Нет	Нет
<code>WM_MOVE</code>	Перемещение окна	Нет	Младшие 16 бит задают X-, а старшие — Y-координату
<code>WM_SIZE</code>	Изменение размера окна	Тип изменения размера	Младшие 16 бит задают ширину, а старшие — высоту
<code>WM_ACTIVATE</code>	Активизация или деактивизация окна (переключение активного окна)	<code>WA_ACTIVE</code> , <code>WA_INACTIVE</code> или <code>WA_CLICKACTIVE</code>	Дескриптор окна, состояние которого также изменяется (из которого или на которое происходит переключение)
<code>WM_SETFOCUS</code>	Получение фокуса ввода	Дескриптор окна, потерявшего фокус	Нет
<code>WM_KILLFOCUS</code>	Потеря фокуса ввода	Дескриптор окна, получившего фокус	Нет
<code>WM_ENABLE</code>	Изменение доступности окна	Если не 0, то окно активизируется	Нет
<code>WM_SETTEXT</code>	Установка текста окна (например, текста заголовка)	Нет	Строка с новым текстом (нуль-терминированная)
<code>WM_GETTEXT</code>	Получение текста окна	Длина буфера	Буфер для помещения текста

Константа	Описание	wParam	lParam
WM_GETTEXTLENGTH	Получение длины текста окна	Нет	Нет
WM_PAINT	Перерисовка окна	Нет	Нет
WM_CLOSE	Закрытие окна	Нет	Нет
WM_QUIT	Завершение работы приложения	Код завершения	Нет
WM_SHOWWINDOW	Показывание или скрывание окна	Если не 0, то окно показывается	Состояние, в котором будет показано окно
WM_DRAWITEM	Перерисовка элемента списка или меню	Идентификатор элемента управления (0 для меню)	Адрес структуры TDrawItemStruct
WM_MEASUREITEM	Создание списка, меню или прочего OwnerDraw-элемента управления	Идентификатор элемента управления (0 для меню)	Адрес структуры TMeasureItemStruct
WM_DELETEITEM	Удаление элемента списка	Идентификатор элемента управления (0 для меню)	Адрес структуры TDeleteItemStruct
WM_SETFONT	Установка шрифта	Дескриптор (HFONT) нового шрифта	Нет
WM_GETFONT	Получение шрифта	Нет	Нет
WM_HELP	Вызов справки	Адрес структуры THelpInfo	Нет
WM_KEYDOWN	Нажата клавиша	Код клавиши	Количество повторений (младшие 16 бит) и дополнительная информация
WM_KEYUP	Отпущена клавиша	Код клавиши	Нет
WM_CHAR	Ввод символа	Код символа	Количество повторений (младшие 16 бит) и дополнительная информация
WM_COMMAND	Выбрана команда меню или уведомление от элемента управления	Младшие 16 бит — идентификатор элемента управления, старшие — код уведомления	Дескриптор элемента управления
WM_MOUSEMOVE	Перемещение указателя мыши	Информация о нажатых клавишах	Младшие 16 бит задают X-, а старшие — Y-координату
WM_LBUTTONDOWN	Нажатие левой кнопки мыши	Информация о нажатых клавишах	Младшие 16 бит задают X-, а старшие — Y-координату
WM_LBUTTONUP	Отпускание левой кнопки мыши	Информация о нажатых клавишах	Младшие 16 бит задают X-, а старшие — Y-координату

Продолжение ➤

Таблица ПЗ.1 (продолжение)

Константа	Описание	wParam	lParam
WM_LBUTTONDOWN	Двойной щелчок левой кнопкой мыши	Информация о нажатых клавишах	Младшие 16 бит задают X-, а старшие — Y-координату
WM_RBUTTONDOWN	Нажатие правой кнопки мыши	Информация о нажатых клавишах	Младшие 16 бит задают X-, а старшие — Y-координату
WM_RBUTTONUP	Отпускание правой кнопки мыши	Информация о нажатых клавишах	Младшие 16 бит задают X-, а старшие — Y-координату
WM_RBUTTONDOWNBLK	Двойной щелчок правой кнопкой мыши	Информация о нажатых клавишах	Младшие 16 бит задают X-, а старшие — Y-координату

Таблица ПЗ.2. Уведомления от элементов управления

Константа	Описание
BN_CLICKED	Нажата кнопка
BN_DBLCLK	Двойной щелчок кнопкой мыши, когда указатель находится над кнопкой
BN_DOUBLECLICKED	
BN_KILLFOCUS	Кнопка потеряла фокус
BN_SETFOCUS	Кнопка получила фокус
STN_CLICKED	Щелчок кнопкой мыши на статическом элементе управления (надписи, рисунке, значке)
STN_DBLCLK	Двойной щелчок кнопкой мыши на статическом элементе управления
STN_DISABLE	Статический элемент управления деактивизирован
STN_ENABLE	Статический элемент управления активизирован
EN_CHANGE	Изменен текст в текстовом поле (Edit)
EN_ERRSPACE	Недостаточно памяти для сохранения вводимого текста
EN_HSCROLL	Горизонтальная прокрутка текста в Edit
EN_KILLFOCUS	Текстовое поле потеряло фокус ввода
EN_MAXTEXT	Введенная строка имеет максимальную длину
EN_SETFOCUS	Текстовое поле получило фокус ввода
EN_UPDATE	Приходит перед перерисовкой содержимого текстового поля
EN_VSCROLL	Вертикальная прокрутка текста в Edit
LBN_DBLCLK	Двойной щелчок кнопкой мыши на элементе списка (ListBox)
LBN_ERRSPACE	Недостаточно памяти для добавления нового элемента в список
LBN_KILLFOCUS	Список потерял фокус ввода
LBN_SELCHANGE	Пользователь отменил выделение в списке
LBN_SELCANCEL	Пользователь выделил элемент списка (приходит до перерисовки списка)

Константа	Описание
LBN_SETFOCUS	Список получил фокус ввода
CBN_CLOSEUP	Свернут раскрывающийся список (ComboBox)
CBN_DBLCLK	Двойной щелчок кнопкой мыши на элементе в раскрывающемся списке
CBN_DROPDOWN	Приходит перед показом (разворачиванием) списка
CBN_EDITCHANGE	Изменилось содержимое текстового поля раскрывающегося списка
CBN_EDITUPDATE	Изменилось содержимое текстового поля раскрывающегося списка (приходит перед перерисовкой текстового поля)
CBN_ERRSPACE	Ошибка при выделении памяти для нового элемента
CBN_KILLFOCUS	ComboBox потерял фокус ввода
CBN_SELCHANGE	Изменилось выделение в раскрывающемся списке
CBN_SELENDCANCEL	Приходит, когда пользователь развернул раскрывающийся список, но не выбрал элемент (например, переключил фокус на другой элемент управления)
CBN_SELENDOK	Приходит, когда пользователь развернул раскрывающийся список и выбрал в нем элемент
CBN_SETFOCUS	ComboBox потерял фокус ввода

Таблица ПЗ.3. Сообщения для управления кнопками

Константа	Описание	wParam	lParam
BM_CLICK	Имитация нажатия кнопки	Нет	Нет
BM_GETCHECK	Получение состояния флажка или переключателя (возвращается BST_CHECKED, BST_INDETERMINATE или BST_UNCHECKED)	Нет	Нет
BM_GETIMAGE	Получение дескриптора изображения (тип возвращаемого значения зависит от типа изображения)	IMAGE_BITMAP или IMAGE_ICON	Нет
BM_GETSTATE	Получение состояния элемента управления (возвращается BST_CHECKED, BST_FOCUS, BST_INDETERMINATE, BST_PUSHED или BST_UNCHECKED)	Нет	Нет
BM_SETCHECK	Установка состояния флажка или переключателя	BST_CHECKED, BST_INDETERMINATE или BST_UNCHECKED	Нет

Продолжение ➤

Таблица П3.3 (продолжение)

Константа	Описание	wParam	lParam
BM_SETIMAGE	Установка изображения кнопки	IMAGE_BITMAP или IMAGE_ICON	Дескриптор изображения
BM_SETSTATE	Установка состояния элемента управления	BST_CHECKED, BST_FOCUS, BST_INDETERMINATE, BST_PUSHED или BST_UNCHECKED	Нет
BM_SETSTYLE	Установка стиля элемента управления	Новый стиль	Не 0, если нужна перерисовка

Таблица П3.4. Сообщения для управления статическими надписями

Константа	Описание	wParam	lParam
STM_GETICON	Получение значка, отображаемого в элементе управления	Нет	Нет
STM_GETIMAGE	Получение изображения	IMAGE_BITMAP, IMAGE_CURSOR, IMAGE_ENHMETAFILE или IMAGE_ICON	Нет
STM_SETICON	Установка значка	Дескриптор иконки	Нет
STM_SETIMAGE	Установка изображения	IMAGE_BITMAP, IMAGE_CURSOR, IMAGE_ENHMETAFILE или IMAGE_ICON	Дескриптор изображения

Таблица П3.5. Основные сообщения для управления текстовым полем

Константа	Описание	wParam	lParam
EM_GETLIMITTEXT	Получение максимально допустимой длины текста (0, если не ограничена)	Нет	Нет
EM_GETLINE	Получение текста заданной строки в многострочном текстовом поле	Номер строки текста	Буфер для помещения текста
EM_GETLINECOUNT	Получение количества строк в многострочном текстовом поле	Нет	Нет
EM_GETMODIFY	Получение флага модификации (изменен ли текст)	Нет	Нет
EM_GETPASSWORDCHAR	Получение символа-заменителя, используемого при вводе паролей	Нет	Нет

Константа	Описание	wParam	lParam
EM_GETSEL	Получение положения выделенного текста	Адрес DWORD для сохранения начальной позиции	Адрес DWORD для сохранения конечной позиции
EM_LINEFROMCHAR	Определение номера строки по номеру символа	Номер символа	Нет
EM_LINEINDEX	Определение номера первого символа заданной строки	Номер строки	Нет
EM_LINELENGTH	Определение длины заданной строки	Номер символа	Нет
EM_REPLACESEL	Замена выделенного текста новым	Если не 0, то операцию можно отменить	Новый текст (нуль-терминированная строка)
EM_SETLIMITTEXT	Задание максимально допустимой длины текста	Длина текста	Нет
EM_SETMODIFY	Установка значения флага модификации	Значение флага (не 0, если содержимое изменилось)	Нет
EM_SETPASSWORDCHAR	Задание символа-заменителя для ввода паролей	Код символа	Нет
EM_SETREADONLY	Запрещение или разрешение редактирования содержимого	Ненулевое значение запрещает редактирование	Нет
EM_SETSEL	Установка выделения	Номер первого выделенного символа	Номер последнего выделенного символа

Таблица ПЗ.6. Основные сообщения для управления списком (ListBox)

Константа	Описание	wParam	lParam
LB_ADDSTRING	Добавление строки в список	Нет	Адрес нуль-терминированной строки
LB_DELETESTRING	Удаление строки из списка	Номер строки	Нет
LB_FINDSTRING	Поиск строки (возвращается индекс строки или LB_ERR)	Номер строки, с которой начать поиск	Адрес нуль-терминированной строки
LB_GETCOUNT	Получение количества элементов в списке	Нет	Нет
LB_GETCURSEL	Получение номера выделенной строки (если нет такого, то возвращается LB_ERR)	Нет	Нет

Продолжение ➤

Таблица П3.6 (продолжение)

Константа	Описание	wParam	lParam
LB_GETSEL	Определение состояния элемента списка (выделен/не выделен)	Номер элемента	Нет
LB_GETSELCOUNT	Получение количества выделенных элементов	Нет	Нет
LB_GETSELITEMS	Получение номеров выделенных элементов	Количество элементов в массиве-буфере	Адрес буфера (массив 32-битных целых)
LB_GETTEXT	Получение строки с заданным номером	Номер строки	Адрес буфера
LB_GETTEXTLEN	Получение длины строки	Номер строки	Нет
LB_INSERTSTRING	Вставка строки	Позиция вставки	Адрес нуль-терминированной строки
LB_RESETCONTENT	Очистка списка	Нет	Нет
LB_SELECTSTRING	Поиск и выделение строки	Номер первого элемента для поиска	Адрес нуль-терминированной строки
LB_SETCOUNT	Установка количества элементов в списке, созданном со стилем LBS_NODATA	Количество элементов	Нет
LB_SETCURSEL	Выделение строки с заданным номером	Номер строки или LB_ERR для снятия выделения	Нет
LB_SETSEL	Установка состояния элемента (выделен/не выделен)	Не 0, если нужно выделить элемент	Номер элемента

Таблица П3.7. Основные сообщения для управления раскрывающимся списком (ComboBox)

Константа	Описание	wParam	lParam
CB_ADDSTRING	Добавление строки в список	Нет	Адрес нуль-терминированной строки
CB_DELETESTRING	Удаление строки из списка	Номер строки	Нет
CB_FINDSTRING	Поиск строки (возвращается индекс строки или CB_ERR)	Номер строки, с которой начать поиск	Адрес нуль-терминированной строки
CB_GETCOUNT	Получение количества элементов в списке	Нет	Нет
CB_GETCURSEL	Получение номера выделенной строки (если нет такого, возвращается LB_ERR)	Нет	Нет

Константа	Описание	wParam	lParam
CB_GETEDITSEL	Получение положения выделенного текста в текстовом поле	Адрес DWORD для сохранения начальной позиции	Адрес DWORD для сохранения конечной позиции
CB_GETLBTEXT	Получение строки с заданным номером	Номер строки	Адрес буфера
CB_GETLBTEXTLEN	Получение длины строки	Номер строки	Нет
CB_INSERTSTRING	Вставка строки	Позиция вставки	Адрес нуль-терминированной строки
CB_LIMITTEXT	Задание максимально допустимой длины текста	Длина текста	Нет
CB_RESETCONTENT	Очистка списка	Нет	Нет
CB_SELECTSTRING	Поиск и выделение строки	Номер первого элемента для поиска	Адрес нуль-терминированный строки
CB_SETCURSEL	Выделение строки с заданным номером	Номер строки или CB_ERR для снятия выделения	Нет
CB_SETEXTSEL	Установка выделения	Номер первого выделенного символа	Номер последнего выделенного символа
CB_SHOWDROPDOWN	Отображение раскрывающегося списка	Если не 0, то список показывается	Нет

Чиртик А. А., Борисок В. В., Корвель Ю. И.
Delphi. Трюки и эффекты (+CD)

Заведующий редакцией
Ведущий редактор
Художник
Корректоры
Верстка

*Д. Гурский
Е. Каляева
Л. Адуевская
Т. Лаврович, Е. Павлович, Ю. Цеханович
Е. Зверев*

Подписано в печать 26.10.06. Формат 70×100¹/₁₆. Усл. п. л. 32,25.
Тираж 3000. Заказ 0000.

ООО «Питер Пресс», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Отпечатано по технологии StP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.

КЛУБ ПРОФЕССИОНАЛ



Основанный Издательским домом «Питер» в 1997 году, книжный клуб «Профессионал» собирает в своих рядах знатоков своего дела, которых объединяет тяга к знаниям и любовь к книгам. Для членов клуба проводятся различные мероприятия и, разумеется, предусмотрены привилегии.

Привилегии для членов клуба:

- карта члена «Клуба Профессионал»;
- бесплатное получение клубного издания — журнала «Клуб Профессионал»;
- дисконтная скидка на всю приобретаемую литературу в размере 10% или 15%;
- бесплатная курьерская доставка заказов по Москве и Санкт-Петербургу;
- участие во всех акциях Издательского дома «Питер» в розничной сети на льготных условиях.

Как вступить в клуб?

Для вступления в «Клуб Профессионал» вам необходимо:

- совершить покупку на сайте **www.piter.com** или в фирменном магазине Издательского дома «Питер» на сумму от **800** рублей без учета почтовых расходов или стоимости курьерской доставки;
- ознакомиться с условиями получения карты и сохранения скидок;
- выразить свое согласие вступить в дисконтный клуб, отправив письмо на адрес: postbook@piter.com;
- заполнить анкету члена клуба (зарегистрированным на нашем сайте этого делать не надо).

Правила для членов «Клуба Профессионал»:

- для продления членства в клубе и получения **скидки 10%**, в течение каждого **шести месяцев** нужно совершать покупки на общую сумму от **800** до **1500** рублей, без учета почтовых расходов или стоимости курьерской доставки;
- Если же за указанный период вы выкупите товара на сумму от **1501** рублей, скидка будет увеличена до **15%** от розничной цены издательства.

Заказать наши книги вы можете любым удобным для вас способом:

- по телефону: (812) 703-73-74;
- по электронной почте: postbook@piter.com;
- на нашем сайте: www.piter.com;
- по почте: 197198, Санкт-Петербург, а/я 619 ЗАО «Питер Пост».

При оформлении заказа укажите:

- ваш регистрационный номер (если вы являетесь членом клуба), фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

КНИГА-ПОЧТОЙ



**ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:**

- по телефону: **(812) 703-73-74;**
- по электронному адресу: **postbook@piter.com;**
- на нашем сервере: **www.piter.com;**
- по почте: **197198, Санкт-Петербург, а/я 619,
ЗАО «Питер Пост».**

**ВЫ МОЖЕТЕ ВЫБРАТЬ ОДИН ИЗ ДВУХ СПОСОБОВ ДОСТАВКИ
И ОПЛАТЫ ИЗДАНИЙ:**



Наложенным платежом с оплатой заказа при получении посылки на ближайшем почтовом отделении. Цены на издания приведены ориентировочно и включают в себя стоимость пересылки по почте (**но без учета авиатарифа**). Книги будут высланы нашей службой «Книга-почтой» в течение двух недель после получения заказа или выхода книги из печати.



Оплата наличными при курьерской доставке (**для жителей Москвы и Санкт-Петербурга**). Курьер доставит заказ по указанному адресу в удобное для вас время в течение трех дней.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, код, количество заказываемых экземпляров.

**Вы можете заказать бесплатный
журнал «Клуб Профессионал»**

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают эксклюзивный ассортимент компьютерной, медицинской,
психологической, экономической и популярной литературы

РОССИЯ

Москва м. «Павелецкая», 1-й Кожевнический переулок, д.10; тел./факс (495) 234-38-15,
255-70-67, 255-70-68; e-mail: sales@piter.msk.ru

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а;
тел./факс (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Воронеж Ленинский пр., д. 169; тел./факс (4732) 39-43-62, 39-61-70;
e-mail: pitervrn@comch.ru

Екатеринбург ул. 8 Марта, д. 267б, офис 202;
тел./факс (343) 256-34-37, 256-34-28; e-mail: piter-ural@isnet.ru

Нижний Новгород ул. Совхозная, д. 13; тел. (8312) 41-27-31;
e-mail: office@nnov.piter.com

Новосибирск ул. Немировича-Данченко, д. 104, офис 502;
тел./факс (383) 211-93-18, 211-27-18, 314-23-89; e-mail: office@nsk.piter.com

Ростов-на-Дону ул. Ульяновская, д. 26; тел. (8632) 69-91-22, 69-91-30;
e-mail: piter-ug@rostov.piter.com

Самара ул. Молодогвардейская, д. 33, литер А2, офис 225; тел. (846) 277-89-79;
e-mail: pitvolga@samtel.ru

УКРАИНА

Харьков ул. Суздальские ряды, д. 12, офис 10–11; тел./факс (1038057) 545-55-64, 751-10-02;
e-mail: piter@kharkov.piter.com

Киев пр. Московский, д. 6, кор. 1, офис 33; тел./факс (1038044) 490-35-68, 490-35-69;
e-mail: office@kiev.piter.com

БЕЛАРУСЬ

Минск ул. Притыцкого, д. 34, офис 2; тел./факс (1037517) 201-48-79, 201-48-81;
e-mail: office@minsk.piter.com



Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 703-73-73.**

E-mail: grigorjan@piter.com



Издательский дом «Питер» приглашает к сотрудничеству авторов.
Обращайтесь по телефонам: **Санкт-Петербург — (812) 703-73-72,**
Москва — (495) 974-34-50.



Заказ книг для вузов и библиотек: (812) 703-73-73.
Специальное предложение — e-mail: kozin@piter.com

Башкортостан

Уфа, «Азия», ул. Гоголя, д. 36, офис 5,
тел./факс (3472) 50-39-00, 51-85-44.
E-mail: asiaufa@ufanet.ru

Дальний Восток

Владивосток, «Приморский торговый дом книги»,
тел./факс (4232) 23-82-12.
E-mail: bookbase@mail.primorye.ru

Хабаровск, «Мирс»,
тел. (4212) 30-54-47, факс 22-73-30.
E-mail: sale_book@bookmirs.khv.ru

Хабаровск, «Книжный мир»,
тел. (4212) 32-85-51, факс 32-82-50.
E-mail: postmaster@worldbooks.kht.ru

Европейские регионы России

Архангельск, «Дом книги»,
тел. (8182) 65-41-34, факс 65-41-34.
E-mail: book@atnet.ru

Калининград, «Вестер»,
тел./факс (0112) 21-56-28, 21-62-07.
E-mail: nshibkova@vester.ru
<http://www.vester.ru>

Северный Кавказ

Ессентуки, «Россы», ул. Октябрьская, 424,
тел./факс (87934) 6-93-09.
E-mail: rossy@kmw.ru

Сибирь

Иркутск, «ПродаЛитЪ»,
тел. (3952) 59-13-70, факс 51-30-70.
E-mail: prodalit@irk.ru
<http://www.prodalit.irk.ru>

Иркутск, «Антей-книга»,
тел./факс (3952) 33-42-47.
E-mail: antey@irk.ru

Красноярск, «Книжный мир»,
тел./факс (3912) 27-39-71.
E-mail: book-world@public.krasnet.ru

Нижевартовск, «Дом книги»,
тел. (3466) 23-27-14, факс 23-59-50.
E-mail: book@nvarovsk.wsnet.ru

Новосибирск, «Топ-книга»,
тел. (3832) 36-10-26, факс 36-10-27.
E-mail: office@top-kniga.ru
<http://www.top-kniga.ru>

Тюмень, «Друг»,
тел./факс (3452) 21-34-82.
E-mail: drug@tyumen.ru

Тюмень, «Фолиант»,
тел. (3452) 27-36-06, факс 27-36-11.
E-mail: foliant@tyumen.ru

Челябинск, ТД «Эврика», ул. Барбюса, д. 61,
тел./факс (3512) 52-49-23.
E-mail: evrika@chel.surnet.ru

Татарстан

Казань, «Таис»,
тел. (8432) 72-34-55, факс 72-27-82.
E-mail: tais@bancorp.ru

Урал

Екатеринбург, магазин № 14,
ул. Челюскинцев, д. 23,
тел./факс (3432) 53-24-90.
E-mail: gvardia@mail.ur.ru

Екатеринбург, «Валео-книга»,
ул. Ключевская, д. 5,
тел./факс (3432) 42-56-00.
E-mail: valeo@etel.ru