
Александр Чиртик

# Программирование на C++

CD-ROM

На диске — программы, исходные коды которых приводятся в книге

ПИТЕР®

ББК 32.973.2-018.1
УДК 004.3
Ч-65

**Чиртик А. А.**
Ч-65 Программирование на C++. Трюки и эффекты (+CD). — СПб.: Питер, 2010. — 352 с.: ил. — (Серия «Трюки и эффекты»).

ISBN 978-5-49807-102-2

Данная книга о программировании на C++ не имеет аналогов. В ней описаны оригинальные приемы создания программ и использования системных ресурсов. Применена самая эффективная методика обучения — на живых примерах, которые можно сразу же использовать при создании собственных приложений. На прилагаемом компакт-диске — коды всех программ, рассмотренных в книге. Данное издание можно рекомендовать и опытным программистам, и новичкам.

ББК 32.973.2-018.1
УДК 004.3



ISBN 978-5-49807-102-2 

# Краткое содержание

# Оглавление

# Введение

В мире компьютерной литературы издается большое количество книг, предназначенных для обучения разным подходам к программированию в среде Borland C++ Builder, а раз таких книг очень много, то в чем же особенность данного издания?

В представленной вашему вниманию книге вы сможете найти примеры реализации разнообразных возможностей среды: начиная от создания фильтров, предназначенных для обработки изображений и звука, и заканчивая созданием средств сканирования жесткого диска компьютера на предмет произведенных без вашего ведома изменений. Кроме того, в некоторых примерах используются функции, выходящие за пределы возможностей библиотеки компонентов Borland за счет использования Windows API.

Данная книга предназначена для программистов, обладающих хотя бы небольшим опытом работы в среде программирования Borland C++ Builder. Здесь вы не встретите описания азов программирования на языке C++ и работы в данной среде разработки, зато найдете множество интересных и, надеюсь, полезных примеров программ. В некоторых из приведенных примеров используются объектно-ориентированные возможности языка C++, поэтому вам, уважаемый читатель, чтобы полностью понять принцип работы применяемых в таких примерах алгоритмов, нужно по крайней мере знать основы построения программ с использованием объектно-ориентированного подхода.

Материал книги сгруппирован по тематике рассматриваемых трюков в восемь глав. В конце книги приведены приложения, содержимое которых может вам пригодиться при заимствовании и самостоятельной доработке примеров из данной книги.

# От издательства

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

Мы будем рады узнать ваше мнение!

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

# Глава 1

## Окна

- ❑ Привлечение внимания к окну
- ❑ Окно приложения
- ❑ Растягиваемые формы
- ❑ Окна нестандартной формы
- ❑ Области окна
- ❑ Настраиваемый интерфейс
- ❑ Окна других приложений

При работе в операционной системе Windows окна нам встречаются повсюду. Отсюда, собственно, и название этой операционной системы, в которой идея организации оконного пользовательского интерфейса прижилась очень хорошо.

Для пользователя понятие «окно» чаще всего означает прямоугольную область в рамке с каким-то содержимым, например полем для редактирования текстового документа и, возможно, полосами прокрутки. Для программиста все несколько интереснее: окнами могут быть как окна приложений (которые видит пользователь), так и кнопки, надписи, сами полосы прокрутки и все, что угодно, — все, что пользователь видит или не видит на экране. Только одни окна будут перекрывающимися (например, главное окно приложения), а другие — дочерними (кнопки, полосы прокрутки и т. д.). Дочерние окна располагаются поверх родительских окон, визуально подчеркивая свою принадлежность родительскому окну.

Теперь стоит оговориться, почему в книге иногда употребляется понятие «окно», а иногда — «форма». Под «окном» понимается любое окно, действительно созданное в системе. Форма — это тоже окно, но свойства формы, а также расположение на ней компонентов задаются визуально в среде разработки. Процесс создания самой формы, равно как и всех ее компонентов, скрыт от программиста. Другими словами, форма — это окно, созданное по некоторому шаблону, причем как внутреннее представление шаблона, так и способ создания окна по нему не должны волновать программиста.

На форму в среде разработки Borland C++ Builder могут быть помещены компоненты. Они могут быть как отображаемыми (надпись, кнопка, набор вкладок), так и неотображаемыми (таймер, серверный сокет и т. д.). Отображаемые компоненты (в книге также называются элементами управления) могут быть как оконными, то есть являться полноценными дочерними окнами Windows, так и не обладающими собственным окном, как, например, компонент TLabel, который отображается на форме, но собственного окна не имеет — он просто в нужный момент отображается в нужном месте формы.

В Windows окно является не только частью графического интерфейса — оно также служит для взаимодействия приложения с пользователем или системой. При возникновении какого-либо события в системе, которое затрагивает работающее приложение (например, пользователь провел указателем мыши над окном приложения, изменились системное время или дата, разрешение монитора), окно приложения получает сообщение. Получение сообщения окном представляется как вызов определенной пользовательской функции, зарегистрированной в системе как функция-обработчик сообщения окна (оконная функция).

Каждое окно принадлежит к определенному оконному классу. К одному оконному классу может принадлежать множество окон. Класс определяет свойства и способ реализации одинаковых окон (например, класс окон кнопок). Так, оконная функция одна для всех окон одного класса. В приведении более подробного описания оконных классов особой необходимости нет, тем более что здесь они не пригодятся.

В завершение несколько слов о том, как обрабатываются полученные окном сообщения. Классическая реализация оконной функции предполагает написание подобия большой конструкции switch, в каждой ветви типа case которой должна быть записана реакция на определенное сообщение. Если же для реализации оконного приложения используется библиотека классов наподобие той, что реализована в Borland, оконная функция скрывается в недрах библиотеки. Реакцией на приход определенного сообщения в таком случае является вызов одного из методов объекта, соответствующего окну — получателю сообщения. Такой метод часто называется обработчиком события, хотя его можно назвать и обработчиком сообщения. За исключением написания сложных алгоритмов, прикладной программист, реализующий приложение с пользовательским интерфейсом, по большей части занимается именно написанием нужных обработчиков событий.

## Привлечение внимания к окну

Существует несколько способов привлечь внимание к окну приложения. Одними из самых простых и в то же время достаточно эффективных из неназойливых способов являются инверсия заголовка окна, требующего внимания, изменение окраски кнопки приложения на Панели задач или и то, и другое вместе. Эти способы часто используются в приложениях.

Для инверсии заголовка окна или кнопки на Панели задач никаких функций перерисовки создавать не нужно — достаточно воспользоваться простой API-функцией FlashWindowEx. Данная функция принимает единственный аргумент — указатель на структуру FLASHWINFO, объявленную следующим образом (листинг 1.1).

**Листинг 1.1.** Объявление структуры FLASHWINFO

```
typedef struct {
    UINT  cbSize;
    HWND  hwnd;
    DWORD dwFlags;
    UINT  uCount;
    DWORD dwTimeout;
} FLASHWINFO;
```

В табл. 1.1 приведены описания полей структуры FLASHWINFO.

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

- FLASHW_CAPTION — инвертирует состояние заголовка окна;
- FLASHW_TRAY — заставляет мигать кнопку на Панели задач;
- FLASHW_ALL — реализует совместное использование операторов FLASHW_CAPTION и FLASHW_TRAY;

- FLASHW_TIMER — заставляет периодически изменяться состояние заголовка окна и/или кнопки на Панели задач вплоть до того момента, пока функция FlashWindowEx не будет вызвана с флагом FLASHW_STOP;
- FLASHW_TIMERNOFG — заставляет периодически изменяться состояние 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.

**Таблица 1.1.** Поля структуры FLASHWINFO

| Поле | Тип | Назначение |
|---|---|---|
| cbSize | UINT | Размер структуры FLASHWINFO (для отслеживания версий) |
| hwnd | HWND | Дескриптор окна |
| dwFlags | DWORD | Набор флагов, задающий режим использования функции FlashWindowEx. Значения этого флага 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 таймаута |

Несмотря на необходимость заполнять поля структуры, пользоваться описываемой функцией крайне просто. Так, в листинге 1.2 приведена функция, инвертирующая заголовок окна (hWnd) и его кнопки на Панели задач с интервалом, определяемым параметром time_out в миллисекундах, определенное количество раз (параметр count).

**Листинг 1.2.** Инвертирование заголовка окна и кнопок на Панели задач

```
void FlashWindow( HWND hWnd, UINT count, DWORD time_out )
{
  FLASHWINFO flash_info = {0};
  flash_info.cbSize = sizeof(flash_info);
  flash_info.hwnd = hWnd;
  flash_info.uCount = count;
  flash_info.dwTimeout = time_out;
  flash_info.dwFlags = FLASHW_ALL;
  ::FlashWindowEx( &flash_info );
}
```

Чтобы использовать приведенную в листинге 1.2 функцию, достаточно определиться с интервалом и количеством «миганий» и получать дескриптор окна, например, следующим образом:

```
FlashWindow( pForm->Handle, 3, 100 );
```

Однако в этом случае не изменяется окраска кнопки на Панели задач. Дело в том, что главным окном приложения при использовании Borland C++ Builder (как и Delphi) является не какая-нибудь их форма, а еще одно окно, которое видимо, но имеет нулевой размер. Дескриптором этого окна является как раз значение свойства `Handle` объекта `Application`. Тогда одновременную инверсию заголовка окна и кнопки на Панели задач можно организовать так:

```
FlashWindow( pForm->Handle, 3, 100 );
FlashWindow( Application->Handle, 3, 500 );
```

# Окно приложения

Как было сказано в предыдущем разделе, оконное приложение, реализованное в Borland C++ Builder, имеет окно, которое не отображается, но все же играет важную роль. Это окно можно очень просто увидеть, для чего достаточно задать ему ненулевой размер. Например, так, как показано в листинге 1.3.

**Листинг 1.3.** Отображение окна приложения

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  ::SetWindowPos(Application->Handle, 0, 0, 0, 300, 200,
                 SWP_NOZORDER | SWP_NOMOVE);
}
```

Обратите внимание, что текст окна (рис. 1.1) и является текстом кнопки на Панели задач. Этот текст может быть легко изменен с помощью свойства `Title` объекта `Application`.

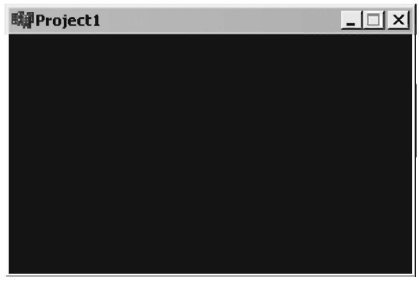


**Рис. 1.1.** Окно приложения

Изменяя видимость (не размер, а именно стиль видимо/невидимо) окна приложения, можно с легкостью скрывать или отображать кнопку на Панели задач. Так, скрыть или показать окно приложения, а с ним и кнопку на Панели задач можно следующим образом (листинг 1.4).

**Листинг 1.4.** Скрытие и отображение кнопки на Панели задач

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  ::ShowWindow(Application->Handle, SW_HIDE); //Скрытие
                                              //кнопки
}
void __fastcall TForm1::Button4Click(TObject *Sender)
{
  ::ShowWindow(Application->Handle, SW_SHOW); //Отображение
                                              //кнопки
}
```

# Растягиваемые формы

При создании сложных форм бывает очень полезно дать пользователю возможность изменять их размер и пропорции. Но как быть с компонентами на растягиваемой или, наоборот, уменьшаемой форме? В данном случае разработчики, использующие среду программирования Borland C++ Builder, могут радоваться: у них в распоряжении есть встроенная возможность привязки компонентов к сторонам формы (так называемый механизм якорей).

Взгляните на группу свойств для задания якорей компонента, показанную на рис. 1.2.

Четыре булевых значения этой группы предназначены для задания привязки краев компонента к краям формы:

- akLeft — левый край компонента сохраняет постоянное расстояние до левого края формы (по умолчанию);
- akTop — верхний край компонента сохраняет постоянное расстояние до верхнего края формы (по умолчанию);
- akRight — правый край компонента сохраняет постоянное расстояние до правого края формы;
- akBottom — нижний край компонента сохраняет постоянное расстояние до нижнего края формы.

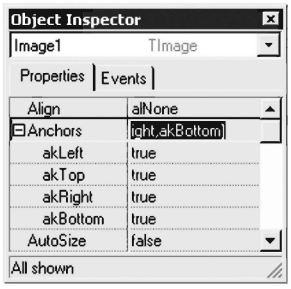


**Рис. 1.2.** Настройка якорей компонента

Таким образом, несколькими щелчками кнопки мыши можно легко создать форму, компоненты которой изменяют свои размеры автоматически при ее растягивании. Пример такой формы приведен на рис. 1.3.

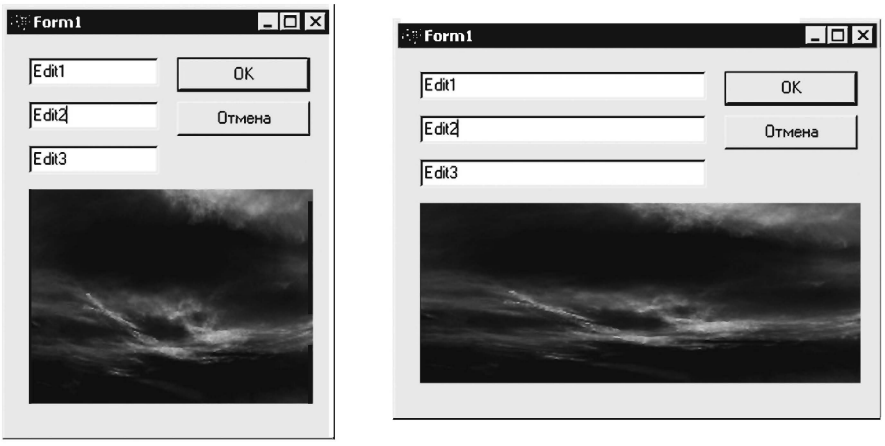


**Рис. 1.3.** Форма до и после изменения размера

Для компонентов показанной на рисунке формы назначены якоря, описанные в табл. 1.2.

**Таблица 1.2.** Якоря формы, показанной на рис. 1.3

| Компонент | Якоря |
|---|---|
| Edit1, Edit2, Edit2 | akLeft, akRight, akTop |
| Button1 (кнопка OK), Button2 (кнопка Отмена) | akRight, akTop |
| Image1 (единственный рисунок на форме) | akLeft, akRight, akTop, akBottom |

При применении описанного здесь механизма стоит учесть еще и возможность задания ограничений на минимальный и максимальный размер как формы, так и любого из ее компонентов. Группа свойств для задания ограничений размера показана на рис. 1.4.

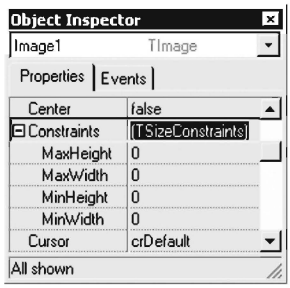


**Рис. 1.4.** Ограничения размеров компонента

При нулевых значениях ограничения на размер формы или компонента не накладываются. Если задать, например, максимальное значение ширины, то форму или компонент нельзя будет растянуть до размера, превышающего заданный. При этом если это ограничение задано для одного из компонентов формы и применяется упомянутый выше механизм якорей, то размер формы станет невозможно увеличивать, как только будет достигнута максимальная ширина одного из компонентов.

# Окна нестандартной формы

Ниже приведены несколько примеров, демонстрирующих способы изменения внешнего вида приложений. Речь идет о создании окон нестандартных непрямоугольных форм. Но сначала ознакомьтесь немного с теорией, чтобы было понятно, как все работает.

## Создание и использование регионов

Рассмотренные далее эффекты основаны на использовании регионов (областей) отсечения — в общем случае сложных геометрических фигур, ограничивающих область прорисовывания окна. По умолчанию окна (в том числе и окна элементов управления) имеют область отсечения, заданную прямоугольным регионом с высотой и шириной, равной высоте и ширине самого окна.

Однако использование регионов прямоугольных форм для указания областей отсечения совсем не обязательно, в чем могут убедиться пользователи Windows XP. Пример использования отсечения по заданному непрямоугольному региону при рисовании произвольного окна наглядно продемонстрирован на рис. 1.5. На этом рисунке буквой «а» обозначен внешний вид, который имела бы форма, будь область

отсечения прямоугольной. Буквой «б» обозначен применяемый регион, формирующий область отсечения. Буквой «в» обозначен вид формы, полученный в результате рисования с отсечением по границам заданного региона.

Теперь вместо рассмотрения подробностей обработки регионов отсечения, которые имеют место в функциях рисования, лучше сразу перейти к рассмотрению операций, позволяющих создавать, удалять и модифицировать регионы.

## Создание и удаление регионов

Создавать несложные регионы различной формы можно с помощью следующих API-функций:

```
HRGN CreateRectRgn( int x1, int y1, int x2, int y2);
HRGN CreateEllipticRgn( int x1, int y1, int x2, int y2);
HRGN CreateRoundRectRgn( int x1, int y1, int x2, int y2, int w,
int h);
```

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


*а* *б* *в*

**Рис. 1.5.** Рисование окна по заданному региону

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

- `x1` и `y1` — горизонтальная и вертикальная координаты левой верхней точки прямоугольника;
- `x2` и `y2` — 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 со скругленными углами. При этом первые четыре параметра функции аналогичны

соответствующим параметрам функции CreateRectRgn. Параметры h и w — ширина и высота сглаживающих углы эллипсов (рис. 1.6).

С помощью трех приведенных функций, если нужно, можно создавать регионы даже очень сложной формы. Это достигается посредством использования многочисленных операций над простыми регионами, как в приведенном далее примере создания региона по битовому шаблону. Однако существует еще одна несложная функция, которая позволяет сразу создавать регионы-многоугольники по координатам вершин многоугольников:

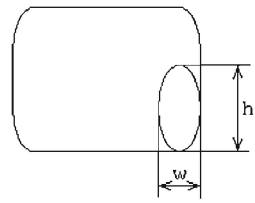


**Рис. 1.6.** Скругление углов прямоугольного региона функцией CreateRoundRectRgn

```
HRGN CreatePolygonRgn(const POINT *points,
int count, int fillmode);
```

Функция CreatePolygonRgn принимает следующие параметры:

- points — указатель на массив структур типа POINT; каждый элемент массива описывает одну вершину многоугольника; координаты не должны повторяться;
- count — количество элементов в массиве, на который указывает параметр points;
- fillmode — режим заливки региона (в данном случае определяет, попадает ли внутренняя область многоугольника в заданный регион).

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

Поскольку регион является GDI-объектом (подробнее о данных объектах читайте в гл. 2), то для его удаления, если он не используется системой, применяется функция удаления GDI-объектов DeleteObject:

```
BOOL DeleteObject(HGDIOBJ hObj);
```

Единственным параметром этой функции является дескриптор GDI-объекта, который после вызова DeleteObject становится недействительным.

## Регион как область отсечения при рисовании окна

Обычно необходимо удалять регион, если он не используется системой. Однако после того как регион назначен окну в качестве области отсечения, удалять его не следует. Функция назначения региона окну имеет следующий вид:

```
int SetWindowRgn(HWND hWnd, HRGN hRgn, BOOL bRedraw);
```

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:

```
int GetWindowRgn(HWND hWnd, HRGN hRgn);
```

Первый параметр функции — дескриптор рассматриваемого 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 — приведены ниже:

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

Ниже приведен пример использования функции GetWindowRgn (предполагается, что приведенный ниже код является телом одного из методов класса формы) (листинг 1.5).

**Листинг 1.5.** Использование функции GetWindowRgn

```
HRGN rgn = ::CreateRectRgn(0,0,0,0); //Первоначальная форма
региона не важна
if ( ::GetWindowRgn(Handle, rgn) != ERROR )
{
  //Операции с копией региона, формирующего область отсечения
  //окна
  //...
}
::DeleteObject(rgn); //Использовалась копия региона, которую
                     //нужно удалить
                     //(здесь или в ином месте, но
                     //самостоятельно)
```

## Операции над регионами

При описании функций создания регионов упоминалось о возможности комбинирования регионов для получения регионов сложных форм. Пришло время кратко рассмотреть операции над регионами. Все операции по комбинированию регионов осуществляются с помощью функции `CombineRgn`:

```
int CombineRgn(HRGN dest, HRGN src1, HRGN src2, int mode);
```

Параметры этой функции имеют следующий смысл:

- `dest` — регион (предварительно созданный), предназначенный для сохранения результата;
- `src1`, `src2` — регионы-аргументы выполнения функции;
- `mode` — тип операции над регионами.

Результат выполнения функции `CombineRgn` при различных значениях параметра `mode` показан в табл. 1.3. В таблице приведены все значения параметра `mode`, кроме `RGN_COPY`, при котором происходит простое копирование региона, заданного параметром `src1`.

**Таблица 1.3.** Операции, выполняемые функцией CombineRgn

| Значение mode | Операция | Пример |
|---|---|---|
| RGN_AND | Пересечение регионов |  |
| RGN_OR | Объединение регионов |  |
| RGN_DIFF | Разность регионов (возвращает часть региона src1, не являющуюся частью src2) |  |
| RGN_XOR | Так называемое исключающее ИЛИ (объединение непересекающихся частей регионов src1 и src2) |  |

Внимательно рассчитывая координаты точек регионов-аргументов, можно создавать регионы самых причудливых форм.

Наконец, после теоретического отступления ознакомьтесь с некоторыми примерами создания и преобразования регионов.

# Непрямоугольная форма

Первым и самым простым примером является создание стандартного непрямоугольного региона и назначение его в качестве области отсечения формы. В листинге 1.6 продемонстрировано, как можно задать область отсечения в форме эллипса.

**Листинг 1.6.** Создание эллиптического региона

```
void __fastcall TForm2::optEllipseClick(TObject *Sender)
{
  //Создание эллиптического региона формы
  HRGN rgn = ::CreateEllipticRgn(0, 0, Width, Height);
  ::SetWindowRgn(Handle, rgn, true);
}
```

В свою очередь, создание и назначение областью отсечения региона в форме скругленного прямоугольника приводится в листинге 1.7.

**Листинг 1.7.** Создание региона в форме прямоугольника с закругленными краями

```
void __fastcall TForm2::optRoundRectClick(TObject *Sender)
{
  //Создание региона-прямоугольника с закругленными краями
  HRGN rgn = ::CreateRoundRectRgn(0, 0, Width, Height,
Width/5, Height/5);
  ::SetWindowRgn(Handle, rgn, true);
}
```

И, наконец, создание региона в форме не слишком сложного многоугольника приведено в листинге 1.8.

**Листинг 1.8.** Создание многоугольного региона

```
void __fastcall TForm2::optPolygonClick(TObject *Sender)
{
  //Создание региона по набору точек
  int w = Width, h = Height;
  POINT pts[] =
  {
    {0,0}, {w,0},
    {4*w/5,h/6}, {w,2*h/6},
    {4*w/5,3*h/6}, {w,4*h/6},
    {4*w/5,5*h/6}, {w,h},
    {0,h}
  };
  int pts_count = sizeof(pts)/sizeof(POINT);
  HRGN rgn = ::CreatePolygonRgn(pts, pts_count, WINDING);
  ::SetWindowRgn(Handle, rgn, true);
}
```

Изменения, происходящие с формой при изменении области отсечения, продемонстрированы на рис. 1.7.

Это наиболее простые случаи применения регионов. Ниже приведен более интересный пример создания комбинированного региона.

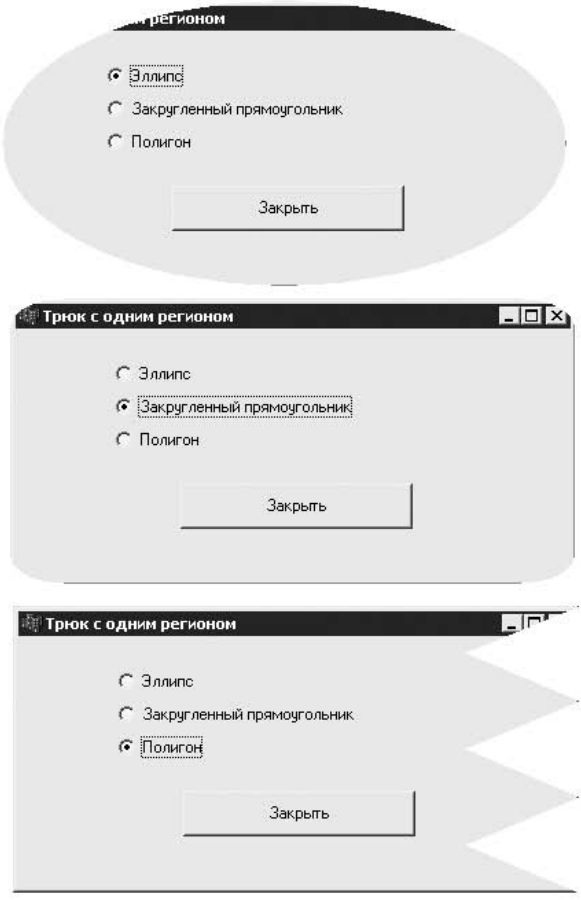


**Рис. 1.7.** Варианты применения простой непрямоугольной области отсечения окна

## «Дырявая» форма

Теперь более сложный пример, демонстрирующий проведение операций над регионами (а точнее, многократное применение одной операции разности регионов). Приведенная ниже программа представляет собой подобие простого графического редактора. В нем с помощью указателя мыши можно указать контур будущего региона отсечения. Внешний вид формы, используемой в рассматриваемом примере, приведен на рис. 1.8.

Переключателями в верхней части формы может выбираться вид создаваемого региона: эллипс, прямоугольник или скругленный прямоугольник, то есть простые регионы, требующие две опорные точки. При щелчке кнопкой мыши точка, в которой находится указатель, становится первой точкой региона. Точка, в которой кнопка мыши будет отпущена, становится второй точкой региона. Во время перемещения

курсора при нажатой кнопке мыши на форме отображается контур будущего региона. Именно в таком состоянии показана форма на рис. 1.8.

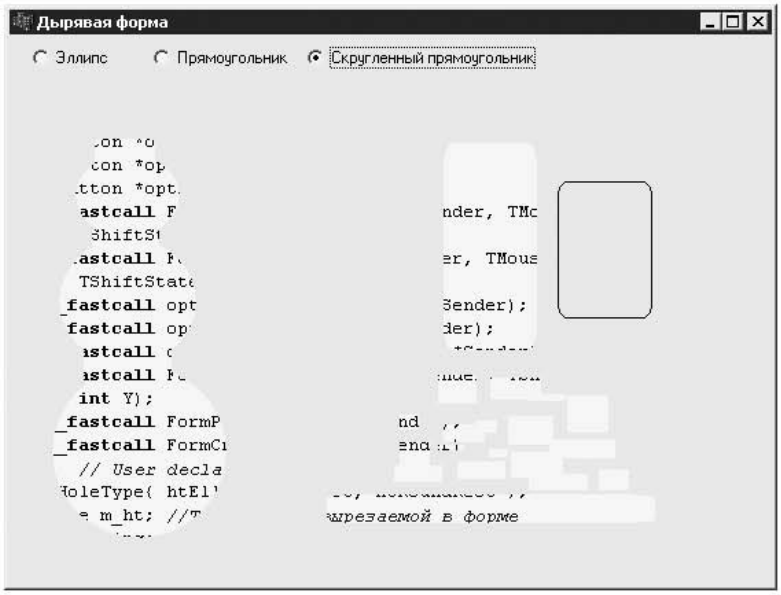


**Рис. 1.8.** «Дырявая» форма

Каждый новый регион исключается от текущего региона, используемого в качестве области отсечения формы. Таким образом, через некоторое время работы с программой в форме образуется приличное количество «дырок». Отсюда и название примера.

Рассмотрение реализации примера начнется с описания основной функции — функции, трансформирующей регион, код которой приведен в листинге 1.9.

**Листинг 1.9.** Создание «дырки» в форме

```
void __fastcall TForm3::FormMouseUp(TObject *Sender, TMouse-
Button Button,
      TShiftState Shift, int X, int Y)
{
  if ( Button == mbLeft && m_drawing )
  {
    m_pt2.x = X;
    m_pt2.y = Y;
    //Переведем координаты точек клиентской области
    //в координаты региона формы
    TRect wndRect;
    ::GetWindowRect(Handle, &wndRect);
    TPoint clientOrg(0, 0);
```

```
    clientOrg = ClientToScreen(clientOrg);
    int dx = clientOrg.x - wndRect.Left;
    int dy = clientOrg.y - wndRect.Top;
    TPoint pt1(m_pt1.x + dx, m_pt1.y + dy);
    TPoint pt2(m_pt2.x + dx, m_pt2.y + dy);
    //Создаем регион-«дырку»
    m_drawing = false;
    HRGN hRgn = NULL;
    if (m_ht == htEllipse)
    {
      hRgn = ::CreateEllipticRgn(pt1.x, pt1.y, pt2.x, pt2.y);
    }
    else if (m_ht == htRect)
    {
      hRgn = ::CreateRectRgn(pt1.x, pt1.y, pt2.x, pt2.y);
    }
    else
    {
      hRgn = ::CreateRoundRectRgn(pt1.x, pt1.y, pt2.x, pt2.y,
                abs(pt2.x - pt1.x)/5, abs(pt2.y - pt1.y)/5);
    }
    //Комбинируем с регионом формы
    HRGN hWndRgn = ::CreateRectRgn(0,0,0,0);
    ::GetWindowRgn(Handle, hWndRgn);
    ::CombineRgn(hWndRgn, hWndRgn, hRgn, RGN_DIFF);
    ::SetWindowRgn(Handle, hWndRgn, true);
    ::DeleteObject(hRgn);
    Refresh();
  }
}
```

Суть всего, что совершается кодом, приведенным в листинге 1.9, очень проста. После завершения выделения области, которую необходимо удалить, остается в распоряжении пара точек (`TPoint`) `m_pt1` и `m_pt2`. Затем анализируется вид региона, который хочет создать пользователь по значению переменной `m_ht`. Далее по координатам двух точек `m_pt1` и `m_pt2` создается регион нужного вида (для скругленного прямоугольника полуоси скругляющего эллипса принимаются равными пятой части ширины и высоты прямоугольника). Создается регион, использующийся как область отсечения формы, в нем вырезается «дырка» с помощью функции `CombineRgn`, после чего полученная разность регионов назначается областью отсечения формы.

В листинге 1.9 заключена основная идея рассматриваемой программы. Но, чтобы все действительно работало так, как описано выше, потребовалось написать несколько дополнительных методов и назначить несколько переменных. Ниже кратко описано, что нужно сделать, чтобы программа начала работать.

С помощью приведенного в листинге 1.10 кода осуществляется объявление формы, по которому можно судить о задействованных для реализации примера обработчиках событий.

**Листинг 1.10.** Объявление формы региона

```
class TForm3 : public TForm
{
__published: // IDE-managed Components
  TRadioButton *optEllipse;
  TRadioButton *optRect;
  TRadioButton *optRoundRect;
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton
Button,
          TShiftState Shift, int X, int Y);
  void __fastcall FormMouseUp(TObject *Sender, TMouseButton
Button,
          TShiftState Shift, int X, int Y);
  void __fastcall optEllipseClick(TObject *Sender);
  void __fastcall optRectClick(TObject *Sender);
  void __fastcall optRoundRectClick(TObject *Sender);
  void __fastcall FormMouseMove(TObject *Sender, TShiftState
Shift, int X,
          int Y);
  void __fastcall FormPaint(TObject *Sender);
  void __fastcall FormCreate(TObject *Sender);
private:  // User declarations
  enum THoleType{ htEllipse, htRect, htRoundRect };
  THoleType m_ht; //Тип «дырки», вырезаемой в форме
  bool m_drawing; //Если true, то начато построение региона-
                  //«дырки»
  TPoint m_pt1, m_pt2; //Начальная и конечная точки для
                       //создания
                       //региона-«дырки»
public:      // User declarations
  __fastcall TForm3(TComponent* Owner);
};
```

Из листинга 1.10 также видно, что три переключателя, используемых для указания вида региона-дырки, называются `optEllipse`, `optRect` и `optRoundRect` для эллипса, прямоугольника и скругленного прямоугольника соответственно. При включении одного из этих переключателей переменной `m_ht` присваивается значение `htEllipse`, `htRect` или `htRoundRect` соответственно.

В листинге 1.11 приведены описания обработчиков щелчков кнопкой мыши и перемещений указателя.

**Листинг 1.11.** Обработка щелчков кнопкой мыши и перемещения указателя

```
void __fastcall TForm3::FormMouseDown(TObject *Sender,
TMouseButton Button,
      TShiftState Shift, int X, int Y)
{
  if ( Button == mbLeft )
  { //Начало построения контура нового региона отсечения
    m_drawing = true;
    m_pt1.x = m_pt2.x = X;
    m_pt1.y = m_pt2.y = Y;
  }
}
void __fastcall TForm3::FormMouseMove(TObject *Sender,
TShiftState Shift,
      int X, int Y)
{
  if (m_drawing)
  {
    m_pt2.x = X;
    m_pt2.y = Y;
    Refresh(); //Перерисовка контура будущего региона
отсечения
  }
}
```

Чтобы в ходе перемещения указателя были видны контуры будущего региона отсечения, реализован следующий обработчик перерисовки формы (листинг 1.12).

**Листинг 1.12.** Создание контуров региона

```
void __fastcall TForm3::FormPaint(TObject *Sender)
{
  if (m_drawing)
  {
    if (m_ht == htEllipse)
    {
      Canvas->Ellipse(m_pt1.x, m_pt1.y, m_pt2.x, m_pt2.y);
    }
    else if (m_ht == htRect)
    {
      Canvas->Rectangle(m_pt1.x, m_pt1.y, m_pt2.x, m_pt2.y);
    }
    else
    {
      Canvas->RoundRect(m_pt1.x, m_pt1.y, m_pt2.x, m_pt2.y,
               abs(m_pt2.x - m_pt1.x)/5, abs(m_pt2.y - m_pt1.
y)/5);
```

```
    }
  }
}
```

И последнее — в ряде случаев первый вызов функции `GetWindowRgn`, задействованной в листинге 1.9, может возвращать пустой регион, потому необходимо явно определить первоначальный регион для области отсечения при создании формы так, как показано в листинге 1.13.

**Листинг 1.13.** Задание первоначального региона для области отсечения

```
void __fastcall TForm3::FormCreate(TObject *Sender)
{
  HRGN hWndRgn = ::CreateRectRgn(0, 0, Width, Height);
  ::SetWindowRgn(Handle, hWndRgn, true);
}
```

Это все, что потребовалось для реализации рассматриваемого примера.

## Использование шаблона

В завершение рассмотрим еще один небольшой пример применения операций над регионами. Теперь регион для области отсечения формы будет вырезаться по битовому шаблону. Шаблоном будет служить растровое изображение. Любой пиксел изображения с цветом, отличным от цвета фона, будет включен в регион, а на месте пиксела, цвет которого соответствует цвету фона, будет «дырка».

Пример формы, регион области отсечения которой составлен по битовому шаблону, показан на рис. 1.9. Для пущей выразительности на форму еще и наложен рисунок, по которому была составлена область отсечения.


**Рис. 1.9.** Регион, вырезанный по растровому шаблону

В листинге 1.14 приведена главная функция этого примера, позволяющая создавать регион по изображению, хранимому в значении параметра `pict`. В результирующий регион включаются все точки, цвет которых не равен значению параметра `backcolor` (цвет фона).

**Листинг 1.14.** Формирование региона по растровому изображению

```
HRGN RegionFromPicture(TPicture *pict, TColor backcolor)
{
  HRGN resRgn = ::CreateRectRgn(0, 0, 0, 0); //Результирующий
регион
  HRGN rgn;
  int x, y, xFirst;
  //Анализируем каждую скан-линию рисунка (по горизонтали)
  for (y = 0; y < pict->Height; y++)
  {
    x = 0;
    while (x < pict->Width)
    {
      if (pict->Bitmap->Canvas->Pixels[x][y] != backcolor)
      {
        xFirst = x;
        x++;
        //Определим часть линии, окрашенной не цветом фона
        while (x < pict->Width &&

pict->Bitmap->Canvas->Pixels[x][y] != backcolor) x++;
        //Создаем регион для части скан-линии и добавляем его
        //к результирующему
        rgn = ::CreateRectRgn(xFirst, y, x-1, y+1);
        ::CombineRgn(resRgn, resRgn, rgn, RGN_OR);
        ::DeleteObject(rgn);
      }
      x++;
    }
  }
  return resRgn;
}
```

Обратите внимание, что для реализации примера оказалось достаточно одного вида операций над регионами — операции ИЛИ. Единственной оптимизацией в этом примере явилось то, что полученный на предыдущих этапах регион соединяется не с каждой новой точкой рисунка подходящего цвета, а с частью скан-линии, состоящей из таких точек.

Для назначения созданного по рисунку региона областью отсечения формы реализован простой обработчик события `FormCreate`, описание которого приведено в листинге 1.15.

**Листинг 1.15.** Обработчик события FormCreate

```
void __fastcall TForm4::FormCreate(TObject *Sender)
{
  ::SetWindowRgn( Handle,
                  RegionFromPicture(m_pict, (TColor)RGB(255,2
55,255)),
                  true );
}
```

Для правильной инициализации и удаления объекта `TPicture`, указатель на который хранится в переменной `m_pict` (элемент класса `TForm4`), достаточно написать следующий код (листинг 1.16).

**Листинг 1.16.** Создание, загрузка и удаление изображения

```
__fastcall TForm4::TForm4(TComponent* Owner)
  : TForm(Owner)
{
  m_pict = new TPicture;
  m_pict->LoadFromFile("шаблон.bmp");
}
__fastcall TForm4::~TForm4()
{
  delete m_pict;
}
```

Если же захочется наложить на форму изображение, по которому была сформирована область отсечения, можно применить обработчик события `FormPaint`, реализованный в листинге 1.17.

**Листинг 1.17.** Перенос изображения на форму

```
void __fastcall TForm4::FormPaint(TObject *Sender)
{
  //Переведем координаты точек клиентской области в
  //координаты
  //региона формы
  TRect wndRect;
  ::GetWindowRect(Handle, &wndRect);
  TPoint clientOrg(0, 0);
  clientOrg = ClientToScreen(clientOrg);
  int dx = clientOrg.x - wndRect.Left;
  int dy = clientOrg.y - wndRect.Top;
  //Собственно рисование
  this->Canvas->Draw(-dx, -dy, m_pict->Bitmap);
}
```

Вся сложность в наложении изображения в точности на область отсечения формы кроется в переводе координат левой верхней точки клиентской области формы

в координаты, начало которых находится в левом верхнем углу формы (в координаты формы). Ведь регион для области отсечения создавался в системе координат формы, а выводить изображение приходится в координатах клиентской области формы.

# Области окна

Как вы, наверное, не раз замечали, многие окна имеют специальные области, такие как строка заголовка, кнопка системного меню (значок в левой части заголовка), границы окна и др. Каждый раз, когда указатель мыши перемещается над окном, окну посылается сообщение `WM_NCHITTEST`. Так система узнает, над какой именно областью окна находится указатель мыши.

Приведенный ниже трюк позволяет подменить расположение границ и строки заголовка окна. Достигается это посредством реализации собственного обработчика события `WMNCHitTest`, код которого приведен в листинге 1.18.

**Листинг 1.18.** Подмена областей окна

```
void __fastcall TForm1::WMNCHitTest(Messages::TWMNCHitTest
&Message)
{
  if ( InControl(borderBottom,Message.Pos) )
    Message.Result = HTBOTTOM;
  else if ( InControl(borderTop,Message.Pos) )
    Message.Result = HTTOP;
  else if ( InControl(borderLeft,Message.Pos) )
    Message.Result = HTLEFT;
  else if ( InControl(borderRight,Message.Pos) )
    Message.Result = HTRIGHT;
  else if ( InControl(borderTopLeft, Message.Pos) )
    Message.Result = HTTOPLEFT;
  else if ( InControl(borderBottomLeft, Message.Pos) )
    Message.Result = HTBOTTOMLEFT;
  else if ( InControl(borderTopRight, Message.Pos) )
    Message.Result = HTTOPRIGHT;
  else if ( InControl(borderBottomRight, Message.Pos) )
    Message.Result = HTBOTTOMRIGHT;
  else if ( InControl(title, Message.Pos) )
    Message.Result = HTCAPTION;
  else // Message.Result = HTCLIENT;
    Message.Result = ::DefWindowProc(Handle, Message.Msg, 0,

65536 * Message.YPos + Message.XPos);
}
```

Чтобы приведенный в листинге 1.18 код начал работать, в объявление класса формы `TForm1` достаточно добавить следующий текст (листинг 1.19).

**Листинг 1.19.** Объявление обработчика события WMNCHitTest и связывание с ним сообщения WM_NCHITTEST

```
void __fastcall WMNCHitTest(Messages::TWMNCHitTest &Message);
BEGIN_MESSAGE_MAP
  VCL_MESSAGE_HANDLER(WM_NCHITTEST, TWMNCHitTest,
WMNCHitTest);
END_MESSAGE_MAP(TForm);
```

Суть трюка такова: на форме (рис. 1.10) размещены графические примитивы (`TShape`), которые в последующем будут считаться частями рамки или строкой заголовка окна. С помощью обработчика `WMNCHitTest` проверяется, над каким компонентом формы находится указатель мыши, и возвращается код соответствующей области. Таким образом, размер формы можно изменить, потянув за один из компонентов `TShape`, формирующих подобие рамки окна, а переместить форму можно за скругленный прямоугольник с надписью Область заголовка окна.


**Рис. 1.10.** «Новые» области окна

Для упрощения проверки положения указателя мыши предусмотрена функция, код которой приведен в листинге 1.20.

**Листинг 1.20.** Проверка принадлежности точки компоненту

```
BOOL InControl(TControl *pCtrl, TSmallPoint pt)
{
  TRect rc = pCtrl->ClientRect;
  TPoint point = pCtrl->ScreenToClient(TPoint(pt.x, pt.y));
  return ::PtInRect(&rc, point);
}
```

Единственная задача, решаемая данной функцией, — перевод экранных координат курсора в координаты клиентской области компонента.

В рассмотренном здесь примере вручную задается расположение лишь некоторых областей, которые могут располагаться в окне. Для определения остальных областей используется обработчик сообщений по умолчанию `DefWindowProc` (API-функция). Ниже приведен полный список значений, которые могут возвращаться при обработке события `WMNCHitTest`:

- `HTBORDER` — возвращается, если указатель мыши находится над границей окна (при неизменяемом размере окна);
- `HTBOTTOM`, `HTTOP`, `HTLEFT`, `HTRIGHT` — значения возвращаются, если указатель находится над нижней, верхней, левой или правой границей окна соответственно (размер окна можно изменить, «потянув» за границу);

- HTBOTTOMLEFT, HTBOTTOMRIGHT, HTTOPLEFT, HTTOPRIGHT — возвращаются, если указатель находится в левом нижнем, правом нижнем, левом верхнем или правом верхнем углу окна (размер окна можно изменять по диагонали);
- HTSIZE, HTGROWBOX — значения возвращаются, если указатель находится над областью, предназначенной для изменения размера окна по диагонали (обычно в правом нижнем углу окна);
- HTCAPTION — возвращается, если указатель находится над строкой заголовка окна (удерживая нажатой кнопку мыши, окно можно перемещать за эту область);
- HTCLIENT — значение возвращается, если указатель находится над клиентской областью окна;
- HTCLOSE — возвращается, если указатель находится над кнопкой закрытий окна;
- HTHELP — значение возвращается, если указатель находится над кнопкой вызова контекстной справки;
- HTREDUCE, HTMINBUTTON — возвращаются, если указатель находится над кнопкой минимизации окна;
- HTZOOM, HTMAXBUTTON — значения возвращаются, если указатель находится над кнопкой максимизации окна;
- HTMENU — возвращается, если указатель находится над полоской меню окна;
- HTSYSMENU — значение возвращается, если указатель находится над иконкой окна (используется для вызова системного меню);
- HTHSCROLL, HTVSCROLL — возвращаются, если указатель находится над вертикальной или горизонтальной полоской 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 (beep), сообщая об ошибке.

# Настраиваемый интерфейс

В этом разделе будут рассмотрены два примера реализации гибкого пользовательского интерфейса: использования технологии Drag & Dock и создания формы с изменяемым расположением компонентов.

## Стыкуемые формы

Технология стыковки Drag & Dock позволяет легко создавать приложения, внешний вид которых в определенных пределах может изменяться пользователем. Разработчики, использующие среду Borland C++ Builder, могут радоваться тому, что возможности стыковки уже встроены в формы и некоторые другие компоненты, а потому для использования технологии стыковки остается настроить лишь пару параметров и написать пару строк кода.

За поведение компонентов и форм в процессе стыковки отвечают два следующих свойства:

- `DockSite` — принимает значение `true`, если компонент может быть портом стыковки, и `false`, если нет;
- `DragKind` — задает вид компонента при перемещении (значение `dkDrag` для простого перемещения и `dkDock` для перемещения при стыковке).

Ниже приведен пример стыковки форм. Приложение состоит из трех форм: порта стыковки (Form1) и двух стыкуемых к порту форм (Form2 и Form3). Значения свойств, относящихся к стыковке, заданы следующим образом:

- Form1 — `DockSite` = `true`, `DragKind` = `dkDrag`;
- Form2, Form3 — `DockSite` = `false`, `DragKind` = `dkDock`.

Внешний вид форм до и после стыковки приведен на рис. 1.11.

В данном примере стыковку форм можно отменить или включить в любой момент. Для этого в форме-порту стыковки (Form1) предусмотрено меню Панели. Для отмены стыковки используется пункт меню Панели ▸ Разбросать, а для включения (принудительной стыковки форм) — пункт меню Панели ▸ Прикрепить. Обработчики для этих пунктов меню приведены в листинге 1.21.

**Листинг 1.21.** Отмена стыковки и принудительная стыковка

```
void __fastcall TForm1::mnuUndockClick(TObject *Sender)
{
  //Открепляем панели
  Form2->ManualDock(NULL);
  Form3->ManualDock(NULL);
}
void __fastcall TForm1::mnuDockClick(TObject *Sender)
{
  //Помещение панелей (двух других форм) на эту форму
  Form2->ManualDock(this);
  Form3->ManualDock(this);
}
```

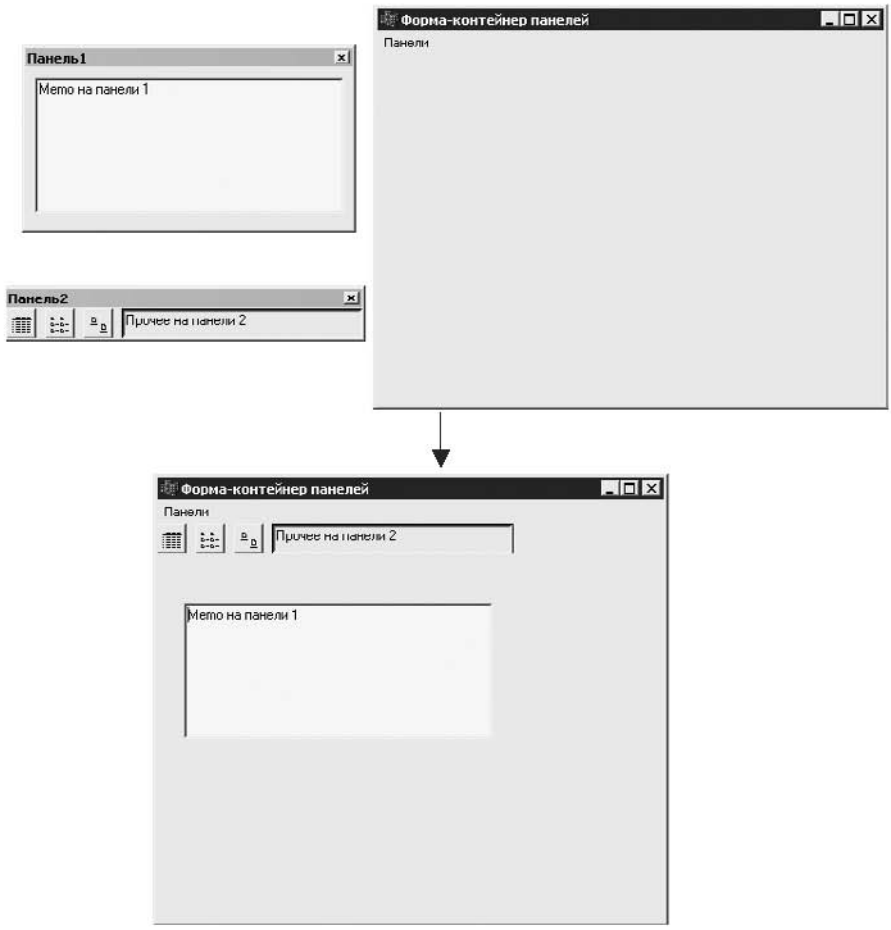


**Рис. 1.11.** Формы до и после стыковки

## Перемещаемые компоненты

Рассматриваемый здесь пример демонстрирует возможность изменения расположения компонентов на форме во время выполнения программы. Пример немного искусственный и может быть реализован только для компонентов, имеющих свойство `Handle` (то есть являющихся полноценными окнами). К слову, компонент TLabel таким компонентом не является, но эти компоненты вполне можно заменять аналогичными (к примеру, вместо TLabel использовать TEdit, настроенный только для чтения).

На рис. 1.12 показан внешний вид формы при первом запуске приложения.

Допустим, необходимо изменить расположение компонентов, заданное разработчиком. Для этого выберите команду меню Перемещение ▸ Включить. В результате форма примет вид, показанный на рис. 1.13.

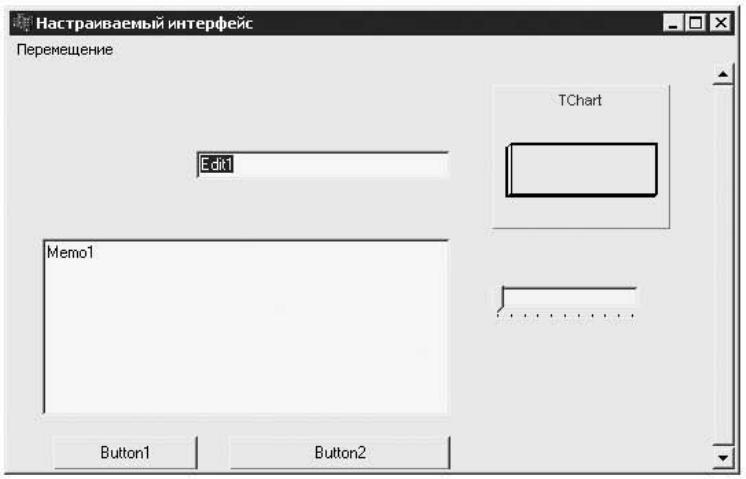


**Рис. 1.12.** Исходный вид формы

Перемещая окна компонентов и изменяя их размер, а также размер формы, можно создать более удобный вид интерфейса (рис. 1.14).

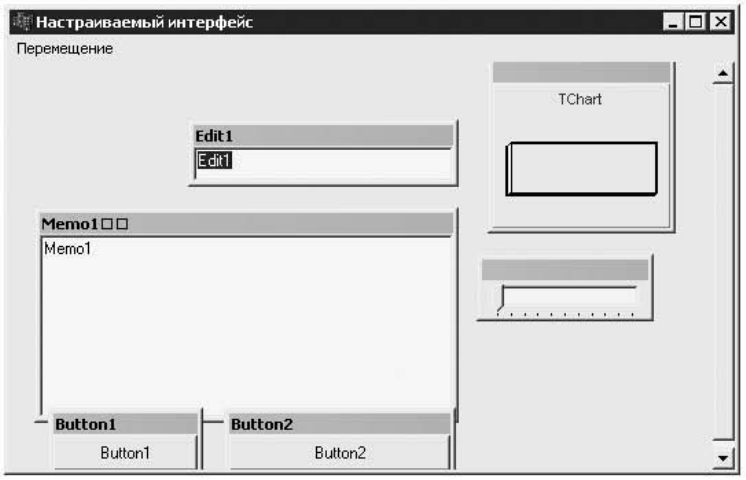


**Рис. 1.13.** Вид формы после включения функции перемещения компонентов

Для завершения нужно закрепить компоненты на форме, выполнив команду меню Перемещение ▸ Выключить. В результате форма примет окончательный вид (рис. 1.15), причем расположение и размер ее компонентов сохранятся при выходе из приложения, и при последующих запусках форма будет принимать заданный ранее вид.

Реализация такого интерфейса на самом деле не представляет никакой сложности. Внешний вид и поведение окна (возможность его перемещения) во многом определяются стилем, который приписан этому окну. Так, окна компонентов формы

часто имеют стиль WS_CHILD, WS_VISIBLE и еще несколько дополнительных стилей, но не имеют рамки и заголовка из-за отсутствия соответствующих стилей. Но что мешает добавить стили, обеспечивающие появление рамки и строки заголовка? Собственно, в этом и кроется суть рассматриваемого здесь трюка. Чтобы сделать доступной возможность перемещения компонентов формы, окнам этих компонентов нужно добавить стили WS_OVERLAPPED, WS_THICKFRAME и WS_CAPTION, а также расширенный стиль WS_EX_TOOLWINDOW, позволяющий задать размер строки заголовка, как у панелей инструментов. Чтобы отключить возможность перемещения, эти стили нужно удалить.

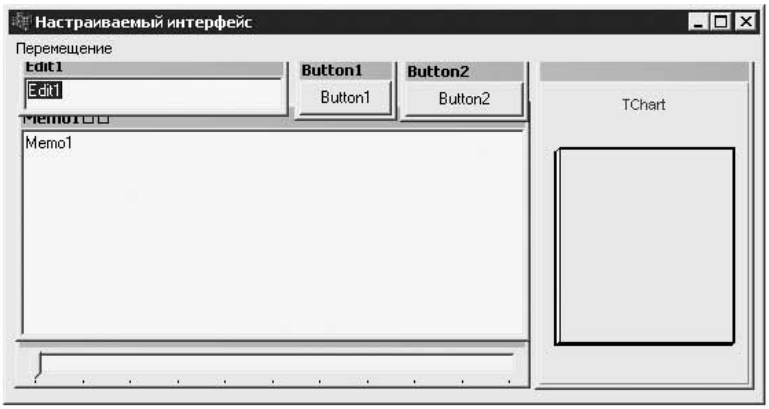


**Рис. 1.14.** Вид формы после перемещения компонентов

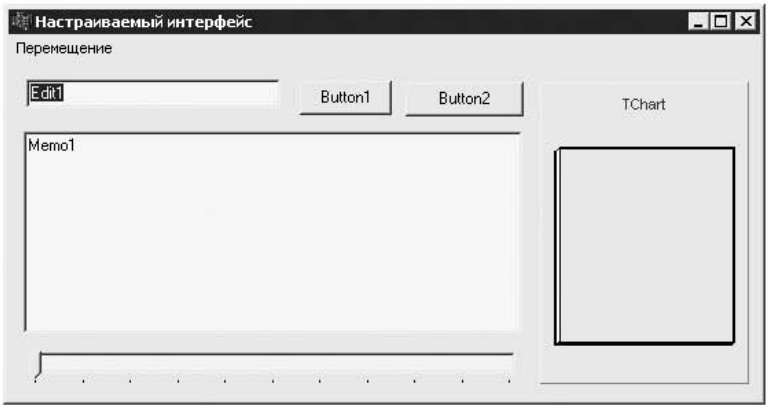


**Рис. 1.15.** Окончательный вид формы

В листинге 1.22 приведен пример реализации функции включения перемещения окна, заданного дескриптором hWnd (свойство Handle оконного компонента).

**Листинг 1.22.** Включение возможности перемещения компонента

```
void MakeMovable(HWND hWnd)
{
  //Разрешаем перемещение элемента управления
```

```
  DWORD style = ::GetWindowLong(hWnd, GWL_STYLE);
  style |= WS_OVERLAPPED | WS_THICKFRAME | WS_CAPTION;
  ::SetWindowLong(hWnd, GWL_STYLE, style);
  DWORD exstyle = ::GetWindowLong(hWnd, GWL_EXSTYLE);
  exstyle |= WS_EX_TOOLWINDOW;
  ::SetWindowLong(hWnd, GWL_EXSTYLE, exstyle);
  //Сделаем так, чтобы положение клиентской области
  //получившегося окна
  //совпадало с первоначальным положением элемента
  //управления.
  //Расчет размера окна
  TRect rc;
  ::GetWindowRect(hWnd, &rc);
  TPoint pt1(rc.left, rc.top), pt2(rc.right, rc.bottom);
  ::ScreenToClient(::GetParent(hWnd), &pt1);
  ::ScreenToClient(::GetParent(hWnd), &pt2);
  rc.left = pt1.x;
  rc.top = pt1.y;
  rc.right = pt2.x;
  rc.bottom = pt2.y;
  ::AdjustWindowRectEx(&rc, style, false, exstyle);
  //Перемещение окна
  DWORD flags = SWP_DRAWFRAME | SWP_NOZORDER;
  ::SetWindowPos(hWnd, NULL, rc.Left, rc.Top,
                 rc.Right - rc.Left, rc.Bottom - rc.Top,
flags);
}
```

Легко заметить, что в листинге 1.22 реализация основной идеи трюка занимает меньшую часть функции. Большая часть функции, как ни странно, отведена под вспомогательный код, рассчитывающий координаты места расположения получаемого окна с рамкой, которые должны быть такими, чтобы клиентская область этого окна находилась в точности там, где был расположен компонент. Таким образом удается избежать проблемы с ограничениями, возникающей при изменении размера компонента (окно с используемой в данном случае тонкой рамкой имеет минимальный предел ширины и высоты). Кроме того, при использовании такого метода реализации компонент во время перемещения выглядит точно так же, как и после выключения возможности перемещения.

Ниже в листинге 1.23 приведен текст реализации функции, позволяющей отключить возможность перемещения компонента. Она также принимает в качестве параметра дескриптор окна `hWnd`, перемещение которого следует завершить.

**Листинг 1.23.** Отключение возможности перемещения компонента

```
void MakeUnmovable(HWND hWnd)
{
```

```
  //Запомним положение клиентской области, чтобы потом
  //правильно
  //расположить элемент управления
  TRect rc;
  ::GetClientRect(hWnd, &rc);
  TPoint pt1(rc.left, rc.top), pt2(rc.right, rc.bottom);
  ::ClientToScreen(hWnd, &pt1);
  ::ScreenToClient(::GetParent(hWnd), &pt1);
  ::ClientToScreen(hWnd, &pt2);
  ::ScreenToClient(::GetParent(hWnd), &pt2);
  //Запрещаем перемещения элемента управления
  DWORD style = ::GetWindowLong(hWnd, GWL_STYLE);
  style = style & ~WS_OVERLAPPED & ~WS_THICKFRAME & ~WS_
CAPTION;
  ::SetWindowLong(hWnd, GWL_STYLE, style);
  style = ::GetWindowLong(hWnd, GWL_EXSTYLE);
  style = style & ~WS_EX_TOOLWINDOW;
  ::SetWindowLong(hWnd, GWL_EXSTYLE, style);
  //Перерисуем в новом состоянии
  DWORD flags = SWP_DRAWFRAME | SWP_NOZORDER;
  ::SetWindowPos(hWnd, NULL, pt1.x, pt1.y, pt2.x - pt1.x,
pt2.y - pt1.y,
                 flags);
}
```

В листинге 1.23 сначала рассчитывается будущее положение компонента на форме по расположению клиентской области окна с рамкой. Потом удаляются стили окна компонента, добавленные функцией `MakeMovable`, и компонент закрепляется на форме в заданном положении.

Включение и выключение возможности перемещения компонентов в приведенном примере происходит при выполнении обработчика выбора пунктов меню (листинг 1.24).

**Листинг 1.24.** Обработчики выбора пунктов меню

```
void __fastcall TForm1::mnuDisableDragClick(TObject *Sender)
{
  //Запрещаем перемещение компонентов
  AllowMove(false);
  mnuDisableDrag->Checked = true;
}
void __fastcall TForm1::mnuEnableDragClick(TObject *Sender)
{
  //Разрешаем перемещение компонентов
  AllowMove(true);
  mnuEnableDrag->Checked = true;
}
```

Текст реализации функции-члена класса `TForm1`, вызываемой в обработчиках выбора пунктов меню, приведен в листинге 1.25. Она в зависимости от значения аргумента `bAllow` разрешает или запрещает перемещение всех компонентов формы, используя рассмотренные ранее функции `MakeMovable` и `MakeUnmovable`.

**Листинг 1.25.** Включение/выключение возможности перемещения всех компонентов формы

```
void TForm1::AllowMove(bool bAllow)
{
  if ( bAllow )
  {
    //Разрешаем перемещение компонентов
    MakeMovable(Button1->Handle);
    MakeMovable(Button2->Handle);
    MakeMovable(Edit1->Handle);
    MakeMovable(Memo1->Handle);
    MakeMovable(Chart1->Handle);
    MakeMovable(TrackBar1->Handle);
  }
  else
  {
    //Запрещаем перемещение компонентов
    MakeUnmovable(Button1->Handle);
    MakeUnmovable(Button2->Handle);
    MakeUnmovable(Edit1->Handle);
    MakeUnmovable(Memo1->Handle);
    MakeUnmovable(Chart1->Handle);
    MakeUnmovable(TrackBar1->Handle);
  }
}
```

Возможность перемещения компонентов формы, описываемая в этом подразделе, будет совершенно бесполезной, если будет отсутствовать возможность сохранения заданного пользователем расположения компонентов. В данном примере сохранение и восстановление координат мест расположения компонентов формы, а также положения и размера самой формы реализовано. Для этого был написан небольшой класс `TLayoutManager`, текст объявления которого приведен в листинге 1.26.

**Листинг 1.26.** Объявление класса TLayoutManager

```
class TLayoutManager
{
  TForm *m_pForm; //Форма, за которую отвечает экземпляр
класса
public:
  TLayoutManager(TForm *pForm);
  void SaveLayout();
  void LoadLayout();
};
```

Экземпляр этого класса инициализируется указателем на форму. Данный класс способен только сохранять и восстанавливать позицию и размер формы, а также расположение компонентов на форме.

Текст реализации класса `TLayoutManager` приводится в листинге 1.27. Она не отличается особой сложностью. Единственное, что стоит помнить, это то, что данные сохраняются в файле с именем формата `<имя_приложения>_<имя_формы>`, например `Project1.exe_Form1`.

**Листинг 1.27.** Реализация класса TLayoutManager

```
TLayoutManager::TLayoutManager(TForm *pForm)
{
  m_pForm = pForm;
}
void TLayoutManager::SaveLayout()
{
  String strFileName = ParamStr(0) + "_" + m_pForm->Name;
  TFileStream *ostr = new TFileStream( strFileName, fmCreate
);
  TRect rc;
  //Сохранение положения и размера формы
  rc.left = m_pForm->Left;
  rc.top = m_pForm->Top;
  rc.right = rc.left + m_pForm->Width;
  rc.Bottom = rc.top + m_pForm->Height;
  ostr->Write(&rc, sizeof(rc));
  //Сохранение расположения компонентов
  for ( int i=0; i<m_pForm->ControlCount; i++ )
  {
    TControl *ctrl = m_pForm->Controls[i];
    rc.left = ctrl->Left;
    rc.top = ctrl->Top;
    rc.right = rc.left + ctrl->Width;
    rc.Bottom = rc.top + ctrl->Height;
    ostr->Write(&rc, sizeof(rc));
  }
  delete ostr;
}
void TLayoutManager::LoadLayout()
{
  try
  {
    String strFileName = ParamStr(0) + "_" + m_pForm->Name;
    TFileStream *istr = new TFileStream(strFileName, fmOpen-
Read);
    TRect rc;
```

```
    //Сохранение положения и размера формы
    istr->Read(&rc, sizeof(rc));
    m_pForm->Left = rc.left;
    m_pForm->Top = rc.top;
    m_pForm->Width = rc.Width();
    m_pForm->Height = rc.Height();
    //Загрузка расположения компонентов
    for ( int i=0;
          i<m_pForm->ControlCount && istr->Position < istr-
>Size;
          i++ )
   {
      istr->Read(&rc, sizeof(rc));
      TControl *ctrl = m_pForm->Controls[i];
      ctrl->Left = rc.left;
      ctrl->Top = rc.top;
      ctrl->Width = rc.Width();
      ctrl->Height = rc.Height();
    }
    delete istr;
  }
  catch (Exception &exception)
  {
    //Исключение может генерироваться при самом первом вызове,
    //когда файл
    //с позициями компонентов еще не создан
  }
}
```

Чтобы правильно использовать класс `TLayoutManager` в приложении, достаточно внести небольшие изменения в класс формы: объявить переменную-член класса типа `TLayoutManager`, инициализировать эту переменную указателем на форму и вызвать методы `LoadLayout` и `SaveLayout` при создании и уничтожении формы. Пример этого приведен в листинге 1.28.

**Листинг 1.28.** Внедрение функции сохранения/загрузки расположения компонентов формы

```
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner), m_LayoutManager(this)
{
}
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  m_LayoutManager.LoadLayout();
}
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
```

```
  if ( mnuEnableDrag->Checked )
  {
    //Закончим перемещение компонентов, чтобы правильно
    //сохранить их
    //расположение
    AllowMove(false);
  }
  m_LayoutManager.SaveLayout();
}
```

В завершение хочется лишь заметить, что приведенный способ реализации является лишь одним из способов реализации настраиваемого интерфейса, да и сохранение таких данных, как расположение компонентов формы, можно выполнить многими способами. Еще один способ хранения настроек программы, который здесь можно применить с не меньшим успехом, рассмотрен в гл. 8.

# Окна других приложений

Интересной особенностью операционной системы Windows является то, что окна одного приложения практически не защищены от доступа из других приложений: можно достаточно легко получить дескриптор любого окна, послать окну сообщения, перехватить сообщения, приходящие окну. В данном разделе на фоне общей темы «Окна» будут рассмотрены самые простые операции, такие как получение дескрипторов окон, а также некоторые манипуляции с окнами других приложений, работающих в системе.

## Скрытие Панели задач

В предыдущем примере воздействие на окно другого приложения не осуществлялось явным образом. Теперь будут рассмотрены простейшие функции манипулирования окнами, а именно: скрытие и отображение Панели задач Windows (полоска, расположенная по умолчанию в нижней правой части экрана) с помощью функции скрытия/отображения окна.

Окно Панели задач принадлежит оконному классу `Shell_TrayWnd` и содержит пустой текст заголовка. Для получения дескриптора этого окна можно воспользоваться API-функцией `FindWindow`:

```
HWND FindWindow(const char *lpClassName, const char
*lpWindowName);
```

Первый аргумент функции `FindWindow` — имя оконного класса, второй — текст заголовка окна. Данная функция возвращает дескриптор окна или значение `NULL`, если не удалось найти окно с заданными характеристиками.

Так, скрытие Панели задач можно реализовать следующим образом (листинг 1.29).

**Листинг 1.29.** Скрытие Панели задач

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  HWND hTaskBar = ::FindWindow("Shell_TrayWnd", "");
  ::ShowWindow(hTaskBar, SW_HIDE);
}
```

И наоборот, отобразить предварительно скрытую Панель задач можно так, как показано в листинге 1.30.

**Листинг 1.30.** Отображение Панели задач

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  HWND hTaskBar = ::FindWindow("Shell_TrayWnd", "");
  ::ShowWindow(hTaskBar, SW_SHOW);
}
```

## Составление списка окон

В завершение обратите внимание на чуть более сложный пример. Приведенное ниже приложение составляет список, или, если хотите, дерево, всех окон, созданных в сессии Windows. Внешний вид приложения после составления перечня окон показан на рис. 1.16. Структура перечня окон, которую можно наблюдать на рисунке, отражает связь «родительское–дочернее окно».

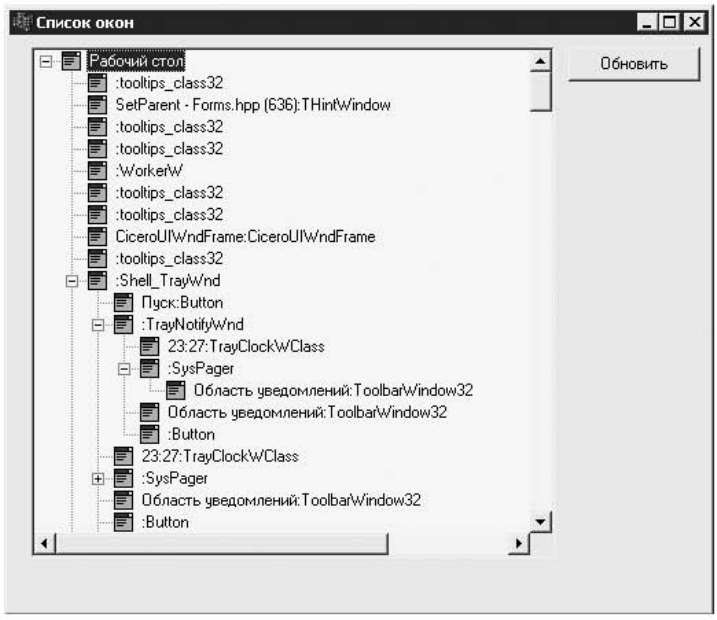


**Рис. 1.16.** Пример списка окон

Реализовать данное приложение довольно просто. В приведенном случае для представления информации о найденных окнах используется компонент TTreeView, имеющий имя tvwWinTree. Для занесения в дерево информации об окнах (а именно: имя класса окна и текст заголовка окна) написан небольшой обработчик кнопки Обновить и одна вспомогательная функция.

Для составления полного перечня окон нужно определить все окна верхнего уровня, затем для каждого окна верхнего уровня определить все дочерние окна, далее все дочерние окна дочерних окон и т. д. Для составления списка окон верхнего уровня используется API-функция EnumWindow, которая вызывается нажатием кнопки Обновить (листинг 1.31).

**Листинг 1.31.** Запуск функции составления списка окон

```
void __fastcall TForm1::cmbRefreshClick(TObject *Sender)
{
  tvwWinTree->Items->Clear();
  //Составление списка открытых окон
  //..не забудем Рабочий стол
  TTreeNode *pDesktop = tvwWinTree->Items->AddChild(NULL,
"Рабочий стол");
  pDesktop->Data = NULL;
  //..запускаем перечисление окон верхнего уровня
  ::EnumWindows((WNDENUMPROC)EnumWindowsFunc,
(LPARAM)pDesktop);
}
```

Функция перечисления окон EnumWindows в качестве первого аргумента принимает указатель на пользовательскую функцию, вызываемую при нахождении каждого окна. Второй аргумент функции EnumWindows — 32-битное значение, передаваемое в пользовательскую функцию вместе с дескриптором найденного окна. В данном случае функцией обратного вызова для EnumWindows является EnumWindowsProc, приведенная в листинге 1.32. Значением же, дополнительно передаваемым в EnumWindowsProc, будет указатель на элемент дерева, соответствующий окну, для которого перечисляются дочерние окна. Первоначально при определении окон верхнего уровня они представляются в дереве дочерними по отношению к Рабочему столу.

**Листинг 1.32.** Функция обратного вызова для перечисления окон

```
BOOL __stdcall EnumWindowsFunc(HWND hwnd, LPARAM lParam)
{
  //Добавляем найденное окно в дерево. Для большей
  //выразительности определим заголовок окна и имя класса
  //окна
  static TCHAR szCaption[256];
  ::GetWindowText(hwnd, szCaption, 256);
  static TCHAR szClassName[256];
```

```
  ::GetClassName(hwnd, szClassName, 256);
  static String strFullName;
  strFullName = szCaption;
  strFullName += ":";
  strFullName += szClassName;
  TTreeNode *pNode =
   Form1->tvwWinTree->Items->AddChild((TTreeNode*)lParam,
strFullName);
  pNode->Data = hwnd;
  pNode->ImageIndex = 0;
  //Перечислим дочерние окна найденного окна
  ::EnumChildWindows(hwnd, (WNDENUMPROC)EnumWindowsFunc,
(LPARAM)pNode);
  //Продолжим перечисление окон текущего уровня
  return true;
}
```

В приведенной в листинге 1.32 функции `EnumWindowsProc` сначала определяется текст заголовка окна (для элементов управления вроде текстового поля это может быть текст, введенный в данное поле), затем — имя оконного класса, и после этого элемент добавляется в дерево (компонент TTreeView на форме). Далее запускается перечисление дочерних окон найденного окна, для чего используется уже другая API-функция — `EnumChildWindows`. Она очень похожа на рассмотренную выше функцию `EnumWindows`, однако принимает уже три аргумента, первым из которых является дескриптор окна, для которого необходимо перечислить дочерние окна, вторым — указатель на такую же функцию обратного вызова, а третьим — пользовательское значение, передаваемое в функцию обратного вызова.

В рассмотренном примере составляется полный список окон, поэтому приведенная в листинге 1.32 функция `EnumWindowsProc` всегда возвращает `true`, что для вызывающей ее системной функции означает необходимость продолжения процесса перечисления окон. Чтобы прервать перечисление, достаточно в функции обратного вызова вернуть значение `false`.

# Глава 2

## Графика

- Рисование на форме
- Простой графический редактор
- Преобразования изображений

Вероятно, первое, что приходит на ум при упоминании операционной системы Windows, так это окна, цветные значки и т. д. Если говорить немного строже, то вспоминается графический интерфейс, который неотделим от этой операционной системы. В саму Windows встроен очень приличный набор функций, отвечающих за графический вывод и образующих подсистему GDI (graphics device interface — интерфейс графических устройств).

Множество функций работы с различными графическими устройствами (видеокартами, принтерами и т. д.) Windows GDI в большинстве случаев скрывает от обычного программиста. Тема эта очень обширна, и потому в представленной здесь теоретической части не будут рассматриваться особенности вывода графической информации при печати.

Наверное, главной заслугой оболочки GDI является то, что при использовании функций рисования не имеет особого значения, на какое конкретно устройство выводится изображение. Вместо этого приходится иметь дело с неким контекстом устройства (device context в терминологии Microsoft). Важной особенностью контекста графического устройства является то, что он предполагает наличие некой поверхности — набора точек или пикселов, — на которой можно рисовать. Возможно, поэтому в библиотеке Borland C++ Builder контекст устройства для вывода (бывают еще и информационные контексты устройства) именуется канвой (Canvas, класс `TCanvas`), то есть поверхностью для рисования.

С контекстом устройства, или канвой, связывается ряд объектов. Вот те из них, которые используются в приведенных в данной главе примерах:

- карандаш (`TPen`) — задает толщину, цвет и способ рисования линий (сплошные, прерывистые и т. д.);
- кисть (`TBrush`) — указывает цвет и способ заливки областей;
- шрифт (`TFont`) — определяет шрифт, используемый для вывода текста;
- область отсечения (`clipping region`) — регион, ограничивающий область, которую можно изменять с использованием функций рисования;
- битовая матрица (`TBitmap`) — массив байт (и структура с параметрами), хранящая растровое изображение, формируемое графическими функциями.

Первые два перечисленных объекта очень просты в использовании, особенно если использовать классы-оболочки `TPen` и `TBrush`. Шрифты настраиваются чуть сложнее (подробнее об этом вы узнаете в подразд. «Вывод текста» разд. «Рисование на форме»). Регионы рассмотрены в гл. 1, а вот о битовых матрицах сейчас поговорим более подробно.

В битовой матрице (речь идет об объекте класса `TBitmap`) хранятся данные растрового изображения. Вас не должно беспокоить, что именно и как хранится в битовой матрице. Главное, что в арсенале класса `TBitmap` предусмотрены свойства, позволяющие получать и менять параметры изображения. Какие именно, вы увидите при

изучении разд. «Простой графический редактор» и разд. «Преобразования изображений».

Так, в классе `TBitmap` реализованы методы загрузки из файла и сохранения в файл, и, самое главное, класс `TCanvas` поддерживает возможность вывода непосредственно битовой матрицы, что открывает большие возможности, например, по предварительной подготовке изображений перед выводом на экран. Кстати, с объектами `TBitmap` также ассоциируются объекты `TCanvas`, при этом все манипуляции с ассоциированной канвой отражаются в `TBitmap`. Об этом и о предварительной подготовке изображений необходимо поговорить чуть подробнее.

Если вы уже имеете опыт разработки графических приложений в MS-DOS или подобной системе, то вы наверняка помните, как хорошо быть уверенным, что нарисованное программой изображение не нужно будет перерисовывать, за исключением случаев, связанных с изменениями, вызванными этой же программой. Иными словами, в MS-DOS одна и только одна программа может использовать для вывода экран. В среде Windows с графическим интерфейсом гораздо сложнее: при выводе один и тот же экран разделяет множество программ, каждая из которых, грубо говоря, выводит свои данные в своем окне. Окна могут перекрываться друг другом, и, как следствие, одно и то же содержимое приложению приходится выводить в окно множество раз. Так, если при написании приложений использовать формы, можно увидеть, что при необходимости перерисовки формы генерируется событие `OnPaint`. Обработчик этого события по умолчанию как раз и занимается рисованием содержимого формы, рисуя фон самой формы и «спуская» событие `OnPaint` ее дочерним компонентам.

Если опять же вспомнить MS-DOS, то там избежать мерцания при рисовании сложных изображений можно было, применив такой прием: изображение можно было создать на другой «странице» видеопамяти, после чего показывать сформированное изображение на экране переключением видеоадаптера с отображения первой страницы на отображение второй страницы (и наоборот). Довольно сложно, но эффективно, не правда ли? В Windows похожая функция обеспечивается возможностью создания так называемого контекста устройства в памяти, который выводит изображение не на экран, а на заданную битовую матрицу. После того как изображение будет сформировано, содержимое битовой матрицы рисуется на экране. Кстати, как раз контекстом устройства в памяти и является класс `TCanvas`, ассоциированный с `TBitmap`. Сам же прием предварительной подготовки изображения используется в рассмотренном ниже графическом редакторе.

И еще несколько вводных слов — на этот раз насчет способов кодирования цвета, применяемых в компьютерной графике. Их тоже довольно много, но знать на этом этапе потребуется только один — кодировку RGB. Суть ее состоит в следующем. Цвет точки определяется интенсивностью (яркостью) трех составляющих — красной (R, red), зеленой (G, green) и синей (B, blue). При максимальных значениях интенсивностей всех трех каналов образуется белый цвет, при минимальных — черный. На каждую составляющую выделяется определенное количество бит. Так,

например, если на каждую составляющую выделяется 8 бит, то для хранения цвета одной точки потребуется 3 байт и, соответственно, будет доступно $2^{24}$ вариаций цвета. Само изображение при этом будет называться 24-разрядным. При меньшем количестве цветов применяются другие приемы кодирования (другие палитры), но в приводимых здесь примерах используется именно 24-разрядная RGB-кодировка.

# Рисование на форме

Настало время применить встроенные графические функции Windows на практике. Начать стоит с самых простых примеров и постепенно переходить к более сложным и интересным.

## Рисование графических примитивов

Для вывода на экран основных графических примитивов, поддерживаемых функциями GDI, можно применить небольшую программу, код которой приведен в листинге 2.1.

**Листинг 2.1.** Вывод на форму графических примитивов

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  //Настройка карандаша
  Canvas->Pen->Width = 3;  //Сплошной толщиной 3 пиксела
  Canvas->Pen->Color = (TColor)RGB(0, 255, 0); //Зеленый
  //Настройка кисти
  Canvas->Brush->Style = bsFDiagonal; //Диагональная слева
                                      //направо
  Canvas->Brush->Color = (TColor)RGB(0, 0, 255); //Синяя
  //Рисование
  //..линия
  Canvas->MoveTo(10, 10);
  Canvas->LineTo(100, 80);
  //..прямоугольник
  Canvas->Rectangle(110, 10, 200, 80);
  //..скругленный прямоугольник
  Canvas->RoundRect(210, 10, 300, 80, 15, 15);
  //..эллипс
  Canvas->Ellipse(310, 10, 400, 80);
  //..дуга эллипса
  Canvas->Arc(10, 100, 100, 170, 10, 100, 80, 200);
  //..сегмент эллипса
  Canvas->Chord(110, 100, 200, 170, 110, 100, 110, 200);
  //..сектор эллипса
  Canvas->Pie(210, 100, 300, 170, 210, 100, 210, 180);
```

```
  //Теперь дуга, сегмент и сектор, нарисованные по часовой
  //стрелке
  ::SetArcDirection(Canvas->Handle, AD_CLOCKWISE);
  Canvas->Arc(10, 200, 100, 270, 10, 200, 80, 300);
  Canvas->Chord(110, 200, 200, 270, 110, 200, 110, 300);
  Canvas->Pie(210, 200, 300, 270, 210, 200, 210, 280);
  TPoint pts[6];
  pts[0] = TPoint(10, 310);
  pts[1] = TPoint(100, 310);
  pts[2] = TPoint(60, 345);
  pts[3] = TPoint(100, 380);
  pts[4] = TPoint(10, 380);
  pts[5] = TPoint(10, 310);
  //Замкнутая полилиния
  Canvas->Polyline(pts, 5);
  //Закрашенный многоугольник (полигон)
  for ( int i=0; i<6; i++ ) pts[i].x+= 100;
  Canvas->Polygon(pts, 5);
}
```

Как можно видеть, в листинге 2.1 приведен обработчик события `OnPaint`. В результате на форме будут отображаться 12 фигур, что и показано на рис. 2.1.

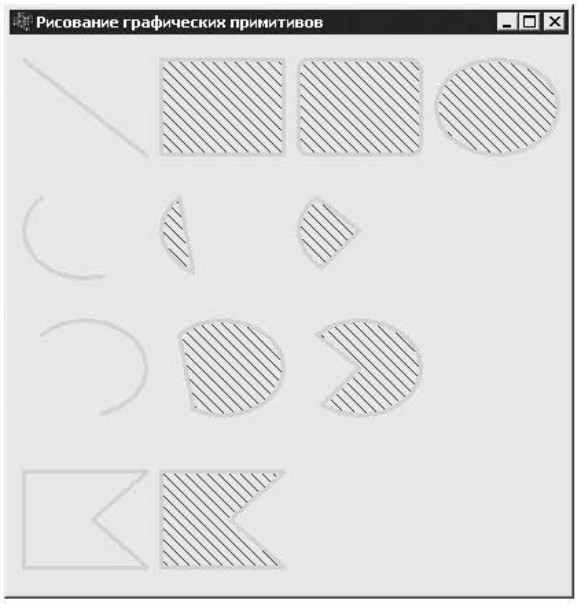


**Рис. 2.1.** Вывод графических примитивов на форму

Из этого же примера можно видеть, как изменяются параметры кисти и карандаша, применяемых для вывода графических объектов.

## Цветовая палитра

В этом гораздо более зрелищном примере также нет ничего сложного. С помощью нехитрых манипуляций красной, зеленой и синей составляющими цвета можно построить довольно симпатичную картину. Код программы, выводящей на форму цветовую палитру, приведен в листинге 2.2.

**Листинг 2.2.** Вывод на форму цветовой палитры

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Refresh();
  //Рисуем прямоугольную палитру
  int xmin = 20, ymin = 70;
  int xmax = Width - 30, ymax = Height - 70;
  int width = xmax - xmin;
  int height = ymax - ymin;
  for ( int y=ymin; y<ymax; y++ )
  {
    for ( int x=xmin; x<xmax; x++ )
    {
      int r = 255 * (x-xmin) / width;
      int g = 255 * (y-ymin) / height;
      int b = 255 - 255 * (x-xmin + y-ymin) / (width +
height);
      Canvas->Pixels[x][y] = (TColor)RGB(r,g,b);
    }
  }
}
```

Вообще подходов к выводу на экран различного рода градаций цветов существует множество. Так, в приведенном в листинге 2.2 примере строится достаточно красивая, но неполная палитра цветов в том смысле, что на ней отсутствуют «чистый» красный и зеленый цвета. Более полная в этом смысле палитра будет построена в конце раздела после рассмотрения градиентов.

## Градации цветов

Достаточно интересную и к тому же полезную картину можно получить и с помощью менее хитрых операций над цветовыми составляющими. Так, по телевизору иногда можно увидеть настроечное изображение вроде показанного на рис. 2.2.

Реализуется показанное на рис. 2.2 очень просто (листинг 2.3). Для обеспечения плавного перехода от черного цвета к одному из трех основных цветов палитры достаточно плавно увеличивать интенсивность соответствующего канала (в данном случае от 0 до 255). Если интенсивность красного, зеленого и синего цветов одинакова, получается серый цвет, а потому строить градации серого ничуть не сложнее, чем градации какого-либо основного цвета.

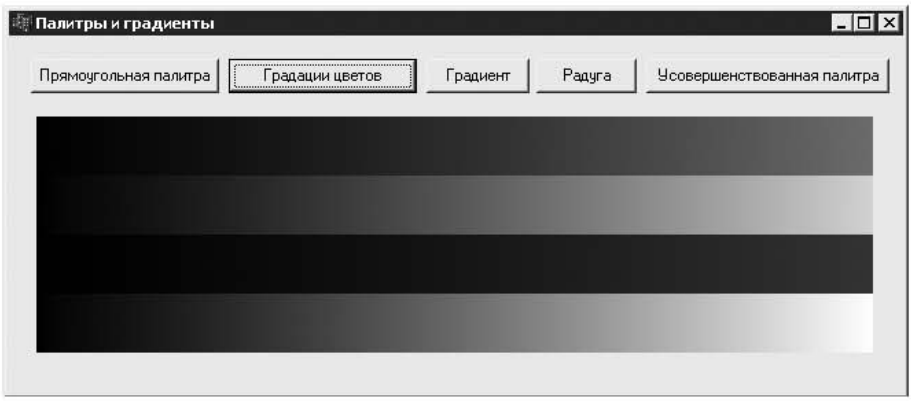


**Рис. 2.2.** Градации цветов и оттенки серого

**Листинг 2.3.** Градации красного, зеленого, синего и серого цветов

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  Refresh();
  //Создает переходы от черного к красному, зеленому и синему
  //цветам, а также градации серого
  int xmin = 20, ymin = 70;
  int xmax = 300, ymax = 280;
  int width = xmax - xmin;
  int height = (ymax-ymin) / 4;
  for ( int x=xmin; x<xmax; x++ )
  {
    int color = 255 * (x-xmin) / width;
    //Красный
    Canvas->Pen->Color = (TColor)RGB(color,0,0);
    Canvas->MoveTo(x, ymin);
    Canvas->LineTo(x, ymin + height);
    //Зеленый
    Canvas->Pen->Color = (TColor)RGB(0,color,0);
    Canvas->MoveTo(x, ymin + height);
    Canvas->LineTo(x, ymin + 2*height);
    //Синий
    Canvas->Pen->Color = (TColor)RGB(0,0,color);
    Canvas->MoveTo(x, ymin + 2*height);
    Canvas->LineTo(x, ymin + 3*height);
    //Серый
    Canvas->Pen->Color = (TColor)RGB(color,color,color);
    Canvas->MoveTo(x, ymin + 3*height);
    Canvas->LineTo(x, ymin + 4*height);
  }
}
```

Правда, на телевидении часто применяются более сложные изображения с бо́льшим количеством цветов. Это чуть более сложный эффект, называемый градиентом, который рассмотрен ниже.

Теперь обратите еще раз внимание на рис. 2.2. Не кажется ли вам странным, что на черно-белом рисунке полосы синего, зеленого и красного цветов не слились в одну широкую серую полосу? Ведь расположенные на одной вертикальной линии пикселы в этих полосах имеют одинаковое значение интенсивности, но только разных цветовых каналов. Дело в том, что человеческий глаз по-разному восприимчив к различным цветам: лучше всего воспринимается зеленый цвет, хуже всего — синий. Графический редактор, при помощи которого подготовлены иллюстрации к данной книге, учитывает этот факт. Как выполнить правильное преобразование цветного изображения в черно-белое, будет продемонстрировано далее в этой главе.

## Градиент и «радуга»

Реализовать следующий рассматриваемый эффект не многим сложнее, чем построить градации цветов. В принципе, предыдущий пример является частным случаем построения градиента. В листинге 2.4 показано, как построить плавный переход от одного любого цвета к любому другому. Правда, в листинге исходный и конечный цвета заданы жестко, но при необходимости это можно легко исправить.

**Листинг 2.4.** Построение градиента

```
void __fastcall TForm1::Button5Click(TObject *Sender)
{
  BYTE r1 = 0, g1 = 255, b1 = 0;
  BYTE r2 = 255, g2 = 0, b2 = 255;
  BYTE r, g, b;
  int xmin = 20, ymin = 70;
  int xmax = Width - 30, ymax = Height - 70;
  int width = xmax - xmin;
  //Рисуем плавный переход от (r1, g1, b1) к (r2, g2, b2)
  for ( int x=xmin; x<xmax; x++ )
  {
    r = r1 + (r2 - r1)*(x - xmin)/width;
    g = g1 + (g2 - g1)*(x - xmin)/width;
    b = b1 + (b2 - b1)*(x - xmin)/width;
    Canvas->Pen->Color = (TColor)RGB(r,g,b);
    Canvas->MoveTo(x, ymin);
    Canvas->LineTo(x, ymax);
  }
}
```

Построенная программой область плавной смены цветов хорошо видна на рис. 2.3, несмотря на то что рисунок черно-белый.


**Рис. 2.3.** Градиент

На основе градиента можно также создать более интересные эффекты. К примеру, можно с легкостью изобразить на форме радугу — для этого достаточно нарисовать переходы красный-оранжевый-желтый-зеленый-голубой-синий-фиолетовый. Именно это и делает программа, код которой приведен в листинге 2.5.

**Листинг 2.5.** «Радуга» на форме

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
  //Цвета «радуги»
  BYTE colors[][3] = {
    {255, 0, 0},     //Красный
    {255, 128, 0},   //Оранжевый
    {255, 255, 0},   //Желтый
    {0, 255, 0},     //Зеленый
    {0, 255, 255},   //Голубой
    {0, 0, 255},     //Синий
    {255, 0, 255}    //Фиолетовый
  };
  int xmin = 20, ymin = 70;
  int xmax = Width - 30, ymax = Height - 70;
  int ccount = sizeof(colors) / (sizeof(BYTE) * 3);
  int width = (xmax - xmin) / (ccount - 1); //Ширина перехода
                                            //между
                                            //двумя цветами
  //Создание переходов между заданными цветами
  for ( int c=0; c<ccount-1; c++ )
  {
    BYTE r1 = colors[c][0], g1 = colors[c][1], b1 =
colors[c][2];
```

```
    BYTE r2 = colors[c+1][0], g2 = colors[c+1][1], b2 =
colors[c+1][2];
    BYTE r, g, b;
    int xstart = xmin + c*width, xend = xmin + (c+1)*width;
    for ( int x=xstart; x<xend; x++ )
    {
      r = r1 + (r2 - r1)*(x - xstart)/width;
      g = g1 + (g2 - g1)*(x - xstart)/width;
      b = b1 + (b2 - b1)*(x - xstart)/width;
      Canvas->Pen->Color = (TColor)RGB(r,g,b);
      Canvas->MoveTo(x, ymin);
      Canvas->LineTo(x, ymax);
    }
  }
}
```

Переходы между цветами «радуги», создаваемой приведенной программой, хорошо видны на рис. 2.4. Сама программа написана так, чтобы с ее помощью можно было нарисовать любое количество переходов между любыми цветами.

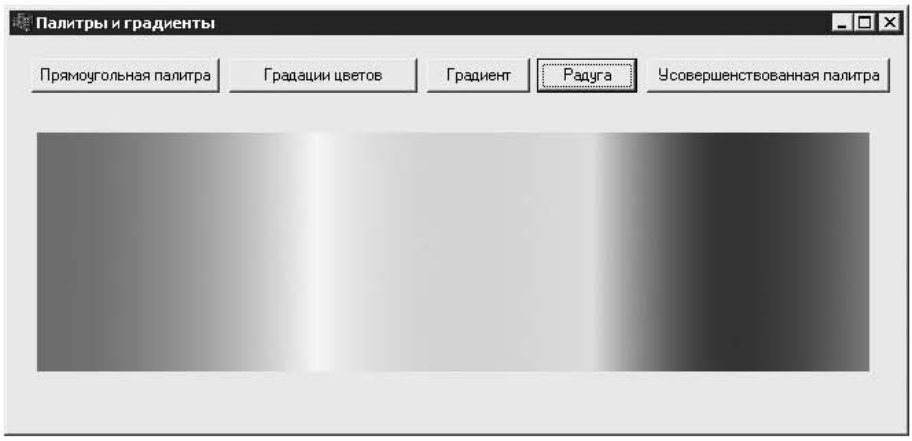


**Рис. 2.4.** «Радуга»

Если вы обратили внимание, в листинге 2.5 цвета «радуги» задаются не цветовыми константами типа `TColor` (`clRed`, `clGreen`, например), а как двумерные массивы байт. Во-первых, зачем упаковывать составляющие цвета в `TColor`, когда все равно понадобится извлекать их для вычисления интенсивности каждого из трех каналов по отдельности. Во-вторых, константы оранжевого цвета найти не удалось, а потому задание цветов именно таким образом позволяет выполнить все единообразно.

Наконец, в-третьих, в данном случае становится видна уловка, на которую приходится идти при построении «радуги» естественным способом. Очевидно, что добавление оранжевого цвета нарушает закономерность, при которой при каждом

переходе (градиенте) то увеличивается до предела интенсивность одного канала, то уменьшается интенсивность другого. Речь идет не о том, что оранжевому цвету в радуге не место, а лишь о том, что сделать «радугу» менее естественной, но более красивой без добавления оранжевого можно, задав цвета, например, следующим образом:

```
BYTE colors[][3] = {
  {255, 0, 255},  //Фиолетовый
  {255, 0, 0},    //Красный
  {255, 255, 0},  //Желтый
  {0, 255, 0},    //Зеленый
  {0, 255, 255},  //Голубой
  {0, 0, 255},    //Синий
  {255, 0, 255}   //Фиолетовый
};
```

В этом случае на рисунке будут хорошо видны широкие полосы, соответствующие оттенкам красного, зеленого и синего цветов.

## Усовершенствованная палитра

Наконец, еще более интересный эффект можно получить, если к только что рассмотренному эффекту «радуга» добавить изменение яркости пиксела в соответствии с изменением вертикальной координаты. Если использовать последовательность цветов, приведенную в конце предыдущего подраздела, то получается симметричное изображение, похожее на показанное на рис. 2.5 (только цветное, естественно).

Получается довольно удобная палитра, не правда ли? В листинге 2.6 приведена программа построения показанной на рис. 2.5 палитры.

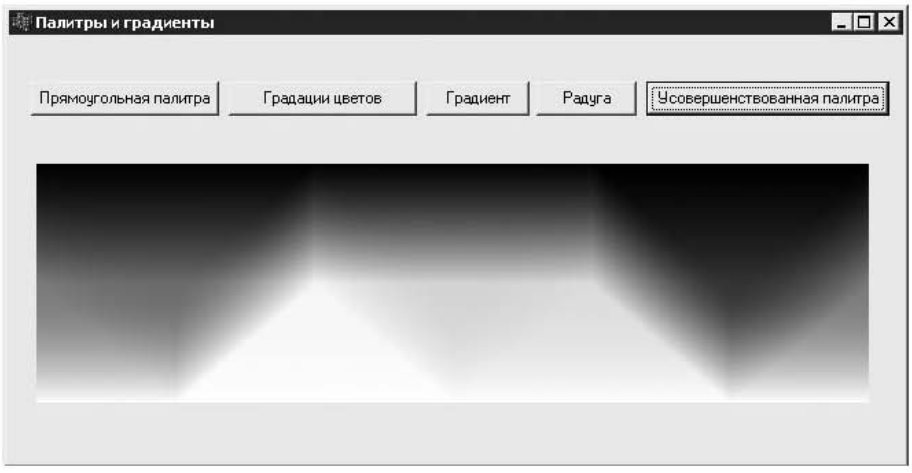


**Рис. 2.5.** Усовершенствованная палитра

**Листинг 2.6.** Усовершенствованная палитра

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  BYTE colors[][3] = {
    {255, 0, 255},  //Фиолетовый
    {255, 0, 0},    //Красный
    {255, 255, 0},  //Желтый
    {0, 255, 0},    //Зеленый
    {0, 255, 255},  //Голубой
    {0, 0, 255},    //Синий
    {255, 0, 255}   //Фиолетовый
  };
  int xmin = 20, ymin = 70;
  int xmax = Width - 30, ymax = Height - 70;
  int ccount = sizeof(colors) / (sizeof(BYTE) * 3);
  int height = (ymax - ymin);
  int width = (xmax - xmin) / (ccount - 1);  //Ширина
                                             //перехода между
                                             //двумя цветами
  for ( int y=ymin; y<ymax; y++ )
  {
    int k = 255 * (y - ymin - height/2) / (height/2);
    //создание переходов между заданными цветами
    for ( int c=0; c<ccount-1; c++ )
    {
      BYTE r1 = colors[c][0], g1 = colors[c][1], b1 =
colors[c][2];
      BYTE r2 = colors[c+1][0], g2 = colors[c+1][1], b2 =
colors[c+1][2];
      BYTE r, g, b;
      int xstart = xmin + c*width, xend = xmin + (c+1)*width;
      for ( int x=xstart; x<xend; x++ )
      {
        r = r1 + (r2 - r1)*(x - xstart)/width;
        r = (k+r)>255 ? 255 : ( (k+r)<0 ? 0 : k+r ); //Изменение
                                                     //яркости
        g = g1 + (g2 - g1)*(x - xstart)/width;
        g = (k+g)>255 ? 255 : ( (k+g)<0 ? 0 : k+g ); //Изменение
                                                     //яркости
        b = b1 + (b2 - b1)*(x - xstart)/width;
        b = (k+b)>255 ? 255 : ( (k+b)<0 ? 0 : k+b ); //Изменение
                                                     //яркости
        Canvas->Pixels[x][y] = (TColor)RGB(r,g,b);
      }
    }
  }
}
```

Как можно понять из листинга 2.6, построение рассматриваемой палитры отличается от построения «радуги» наличием дополнительного цикла, выполненного по вертикальной координате. При каждой итерации внешнего цикла рассчитывается изменение яркости (k) пиксела в зависимости от его вертикальной координаты. В приведенной программе k изменяется от –255 до +255. Далее перед самым выводом каждого пиксела на экран интенсивность каждого канала изменяется на k, но так, чтобы не произошло переполнение (значение интенсивности не получилось выше 255 или меньше нуля).

## Вывод текста

Windows GDI обладает также возможностями и по выводу текста. Текст может быть выведен на экран разными шрифтами, под разными углами, с различными настройками вида и цвета фона и цвета текста.

Здесь, однако, будут затронуты лишь некоторые базовые особенности вывода текста, а именно: варианты вывода текста, продемонстрированные на рис. 2.6.

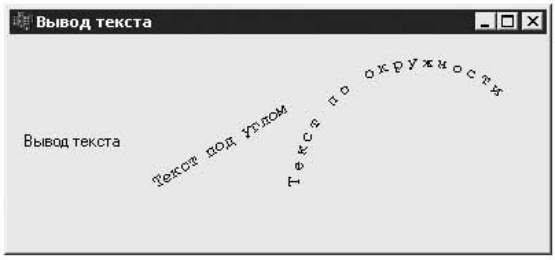


**Рис. 2.6.** Варианты вывода текста

Просматривая листинг 2.7, можно увидеть, насколько просто вывести текст с настройками шрифта, заданными в окне свойств формы, — для этого достаточно вызвать метод TextOut объекта Canvas, ассоциированный с формой.

Далее в листинге 2.7 текст выводится под углом 30° к горизонтальной оси, для чего используется функция DrawTextWithAngle (что при этом происходит, рассказано ниже).

**Листинг 2.7.** Различные варианты вывода текста

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  //Простая печать текста
  Canvas->TextOut(10, 70, "Вывод текста");
  //Вывод текста под углом 30° к горизонтальной оси
  DrawTextWithtAngle(Canvas, 100, 100, 300, "Текст под углом"
);
  //Вывод текста по дуге окружности (буква на каждые 7°
  //окружности)
```

```
  String text = "Текст по окружности";
  int r = 100;
  int x0 = 300, y0 = 110;
  char ch[2] = {0};
  for ( int i=0; i<text.Length(); i++ )
  {
    int angle = 180 - 7 * i;
    double angle_rad = PI * angle / 180;
    int x = x0 + r*cos(angle_rad);
    int y = y0 - r*sin(angle_rad);
    ch[0] = text[i+1];
    DrawTextWithtAngle(Canvas, x, y, (angle - 90) * 10, ch);
  }
}
```

И, наконец, в конце листинга 2.7 реализован вывод текста по окружности радиуса 100 пикселов. Причем буквы текста выводятся через каждые 7° окружности. Основной сложностью при таком выводе текста является подсчет, в какой точке и под каким углом должна быть выведена очередная буква. Вывод же текста осуществляется при помощи той же функции `DrawTextWithAngle`, текст реализации которой приведен в листинге 2.8.

**Листинг 2.8.** Вывод текста под заданным углом

```
void DrawTextWithtAngle(TCanvas *Canvas, int x, int y, int
angle,
                        const char *text)
{
  //Сначала нужно создать соответствующий шрифт
  TLogFont font = {0};
  font.lfHeight = 14;
  font.lfWeight = FW_REGULAR;
  font.lfEscapement = angle;
  font.lfOrientation = angle;
  strcpy(font.lfFaceName, "Courier New");
  Canvas->Font->Handle = ::CreateFontIndirect(&font);
  //Теперь можно вывести текст
  Canvas->TextOut(x, y, text);
}
```

Думается, назначение аргументов функции `DrawTextWithtAngle` понятно и не нуждается в пояснении. Стоит только лишь уточнить, что угол, под которым выводится текст, передается в градусах, причем значение угла должно быть умножено на 10, а это значит, что максимальной точностью задания угла вывода текста является 0,1°.

Теперь более подробно о том, как же вывести текст под углом. Для этого придется создать специальный шрифт. Ничего особенно сложного в этом нет. Для создания

нужного шрифта достаточно заполнить структуру TLogFont, передать ее в API-функцию создания шрифта — и можно пользоваться созданным шрифтом. Структура TLogFont выглядит приблизительно следующим образом:

```
struct TLogFont
{
    LONG       lfHeight;
    LONG       lfWidth;
    LONG       lfEscapement;
    LONG       lfOrientation;
    LONG       lfWeight;
    BYTE       lfItalic;
    BYTE       lfUnderline;
    BYTE       lfStrikeOut;
    BYTE       lfCharSet;
    BYTE       lfOutPrecision;
    BYTE       lfClipPrecision;
    BYTE       lfQuality;
    BYTE       lfPitchAndFamily;
    CHAR       lfFaceName[LF_FACESIZE];
};
```

В листинге 2.8 использовались лишь 5 из 14 полей структуры TLogFont. Ниже перечислены назначения всех полей этой структуры.

- lfHeight — задает высоту создаваемого шрифта в логических единицах.
- lfWidth — задает среднюю ширину символов шрифта.
- lfEscapement — задает угол между направляющим вектором (вдоль которого выводится текст) и горизонтальной осью. Значения поля указываются в десятых долях градуса. Поле может иметь значение, отличающееся от значения поля lfOrientation в операционных системах Windows NT/2000 и выше в графическом режиме GM_ADVANCED.
- lfOrientation — определяет угол, под которым к горизонтальной оси выводится каждый символ текста. Значение также указывается в десятых долях градуса.
- lfWeight — задает толщину линий шрифта; значения указываются в пределах от 0 до 1000. У поля предусмотрено также несколько констант для задания часто используемых значений (в данном примере FW_REGULAR, она же FW_NORMAL, соответствует значению 400 или обычной толщине линий).
- lfItalic — указывает наклонное (если true или 1) начертание текста.
- lfUnderline — если true или 1, то текст подчеркивается.
- lfStrikeOut — если true или 1, то текст перечеркивается.
- lfCharSet — определяет набор символов шрифта; содержит также ряд предопределенных констант. Значением поля по умолчанию является DEFAULT_

CHARSET, приводящее к тому, что система сама выбирает нужное значение в соответствии с текущими языковыми настройками.

- lfOutPrecision — определяет способ выбора системой шрифта, если по заданным параметрам подходят несколько шрифтов с одинаковым названием (например, выбирать ли только trueType-шрифты или же растровые). Часто можно обойтись значением по умолчанию OUT_DEFAULT_PRECIS.
- lfClipPrecision — определяет способ вывода на экран символов, частично не помещающихся в области отсечения; проще всего использовать CLIP_DEFAULT_PRECIS.
- lfQuality — задает качество вывода шрифта.
- lfPitchAndFamily — определяет шаг при выводе символов (используя значения DEFAULT_PITCH, FIXED_PITCH или VARIABLE_PITCH) и семейство шрифтов (используя значения FF_DECORATIVE, FF_DONTCARE, FF_MODERN, FF_ROMAN, FF_SCRIPT или FF_SWISS).
- lfFaceName — содержит название нужного шрифта (не более 31 символа в сочетании с нулевым символом в конце строки). Если не задано (значением поля является NULL), то система возвращает первый шрифт, подходящий по остальным заданным параметрам.

При выборе шрифта для вывода текста под различными углами следует учитывать еще и то, что растровым, векторным или trueType является запрашиваемый шрифт. Дело в том, что растровые шрифты, которым, кстати, является используемый по умолчанию системный шрифт, поддерживают способ вывода текста только параллельно горизонтальной оси.

Стоит также заметить, что многие возможности изменения параметров шрифта доступны при использовании свойств объекта Font, ассоциированного с объектом Canvas. Прибегать к использованию Windows API, как в приведенном здесь примере, потребуется достаточно редко.

В завершение стоит упомянуть следующий момент: если при выводе текста поверх какого-либо рисунка или компонента вас будет смущать то, что фон выводимого текста «затирает» изображение, вы можете изменить вид фона текста с помощью API-функции SetBkMode следующим образом:

```
int SetBkMode(HDC hdc,  int iBkMode);
```

В качестве первого аргумента эта функция принимает дескриптор контекста устройства (Canvas->Handle), а второго — вид фона. Вид фона может быть прозрачным или непрозрачным (по умолчанию), при этом используются значения второго параметра TRANSPARENT и OPAQUE соответственно.

## Использование областей отсечения

Если помните, в гл. 1 затрагивалась тема использования сложных регионов в качестве областей отсечения при выводе формы на экран, но что мешает использовать

регионы как области отсечения при самостоятельном рисовании? Window API содержит функции задания областей отсечения для контекста устройства (`TCanvas` в терминах библиотеки Borland C++ Builder), а раз возможность есть, то почему бы ею не воспользоваться?

В следующем примере демонстрируется возможность создания довольно сложного региона, который будет использоваться как область отсечения при выводе текста. На рис. 2.7 показан результат таких манипуляций.

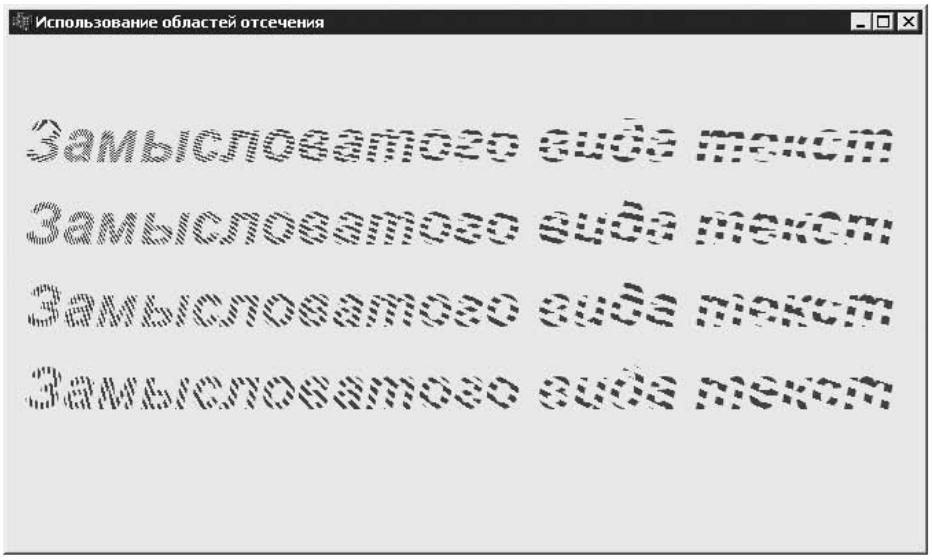


**Рис. 2.7.** Использование области отсечения

Работа реализованной в рамках рассматриваемого примера программы состоит из двух этапов. Сначала в конструкторе формы (листинг 2.9) создается комбинация «регион-многоугольник». Многоугольник, в свою очередь, образуется путем сложения треугольников, для каждого из которых одна вершина расположена в центре окружности, а две других находятся на дуге окружности на угловом расстоянии 0,5° друг от друга, причем в многоугольник включаются только нечетные треугольники.

**Листинг 2.9.** Создание сложного региона

```
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
  //Создание региона отсечения для рисования на форме
  int r = 1000;
  int x0 = 0;
  int y0 = 0;
  float angle_r, a;
  int x, y, p;
```

```
  TPoint pts[270];
  for ( a=0, p=0; a<=90; a+=1.0, p+=3 )
  {
    pts[p] = TPoint(x0, y0);
    angle_r = a * PI / 180;
    x = x0 + r * cos( angle_r );
    y = y0 + r * sin( angle_r );
    pts[p+1] = TPoint(x, y);
    angle_r = (a+0.5) * PI / 180;
    x = x0 + r * cos( angle_r );
    y = y0 + r * sin( angle_r );
    pts[p+2] = TPoint(x, y);
  }
  m_hRgn = ::CreatePolygonRgn(pts, 270, WINDING);
}
```

Внешний вид создаваемого таким образом региона показан на рис. 2.8. Естественно, радиус окружности при построении региона выбран заведомо б**о**льшим, чем ширина и высота формы.

Из-за слишком мелкого шага между образующими регион треугольниками и того, что пикселы на экране имеют все-таки фиксированный и не слишком маленький размер, в левом верхнем углу формы виден результат многочисленных наложений треугольников друг на друга, хотя в рассматриваемом примере это даже на руку.

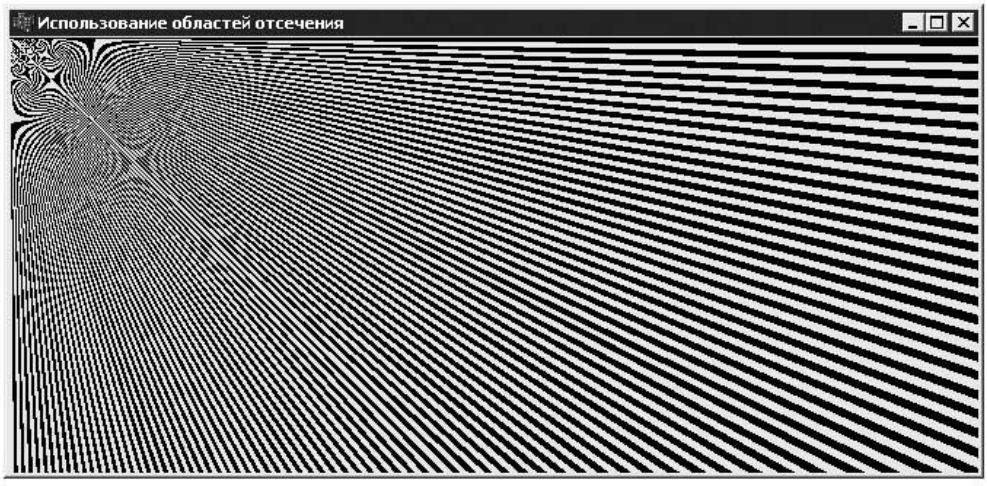


**Рис. 2.8.** Так выглядит область отсечения

Сам процесс рисования на форме с использованием отсечения прост до предела и, по сути, отличается от применявшегося ранее всего одной строкой кода (листинг 2.10).

**Листинг 2.10.** Вывод текста с использованием отсечения

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  ::SelectClipRgn(Canvas->Handle, m_hRgn);
  Canvas->Font->Size = 32;
  Canvas->Font->Name = "Cuorier New";
  Canvas->Font->Style = TFontStyles() << fsBold << fsItalic;
  Canvas->Font->Color = (TColor)RGB(0,100,0);
  Canvas->TextOut(10, 20, "Замысловатого вида текст");
  Canvas->TextOut(10, 80, "Замысловатого вида текст");
  Canvas->TextOut(10, 140, "Замысловатого вида текст");
  Canvas->TextOut(10, 200, "Замысловатого вида текст");
}
```

При вызове функции `SelectClipRgn` система производит копирование переданного региона, а потому после этого его можно спокойно удалять. Чтобы отменить использование области отсечения, достаточно вызвать функцию `SelectClipRgn`, передав в качестве второго параметра значение `NULL`.

В завершение хотелось бы напомнить, что не стоит забывать удалять созданные в процессе работы программы объекты GDI. В данном примере для региона функция `DeleteObject` вызывается в деструкторе формы.

# Простой графический редактор

Далее в качестве наглядного примера работы с графикой средствами Windows GDI будет рассмотрен процесс создания довольно простого графического редактора. Рассматриваемый здесь редактор обладает следующими возможностями:

- загрузка изображений из файла и сохранение их в файл на диске;
- рисование с использованием инструментов типа «кисть», «линия», «прямоугольник», «скругленный прямоугольник», «эллипс», «многоугольник» и «заливка»;
- преобразование изображений, например сжатие, поворот, изменение яркости и др. (отдельно способы реализации данных возможностей рассмотрены в разд. «Преобразование изображений»).

На рис. 2.9 показан итоговый вид описываемого графического редактора.

Если идея создания подобного редактора вас заинтересовала, то наберитесь терпения и ознакомьтесь со следующими подразделами.

## Класс графического редактора

Графический редактор будет реализован в отдельном классе `TGraphEditor`, объявление которого приведено в листинге 2.11.

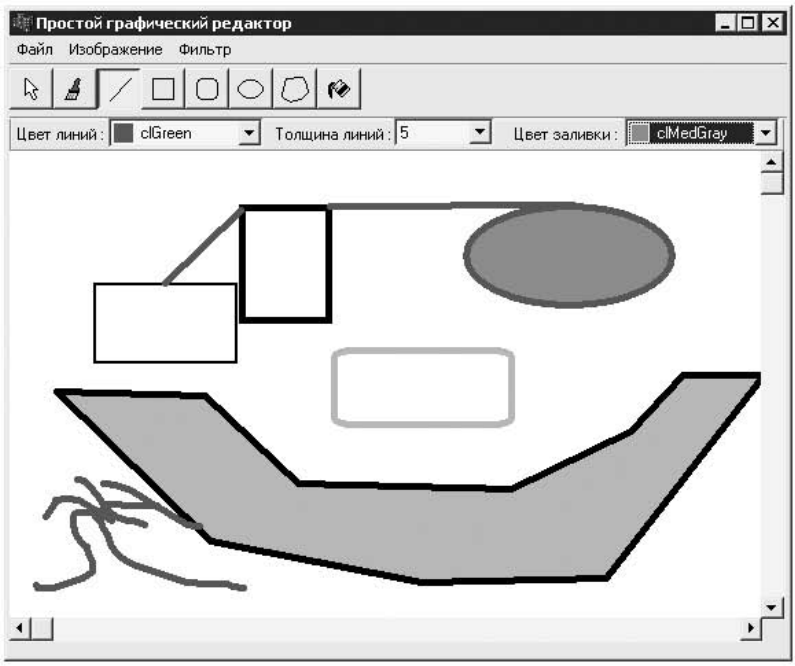


**Рис. 2.9.** Графический редактор в работе

**Листинг 2.11.** Класс TGraphEditor

```
class TGraphEditor
{
public:
  TGraphEditor( IGraphEditorEventListener *listener );
  ~TGraphEditor();
  //Редактирование
  void Clear();
  void SetCurrentTool(TEditorToolType tool_type);
  TEditorToolType GetCurrentTool();
  bool ApplyTransition(TEditorTransitionType transition_
type);
  //Настройка редактора и режима рисования
  void SetSize(int width, int height);
  void SetSize(TSize new_size);
  TSize GetSize();
  void SetLineColor(TColor new_color){ m_line_color = new_
color; }
  TColor GetLineColor(){ return m_line_color; }
  void SetLineWidth(int new_width){ m_line_width = new_width;
}
  int GetLineWidth(){ return m_line_width; }
```

```
  void SetFillColor(TColor new_color){ m_fill_color = new_
color; }
  TColor GetFillColor(){ return m_fill_color; }
  //Открытие/сохранение
  void OpenFile(AnsiString filename);
  void SaveFile(AnsiString filename);
  //Обработка ввода от мыши
  void MouseDown(HWND hWnd, TMouseButton Button, TShiftState
Shift, TPoint point);
  void MouseUp(HWND hWnd, TMouseButton Button, TShiftState
Shift, TPoint point);
  void MouseMove(HWND hWnd, TShiftState Shift, TPoint point);
  //Рисование
  void Draw(TCanvas *canvas, TRect visible_rect);
private:
  void BeginDrawing(HWND hWnd, TPoint point);
  void EndDrawing();
private:
  Graphics::TBitmap *m_image, *m_work_image;
  TransitionsMap m_transitions; //Все преобразования
                                //изображения
  ToolsMap m_tools;   //Все инструменты редактора
  ToolsMap::const_iterator m_current_tool; //Выбранный
                                           //инструмент
  TEditorToolType m_last_tool_id; //Идентификатор прошлого
                                  //инструмента
  bool m_drawing; //Если true, то в данный момент применяется
                  //инструмент,
                  //например рисуется линия
  //Настройки рисования
  TColor m_line_color;  //Цвет линий
  int m_line_width;     //Толщина линий
  TColor m_fill_color;  //Цвет заливки
  IGraphEditorEventListener *m_listener;
};
```

Создание редактора начинается с конструктора. Он принимает указатель на `IGraphEditorEventListener` — интерфейс, который должен реализовывать использующий редактор объект (в данном случае это форма). Указатель на `IGraphEditorEventListener` сохраняется в переменной `m_listener` и используется для уведомления контейнера о различных изменениях. Объявление `IGraphEditorEventListener` осуществляется следующим образом:

```
class IGraphEditorEventListener
{
public:
```

```
  virtual void OnNeedRepaintAll(TGraphEditor *editor) = 0;
  virtual void OnNeedRepaintVisible(TGraphEditor *editor) =
0;
  virtual void OnImageSizeChanged(TGraphEditor *editor) = 0;
  virtual void OnToolChanged(TGraphEditor *editor) = 0;
};
```

Метод `OnNeedRepaintAll` вызывается, если для корректного отображения области рисования (прямоугольника, отведенного графическому редактору) нужно перерисовать всю область рисования. Метод `OnNeedRepaintVisible` вызывается, если изменена и, соответственно, должна быть обновлена видимая часть изображения. Вызовом метода `OnImageSizeChanged` объект (форма в нашем случае), использующий редактор, уведомляется об изменении размера изображения. И, наконец, метод `OnToolChanged` используется с целью уведомления об изменении текущего инструмента (подробнее об этом рассказано ниже при рассмотрении инструментов рисования).

Далее в объявлении класса следует метод `Clear`, из названия которого понятно, что он предназначен для очистки чего-то, а если говорить точно, то для закрашивания всей области редактируемого изображения белым цветом (листинг 2.12).

**Листинг 2.12.** Очистка рисунка

```
void TGraphEditor::Clear()
{
  m_image->Canvas->Brush->Color = clWhite;
  m_image->Canvas->FillRect(TRect(0,0,m_image->Width, m_
image->Height));
  m_listener->OnNeedRepaintVisible(this);
}
```

Далее следуют два метода управления текущим инструментом: `SetCurrentTool`, который определяет выбор необходимого инструмента редактора, и `GetCurrentTool`, который возвращает идентификатор текущего инструмента. Инструментом может быть кисть, прямоугольник, эллипс и т. д. Идентификаторы реализованных инструментов хранятся в перечислении `TEditorToolType`:

```
enum TEditorToolType
{
  toolArrow,     //Указатель
  toolBrush,     //Кисть
  toolLine,      //Прямая линия
  toolRect,      //Прямоугольник
  toolRoundRect, //Скругленный прямоугольник
  toolEllipse,   //Эллипс
  toolPolygon,   //Полигон (многоугольник)
  toolFill,      //Заливка
};
```

Данные же самих инструментов редактора хранятся в карте (map), доступ к элементам которой производится по идентификатору инструмента:

```
typedef map<TEditorToolType, TEditorToolInfo*> ToolsMap;
```

в классе редактора соответствующая переменная:

```
ToolsMap m_tools;
```

Структура TEditorToolInfo, указатели на которую хранятся в m_tools, будет рассмотрена чуть позже, а вот способ реализации методов SetCurrentTool и GetCurrentTool приведен в листинге 2.13.

**Листинг 2.13.** Управление текущим инструментом

```
void TGraphEditor::SetCurrentTool(TEditorToolType tool_type)
{
  //Установка нового инструмента
  m_last_tool_id = (*m_current_tool).first;
  m_current_tool = m_tools.find(tool_type);
  (*m_current_tool).second->Tool->Reset();
}
TEditorToolType TGraphEditor::GetCurrentTool()
{
  //Возвращение идентификатора текущего инструмента
  return (*m_current_tool).first;
}
```

Здесь m_current_tool — закрытая переменная класса TGraphEditor (листинг 2.11), а точнее, итератор карты инструментов. По данным, записанным в этой переменной, редактор определяет в нужный ему момент то, каким инструментом ему необходимо воспользоваться.

Как было сказано в самом начале, данный графический редактор позволяет применять к изображению и различные преобразования. Применение определенного преобразования происходит при вызове метода ApplyTransition. Единственный аргумент этого метода — идентификатор преобразования, который берется из перечисления TEditorTransitionType:

```
enum TEditorTransitionType
{
  trReflect,     //Отражение
  trRotate,       //Поворот
  trStretch,      //Растяжение/сжатие
  trGrayScale,    //Черно-белое изображение
  trInversion,    //Инверсия цветов
  trBlurFilter,   //Размытие
  trMedianFilter,//Удаление помех
  trEmbossFilter,//Тиснение
```

```
  trBrightness,  //Регулятор яркости
  trMixing,      //Смешивание
};
```

Данные преобразований, так же как и данные инструментов, хранятся в карте (`map`), ключом которой являются уже идентификаторы преобразований:

```
typedef map<TEditorTransitionType, TEditorTransitionInfo*>
TransitionsMap;
```

в классе редактора соответствующая переменная:

```
TransitionsMap m_transitions;
```

Как реализовано применение преобразования к изображению, показано в листинге 2.14.

**Листинг 2.14.** Применение преобразования

```
bool TGraphEditor::ApplyTransition(TEditorTransitionType
transition_type)
{
  //Применяем к рисунку заданное преобразование
  TSize last_size = GetSize();
  if ( m_transitions[transition_type]->Transition->Execute(m_
image) )
  {
    if ( last_size.cx != m_image->Width || last_size.cy != m_
image->Height )
    {
      //Отреагируем на изменение размера изображения
      m_work_image->Width = m_image->Width;
      m_work_image->Height = m_image->Height;
      m_listener->OnImageSizeChanged(this);
      m_listener->OnNeedRepaintAll(this);
    }
    else
      m_listener->OnNeedRepaintVisible(this);
    return true;
  }
  return false;
}
```

Далее в объявлении класса `TGraphEditor` следуют три метода для получения (`GetSize`) и установки размера рисунка (два варианта метода `SetSize`), текст реализации которых приведен в листинге 2.15.

**Листинг 2.15.** Получение и установка размера изображения

```
void TGraphEditor::SetSize(int width, int height)
{
```

```
  m_image->Width = width;
  m_image->Height = height;
  m_work_image->Width = width;
  m_work_image->Height = height;
  if ( m_listener )
  {
    m_listener->OnImageSizeChanged(this);
    m_listener->OnNeedRepaintAll(this);
  }
}
void TGraphEditor::SetSize(TSize new_size)
{
  SetSize(new_size.cx, new_size.cy);
}
TSize TGraphEditor::GetSize()
{
  TSize size;
  size.cx = m_image->Width;
  size.cy = m_image->Height;
  return size;
}
```

Далее в листинге 2.11 следуют методы управления настройками рисования редактора. Желание не слишком усложнять программу привело к тому, что при рисовании графический редактор учитывает только:

- установленный цвет линий (методы `SetLineColor` и `GetLineColor`), при этом линии всегда сплошные;
- толщину линий (методы `SetLineWidth` и `GetLineWidth`);
- цвет заливки или цвет фона при рисовании прямоугольников, эллипсов и т. д. (методы `SetFillColor` и `GetFillColor`), при этом фон всегда сплошной.

Наконец, последними рассмотренными здесь методами являются методы открытия и сохранения рисунка `OpenFile` и `SaveFile`, пример реализации которых приведен в листинге 2.16.

**Листинг 2.16.** Открытие и сохранение рисунка

```
void TGraphEditor::OpenFile(AnsiString filename)
{
  m_image->LoadFromFile(filename);
  m_image->PixelFormat = pf24bit;
  m_work_image->Width = m_image->Width;
  m_work_image->Height = m_image->Height;
  m_listener->OnImageSizeChanged(this);
  m_listener->OnNeedRepaintAll(this);
}
```

```
void TGraphEditor::SaveFile(AnsiString filename)
{
  m_image->SaveToFile(filename);
}
```

Ничего особенного в этих методах нет. Стоит только пояснить, что рассматриваемый здесь графический редактор работает только с 24-разряднымм изображениями. В таких изображениях на один цветовой канал в кодировке RGB выделен 1 байт — именно на это ориентированы рассматриваемые далее в разделе «Преобразования изображений» фильтры и эффекты.

## Встраивание редактора в приложение

Как было сказано в самом начале, графический редактор, реализуемый классом `TGraphEditor`, встраивается в форму. В данном случае это будет форма `frmMain`, изображенная на рис. 2.10.

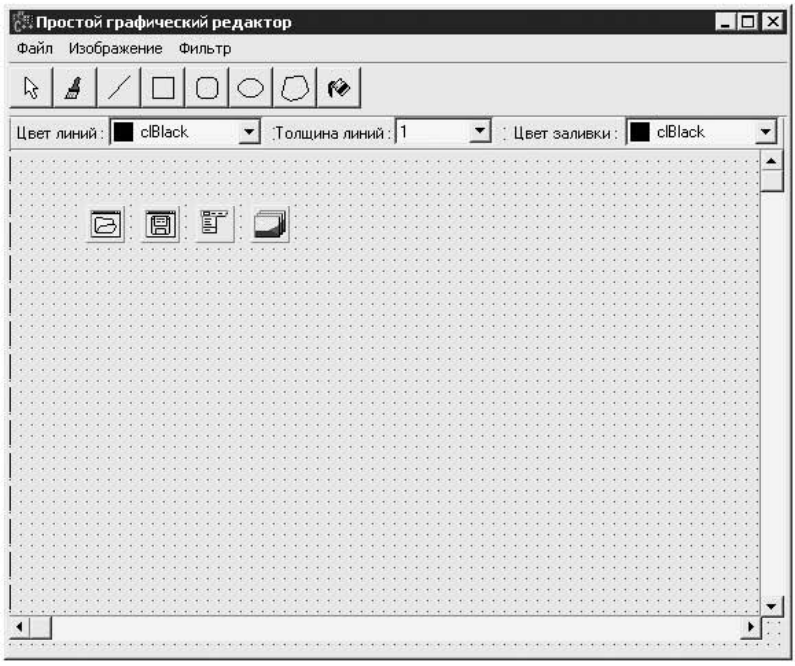


**Рис. 2.10.** Главная форма графического редактора

Главным, что необходимо для правильной работы редактора, на показанной на рисунке форме является компонент `TPaintBox` в центре и две полосы прокрутки. С помощью этих компонентов обеспечивается работа самых главных функций редактора: правильного отображения и прокрутки изображения.

Упоминаемый компонент `TPaintBox` назван `pict` (свойство `Name`). Для него заданы якоря `akLeft`, `akTop`, `akRight` и `akBottom` (свойство `Anchors`). Вертикальная

полоса прокрутки имеет имя vertScroll, и для нее заданы якоря akTop, akRight и akBottom, и, наконец, горизонтальная полоса прокрутки имеет имя horzScroll и якоря akLeft, akRight и akBottom.

Компонент TPaintBox с автоматически изменяемыми размерами избавляет от необходимости постоянно вычислять размер области, занимаемой рисунком. Вместе с тем этот компонент образует поверхность (канву, или Canvas), на которую графическим редактором выводится изображение. Собственно, этим роль компонента и ограничивается.

Теперь стоит вспомнить о методе Draw класса TGraphEditor. Текст его реализации приведен в листинге 2.17.

**Листинг 2.17.** Вывод изображения

```
void TGraphEditor::Draw(TCanvas *canvas, TRect visible_rect)
{
  //Отрисуем изображение в памяти (на m_work_image)
  TRect dest_rect( 0, 0, visible_rect.Width(), visible_rect.
Height() );
  m_work_image->Canvas->CopyRect(visible_rect,m_image-
>Canvas,visible_rect);
  if ( m_drawing )
  {
    //Применяемый инструмент (если он является инструментом
    //рисования)
    TDrawingTool *tool = dynamic_cast<TDrawingTool*>
                         ((*m_current_tool).second->Tool);
    if ( tool )
    {
      m_work_image->Canvas->Pen->Color = m_line_color;
      m_work_image->Canvas->Pen->Width = m_line_width;
      m_work_image->Canvas->Brush->Color = m_fill_color;
      tool->Draw(m_work_image->Canvas);
      m_work_image->Canvas->Pen->Color = clBlack;
      m_work_image->Canvas->Pen->Width = 1;
      m_work_image->Canvas->Brush->Color = clWhite;
    }
  }
  //Копирование полученного изображения на компонент
  canvas->CopyRect( dest_rect, m_work_image->Canvas, visible_
rect );
}
```

Как видно, помимо указателя на TCanvas, куда выводится изображение, метод Draw получает координаты некой видимой области. Дело в том, что редактируемое изображение по размеру может быть намного больше только что рассмотренного

компонента TPaintBox на форме. В таких случаях этот компонент в сочетании с полосами прокрутки задает координаты «окна», находящегося над определенной частью рисунка, и именно содержимое этого «окна» отображается компонентом TPaintBox.

Метод Draw редактора обладает еще одной особенностью: перед выводом на экран запрашиваемая часть изображения формируется сначала в битовой матрице m_work_image, что позволяет исключить мерцания во время прорисовки текущего состояния применяемого инструмента, например текущего размера добавляемого на рисунок прямоугольника (подробнее об инструментах рисования рассказано чуть ниже).

Чтобы обеспечить правильную прокрутку изображения, при изменении размера формы нужно пересчитывать пределы значений полос прокрутки, что и выполняется методом RecalcScrolls, вызываемым обработчиком события OnResize (листинг 2.18).

**Листинг 2.18.** Реакция на изменение размера формы

```
void __fastcall TfrmMain::FormResize(TObject *Sender)
{
  RecalcScrolls();
}
void TfrmMain::RecalcScrolls()
{
  if (m_deleted) return;
  //Переустановка диапазонов полос прокрутки рисунка
  TSize image_size = m_editor.GetSize();
  if ( image_size.cx <= pict->Width )
  {
    //Рисунок полностью видим по ширине
    horzScroll->Max = 0;
    horzScroll->Position = 0;
    horzScroll->Enabled = false;
  }
  else
  {
    //Нужна прокрутка по ширине
    horzScroll->Max = image_size.cx - pict->Width;
    horzScroll->Enabled = true;
  }
  if ( image_size.cy <= pict->Height )
  {
    //Рисунок полностью видим по высоте
    vertScroll->Max = 0;
    vertScroll->Position = 0;
    vertScroll->Enabled = false;
  }
```

```
  else
  {
    //Нужна прокрутка по высоте
    vertScroll->Max = image_size.cy - pict->Height;
    vertScroll->Enabled = true;
  }
}
```

Чтобы в компоненте `TPaintBox` показывалась только часть рисунка, нужно не забыть вызвать метод `TGraphEditor::Draw`, определив тем самым границы видимой части рисунка (листинг 2.19).

**Листинг 2.19.** Вывод видимой части рисунка

```
void __fastcall TfrmMain::pictPaint(TObject *Sender)
{
  RepaintImage();
}
void __fastcall TfrmMain::horzScrollScroll(TObject *Sender,
      TScrollCode ScrollCode, int &ScrollPos)
{
  RepaintImage();
}
void __fastcall TfrmMain::vertScrollScroll(TObject *Sender,
      TScrollCode ScrollCode, int &ScrollPos)
{
  RepaintImage();
}
void TfrmMain::RepaintImage()
{
  TRect rc;
  rc.Left = horzScroll->Position;
  rc.Right = rc.Left + pict->Width;
  rc.Top = vertScroll->Position;
  rc.Bottom = rc.Top + pict->Height;
  m_editor.Draw(pict->Canvas, rc);
}
```

Теперь необходимо вспомнить об интерфейсе `IGraphEditorEventListener`, который должна реализовать форма. Все необходимые изменения, которые должны быть произведены при объявлении класса `TfrmMain`, указаны в листинге 2.20.

**Листинг 2.20.** Изменения класса TfrmMain

```
class TfrmMain : public TForm, public IGraphEditorEventListen
er
{
//...
public: //IGraphEditorEventListener
```

```
  virtual void OnNeedRepaintAll(TGraphEditor *editor);
  virtual void OnNeedRepaintVisible(TGraphEditor *editor);
  virtual void OnImageSizeChanged(TGraphEditor *editor);
  virtual void OnToolChanged(TGraphEditor *editor);
};
```

Как видите, тем самым был всего лишь унаследован `IGraphEditorEventListener` и переопределены его методы, реализация которых проста до предела (листинг 2.21).

**Листинг 2.21.** Реализация IGraphEditorEventListener

```
__fastcall TfrmMain::TfrmMain(TComponent* Owner)
  : TForm(Owner), m_editor(this) //Передача редактору
указателя
                                 //на IGraphEditorEventLis-
tener
{
  m_deleted = false;
}
void TfrmMain::OnNeedRepaintAll(TGraphEditor *editor)
{
  Refresh();
}
void TfrmMain::OnNeedRepaintVisible(TGraphEditor *editor)
{
  RepaintImage();
}
void TfrmMain::OnImageSizeChanged(TGraphEditor *editor)
{
  RecalcScrolls();
}
void TfrmMain::OnToolChanged(TGraphEditor *editor)
{
  RefreshToolbox();
}
```

Функция `RefreshToolbox`, вызываемая `TfrmMain::OnToolChanged`, синхронизирует состояния панели инструментов и компонентов, задающих настройки рисования (см. листинг 2.25). Сама функция выглядит так, как показано в листинге 2.22.

**Листинг 2.22.** Синхронизация вида формы с состоянием редактора

```
void TfrmMain::RefreshToolbox()
{
  //Выделение текущего инструмента редактора
  switch( m_editor.GetCurrentTool() )
  {
  case toolArrow:
```

```
    arrowTool->Down = true;
    break;
  case toolBrush:
    brushTool->Down = true;
    break;
  case toolLine:
    lineTool->Down = true;
    break;
  case toolRect:
    rectTool->Down = true;
    break;
  case toolRoundRect:
    roundRectTool->Down = true;
    break;
  case toolEllipse:
    ellipseTool->Down = true;
    break;
  case toolPolygon:
    polygonTool->Down = true;
    break;
  case toolFill:
    fillTool->Down = true;
    break;
  }
  //Текущие настройки рисования
  lineColor->Selected = m_editor.GetLineColor();
  fillColor->Selected = m_editor.GetFillColor();
  String width_str(m_editor.GetLineWidth());
  lineWidth->ItemIndex = lineWidth->Items->IndexOf(width_
str);
}
```

Вот, в принципе, и все, что нужно, чтобы графический редактор позволял показывать и прокручивать изображение. А чтобы можно было еще и рисовать, придется для формы определить также обработчики сообщений, поступающих от мыши, в которых вызвать соответствующие методы класса `TGraphEditor` (листинг 2.23).

**Листинг 2.23.** Обработка сообщений, поступающих от мыши

```
void __fastcall TfrmMain::pictMouseDown(TObject *Sender,
TMouseButton Button,
      TShiftState Shift, int X, int Y)
{
  m_editor.MouseDown( Handle, Button, Shift, TPoint(
                       X + horzScroll->Position,
                       Y + vertScroll->Position) );
}
```

```
void __fastcall TfrmMain::pictMouseMove(TObject *Sender,
TShiftState Shift,
      int X, int Y)
{
  m_editor.MouseMove( Handle, Shift, TPoint(
                       X + horzScroll->Position,
                       Y + vertScroll->Position) );
}
void __fastcall TfrmMain::pictMouseUp(TObject *Sender,
TMouseButton Button,
      TShiftState Shift, int X, int Y)
{
  m_editor.MouseUp( Handle, Button, Shift, TPoint(
                     X + horzScroll->Position,
                     Y + vertScroll->Position) );
}
```

Если нужно реализовать возможность загрузки изображений из файла и сохранения создаваемых в редакторе изображений в файл, то необходимо добавить на форму соответствующие компоненты (в данном примере `openDlg` и `saveDlg`) и задействовать их при выборе соответствующих пунктов (листинг 2.24).

**Листинг 2.24.** Открытие и сохранение рисунка

```
void __fastcall TfrmMain::mnuOpenClick(TObject *Sender)
{
  if ( openDlg->Execute() )
    m_editor.OpenFile(openDlg->FileName);
}
void __fastcall TfrmMain::mnuSaveClick(TObject *Sender)
{
  if ( saveDlg->Execute() )
    m_editor.SaveFile(saveDlg->FileName);
}
```

Если вы помните, при рисовании редактор учитывает заданный цвет и толщину линий, а также цвет заливки объекта — для задания этих настроек на форме предусмотрены компоненты `lineColor` (`TColorBox`), `lineWidth` (`TComboBox`) и `fillColor` (`TColorBox`) соответственно. В листинге 2.25 продемонстрирован способ задания настроек рисования с помощью указанных компонентов.

**Листинг 2.25.** Задание настроек рисования

```
void __fastcall TfrmMain::lineColorChange(TObject *Sender)
{
  m_editor.SetLineColor(lineColor->Selected);
}
void __fastcall TfrmMain::lineWidthChange(TObject *Sender)
{
```

```
  int width = (*lineWidth->Items)[lineWidth->ItemIndex].
ToInt();
  m_editor.SetLineWidth(width);
}
void __fastcall TfrmMain::fillColorChange(TObject *Sender)
{
  m_editor.SetFillColor(fillColor->Selected);
}
```

Для выбора инструмента графического редактора на форме предусмотрена панель инструментов (`TToolBar`). За каждой кнопкой этой панели закреплена команда вызова метода `SetCurrentTool` редактора, как показано в листинге 2.26. Поскольку расписывать обработчики нажатий всех кнопок особого смысла нет, то показаны обработчики всего двух кнопок.

**Листинг 2.26.** Выбор инструмента для рисования

```
void __fastcall TfrmMain::arrowToolClick(TObject *Sender)
{
  m_editor.SetCurrentTool(toolArrow); //Выбираем инструмент
"Указатель"
}
void __fastcall TfrmMain::brushToolClick(TObject *Sender)
{
  m_editor.SetCurrentTool(toolBrush); //Выбираем инструмент
"Кисть"
}
```

И, наконец, последнее: для применения к рисунку преобразований нужно создать соответствующие пункты меню, при выборе которых будет вызываться метод `ApplyTransition` и передаваться ему идентификатор нужного преобразования (листинг 2.27).

**Листинг 2.27.** Применение преобразований

```
void __fastcall TfrmMain::mnuGrayScaleClick(TObject *Sender)
{
  m_editor.ApplyTransition(trGrayScale); //Получение черно-
                                         //белого
                                         //изображения
}
void __fastcall TfrmMain::mnuStretchClick(TObject *Sender)
{
  m_editor.ApplyTransition(trStretch);  //Растягивание/сжатие
                                        //изображения
}
```

За исключением мелких деталей, выше было описано все, что нужно сделать, чтобы внедрить в форму графический редактор. Далее будут рассмотрены спосо-

бы реализации инструментов рисования и выполнения преобразований изображения, причем если первая тема рассмотрена именно как часть создания графического редактора, то вторая вынесена в отдельный раздел, поскольку различные преобразования изображений — тема гораздо более объемная и вместе с тем еще и интересная.

## Инструменты для рисования

Перед тем как приступить к рассмотрению способов реализации инструментов рисования графического редактора на уровне исходного кода, придется немного отвлечься от программирования. Для начала нужно определиться, что же в будущем редакторе будет считаться процессом рисования.

Пусть рисование, то есть применение инструмента, начинается с нажатия левой кнопки мыши и заканчивается или одновременно с этим (для инструментов вроде «заливки»), или уже после отпускания этой кнопки. Получается, что инструментам требуется минимум одна точка, но могут потребоваться и две (для создания линии, прямоугольника и т. д.). При построении же многоугольника нужно более двух точек, но их точное количество заранее не известно (построение многоугольника может быть завершено пользователем в любой момент). Таким образом, для некоторых инструментов («многоугольник») требуется добавление точек в ходе рисования, которое будет выполняться щелчком правой кнопкой мыши при удерживаемой в нажатом состоянии левой кнопке. Кроме того, при перемещении указателя мыши в ходе рисования было бы не лишним сообщать применяемому инструменту текущее положение указателя, чтобы иметь возможность видеть на рисунке реакцию инструмента на перемещение указателя. Наконец, после применения некоторых инструментов было бы полезно обеспечить автоматическое возвращение к ранее выбранному в редакторе инструменту.

Приведенные в предыдущем абзаце размышления нашли отражение в наборе методов, включенных в базовый класс инструмента графического редактора (листинг 2.28).

**Листинг 2.28.** Базовый класс инструмента редактора

```
class TEditorTool
{
public:
  virtual ~TEditorTool(){};
  virtual void Reset() = 0;
  virtual bool AddPoint(TPoint point) = 0;
  virtual void TestPoint(TPoint point) = 0;
};
```

Метод `Reset` класса `TEditorTool` предназначен для подготовки инструмента к работе (инициализация переменных, к примеру) и должен вызываться перед каждым применением инструмента. Метод `AddPoint` предназначен для сообщения инструменту координат точек, указанных пользователем (при нажатии

и отпускании левой кнопки мыши, а также при каждом щелчке правой кнопкой мыши в оговоренных ранее случаях). Как только метод `AddPoint` возвращает `false`, считается, что все нужные точки сообщены инструменту и рисование завершено (заметьте, что отпускание левой кнопки мыши — это еще один способ завершения процесса рисования). Метод `TestPoint` должен использоваться для сообщения инструменту текущего положения указателя мыши в процессе рисования.

**ПРИМЕЧАНИЕ**

Координаты точек, передаваемые в TEditorTool::AddPoint и TEditorTool::TestPoint, должны быть в системе координат рисунка.

Как можно заметить, у класса `TEditorTool` нет метода, который бы позволял нарисовать создаваемую инструментом фигуру. Однако это нужно не для всех инструментов («заливка», например), потому инструменты, которым «есть что показать» в процессе рисования, в данном случае будут унаследованы от класса `TDrawingTool`, показанного в листинге 2.29.

**Листинг 2.29.** Класс TDrawingTool

```
//Графические инструменты (все, кроме пипетки)
class TDrawingTool : public TEditorTool
{
protected:
  vector<TPoint> m_points; //Опорные точки для инструмента
public:
  virtual void Draw(TCanvas *canvas) = 0;
  virtual void Reset(){ m_points.clear(); }
};
```

В классе `TDrawingTool` появился метод `Draw`, который и предназначен именно для вывода на экран создаваемой инструментом фигуры в процессе рисования. Также в классе `TDrawingTool` частично реализована логика работы с точками, используемыми инструментами.

Теперь можно приступать к рассмотрению конкретных инструментов. Вначале самый простой — инструмент «указатель». Главная задача этого инструмента — ничего не делать (листинг 2.30).

**Листинг 2.30.** Инструмент «указатель»

```
class TArrowTool : public TEditorTool
{
public:
  virtual void Reset(){}
  virtual bool AddPoint(TPoint point){ return false; }
  virtual void TestPoint(TPoint point){}
};
```

Этот «ничего не делающий» инструмент нужен лишь для того, чтобы перевести графический редактор в режим простого просмотра изображения, при котором пользователь не боялся бы случайно его испортить.

Для инструментов «линия», «прямоугольник», «эллипс», «скругленный прямоугольник», или, иначе говоря, для всех инструментов, для работы которых необходимы две точки, будет введен еще один базовый класс — `TTwoPointTool`. В этом классе (листинг 2.31) реализована общая для всех инструментов подобного рода логика работы с двумя опорными точками, используемыми для построения фигуры.

**Листинг 2.31.** Класс TTwoPointTool

```
class TTwoPointTool : public TDrawingTool
{
public:
  virtual bool AddPoint(TPoint point)
  {
    if (m_points.size() < 2 )
      m_points.push_back(point);
    else
      m_points[1] = point;
    return m_points.size() < 2;
  }
  virtual void TestPoint(TPoint point)
  {
    if ( m_points.size() == 1 )
      m_points.push_back(point);
    else
      m_points[1] = point;
  }
};
```

Для всех инструментов, классы которых унаследованы от `TTwoPointTool`, остается лишь реализовать метод `Draw` так, как показано в листинге 2.32.

**Листинг 2.32.** Инструменты, использующие две опорные точки

```
//Инструмент "Линия"
class TLineTool : public TTwoPointTool
{
public:
  virtual void Draw(TCanvas *canvas)
  {
    if ( m_points.size() == 2 )
    {
      canvas->MoveTo(m_points[0].x, m_points[0].y);
      canvas->LineTo(m_points[1].x, m_points[1].y);
    }
```

```
  }
};
//Инструмент "Прямоугольник"
class TRectTool : public TTwoPointTool
{
public:
  virtual void Draw(TCanvas *canvas)
  {
    if ( m_points.size() == 2 )
    {
      canvas->Rectangle(m_points[0].x, m_points[0].y,
                        m_points[1].x, m_points[1].y);
    }
  }
};
//Инструмент "Скругленный прямоугольник"
class TRoundRectTool : public TTwoPointTool
{
public:
  virtual void Draw(TCanvas *canvas)
  {
    if ( m_points.size() == 2 )
    {
      canvas->RoundRect(m_points[0].x, m_points[0].y,
                        m_points[1].x, m_points[1].y,
                        abs( m_points[1].x - m_points[0].x ) /
5,
                        abs( m_points[1].y - m_points[0].y ) /
5);
    }
  }
};
//Инструмент "Эллипс"
class TEllipseTool : public TTwoPointTool
{
public:
  virtual void Draw(TCanvas *canvas)
  {
    if ( m_points.size() == 2 )
    {
      canvas->Ellipse(m_points[0].x, m_points[0].y,
                      m_points[1].x, m_points[1].y);
    }
  }
};
```

Далее рассмотрен инструмент «заливка». Понятно, что этому инструменту требуется указание всего одной точки рисунка. Также, в отличие от приведенных ранее инструментов, данному инструменту для правильного выполнения заливки нужно передать битовую матрицу, в которой хранится рисунок, и цвет самой заливки. В листинге 2.33 приводится текст реализации инструмента «заливка» с учетом указанных особенностей.

**Листинг 2.33.** Инструмент «заливка»

```
class TFillTool : public TEditorTool
{
  Graphics::TBitmap *m_image;
  TColor *m_fill_color;
public:
  TFillTool(Graphics::TBitmap *image, TColor *fill_color)
  {
    m_image = image;
    m_fill_color = fill_color;
  }
  virtual void Reset(){}
  virtual bool AddPoint(TPoint point)
  {
    //Заливка области рисунка
    m_image->Canvas->Brush->Color = *m_fill_color;
    m_image->Canvas->FloodFill( point.x, point.y,

m_image->Canvas->Pixels[point.x][point.y],
                                fsSurface );
    return false;
  }
  virtual void TestPoint(TPoint point){}
};
```

Указатели, которые необходимо передать в конструктор класса `TFillTool`, вы увидите чуть позже (в листинге 2.36). Сначала же будет рассмотрен способ реализации инструмента «многоугольник» (или «полигон») (листинг 2.34).

**Листинг 2.34.** Инструмент «многоугольник»

```
class TPolygonTool : public TDrawingTool
{
public:
  virtual bool AddPoint(TPoint point)
  {
    m_points.push_back(point);
    m_points.push_back(point);
    return true;
  }
```

```
  virtual void TestPoint(TPoint point)
  {
    m_points[m_points.size()-1] = point;
  }
  virtual void Draw(TCanvas *canvas)
  {
    if ( m_points.size() > 0 )
    {
      TPoint *pts = new TPoint[m_points.size()];
      for ( unsigned int i=0; i<m_points.size(); i++ )
      {
        pts[i] = m_points[i];
      }
      canvas->Polygon( pts, m_points.size()-1 );
      delete []pts;
    }
  }
};
```

Как видите, для многоугольника пришлось реализовать собственный алгоритм работы с не определенным заранее количеством опорных точек, который получился даже проще, чем алгоритм для инструментов, работающих с двумя точками. Вероятно, основной особенностью этого инструмента является то, что его метод `AddPoint` всегда возвращает `true`, то есть построение многоугольника должен завершить сам пользователь (как это реализовать, показано чуть ниже).

Наконец, текст реализации последнего предусмотренного в данном примере инструмента «кисть» приведен в листинге 2.35. Как видите, между инструментами «кисть» и «многоугольник» существует много общего, только при построении многоугольника новые точки добавляются пользователем явно (добавление точек в массив происходит в методе `AddPoint`, который вызывается нажатием кнопки мыши), а при рисовании кистью новые точки добавляются автоматически при перемещении указателя над рисунком (метод `TestPoint`).

**Листинг 2.35.** Инструмент «кисть»

```
class TBrushTool : public TDrawingTool
{
public:
  virtual bool AddPoint(TPoint point)
  {
    m_points.push_back(point);
    return m_points.size() == 1;
  }
  virtual void TestPoint(TPoint point)
  {
    m_points.push_back(point);
```

```
  }
  virtual void Draw(TCanvas *canvas)
  {
    if ( m_points.size() > 0 )
    {
      TPoint *pts = new TPoint[m_points.size()];
      for ( unsigned int i=0; i<m_points.size(); i++ )
      {
        pts[i] = m_points[i];
      }
      canvas->Polyline( pts, m_points.size()-1 );
      delete []pts;
    }
  }
};
```

Рассмотрение способов реализации инструментов на этом можно считать законченным. Как же это все работает на деле? Как вы, вероятно, помните, все сведения об инструментах для рисования класс `TGraphEditor` хранит в карте (`map`). Сведения же об инструментах, которые нужны редактору, можно увидеть в листинге 2.36.

**Листинг 2.36.** Данные инструмента, используемые редактором

```
struct TEditorToolInfo
{
  TEditorTool *Tool;
  bool HasManyPoints; //Если true, то для инструмента нужно
более двух точек
  TAfterToolAction AfterToolAction;
  ///////////////////
  TEditorToolInfo(TEditorTool *tool, bool many_points,
                  TAfterToolAction after_tool_action)
  {
    Tool = tool;
    HasManyPoints = many_points;
    AfterToolAction = after_tool_action;
  }
  ~TEditorToolInfo()
  {
    delete Tool;
  }
};
typedef map<TEditorToolType, TEditorToolInfo*> ToolsMap;
```

Как видите, набор полей структуры `TEditorToolInfo` отражает рассмотренные в самом начале особенности работы различных инструментов рисования. Так, помимо указателя на экземпляр инструмента, в этой структуре указывается, нужно

ли инструменту более двух точек (поле HasManyPoints), и если нужно, то при каждом щелчке правой кнопкой мыши (при нажатой левой кнопке) инструменту в метод AddPoint передаются координаты соответствующей точки рисунка. Также в структуре TEditorToolInfo хранится информация о том, какое действие должен выполнить графический редактор после применения инструмента. На момент реализации этого примера было предусмотрено два действия:

```
enum TAfterToolAction
{
  actionNo,       //Нет действия
  actionLastTool, //Переключить на предыдущий инструмент
};
```

Несмотря на то что здесь используется только actionNo, если у вас возникнет желание добавить новый инструмент типа «пипетка» (как в графическом редакторе Paint), после которого обычно выбирается предыдущий инструмент, то проблем с встраиванием его в приведенный здесь графический редактор возникнуть не должно.

Данные инструментов записываются во внутреннюю переменную m_tools в конструкторе класса TGraphEditor (листинг 2.37). Именно здесь происходит создание экземпляров инструментов и связывание их с идентификаторами, заданными в перечислении TEditorToolType.

**Листинг 2.37.** Заполнение данных об инструментах

```
TGraphEditor::TGraphEditor(IGraphEditorEventListener *listen-
er)
{
  ...
  //Инициализация инструментов
  m_tools[toolArrow] = new TEditorToolInfo(new TArrowTool,
false, actionNo);
  m_tools[toolBrush] = new TEditorToolInfo(new TBrushTool,
false, actionNo);
  m_tools[toolLine] = new TEditorToolInfo(new TLineTool,
false, actionNo);
  m_tools[toolRect] = new TEditorToolInfo(new TRectTool,
false, actionNo);
  m_tools[toolRoundRect] = new TEditorToolInfo(new TRoun-
dRectTool, false,
                           actionNo);
  m_tools[toolEllipse] = new TEditorToolInfo(new TEllipseTool,
false,
                         actionNo);
  m_tools[toolPolygon] = new TEditorToolInfo(new TPolygon-
Tool, true,
                         actionNo);
```

```
  m_tools[toolFill] = new TEditorToolInfo(new TFillTool(m_
image,
                        &m_fill_color), false, actionNo);
  m_last_tool_id = toolArrow;
  m_current_tool = m_tools.find(toolArrow);
  ...
}
```

Процесс использования инструментов рисования тесно связан с обработкой сообщений ввода, поступающих от мыши. Ранее в листинге 2.23 было показано, что при нажатии кнопок мыши, их отпускании, а также при перемещении указателя мыши над рисунком вызываются методы `MouseDown`, `MouseUp` и `MouseMove` соответственно класса `TGraphEditor`. Текст реализации этих методов приведен в листинге 2.38.

**Листинг 2.38.** Обработка сообщений ввода от мыши

```
void TGraphEditor::MouseDown(HWND hWnd, TMouseButton Button,
                             TShiftState Shift, TPoint point)
{
  bool many_points = (*m_current_tool).second->HasManyPoints;
  if ( !m_drawing && Button == mbLeft )
  {
    BeginDrawing( hWnd, point );
  }
  else if ( Button == mbRight && many_points )
  {
    //Добавление новой точки опорной точки инструмента
    if ( !(*m_current_tool).second->Tool->AddPoint(point) )
    {
      //Последняя требуемая опорная точка создана
      EndDrawing();
    }
  }
}
void TGraphEditor::MouseUp(HWND hWnd, TMouseButton Button,
                           TShiftState Shift, TPoint point)
{
  if ( m_drawing && Button == mbLeft )
  {
    //Завершение рисования
    (*m_current_tool).second->Tool->AddPoint(point);
    EndDrawing();
  }
}
void TGraphEditor::MouseMove(HWND hWnd, TShiftState Shift,
TPoint point)
{
```

```
  if ( m_drawing )
  {
    (*m_current_tool).second->Tool->TestPoint(point);
    m_listener->OnNeedRepaintVisible(this);
  }
}
```

Как видите, в обработке сообщений, поступающих от мыши, нет ничего сложного. Так, при нажатии кнопки мыши возможны два варианта реакции программы на совершаемое действие:

- если нажата левая кнопка мыши, то начинается рисование (применение инструмента);
- если нажата правая кнопка мыши в ходе рисования, то применяемому в данный момент инструменту передаются координаты точки нахождения указателя, и если инструменту точек больше не требуется (`AddPoint` возвращает `false`), то рисование прекращается.

После отпускания левой кнопки мыши начатое рисование завершается. При перемещении указателя над рисунком во время рисования координаты указателя мыши передаются применяемому инструменту рисования.

В листинге 2.38 можно увидеть процедуры вызова двух вспомогательных методов: `BeginDrawing` (который подготавливает редактор к применению инструмента рисования) и `EndDrawing` (который сохраняет результат применения инструмента). Текст реализации этих двух методов приводится в листинге 2.39.

**Листинг 2.39.** Начало и завершение рисования

```
//Начало рисования
void TGraphEditor::BeginDrawing(HWND hWnd, TPoint point)
{
  TEditorTool *tool = (*m_current_tool).second->Tool;
  tool->Reset();
  if ( tool->AddPoint( point ) )
  {
    //Начало рисования
    m_drawing = true;
    ::SetCapture(hWnd);
  }
  else
  {
    EndDrawing();
  }
}
//Завершение рисования
void TGraphEditor::EndDrawing()
{
```

```
  m_drawing = false;
  ::ReleaseCapture();
  TDrawingTool *tool =
   dynamic_cast<TDrawingTool*>((*m_current_tool).second-
>Tool);
  if ( tool )
  {
    //Сохранение изменений, внесенных инструментом
    m_image->Canvas->Pen->Color = m_line_color;
    m_image->Canvas->Pen->Width = m_line_width;
    m_image->Canvas->Brush->Color = m_fill_color;
    tool->Draw(m_image->Canvas);
  }
  //Определение действия, выполняемого после применения
  //инструмента
  switch ( (*m_current_tool).second->AfterToolAction )
  {
  case actionNo:
    //Никакие действия не производятся
    break;
  case actionLastTool:
    //Возврат к предыдущему инструменту
    SetCurrentTool(m_last_tool_id);
    m_listener->OnToolChanged(this);
    break;
  }
  m_listener->OnNeedRepaintVisible(this);
}
```

Вот, собственно, и все, что необходимо выполнить, чтобы создать несложный графический редактор, подобный рассмотренному. Пока этот редактор способен выполнять не слишком много, но, пожалуй, служит наглядным примером реализации возможностей Windows GDI.

В следующем разделе будут рассмотрены преобразования изображений, которые могут быть очень легко внедрены в описываемый графический редактор.

# Преобразования изображений

Существует множество способов преобразовать изображение. Различные алгоритмы преобразования используются во многих программах: от простейших приложений для просмотра изображения до профессиональных графических редакторов.

В этом разделе будут рассмотрены несколько несложных приемов работы с изображениями. Некоторые примеры полностью основаны на встроенных в библиотеку Borland C++ Builder возможностях, таких, например, как сжатие и растяжение

изображений, но большинство примеров представляют собой программы, позволяющие манипулировать изображениями на уровне отдельных точек рисунка. Все приведенные здесь примеры встроены в рассмотренный выше графический редактор.

Отдельные преобразования выполнены в отдельных классах — преобразователях изображений. Классы, позволяющие реализовать каждый конкретный преобразователь, наследуются от приведенного в листинге 2.40 абстрактного класса.

**Листинг 2.40.** Базовый класс для всех преобразователей

```
class TEditorTransition
{
public:
  virtual ~TEditorTransition(){}
  virtual bool Execute(Graphics::TBitmap *image) = 0;
};
```

Так, для каждого преобразователя должен быть реализован один-единственный метод, принимающий указатель на `TBitmap`, который содержит исходное изображение (сюда же должен записываться результат преобразования), и возвращающий `true` при успешном выполнении преобразования.

Процесс использования преобразований в программе состоит из двух этапов. Вначале создаются экземпляры всех преобразователей — к примеру, в данном графическом редакторе объекты преобразователей создаются явно в конструкторе класса `TGraphEditor` (листинг 2.41).

**Листинг 2.41.** Создание экземпляров преобразователей

```
TGraphEditor::TGraphEditor(IGraphEditorEventListener *listen-
er)
{
  ...
  m_transitions[trReflect] =
   new TEditorTransitionInfo(new TReflectTransition);
  m_transitions[trRotate] =
   new TEditorTransitionInfo(new TRotateTransition);
  m_transitions[trStretch] =
   new TEditorTransitionInfo(new TStretchTransition);
  m_transitions[trGrayScale] =
   new TEditorTransitionInfo(new TGrayScaleTransition);
  m_transitions[trInversion] =
   new TEditorTransitionInfo(new TInversionTransition);
  m_transitions[trBlurFilter] =
   new TEditorTransitionInfo(new TBlurFilter);
  m_transitions[trMedianFilter] =
   new TEditorTransitionInfo(new TMedianFilter);
  m_transitions[trEmbossFilter] =
   new TEditorTransitionInfo(new TEmbossFilter);
```

```
  m_transitions[trBrightness] =
   new TEditorTransitionInfo(new TBrightnessTransition);
  m_transitions[trMixing] =
   new TEditorTransitionInfo(new TMixingTransition);
}
```

Здесь, в частности, с созданными объектами еще и связываются идентификаторы (значения из перечисления `TEditorTransitionType`). Указатели на созданные объекты передаются в конструктор структуры `TEditorTransitionInfo` (аналогична структуре `TEditorToolInfo`), а хранится только указатель на объект-преобразователь.

Второй этап — непосредственно применение преобразования, которое выглядит как вызов метода `Execute` нужного преобразователя в нужный момент. Так, в класс `TGraphEditor` для этого добавлен метод `ApplyTransition`; в нем и вызывается метод `Execute` преобразователя, идентификатор которого передается как значение параметра `ApplyTransition` (листинг 2.42).

**Листинг 2.42.** Применение преобразования в графическом редакторе

```
bool TGraphEditor::ApplyTransition(TEditorTransitionType
transition_type)
{
  //Применение к рисунку заданного преобразования
  TSize last_size = GetSize();
  if ( m_transitions[transition_type]->Transition->Execute(m_
image) )
  {
    if ( last_size.cx != m_image->Width || last_size.cy != m_
image->Height )
    {
      //Реакция на изменение размера изображения
      m_work_image->Width = m_image->Width;
      m_work_image->Height = m_image->Height;
      m_listener->OnImageSizeChanged(this);
      m_listener->OnNeedRepaintAll(this);
    }
    else
      m_listener->OnNeedRepaintVisible(this);
    return true;
  }
  return false;
}
```

Единственной сложностью при встраивании преобразований в редактор является необходимость проверки полученного изображения на предмет наличия изменений, вызванных применением преобразования (которые могут произойти, например, при повороте рисунка).

Чтобы упростить рассмотренные здесь преобразования цветов, пришлось пойти на определенное ограничение: преобразования работают только с 24-разрядными изображениями. Как говорилось выше, в кодировке RGB на один пиксел такого изображения отводится 3 байт (по байту для каждого цветового канала). Для большей наглядности доступ к цветам пикселов производится только с использованием структуры `TPixel`, которая выглядит следующим образом:

```
struct TPixel
{
  BYTE B;
  BYTE G;
  BYTE R;
};
```

Кстати, в этой структуре отображается действительная последовательность, в которой хранятся данные цветовых каналов в битовой матрице.

## Отражение

Для начала ознакомьтесь с эффектом, который заключается в отражении рисунка по горизонтали и/или вертикали. Пример последовательного применения горизонтального и вертикального отражения показан на рис. 2.11.


**Рис. 2.11.** Отражение рисунка по горизонтали и вертикали

Реализация этого преобразования очевидна: при отражении по горизонтали нужно поменять местами точки левее и правее центра рисунка, а при отражении по вертикали поменять местами точки выше и ниже центра рисунка. Если нужно отразить рисунок и по горизонтали, и по вертикали, то последовательно выполняются два отражения. Текст реализации эффекта отражения приведен в листинге 2.43.

**Листинг 2.43.** Отражение рисунка

```
bool TReflectTransition::Execute(Graphics::TBitmap *image)
{
  assert( image->PixelFormat == pf24bit );
  static bool bHorz = false, bVert = false;
  if ( !frmReflect->Execute(bHorz, bVert) )
    return false;
  else if ( !bHorz && !bVert )
    return true; //Ничего делать не надо
  if ( bHorz )
  {
    //Выполнение отражения по горизонтали
    TPixel buff;
    for ( int y=0; y<image->Height; y++ )
    {
      TPixel *line = (TPixel*)image->ScanLine[y];
      for ( int x=0; x<image->Width/2; x++ )
      {
        buff = line[x];
        line[x] = line[image->Width - x - 1];
        line[image->Width - x - 1] = buff;
      }
    }
  }
  if ( bVert )
  {
    //Выполнение отражения по вертикали (операцию можно
    //оптимизировать, если
    //перемещать строки рисунка целиком, а не по отдельности
    //каждую точку)
    TPixel *buff = new TPixel[image->Width];
    long bytes_count = sizeof(TPixel)*image->Width;
    for ( int y=0; y<image->Height/2; y++ )
    {
      TPixel *first_line = (TPixel*)image->ScanLine[y];
      TPixel *last_line = (TPixel*)image->ScanLine[image->
Height - y - 1];
      memcpy(buff, first_line, bytes_count);
      memcpy(first_line, last_line, bytes_count);
      memcpy(last_line, buff, bytes_count);
    }
    delete buff;
  }
  return true;
}
```

Единственной особенностью приведенного в листинге 2.43 способа реализации отражения является то, что при отражении по вертикали была применена небольшая хитрость — менялись местами не отдельные точки рисунка, а целые строки (скан-линии).

В самом начале выполнения `TReflectTransition::Execute` показывается форма `frmReflect`, с помощью которой пользователь может выбрать способ отражения рисунка, для чего на форму помещены два флажка (рис. 2.12).

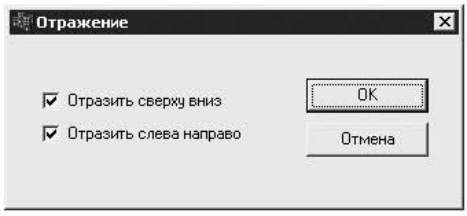


**Рис. 2.12.** Форма настройки отражения

Описание метода `Execute` этой формы и обработчиков кнопок OK и Отмена приведено в листинге 2.44.

**Листинг 2.44.** Реализация формы frmReflect

```
bool TfrmReflect::Execute(bool &bHorz, bool &bVert)
{
  chkLeftToRight->Checked = bHorz;
  chkUpToDown->Checked = bVert;
  if ( ShowModal() == 1 )
  {
    bHorz = chkLeftToRight->Checked;
    bVert = chkUpToDown->Checked;
    return true;
  }
  else
    return false;
}
void __fastcall TfrmReflect::cmbOKClick(TObject *Sender)
{
  ModalResult = 1;
}
void __fastcall TfrmReflect::cmbCancelClick(TObject *Sender)
{
  ModalResult = 2;
}
```

Формы для настройки приведенных ниже преобразований реализуются и ведут себя аналогичным образом. Как видите, поведение таких форм программируется очень просто, потому заострять внимание на этом больше не будем.

## Поворот

Одной из наиболее часто применяемых к изображениям операций является поворот (рис. 2.13).

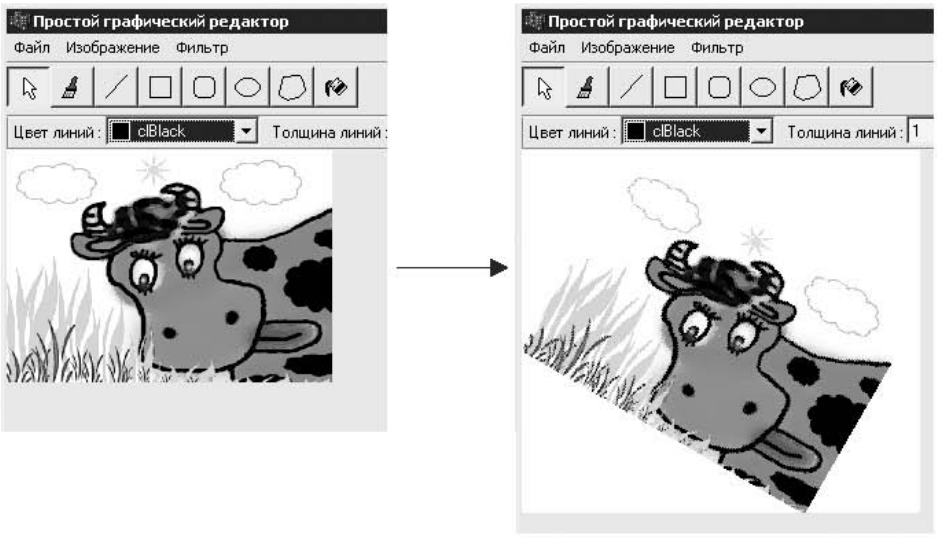


**Рис. 2.13.** Поворот изображения на 30°

Запрограммировать поворот на 90°, 180 или 270° очень просто, а вот для того, чтобы повернуть изображение на любой другой угол, придется произвести небольшие расчеты.

Пусть исходная точка имела координаты ($x_0$; $y_0$). Тогда, чтобы определить новое положение точки после поворота изображения на угол $\alpha$ вокруг центра координат, достаточно применить следующие формулы:

$$x = x_0 \times \cos \alpha - y_0 \times \sin \alpha$$
$$y = x_0 \times \sin \alpha + y_0 \times \cos \alpha$$

Однако то, что хорошо в теории, не всегда удается на практике. Так, если применить приведенные формулы напрямую, может сложиться ситуация, при которой для некоторых точек будет получено одно и то же конечное значение координат, а некоторые пикселы нового изображения останутся незакрашенными. Произойдет это отнюдь не из-за погрешности расчетов, а лишь потому, что растровое изображение дискретно (состоит из конечного набора точек), и если в результате расчета, например, получится, что нужно закрасить пиксел с координатами (125,5; 136,6), то придется каким-то образом округлить значения координат. Вот и получится, что в одни точки нового изображения попало три-четыре исходные точки, а в другие — ни одной.

Простейшим выходом из положения является применение обратного алгоритма поворота — в этом случае вычисляется размер изображения после поворота, а затем

уже для каждой точки нового изображения проверяется, какой точке исходного изображения она соответствует. Для этого применяются следующие формулы (по сути, это формулы для поворота вокруг центра координат на угол $\alpha$):

$$x_0 = x \times \cos\alpha - y \times \sin\alpha$$
$$y_0 = -x \times \sin\alpha + y \times \cos\alpha$$

Недостатком этого простейшего подхода является то, что некоторые точки исходного изображения при повороте потеряются по той же причине, которая была описана выше, но вид повернутого изображения в этом случае будет уже значительно лучше.

Текст реализации алгоритма поворота приведен в листинге 2.45.

**Листинг 2.45.** Поворот изображения

```
bool TRotateTransition::Execute(Graphics::TBitmap *image)
{
  static int angle = 90;
  if ( frmRotate->Execute(angle) )
  {
    //Поворот изображения вокруг левой верхней точки по часовой
    //стрелке
    //..расчет положения углов изображения после поворота
    float cosA = cos( M_PI * angle / 180 );
    float sinA = sin( M_PI * angle / 180 );
    int x2 = image->Width * cosA;
    int y2 = image->Width * sinA;
    int x3 = image->Width * cosA - image->Height * sinA;
    int y3 = image->Width * sinA + image->Height * cosA;
    int x4 = -image->Height * sinA;
    int y4 = image->Height * cosA;
    //..определение размера изображения и координат точки,
    //принимаемой за
    //  начало координат полученного изображения — точки (xmin,
    //ymin)
    int xmin = min(0, min(x2, min(x3, x4)));
    int xmax = max(0, max(x2, max(x3, x4)));
    int ymin = min(0, min(y2, min(y3, y4)));
    int ymax = max(0, max(y2, max(y3, y4)));
    ///..подготовка к копированию с поворотом
    Graphics::TBitmap *src_bmp = new Graphics::TBitmap;
    src_bmp->Assign(image);
    image->Width = xmax - xmin;
    image->Height = ymax - ymin;
    //..собственно поворот
    TPixel whitePixel;
    whitePixel.R = whitePixel.G = whitePixel.B = 255; //Это
```

```
будет цвет фона
    for ( int y=0; y<image->Height; y++ )
    {
      for ( int x=0; x<image->Width; x++ )
      {
        //Расчет исходных координат точки на неповернутом
        //изображении,
        //имеющей координаты (x, y) на повернутом
        int x0 = (x+xmin) * cosA + (y+ymin) * sinA;
        int y0 = -(x+xmin) * sinA + (y+ymin) * cosA;
        if ( x0 >= 0 && x0 < src_bmp->Width &&
             y0 >= 0 && y0 < src_bmp->Height )
          ((TPixel*)image->ScanLine[y])[x] =
           ((TPixel*)src_bmp->ScanLine[y0])[x0];
        else
          ((TPixel*)image->ScanLine[y])[x] = whitePixel;
      }
    }
    delete src_bmp;
    return true;
  }
  else
    return false;
}
```

Код метода `TRotateTransition::Execute` полностью отражает сказанное об использованном методе поворота. Остается лишь добавить, что в качестве цвета фона изображения (см. рис. 2.13) принимается белый цвет.

## Растяжение и сжатие

Еще одной полезной возможностью, которая может быть реализована графическим редактором, является растяжение и сжатие изображения. Пример применения к рисунку эффекта сжатия по горизонтали и растяжения по вертикали приведен на рис. 2.14.

Алгоритм растяжения/сжатия изображения довольно хитрый, но, к счастью, реализовывать его самостоятельно не потребуется — достаточно лишь воспользоваться методом `StretchDraw` класса `TBitmap` (листинг 2.46).

**Листинг 2.46.** Растяжение/сжатие изображения

```
bool TStretchTransition::Execute(Graphics::TBitmap *image)
{
  static int width_per = 100, height_per = 100;
  if ( frmStretch->Execute(width_per, height_per) &&
       (width_per != 100 || height_per != 100 ) )
  {
```

```
    //Изменение размера изображения
    Graphics::TBitmap *src_bmp = new Graphics::TBitmap;
    src_bmp->Assign(image);
    image->Width = image->Width * width_per / 100;
    image->Height = image->Height * height_per / 100;
    image->Canvas->StretchDraw(
     TRect(0,0,image->Width,image->Height), src_bmp);
    delete src_bmp;
    return true;
  }
  else
    return false;
}
```

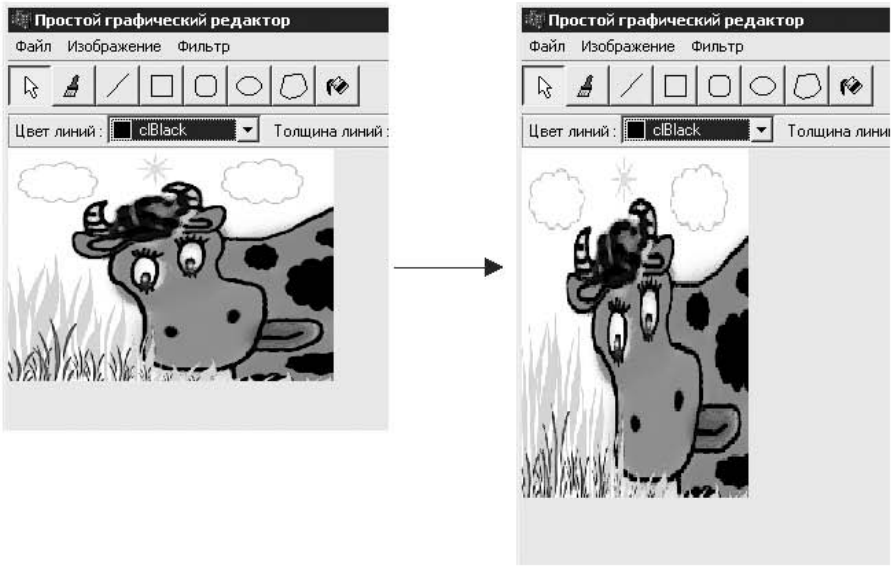


**Рис. 2.14.** Сжатие изображения по горизонтали и растяжение по вертикали

Думается, больше говорить об этом не имеет смысла — остается лишь использовать этот эффект и радоваться легкой программистской «добыче».

## Инверсия цветов

Теперь можно переходить к рассмотрению эффектов, манипулирующих цветами пикселов рисунка, и первым следует эффект инверсии цветов, результат применения которого показан на рис. 2.15.

Как вычислить обратный цвет для каждого пиксела? Да очень просто, в чем можно легко убедиться, ознакомившись с текстом реализации этого эффекта, приведенным в листинге 2.47.

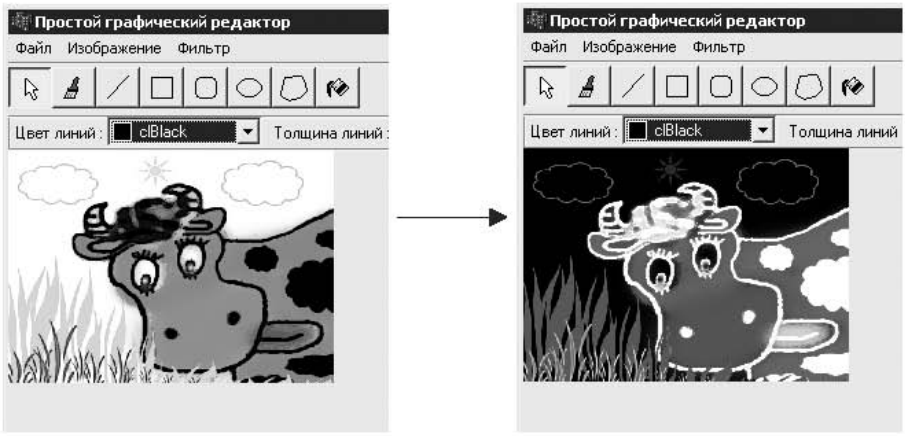


**Рис. 2.15.** Инверсия цветов

**Листинг 2.47.** Инвертирование цветов

```
bool TInversionTransition::Execute(Graphics::TBitmap *image)
{
  for ( int y=0; y<image->Height; y++ )
  {
    TPixel *line = (TPixel*)image->ScanLine[y];
    for ( int x=0; x<image->Width; x++ )
    {
      line[x].R = 255 - line[x].R;
      line[x].G = 255 - line[x].G;
      line[x].B = 255 - line[x].B;
    }
  }
  return true;
}
```

Вообще при реализации этого эффекта можно было обойтись и без приведения строк рисунка к указателю на `TPixel`, и без манипулирования отдельными цветовыми каналами — можно было просто выполнить операцию 255 – x к каждому байту рисунка, где x — значение интенсивности канала цвета.

## Черно-белое изображение

Трудно ли цветное изображение преобразовать в черно-белый вариант? Очень просто. Этот эффект уже был почти реализован в начале главы при создании градаций серого цвета. Как вы, возможно, помните, тогда было сказано, что человеческий глаз имеет различную восприимчивость к красному, зеленому и синему цветам. Чтобы это учесть, достаточно вместо среднего арифметического значения интенсивностей каналов использовать их взвешенную сумму, как показано в листинге 2.48.

**Листинг 2.48.** Преобразование цветного изображения в черно-белое

```
bool TGrayScaleTransition::Execute(Graphics::TBitmap *image)
{
  BYTE gray;
  for ( int y=0; y<image->Height; y++ )
  {
    TPixel *line = (TPixel*)image->ScanLine[y];
    for ( int x=0; x<image->Width; x++ )
    {
      //Преобразование цветов в оттенки серого
      gray = 0.3*line[x].R + 0.59*line[x].G + 0.11*line[x].B;
      line[x].R = line[x].G = line[x].B = gray;
    }
  }
  return true;
}
```

По коэффициентам (которые не мной были придуманы) можно судить, что лучше всего человеческий глаз воспринимает зеленый цвет и хуже всего — синий.

## Изменение яркости

Изменение общей яркости рисунка (рис. 2.16) является еще одним эффектом, способ реализации которого практически очевиден.

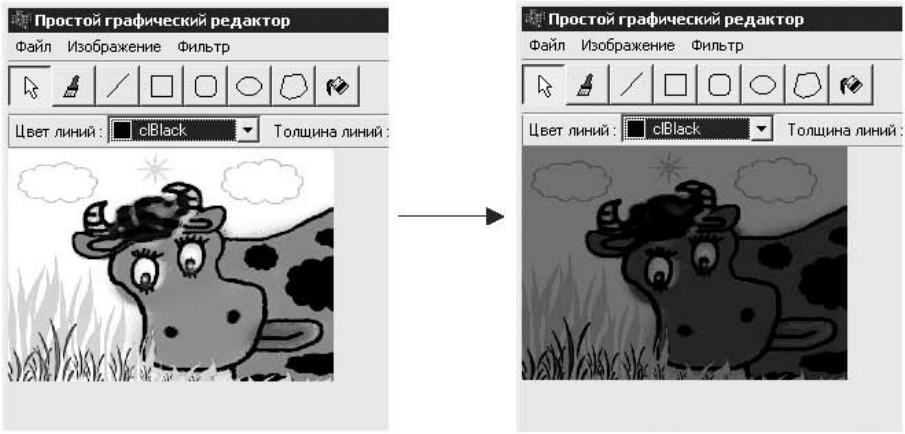


**Рис. 2.16.** Изменение яркости

Для изменения яркости рисунка в целом достаточно прибавить к значениям интенсивностей цветовых составляющих каждого пиксела либо отнять от этих значений определенное (одинаковое для всех каналов) число (листинг 2.49). При этом остается только контролировать, чтобы полученные в результате значения не выходили за заданные пределы (от 0 до 255).

**Листинг 2.49.** Изменение яркости

```
bool TBrightnessTransition::Execute(Graphics::TBitmap *image)
{
  static int delta_br = 0;
  if ( frmBrightness->Execute(delta_br) )
  {
    //Изменение интенсивности каждого канала на delta_br
    for ( int y=0; y<image->Height; y++ )
    {
      TPixel *line = (TPixel*)image->ScanLine[y];
      for ( int x=0; x<image->Width; x++ )
      {
        if ( delta_br > 0 )
        {
          line[x].R =
           (delta_br + line[x].R > 255) ? 255 : delta_br +
line[x].R;
          line[x].G =
           (delta_br + line[x].G > 255) ? 255 : delta_br +
line[x].G;
          line[x].B =
           (delta_br + line[x].B > 255) ? 255 : delta_br +
line[x].B;
        }
        else
        {
          line[x].R = (delta_br + line[x].R < 0) ? 0 :
delta_br + line[x].R;
          line[x].G = (delta_br + line[x].G < 0) ? 0 :
delta_br + line[x].G;
          line[x].B = (delta_br + line[x].B < 0) ? 0 :
delta_br + line[x].B;
        }
      }
    }
    return true;
  }
  else
   return false;
}
```

## Смешивание изображений

Эффект смешивания изображений (или муар) также основан на несложных манипуляциях с цветами отдельных пикселов каждого изображения, но в результате

применения этого несложного эффекта получается очень даже интересный результат (рис. 2.17).

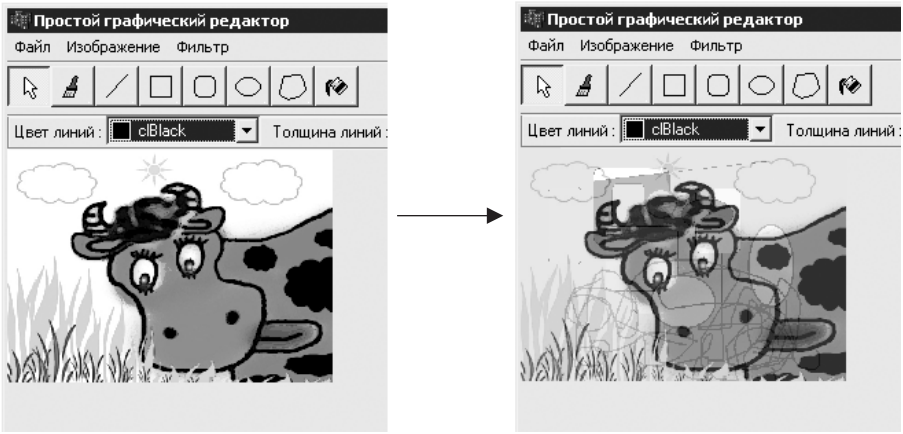


**Рис. 2.17.** Смешивание изображений

Текст реализации эффекта смешивания приведен в листинге 2.50.

**Листинг 2.50.** Смешивание изображений

```
bool TMixingTransition::Execute(Graphics::TBitmap *image)
{
  Graphics::TBitmap *another_image = new Graphics::TBitmap;
  static int mix_percent = 50; //Процент «примеси» другого
                               //изображения
  if ( frmMixing->Execute(another_image, mix_percent) )
  {
    another_image->PixelFormat = pf24bit; //Работа
                                          //осуществляется
                                          //с 24-битными
                                          //изображениями
    if ( another_image->Width != image->Width ||
         another_image->Height != image->Height )
    {
      //Выбранное изображение создается такого же размера, как
      //и показанное
      //в редакторе
      Graphics::TBitmap *temp = new Graphics::TBitmap;
      temp->Assign(another_image);
      another_image->Width = image->Width;
      another_image->Height = image->Height;
      another_image->Canvas->StretchDraw(
        TRect(0,0,another_image->Width,image->Height), temp);
      delete temp;
    }
```

```
    //Собственно смешивание
    TPixel pix1, pix2, pix;
    float p1 = (100 - mix_percent) / 100.0; //Доля исходного
                                            //изображения
    float p2 = mix_percent / 100.0;       //Доля примешиваемого
                                          //изображения
    for ( int y=0; y<image->Height; y++ )
    {
      for ( int x=0; x<image->Width; x++ )
      {
        pix1 = ((TPixel*)image->ScanLine[y])[x];
        pix2 = ((TPixel*)another_image->ScanLine[y])[x];
        pix.R = p1 * pix1.R + p2 * pix2.R;
        pix.G = p1 * pix1.G + p2 * pix2.G;
        pix.B = p1 * pix1.B + p2 * pix2.B;
        ((TPixel*)image->ScanLine[y])[x] = pix;
      }
    }
    delete another_image;
    return true;
  }
  else
  {
    delete another_image;
    return false;
  }
}
```

Понятно, что, кроме исходного рисунка, потребуется еще и второе изображение, при этом примешиваемое изображение должно быть такого же размера, как и исходное. Контролю над выполнением этого условия посвящена значительная часть кода, приведенного в листинге 2.50.

Суть же самого эффекта состоит в вычислении взвешенной суммы цветовых составляющих исходного и нового изображений. Весовые коэффициенты (`p1` и `p2`) вычисляются на основе того, какую часть в итоговом изображении (в процентах) должны составлять данные примешиваемого изображения.

Для выбора примешиваемого изображения и указания его доли в получаемом в результате изображении применяется форма `frmMixing`, показанная на рис. 2.18.

Текст реализации этой формы приводится в листинге 2.51.

**Листинг 2.51.** Форма для настройки эффекта «смешивание»

```
void __fastcall TfrmMixing::cmbBrowseClick(TObject *Sender)
{
  if ( openDlg->Execute() )
  {
```

```
    try
    {
      img->Picture->LoadFromFile(openDlg->FileName);
      cmbOK->Enabled = true;
    }
    catch(EInvalidGraphic &)
    {
      ::MessageBox(Handle, "Ошибка при загрузке рисунка",
"Ошибка",
                   MB_ICONEXCLAMATION);
    }
  }
}
void __fastcall TfrmMixing::cmbCancelClick(TObject *Sender)
{
  ModalResult = 2;
}
void __fastcall TfrmMixing::cmbOKClick(TObject *Sender)
{
  try
  {
    int percent = txtPercent->Text.ToInt();
    if ( percent > 100 || percent < 0 )
    {
      ::MessageBox(Handle, "Число должно быть от 0 до 100",
                   "Информация", MB_ICONINFORMATION);
      return;
    }
    ModalResult = 1;
  }
  catch(EConvertError &)
  {
    ::MessageBox(Handle, "Введите число в текстовое поле",
                 "Информация", MB_ICONINFORMATION);
  }
}
bool TfrmMixing::Execute(Graphics::TBitmap *image, int &per-
cent)
{
  txtPercent->Text = percent;
  if ( ShowModal() == 1 )
  {
    percent = txtPercent->Text.ToInt();
    image->Assign(img->Picture->Bitmap);
    return true;
  }
```

```
  else
    return false;
}
```

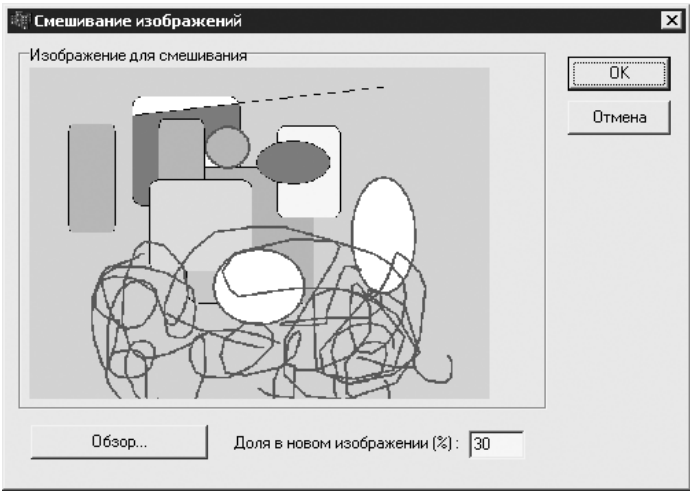


**Рис. 2.18.** Форма настройки эффекта «смешивание»

На самой форме компонент, который используется для отображения выбранного изображения, имеет имя `img`, а текстовое поле, предназначенное для указания доли выбранного изображения при смешивании, — `txtPercent`.

## Добавление фильтров

В завершение будут рассмотрены три несложных фильтра: размытие (усредняющий фильтр), удаление помех (медианный фильтр) и «тиснение». Такие фильтры часто называют матричными, поскольку они основаны на вычислении цвета пиксела в зависимости от цветов окружающих его пикселов.

В данном случае реализованы фильтры, в которых рассматривается цвет текущего пиксела и восьми соседних, то есть используется матрица 3 × 3, которая, как окно, «скользит» по поверхности изображения. Цвет каждого пиксела, попавшего в центр такой матрицы, вычисляется по определенным правилам в зависимости от цветов остальных элементов матрицы. Конкретные правила вычисления зависят от выбора фильтра.

Для начала нужно создать новый класс, который будет базовым для всех приведенных матричных фильтров (листинг 2.52).

**Листинг 2.52.** Базовый класс матричных фильтров

```
class TMatrixFilter : public TEditorTransition
{
public:
  virtual bool Execute(Graphics::TBitmap *image);
```

```
protected:
  virtual TPixel GetColor(TPixel **frame) = 0;//Должны быть
                                              //реализованы
                                              //дочерние
                                              //классы
};
```

Как видите, класс `TMatrixFilter` является абстрактным. Чтобы реализовать конкретный матричный фильтр, придется унаследовать класс фильтра от `TMatrixFilter` и реализовать один-единственный метод `GetColor`, который и должен вычислять по нужным правилам цвет центрального пиксела в матрице, передаваемой параметром `frame`.

Алгоритм формирования самой матрицы и создания итогового изображения уже реализован в классе `TMatrixFilter` (листинг 2.53).

**Листинг 2.53.** Базовая реализация метода Execute для матричных фильтров

```
bool TMatrixFilter::Execute(Graphics::TBitmap *image)
{
  Graphics::TBitmap *src_bmp = new Graphics::TBitmap;
  src_bmp->Assign(image);
  //Применение матричного фильтра 3 × 3
  TPixel *frame[3];
  for ( int y=1; y<image->Height-1; y++ )
  {
    for ( int x=1; x<image->Width-1; x++ )
    {
      frame[0] = ((TPixel*)src_bmp->ScanLine[y-1]) + x - 1;
      frame[1] = ((TPixel*)src_bmp->ScanLine[y]) + x - 1;
      frame[2] = ((TPixel*)src_bmp->ScanLine[y+1]) + x - 1;
      ((TPixel*)image->ScanLine[y])[x] = GetColor(frame);
    }
  }
  delete src_bmp;
  return true;
}
```

При реализации матричных фильтров одним затруднительным моментом является обработка крайних точек рисунка. В алгоритме реализации, приведенном в листинге 2.53, крайние точки попросту не обрабатываются, то есть в действительности фильтр применяется к изображению на два пиксела по горизонтали и два пиксела по вертикали меньше исходного. Это сделано для того, чтобы не усложнять алгоритм, приводимый в книге.

Как вариант, для решения данной проблемы в алгоритм можно добавить процедуру проверки пикселов на предмет того, являются ли они крайними. Если да, то значения цветов недостающих пикселов могут быть получены путем копирования соот-

ветствующих значений из последней или предпоследней строки (столбца), но тогда придется также отдельно рассматривать и угловые точки исходного изображения.

## Размытие

Наконец, можно переходить к реализации фильтров. В листинге 2.54 приведен код усредняющего фильтра, предлагающий наиболее простой способ реализации эффекта размытия.

**Листинг 2.54.** Усредняющий фильтр

```
TPixel TBlurFilter::GetColor(TPixel **frame)
{
  //Результирующий цвет точки — среднее арифметическое цветов
  //точек
  //в окне 3×3
  int r=0, g=0, b=0;
  for ( int x=0; x<3; x++ )
  {
    for ( int y=0; y<3; y++ )
    {
      r += frame[y][x].R;
      g += frame[y][x].G;
      b += frame[y][x].B;
    }
  }
  TPixel result;
  result.R = r/9;
  result.G = g/9;
  result.B = b/9;
  return result;
}
```

Результат применения усредняющего фильтра показан на рис. 2.19.

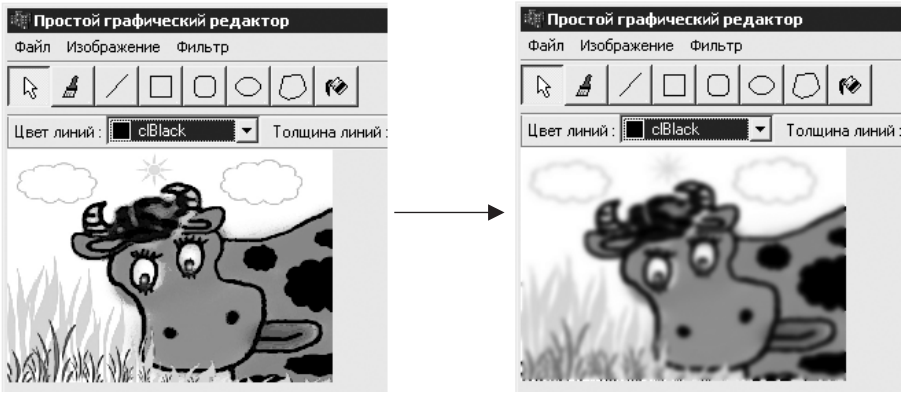


**Рис. 2.19.** Размытие

Суть действия приведенного фильтра понятна из его названия: цвет центральной точки матрицы вычисляется как среднее арифметическое значение цветов пикселов матрицы.

## Удаление помех

Следующий фильтр часто хорошо справляется с удалением помех (случайных точек, полос), что видно на рис. 2.20.


**Рис. 2.20.** Удаление помех

Этот фильтр называют медианным, поскольку составляющие цвета центральной точки вычисляются как медиана соответствующих составляющих цветов этой точки и точек, расположенных рядом. К сожалению, медианный фильтр удаляет с изображения и мелкие детали (что также хорошо видно на рис. 2.20), поэтому не любое изображение может быть успешно обработано этим фильтром.

**ПРИМЕЧАНИЕ**

Медиана — элемент, расположенный в середине отсортированного массива.

В листинге 2.55 приведен текст реализации медианного фильтра.

**Листинг 2.55.** Медианный фильтр

```
TPixel TMedianFilter::GetColor(TPixel **frame)
{
  //Результирующий цвет точки — медиана цветов точек в окне
  //3 × 3
  //..составление массивов цветов точек в окне (по каждому
  //каналу отдельно)
  BYTE r_vals[9], g_vals[9], b_vals[9];
  for ( int x=0; x<3; x++ )
  {
```

```
    for ( int y=0; y<3; y++ )
    {
      r_vals[x*3 + y] = frame[x][y].R;
      g_vals[x*3 + y] = frame[x][y].G;
      b_vals[x*3 + y] = frame[x][y].B;
    }
  }
  //..сортировка массивов и нахождение медиан
  sort(r_vals, r_vals+9);
  sort(g_vals, g_vals+9);
  sort(b_vals, b_vals+9);
  TPixel result;
  result.R = r_vals[4];
  result.G = g_vals[4];
  result.B = b_vals[4];
  return result;
}
```

## Тиснение

В завершение будет рассмотрен еще один простой матричный фильтр, создающий эффект тиснения (рис. 2.21).

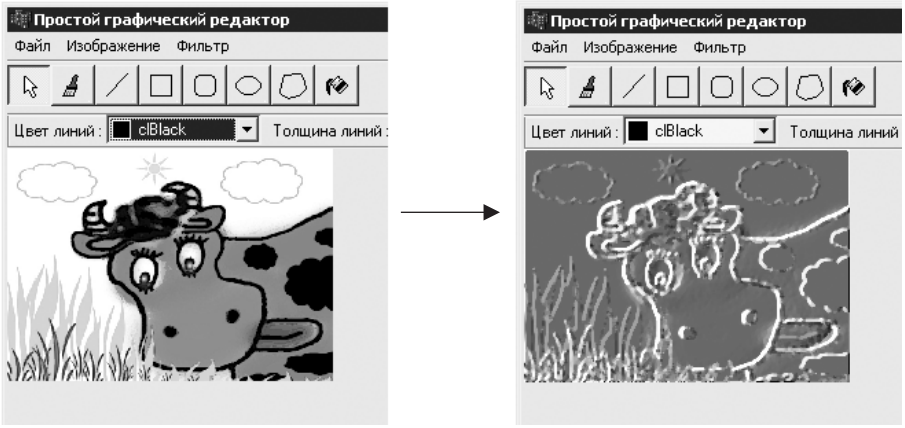


**Рис. 2.21.** Тиснение

При создании эффекта тиснения цветовые составляющие центрального пиксела вычисляются на основе цветов соседних пикселов (листинг 2.56).

**Листинг 2.56.** Фильтр «тиснение»

```
TPixel TEmbossFilter::GetColor(TPixel **frame)
{
  int r = frame[0][0].R + frame[0][1].R - frame[0][2].R +
          frame[1][0].R                 - frame[1][2].R +
          frame[2][0].R - frame[2][1].R - frame[2][2].R;
```

```
  int g = frame[0][0].G + frame[0][1].G – frame[0][2].G +
          frame[1][0].G                 – frame[1][2].G +
          frame[2][0].G - frame[2][1].G – frame[2][2].G;
  int b = frame[0][0].B + frame[0][1].B – frame[0][2].B +
          frame[1][0].B                 – frame[1][2].B +
          frame[2][0].B - frame[2][1].B – frame[2][2].B;
  TPixel result;
  result.R = (r+128 > 255) ? 255 : r + 128;
  result.G = (g+128 > 255) ? 255 : g + 128;
  result.B = (b+128 > 255) ? 255 : b + 128;
  return result;
}
```

На этом придется закончить и так получившуюся очень длинной главу о возможностях использования графики в программах, работающих на C++ и разработанных в среде Borland C++ Builder. Конечно, тут представлена лишь очень малая часть приемов и алгоритмов, применяемых при работе с графикой, но хочется верить, что хоть что-нибудь из представленного пригодится вам на практике.

# Глава 3

## Меню и графические списки

- Меню
- Графические списки

Эта глава частично продолжает предыдущую. Значительную ее часть составляют примеры использования некоторых из показанных ранее приемов рисования, которые применяются при создании довольно эффектных меню и компонентов. Но все это содержится преимущественно во второй части главы. В первых же ее разделах рассказывается о некоторых приемах работы с такой частью интерфейса приложений, как меню.

# Меню

Меню, наверное, является одним из старейших средств обработки пользовательских команд ввода в программах с графическим интерфейсом. Пользоваться хорошо организованным меню просто и удобно, поэтому приложения редко обходятся без него (не считая, конечно, совсем простых приложений, формы которых состоят, образно говоря, из текстового поля и двух кнопок).

Здесь в предложенных подразделах приведены несколько примеров, демонстрирующих несколько необычных способов использования меню в приложениях, реализованных в среде Borland C++ Builder.

## Добавление пунктов в системное меню

Меню является стандартным для Windows средством ввода. В связи с этим в Windows API предусмотрена целая группа функций, направленных на реализацию методов работы с меню, а сам довольно специфический процесс вывода меню на экран реализован самой операционной системой. По этой причине при разработке прикладных программ по большому счету нужно заботиться только о наполнении меню и обработке поступающих от него команд.

Для многих окон щелчком кнопкой мыши на значке, который расположен на полосе заголовка, можно вызвать так называемое системное меню окна. В нем обычно находятся команды, обеспечивающие стандартное поведение окна: Свернуть, Развернуть, Закрыть и др.

Используя функции Windows API, предназначенные для манипулирования меню, можно легко изменять системное меню, например, добавляя в него свои подменю. Собственно, на реализацию данной возможности и направлен код, приведенный в листинге 3.1.

**Листинг 3.1.** Добавление пунктов в системное меню формы

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  //Получаем системное меню окна
  HMENU hSysMenu = ::GetSystemMenu(Handle, false);
  //Создаем подменю средствами Windows API
  HMENU hSubMenu = ::CreatePopupMenu();
  ::AppendMenu( hSubMenu, MF_ENABLED, COMMAND_1, "Мой
пункт 1" );
```

```
  ::AppendMenu( hSubMenu, MF_DISABLED | MF_GRAYED, COMMAND_
2,"Мой пункт 2");
  ::AppendMenu( hSubMenu, MF_SEPARATOR, 0, "" );
  ::AppendMenu( hSubMenu, MF_ENABLED, COMMAND_3, "Мой пункт
3" );
  //Добавим разделитель и подменю в системное меню окна
  ::AppendMenu( hSysMenu, MF_SEPARATOR, 0, "" );
  ::AppendMenu( hSysMenu, MF_POPUP, (UINT)hSubMenu, "Мое
подменю" );
}
```

Созданное подменю будет выглядеть приблизительно так, как показано на рис. 3.1.

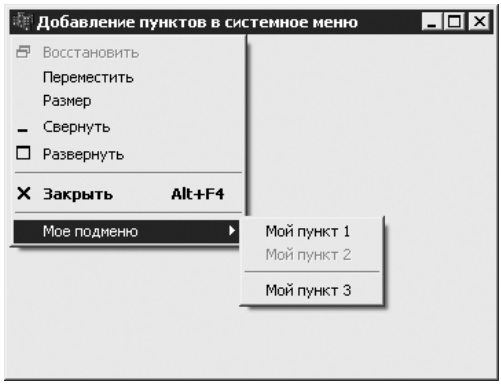


**Рис. 3.1.** Собственное подменю в системном меню окна

Обратите внимание, что в листинге 3.1 системные функции оперируют так называемым дескриптором меню (`HMENU`). С самим же пунктом меню связаны по меньшей мере стиль или флаг (в листинге 3.1 это константы, начинающиеся с `MF_`). Для пунктов меню, не являющихся разделителями, важен также текст, а вот самым важным, наверное, атрибутом меню все же является идентификатор (номер команды), по которому и определяется то, какой именно пункт меню выбрал пользователь.

По понятным причинам идентификатор каждого пункта меню должен быть уникальным в приложении. Так, используемые в листинге 3.1 идентификаторы пунктов меню объявлены следующим образом:

```
const UINT COMMAND_1 = 1000;
const UINT COMMAND_2 = 1001;
const UINT COMMAND_3 = 1002;
```

Однако мало лишь создать меню — нужно еще обработать команду выбора его новых пунктов. Для этого можно написать обработчик сообщения `WM_SYSCOMMAND`, пример объявления которого приведен в листинге 3.2.

**Листинг 3.2.** Объявление обработчика сообщения WM_SYSCOMMAND

```
class TForm1 : public TForm
{
  ...
  void __fastcall WMSysCommand(Messages::TWMSysCommand &com-
mand);
  BEGIN_MESSAGE_MAP
    VCL_MESSAGE_HANDLER(WM_SYSCOMMAND, TWMSysCommand, WMSys-
Command);
  END_MESSAGE_MAP(TForm)
};
```

Как видите, помимо объявления обработчика `WMSysCommand`, потребовалось также прописать его в так называемой карте сообщений. Текст же реализации обработчика `WMSysCommand` приведен в листинге 3.3.

**Листинг 3.3.** Обработка команд системного меню

```
void __fastcall TForm1::WMSysCommand(Messages::TWMSysCommand
&command)
{
  switch ( command.CmdType )
  {
  case COMMAND_1:
    Application->MessageBox( "Выбран пункт меню 1",
"Системное меню", 0 );
    command.Result = true;
    break;
  case COMMAND_2:
    Application->MessageBox( "Выбран пункт меню 2",
"Системное меню", 0 );
    command.Result = true;
    break;
  case COMMAND_3:
    Application->MessageBox( "Выбран пункт меню 3",
"Системное меню", 0 );
    command.Result = true;
    break;
  default:
    command.Result = ::DefWindowProc( Handle, command.Msg,
command.CmdType,

65536 * command.YPos + command.XPos );
  }
}
```

Вряд ли есть что-то сложное в реализации приведенного в листинге 3.3 обработчика: здесь по коду выбранной команды меню определяется действие, которое

должно быть произведено. Единственное, что может показаться странным, так это необходимость вызова API-функции `DefWindowProc`. Она выполняется для того, чтобы из меню не исчезли системные команды, такие как Развернуть, Свернуть и т. д. Вызываемый системный обработчик сообщений позволяет обеспечить правильную реакцию на выбор этих пунктов меню.

## Динамическое создание меню

При программировании с использованием среды Borland C++ Builder в создании меню для формы или всплывающего меню нет ничего сложного — для этого достаточно лишь поместить на форму соответственно компонент MainMenu (меню формы) или PopupMenu (всплывающее меню), воспользоваться редактором меню для создания необходимых пунктов меню и написать соответствующие обработчики. Такой подход настолько удобен и прост, что вряд ли в состав главы стоит включать пример его применения, но создавать меню и обрабатывать события, поступающие от него, можно и несколько другим образом.

Иногда формировать часть меню или все его целиком удобнее в ходе выполнения программы. В качестве примера можно привести приложение, использующее постоянно изменяющийся или расширяющийся набор инструментов и функций — так называемых плагинов. В ходе разработки такого приложения могут быть неизвестны количество и тип реализованных в конечном счете инструментов, тем более что при публикации программного интерфейса, который должен реализовывать каждый плагин, за их реализацию могут приняться и сторонние разработчики.

Здесь приводится небольшой пример того, как можно создать меню в ходе выполнения программы и связать созданные пункты меню с необходимыми действиями. Для упрощения примера в качестве действия, выполняемого созданными «на лету» пунктами меню, выбран запуск функции из библиотеки DLL (имитация запуска плагинов).

Структура меню и связанные с его пунктами плагины в данном примере задаются в файле `plugins.ini`, который может выглядеть так, как показано в листинге 3.4.

**Листинг 3.4.** Задание структуры меню

```
[MAIN]
PluginsCount=3
[PLUGIN1]
MenuName=Модули\Дополнительные модули\Модуль 1
File=plugins\plugin1\plugin1.dll
[PLUGIN2]
MenuName=Модули\Дополнительные модули\Модуль 2
File=plugins\plugin2\plugin2.dll
[PLUGIN3]
MenuName=Модули\Модуль 3
File=plugins\plugin3\plugin3.dll
```

Единственный параметр секции MAIN определяет количество плагинов (пунктов) меню. Далее следует нужное количество секций, содержащих сведения о названии пункта меню (MenuName) и пути к DLL (File), в которой реализован плагин.

При обработке приведенного в листинге 3.4 файла создается иерархия меню, показанная на рис. 3.2.

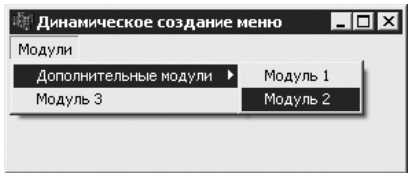


**Рис. 3.2.** Созданное по описанию в INI-файле меню

Реализация выглядит достаточно просто: в обработчике FormCreate формы открывается файл INI, содержащий описание меню, и в цикле создается меню для каждого указанного в файле плагина (листинг 3.5).

**Листинг 3.5.** Обработчик создания формы

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  //Загрузка описаний плагинов из INI-файла
  TIniFile *ini_file =
   new TIniFile(ExtractFilePath(Application->ExeName) +
"plugins.ini");
  int plugins_count = ini_file->ReadInteger( "MAIN",
"PluginsCount", 0 );
  for ( int p=1; p<=plugins_count; p++ )
  {
    String section = "PLUGIN" + IntToStr(p);
    String menu_name = ini_file->ReadString( section,
"MenuName", "" );
    String file_name = ini_file->ReadString( section, "File",
"" );
    CreateMenuItemForPlugin( menu_name, file_name );
  }
  delete ini_file;
}
```

Используемая в листинге 3.5 функция CreateMenuItemForPlugin выглядит немного сложнее (листинг 3.6).

**Листинг 3.6.** Создание иерархии пунктов меню

```
void TForm1::CreateMenuItemForPlugin( const String &menu_
name,
                                      const String &file_
name )
{
  //Сначала выделим названия корневого меню и остальных
  //подменю
  vector<String> names;
```

```
  StringToParts( menu_name, '\\', names );
  if ( names.size() == 0 ) return;
  //Найдем или создадим корневое меню
  TMenuItem *submenu = NULL;
  for ( int m=0; m<menu->Items->Count; m++ )
  {
    if ( (*menu->Items)[m]->Caption == names[0] )
      submenu = (*menu->Items)[m]; //Уже есть такое корневое
                                   //меню
  }
  if ( submenu == NULL )
  {
    //Нет корневого меню с нужным названием — создадим его
    submenu = new TMenuItem(menu);
    submenu->Caption = names[0];
    menu->Items->Add( submenu );
  }
  //Создадим иерархию подменю
  TMenuItem *item = submenu;
  for ( unsigned int i=1; i<names.size(); i++ )
  {
    //Ищем i-е подменю
    item = NULL;
    for ( int s=0; s<submenu->Count; s++ )
    {
      if ( submenu->Items[s]->Caption == names[i] )
      {
        item = submenu->Items[s]; //Есть такое подменю
        break;
      }
    }
    if ( item == NULL )
    {
      //Придется создать подменю
      item = new TMenuItem(submenu);
      item->Caption = names[i];
      submenu->Add( item );
    }
    submenu = item;
  }
  //Создадим объект для обработки выбора пункта меню (для
  //запуска плагина)
  item->Action =
   new TPluginAction( item->Caption.c_str(), file_name.c_str(),
item );
}
```

Идея, которую реализует функция CreateMenuItemForPlugin, весьма проста: полное название меню (по сути, путь в иерархии меню) разбивается на составляющие, после чего в существующее меню формы, если есть необходимость, добавляется подменю. Функция разбиения полного имени пункта меню на части приведена в листинге 3.7.

**Листинг 3.7.** Разбиение полного имени меню на составляющие

```
void StringToParts( const String &text, const char delimiter,
                    vector<String> &parts )
{
  int start = 1;
  for ( int i=1; i<=text.Length(); i++ )
  {
    if ( text[i] == delimiter )
    {
      if ( start < i )
        parts.push_back(text.SubString( start, i - start ));
      start = i+1;
    }
  }
  if ( start < text.Length() )
    parts.push_back(text.SubString( start, text.Length() -
start + 1 ));
}
```

В самом конце функции CreateMenuItemForPlugin (листинг 3.6) созданный для плагина пункт меню связывается со своим обработчиком. Для обработки выбора создаваемых пунктов меню используются объекты одного и того же класса TPluginAction, производного от TAction (листинг 3.8).

**Листинг 3.8.** Класс TPluginAction

```
class TPluginAction : public TAction
{
  String m_PluginFileName;
public:
  __fastcall TPluginAction( const char *name,
                            const char *lpszPluginFileName,
                            TComponent* owner );
protected:
  void __fastcall ExecutePlugin(System::TObject* Sender);
};
```

Конструктор класса TPluginAction принимает в качестве аргументов название действия (именно это название отображается в качестве названия пункта меню), путь к файлу DLL плагина и владельца создаваемого объекта (в данном случае это объект класса TMenuItem). Текст реализации класса TPluginAction приведен в листинге 3.9.

**Листинг 3.9.** Реализация класса TPluginAction

```
__fastcall TPluginAction::TPluginAction( const char *name,
                                         const char *filename,
                                         TComponent* owner )
:TAction( owner )
{
  m_PluginFileName = filename;
  Caption = name;
  OnExecute = ExecutePlugin;
}
//Обработчик, вызываемый при выборе пункта меню
void __fastcall TPluginAction::ExecutePlugin(System::TObject*
Sender)
{
  //Запускаем функцию из DLL (применяем плагин)
  //..для этого сначала загружаем DLL
  HMODULE hModule = ::LoadLibrary( m_PluginFileName.c_str()
);
  if ( hModule == NULL )
  {
    String mess = "Файл " + m_PluginFileName;
    mess += " не найден";
    Application->MessageBox( mess.c_str(), "Ошибка", MB_
ICONEXCLAMATION );
    return;
  }
  //..получаем адрес нужной функции (Execute)
  PluginFunction func = (PluginFunction)::GetProcAddress(hMod
ule,"Execute");
  if ( func == NULL )
  {
    String mess = "Не удалось определить адрес функции
Execute";
    Application->MessageBox( mess.c_str(), "Ошибка", MB_
ICONEXCLAMATION );
    return;
  }
  //..запустим функцию из DLL
  func();
  //..выгрузим DLL
  ::FreeModule( hModule );
}
```

Действия, производимые в конструкторе класса `TPluginAction`, должны быть понятны. На всякий случай стоит упомянуть, что в конструкторе функция `ExecutePlugin` регистрируется в качестве обработчика выбора пунктов меню. Сама

функция ExecutePlugin выполняет загрузку DLL, путь к которой был передан в конструктор, и после этого пытается найти и запустить из DLL функцию Execute, указатель на которую объявлен следующим образом:

```
typedef void __stdcall (*PluginFunction)();
```

Для полноты примера ниже приведено описание способа реализации одного плагина. Как было сказано ранее, он реализован в DLL. Чтобы создать DLL, выполните команду меню File ▸ New ▸ Other и в открывшемся окне на закладке New выберите компонент DLL Wizard. В следующем окне (DLL Wizard) никаких флажков устанавливать не нужно. В созданном таким образом проекте вставляется функция Execute, пример которой приведен в листинге 3.10.

**Листинг 3.10.** Реализация плагина

```
extern "C"
{
  __declspec(dllexport) void __stdcall Execute()
  {
    MessageBox(NULL, "Вы запустили модуль 1", "Сообщение",
0);
  }
};
```

Проекты двух плагинов вы можете найти на прилагаемом компакт-диске. Третий специально не реализован, чтобы можно было проверить, правильно ли ведет себя программа при неверном указании пути к плагину.

## Графические меню

Почти любой программист, использующий Borland C++ Builder, знает, что в меню можно с легкостью добавлять значки, делая его и красивее, и нагляднее. Для этого достаточно поместить на форму компонент ImageList и указать его в свойстве Images компонента меню. После этого для пунктов меню можно будет указывать номер значка из ImageList (свойство ImageIndex). Но что делать, если сочетания «значок + надпись» недостаточно, например, если нужно создать меню для выбора стиля линий в графическом редакторе (рис. 3.3)?

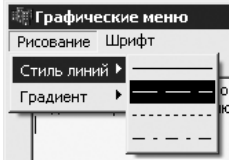


**Рис. 3.3.** Меню Стиль линий

Тут пригодится свойство меню OwnerDraw. Установка в окне Object Inspector в качестве его значения true говорит о том, что при выводе на экран всех или только некоторых пунктов меню разработчик проекта хочет самостоятельно нарисовать их содержимое.

Для реализации такой возможности можно определить обработчики событий OnMeasureItem и OnDrawItem для пунктов меню и реализовать их прямо в классе формы. Обработчик OnMeasureItem должен возвращать ширину и высоту пункта

меню (вызывается до вывода меню на экран), а в обработчике OnDrawItem должны быть предусмотрены действия по выводу изображения пункта меню на экран.

Подход, предполагающий написание обработчиков OnDrawItem и OnMeasureItem прямо в классе формы, конечно, очень прост, но, на мой взгляд, он годится только для задач, когда нужно нарисовать всего два-три пункта меню. В противном случае класс формы разрастется до ужасающих размеров, причем в самом худшем варианте (по одному обработчику на пункт меню) один и тот же код придется копировать много раз.

Кому как, а я предпочитаю создавать класс, производный от TMenuItem, на который возлагаются все операции рисования пунктов меню. Для приведенных в этом подразделе примеров этим классом будет выступать TDrawingMenuItem, описание которого приведено в листинге 3.11.

**Листинг 3.11.** Класс графического пункта меню

```
//Абстрактный класс «рисовальщика», выводящего изображение
//на поверхность меню
class TDrawer
{
public:
  virtual void Measure( TCanvas *canvas, int &width, int
&height ) = 0;
  virtual void Draw( TCanvas *canvas, const TRect &rect, bool
selected )=0;
};
//Класс графического пункта меню
class TDrawingMenuItem : public TMenuItem
{
  TDrawer *m_drawer;
public:
  __fastcall TDrawingMenuItem( TDrawer *drawer, TComponent
*owner )
             : TMenuItem( owner )
  {
    m_drawer = drawer;
    OnMeasureItem = Measure;
    OnDrawItem = Draw;
  }
  __fastcall virtual ~TDrawingMenuItem()
  {
    delete m_drawer;
  }
protected:
  void __fastcall Measure(System::TObject* Sender,
                          Graphics::TCanvas* ACanvas,
                          int &Width, int &Height)
  {
```

```
    //Получим размеры пункта меню
    m_drawer->Measure( ACanvas, Width, Height );
  }
  void __fastcall Draw(System::TObject* Sender,
                       Graphics::TCanvas* ACanvas,
                       const Types::TRect &ARect, bool Se-
lected)
  {
    //Нарисуем пункт меню
    m_drawer->Draw( ACanvas, ARect, Selected );
  }
};
```

На самом деле объекты класса `TDrawingMenuItem` только управляют ходом рисования, вызывая методы `Measure` и `Draw` по указателю на передаваемые в конструктор объекты-рисовальщики. Классы всех рисовальщиков будут производными от `TDrawer` (также приведен в листинге 3.11), поэтому для создания нового вида графического меню (отличающегося не стилем, а цветом линий, например) придется лишь реализовать новый класс, производный от `TDrawer`.

В листинге 3.12 приведено описание класса рисовальщика, использованного для вывода пунктов меню Стиль линий (см. рис. 3.3).

**Листинг 3.12.** Рисование пунктов меню Стиль линий

```
class TLineStyleDrawer : public TDrawer
{
  TPenStyle m_style;  //Стиль линий
public:
  TLineStyleDrawer( TPenStyle style ){ m_style = style; }
  virtual void Measure( TCanvas *canvas, int &width, int
&height )
  {
    width = 50;
    height = 17;
  }
  virtual void Draw( TCanvas *canvas, const TRect &rect, bool
selected )
  {
    canvas->Brush->Style = bsSolid;
    canvas->Pen->Style = m_style;
    if ( selected )
    {
      canvas->Brush->Color = clMenuHighlight;
      canvas->Pen->Color = clHighlightText;
    }
    else
    {
```

```
      canvas->Brush->Color = clMenu;
      canvas->Pen->Color = clMenuText;
    }
    canvas->FillRect( rect );
    int x0 = rect.left + 2;
    int y0 = rect.top + rect.Height()/2;
    canvas->MoveTo( x0, y0 );
    canvas->LineTo( x0 + rect.Width() - 4, y0 );
  }
};
```

Процедура же создания пунктов меню Стиль линий выглядит следующим образом (листинг 3.13).

**Листинг 3.13.** Заполнение подменю Стиль линий

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  //Создание пунктов меню для стиля линий
  mnuStyles->Add(
   new TDrawingMenuItem(new TLineStyleDrawer(psSolid), mnu-
Styles) );
  mnuStyles->Add(
   new TDrawingMenuItem(new TLineStyleDrawer(psDash), mnu-
Styles) );
  mnuStyles->Add(
   new TDrawingMenuItem(new TLineStyleDrawer(psDot), mnu-
Styles) );
  mnuStyles->Add(
   new TDrawingMenuItem(new TLineStyleDrawer(
psDashDot), mnuStyles) );
  //...
}
```

В качестве примера реализован еще один вид меню: меню выбора вида градиента, которое показано на рис. 3.4.


**Рис. 3.4.** Меню Градиент

Описание класса рисовальщика для пунктов этого меню приведено в листинге 3.14.

**Листинг 3.14.** Рисование пунктов меню Градиент

```
class TGradientDrawer : public TDrawer
{
  vector<TColor> m_colors; //Цвета для построения градиента
public:
  TGradientDrawer( const vector<TColor> &colors ){ m_colors =
colors; }
  TGradientDrawer( TColor color1, TColor color2 )
  {
```

```
    m_colors.push_back(color1);
    m_colors.push_back(color2);
  }
  TGradientDrawer( TColor color1, TColor color2, TColor
color3 )
  {
    m_colors.push_back(color1);
    m_colors.push_back(color2);
    m_colors.push_back(color3);
  }
  virtual void Measure( TCanvas *canvas, int &width, int
&height )
  {
    width = 50;
    height = 20;
  }
  virtual void Draw( TCanvas *canvas, const TRect &rect, bool
selected )
  {
    //Заливаем фон в зависимости от состояния (выделен/не
    //выделен)
    canvas->Brush->Style = bsSolid;
    if ( selected )
      canvas->Brush->Color = clMenuHighlight;
    else
      canvas->Brush->Color = clMenu;
    canvas->FillRect( rect );
    canvas->Pen->Style = psSolid;
    int xmin = rect.left + 3, ymin = rect.top + 3;
    int xmax = rect.Width() - 6 + xmin, ymax = rect.Height() -
6 + ymin;
    int ccount = m_colors.size();
    int width = (xmax - xmin) / (ccount - 1);//Ширина перехода
                                             //между
                                             //двумя цветами
    //Нарисуем переходы между заданными цветами
    for ( int c=0; c<ccount-1; c++ )
    {
      BYTE r1 = GetRValue(m_colors[c]);
      BYTE g1 = GetGValue(m_colors[c]);
      BYTE b1 = GetBValue(m_colors[c]);
      BYTE r2 = GetRValue(m_colors[c+1]);
      BYTE g2 = GetGValue(m_colors[c+1]);
      BYTE b2 = GetBValue(m_colors[c+1]);
      BYTE r, g, b;
```

```
      int xstart = xmin + c*width, xend = xmin + (c+1)*width;
      for ( int x=xstart; x<xend; x++ )
      {
        r = r1 + (r2 - r1)*(x - xstart)/width;
        g = g1 + (g2 - g1)*(x - xstart)/width;
        b = b1 + (b2 - b1)*(x - xstart)/width;
        canvas->Pen->Color = (TColor)RGB(r,g,b);
        canvas->MoveTo(x, ymin);
        canvas->LineTo(x, ymax);
      }
    }
  }
};
```

Процедура создания пунктов меню выбора вида градиента может выглядеть так, как показано в листинге 3.15.

**Листинг 3.15.** Заполнение подменю Градиент

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  //Создание пунктов меню для градиента
  mnuGradient->Add( new TDrawingMenuItem(
   new TGradientDrawer(clBlack, clWhite), mnuGradient) );
  mnuGradient->Add( new TDrawingMenuItem(
   new TGradientDrawer(clRed, clGreen), mnuGradient) );
  mnuGradient->Add( new TDrawingMenuItem(
   new TGradientDrawer(clLime, clBlue), mnuGradient) );
  mnuGradient->Add( new TDrawingMenuItem(
   new TGradientDrawer(clYellow, clPurple), mnuGradient) );
  mnuGradient->Add( new TDrawingMenuItem(
   new TGradientDrawer(clRed, clGreen, clBlue), mnuGradient)
);
  mnuGradient->Add( new TDrawingMenuItem(
   new TGradientDrawer(clFuchsia, clYellow, clAqua), mnuGra-
dient) );
  //...
}
```

Для рассмотренных выше графических меню пока не создавались обработчики, то есть не была определена реакция формы на выбор пунктов этих меню. Ниже показаны два графических меню (рис. 3.5), для которых будут реализованы обработчики, причем оба эти меню будут взаимодействовать не только с формой, но и друг с другом.

В качестве меню будут выступать меню выбора названия и размера шрифта, причем в меню выбора размера шрифта текст разного размера будет выводиться выбранным во втором меню шрифтом.

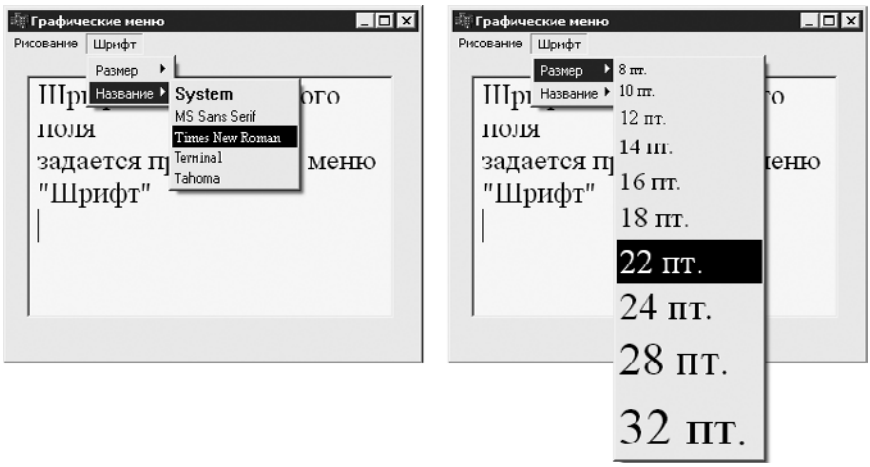


**Рис. 3.5.** Меню выбора названия и размера шрифта

Как и в приведенных ранее примерах, для новых графических меню написаны только классы рисовальщиков. Так, описание класса `TFontSizeDrawer`, выводящего в меню текст разного размера, приведено в листинге 3.16.

**Листинг 3.16.** Рисование пунктов меню Размер

```
class TFontSizeDrawer : public TFontDrawerBase
{
  int m_size; //Размер шрифта, задаваемый этим пунктом меню
public:
  TFontSizeDrawer( int size, TChoosenFont *font ) :
TFontDrawerBase(font)
  {
    m_size = size;
  }
  virtual void Measure( TCanvas *canvas, int &width, int
&height )
  {
    //Определим, какой размер будет занимать надпись нужным
    //шрифтом
    canvas->Font->Name = m_ChoosenFont->GetName();
    canvas->Font->Size = m_size;
    TSize size = canvas->TextExtent( IntToStr(m_size) + "
пт." );
    width = size.cx + 4;
    height = size.cy + 4;
  }
  virtual void Draw( TCanvas *canvas, const TRect &rect, bool
selected )
  {
```

```
    //Подготовим фон
    PrepairBackground( canvas, rect, selected );
    //Покажем текст шрифтом заданного размера
    canvas->Font->Name = m_ChoosenFont->GetName();
    canvas->Font->Size = m_size;
    canvas->TextOut( rect.left + 2, rect.top + 2, IntToStr(m_
size) +" пт.");
  }
};
```

В листинге 3.17 приведен класс `TFontNameDrawer`, выводящий на экран пункты меню выбора названия шрифта.

**Листинг 3.17.** Рисование пунктов меню Название

```
class TFontNameDrawer : public TFontDrawerBase
{
  String m_name; //Название шрифта, задаваемое этим пунктом
меню
public:
  TFontNameDrawer( const String &name, TChoosenFont *font )
  : TFontDrawerBase(font)
  {
    m_name = name;
  }
  virtual void Measure( TCanvas *canvas, int &width, int
&height )
  {
    //Определим, какой размер будет занимать надпись нужным
    //шрифтом
    canvas->Font->Name = m_name;
    TSize size = canvas->TextExtent( m_name );
    width = size.cx + 4;
    height = size.cy + 4;
  }
  virtual void Draw( TCanvas *canvas, const TRect &rect, bool
selected )
  {
    //Подготовим фон
    PrepairBackground( canvas, rect, selected );
    //Покажем текст шрифтом заданного размера
    canvas->Font->Name = m_name;
    canvas->TextOut( rect.left + 2, rect.top + 2, m_name );
  }
};
```

Базовым для двух приведенных классов является `TFontDrawerBase`, приведенный в листинге 3.18.

**Листинг 3.18.** Класс TFontDrawerBase

```
class TFontDrawerBase : public TDrawer
{
public:
  virtual ~TFontDrawerBase(){}
protected:
  TFontDrawerBase( TChoosenFont *font ){ m_ChoosenFont =
font; }
  void PrepairBackground(TCanvas *canvas, const TRect &rect,
bool selected)
  {
    //Рисуем фон в зависимости от состояния меню
    canvas->Brush->Style = bsSolid;
    if ( selected )
      canvas->Brush->Color = clMenuHighlight;
    else
      canvas->Brush->Color = clMenu;
    canvas->FillRect( rect );
  }
  TChoosenFont *m_ChoosenFont; //Указывает на объект-
                               //хранилище параметров
                               //используемого шрифта
};
```

В классе `TFontDrawerBase` реализован общий для всех подклассов метод `PrepairBackground`, который определяет фон пункта меню в зависимости от состояния: выделен или не выделен. Но самое главное — в данном классе хранится указатель на объект `TChoosenFont`. Именно в этом объекте хранятся настройки шрифта, который используется приложением. Указатель на этот объект можно использовать как для изменения, так и для получения настроек шрифта, что, собственно, и делается в методах `Measure` и `Draw` класса `TFontSizeDrawer` (см. листинг 3.16).

Создаются пункты меню выбора размера и названия шрифта следующим образом (листинг 3.19).

**Листинг 3.19.** Создание пунктов меню выбора размера и названия шрифта

```
void AddFontSizeMenuItem( TMenuItem *menu, int size, TChoos-
enFont *font )
{
  //Создание самого пункта меню
  TMenuItem *item =
   new TDrawingMenuItem(new TFontSizeDrawer(size, font),
menu);
  menu->Add( item );
```

```
  //Создание события для обработки изменения размера шрифта
  item->Action = new TFontSizeAction( size, font, item );
}
void AddFontNameMenuItem( TMenuItem *menu, const String
&name, TChoosenFont *font )
{
  //Создание самого пункта меню
  TMenuItem *item =
   new TDrawingMenuItem(new TFontNameDrawer(name, font),
menu);
  menu->Add( item );
  //Создание события для обработки изменения названия шрифта
  item->Action = new TFontNameAction( name, font, item );
}
```

Заметьте, что с созданными пунктами меню (листинг 3.19) связываются и обработчики: экземпляры класса `TFontNameAction` для пунктов меню выбора названия и класса `TFontSizeAction` для пунктов меню выбора размера шрифта. Текст реализации обоих обработчиков приведен в листинге 3.20.

**Листинг 3.20.** Обработчики для меню настроек шрифта

```
class TFontSizeAction : public TAction
{
  int m_size;  //Размер шрифта, устанавливаемый при
               //возникновении события
  TChoosenFont *m_font; //Контейнер параметров шрифта
public:
   __fastcall TFontSizeAction( int size, TChoosenFont *font,
                               TComponent *owner )
   : TAction( owner )
  {
    m_size = size;
    m_font = font;
    OnExecute = SetNewSize;
  }
  void __fastcall SetNewSize(System::TObject* Sender)
  {
    m_font->SetSize( m_size );
  }
};
class TFontNameAction : public TAction
{
  String m_name;  //Название шрифта, устанавливаемое
                  //при возникновении события
  TChoosenFont *m_font; //Контейнер параметров шрифта
public:
```

```
    __fastcall TFontNameAction( const String &name, TChoosenFont
*font,
                                TComponent *owner )
    : TAction( owner )
  {
    m_name = name;
    m_font = font;
    OnExecute = SetNewName;
  }
  void __fastcall SetNewName(System::TObject* Sender)
  {
    m_font->SetName( m_name );
  }
};
```

Идея реализации классов `TFontSizeAction` и `TFontSizeAction` очень проста: при возникновении события (выборе пункта меню) по указателю на объект `TChoosenFont`, переданному в конструкторы классов `TFontSizeAction` и `TFontNameAction`, изменяются размер и название шрифта соответственно.

Что за класс `TChoosenFont`, который здесь так часто упоминается? Это простой класс, в полях которого хранятся настройки шрифта. В приведенном случае ими будут только его название и размер. Текст реализации данного класса приведен в листинге 3.21.

**Листинг 3.21.** Реализация класса TChoosenFont

```
//Интерфейс, который реализует объект, реагирующий на
//изменения
//в TChoosenFont
class IChoosenFontListener
{
public:
  virtual void OnFontChange( TChoosenFont *sender ) = 0;
};
//Параметры используемого шрифта
class TChoosenFont
{
  int m_size;                          //Размер шрифта
  String m_name;                       //Название шрифта
  IChoosenFontListener *m_listener;    //«Слушатель» изменений
                                       //объекта
public:
  TChoosenFont( IChoosenFontListener *listener, TFont *init_
font )
  {
    m_listener = listener;
```

```
    //Первоначальная синхронизация хранимых значений
    //с параметрами
    //шрифта приложения
    m_size = init_font->Size;
    m_name = init_font->Name;
  }
  //Свойства (хранимые параметры шрифта)
  const int GetSize() const { return m_size; }
  void SetSize( int size )
  {
    if ( m_size != size )
    {
      m_size = size;
      m_listener->OnFontChange(this);
    }
  }
  const String &GetName() const { return m_name; }
  void SetName( const String &name )
  {
    if ( m_name != name )
    {
      m_name = name;
      m_listener->OnFontChange(this);
    }
  }
};
```

Кроме того, объект класса `TChoosenFont` содержит алгоритмы, позволяющие уведомлять необходимые элементы формы об изменениях параметров шрифта, — для этого использующий `TChoosenFont` класс должен реализовать интерфейс `IChoosenFontListener`, приведенный в начале листинга 3.21. В данном примере объект `TChoosenFont` создается и хранится формой, а форма еще и реагирует на изменения параметров шрифта (листинги 3.22 и 3.23).

**Листинг 3.22.** Объявление класса формы

```
class TForm1 : public TForm, public IChoosenFontListener
{
...
private:  // User declarations
  TChoosenFont *m_FontSettings; //Хранит параметры шрифта
...
//IChoosenFontListener
  void OnFontChange( TChoosenFont *sender );
};
```

При изменении параметров шрифта изменяется и текст в текстовом поле на форме. Текст реализации реакции формы на эти изменения приведен в листинге 3.23.

**Листинг 3.23.** Реализация реакции на изменение параметров шрифта

```
void TForm1::OnFontChange( TChoosenFont *sender )
{
  //Синхронизируем состояние m_FontSettings и шрифт
  //в текстовом поле
  txtEditor->Font->Size = m_FontSettings->GetSize();
  txtEditor->Font->Name = m_FontSettings->GetName();
}
```

Наконец, создание объекта `TChoosenFont` и наполнение меню, предназначенного для работы со шрифтом, может выглядеть так, как показано в листинге 3.24.

**Листинг 3.24.** Заполнение меню Шрифт

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  //Создание и инициализация объекта для хранения настроек
  //шрифта
  m_FontSettings = new TChoosenFont( this, txtEditor->Font );
  //...
  //Создание пунктов меню для выбора размера шрифта
  AddFontSizeMenuItem( mnuFontSize, 8, m_FontSettings );
  AddFontSizeMenuItem( mnuFontSize, 10, m_FontSettings );
  AddFontSizeMenuItem( mnuFontSize, 12, m_FontSettings );
  AddFontSizeMenuItem( mnuFontSize, 14, m_FontSettings );
  AddFontSizeMenuItem( mnuFontSize, 16, m_FontSettings );
  AddFontSizeMenuItem( mnuFontSize, 18, m_FontSettings );
  AddFontSizeMenuItem( mnuFontSize, 22, m_FontSettings );
  AddFontSizeMenuItem( mnuFontSize, 24, m_FontSettings );
  AddFontSizeMenuItem( mnuFontSize, 28, m_FontSettings );
  AddFontSizeMenuItem( mnuFontSize, 32, m_FontSettings );
  //Создание пунктов меню для выбора названия шрифта
  AddFontNameMenuItem( mnuFontName, "System", m_FontSettings
);
  AddFontNameMenuItem( mnuFontName, "MS Sans Serif", m_
FontSettings );
  AddFontNameMenuItem( mnuFontName, "Times New Roman", m_
FontSettings );
  AddFontNameMenuItem( mnuFontName, "Terminal", m_FontSet-
tings );
  AddFontNameMenuItem( mnuFontName, "Tahoma", m_FontSettings
);
}
```

Вот и замкнулась цепочка классов, используемых в этом примере. В итоге все работает: с помощью меню можно устанавливать параметры шрифта в текстовом поле, а меню могут реагировать на изменения настроек шрифта, вносимые другими меню,

при этом меню не содержат информацию ни друг о друге, ни о форме, да и форма не владеет большим количеством данных об особенностях реализации меню.

# Графические списки

Было бы странным, если бы показанными в предыдущем разделе средствами можно было создавать только графические меню. В действительности возможность написания собственной процедуры рисования предусмотрена для многих компонентов. В этом разделе будут показаны, возможно, наиболее простые случаи: рисование пунктов списков ComboBox и ListBox.

## Графический список ComboBox

На рис. 3.6 приведен пример того, как могут выглядеть графические списки ComboBox.

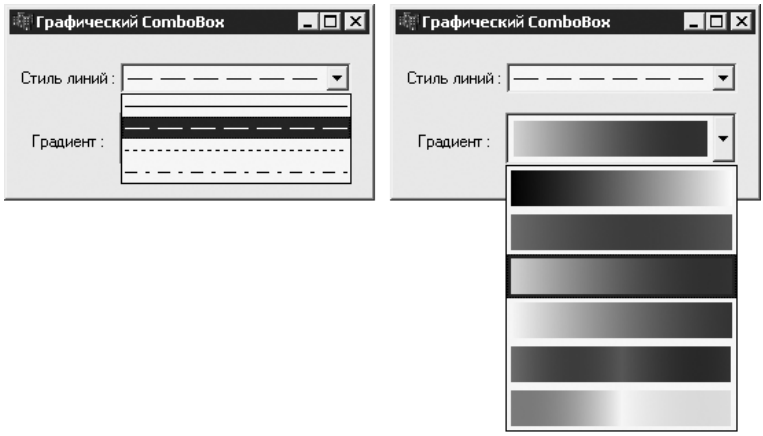


**Рис. 3.6.** Графические списки ComboBox

Чтобы создать раскрывающийся список, аналогичный показанным на рисунке, достаточно выполнить всего два действия. Для начала нужно определить стиль (свойство `Style`, доступ к которому осуществляется через меню окна Object Inspector): `csOwnerDrawFixed` или `csOwnerDrawVariable`. В первом случае будет создан список с элементами одинаковой высоты (определяется свойством `ItemHeignt`), во втором — список с элементами различной высоты.

Аналогично графическим меню при перерисовке элементов списка генерируются события `MeasureItem` (если выбран стиль `csOwnerDrawVariable`) и `DrawItem`. Ниже приведены примеры создания списков с элементами одинаковой высоты, а потому реализован только обработчик события `DrawItem`. Наверное, простейший способ реализовать подобный список — дать обработать событие `DrawItem` форме, но, честно говоря, не хочется наполнять класс формы специализированными обработчиками, ведь это только в данном примере графических списков на форме

только два, а в реальности их может быть значительно больше. Поэтому лучше, на мой взгляд, воспользоваться возможностью ComboBox ассоциировать с каждой строкой списка указатель на TObject.

Это значит, что будут написаны собственные классы для элементов списков (по одному классу на каждый список) и именно в объектах этих классов будут храниться необходимые данные. Естественно, написанные классы будут предусматривать еще и возможность вывода элементов списка на экран. В таком случае для всех графических списков ComboBox можно будет обойтись одним обработчиком OnDrawItem (листинг 3.25). В этом же листинге показано и наполнение списков элементами.

**Листинг 3.25.** Обработчик события DrawItem и заполнение раскрывающихся списков

```
void __fastcall TForm1::ComboBoxDrawItem(TWinControl *Con-
trol, int Index,

TRect &Rect, TOwnerDrawState State)
{
  TComboBox *combo = dynamic_cast<TComboBox *>(Control);
  TItemDrawer *item =
   dynamic_cast<TItemDrawer *>(combo->Items->Objects[Index]);
  item->Draw( combo->Canvas, Rect, State.Contains(odSelected)
);
}
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  //Формирование списка градиентов
  cboGradient->AddItem( "", new TGradientItem(clBlack,
clWhite) );
  cboGradient->AddItem( "", new TGradientItem(clRed, clGreen)
);
  cboGradient->AddItem( "", new TGradientItem(clLime, clBlue)
);
  cboGradient->AddItem( "", new TGradientItem(clYellow,
clPurple) );
  cboGradient->AddItem( "", new TGradientItem(clRed, clGreen,
clBlue) );
  cboGradient->AddItem( "", new TGradientItem(clFuchsia,
clYellow, clAqua));
  //Формирование списка стилей линий
  cboStyle->AddItem( "", new TPenStyleItem(psSolid) );
  cboStyle->AddItem( "", new TPenStyleItem(psDash) );
  cboStyle->AddItem( "", new TPenStyleItem(psDot) );
  cboStyle->AddItem( "", new TPenStyleItem(psDashDot) );
}
```

Поскольку все данные, как было условлено ранее, хранятся в ассоциированных с элементами списках объектов, возможность указания текста для элементов списка не используется.

Все классы элементов списков наследуются от абстрактного класса `TItemDrawer`, описание которого приведено в листинге 3.26.

**Листинг 3.26.** Класс TItemDrawer

```
class TItemDrawer : public THeapObject
{
public:
  __fastcall TItemDrawer(){}
  virtual __fastcall ~TItemDrawer(){}
  virtual void Draw( TCanvas *canvas, const TRect &rect, bool
selected )=0;
};
```

Абстрактным в классе `TItemDrawer` является только метод `Draw`, который и должны реализовать классы элементов каждого списка. Так, текст реализации класса `TPenStyleItem`, хранящего стиль линий и выводящего соответствующий элемент списка на экран, приведен в листинге 3.27.

**Листинг 3.27.** Класс TPenStyleItem (элемент списка Стиль линий)

```
class TPenStyleItem : public TItemDrawer
{
  TPenStyle m_style; //Стиль, задаваемый элементом
public:
  TPenStyleItem( TPenStyle style ){ m_style = style; }
  TPenStyle GetStyle(){ return m_style; }
  virtual void Draw( TCanvas *canvas, const TRect &rect, bool
selected )
  {
    //Заливаем фон в зависимости от состояния (выделен/не
    //выделен)
    canvas->Brush->Style = bsSolid;
    if ( selected )
    {
      canvas->Brush->Color = clHighlight;
      canvas->Pen->Color = clHighlightText;
    }
    else
    {
      canvas->Brush->Color = clWindow;
      canvas->Pen->Color = clWindowText;
    }
    canvas->FillRect( rect );
```

```
    //Рисуем линию нужным стилем
    canvas->Pen->Style = m_style;
    int y = (rect.top + rect.bottom) / 2;
    canvas->MoveTo( rect.left + 2, y );
    canvas->LineTo( rect.right - 2, y );
  }
};
```

В листинге 3.28 приведен текст реализации класса `TGradientItem`, который содержит информацию об опорных цветах, используемых при построении градиента, а также выводит на экран сам градиент при рисовании элемента списка.

**Листинг 3.28.** Класс TGradientItem (элемент списка Градиент)

```
class TGradientItem : public TItemDrawer
{
  vector<TColor> m_colors; //Цвета для построения градиента
public:
  TGradientItem( const vector<TColor> &colors ){ m_colors =
colors; }
  TGradientItem( TColor color1, TColor color2 )
  {
    m_colors.push_back(color1);
    m_colors.push_back(color2);
  }
  TGradientItem( TColor color1, TColor color2, TColor color3
)
  {
    m_colors.push_back(color1);
    m_colors.push_back(color2);
    m_colors.push_back(color3);
  }
  const vector<TColor> GetColors(){ return m_colors; }
  virtual void Draw( TCanvas *canvas, const TRect &rect, bool
selected )
  {
    //Заливаем фон в зависимости от состояния (выделен/не
    //выделен)
    canvas->Brush->Style = bsSolid;
    if ( selected )
      canvas->Brush->Color = clHighlight;
    else
      canvas->Brush->Color = clWindow;
    canvas->FillRect( rect );
    canvas->Pen->Style = psSolid;
    int xmin = rect.left + 3, ymin = rect.top + 3;
    int xmax = rect.Width() - 6 + xmin, ymax = rect.Height() - 6
+ ymin;
```

```
    int ccount = m_colors.size();
    int width = (xmax - xmin) / (ccount - 1);//Ширина перехода
                                             //между
                                             //двумя цветами
    //Нарисуем переходы между заданными цветами
    for ( int c=0; c<ccount-1; c++ )
    {
      BYTE r1 = GetRValue(m_colors[c]);
      BYTE g1 = GetGValue(m_colors[c]);
      BYTE b1 = GetBValue(m_colors[c]);
      BYTE r2 = GetRValue(m_colors[c+1]);
      BYTE g2 = GetGValue(m_colors[c+1]);
      BYTE b2 = GetBValue(m_colors[c+1]);
      BYTE r, g, b;
      int xstart = xmin + c*width, xend = xmin + (c+1)*width;
      for ( int x=xstart; x<xend; x++ )
      {
        r = r1 + (r2 - r1)*(x - xstart)/width;
        g = g1 + (g2 - g1)*(x - xstart)/width;
        b = b1 + (b2 - b1)*(x - xstart)/width;
        canvas->Pen->Color = (TColor)RGB(r,g,b);
        canvas->MoveTo(x, ymin);
        canvas->LineTo(x, ymax);
      }
    }
  }
};
```

Вот, собственно, и все, что относится к процедуре создания графического списка ComboBox. Позволю себе лишь остановиться еще на одной небольшой тонкости, использованной в этом примере. Как можно увидеть из текста листинга 3.25, указатели на создаваемые с помощью оператора `new` объекты классов `TGradientItem` и `TPenStyleItem` нигде, кроме свойства `Objects` списка, не сохраняются. Мне не хотелось загромождать и без того не короткие листинги еще и кодом учета созданных объектов, поэтому для создаваемых элементов списков был написан примитивный так называемый «сборщик мусора» (листинг 3.29).

**Листинг 3.29.** Удаление элементов списка (сборщик мусора)

```
class THeapObjects : public vector<TObject*>
{
public:
  ~THeapObjects()
  {
    for ( unsigned int i=0; i<size(); i++ )
      delete (*this)[i];
  }
```

```
} g_HeapObjects ;
class THeapObject : public TObject
{
public:
  __fastcall THeapObject(){ g_HeapObjects.push_back(this); }
  virtual __fastcall ~THeapObject(){}
};
```

Работает этот «сборщик» очень просто. Если вы внимательно просматривали листинг 3.26, то наверняка заметили, что класс `TItemDrawer` наследуется от некого `THeapObject`. Так вот, указатели на все объекты, классы которых производны от `THeapObject`, регистрируются в глобальном объекте-контейнере (по сути, массиве) `g_HeapObjects`. После завершения работы приложения вызывается деструктор класса `THeapObjects`, который удаляет все созданные объекты.

Такой подход, конечно, не является особенно гибким и универсальным, но в некоторых случаях может оказаться весьма полезным. Этот же подход к управлению созданными объектами применен и в следующем примере.

## Графический список ListBox

С такой же легкостью, с какой были добавлены графические объекты в список ComboBox, они могут быть добавлены и в список ListBox. Два рассмотренных далее графических списка выглядят аналогично изображенным на рис. 3.7.

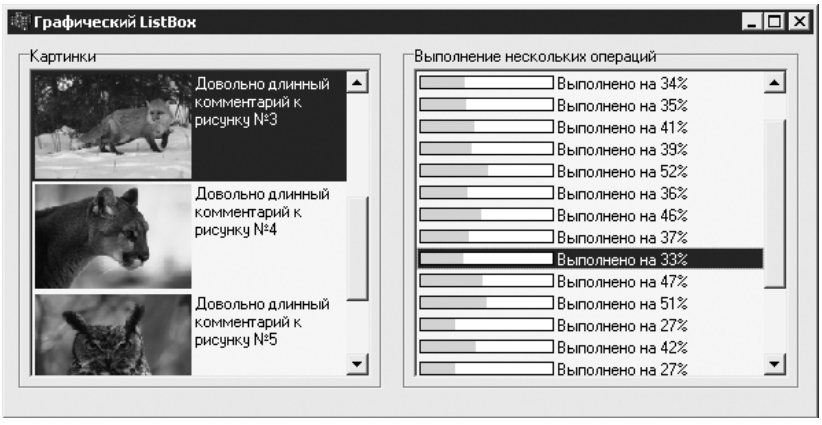


**Рис. 3.7.** Графические списки TListBox

Чтобы можно было переопределить вывод на экран элементов списка, для свойства `Style`, доступного в меню окна Object Inspector, можно установить значение `lbOwnerDrawFixed` или `lbOwnerDrawVariable`. Все остальные настройки аналогичны настройкам графического списка ComboBox.

В приведенных ниже примерах используются списки с одинаковой высотой элементов (для первого списка `ItemHeight` составляет 80, для второго — 16). Текст

обработчика события DrawItem для списков, реализованный в классе формы, выглядит следующим образом (листинг 3.30).

**Листинг 3.30.** Обработчик DrawItem

```
void __fastcall TForm1::ListBoxDrawItem(TWinControl *Control,
int Index,

TRect &Rect, TOwnerDrawState State)
{
  TListBox *list = dynamic_cast<TListBox*>(Control);
  TItemDrawer *item =
   dynamic_cast<TItemDrawer*>(list->Items->Objects[Index]);
  item->Draw( list->Canvas, Rect, State.Contains(odSelected)
);
}
```

Хранением и выводом на экран изображений и многострочных комментариев к ним (левый список на рис. 3.7) занимается класс TImageItem, описание которого приведено в листинге 3.31.

**Листинг 3.31.** Класс TImageItem

```
class TImageItem : public TItemDrawer
{
  Graphics::TBitmap *m_image; //Картинка
  String m_text;              //Комментарий к картинке
public:
  TImageItem( const String &text, Graphics::TBitmap *image )
  {
    m_text = text;
    m_image = image;
  }
  Graphics::TBitmap *Image(){ return m_image; }
  String &Text(){ return m_text; }
//TItemDrawer
  virtual void Draw( TCanvas *canvas, const TRect &rect, bool
selected )
  {
    //Заливаем фон в зависимости от состояния (выделен/не
    //выделен)
    canvas->Brush->Style = bsSolid;
    if ( selected )
      canvas->Brush->Color = clHighlight;
    else
      canvas->Brush->Color = clWindow;
    canvas->FillRect( rect );
    //Выводим картинку
```

```
    TRect image_rect;
    image_rect.left = rect.left + 2;
    image_rect.top = rect.top + 2;
    image_rect.right = rect.left + 2 + (rect.Height()-4) * 3
/ 2;
    image_rect.bottom = rect.bottom - 2;
    canvas->StretchDraw( image_rect, m_image );
    //Выводим текст
    TRect text_rect;
    text_rect.left = image_rect.right + 2;
    text_rect.right = rect.right - 2;
    text_rect.top = rect.top + 2;
    text_rect.bottom = rect.bottom - 2;
    ::DrawText( canvas->Handle, m_text.c_str(), m_text.
Length(),
                &text_rect, DT_LEFT | DT_WORDBREAK );
  }
};
```

Хранение и вывод элементов списка, расположенного на рисунке слева, возложены на класс `TProcessItem` (листинг 3.32). Обратите внимание, что, помимо текста и процента якобы выполненной работы, в объектах класса `TProcessItem` сохраняется и указатель на список, в котором каждый элемент находится. Благодаря этому при изменении текста или процента выполненной работы обновляется вид списка на экране.

**Листинг 3.32.** Класс TProcessItem

```
class TProcessItem : public TItemDrawer
{
  int m_percent; //Процент выполненной работы
  String m_text; //Выполнено
  //Для обновления при изменении
  TListBox *m_list;
  bool m_drawed;
public:
  TProcessItem( const String &init_text, TListBox *list )
  {
    m_percent = 0;
    m_text = init_text;
    m_list = list;
    m_drawed = false;
  }
  void SetValue( int percent, const String &text )
  {
    m_percent = percent;
    m_text = text;
```

```
    if ( m_drawed )
    {
      //Список нужно перерисовать
      m_list->Invalidate();
    }
  }
  //TItemDrawer
  virtual void Draw( TCanvas *canvas, const TRect &rect, bool
selected )
  {
    m_drawed = true;
    //Заливаем фон в зависимости от состояния (выделен/не
    //выделен)
    canvas->Brush->Style = bsSolid;
    if ( selected )
      canvas->Brush->Color = clHighlight;
    else
      canvas->Brush->Color = clWindow;
    canvas->FillRect( rect );
    //Рассчитываем положение подобия TProgressBar
    TRect p_rect;
    p_rect.left = rect.left + 2;
    p_rect.right = 100;
    p_rect.top = rect.top + 3;
    p_rect.bottom = rect.bottom - 3;
    //Выводим текст
    TRect text_rect;
    text_rect.left = p_rect.right + 2;
    text_rect.right = rect.right - 2;
    text_rect.top = rect.top + 2;
    text_rect.bottom = rect.bottom - 2;
    ::DrawText( canvas->Handle, m_text.c_str(), m_text.
Length(),
                &text_rect, DT_LEFT | DT_SINGLELINE | DT_
VCENTER );
    //Рисуем подобие TProgressBar
    //..рамка и белый фон
    canvas->Brush->Color = clWhite;
    canvas->Pen->Color = clBlack;
    canvas->Rectangle(p_rect);
    //..сама полоска, соответствующая выполненной части
    //работы
    canvas->Brush->Color = clLime;
    canvas->Pen->Color = clLime;
    p_rect.left++;
```

```
    p_rect.right = p_rect.left + (p_rect.Width() - 1) * m_
percent / 100;
    p_rect.top++;
    p_rect.bottom--;
    canvas->Rectangle(p_rect);
  }
};
```

Способ заполнения обоих рассмотренных списков отражен в листинге 3.33. Левый список на форме (см. рис. 3.7) имеет имя `lstImages`, правый — `lstProcess`.

**Листинг 3.33.** Заполнение графических списков

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  //Заполнение ListBox с рисунками
  String str = "Довольно длинный комментарий к рисунку №";
  for ( int i=1; i<=5; i++ )
  {
    Graphics::TBitmap *bmp = new Graphics::TBitmap;
    bmp->LoadFromFile( "Image" + IntToStr(i) + ".bmp" );
    lstImages->AddItem( "", new TImageItem( str + IntToStr(i),
bmp ) );
  }
  //Заполнение ListBox с ходом выполнения разных действий
  for ( int j=0; j<20; j++ )
  {
    TProcessItem *item = new TProcessItem( "Ждем начала",
lstProcess );
    lstProcess->AddItem( "", item );
    m_processes.push_back( new TProcess(item) );
  }
}
```

Обратите внимание, что, помимо помещения элементов в список `lstProcess`, созданные экземпляры `TProcessItem` передаются в конструктор некого класса `TProcess`. В рамках данного примера это и будет некоторый процесс, ход выполнения которого отражается в соответствующем элементе списка. В этом примере создается 20 таких «процессов». Способ реализации класса `TProcess`, описанный в листинге 3.34, крайне прост.

**Листинг 3.34.** Класс TProcess

```
class TProcess : public THeapObject
{
  int m_percent;         //Часть выполненной работы
  TProcessItem *m_item; //Объект для отображения хода
выполнения
```

```
public:
  __fastcall TProcess( TProcessItem *item )
  {
    m_item = item;
  }
  void NextStep()
  {
    if ( m_percent < 100 )
    {
      //Считаем, что работы выполнено на 1 % больше
      m_percent++;
      if ( m_percent < 100 )
        m_item->SetValue( m_percent, "Выполнено на " +
                          IntToStr(m_percent) + "%" );
      else
        m_item->SetValue( m_percent, "Выполнено" );
    }
  }
};
```

Единственный, за исключением конструктора, метод класса `TProcess` в данном примере вызывается по таймеру (листинг 3.35).

**Листинг 3.35.** Выполнение очередной части работы

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
  m_processes[random(m_processes.size())]->NextStep();
}
```

В действительности за классом `TProcess` может скрываться все, что угодно: процесс копирования больших файлов, сканирования диска, сложный математический расчет и т. д., — но приводимый здесь подход может использоваться и в этих случаях, разве что потребуется предусмотреть корректное взаимодействие разных процессов или потоков. Однако это тема уже для другого разговора, который, к сожалению, не является частью данной книги.

# Глава 4

## Мультимедиа

- Компоненты для работы с видео и звуком
- Использование Windows API для работы со звуком
- Низкоуровневая работа со звуком

Время от времени при разработке приложений для Windows появляется необходимость разнообразить интерфейс с помощью анимации или же добавить к программе звуковое сопровождение. Речь, конечно, идет не об аудио- или видеопроигрывателях или редакторах, для которых работа с мультимедиаданными является основной задачей.

В этой главе вы познакомитесь с высокоуровневыми средствами работы с файлами мультимедиа, а именно: средствами обработки видео и звука по средством стандартных компонентов и некоторыми API-функциями, позволяющими воспроизводить звук. В последней части главы будут более подробно рассмотрены основные принципы работы со звуковыми данными, имеющими цифровой формат. Также в этой главе приведен пример программы, предназначенной для синтеза звуков и звукового редактора.

# Компоненты для работы с видео и звуком

Рассмотрение данной главы стоит начать с обзора компонентов, предназначенных для работы с видео и звуком.

## Компонент Animate

Обычно в среде Borland C++ Builder доступны по меньшей мере два компонента, позволяющих работать с мультимедиа-данными. Одним из них является компонент Animate (кнопка  закладки Win32 палитры компонентов).

Компонент Animate, как можно догадаться из его названия, предназначен для того, чтобы облегчить задачу внедрения в приложения небольших элементов анимации. Анимация, которую может воспроизводить рассматриваемый компонент, должна храниться в несжатых видеофайлах формата AVI и не содержать звука. При этом максимальный размер воспроизводимых видеофайлов ограничен размером 64 Кбайт.

На рис. 4.1 показано, как может выглядеть форма с компонентом Animate при воспроизведении видеофайла.

Как же использовать компонент Animate? В самом простом случае достаточно указать имя файла в свойстве `FileName` компонента или идентификатор одного из стандартных видеофайлов Windows (например, копирование файла, показанное на рис. 4.1) в свойстве `CommonAVI`. Теперь, если установить свойство `Active` компонента в значение `true`, начнется воспроизведение. Естественно, настраивать свойства компонента Animate нужно уже после того, как сам компонент помещен на форму.

Схема работы компонента Animate, почти аналогичная описанной выше, применена в примере, внешний вид которого и показан на рис. 4.1. Часть кода, относящаяся собственно к использованию компонента Animate, настолько проста, что вряд ли

стоит приводить ее в тексте книги (пример доступен на прилагаемом к книге компакт-диске). Суть реализации примера укладывается в приведенную схему.

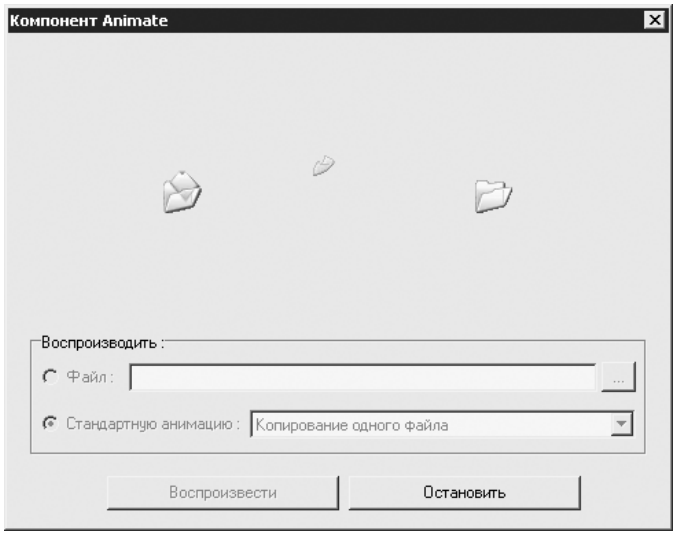


**Рис. 4.1.** Воспроизведение видеофайла компонентом Animate

Выше было приведено весьма общее описание принципа использования компонента Animate. Далее будут более подробно рассмотрены возможности, предоставляемые данным компонентом, а также особенности, которые необходимо учитывать при использовании этих возможностей.

В качестве параметров настройки компонента Animate предусмотрено несколько специфичных для этого компонента свойств. Основные, на мой взгляд, свойства, а именно свойства, определяющие, что именно будет воспроизводить компонент Animate, представлены ниже.

Компонент Animate позволяет проигрывать следующие видеоданные:

- стандартную анимацию Windows; в этом случае с помощью свойства `CommonAVI` необходимо указать тип воспроизводимой анимации; значениями свойства `CommonAVI` могут быть:
  - `aviNone` — значение по умолчанию, означающее, что стандартная анимация Windows не используется;
  - `aviCopyFile` — определяет анимацию, связанную с операциями копирования единственного файла;
  - `aviCopyFiles` — анимация, связанная с операциями копирования нескольких файлов;
  - `aviDeleteFile` — определяет анимацию, связанную с операцией удаления файла;

- `aviEmptyRecycle` — анимация, связанная с операцией очистки Корзины Windows;
- `aviRecycleFile` — определяет анимацию, связанную с операцией удаления (перемещения) файла в Корзину Windows;
- `aviFindComputer` — анимация, связанная с операцией поиска компьютера в сети;
- `aviFindFile` — определяет анимацию, связанную с операцией поиска файла;
- `aviFindFolder` — анимация, связанная с операцией поиска папки;

- пользовательские видеофайлы, хранящиеся на диске компьютера; в этом случае в качестве значения свойства `FileName` указывается путь к видеофайлу;
- видеоданные, сохраненные в секции ресурсов одного из модулей приложения (файлы EXE или DLL); в этом случае в свойстве `ResHandle` компонента указывается дескриптор модуля, а в одном из свойств `ResId` или `ResName` — соответственно целочисленный или строковый идентификатор ресурса (перечисленные свойства доступны только во время выполнения программы, то есть только программно).

**ПРИМЕЧАНИЕ**

Более подробно работа с ресурсами, в том числе способы реализации возможностей воспроизведения мультимедиа-данных, хранимых в ресурсах, рассмотрена в гл. 7.

Ниже перечислены остальные свойства компонента Animate.

- `Active` — после выбора в этом свойстве значения `true` начинается воспроизведение анимации. Для остановки воспроизведения достаточно изменить значение свойства на `false`.
- `Center` — данное свойство полезно, если размер кадров видеоклипа отличается от размера компонента Animate. Если установлено значение `true`, то воспроизводимое содержимое отображается по центру компонента, если же выбрано значение `false`, то изображение прижимается к левому верхнему краю компонента.
- `FrameCount` — указывает количество кадров в выбранном видеоклипе; изменение свойства осуществляется только программно.
- `FrameHeight` — задает высоту кадров видеоклипа.
- `FrameWidth` — указывает ширину кадров видеоклипа.
- `Open` — изменение свойства осуществляется только программно. Значение свойства `true` означает, что содержимое видеоклипа загружено в память и компонент готов к воспроизведению.
- `Repetitions` — определяет количество повторений анимации при воспроизведении; если указано значение `0`, то анимация воспроизводится по кругу бесконечно.

- `StartFrame` — задает номер кадра, с которого будет начинаться воспроизведение; нумерация начинается с `1`, при этом кадр с указанным номером будет показываться после открытия видеоклипа, если воспроизведение не начато.
- `StopFrame` — определяет номер последнего воспроизводимого кадра анимации; если указано значение `0`, то видеоклип будет воспроизводиться до конца.
- `Transparent` — задает прозрачность фона видеоклипа. Если в качестве значения этого свойства указано `true`, фон становится прозрачным.

Для компонента Animate также предусмотрены несколько методов, которые, по большому счету, дублируют функционал, доступный при использовании свойств.

- `void Play (Word FromFrame, Word ToFrame, int Count)` — воспроизводит анимацию с кадра `FromFrame` до кадра `ToFrame` с количеством повторений, равным `Count`. Вызов этого метода позволяет сократить объем кода, заменяя работу со свойствами `StartFrame`, `StopFrame`, `Repetitions` и `Active` работой со всего одной строкой кода вызова метода `Play`.
- `void Stop()` — прекращает воспроизведение. Вызов метода эквивалентен установке свойства `Active` в `false`.
- `void Reset()` — устанавливает значение свойства `StartFrame` равным `1`, а свойства `StopFrame` равным `0`.
- `void Seek(short Frame)` — отображает кадр с заданным номером. При этом было экспериментально замечено, что после установки свойства `Active` в положение `true` воспроизведение начинается со значения, ранее заданного в свойстве `StartFrame`, а не с переданного в функцию `Seek`.

Наконец, компонент Animate генерирует четыре специфических события:

- `OnOpen` — если данные клипа загружены, то свойство `Open` устанавливается в значение `true`;
- `OnClose` — если данные клипа выгружены, то свойство `Open` устанавливается в значение `false`;
- `OnStart` — если начато воспроизведение, то свойство `Active` устанавливается в значение `true`;
- `OnStop` — если воспроизведение остановлено, то свойство `Active` устанавливается в значение `false`.

Используя описанные выше свойства, методы и события компонента Animate, вы теперь можете с легкостью внедрять элементы анимации в свои приложения.

## Компонент MediaPlayer

Даже беглого знакомства с описанным в предыдущем подразделе компонентом Animate достаточно, чтобы понять, что им не удастся обойтись, если нужно воспроизводить большие видеофайлы или проигрывать хотя бы небольшие музыкальные

фрагменты. Здесь понадобится другой компонент — MediaPlayer. Он доступен на вкладке System палитры компонентов (кнопка ).

Компонент MediaPlayer, по сути, представляет собой завершенный мультимедиа-проигрыватель, позволяющий воспроизводить файлы аудио и видео различных форматов, причем MediaPlayer реализован как графический компонент, то есть он обладает органами управления воспроизведением (рис. 4.2).

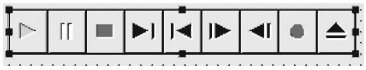

**Рис. 4.2.** Компонент MediaPlayer

Компонент MediaPlayer создан на основе набора API-функций, реализующих так называемый MCI (multimedia control interface, или интерфейс (в смысле программный) для управления устройствами мультимедиа).

Упрощенно о MCI можно сказать следующим образом. Он реализует программный интерфейс абстрактного устройства мультимедиа, поддерживающего ряд операций: операции открытия, загрузки данных, воспроизведения, остановки и навигации, закрытия устройства, причем команды устройству отдаются в виде строк (например, `play d:\music.wav from 1 to 100`). При использовании MCI программисту необходимо лишь знать формат команд или уметь пользоваться набором функций, которые формируют строки команд на основе переданных параметров.

Поскольку в составе Borland C++ Builder имеется готовый компонент MediaPlayer, то, пожалуй, на этом знакомство собственно с MCI будет закончено. Чуть ниже будут кратко рассмотрены основные вопросы, связанные с использованием самого компонента MediaPlayer, а в следующих двух разделах будут приведены примеры создания универсального проигрывателя, позволяющего воспроизводить звуковые и видеофайлы, и проигрывателя компакт-дисков.

Компонент MediaPlayer обладает довольно большим количеством специфичных свойств, поэтому, чтобы все-таки не превращать данную книгу в справочник, ниже рассмотрены только те свойства, которые задействованы в приведенных в этой книге примерах.

- `AutoOpen` — если значением свойства указано `true`, то сразу после создания компонента во время выполнения будет произведена попытка открытия соответствующего устройства.
- `DeviceType` — определяет тип данных мультимедиа (и, соответственно, тип устройства мультимедиа). Может принимать следующие значения:
  - `dtAutoSelect` — подсистема MCI сама определяет тип данных мультимедиа;
  - `dtAVIVideo` — видео в формате AVI;
  - `dtCDAudio` — аудиокомпакт-диск;
  - `dtDAT` — цифровой кассетный аудиопроигрыватель;

- dtDigitalVideo — цифровое видео;
- dtMMMovie — видео в формате multimedia movie;
- dtOverlay — устройство для наложения изображения;
- dtScanner — сканер;
- dtSequencer — MIDI-файл;
- dtVCR — аналоговое видео;
- dtVideodisc — видеодиск;
- dtWaveAudio — несжатое аудио;
- dtOther — неопределенный тип.

- Display — во время выполнения программы в качестве значения этого свойства можно указать компонент формы, на который предполагается выводить изображение при воспроизведении видео.
- FileName — задает имя файла с данными мультимедиа.
- Length — определяет длину загруженных в проигрыватель данных мультимедиа. Конкретные значения зависят от используемого формата времени (определяется свойством TimeFormat). В примерах этой книги используются значения только в миллисекундах.
- Mode — позволяет получить информацию о текущем состоянии проигрывателя. Может принимать значения:
  - mpNotReady — устройство не готово (не открыто либо не загружены данные мультимедиа);
  - mpStopped — данные мультимедиа загружены, устройство готово к воспроизведению;
  - mpPlaying — идет воспроизведение;
  - mpRecording — идет запись;
  - mpSeeking — идет перемещение по мультимедиаданным (поиск заданной позиции);
  - mpPaused — воспроизведение поставлено на паузу;
  - mpOpen — устройство открыто.
- Position — позволяет получить информацию о текущей позиции воспроизведения данных мультимедиа; значение свойства зависит от используемого формата времени.
- TrackLength[номер_дорожки] — указывает длину дорожки с заданным номером (нумерация дорожек начинается с нуля).
- TrackPosition[номер_дорожки] — указывает позицию начала дорожки с заданным номером (нумерация дорожек начинается с нуля).
- Tracks — указывает количество дорожек в данных мультимедиа.

Ниже также приведено описание методов компонента MediaPlayer, которые пригодились при реализации приведенных далее примеров (собственно, эти методы и используются для подачи устройству мультимедиа команд, о которых было упомянуто в начале этого раздела):

- `Open` — открывает устройство;
- `Rewind` — перемещает позицию начала воспроизведения в начало данных мультимедиа;
- `Play` — запускает воспроизведение;
- `Stop` — останавливает воспроизведение;
- `Pause` — ставит воспроизведение файла на паузу;
- `Close` — закрывает устройство;
- `Eject` — извлекает носитель из устройства (например, компакт-диск из дисковода).

На этом чисто теоретическое рассмотрение компонента MediaPlayer можно считать завершенным и переходить к более интересной практической части.

## Универсальный проигрыватель

Рассмотренный в данном подразделе простой проигрыватель позволяет прослушивать звуковые файлы в формате WAV и видеофайлы в формате AVI. Внешний вид формы проигрывателя на этапе его разработки показан на рис. 4.3.

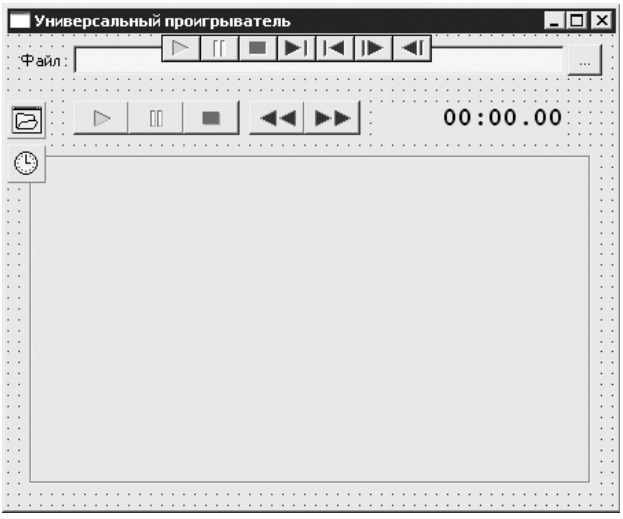


**Рис. 4.3.** Универсальный проигрыватель на этапе разработки

Как видите, при разработке примера было решено поместить на форму собственные кнопки управления проигрывателем, так как это предоставляет больше свободы при создании внешнего вида программы.

Кроме кнопок, управляющих проигрывателем, на форме присутствуют текстовые поля, предназначенные для отображения полного имени воспроизводимого файла, и кнопка  , служащая для выбора этого файла. Особенностью данного проигрывателя является наличие текстового индикатора позиции воспроизведения, в котором время отображается с точностью до сотых долей секунды.

Рассмотрение реализации проигрывателя стоит начать отнюдь не с компонента MediaPlayer, а с описания способа, благодаря которому обеспечивается адекватное оформление внешнего вида проигрывателя в зависимости от его состояния. Под оформлением в данном случае понимается состояние (значение свойства `Enabled`) управляющих кнопок. Согласитесь, было бы не слишком правильно оставлять пользователю возможность нажать, например, кнопку начала воспроизведения до того, как будет выбран файл. То же касается кнопок перемотки, паузы и т. д.

В данном случае можно выделить четыре состояния, в которых может находиться программа. Если состояния задать путем перечисления, завести для формы переменную-член класса, хранящую информацию о текущем состоянии, и реализовать возможность своевременного изменения значения переменной состояния, то можно с легкостью создать адекватный вид и соответствующее поведение формы, реализовав одну-единственную функцию-член, которая будет вызываться при каждом действии, изменяющем состояние проигрывателя. Именно так и сделано в приводимом примере.

В листинге 4.1 приведен фрагмент класса формы, содержащего описание перечисления состояний, переменную `m_State`, хранящую текущее состояние проигрывателя, и объявление функции `RefreshControls`, отвечающей за надлежащее оформление формы проигрывателя.

**Листинг 4.1.** Состояния проигрывателя

```
class TForm1 : public TForm
{
...
private:  // User declarations
  //Состояние проигрывателя (для правильного оформления
  //и поведения формы)
  enum TPlayerState
  {
    psNoFile,     //Первоначальное состояние (нечего
                  //воспроизводить)
    psFileOpened, //Задан файл, можно воспроизводить
    psPlaying,    //Идет воспроизведение
    psPaused      //Пауза
  }m_State;
  void RefreshControls();
...
};
```

Реализация функции RefreshControls приведена в листинге 4.2. Как видите, она получилась очень простой.

**Листинг 4.2.** Оформление формы в соответствии с состоянием проигрывателя

```
void TForm1::RefreshControls()
{
  //Оформление компонентов в соответствии с состоянием
  //проигрывателя
  switch ( m_State )
  {
  case psNoFile:
    cmbPlay->Enabled = false;
    cmbPause->Enabled = false;
    cmbStop->Enabled = false;
    cmbForw->Enabled = false;
    cmbBack->Enabled = false;
    cmbBrowse->Enabled = true;
    refresh_timer->Enabled = false;
    break;
  case psFileOpened:
    cmbPlay->Enabled = true;
    cmbPause->Enabled = false;
    cmbStop->Enabled = false;
    cmbForw->Enabled = true;
    cmbBack->Enabled = true;
    cmbBrowse->Enabled = true;
    refresh_timer->Enabled = false;
    break;
  case psPlaying:
    cmbPlay->Enabled = false;
    cmbPause->Enabled = true;
    cmbStop->Enabled = true;
    cmbForw->Enabled = false;
    cmbBack->Enabled = false;
    cmbBrowse->Enabled = false;
    refresh_timer->Enabled = true;
    break;
  case psPaused:
    cmbPlay->Enabled = true;
    cmbPause->Enabled = false;
    cmbStop->Enabled = true;
    cmbForw->Enabled = true;
    cmbBack->Enabled = true;
    cmbBrowse->Enabled = false;
    refresh_timer->Enabled = false;
```

```
    break;
  }
  RefreshTimePos();
}
```

В самом конце функции `RefreshControls` можно заметить процедуру вызова еще одной функции-члена класса `TForm1` — функции `RefreshTimePos` (листинг 4.3). Это последняя реализованная в рамках примера вспомогательная функция, отвечающая за правильный вид индикатора текущей позиции в воспроизводимом файле.

**Листинг 4.3.** Оформление индикатора позиции воспроизведения

```
void TForm1::RefreshTimePos()
{
  //Установка показаний позиции в воспроизводимом файле
  switch ( m_State )
  {
  case psNoFile:
    lblPos->Caption = "";
    break;
  case psFileOpened:
  case psPlaying:
  case psPaused:
    {
      //Переводим время из миллисекунд в минуты, секунды
      //и миллисекунды*10
      long time = player->Position;
      long mm = time / 60000;
      time -= mm * 60000;
      long ss = time / 1000;
      long ms10 = (time - ss*1000)/10;
      char buff[20];
      sprintf( buff, "%0.2d:%0.2d.%0.2d", mm, ss, ms10 );
      lblPos->Caption = buff;
    }
    break;
  }
}
```

Как ни странно, код управления компонентом MediaPlayer является самой простой частью примера. При написании кода, выполняющегося первым при запуске приложения, в конструкторе формы нужно не забыть определить исходное состояние проигрывателя, а при создании формы указать, что MediaPlayer может выводить видео в окне компонента с именем `video_pane` (компонент Panel). Сказанное реализовано в листинге 4.4.

**Листинг 4.4.** Инициализация приложения

```
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
  m_State = psNoFile;
}
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  player->Display = video_pane;
  RefreshControls();
}
```

Единственное, что усложняет функцию открытия файла, — необходимость обработки возможных на этом шаге ошибок (см. описание обработчика нажатия кнопки `cmbBrowse` в листинге 4.5).

**Листинг 4.5.** Открытие файла

```
void __fastcall TForm1::cmbBrowseClick(TObject *Sender)
{
  //Выбираем файл для воспроизведения
  if ( dlgOpen->Execute() )
  {
    try
    {
      //Пытаемся открыть указанный файл
      player->FileName = dlgOpen->FileName;
      player->DeviceType = dtAutoSelect;
      player->Open();
      player->TimeFormat = tfMilliseconds;
      txtFile->Text = dlgOpen->FileName;
      m_State = psFileOpened;
    }
    catch ( Exception * )
    {
      Application->MessageBox("Ошибка при открытии файла",
                        Application->Title.c_str(), MB_
ICONEXCLAMATION);
      player->FileName = "";
      txtFile->Text = "";
    }
    RefreshControls();
  }
}
```

Обработчики для остальных кнопок, отвечающих за запуск, остановку и перемотку, настолько просты, что вряд ли нуждаются в отдельном пояснении. Описание всех этих обработчиков приведено в листинге 4.6.

**Листинг 4.6.** Управление воспроизведением

```
void __fastcall TForm1::cmbPlayClick(TObject *Sender)
{
  //Начало/продолжение воспроизведения
  m_State = psPlaying;
  RefreshControls();
  player->Play();
}
void __fastcall TForm1::cmbPauseClick(TObject *Sender)
{
  //Остановка воспроизведения
  player->Pause();
  m_State = psPaused;
  RefreshControls();
}
void __fastcall TForm1::cmbStopClick(TObject *Sender)
{
  //Завершение воспроизведения
  player->Stop();
  player->Rewind();
  m_State = psFileOpened;
  RefreshControls();
}
void __fastcall TForm1::cmbBackClick(TObject *Sender)
{
  //Перемотка назад на 10%
  long new_pos = player->Position - player->Length/10;
  player->Position = (new_pos >= 0) ? new_pos : 0;
  RefreshControls();
}
void __fastcall TForm1::cmbForwClick(TObject *Sender)
{
  //Перемотка вперед на 10%
  long new_pos = player->Position + player->Length/10;
  player->Position = (new_pos < player->Length) ? new_pos :
player->Length;
  RefreshControls();
}
```

Единственное, о чем не стоит забывать при написании приведенных в листинге 4.6 обработчиков, так это своевременно изменять переменную `m_State` и обновлять внешний вид формы с помощью метода функции `RefreshControls` (листинг 4.7).

Последнее, в чем заключается особенность данного проигрывателя, — надпись-индикатор текущей позиции воспроизведения. Он реализован, что называется, «в лоб», с помощью обычного таймера с таймаутом 10 мс.

**Листинг 4.7.** Обновление индикатора позиции воспроизведения

```
void __fastcall TForm1::refresh_timerTimer(TObject *Sender)
{
  if ( m_State == psPlaying && player->Mode != mpPlaying )
  {
    //Закончилось воспроизведение
    m_State = psFileOpened;
    RefreshControls();
  }
  else
  {
    //Просто обновим показания счетчика воспроизведения
    RefreshTimePos();
  }
}
```

Как видите, обработчик таймера также используется и для контроля над процессом воспроизведения компонентом MediaPlayer.

Наконец, проигрыватель полностью готов. На рис. 4.4 показан внешний вид приложения при воспроизведении аудио- и видеофайлов.

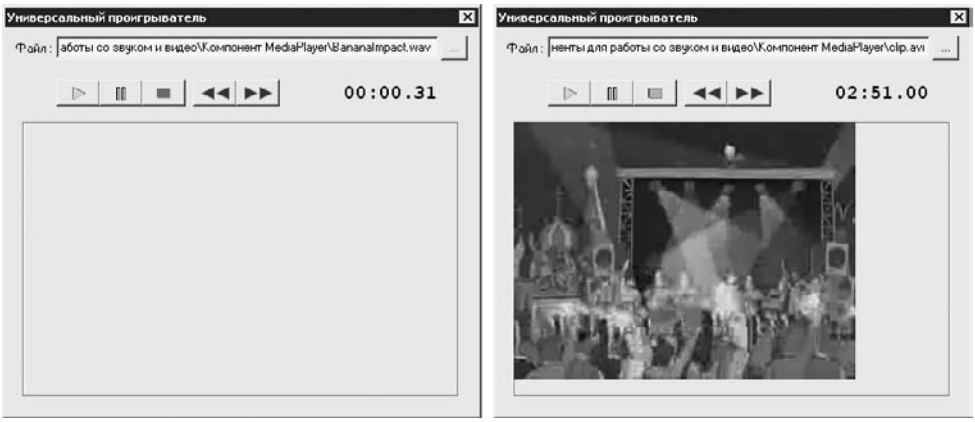


**Рис. 4.4.** Универсальный проигрыватель в работе

Напоследок стоит дать ответ на один из возможных вопросов: почему приведенный здесь «универсальный» проигрыватель может воспроизводить только файлы WAV и AVI? Честно говоря, лишь потому, что при разработке примера было решено обратить ваше внимание не на возможность воспроизведения множества различных форматов файлов мультимедиа (именно поэтому в фильтр окна выбора файла были включены лишь форматы WAV и AVI), а на сам процесс использования компонента MediaPlayer в приложении. При желании перечень поддерживаемых проигрывателем типов файлов можно с легкостью расширить, изменив лишь фильтр окна выбора файла.

## Проигрыватель компакт-дисков

Процесс управления воспроизведением компакт-диска несколько отличается от управления воспроизведением отдельного файла. Главным отличием является то, что компакт-диск может содержать несколько дорожек.

Внешний вид формы проигрывателя компакт-дисков на этапе разработки показан на рис. 4.5.


**Рис. 4.5.** Проигрыватель компакт-дисков на этапе разработки

Как видите, по сравнению с проигрывателем файлов, рассмотренным выше, в проигрывателе компакт-дисков появились новые команды: команда переключения дорожек, а также команда извлечения диска. К тому же добавилась надпись, отражающая номер воспроизводимой дорожки.

В плане управления внешним видом формы проигрыватель реализован практически так же, как и предыдущий пример. Единственное, что изменилось, так это названия состояний, которые теперь показывают, что проигрыватель работает с компакт-дисками, а не с файлами (листинг 4.8).

**Листинг 4.8.** Состояния проигрывателя компакт-дисков

```
class TForm1 : public TForm
{
...
private:  // User declarations
  //Состояние проигрывателя (для правильного оформления
  //и поведения формы)
  enum TPlayerState
  {
    psNoDisc,     //Первоначальное состояние (нечего
                  //воспроизводить)
    psDiscOpened, //Задан диск, можно воспроизводить
    psPlaying,    //Идет воспроизведение
    psPaused      //Пауза
  }m_State;
  void RefreshControls();
...
};
```

Изначально проигрыватель находится в состоянии `psNoDisc`. Соответствующее значение переменной `m_State` присваивается в обработчике `FormCreate` класса `TForm1`.

Текст реализации функции RefreshControls, учитывающий новые кнопки проигрывателя, приведен в листинге 4.9.

**Листинг 4.9.** Оформление формы в соответствии с состоянием

```
void TForm1::RefreshControls()
{
  //Оформление компонентов в соответствии с состоянием
  //проигрывателя
  switch ( m_State )
  {
  case psNoDisc:
    cmbOpen->Enabled = true;
    cmbPlay->Enabled = false;
    cmbPause->Enabled = false;
    cmbStop->Enabled = false;
    cmbForw->Enabled = false;
    cmbBack->Enabled = false;
    cmbNext->Enabled = false;
    cmbPrev->Enabled = false;
    cmbEject->Enabled = true;
    refresh_timer->Enabled = false;
    break;
  case psDiscOpened:
    cmbOpen->Enabled = true;
    cmbPlay->Enabled = true;
    cmbPause->Enabled = false;
    cmbStop->Enabled = false;
    cmbForw->Enabled = true;
    cmbBack->Enabled = true;
    cmbNext->Enabled = true;
    cmbPrev->Enabled = true;
    cmbEject->Enabled = true;
    refresh_timer->Enabled = false;
    break;
  case psPlaying:
    cmbOpen->Enabled = false;
    cmbPlay->Enabled = false;
    cmbPause->Enabled = true;
    cmbStop->Enabled = true;
    cmbForw->Enabled = true;
    cmbBack->Enabled = true;
    cmbNext->Enabled = true;
    cmbPrev->Enabled = true;
    cmbEject->Enabled = false;
    refresh_timer->Enabled = true;
    break;
```

```
  case psPaused:
    cmbOpen->Enabled = true;
    cmbPlay->Enabled = true;
    cmbPause->Enabled = false;
    cmbStop->Enabled = true;
    cmbForw->Enabled = true;
    cmbBack->Enabled = true;
    cmbNext->Enabled = true;
    cmbPrev->Enabled = true;
    cmbEject->Enabled = true;
    refresh_timer->Enabled = false;
    break;
  }
  RefreshTrack();
}
```

Как можно заметить, в конце функции `RefreshControls` вызывается другая вспомогательная функция с именем `RefreshTrack`. Эта функция во многом схожа с функцией `RefreshTimePos` из предыдущего примера: она отображает номер воспроизводимой дорожки и позицию в этой дорожке (листинг 4.10).

**Листинг 4.10.** Отображение номера дорожки и позиции воспроизведения

```
void TForm1::RefreshTrack()
{
  //Обновление показаний номера дорожки и позиции на дорожке
  switch ( m_State )
  {
  case psNoDisc:
    lblPos->Caption = "";
    lblTrack->Caption = "";
    break;
  case psDiscOpened:
  case psPlaying:
  case psPaused:
    {
      char buff[20];
      //Выводим текущее положение воспроизведения
      player->TimeFormat = tfTMSF;
      long pos = player->Position;
      long track = MCI_TMSF_TRACK(pos);
      long mm = MCI_TMSF_MINUTE(pos);
      long ss = MCI_TMSF_SECOND(pos);
      sprintf( buff, "%0.2d:%0.2d", mm, ss );
      lblPos->Caption = buff;
      //Покажем также номер дорожки
      sprintf( buff, "Дорожка: %0.2d/%0.2d", track, player-
>Tracks );
```

```
      lblTrack->Caption = buff;
    }
    break;
  }
}
```

Обновление индикатора позиции воспроизведения и номера дорожки происходит по таймеру почти так же, как и в предыдущем примере (листинг 4.11).

**Листинг 4.11.** Обновление номера дорожки и позиции воспроизведения

```
void __fastcall TForm1::refresh_timerTimer(TObject *Sender)
{
  if ( m_State == psPlaying && player->Mode != mpPlaying )
  {
    //Закончилось воспроизведение
    m_State = psDiscOpened;
    RefreshControls();
  }
  else
  {
    //Просто обновим показания счетчика воспроизведения
    RefreshTrack();
  }
}
```

Наконец, самой простой частью проигрывателя является реализация обработчиков кнопок, управляющих работой проигрывателя. Данные обработчики реализованы крайне просто, а потому все они, кроме обработчиков кнопок перемотки назад и вперед, приведены в листинге 4.12.

**Листинг 4.12.** Управление работой проигрывателя

```
void __fastcall TForm1::cmbOpenClick(TObject *Sender)
{
  //Попытка открыть диск
  player->Open();
  if ( player->Error )
  {
    //Не удалось открыть диск
    Application->MessageBox( player->ErrorMessage.c_str(),
                        Application->Title.c_str(), MB_
ICONEXCLAMATION );
  }
  m_State = (player->Mode == mpStopped) ? psDiscOpened :
psNoDisc;
  RefreshControls();
}
```

```
void __fastcall TForm1::cmbPlayClick(TObject *Sender)
{
  //Начало/продолжение воспроизведения
  m_State = psPlaying;
  RefreshControls();
  player->Play();
}
void __fastcall TForm1::cmbPauseClick(TObject *Sender)
{
  //Остановка воспроизведения
  player->Pause();
  m_State = psPaused;
  RefreshControls();
}
void __fastcall TForm1::cmbStopClick(TObject *Sender)
{
  //Остановка воспроизведения с перемоткой к началу дорожки
  player->Stop();
  player->Rewind();
  m_State = psDiscOpened;
  RefreshControls();
}
void __fastcall TForm1::cmbPrevClick(TObject *Sender)
{
  //Переход к предыдущей дорожке
  player->Previous();
  RefreshControls();
}
void __fastcall TForm1::cmbNextClick(TObject *Sender)
{
  //Переход к следующей дорожке
  player->Next();
  RefreshControls();
}
void __fastcall TForm1::cmbEjectClick(TObject *Sender)
{
  //Извлечение диска
  player->Eject();
  RefreshControls();
}
```

В рассматриваемом примере хотелось бы реализовать перемотку таким образом, чтобы текущая позиция смещалась на 10 % в пределах одной дорожки, а не всего диска, поэтому в обработчики кнопок для перемотки назад и вперед были добавлены алгоритмы соответствующих проверок. Так, при перемотке вперед, если новая

позиция воспроизведения выходит за пределы текущей дорожки, проигрыватель переключается на начало следующей дорожки (листинг 4.13).

**Листинг 4.13.** Перемотка вперед

```
void __fastcall TForm1::cmbForwClick(TObject *Sender)
{
  //Перемотка вперед на 10 % длины дорожки
  player->TimeFormat = tfTMSF;
  int track = MCI_TMSF_TRACK(player->Position);
  player->TimeFormat = tfMilliseconds;
  long new_pos = player->Position + player->
TrackLength[track]/10;
  if (new_pos < player->Length && new_pos < player->
TrackPosition[track+1])
    player->Position = new_pos;
  else
    //Переходим в начало следующей дорожки
    player->Position = player->TrackPosition[track+1];
  RefreshControls();
}
```

При перемотке назад, наоборот, проверяется, чтобы рассчитанная позиция воспроизведения не выходила за начало текущей дорожки, а в случае возникновения подобной ситуации выполнялась корректировка позиции путем установки на начало текущей дорожки (листинг 4.14). Это означает, что кнопка перемотки назад в данном примере действует только в пределах одной дорожки.

**Листинг 4.14.** Перемотка назад

```
void __fastcall TForm1::cmbBackClick(TObject *Sender)
{
  //Перемотка назад на 10 % длины дорожки
  player->TimeFormat = tfTMSF;
  int track = MCI_TMSF_TRACK(player->Position);
  player->TimeFormat = tfMilliseconds;
  long new_pos = player->Position - player->
TrackLength[track]/10;
  if ( new_pos >= player->TrackPosition[track] )
    player->Position = new_pos;
  else
    //Перемотали в начало текущей дорожки
    player->Position = player->TrackPosition[track];
  RefreshControls();
}
```

Теперь можно вставлять в дисковод компакт-диск и наслаждаться музыкой, используя собственный проигрыватель. Внешний вид приложения при воспроизведении диска показан на рис. 4.6.

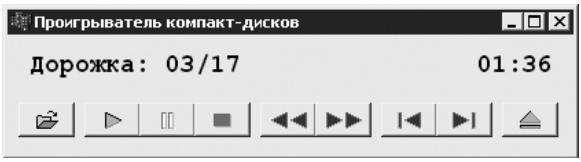


**Рис. 4.6.** Проигрыватель при воспроизведении диска

На этом завершается рассмотрение стандартных мультимедиа-компонентов, поставляемых с Borland C++ Builder. В следующих разделах приведен ряд примеров, демонстрирующих, как можно обойтись только средствами Windows API при выполнении самых востребованных операций работы со звуком.

# Использование Windows API для работы со звуком

В этом разделе будут рассмотрены несколько функций, которыми обладает Windows API, позволяющих воспроизводить разнообразные звуки: простой звук с заданной длительностью и частотой, звуки стандартных сообщений Windows и звуковые файлы формата WAV.

## Воспроизведение звука с помощью встроенного динамика

Сначала будет рассмотрен самый простой случай, когда требуется воспроизвести звук с помощью встроенного динамика компьютера. Для этого можно использовать API-функцию `Beep`:

```
BOOL Beep(
  DWORD dwFreq,     //Частота звука в герцах
  DWORD dwDuration //Длительность звука в миллисекундах
);
```

Функция возвращает `true` в случае успеха и `false`, если по какой-то причине звук воспроизвести не удалось. Использовать функцию `Beep` крайне просто, например:

```
Beep(200, 1000);  //Воспроизводит звук с частотой 200Гц
                  //в течение 1 секунды
Beep(1000, 1000); //Воспроизводит звук с частотой 1000Гц
                  //в течение 1 секунды
Beep(5000, 1000); //Воспроизводит звук с частотой 5000Гц
                  //в течение 1 секунды
```

На этом, пожалуй, повествование о функции `Beep` можно и завершить. Вряд ли в наше время можно найти широкое применение этой функции — разве что она

может оказаться полезной в детских игрушках наподобие простого синтезатора, рассмотренного ниже.

## Простой синтезатор

Казалось бы, почти бесполезной функции `Beep` можно найти довольно интересное применение, если попытаться превратить компьютер в простой синтезатор. Именно это и будет реализовано в данном подразделе.

Внешний вид программы-синтезатора приведен на рис. 4.7 — в данном случае воспроизводится нота соль.


**Рис. 4.7.** Внешний вид синтезатора

Для оформления внешнего вида приложения достаточно поместить на форму компонент `Image`, который и будет использоваться для отображения клавиатуры синтезатора и подсветки проигрываемой в текущий момент ноты. Как видно на рис. 4.7, данный синтезатор рассчитан всего на одну октаву. Чтобы показать все состояния синтезатора, придется запастись 14 изображениями: одно изображение для состояния, когда ни одна клавиша не нажата, и 13 изображений для случая нажатия каждой из нот.

**ПРИМЕЧАНИЕ**

Все рисунки, необходимые для создания синтезатора, вы найдете на прилагаемом к книге компакт-диске.

Теперь о том, как можно закрепить за клавишами на клавиатуре ноты и соответствующие изображения. В данном примере такая связь устанавливается с помощью массива структур, заполняемого при инициализации приложения. Тексты объявления структур приведены в листинге 4.15.

**Листинг 4.15.** Структура с информацией об одной ноте

```
struct TNoteInfo
{
  long Frequency;    //Частота звука, соответствующего ноте
  WORD Key;          //Код клавиши, закрепленной за нотой
  String ImageFile;  //Имя файла рисунка клавиатуры для ноты
  Graphics::TBitmap *Image; //Рисунок клавиатуры для ноты
  TNoteInfo( long freq, WORD key, String imagefile )
  {
    Frequency = freq;
    Key = key;
    ImageFile = imagefile;
    //Загрузка картинки из файла
```

```
    Image = new Graphics::TBitmap;
    Image->LoadFromFile( ImageFile );
  }
  ~TNoteInfo()
  {
    delete Image;
  }
};
```

В классе `TForm1` для хранения массива структур `TNoteInfo` присутствует переменная `m_Notes`, а для хранения количества записей в массиве — переменная `m_NotesCount`. Код заполнения массива `m_Notes` приведен в листинге 4.16.

**Листинг 4.16.** Связывание клавиш с нотами и рисунками синтезатора

```
#include <Qt.hpp> // В этом файле объявлены константы кодов
клавиш
...
//Заполнение информации о нотах
TNoteInfo TForm1::m_Notes[] =
{
  TNoteInfo(-1, 0, "Keys\\00.bmp"),
  TNoteInfo(262, Key_S, "Keys\\01.bmp"), //до
  TNoteInfo(294, Key_D, "Keys\\02.bmp"), //ре
  TNoteInfo(330, Key_F, "Keys\\03.bmp"), //ми
  TNoteInfo(349, Key_G, "Keys\\04.bmp"), //фа
  TNoteInfo(392, Key_H, "Keys\\05.bmp"), //соль
  TNoteInfo(440, Key_J, "Keys\\06.bmp"), //ля
  TNoteInfo(494, Key_K, "Keys\\07.bmp"), //си
  TNoteInfo(523, Key_L, "Keys\\08.bmp"), //до
  TNoteInfo(277, Key_E, "Keys\\09.bmp"), //до#
  TNoteInfo(311, Key_R, "Keys\\10.bmp"), //ре#
  TNoteInfo(370, Key_Y, "Keys\\11.bmp"), //фа#
  TNoteInfo(415, Key_U, "Keys\\12.bmp"), //соль#
  TNoteInfo(460, Key_I, "Keys\\13.bmp")  //ля#
};
//Вычисление количества нот
int TForm1::m_NotesCount = sizeof(m_Notes) /
sizeof(TNoteInfo);
```

В данном случае за клавишами на клавиатуре были закреплены ноты первой октавы путем установления соответствующих частот нот.

Для завершения реализации интерфейса программы осталось только добавить в код обработчики нажатия и отпускания клавиш. Ниже приведен код обработчика нажатия клавиш (листинг 4.17).

**Листинг 4.17.** Обработка нажатия и отпускания клавиш

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD
&Key,
      TShiftState Shift)
{
  for ( int i=1; i<m_NotesCount; i++ )
  {
    if ( m_Notes[i].Key == Key )
    {
      //Определяем частоту ноты
      long freq = m_Notes[i].Frequency;
      if ( Shift.Contains(ssShift) )
      {
        //Shift опускает ноту на октаву ниже
        freq /= 2;
      }
      else if ( Shift.Contains(ssCtrl) )
      {
        //Ctrl поднимает ноту на октаву выше
        freq *= 2;
      }
      //Начинаем воспроизведение ноты
      m_Synth.BeginPlay( freq );
      imgKeys->Picture->Bitmap = m_Notes[i].Image;
      return;
    }
  }
  //Никакая нота не воспроизводится
  m_Synth.EndPlay();
  imgKeys->Picture->Bitmap = m_Notes[0].Image;
}
```

Если на клавиатуре нажата клавиша, соответствующая одной из нот, то вызывается метод `BeginPlay` пока неопределенного объекта `m_Synth`. Вызов этого метода начинает воспроизведение ноты, при этом в компоненте `Image` отображается рисунок, соответствующий текущему состоянию синтезатора.

Как видно из листинга 4.17, несмотря на то что заданы частоты только нот первой октавы, диапазон синтезатора был легко расширен до трех октав. Так, если нажата клавиша Shift, то синтезатор воспроизводит ноты малой октавы, а если нажата клавиша Ctrl — ноты второй октавы. Все дело в закономерности: частоты звуков, отличающихся на одну октаву, различаются ровно в два раза.

Для прекращения звучания ноты достаточно отпустить соответствующую клавишу на клавиатуре — при этом будет выполнен обработчик события `KeyDown`, описание которого приведено в листинге 4.18.

**Листинг 4.18.** Прекращение звучания ноты

```
void __fastcall TForm1::FormKeyUp(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
  //Завершение воспроизведения ноты
  m_Synth.EndPlay();
  imgKeys->Picture->Bitmap = m_Notes[0].Image;
}
```

Теперь можно сказать, что проигрыватель почти готов. Почти — потому что остался один нюанс, а именно класс `TSynth`, в котором реализовано то, без чего данный синтезатор не смог бы работать, — асинхронное воспроизведение звука. Дело в том, что вызов функции `Beep` блокирует приложение на время воспроизведения звука, а с этим, честно говоря, менее всего хочется столкнуться при построении синтезатора, который должен незамедлительно реагировать на нажатия клавиш.

Предлагаемое решение основано на идее выполнять функцию `Beep` в отдельном потоке. В таком случае во время вызова функции `Beep` будет блокироваться вызывающий его дополнительный поток, а не основной поток приложения. Это даст возможность продолжать обработку пользовательского ввода, то есть своевременно определить момент, когда пользователь отпустит клавишу, чтобы вовремя прервать звучание ноты.

Тексты реализации класса `TSynth`, а также класса вспомогательного потока `TBeepThread` приведены в листинге 4.19.

**Листинг 4.19.** Реализация асинхронного воспроизведения звука

```
//Класс для непрерывного воспроизведения звука в отдельном
//потоке
class TBeepThread : public TThread
{
  long m_Frequency; //Частота вопроизводимого звука
public:
  TBeepThread( long frequency ) : TThread( false )
  {
    m_Frequency = frequency; //Запоминаем частоту звука
  }
protected:
  virtual void __fastcall Execute()
  {
    //Воспроизводим звук заданной частоты,
    //пока не будет остановлен поток
    while( !Terminated )
    {
      ::Beep( m_Frequency, 50 );
```

```
    }
  }
};
//Класс "Синтезатора", воспроизводящего заданный звук
class TSynth
{
  TBeepThread *m_pPlayer;  //Поток, воспроизводящий звук
public:
  TSynth()
  {
    m_pPlayer = NULL;
  }
  ~TSynth()
  {
    EndPlay();
  }
  void BeginPlay( long frequency )
  {
    EndPlay();
    //Запуск воспроизведения нового звука
    m_pPlayer = new TBeepThread( frequency );
  }
  void EndPlay()
  {
    if ( IsPlaying() )
    {
      //Завершаем воспроизведение
      m_pPlayer->Terminate();
      delete m_pPlayer;
      m_pPlayer = NULL;
    }
  }
  bool IsPlaying()
  {
    //Считаем, что звук проигрывается, если создан
    //предназначенный для этого поток
    return m_pPlayer != NULL;
  }
};
```

Принцип совместной работы приведенных в листинге 4.19 классов заключается в следующем. Если нужно начать воспроизведение звука (вызван метод `BeginPlay` объекта класса `TSynth`), то создается объект класса `TBeepThread`, которому через конструктор передается частота нужного звука. Единственная задача вспомогательного потока — непрерывно вызывать функцию `Beep`.

Чтобы завершить воспроизведение звука (при вызове метода `TSynth::EndPlay`), нужно завершить вспомогательный поток. Для этого вызывается метод объекта потока `Terminate`, который, в свою очередь, устанавливает значение флага `Terminated` объекта потока в `true`, а поскольку при реализации вспомогательного потока в данном примере после каждого вызова функции `Beep` выполняется проверка значения флага `Terminated`, то вспомогательный тут же завершается.

Теперь пример полностью готов, и можно порадовать себя, младшую сестру или брата возможностью поиграть на самодельном синтезаторе.

## Звуки сообщений Windows

Вспомните стандартные окна сообщений Windows: каждому типу окна соответствует свой собственный звук, причем звуки могут отличаться в зависимости от настроек Windows. Если когда-нибудь возникнет необходимость воспроизвести звуки, не показывая самих окон сообщений (например, при реализации собственных окон сообщений), на помощь может прийти API-функция `MessageBeep`:

```
BOOL MessageBeep(UINT uType);
```

Единственным параметром этой функции является идентификатор воспроизводимого звука. В случае успешного воспроизведения функция возвращает значение `true`, а в противном случае — `false`.

Идентификаторы звуков (за исключением одного значения) задаются с помощью тех же констант, что и идентификаторы типов сообщений в функции `MessageBox`:

- `-1` — вызывает звук системного динамика;
- `MB_OK` — воспроизводит звук по умолчанию;
- `MB_ICONASTERISK` — звук, соответствующий вызову окна сообщения с информацией;
- `MB_ICONEXCLAMATION` — воспроизводит звук, соответствующий вызову окна сообщения об ошибке;
- `MB_ICONHAND` — звук, соответствующий вызову окна сообщения о критической ошибке;
- `MB_ICONQUESTION` — воспроизводит звук, соответствующий вызову окна с вопросом.

В завершение приведу пару примеров вызова функции `MessageBeep`:

```
MessageBeep(MB_ICONQUESTION); //Вопрос
MessageBeep(MB_ICONEXCLAMATION); //Восклицание
```

## Воспроизведение звуковых файлов и не только

Последней рассматриваемой API-функцией, предназначенной для работы со звуком, будет универсальная функция `PlaySound`. Она позволяет воспроизводить

стандартные звуки, закрепленные за основными событиями Windows; звуковые файлы формата WAV; звуковые данные из ресурсов, а также звуковые данные, сохраненные в оперативной памяти.

Способ объявления функции `PlaySound` выглядит следующим образом:

```
BOOL PlaySound(LPCSTR pszSound, HMODULE hmod, DWORD
fdwSound);
```

Смысл первого параметра функции, несмотря на то что в ее объявлении он имеет строковый тип, зависит от значения флагов, передаваемых в третьем параметре функции. Второй параметр `hmod` используется только в том случае, если с помощью функции `PlaySound` требуется воспроизвести звук, сохраненный в секции ресурсов исполняемого файла, — во всех остальных случаях он должен иметь значение `NULL`. Работа с ресурсами, в том числе и способы воспроизведения звуков, сохраненных в ресурсах приложений, будет рассмотрена позже в гл. 7.

Наконец, третий параметр с именем `fdwSound` представляет собой набор флагов, определяющих режим работы функции `PlaySound`. Флаги объединяются с помощью операции побитового ИЛИ, и их можно условно разделить на несколько групп. Первая группа флагов определяет смысл параметра `lpszSound`.

- `SND_ALIAS` — если данный флаг установлен, то считается, что параметр `lpszSound` является псевдонимом (alias) звука, закрепленного за определенным системным событием. Значения и описания распространенных псевдонимов будут приведены ниже.
- `SND_FILENAME` — если данный флаг установлен, то считается, что параметр `lpszSound` указывает путь файла на диске.
- `SND_MEMORY` — если данный флаг установлен, то считается, что адрес, записанный в параметр `lpszSound` (ведь это указатель), является адресом в памяти, по которому размещены специальным образом отформатированные звуковые данные. Этот вариант будет подробно рассмотрен в следующем разделе главы.
- `SND_RESOURCE` — если данный флаг установлен, то считается, что значение параметра `lpszSound` является идентификатором ресурса, в котором хранятся звуковые данные.

Очевидно, что одновременно может быть установлен лишь один из приведенных выше флагов. Однако в сочетании с рассмотренными флагами могут использоваться приведенные ниже флаги, определяющие режим работы функции.

- `SND_ASYNC` — если данный флаг установлен, то звук воспроизводится асинхронно, то есть выполнение программы не приостанавливается во время его звучания. Чтобы остановить асинхронное воспроизведение, достаточно вызвать функцию `PlaySound` с параметром `lpszSound`, равным `NULL`.

- SND_SYNC — установка этого флага, напротив, указывает функции PlaySound, что звук нужно воспроизвести в синхронном режиме. В этом случае возврат из функции PlaySound происходит только после окончания воспроизведения (аналогично функции Beep).
- SND_LOOP — этот флаг может использоваться только при воспроизведении звука в асинхронном режиме (то есть с флагом SND_ASYNC). При его установке воспроизведение повторяется по кругу до тех пор, пока функция PlaySound не будет вызвана с параметром lpszSound, равным NULL.
- SND_NODEFAULT — установка данного флага отменяет воспроизведение звука по умолчанию при неудачной попытке его воспроизведения (по умолчанию, если функции PlaySound не удалось воспроизвести заданный звук, она пытается воспроизвести системный звук, заданный по умолчанию).

Выше приведены не все флаги функции PlaySound, а лишь те, которые могут пригодиться в рассмотренных в данной главе примерах.

Для демонстрации возможностей функции PlaySound мной был разработан небольшой пример, однако в книге он рассматриваться не будет, так как его реализация основана на очень простой идее: паре вызовов функции PlaySound. Тем не менее пример доступен на прилагаемом к книге компакт-диске. Ниже приведено несколько примеров вызовов функции PlaySound, демонстрирующих особенности ее использования.

Так, чтобы воспроизвести файл Sound.wav, хранящийся на диске D: в синхронном режиме, достаточно выполнить следующий вызов:

```
PlaySound("D:\Sound.wav", NULL, SND_FILENAME | SND_SYNC);
```

Чтобы запустить воспроизведение этого же файла по кругу в асинхронном режиме, достаточно выполнить такой вызов:

```
PlaySound("D:\Sound.wav", NULL, SND_FILENAME | SND_ASYNC |
SND_LOOP);
```

Для остановки воспроизведения достаточно добавить в программу следующую строку:

```
PlaySound(NULL, NULL, 0);
```

Также, чтобы воспроизвести звук, который Windows обычно проигрывает при очистке корзины, можно выполнить такой вызов:

```
PlaySound("EmptyRecycleBin", NULL, SND_ALIAS | SND_SYNC);
```

Здесь параметр "EmptyRecycleBin" и является псевдонимом соответствующего звука. В табл. 4.1 приведены значения и описания основных псевдонимов звуков, используемых Windows.

**Таблица 4.1.** Псевдонимы звуков для событий Windows

| Псевдоним | Событие |
|---|---|
| .Default | Стандартный звук (звук по умолчанию) |
| AppGPFault | Ошибка приложения |
| Close | Закрытие приложения |
| EmptyRecycleBin | Очистка Корзины |
| MailBeep | Уведомление о приходе почты |
| Maximize | Развертывание окна |
| MenuCommand | Выбор команды меню |
| MenuPopup | Открытие всплывающего меню |
| Minimize | Сворачивание окна |
| Open | Открытие приложения |
| RestoreDown | Переход развернутого окна в нормальное состояние |
| RestoreUp | Восстановление окна из значка |
| SystemAsterisk | Вызов окна сообщения с информацией |
| SystemExclamation | Вызов окна с восклицанием |
| SystemExit | Выход из Windows |
| SystemHand | Критическая ошибка |
| SystemQuestion | Вызов окна с вопросом |
| SystemStart | Запуск Windows |

На этом, пожалуй, можно завершить повествование о возможностях работы со звуком Windows API. Описанная здесь функция `PlaySound` в действительности является одним из самых простых средств работы со звуковыми данными, но предоставляемых ей возможностей более чем достаточно не только для создания обычных приложений, где звуковые эффекты используются лишь время от времени, да и то чтобы развлечь пользователя, но и для построения приложений, позволяющих полноценно обрабатывать звук. В этом вы сможете убедиться во время изучения следующего раздела.

# Низкоуровневая работа со звуком

В завершение главы будут рассмотрены наиболее интересные (по крайней мере на мой взгляд) примеры. Первая из приведенных здесь программ предназначена для генерации звуков. Вторая же программа иллюстрирует способ создания небольшого звукового редактора.

## Цифровое кодирование звука

Для полного понимания описываемых здесь примеров необходимо знание о внутреннем представлении цифровых аудиоданных, поэтому для начала хотя бы в общих чертах будут рассмотрены способы хранения звуковых данных с помощью компьютера.

На самом деле ничего особо сложного в цифровом кодировании звука нет. Ниже рассмотрен процесс оцифровки звука в сравнении с его записью на такой носитель, как магнитная лента для обычного (аналогового) магнитофона. Например, необходимо записать звук гармонического колебания с частотой 1000 Гц (рис. 4.8).

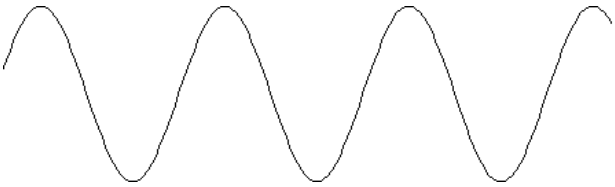

**Рис. 4.8.** Синусоида с частотой 1000 Гц

Если для записи использовать аналоговый магнитофон, то изменения интенсивности звука, которые видны на рис. 4.8, будут с большой точностью сохранены на магнитной ленте путем плавного изменения намагниченности ее участков (помехи, вносимые электронными узлами магнитофона, не учитываются). Ключевое слово тут — «плавного», то есть синусоида изменения интенсивности звука будет преобразована в аналогичную синусоиду изменения напряжения на входе усилителя и в итоге в изменения намагниченности ленты кассеты. При воспроизведении произойдет обратное преобразование и будет проигран тот же звук, что и записывался (опять же, если не учитывать искажения, вносимые усилителем магнитофона).

Если же необходимо сохранить звук на таком носителе, как, к примеру, жесткий диск, то возникает ограничение: на цифровом носителе информация может быть записана только в дискретном виде, то есть двоичным числом. Особенности же сохранения информации конкретным жестким диском остаются, конечно, неизвестны, но и знать это незачем.

Для сохранения звука в цифровом виде достаточно определиться с выбором следующих значений.

1. Ввести шкалу интенсивности звука (например, от –127 до 128).
2. Задать достаточно малый промежуток времени, например 0,022 мс. Величина, обратная величине этого промежутка времени, называется частотой дискретизации. В данном примере она составляет около 44 кГц.
3. Измерить интенсивность звука через указанный промежуток времени, используя при этом для оценки интенсивности введенную в п. 1 шкалу. В данном случае измерения проводятся 44 000 раз в секунду.

Если таким образом обработать исходный звук, то на выходе получится последовательность цифр (значений интенсивности), которые часто называют сэмплами (от англ. sample).

Зная частоту дискретизации, которая использовалась при измерениях, и диапазон шкалы, можно воспроизвести звук, который будет мало чем отличаться от ориги-

нала. Чем выше частота дискретизации, тем выше качество звука, но и тем больше места полученный звуковой файл занимает на диске.

Собственно, упомянутая выше последовательность сэмплов, значение частоты дискретизации, диапазон шкалы и еще некоторая служебная информация и составляют то, что записано в файле, хранящем несжатую монофоническую аудиозапись.

Соответственно, если необходимо сохранить стереозвук, то нужно оцифровать звук двух каналов, то есть сохранить сэмплы обоих каналов, в результате чего размер файла возрастет приблизительно в два раза.

В завершение, возможно, уже наскучившей вам теоретической части ради интереса посчитаем, сколько может занимать файл несжатого аудио. К примеру, записывается по меньшей мере 16-битный стереозвук (шкала от –32 767 до 32 768) хорошего качества; оцифрованный при частоте дискретизации, равной 44 кГц, звук длится одну минуту. Несложный расчет будет выглядеть следующим образом:

2 байт × 44 000 × 60 × 2 = 10 560 000 байт = 10,1 Мбайт

В то же время сжатый звук в формате MP3, который будет звучать немного хуже (и далеко не каждый это может заметить), будет занимать ориентировочно раз в 10 меньше. Таким образом, неудивительно, что в наше время можно найти относительно немного истинных ценителей музыки, хранящих ее на компьютере в несжатом виде, то есть в файлах WAV или подобных ему форматах.

## Генератор звуков

Как бы ни было расточительно хранить аудиоданные в несжатом виде, но именно в таком виде их удобнее всего использовать. По крайней мере это верно при написании простых программ, работающих со звуком, которые представлены в этом подразделе.

Описанная здесь программа позволяет генерировать несколько простых звуков и записывать их в файл, а именно: звук с частотой 1000 Гц, состоящий из импульсов синусоидальной, треугольной и прямоугольной формы, а также шум, представляющий собой случайное изменение интенсивности звука (рис. 4.9).

Внешний вид программы приведен на рис. 4.10.

Переключатель, имеющий четыре положения, предназначен для выбора воспроизводимого или записываемого в файл звука. Назначение кнопок, как мне думается, понятно из их названий.

Коды обработчиков кнопок `Воспроизвести` и `Остановить` приведены в листинге 4.20.

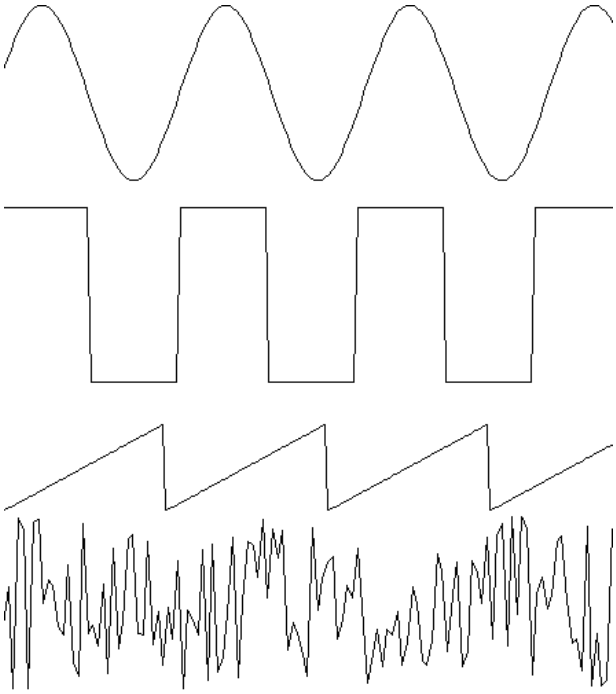

**Рис. 4.9.** Графики генерируемых звуков

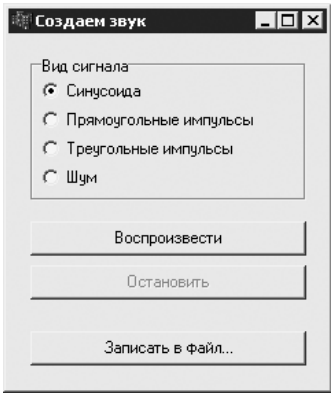


**Рис. 4.10.** Внешний вид генератора звуков

**Листинг 4.20.** Начало и остановка воспроизведения

```
void __fastcall TForm1::cmbPlayClick(TObject *Sender)
{
  //Запускаем воспроизведение сигнала нужной формы
  SignalFunctionPtr pSignalFunc;
  switch ( SoundType->ItemIndex )
  {
  case 0: pSignalFunc = SinusSignal; break;
  case 1: pSignalFunc = SquareSignal; break;
  case 2: pSignalFunc = TriangleSignal; break;
  case 3: pSignalFunc = NoiseSignal; break;
  }
  //Пытаемся воспроизвести сигнал, сгенерированный выбранной
  //функцией
  if ( StartPlay( pSignalFunc, 1000, 44000, 1000 ) )
  {
    cmbStop->Enabled = true;
    cmbPlay->Enabled = false;
  }
  else
  {
```

```
    Application->MessageBox( "Не удалось воспроизвести звук",
"Ошибка",
                             MB_ICONEXCLAMATION );
  }
}
void __fastcall TForm1::cmbStopClick(TObject *Sender)
{
  //Останавливаем воспроизведение
  StopPlay();
  cmbPlay->Enabled = true;
  cmbStop->Enabled = false;
}
```

Как видно из предложенного листинга, в обработчике для кнопки Воспроизвести вызывается вспомогательная функция `StartPlay`, прототип которой выглядит следующим образом:

```
BOOL StartPlay( SignalFunctionPtr pSignal, DWORD dwFrequency,
                DWORD dwSamplesPerSec, DWORD dwLength );
```

Вторым параметром функции является частота звука, который необходимо получить. Третьим параметром — частота дискретизации или число сэмплов в секунду. Четвертым параметром — длина генерируемого звука в миллисекундах. Наконец, первым параметром функции `StartPlay` является указатель на функцию, которая и позволяет получить сигнал заданной формы. Объявление типа `SignalFunctionPtr` и тексты четырех реализаций функций, генерирующих сигнал заданной формы, приведены в листинге 4.21.

**Листинг 4.21.** Функции генерации сигнала разной формы

```
//Объявление указателя на функцию
typedef WORD (*SignalFunctionPtr)( float t );
//Функция для генерации синусоиды
WORD SinusSignal( float t )
{
  return 32767 * sin(2*M_PI*t);
}
//Функция для генерации прямоугольных импульсов
WORD SquareSignal( float t )
{
  return (t - (int)t) < 0.5 ? -32768 : 32767;
}
//Функция для генерации треугольных импульсов
WORD TriangleSignal( float t )
{
  return (t - (int)t) * 32767;
}
//Функция для генерации шума
```

```
WORD NoiseSignal( float t )
{
  return 65535 * rand() / RAND_MAX - 32768;
}
```

Идея, положенная в основу функции `StartPlay`, предельно проста. На входе создается периодическая функция, которая предоставляет 16-битные сэмплы, то есть численные значения интенсивности звука, в пределах периода. Остается лишь записать эти сэмплы в участок памяти определенным образом, чтобы их могла воспроизвести уже много раз упоминавшаяся в этой главе API-функция `PlaySound`.

Формат данных, принимаемых функцией `PlaySound`, точно такой же (насколько я знаю, по крайней мере), как и формат файлов WAV. Для монофонического аудиофайла формат следующий: заголовок, за которым следуют данные сэмплов. В действительности же все немного сложнее, но в данном случае такого описания более чем достаточно. Код объявления структуры заголовка приведен в листинге 4.22.

**Листинг 4.22.** Заголовок WAV-файла

```
struct WAVEHEADER
{
   char    sigRIFF[4];      // Должно быть равно "RIFF"
   DWORD   sizeRIFFch;      // Размер блока RIFF
   char    sigWAVE[4];      // Должно быть равно "WAVE"
   char    sigFMT[4];       // Должно быть равно "fmt"
   DWORD   sizeFMTch;       // Размер FMT
   WORD    wFormatTag;      // Категория формата, для PCM = 1
   WORD    wChannels;       // Количество каналов: 1 - моно 2 -
                            //стерео
   DWORD   dwSamplesPerSec; // Количество сэмплов в секунду
                            // (частота дискретизации)
   DWORD   dwAvgBytesPerSec;// Среднее количество байт
                            //в секунду
   WORD    wBlockAlign;     // Выравнивание данных
   WORD    wBitPerSample;   // Бит в сэмпле
   char    sigDATA[4];      // Должно быть равно "data"
   DWORD   sizeDATAch;      // Размер звуковых данных
};
```

Наконец, текст реализации самой функции `StartPlay` приведен в листинге 4.23.

**Листинг 4.23.** Генерация и воспроизведение звука

```
BOOL StartPlay( SignalFunctionPtr pSignal, DWORD dwFrequency,
                DWORD dwSamplesPerSec, DWORD dwLength )
{
  if ( pWaveBuffer != NULL ) return false;
```

```
  //Рассчитываем количество сэмплов и выделяем память для
  //звуковых данных
  DWORD dwSamples = dwSamplesPerSec * dwLength / 1000;
  DWORD dwSize = sizeof(WAVEHEADER) + dwSamples *
sizeof(WORD);
  pWaveBuffer = new BYTE[dwSize];
  //Заполняем заголовок WAVE-данных
  WAVEHEADER *pHead = (WAVEHEADER*)pWaveBuffer;
  memcpy(pHead->sigRIFF, "RIFF", 4);
  memcpy(pHead->sigWAVE, "WAVE", 4);
  pHead->sizeRIFFch = dwSize - 8;
  memcpy(pHead->sigFMT, "fmt ", 4);
  pHead->sizeFMTch = 16;
  pHead->wFormatTag = 1;
  pHead->wChannels = 1;
  pHead->dwSamplesPerSec = dwSamplesPerSec;
  pHead->dwAvgBytesPerSec = dwSamplesPerSec;
  pHead->wBlockAlign = 2;
  pHead->wBitPerSample = 16;
  memcpy(pHead->sigDATA, "data", 4);
  pHead->sizeDATAch = dwSize - sizeof(WAVEHEADER);
  //Формируем набор сэмплов с помощью заданной функции-
  //генератора сигнала
  WORD *samples = (WORD*)(pWaveBuffer + sizeof(WAVEHEADER));
  float fSamplesPerPeriod = dwSamplesPerSec / dwFrequency;
  for(DWORD i=0; i<dwSamples; i++)
  {
    samples[i] = pSignal( i / fSamplesPerPeriod );
  }
  // Проигрываем звук
  return ::PlaySound(pWaveBuffer, NULL, SND_MEMORY | SND_
ASYNC | SND_LOOP);
}
```

Адрес памяти с данными, формируемыми функцией `StartPlay`, сохраняется в указателе `pWaveBuffer` — глобальной переменной в пределах модуля. Описание другой функции — `StopPlay`, вызываемой в обработчике кнопки Остановить, представлено в листинге 4.24. Кроме остановки воспроизведения, эта функция также освобождает ненужный более участок памяти, на который указывает `pWaveBuffer`.

**Листинг 4.24.** Остановка воспроизведения и освобождение памяти

```
void StopPlay()
{
  if ( pWaveBuffer )
  {
```

```
    //Остановим воспроизведение и удалим данные звука
    ::PlaySound(NULL, NULL, 0);
    delete []pWaveBuffer;
    pWaveBuffer = NULL;
  }
}
```

Напоследок пару слов о том, как записать сгенерированный программой звук в файл. Код функции, позволяющей это сделать, приведен в листинге 4.25.

**Листинг 4.25.** Сохранение звука в WAV-файл

```
BOOL SaveSound( const char *filename, SignalFunctionPtr pSig-
nal,

DWORD dwFrequency, DWORD dwSamplesPerSec, DWORD dwLength )
{
  //Рассчитываем количество сэмплов и выделяем память для
  //звуковых данных
  DWORD dwSamples = dwSamplesPerSec * dwLength / 1000;
  DWORD dwSize = sizeof(WAVEHEADER) + dwSamples *
sizeof(WORD);
  BYTE *pBuffer = new BYTE[dwSize];
  //Заполняем заголовок WAVE-данных
  WAVEHEADER *pHead = (WAVEHEADER*)pBuffer;
  memcpy( pHead->sigRIFF, "RIFF", 4 );
  memcpy( pHead->sigWAVE, "WAVE", 4 );
  pHead->sizeRIFFch = dwSize - 8;
  memcpy( pHead->sigFMT, "fmt ", 4 );
  pHead->sizeFMTch = 16;
  pHead->wFormatTag = 1;
  pHead->wChannels = 1;
  pHead->dwSamplesPerSec = dwSamplesPerSec;
  pHead->dwAvgBytesPerSec = dwSamplesPerSec;
  pHead->wBlockAlign = 2;
  pHead->wBitPerSample = 16;
  memcpy( pHead->sigDATA, "data", 4 );
  pHead->sizeDATAch = dwSize - sizeof(WAVEHEADER);
  //Формируем набор сэмплов с помощью заданной функции-
  //генератора сигнала
  WORD *samples = (WORD*)(pBuffer + sizeof(WAVEHEADER));
  float fSamplesPerPeriod = dwSamplesPerSec / dwFrequency;
  for( DWORD i=0; i<dwSamples; i++ )
  {
    samples[i] = pSignal( i / fSamplesPerPeriod );
  }
  //Записываем звуковые данные в файл
```

```
  BOOL bResult = false;
  long hFile = FileCreate(filename);
  if ( hFile != -1 )
  {
    bResult = FileWrite( hFile, pBuffer, dwSize ) ==
(int)dwSize;
  }
  FileClose(hFile);
  delete []pBuffer;
  return bResult;
}
```

В действительности функция `SaveSound` мало чем отличается от приведенной в листинге 4.23 функции `StartPlay`. По сути, единственное отличие состоит в том, что вместо передачи участка памяти со звуковыми данными API-функции `PlaySound` эти данные записываются на диск.

## Редактор звука

Поскольку уже приводилось описание внутреннего строения файлов, содержащих несжатые аудиоданные (WAV), то почему бы теперь не попробовать самостоятельно создать несложный редактор, позволяющий применять к звуковым данным различные эффекты? Собственно, созданию такого редактора и посвящен этот подраздел.

Так, приведенный в данном подразделе пример программы позволяет:

- загружать данные аудиофайлов;
- строить графики звуковых колебаний по данным из аудиофайла;
- ускорять или замедлять воспроизведение;
- увеличивать или уменьшать громкость звука;
- изменять порядок воспроизведения на обратный;
- создавать эффект эха;
- сохранять измененные аудиоданные в файл.

Внешний вид редактора звука приведен на рис. 4.11. Для изменения масштаба графика предусмотрен регулятор в нижней части формы, для навигации по графику — горизонтальная полоса прокрутки. Команды (открытие, сохранение, применение эффектов) доступны в меню.

Работа нашего редактора основана на использовании двух специально созданных классов:

- `TSimpleGraph` — реализует простейший графопостроитель;
- `TWaveManager` — реализует возможность загрузки звуковых данных из файла и сохранения их в файл, а также операций над звуковыми данными.

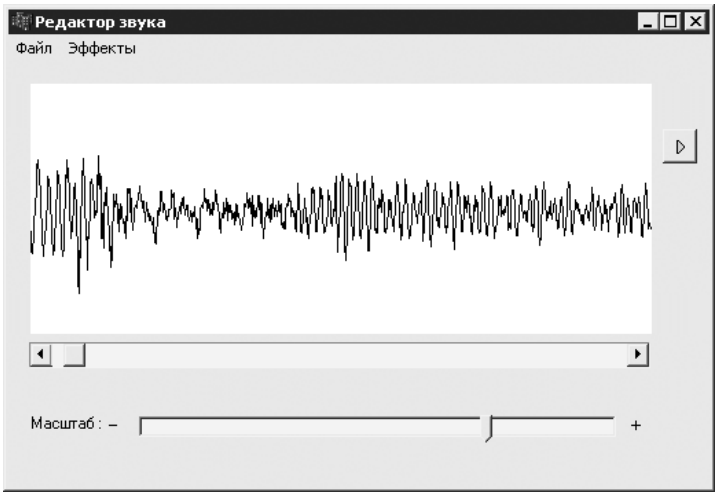


**Рис. 4.11.** Внешний вид редактора звука

Правда, если быть точным, стоит упомянуть еще и о классе формы (`TForm1`), который содержит относительно немного кода, организующего работу разрозненных частей приложения.

## Графопостроитель для звукового редактора

Класс `TSimpleGraph`, предназначенный для построения графика, написан так, чтобы его можно было легко использовать повторно в самых разных приложениях, а не только в данном звуковом редакторе (листинг 4.26).

**Листинг 4.26.** Реализация графопостроителя

```
//Этот интерфейс должен поддерживать источник данных для
//графика
interface ISimpleGraphDataSource
{
  virtual float GetMaxX() = 0;
  virtual float GetMinX() = 0;
  virtual float GetMaxY() = 0;
  virtual float GetMinY() = 0;
  virtual float GetX(long index) = 0;
  virtual float GetY(long index) = 0;
  virtual long GetValuesGount() = 0;
};
// Собственно класс простейшего графопостроителя
class TSimpleGraph
{
  ISimpleGraphDataSource *m_pDataSource;
public:
  TSimpleGraph(){ m_pDataSource = NULL; }
```

```
  virtual void Draw( const TRect &rect, TCanvas *pCanvas )
  {
    pCanvas->FillRect( rect );
    if(m_pDataSource == NULL) return; //Не задан источник
                                      //данных для графика
    if(m_pDataSource->GetValuesGount() == 0) return; //Нечего
                                                     //рисовать
    int x1 = rect.left, x2 = rect.right, width = rect.
Width();
    int y1 = rect.top, y2 = rect.bottom, height = rect.
Height();
    long v_count = m_pDataSource->GetValuesGount();
    float vx_min = m_pDataSource->GetMinX();
    float vx_max = m_pDataSource->GetMaxX();
    float vx_range = vx_max - vx_min;
    float vy_min = m_pDataSource->GetMinY();
    float vy_max = m_pDataSource->GetMaxY();
    float vy_range = vy_max - vy_min;
    //Строим график, соединяя опорные точки линиями
    int x = x1 + (m_pDataSource->GetX(0) - vx_min) / vx_range *
width;
    int y = y2 - (m_pDataSource->GetY(0) - vy_min) / vy_range *
height;
    pCanvas->MoveTo(x, y);
    for ( long i=1; i<v_count; i++ )
    {
      x = x1 + (m_pDataSource->GetX(i) - vx_min) / vx_range *
width;
      y = y2 - (m_pDataSource->GetY(i) - vy_min) / vy_range *
height;
      pCanvas->LineTo(x, y);
    }
  }
  virtual void SetDataSource( ISimpleGraphDataSource *pData-
Source )
  {
    m_pDataSource = pDataSource;
  }
};
```

Единственное, что позволяет делать приведенный в листинге 4.26 класс, это рисовать на канве (`TCanvas`) в заданном прямоугольнике график изменений пары значений, поступающих из внешнего источника. Все, что известно реализуемому графопостроителю о потенциальном источнике данных, — это реализуемый им интерфейс (см. текст объявления типа `ISimpleGraphDataSource` в начале листинга 4.26).

**ПРИМЕЧАНИЕ**

На всякий случай стоит уточнить, что interface в недрах подключаемых заголовочных файлов объявлен как #define interface struct. Само же ключевое слово «struct» в C++, по сути, является лишь сокращенной записью «class{public:», а так называемый «интерфейс» (interface) для программиста на C++ обычно представляет собой всего лишь абстрактный класс со всеми открытыми (public) виртуальными (абстрактными) методами.

Смысл методов интерфейса `ISimpleGraphDataSource` заключается в следующем.

- `GetMaxX`, `GetMinX`, `GetMaxY`, `GetMinY` — позволяют графопостроителю получить предварительную информацию о диапазоне значений по осям абсцисс (X) и ординат (Y). Это дает возможность правильно рассчитать масштабные коэффициенты для перевода значений X и Y в кординаты точек на канве.
- `GetValuesCount` — дает возможность узнать количество пар значений, которые будут отображены на графике.
- `GetX`, `GetY` — возвращают значения координат X и Y соответственно i-й точки графика.

При реализации интерфейса `ISimpleGraphDataSource` объект-источник данных предоставляет всю информацию, необходимую для правильного построения графика. По крайней мере в данном случае этой информации более чем достаточно.

На этом особенности реализации части программы, непосредственно отвечающей за вывод графика на экран, исчерпаны. Такие задачи, как масштабирование и прокрутка графика, будут рассмотрены чуть позже во время составления из написанных классов законченного приложения.

## Редактор звуковых данных

Второй, и он же основной, класс программы — это класс `TWaveManager`, в котором реализована вся логика работы со звуковыми данными: их загрузка, сохранение, воспроизведение, применение эффектов и работа непосредственно с сэмплами звуковых данных.

Код объявления класса `TWaveManager` приведен в листинге 4.27.

**Листинг 4.27.** Объявление класса, работающего со звуковыми данными

```
//Класс для работы со звуковыми данными WAVE-файла (моно)
class TWaveManager
{
  TWaveHeader m_Header;  //Заголовок звукового файла
  BYTE *m_Samples;       //Собственно звуковые данные
public:
  TWaveManager();
  ~TWaveManager();
  //Загрузка/сохранение файла
```

```
  bool Open( const char *filename );
  bool Save( const char *filename );
  //Воспроизведение
  bool Play();
  //Доступ к звуковым данным
  long GetSamplesCount();
  long GetSample( long index );
  void SetSample( long index, long sample );
  long GetMaxSampleValue();
  long GetMinSampleValue();
  //Звуковые эффекты
  void ChangeVolume( float delta_volume_percent );
  void Reverse();
  void IncreaseSpeed();
  void DecreaseSpeed();
  void Echo();
private:
  void Clear();
  void Assign( TWaveHeader &header, BYTE *pSamples );
  long GetSample(TWaveHeader &header, BYTE *pSamples, long
index );
  void SetSample(TWaveHeader &header,BYTE *pSamples,long
index,long sample);
};
```

Рассмотрение реализации класса `TWaveManager` начнется с описания способа хранения им звуковых данных, а хранятся эти данные следующим образом: отдельно заголовок (структура `TWaveHeader`, приведенная ранее в листинге 4.22) и отдельно массив байт, представляющих собой последовательность сэмплов. Код инициализации и очистки данных показан в листинге 4.28.

**Листинг 4.28.** Инициализация и очистка данных

```
TWaveManager::TWaveManager()
{
  ZeroMemory( &m_Header, sizeof(TWaveHeader) );
  m_Samples = NULL;
}
TWaveManager::~TWaveManager()
{
  Clear();
}
void TWaveManager::Clear()
{
  ZeroMemory( &m_Header, sizeof(TWaveHeader) );
  if ( m_Samples )
  {
```

```
    delete []m_Samples;
    m_Samples = NULL;
  }
}
```

Далее будет рассмотрена процедура загрузки данных из файла. Поскольку внутреннее представление данных, используемое классом `TWaveManager`, полностью соответствует структуре WAV-файла, то загрузка этих данных предельно упрощается (листинг 4.29).

**Листинг 4.29.** Загрузка данных из файла

```
bool TWaveManager::Open( const char *filename )
{
  int hFile = FileOpen( filename, fmOpenRead );
  if ( hFile == -1 ) return false;
  //Читаем заголовок файла
  TWaveHeader header = {0};
  int nBytesRead = FileRead( hFile, &header,
sizeof(TWaveHeader) );
  if ( nBytesRead != sizeof(TWaveHeader) )
  {
    //Не удалось прочитать заголовок файла
    FileClose(hFile);
    return false;
  }
  else if ( header.wChannels != 1 )
  {
    //Этот класс работает только с файлами, в которых записан
    //один звуковой канал и количество бит на сэмпл составляет
    //8 или 16
    FileClose(hFile);
    return false;
  }
  //Читаем звуковые данные
  BYTE *pSamples = new BYTE[header.sizeDATAch];
  FileSeek( hFile, nBytesRead, 0 );
  nBytesRead = FileRead( hFile, pSamples, header.sizeDATAch
);
  if ( nBytesRead != (int)header.sizeDATAch )
  {
    //Не удалось прочитать звуковые данные
    delete []pSamples;
    FileClose(hFile);
    return false;
  }
  FileClose(hFile);
```

```
  //Данные успешно загружены из файла
  Assign( header, pSamples );
  return true;
}
void TWaveManager::Assign( TWaveHeader &header, BYTE *pSam-
ples )
{
  Clear();
  CopyMemory( &m_Header, &header, sizeof(TWaveHeader) );
  m_Samples = pSamples;
}
```

Текст реализации операции, обратной загрузке, то есть операции сохранения данных в файл формата WAV, приведен в листинге 4.30.

**Листинг 4.30.** Сохранение данных в файл

```
bool TWaveManager::Save( const char *filename )
{
  //Записываем звуковые данные в файл
  bool bResult = false;
  long hFile = FileCreate(filename);
  if ( hFile != -1 )
  {
    int nSize = sizeof(TWaveHeader);
    bResult = FileWrite( hFile, &m_Header, nSize ) == nSize;
    bResult = bResult && FileSeek( hFile, nSize, 0 ) != -1;
    nSize = m_Header.sizeDATAch;
    bResult = bResult && FileWrite( hFile, m_Samples,
nSize ) == nSize;
  }
  FileClose(hFile);
  return bResult;
}
```

Процесс воспроизведения реализован с использованием API-функции `PlaySound`. В данном случае для простоты используется синхронный режим воспроизведения звука (листинг 4.31).

**Листинг 4.31.** Воспроизведение звука

```
bool TWaveManager::Play()
{
  if ( m_Header.sizeDATAch == 0 )
    return false;
  //Подготовка данных для воспроизведения
  BYTE *pBuffer = new BYTE[sizeof(TWaveHeader) + m_Header.
sizeDATAch];
  CopyMemory(pBuffer, &m_Header, sizeof(TWaveHeader));
```

```
  CopyMemory(pBuffer + sizeof(TWaveHeader), m_Samples, m_
Header.sizeDATAch);
  //Воспроизведение звуковых данных (синхронное)
  BOOL res = ::PlaySound((const char*)pBuffer, NULL, SND_
MEMORY | SND_SYNC);
  delete []pBuffer;
  return res;
}
```

На этом рассмотрение вспомогательных функций завершено. Остается самое интересное: непосредственная работа со звуковыми данными на самом низком уровне, то есть на уровне отдельных сэмплов.

Данный звуковой редактор поддерживает работу как с 8-, так и с 16-разрядными сэмплами. Информация о разрядности сэмплов всегда доступна из заголовка `TWaveHeader`. В листинге 4.32 приведен текст реализации функций `GetSample` и `SetSample`, составляющих ядро нашего звукового редактора.

**Листинг 4.32.** Функции чтения и записи отдельных сэмплов

```
long TWaveManager::GetSample( TWaveHeader &header, BYTE
*pSamples,
                              long index )
{
  //Определяет положение сэмпла в памяти и возвращает его
  //значение
  BYTE *pSampleStart = pSamples + header.wBlockAlign * index;
  long sample;
  switch ( header.wBitPerSample )
  {
  case 16:
    {
      short sh_sample = *(short*)pSampleStart;
      sample = sh_sample;
    }
    break;
  case 8:
    {
      char ch_sample = *(char*)pSampleStart;
      sample = ch_sample;
    }
    break;
  }
  return sample;
}
void TWaveManager::SetSample( TWaveHeader &header, BYTE
*pSamples,
                              long index, long sample )
```

```
{
  //Определяет положение сэмпла в памяти и записывает туда
  //новое значение
  BYTE *pSampleStart = pSamples + header.wBlockAlign * index;
  switch ( header.wBitPerSample )
  {
  case 16:
    *(short*)pSampleStart = static_cast<short>(sample);
    break;
  case 8:
    *(char*)pSampleStart = static_cast<char>(sample);
    break;
  }
}
```

Для реализации звуковых эффектов и доступа к сэмплам извне (в частности, при построении графика) в данном случае также пригодятся небольшие функции, описания которых приведены в листинге 4.33.

**Листинг 4.33.** Функции-оболочки для доступа к звуковым данным

```
//Возвращает полное число сэмплов звуковых данных
long TWaveManager::GetSamplesCount()
{
  if ( m_Header.sizeDATAch )
    return m_Header.sizeDATAch * 8 / m_Header.wBitPerSample;
  else
    return 0;
}
//Возвращает значение сэмпла с заданным номером
long TWaveManager::GetSample( long index )
{
  return GetSample( m_Header, m_Samples, index );
}
//Присваивает сэмплу с заданным номером новое значение
void TWaveManager::SetSample( long index, long sample )
{
  SetSample( m_Header, m_Samples, index, sample );
}
//Возвращает максимальное значение сэмпла
long TWaveManager::GetMaxSampleValue()
{
  return (1 << (m_Header.wBitPerSample - 1)) - 1;
}
//Возвращает минимальное значение сэмпла
long TWaveManager::GetMinSampleValue()
{
```

```
  return - (1 << (m_Header.wBitPerSample - 1));
}
```

Выше перечислены все функции, которые понадобятся далее. Теперь, наконец, можно переходить к реализации звуковых эффектов. Честно говоря, по сравнению со всем кодом, который пришлось написать для звукового редактора, эффекты оказалось реализовать чуть ли не проще всего. Так получилось потому, что сами алгоритмы, реализующие звуковые эффекты, крайне просты, к тому же благодаря вспомогательным функциям, описанным выше, работать с сэмплами звукового файла стало очень просто.

Так, описание функции `ChangeVolume`, позволяющей увеличивать и уменьшать громкость звука на определенное количество процентов, приведено в листинге 4.34.

**Листинг 4.34.** Изменение громкости

```
void TWaveManager::ChangeVolume( float delta_volume_percent )
{
  //Изменяет громкость на заданное количество процентов
  float koeff = (delta_volume_percent + 100) / 100.0 ;
  long count = GetSamplesCount();
  long max_sample = GetMaxSampleValue();
  long min_sample = GetMinSampleValue();
  long sample;
  for ( long i=0; i<count; i++ )
  {
    sample = GetSample(i) * koeff;
    if ( sample >= max_sample )
      SetSample( i, max_sample );
    else if ( sample <= min_sample )
      SetSample( i, min_sample );
    else
      SetSample( i, sample );
  }
}
```

Идея, положенная в алгоритм изменения громкости, очень проста. Для этого достаточно перевести переданное в функцию число, выраженное в процентах, в множитель, означающий степень изменения громкости звука. Затем остается лишь умножить на полученный множитель значение каждого сэмпла, не забывая при этом проверять, чтобы полученный результат не выходил за пределы допустимых для используемой разрядности данных значений.

Способ реализации следующего эффекта — инверсии, то есть записи звуковых данных в обратном порядке, — и того проще. В этом вы можете убедиться, изучив текст листинга 4.35.

**Листинг 4.35.** Инверсия данных

```
void TWaveManager::Reverse()
{
  //Изменяет порядок следования сэмплов на обратное
  //(звук будет воспроизводиться наоборот)
  long count = GetSamplesCount();
  DWORD sample1, sample2;
  long i, j;
  for ( i = 0, j = count-1; i<j; i++, j-- )
  {
    sample1 = GetSample(i);
    sample2 = GetSample(j);
    SetSample( i, sample2 );
    SetSample( j, sample1 );
  }
}
```

Для реализации инверсии, как ни странно, достаточно лишь выполнить запись сэмплов в обратном порядке.

Чуть более сложно выполняется ускорение звучания звукового клипа, описание способа реализации которого приведено в листинге 4.36.

**Листинг 4.36.** Увеличение скорости в два раза

```
void TWaveManager::IncreaseSpeed()
{
  //Увеличение скорости в 2 раза. При этом исходные звуковые
  //данные будут
  //укорочены в 2 раза (если число сэмплов нечетное, то
  //последний
  //сэмпл отбрасывается)
  TWaveHeader new_header(m_Header);
  new_header.sizeDATAch = m_Header.sizeDATAch / 2;
  new_header.sizeRIFFch = new_header.sizeDATAch +
sizeof(TWaveHeader) - 8;
  BYTE *pNewSamples = new BYTE[new_header.sizeDATAch];
  //Значения нового набора сэмплов будут вычисляться как
  //средние
  //арифметические четных и нечетных значений старого набора
  //сэмплов
  DWORD sample1, sample2, new_sample;
  long new_count = GetSamplesCount() / 2;
  for ( long i=0; i<new_count; i++ )
  {
    sample1 = GetSample(i*2);
    sample2 = GetSample(i*2+1);
```

```
    new_sample = sample1/2 + sample2/2;
    SetSample( new_header, pNewSamples, i, new_sample );
  }
  //Сохраняем новые звуковые данные
  Assign( new_header, pNewSamples );
}
```

Почему скорость изменяется именно в два раза? Только потому, что сделать это проще всего. Можно пойти разными путями, чтобы реализовать ускорение звучания именно в два раза, однако самый простой способ сделать это — просто «выкинуть» все четные или нечетные сэмплы. Тем не менее в данном примере применен другой подход (хочется верить, что немного лучший). Чтобы уменьшить число сэмплов в два раза, берутся нечетный и следующий четный сэмплы и находится их среднее арифметическое. Полученное значение и является тем сэмплом, который заменяет два исходных. Собственно, вот и вся идея.

Обратный эффект — замедление звучания звуковой дорожки — реализуется совсем иначе. Чтобы замедлить звучание, нужно, наоборот, добавить сэмплы (листинг 4.37).

**Листинг 4.37.** Уменьшение скорости в два раза

```
void TWaveManager::DecreaseSpeed()
{
  //Уменьшение скорости в два раза. При этом исходные
звуковые данные будут
  //увеличены в два раза (минус 1 сэмпл)
  TWaveHeader new_header(m_Header);
  new_header.sizeDATAch = m_Header.sizeDATAch * 2 - 1;
  new_header.sizeRIFFch = new_header.sizeDATAch +
sizeof(TWaveHeader) - 8;
  BYTE *pNewSamples = new BYTE[new_header.sizeDATAch];
  //Промежуточные значения нового набора сэмплов будут
  //вычисляться
  //как средние арифметические четных и нечетных значений
  //старого
  //набора сэмплов
  DWORD sample1, sample2, new_sample;
  long old_count = GetSamplesCount();
  for ( long i=0; i<old_count-1; i++ )
  {
    sample1 = GetSample(i);
    sample2 = GetSample(i+1);
    new_sample = sample1/2 + sample2/2;
    SetSample( new_header, pNewSamples, 2*i, sample1 );
    SetSample( new_header, pNewSamples, 2*i+1, new_sample );
  }
  //Сохраняем новые звуковые данные
```

```
  Assign( new_header, pNewSamples );
}
```

Для добавления новых сэмплов в звуковую дорожку рассчитывается среднее арифметическое каждых двух соседних сэмплов, и затем полученное значение вставляется между двумя исходными сэмплами в качестве необходимого промежуточного сэмпла. Так удастся увеличить время звучания клипа в два раза.

Наконец, последний эффект, поддерживаемый данным звуковым редактором, — эффект эха, описание способа реализации которого приведено в листинге 4.38.

**Листинг 4.38.** Эффект эха

```
void TWaveManager::Echo()
{
  //Эффект эха. Сэмпл повторяется через определенное время
  //с уменьшенной амплитудой
  float T_wait = 0.2;  //Время задержки эха (в секундах)
  long samples_wait = m_Header.dwSamplesPerSec * T_wait;
  //количество
  // сэмплов, через которое слышится эхо. На это число
  //возрастет размер
  // звуковых данных
  float echo_strength = 0.3; //Относительная громкость эха
  TWaveHeader new_header(m_Header);
  new_header.sizeDATAch = m_Header.sizeDATAch +

samples_wait * m_Header.wBitPerSample / 8;
  new_header.sizeRIFFch = new_header.sizeDATAch +
sizeof(TWaveHeader) - 8;
  BYTE *pNewSamples = new BYTE[new_header.sizeDATAch];
  long i;
  //До появления эха звуковые данные не меняются
  for ( i=0; i<samples_wait; i++ )
  {
    SetSample( new_header, pNewSamples, i, GetSample(i) );
  }
  //Значения нового набора сэмплов после samples_wait будут
  //вычисляться
  //как сумма значений оригинального сэмпла и эха ранее
  //следовавшего сэмпла
  long old_sample, echo_sample, sample;
  long count = GetSamplesCount();
  long max_sample = GetMaxSampleValue();
  long min_sample = GetMinSampleValue();
  for ( i=0; i<count; i++ )
  {
```

```
    if ( i + samples_wait < count  )
      old_sample = GetSample(i + samples_wait);
    else
      old_sample = 0;
    echo_sample = GetSample(i) * echo_strength;
    sample = old_sample + echo_sample;
    if ( sample >= max_sample )
      sample = max_sample;
    else if ( sample <= min_sample )
      sample = min_sample;
    SetSample( new_header, pNewSamples, i + samples_wait,
sample );
  }
  //Сохраняем новые звуковые данные
  Assign( new_header, pNewSamples );
}
```

Идея реализации этого эффекта также достаточно проста. В данном случае эхо реализуется как однократное повторение каждого сэмпла с задержкой 0,2 с. Громкость эха составляет 30 % громкости исходного звука. Эхо смешивается с исходным звуком с помощью обычного сложения значения интенсивности исходного сэмпла и рассчитанного значения интенсивности эха в данный момент времени.

## Сборка элементов редактора

Теперь пришло время собрать части звукового редактора вместе. Для начала необходимо разместить на форме (`TForm1`) компоненты так, как показано на рис. 4.12.

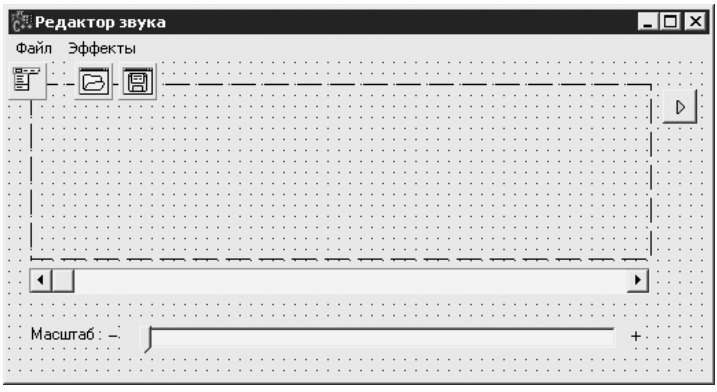


**Рис. 4.12.** Форма редактора звука на этапе разработки

Если говорить подробно, то на форму нужно поместить:

- меню;
- компонент OpenDialog, предназначенный для реализации возможности открытия файла, с идентификатором `OpenDialog1`;

- компонент SaveDialog, предназначенный для реализации возможности сохранения файла, с идентификатором SaveDialog1;
- компонент PaintBox с идентификатором graph_image; клиентская область этого компонента будет служить для вывода графика звукового сигнала;
- кнопку SpeedButton с идентификатором cmbPlay, путем нажатия которой будет запускаться воспроизведение редактируемого звука;
- полосу прокрутки ScrollBar с идентификатором scrlPosition; она будет использоваться для прокрутки графика сигнала;
- регулятор TrackBar с идентификатором tbScale и подписью (компонент Label) к нему.

Далее нужно включить в класс формы TForm1 экземпляры объектов TSimpleGraph и TWaveManager. Кроме того, форма реализует интерфейс ISimpleGraphDataSource (листинг 4.39).

**Листинг 4.39.** Фрагменты объявления класса формы TForm1

```
class TForm1 : public TForm
             , public ISimpleGraphDataSource
{
...
private:  // Пользовательское объявление
  TSimpleGraph m_Graph;
  TWaveManager m_WaveManager;
...
// ISimpleGraphDataSource
  virtual float GetMaxX();
  virtual float GetMinX();
  virtual float GetMaxY();
  virtual float GetMinY();
  virtual float GetX(long index);
  virtual float GetY(long index);
  virtual long GetValuesGount();
...
private:
  long SamplesPerScreen();
  void RefreshScale();};
```

Код, отвечающий за правильную работу построителя графика: инициализацию объекта графика, его перерисовку и реализацию интерфейса ISimpleGraphDataSource, — приведен в листинге 4.40.

**Листинг 4.40.** Реализация построителя графика

```
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
```

```
  m_Graph.SetDataSource(this);
}
void __fastcall TForm1::graph_imagePaint(TObject *Sender)
{
  //Рисование графика звукового сигнала
  graph_image->Canvas->Pen->Color = clBlack;
  graph_image->Canvas->Brush->Color = clWhite;
  m_Graph.Draw( graph_image->ClientRect, graph_image->Canvas
);
}
// Реализация ISimpleGraphDataSource
float TForm1::GetMaxX()
{
  return scrlPosition->Position + SamplesPerScreen();
}
float TForm1::GetMinX()
{
  return scrlPosition->Position;
}
float TForm1::GetMaxY()
{
  return m_WaveManager.GetMaxSampleValue();
}
float TForm1::GetMinY()
{
  return m_WaveManager.GetMinSampleValue();
}
float TForm1::GetX(long index)
{
  return scrlPosition->Position + index;
}
float TForm1::GetY(long index)
{
  return m_WaveManager.GetSample(scrlPosition->Position +
index);
}
long TForm1::GetValuesGount()
{
  if ( SamplesPerScreen() > m_WaveManager.GetSamplesCount() )
    return m_WaveManager.GetSamplesCount();
  else
    return SamplesPerScreen();
}
```

Также на форму полностью возложено управление прокруткой с учетом масштаба графика (листинг 4.41).

**Листинг 4.41.** Масштабирование и прокрутка графика

```
void __fastcall TForm1::scrlPositionScroll(TObject *Sender,
      TScrollCode ScrollCode, int &ScrollPos)
{
  graph_image->Refresh();
}
void __fastcall TForm1::tbScaleChange(TObject *Sender)
{
  //Изменился масштаб графика
  scrlPosition->Max = m_WaveManager.GetSamplesCount() -
SamplesPerScreen();
  graph_image->Refresh();
}
long TForm1::SamplesPerScreen()
{
  //Количество сэмплов, которое умещается на экране, зависит
  //от масштаба
  return -tbScale->Position;
}
void TForm1::RefreshScale()
{
  //Приспосабливаем полосу прокрутки графика к новой
  //длительности клипа
  tbScale->Min = -m_WaveManager.GetSamplesCount();
  tbScale->Max = -20;
  scrlPosition->Max = m_WaveManager.GetSamplesCount() -
SamplesPerScreen();
}
```

Масштаб графика определяется количеством отображаемых на графике сэмплов. Он может изменяться от 20 (максимальное увеличение) до полного количества сэмплов в звуковых данных (на графике отображаются все сэмплы файла). При этом для компонента `tbScale`, управляющего масштабом, используется отрицательная шкала, чтобы при перемещении ползунка вправо получить действительно увеличение масштаба, а не наоборот.

Диапазон значений полосы прокрутки `scrlPosition`, в свою очередь, зависит от количества сэмплов и масштаба графика (см. последнюю строку объявления метода `RefreshScale` в листинге 4.41).

Ниже в листинге 4.42 показан алгоритм загрузки и сохранения звуковых данных. При реализации открытия файла пришлось немного поработать над правильным исходным оформлением приложения.

**Листинг 4.42.** Открытие и сохранение файла

```
void __fastcall TForm1::mnuOpenClick(TObject *Sender)
{
```

```
  //Открытие файла
  if ( OpenDialog1->Execute() )
  {
    if ( m_WaveManager.Open(OpenDialog1->FileName.c_str()) )
    {
      RefreshScale();
      tbScale->Enabled = true;
      tbScale->Position = tbScale->Min;
      scrlPosition->Enabled = true;
      scrlPosition->Position = 0;
      graph_image->Refresh();
    }
    else
    {
      tbScale->Enabled = false;
      tbScale->Position = tbScale->Min;
      scrlPosition->Enabled = false;
      scrlPosition->Position = 0;
      Application->MessageBox( "Не удалось открыть указанный
файл",
                               "Ошибка", MB_ICONEXCLAMATION
);
    }
  }
}
void __fastcall TForm1::mnuSaveClick(TObject *Sender)
{
  //Сохранение файла
  if ( SaveDialog1->Execute() )
  {
    if ( !m_WaveManager.Save(SaveDialog1->FileName.c_str()) )
    {
      Application->MessageBox( "Не удалось записать файл",
                               "Ошибка", MB_ICONEXCLAMATION
);
    }
  }
}
```

Остальной код формы представляет собой не что иное, как простые вызовы методов объекта `m_WaveManager`, осуществляемые в ответ на действия пользователя (листинг 4.43).

**Листинг 4.43.** Команды редактора

```
void __fastcall TForm1::cmbPlayClick(TObject *Sender)
{
```

```
  //Воспроизведение
  Screen->Cursor = crHourGlass;
  m_WaveManager.Play();
  Screen->Cursor = crDefault;
}
void __fastcall TForm1::mnuIncVolumeClick(TObject *Sender)
{
  //Увеличение громкости на 30 %
  m_WaveManager.ChangeVolume(+30);
  RefreshScale();
  graph_image->Refresh();
}
void __fastcall TForm1::mnuDecVolumeClick(TObject *Sender)
{
  //Уменьшение громкости на 30 %
  m_WaveManager.ChangeVolume(-30);
  RefreshScale();
  graph_image->Refresh();
}
void __fastcall TForm1::mnuReverseClick(TObject *Sender)
{
  //Инверсия
  m_WaveManager.Reverse();
  RefreshScale();
  graph_image->Refresh();
}
void __fastcall TForm1::mnuFasterClick(TObject *Sender)
{
  //Увеличение скорости
  m_WaveManager.IncreaseSpeed();
  RefreshScale();
  graph_image->Refresh();
}
void __fastcall TForm1::mnuSlowerClick(TObject *Sender)
{
  //Уменьшение скорости
  m_WaveManager.DecreaseSpeed();
  RefreshScale();
  graph_image->Refresh();
}
void __fastcall TForm1::mnuEchoClick(TObject *Sender)
{
  //Добавление эха
  m_WaveManager.Echo();
  RefreshScale();
```

```
  Refresh();
}
```

Наконец, редактор звука полностью готов, и теперь можно опробовать на практике все его возможности. Конечно, до профессиональной программы по обработке звука ему еще очень далеко, но ведь это лишь учебный пример, который был создан всего за несколько часов. Однако хочется надеяться, что вы сможете даже из этого примера получить что-нибудь полезное или что вам по крайней мере было интересно создавать звуковой редактор своими руками.

# Глава 5

## Мышь и клавиатура

- Мышь
- Клавиатура

Самыми распространенными средствами ввода информации в компьютер являются манипулятор мышь и клавиатура. Практически невозможно представить себе персональный компьютер без этих устройств, поскольку клавиатура обеспечивает полноценный ввод текстовой информации, а мышь является наиболее простым, интуитивно понятным средством работы с графическим интерфейсом. По этой причине существует масса возможностей по созданию различного рода трюков, связанных с мышью и клавиатурой.

# Мышь

Начало главы посвящено описанию простых операций с мышью. Несмотря на простоту информации, получаемой от этого средства ввода данных, в которой содержатся лишь координаты указателя и регистрируется факт нажатия и отпускания одной из кнопок, написать программу, выполняющую сложные действия в ответ на команды от мыши, не так просто, в чем вы уже могли убедиться при изучении примеров из гл. 2.

## Проверка наличия мыши

Для начала вполне логичным было бы программным путем определить факт наличия мыши в системе. Описание одного из способов определения наличия мыши приведено в листинге 5.1.

**Листинг 5.1.** Определение факта наличия мыши

```
bool MousePresent()
{
  // С помощью вызова функции GetSystemMetrics определяем
  // наличие мыши в системе
  return ::GetSystemMetrics(SM_MOUSEPRESENT) != 0;
}
```

Здесь используется API-функция `GetSystemMetrics`, имеющая в общем случае следующий вид:

```
int GetSystemMetrics(int nIndex);
```

Единственным параметром этой функции задается идентификатор «системной метрики», который требуется получить. Одной из таких «метрик» как раз и является наличие в системе мыши.

## Координаты указателя мыши

Помимо функции слежения за положением указателя мыши, в обработчике событий `MouseMove` существуют еще несложные и более универсальные способы определения текущего положения указателя. Один из таких способов продемонстрирован в листинге 5.2.

**Листинг 5.2.** Определение координат указателя мыши

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
  //Определяем положение указателя мыши
  TPoint pt;
  ::GetCursorPos(&pt);
  txtX->Text = IntToStr(pt.x);
  txtY->Text = IntToStr(pt.y);
}
```

Для определения координат мыши в листинге 5.2 используется API-функция `GetCursorPos`. Передав в эту функцию переменную `pt` типа `TPoint`, можно получить текущие экранные координаты курсора.

Кроме того, мышь можно передвигать программным путем. Следующий пример демонстрирует, как можно нажатием кнопки на форме передвинуть указатель по диагонали в левый верхний угол экрана (листинг 5.3).

**Листинг 5.3.** Перемещение указателя мыши

```
void __fastcall TForm1::cmbMoveMouseClick(TObject *Sender)
{
  //Передвигаем указатель из текущего положения
  //в левый верхний угол экрана
  TPoint pt;
  ::GetCursorPos(&pt);
  double alpha = atan((double)pt.y/(double)pt.x);
  int len = sqrt(pt.x*pt.x + pt.y*pt.y);
  for ( int i=0; i<len; i++ )
  {
    ::SetCursorPos(pt.x - i*cos(alpha), pt.y - i*sin(alpha));
    ::Sleep(1);
  }
}
```

## Захват указателя мыши

Существуют несколько задач, для выполнения которых необходимо иметь возможность получать сообщения от мыши даже тогда, когда указатель находится за пределами формы. За примером далеко ходить не надо: запустите редактор Paint (или приведенный в гл. 2 графический редактор), 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 Paint. Есть ли на рисунке часть линии, которую вы рисовали, двигая курсор за пределами окна (должна быть)?

Захват указателя полезен и в других случаях, поэтому здесь будет рассмотрен способ его реализации (а сделать это действительно просто). В листинге 5.4 приведены

описания обработчиков событий нажатия и отпускания кнопки мыши, которые реализуют захват указателя во время удерживания кнопки мыши в нажатом состоянии.

**Листинг 5.4.** Захват и освобождение указателя мыши

```
void __fastcall TForm1::FormMouseDown(TObject *Sender,
TMouseButton Button,

TShiftState Shift, int X, int Y)
{
  //Захват указателя мыши
  ::SetCapture(Handle);
}
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouse-
Button Button,
                                    TShiftState Shift, int X,
int Y)
{
  //Освобождение указателя
  ::ReleaseCapture();
}
```

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

**ПРИМЕЧАНИЕ**

При использовании функции SetCapture окно будет получать сообщения от указателя мыши, находящегося не над окном, только в случаях, если клавиша мыши нажата либо если курсор находится над одним из окон, созданных тем же потоком (независимо от факта нажатия кнопки мыши).

Можно также упомянуть о API-функции `GetCapture`. Данная функция не принимает аргументов и возвращает дескриптор окна, захватившего ранее указатель. С помощью этой функции можно, например, удостовериться, что захватом указателя мыши не нарушается работа других приложений (что, правда, маловероятно).

## Ограничение перемещения указателя

С помощью несложных манипуляций можно легко ограничить перемещение указателя мыши определенной областью экрана (прямоугольником) — для этого используется API-функция `ClipCursor`. Эта функция принимает в качестве параметра структуру `TRect`, содержащую координаты прямоугольника, в пределах

которого может перемещаться указатель, и возвращает отличное от нуля значение в случае успешной установки ограничения.

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

Способ использования функций `ClipCursor` и `GetClipCursor` продемонстрирован в листинге 5.5.

**Листинг 5.5.** Ограничение перемещения указателя

```
void __fastcall TForm1::FormMouseDown(TObject *Sender,
TMouseButton Button,

TShiftState Shift, int X, int Y)
{
  if ( !m_bCursorClipped )
  {
    //Сохраняем ранее заданную область перемещения указателя
    //мыши
    ::GetClipCursor(&m_LastClipRect);
    //Ограничиваем область перемещения указателя клиентской
    //областью формы
    TRect rc = GetClientRect();
    rc.left += ClientOrigin.x;
    rc.right += ClientOrigin.x;
    rc.top += ClientOrigin.y;
    rc.bottom += ClientOrigin.y;
    m_bCursorClipped = ::ClipCursor(&rc);
  }
}
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouse-
Button Button,
                                    TShiftState Shift, int X,
int Y)
{
  if ( m_bCursorClipped )
  {
    //Восстанавливаем область перемещения указателя мыши
    ::ClipCursor(&m_LastClipRect);
    m_bCursorClipped = false;
  }
}
```

В этом листинге реализована пара обработчиков событий нажатия и отпускания кнопки мыши. При этом во время удерживания левой кнопки мыши в нажатом состоянии перемещение указателя ограничивается клиентской областью формы.

Перед применением ограничения перемещения указателя происходит сохранение его прежней области перемещения, чтобы затем можно было в точности восстановить область перемещения указателя, которая была задана до запуска данного примера. Для отмены ограничения перемещения указателя служит обработчик отпускания кнопки мыши, передающий в API-функцию `ClipCursor` координаты области перемещения указателя, действовавшие изначально (и сохраненные в переменной `m_LastClipRect`).

**ПРИМЕЧАНИЕ**

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

## Инверсия функций кнопок мыши

Как известно, такие распространенные операционные системы, как Windows, предоставляют возможность работать с компьютером широкому кругу пользователей. Со стороны разработчиков было бы глупо не предусмотреть возможность простой адаптации мыши для ее использования правшой и левшой. К тому же адаптировать мышь к таким различиям намного проще, нежели выпускать манипуляторы различных типов: конструкцию-то изменять не надо — достаточно просто программно поменять функции кнопок мыши.

Способ изменения функций левой и правой кнопок мыши в ответ на нажатия кнопок на форме демонстрирует листинг 5.6.

**Листинг 5.6.** Изменение назначений кнопок мыши

```
void __fastcall TForm1::cmbSwapClick(TObject *Sender)
{
  //Меняем кнопки мыши местами
  ::SwapMouseButton(true);
  cmbRestore->Enabled = true;
  cmbSwap->Enabled = false;
}
void __fastcall TForm1::cmbRestoreClick(TObject *Sender)
{
  //Восстанавливаем назначения правой и левой кнопок
  ::SwapMouseButton(false);
  cmbRestore->Enabled = false;
  cmbSwap->Enabled = true;
}
```

В листинге 5.6 не учитывается тот факт, что инверсия кнопок мыши может быть изначально установлена при запуске операционной системы или приложения (например, если за компьютером работает левша). Чтобы точно знать, была ли ранее применена инверсия к кнопкам мыши, можно использовать возвращаемое

функцией SwapMouseButton значение — если это значение отлично от нуля, то, значит, ранее функции кнопок мыши были инвертированы.

## Вычисление расстояния, пройденного указателем мыши

Обратите внимание на небольшую программу, которая носит скорее познавательный, чем практический характер. Эта программа позволяет вычислять расстояние в метрах (в буквальном смысле), которое преодолевает указатель мыши за время работы программы. Внешний вид формы данного приложения показан на рис. 5.1.

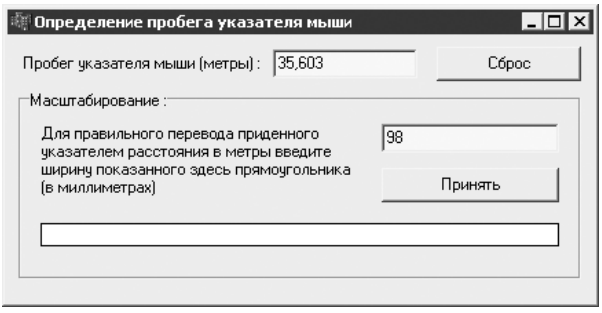


**Рис. 5.1.** Программа измерения пробега указателя мыши

Использование этой программы не представляет никаких сложностей: сразу после запуска она начинает измерять расстояние, пройденное указателем мыши, в пикселах. Нижняя группа элементов управления нужна для правильного преобразования пройденного расстояния в метры. Чтобы программа правильно преобразовывала пройденное расстояние в метры, нужно линейкой измерить ширину белого прямоугольника, ввести полученное значение (в миллиметрах) в текстовое поле и нажать кнопку Принять.

Теперь можно переходить к рассмотрению реализации этого приложения. В табл. 5.1 приведены сведения о настройке элементов управления, не являющихся рамками или статическими надписями.

**Таблица 5.1.** Параметры элементов управления формы, показанной на рис. 5.1

| Название (свойство name) | Описание | Измененные свойства и их значения |
|---|---|---|
| Timer1 | При каждом срабатывании этого таймера происходит определение положения указателя мыши | Interval = 1 |
| txtDistance | Это текстовое поле используется для отображения пройденного указателем расстояния | ReadOnly = true |

*Продолжение ⇩*

**Таблица 5.1** *(продолжение)*

| Название (свойство name) | Описание | Измененные свойства и их значения |
|---|---|---|
| cmbReset | При нажатии этой кнопки сбрасываются показания счетчика пройденного расстояния | Caption = "Сброс" |
| Shape1 | Ширина этого прямоугольника измеряется для определения масштаба | Ширина и высота произвольны, но чем больше ширина, тем точнее масштабирование |
| txtWidth | Текстовое поле, предназначенное для указания ширины прямоугольника Shape1 | Text = "98" (значение для ЭЛТ монитора 17' с разрешением 1024 × 768) |
| cmbApplyScale | При нажатии этой кнопки устанавливается новый коэффициент перевода расстояния в метры | Caption = "Применить" |

В листинге 5.7 также приведены тексты объявлений переменных-членов класса `TForm1` и методов, добавленных вручную.

**Листинг 5.7.** Форма для измерения пробега указателя

```
class TForm1 : public TForm
{
  //...
private:  // Объявления пользователя
  TPoint m_LastPos;  //Предыдущее положение указателя мыши
  double m_Distance; //Пройденное расстояние (в пикселах)
  double m_Scale;    //Коэффициент перевода расстояния
в метры
  void ShowDistance();
  //...
};
```

Суммарное пройденное указателем расстояние в пикселах сохраняется в переменной `m_Distance`. Перевод этого расстояния в метры осуществляется следующим образом (листинг 5.8).

**Листинг 5.8.** Перевод расстояния в метры с учетом масштаба

```
void TForm1::ShowDistance()
{
  //Пересчитываем текущий пробег в метры и показываем его
  //в текстовом поле
  //..подсчитываем расстояние с учетом масштаба
  double distanceMetters = m_Scale * m_Distance;
  //..округляем до трех знаков (мм) и показываем
  distanceMetters = (int)(distanceMetters * 1000) * 0.001;
  txtDistance->Text = FloatToStr(distanceMetters);
}
```

В приведенном в листинге 5.8 расчете нет ничего сложного, как, собственно, нет ничего сложного и во всем примере.

Главная процедура приложения — обработчик для таймера `Timer1`. Таймер работает с максимальной для него частотой (не раз в 1 мс, конечно, но около 18 раз в секунду). Текст обработчика `Timer1Timer` приведен в листинге 5.9.

**Листинг 5.9.** Подсчет разницы между положениями указателя мыши

```
#define sqr(a) ((a)*(a))
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
  TPoint cur_pos;
  ::GetCursorPos(&cur_pos);
  if ( cur_pos.x != m_LastPos.x || cur_pos.y != m_LastPos.y )
  {
    //Вычисляем разницу между текущим и предыдущим положениями
    //указателя
    double delta = sqrt(sqr(cur_pos.x - m_LastPos.x) +
                        sqr(cur_pos.y - m_LastPos.y));
    m_Distance += delta;
    //Не забываем сохранить новые координаты указателя
    m_LastPos = cur_pos;
    //Обновим показания в текстовом поле
    ShowDistance();
  }
}
```

Алгоритм принятия нового коэффициента пересчета расстояния в метры, происходящего после нажатия кнопки Принять, приведен в листинге 5.10.

**Листинг 5.10.** Принятие нового коэффициента перевода расстояния в метры

```
void __fastcall TForm1::cmbApplyScaleClick(TObject *Sender)
{
  //Принятие новой шкалы для перевода расстояния в метры
  double newScale = 0.001 * StrToInt(txtWidth->Text) /
Shape1->Width;
  if ( newScale <= 0 )
  {
    Application->MessageBox("Введите значение больше нуля",
"Ошибка",
                            MB_ICONINFORMATION);
    txtWidth->Text = IntToStr((int)(m_Scale * Shape1->Width *
1000));
  }
  else
  {
    m_Scale = newScale;
```

```
    ShowDistance();
  }
}
```

В завершение осталось реализовать только код инициализации при запуске программы и обработчик нажатия кнопки `cmbReset` (листинг 5.11).

**Листинг 5.11.** Инициализация при запуске и код сброса счетчика

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  m_Scale = 0.001 * 96 / Shape1->Width;
  m_Distance = 0;
  ::GetCursorPos(&m_LastPos);
}
void __fastcall TForm1::cmbResetClick(TObject *Sender)
{
  //Сброс показаний счетчика
  m_Distance = 0;
  ShowDistance();
}
```

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

## Подсвечивание элементов управления

В завершение будет рассмотрен несложный, но достаточно полезный пример программы, позволяющей сделать интерфейс приложения более «живым»: изменять внешний вид элементов управления при наведении на них указателя мыши.

В листинге 5.12 продемонстрирован способ выполнения статической надписи в виде гиперссылки (для большего эффекта такой надписи можно установить значение свойства `Cursor` равным `crHandPoint` на этапе проектирования формы).

**Листинг 5.12.** Подчеркивание и изменение цвета надписи

```
void __fastcall TForm1::Label2MouseMove(TObject *Sender,
TShiftState Shift,
                                        int X, int Y)
{
  Label2->Font->Style = TFontStyles() << fsUnderline;
  Label2->Font->Color = clBlue;
}
```

Осталось добавить обработчик события `Click` для надписи — и получится вполне правдоподобная гиперссылка, правда, выполнять она сможет любое действие, а не только открытие определенной веб-страницы.

Кроме того, вы можете изменить такой атрибут шрифта, как начертание, например, для стандартной кнопки. Как это можно сделать, продемонстрировано в листинге 5.13.

**Листинг 5.13.** Изменение начертания шрифта

```
void __fastcall TForm1::Button2MouseMove(TObject *Sender,

TShiftState Shift, int X, int Y)
{
  Button2->Font->Style = TFontStyles() << fsItalic << fsBold;
}
```

В листингах 5.12 и 5.13 используется обработчик события `MouseMove` вместо `MouseEnter` по различным причинам. Во-первых, для стандартных кнопок (вкладка Standard палитры компонентов) не предусмотрено событие `MouseEnter`. Во-вторых, как бы смешно это ни звучало, так надежнее, и, наконец, одним из простейших путей отмены «подсветки» элементов управления является написание обработчика события `MouseMove` для формы (листинг 5.14).

**Листинг 5.14.** Отключение «подсветки» элементов управления

```
void __fastcall TForm1::FormMouseMove(TObject *Sender,
TShiftState Shift,
      int X, int Y)
{
  //Отмена «подсветки» элементов управления
  Button2->Font->Style = TFontStyles();
  Label2->Font->Style = TFontStyles();
  Label2->Font->Color = clWindowText;
}
```

# Клавиатура

Клавиатура традиционно является основным средством ввода информации в компьютер, поэтому несправедливо было бы обойти стороной и не рассмотреть некоторые не так часто используемые или не такие очевидные возможности работы с клавиатурой.

## Определение информации о клавиатуре

Начать стоит с небольшого примера программы, позволяющей получать некоторую информацию о клавиатуре (листинг 5.15).

**Листинг 5.15.** Определение информации о клавиатуре

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  //Определяем тип клавиатуры
  switch ( ::GetKeyboardType(0) )
  {
  case 1: txtType->Text = "PC/XT или совместимая (83
клавиши)"; break;
  case 2: txtType->Text = "Olivetti \"ICO\" (102 клавиши)";
break;
  case 3: txtType->Text = "PC/AT (84 клавиши) или
аналогичная"; break;
  case 4: txtType->Text = "Расширенная (101 или 102 клавиши)";
break;
  case 5: txtType->Text = "Nokia 1050 или похожая"; break;
  case 6: txtType->Text = "Nokia 9140 или похожая"; break;
  case 7: txtType->Text = "японская"; break;
  }
  //Определяем код типа производителя
  txtSubtype->Text = IntToStr(::GetKeyboardType(1));
  //Определяем количество функциональных клавиш
  txtFnKeys->Text = IntToStr(::GetKeyboardType(2));
}
```

Реализация примера основана на использовании API-функции `GetKeyboardType`, возвращающей значение параметра клавиатуры с заданным целочисленным идентификатором. В данном случае:

- `0` — идентификатор типа клавиатуры;
- `1` — идентификатор подтипа клавиатуры (OEM-код производителя);
- `2` — идентификатор количества функциональных клавиш (F1, F2, F3 и т. д.).

При создании формы происходит заполнение текстовых полей информацией о типе клавиатуры, коде типа, присвоенном производителем, и количестве функциональных клавиш. Пример результата определения информации о клавиатуре приведен на рис. 5.2.


**Рис. 5.2.** Информация о клавиатуре

## Опрос клавиатуры

Существует достаточно удобная альтернатива обработке событий клавиатурного ввода, которая может оказаться особенно полезной, если необходимо знать состояние сразу нескольких клавиш (например, при самостоятельном написании простейших компьютерных игр).

В листинге 5.16 приведен пример кода обработчика события `Timer1Timer`, определяющего состояния клавиш ↑, ↓, ←, →, а также Пробел, Enter, Ctrl (правый), Shift (правый) и Alt (правый).

**Листинг 5.16.** Определение состояний некоторых клавиш

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
  //Получаем состояния клавиш
  TKeyboardState buttons;
  ::GetKeyboardState(buttons);
  //Отобразим состояния клавиш
  //..пробел
  if ( buttons[VK_SPACE] & 128 )
    ::SendMessage(cmbSpace->Handle, BM_SETSTATE, BST_CHECKED,
0);
  else
    ::SendMessage(cmbSpace->Handle, BM_SETSTATE, BST_UNCHECKED,
0);
  //..enter
  if ( buttons[VK_RETURN] & 128 )
    ::SendMessage(cmbEnter->Handle, BM_SETSTATE, BST_CHECKED,
0);
  else
    ::SendMessage(cmbEnter->Handle, BM_SETSTATE, BST_UNCHECKED,
0);
  //..правый Ctrl
  if ( buttons[VK_RCONTROL] & 128 )
    ::SendMessage(cmbRCtrl->Handle, BM_SETSTATE, BST_CHECKED,
0);
  else
    ::SendMessage(cmbRCtrl->Handle, BM_SETSTATE, BST_UNCHECKED,
0);
  //..правый Alt
  if ( buttons[VK_RMENU] & 128 )
    ::SendMessage(cmbRAlt->Handle, BM_SETSTATE, BST_CHECKED,
0);
  else
   ::SendMessage(cmbRAlt->Handle, BM_SETSTATE, BST_UNCHECKED,
0);
```

```
  //..правый Shift
  if ( buttons[VK_RSHIFT] & 128 )
    ::SendMessage(cmbRShift->Handle, BM_SETSTATE, BST_CHECKED,
0);
  else
    ::SendMessage(cmbRShift->Handle, BM_SETSTATE, BST_UN-
CHECKED, 0);
  //..вверх
  if ( buttons[VK_UP] & 128 )
    ::SendMessage(cmbUp->Handle, BM_SETSTATE, BST_CHECKED,
0);
  else
    ::SendMessage(cmbUp->Handle, BM_SETSTATE, BST_UNCHECKED,
0);
  //..вниз
  if ( buttons[VK_DOWN] & 128 )
    ::SendMessage(cmbDown->Handle, BM_SETSTATE, BST_CHECKED,
0);
  else
    ::SendMessage(cmbDown->Handle, BM_SETSTATE, BST_UNCHECKED,
0);
  //..влево
  if ( buttons[VK_LEFT] & 128 )
    ::SendMessage(cmbLeft->Handle, BM_SETSTATE, BST_CHECKED,
0);
  else
    ::SendMessage(cmbLeft->Handle, BM_SETSTATE, BST_UNCHECKED,
0);
  //..вправо
  if ( buttons[VK_RIGHT] & 128 )
    ::SendMessage(cmbRight->Handle, BM_SETSTATE, BST_CHECKED,
0);
  else
    ::SendMessage(cmbRight->Handle, BM_SETSTATE, BST_UNCHECKED,
0);
}
```

Для определения состояния клавиш используется API-функция `GetKeyboard-State`, заполняющая массив `buttons` значениями, характеризующими текущее состояние кнопки. Здесь употреблено слово «массив», поскольку тип `TKeyboardState` в заголовочных файлах определен следующим образом:

```
typedef Byte TKeyboardState[256];
```

При этом значения в массиве `buttons` трактуются так:

- если установлен старший бит (проверка чего и осуществляется в листинге 5.16), то считается, что клавиша в данный момент нажата;

- если установлен младший бит, то считается, что 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).

В данном примере каждой контролируемой клавише (см. листинг 5.16) соответствует кнопка на форме. Для принудительной установки кнопки в нажатое или ненажатое состояние посылается сообщение BM_SETSTATE. Пример состояния клавиш в некоторый момент времени показан на рис. 5.3.

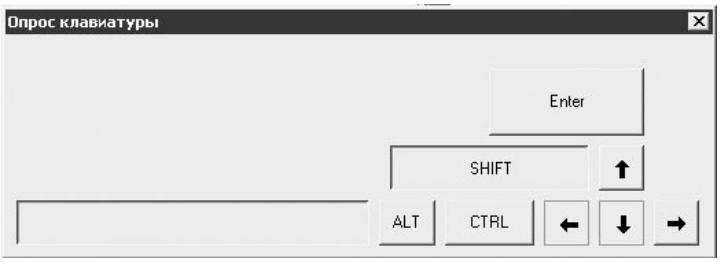


**Рис. 5.3.** Состояние некоторых клавиш клавиатуры

Интересно, что рассмотренный выше способ работы с клавиатурой можно использовать даже для выявления неисправных клавиш, а также для определения максимального количества одновременно нажатых клавиш, которое допускает ваша клавиатура.

## Имитация нажатия клавиш

Оказывается, состояние клавиш на клавиатуре можно не только определять, но и программно изменять. Для этого даже не обязательно писать сложный код для взаимодействия с драйвером клавиатуры. Ниже будет рассмотрен один из способов реализации нажатия клавиш программным путем, который крайне прост благодаря наличию специально предусмотренной для имитации клавиатурного ввода API-функции keybd_event.

Назначения параметров этой функции стоит пояснить на примере (листинг 5.17).

**Листинг 5.17.** Вызов меню Пуск

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  //Имитируем нажатие клавиши Windows
  ::keybd_event(VK_LWIN, 0, 0, 0);
  ::keybd_event(VK_LWIN, 0, KEYEVENTF_KEYUP, 0);
}
```

В данном случае интересны прежде всего первый и третий параметры функции keybd_event (второй не используется, а четвертый предназначен для указания

дополнительной информации, имеющей отношение к нажатию клавиши). Так, с помощью первого параметра функции передается код «нажимаемой» или «отпускаемой» клавиши, а третий параметр равен нулю при «нажатии» и константе KEYEVENTF_KEYUP при «отпускании» клавиши.



**ВНИМАНИЕ**

Главное при использовании функции keybd_event — не забывать «отпускать» программно нажатые клавиши (как это делается в приведенных в данном подразделе примерах). Иначе есть риск возникновения значительного числа ошибок клавиатурного ввода.

Пример реализации программного нажатия клавиши Prt Scrn (создания копии изображения экрана), аналогичной приведенной в листинге 5.17, показан в листинге 5.18.

**Листинг 5.18.** Создание копии изображения экрана

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  //Нажимаем Prt Scrn
  ::keybd_event(VK_SNAPSHOT, 0, 0, 0);
  ::keybd_event(VK_SNAPSHOT, 0, KEYEVENTF_KEYUP, 0);
}
```

Кроме того, в листинге 5.19 приведен пример реализации программного нажатия сочетания клавиш Windows+M, используемого для сворачивания всех окон.

**Листинг 5.19.** Сворачивание всех окон

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  //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);
}
```

Если добавить к этому сочетанию клавишу Shift, можно получить код, выполнение которого приводит к восстановлению первоначального состояния окон.

Последний пример иллюстрирует возможность использования программного нажатия клавиш для ускорения доступа к программам (имеется в виду программное нажатие сочетаний клавиш, ассоциированных с ярлыками меню Пуск или ярлыками, расположенными на Рабочем столе). Например, если на вашем компьютере используется сочетание клавиш Ctrl+Alt+E для запуска Internet Explorer, то запустить его можно с помощью программного нажатия этого сочетания (листинг 5.20).

**Листинг 5.20.** Быстрый запуск программы

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
  //Нажимаем сочетание 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);
  ::keybd_event(VK_MENU, 0, KEYEVENTF_KEYUP, 0);
  ::keybd_event(VK_CONTROL, 0, KEYEVENTF_KEYUP, 0);
}
```

Способ, реализация которого приведена в последнем примере, особенно полезен, если необходимо запустить сразу несколько приложений (правда, для этого ярлыкам этих приложений должны быть предварительно назначены сочетания клавиш).

## «Бегающие огни» на клавиатуре

В завершение стоит рассмотреть довольно интересный пример, который основан на использовании возможности программного нажатия клавиш Caps Lock, Num Lock и Scroll Lock. Как известно, этим клавишам, по крайней мере на большинстве клавиатур, соответствует три световых индикатора. Идея, положенная в основу примера, состоит в выполнении последовательных нажатий упоминаемых тут клавиш, которые автоматически сопровождаются включением и затем выключением соответствующих индикаторов.

Перед началом рассмотрения основных процедур примера ознакомьтесь с текстом функции `PressKey`, которая далее будет неоднократно использоваться. Эта функция, описание которой приведено в листинге 5.21, имитирует нажатие и отпускание клавиши с переданным кодом.

**Листинг 5.21.** Нажатие одиночной клавиши

```
void PressKey(char keyCode)
{
  //Имитация нажатия клавиши с заданным кодом
  ::keybd_event(keyCode, 0, 0, 0 );
  ::keybd_event(keyCode, 0, KEYEVENTF_KEYUP, 0 );
}
```

Управление функцией осуществляется с помощью кнопки (помимо нее, на форме также должны присутствовать текстовое поле, предназначенное для указания интервала между изменениями состояний индикаторов, и таймер со свойством `Enabled`, равным `false`) (листинг 5.22).

**Листинг 5.22.** Запуск и остановка мерцаний

```
void __fastcall TForm1::cmbStartClick(TObject *Sender)
{
```

```
  if ( cmbStart->Caption == "Пуск" )
  {
    //Сохраняем первоначальные состояния клавиш
    m_InitCaps = ::GetKeyState(VK_CAPITAL) & 1;
    m_InitNum = ::GetKeyState(VK_NUMLOCK) & 1;
    m_InitScroll = ::GetKeyState(VK_SCROLL) & 1;
    //Включаем только Caps Lock
    if ( !m_InitCaps ) PressKey(VK_CAPITAL);
    if ( m_InitNum ) PressKey(VK_NUMLOCK);
    if ( m_InitScroll ) PressKey(VK_SCROLL);
    m_CurCaps = true;
    m_CurNum = false;
    m_CurScroll = false;
    //Запускаем «бегающие огни»
    txtInterval->Enabled = false;
    Timer1->Interval = StrToInt(txtInterval->Text);
    Timer1->Enabled = true;
    cmbStart->Caption = "Стоп";
  }
  else
  {
    //Останавливаем «бегающие огни»
    Timer1->Enabled = false;
    txtInterval->Enabled = true;
    cmbStart->Caption = "Пуск";
    //Восстанавливаем первоначальные состояния клавиш
    if ( m_InitCaps != m_CurCaps ) PressKey(VK_CAPITAL);
    if ( m_InitNum != m_CurNum ) PressKey(VK_NUMLOCK);
    if ( m_InitScroll != m_CurScroll ) PressKey(VK_SCROLL);
  }
}
```

В листинге 5.22 задействованы добавленные вручную переменные-члены класса `TForm1`:

- `m_InitCaps`, `m_InitNum`, `m_InitScroll` — используются для запоминания первоначальных состояний клавиш Caps Lock, Num Lock и Scroll Lock и их восстановления после завершения выполнения программы;
- `m_CurCaps`, `m_CurNum`, `m_CurScroll` — применяются для быстрого определения текущего состояния клавиш.

Смена порядка изменения состояния индикаторов происходит при каждом срабатывании таймера `Timer1` (листинг 5.23).

**Листинг 5.23.** Изменение состояния индикаторов

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
```

```
  //Изменяем состояние индикаторов на клавиатуре
  if ( m_CurCaps )
  {
    //Переключение с Caps Lock на Num Lock
    PressKey(VK_NUMLOCK);
    PressKey(VK_CAPITAL);
    m_CurCaps = false;
    m_CurNum = true;
  }
  else if ( m_CurNum )
  {
    //Переключение с Num Lock на Scroll Lock
    PressKey(VK_SCROLL);
    PressKey(VK_NUMLOCK);
    m_CurNum = false;
    m_CurScroll = true;
  }
  else
  {
    //Переключение с Scroll Lock на Caps Lock
    PressKey(VK_CAPITAL);
    PressKey(VK_SCROLL);
    m_CurScroll = false;
    m_CurCaps = true;
  }
}
```

**ПРИМЕЧАНИЕ**

Если у вашей клавиатуры порядок изменения состояния индикаторов отличен от предполагаемого (Caps Lock ▸ Num Lock ▸ Scroll Lock) в какую-нибудь сторону, то измените их порядок в листинге 5.23, чтобы «бегающие огни» действительно «бежали».

Теперь можно запускать соответствующую заставку — и получится неплохое украшение, например, для новогодней елки... из компьютера.

# Глава 6

## Папки, файлы, диски

- ❑ Диски
- ❑ Папки и пути
- ❑ Файлы и не только

В этой главе вы сможете познакомиться с некоторыми способами получения полезной информации о файловой системе компьютера. Примеры, которые рассматриваются в данной главе, большей частью основаны на использовании API-функций, позволяющих получать информацию, так сказать, из первых рук.

Разработчики среды Borland C++ Builder при написании собственных библиотек не обошли стороной и реализовали возможность работы с файловой системой. Так, в модуле `SysUtils` (файл `SysUtils.hpp`) содержится множество функций, позволяющих преобразовывать пути, создавать, переименовывать, удалять, записывать файлы, а также выполнять много других операций. Поэтому в данной главе в первую очередь будут рассмотрены API-функции, которые позволяют получать информацию, недоступную при использовании функций модуля `SysUtils`. Кроме того, здесь вы найдете примеры реализации достаточно интересных алгоритмов выполнения некоторых операций над файлами и папками.

# Диски

Как вы, наверное, не раз могли убедиться, ряд приложений (хотя бы тот же Internet Explorer) получает гораздо больше информации о дисках, нежели просто их буквенное обозначение или размер. На самом деле информацию эту никто особо и не скрывает — чтобы ее получить, достаточно знать и уметь пользоваться набором соответствующих API-функций.

В данном разделе рассмотрены способы определения обозначений всех установленных на компьютере дисков, их меток, серийных номеров томов и другой информации о файловой системе. Прочитав эту главу, вы также узнаете, как можно программно изменять метки дисков.

Коды всех рассмотренных в этом подразделе функций вы можете найти в модуле `DriveUtils` демонстрационного проекта, который содержится на компакт-диске.

## Сбор информации о дисках

Получить полный список обозначений дисков, подключенных к компьютеру (строк вида `буква:\`), можно с помощью следующей функции (листинг 6.1).

**Листинг 6.1.** Определение обозначений дисков

```
int GetDriveLetters(vector<string> &drives)
{
  char buffer[110];
  int len = ::GetLogicalDriveStrings(110, buffer);
  //Разбираем строку, возвращенную функцией
  //GetLogicalDriveStrings
  int start = 0;
  for ( int i=0; i<len; i++ )
  {
```

```
    if ( buffer[i] == '\0' && start != i )
    {
      //Нашли обозначение очередного диска
      drives.push_back(buffer + start);
      start = i+1;
    }
  }
  return drives.size();
}
```

Данная функция принимает ссылку на вектор строк и заполняет его строками с путями корневых папок каждого диска (например, `C:\`), а возвращает количество строк, добавленных в вектор `drives`. Вся сложность использования этой функции состоит в необходимости выделения путей из строки, заполняемой API-функцией `GetLogicalDriveStrings`.

Кроме API-функции `GetLogicalDriveStrings`, для получения информации об обозначениях дисков можно использовать еще по меньшей мере одну функцию — `GetLogicalDrives`. Эта функция не содержит аргументов и возвращает значение типа `DWORD`, представляющее собой битовую маску. Состояние каждого бита этой маски (от первого до 26-го) свидетельствует о наличии либо отсутствии диска под соответствующей его номеру буквой латинского алфавита. Алгоритм выделения информации из маски (и составления списка дисков) может выглядеть следующим образом (листинг 6.2).

**Листинг 6.2.** Составление списка дисков (второй вариант)

```
int GetDriveLetters2(vector<string> &drives)
{
  //Получаем маску, характеризующую наличие дисков
  DWORD mask = ::GetLogicalDrives();
  if ( mask == 0 ) return 0;
  //Разбираем маску (определяем значения первых 26 битов)
  DWORD i = 1;
  for ( string letter = "A"; letter != "Z"; letter[0]++ )
  {
    if ( (mask & i) != 0 )
    {
      //Есть диск под текущей буквой
      drives.push_back(letter + ":\\");
    }
    i <<= 1; //Переходим к следующему биту
  }
  return drives.size();
}
```

Теперь самое интересное — определение детальной информации о файловой системе каждого диска. Много интересной информации об используемой на диске фай-

ловой системе можно получить с помощью API-функции `GetVolumeInformation`. Эта функция имеет следующий прототип:

```
BOOL GetVolumeInformation(LPCSTR lpRootPathName, LPSTR lpVol-
umeNameBuffer,
 DWORD nVolumeNameSize, LPDWORD lpVolumeSerialNumber,
 LPDWORD lpMaximumComponentLength, LPDWORD lpFileSystemFlags,
 LPSTR lpFileSystemNameBuffer, DWORD nFileSystemNameSize);
```

Код объявления функции выглядит достаточно устрашающе за счет большого количества параметров, но на самом деле использовать функцию `GetVolumeInformation` очень просто. Чтобы не вдаваться в долгое описание параметров, ниже приведен пример с ее использованием (листинг 6.3).

**Листинг 6.3.** Получение информации о диске

```
//Функция определяет информацию о диске. Возвращает false,
//если
//возникла ошибка
bool GetDriveInformation(string root, DriveInfo *pInfo)
{
  char bufDriveName[101], bufFSName[101];
  DWORD SN, maxFileName, fsOptions;
  //Определение информации о диске
  if ( !::GetVolumeInformation( root.c_str(), bufDriveName,
100,
        &SN, &maxFileName, &fsOptions, bufFSName, 100 ) )
  {
    //Ошибка
    return false;
  }
  //Заполняем структуру информацией о диске
  pInfo->DriveLabel = bufDriveName;
  pInfo->FileSystemName = bufFSName;
  pInfo->SerialNumber = SN;
  pInfo->MaxFileNameLen = maxFileName;
  pInfo->DriveType = GetDriveTypeName(root);
  //..параметры файловой системы
  pInfo->FileSystemOptions.CaseSensitive = fsOptions & FS_
CASE_SENSITIVE;
  pInfo->FileSystemOptions.SupportCompression =
   fsOptions & FS_FILE_COMPRESSION;
  pInfo->FileSystemOptions.IsCompressed = fsOptions & FS_VOL_
IS_COMPRESSED;
  return true;
}
```

Анализируя текст приведенного выше листинга, можно заметить, что функции GetVolumeInformation, кроме пути диска (первый параметр), передаются также:

- буфер для сохранения 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);
- указатель на переменную типа DWORD для записи в нее значения максимальной длины пути имени файла или папки;
- указатель на переменную типа DWORD для записи в нее набора битовых флагов, содержащих некоторые параметры файловой системы;
- буфер для хранения названия файловой системы и длина этого буфера.

В результате выполнения описанной функции заполняется структура DriveInfo, код объявления которой, а также содержащейся в ней структуры FSOptions, хранящей извлеченные из битовой маски флаги, приведен в листинге 6.4.

**Листинг 6.4.** Объявление структур для хранения информации о диске

```
//Структура сведений о файловой системе
struct FSOptions
{
  bool CaseSensitive;       //При сравнении путей учитывает
                            //регистр
  bool SupportCompression; //Файловая система поддерживает
                            //сжатие
  bool IsCompressed;        //Диск сжат (compressed)
};
//Структура с информацией о диске
struct DriveInfo
{
  string DriveLabel;            //Метка диска
  string DriveType;             //Тип носителя
  string FileSystemName;        //Файловая система диска
  FSOptions FileSystemOptions; //Параметры файловой системы
  DWORD SerialNumber;           //Серийный номер тома
  DWORD MaxFileNameLen;         //Максимальная длина имени
файла
};
```

Напоследок осталось рассмотреть только способ определения типа носителя диска с помощью API-функции GetDriveType. Эта функция принимает единственный параметр, определяющий корневую папку диска (например, C:\ — причем обратная косая черта в конце обязательна). Функция GetDriveType возвращает целочисленное значение, указывающее тип диска. Вариант кода, позволяющего получить текстовое описание типов дисков с использованием этой функции, приведен в листинге 6.5.

**Листинг 6.5.** Определение типа носителя диска

```
string GetDriveTypeName(string root)
{
  switch ( ::GetDriveType(root.c_str()) )
  {
  case DRIVE_UNKNOWN:
    return "Не определен";
  case DRIVE_REMOVABLE:
    return "Сменный";
  case DRIVE_FIXED:
    return "Фиксированный";
  case DRIVE_REMOTE:
    return "Сетевой";
  case DRIVE_CDROM:
    return "CDROM";
  case DRIVE_RAMDISK:
    return "RAM-диск";
  default:
    return "";
  }
}
```

## Изменение метки диска

Как вы думаете, сложно ли изменить метку диска? На самом деле достаточно просто: сложность состоит лишь в поиске необходимой функции. В данном случае можно применить API-функцию `SetVolumeLabel`, пример использования которой приведен в листинге 6.6.

**Листинг 6.6.** Изменение метки диска

```
bool SetDriveLabel(string root, string newLabel)
{
  return ::SetVolumeLabel(root.c_str(), newLabel.c_str()) ==
true;
}
```

В этом листинге приведена функция-оболочка для API-функции изменения метки диска. Приведенная функция дополняет созданную в рамках этой главы минибиблиотеку полезных функций, предназначенных для работы с дисками.

## Программа просмотра свойств дисков

В завершение раздела будет рассмотрен пример, обобщающий все, что было описано выше, а именно: приложение, выводящее детальную информацию о любом диске, который подключен к компьютеру.

Внешний вид этого приложения приведен на рис. 6.1.

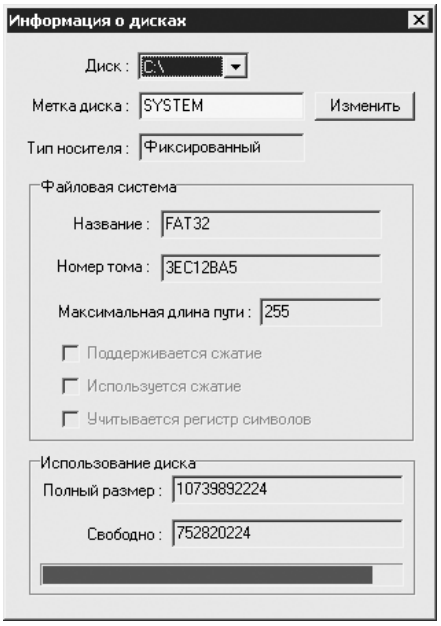


**Рис. 6.1.** Свойства диска

Работа приведенной на рис. 6.1 формы организована предельно просто. Сначала при создании формы составляется список дисков (а также выделяется первый диск и загружается информация о нем) (листинг 6.7).

**Листинг 6.7.** Составление списка дисков

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  //Заполнение списка дисков
  vector<string> drives;
  if ( GetDriveLetters(drives) )
  {
    for ( unsigned int i=0; i<drives.size(); i++ )
    {
      cboDrive->AddItem(drives[i].c_str(), NULL);
    }
    cboDrive->ItemIndex = 0;
  }
  UpdateDriveInfo();
}
```

Загрузка информации о дисках происходит в функции `TForm1::UpdateDriveInfo`, которая, в свою очередь, вызывается при создании формы (что видно из текста листинга 6.7) и при изменении выбора в списке дисков (листинг 6.8).

**Листинг 6.8.** Загрузка информации о выбранном диске

```
void __fastcall TForm1::cboDriveChange(TObject *Sender)
{
  //Покажем информацию о выбранном диске
  UpdateDriveInfo();
}
void TForm1::UpdateDriveInfo()
{
  if ( cboDrive->ItemIndex != -1 )
  {
    //Получение и отображение информации о выбранном диске
    string drive = cboDrive->Items->Strings[cboDrive->
ItemIndex].c_str();
    DriveInfo info;
    GetDriveInformation(drive, &info);
    txtLabel->Text = info.DriveLabel.c_str();
    txtDriveType->Text = info.DriveType.c_str();
    txtFSName->Text = info.FileSystemName.c_str();
    txtFSSerialNum->Text = IntToHex((int)info.SerialNumber,
8);
    txtFSMaxPathLen->Text = IntToStr(info.MaxFileNameLen);
    chkFSCompression->Checked = info.FileSystemOptions.
SupportCompression;
    chkFSCompressed->Checked = info.FileSystemOptions.IsCom-
pressed;
    chkFSCaseSensitive->Checked = info.FileSystemOptions.
CaseSensitive;
    //Доступное место на диске
    __int64 full_space = DiskSize(drive[0] - 'A' + 1);
    __int64 free_space = DiskFree(drive[0] - 'A' + 1);
    txtFullSpace->Text = IntToStr(full_space);
    txtFreeSpace->Text = IntToStr(free_space);
    if ( full_space > 0 )
      //ProgressBar не поддерживает 64-битные значения, поэтому
      //покажем,
      //какой процент диска уже используется
      pbSpaceUsage->Position=100-(double)free_space/
(double)full_space*100;
    else
      pbSpaceUsage->Position = 0;
    txtLabel->Enabled = true;
    cmbSetLabel->Enabled = true;
  }
  else
  {
```

```
    //Очистка формы, так как диск не выбран
    txtLabel->Text = "";
    txtDriveType->Text = "";
    txtFSName->Text = "";
    txtFSSerialNum->Text = "";
    txtFSMaxPathLen->Text = "";
    chkFSCompression->Checked = false;
    chkFSCompressed->Checked = false;
    chkFSCaseSensitive->Checked = false;
    txtFullSpace->Text = "";
    txtFreeSpace->Text = "";
    pbSpaceUsage->Position = 0;
    txtLabel->Enabled = false;
    cmbSetLabel->Enabled = false;
  }
}
```

После нажатия кнопки Изменить производится попытка присвоить выбранному в списке диску метку, указанную в соответствующем текстовом поле (txtLabel) (листинг 6.9).

**Листинг 6.9.** Изменение метки диска

```
void __fastcall TForm1::cmbSetLabelClick(TObject *Sender)
{
  if ( cboDrive->ItemIndex != -1 )
  {
    //Пытаемся присвоить диску новую метку
    string drive = cboDrive->Items->Strings[cboDrive->
ItemIndex].c_str();
    string new_label = txtLabel->Text.c_str();
    if ( !SetDriveLabel(drive, new_label) )
      Application->MessageBoxA("Не удалось изменить метку
диска",
                               "Ошибка", MB_ICONINFORMATION);
  }
}
```

# Папки и пути

Ниже рассмотрены некоторые полезные примеры алгоритмов, позволяющих узнавать расположение важных компонентов Windows. Также в этом разделе приведены некоторые алгоритмы обзора папок, применяемые для поиска, копирования, удаления, перемещения данных и сбора информации о дереве папок.

Прежде чем перейти к рассмотрению решений конкретных задач, следует сказать несколько слов о константе, которая используется в некоторых из приведенных

здесь примеров. Речь идет о константе MAX_PATH, имеющей значение 260. Эта константа используется явно или неявно API-функциями для указания максимально возможной длины пути. Здесь налицо своего рода парадокс: хотя файловая система, такая как FAT32, и реализована так, что может поддерживать неограниченное количество вложенных папок, в реальности создать даже две вложенные папки с именами длиной, к примеру, 255 символов не получится.

После тщательной проверки вышесказанного выяснилось, что создать даже одну папку с именем, имеющим длину 255 символов в корневой папке диска (например, C:\), невозможно — максимальная длина имени ограничена 244 символами. С учетом длины имени корневой папки (в сумме получается 247 символов) можно предположить, что система резервирует оставшиеся 13 символов для того, чтобы сохранить возможность создавать в папке файлы с именами, имеющими формат 8.3 (DOS)[1].

## Системные папки Windows и System

Приходилось ли вам когда-нибудь писать приложения, работоспособность которых зависела от расположения системных папок операционной системы Windows? Если да, то вы наверняка хорошо знаете, насколько ненадежно предположение о том, что папка Windows расположена по адресу C:\Windows, а System — C:\Windows\System, ведь во время установки операционной системы ничто не мешает указать в качестве места назначения, например, диск E:, а папку для Windows назвать Linux. Кроме того, системная папка Windows на платформе NT имеет имя System32, и совсем не обязательно, что оно останется тем же в следующей версии Windows.

В таких и многих других случаях выручают API-функции GetWindowsDirectory и GetSystemDirectory. Обе указанные функции 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 реализовать функции-оболочки, работающие с объектами-строками, что, собственно, и сделано при написании главы: описание всех реализованных функций вы можете найти в модуле PathUtils демонстрационного проекта, который находится на компакт-диске. Итак, функция определения адреса папки Windows выглядит следующим образом (листинг 6.10).

**Листинг 6.10.** Определение папки Windows

```
string GetWinDir()
{
  char buffer[MAX_PATH+1] = "";
  ::GetWindowsDirectory(buffer, MAX_PATH);
  return string(buffer);
}
```

[1] Имя файла состоит из восьми символов, отведенных под название файла, и трех, зарезервированных под расширение.

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

## Имена временных файлов

Для централизованного хранения временных данных, необходимых для обеспечения работы приложений, в Windows предусмотрена специальная папка Temp. Адрес расположения этой папки в Windows может быть различным, причем в многопользовательских версиях Windows (NT, 2000, XP) он может изменяться для различных пользователей. Тем не менее расположение папки Temp можно без труда определить с помощью API-функции GetTempPath.

Функция GetTempPath принимает в качестве своих параметров строковый буфер и его длину и возвращает количество символов, записанных в переданную строку, или 0 в случае возникновения ошибки. Описание способа использования функции-оболочки, скрывающей работу со строковым буфером, приведено в листинге 6.11.

**Листинг 6.11.** Определение адреса папки временных файлов

```
string GetTempDir()
{
  char buffer[MAX_PATH+1] = "";
  ::GetTempPath(MAX_PATH, buffer);
  return string(buffer);
}
```

Кроме того, Windows API предусматривает очень полезный алгоритм, использование которого избавляет программиста от необходимости подбирать имена временным файлам так, чтобы они были уникальными в пределах заданной папки (ей не обязательно должна быть папка Temp). Эта функция имеет название GetTempFileName. Ниже приведен пример использования этой API-функции (листинг 6.12).

**Листинг 6.12.** Назначение имени временному файлу

```
string GetTempFile(string dir, string prefix)
{
  if ( dir.empty() ) dir = GetTempDir();
  //Получение имени временного файла (система сама выбирает
  //имя,
  //являющееся уникальным для заданной папки)
  char buffer[MAX_PATH+1] = "";
  ::GetTempFileName(dir.c_str(), prefix.c_str(), 0, buffer);
  return string(buffer);
}
```

Функция GetTempFileName, код которой приведен в листинге 6.12, в качестве папки, используемой для хранения временных файлов по умолчанию, использует

папку Temp, однако эту функцию можно использовать для получения имен файлов в пределах любой папки.

Кроме пути папки, где нужно создать временный файл, функция `GetTempFileName` принимает в качестве своих параметров строку-префикс для имени временного файла и целочисленное значение (третий параметр). Если это значение не равно нулю, то оно в шестнадцатеричной форме прибавляется к строке `prefix` справа, и никаких проверок на уникальность получившегося имени файла при этом не производится. Если же передать `0` в качестве третьего параметра, то системой автоматически будет сформировано шестнадцатеричное значение таким образом, чтобы получившееся имя файла было уникальным в заданной папке, и при этом создастся сам файл.

Буфер, последний параметр, передаваемый функции `GetTempFileName`, должен вмещать количество символов, равное по меньшей мере константе `MAX_PATH`, так как функция записывает в него полный путь временного файла.

Пример результата работы функций определения адреса системных папок Windows, папки временных файлов, а также выбора имени для временного файла показан на рис. 6.2.

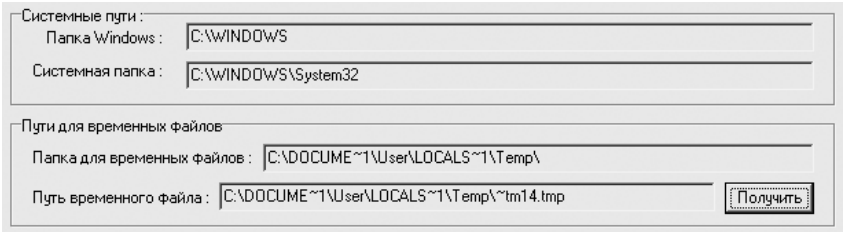


**Рис. 6.2.** Адреса папок Windows, System, Temp и пример имени временного файла

## Прочие системные пути

Кроме рассмотренных выше, в Windows существует ряд других системных путей, которые так или иначе могут пригодиться. Определяются же эти пути не менее просто, чем папки Windows или System (листинг 6.13).

**Листинг 6.13.** Определение прочих системных путей

```
string GetSpecialDir(int dirtype)
{
  char buffer[MAX_PATH+1] = "";
  ::SHGetSpecialFolderPath(0, buffer, dirtype, false);
  return string(buffer);
}
```

Здесь используется функция командной оболочки файловой системы (Windows Shell) `SHGetSpecialFolderPath`, код объявления которой приведен в файле `ShlObj`.

hpp. Среди параметров этой функции самыми значимыми (кроме буфера длиной минимум MAX_PATH символов) являются два последних параметра. Первый из них используется для указания папки, адрес расположения которой необходимо получить. Если последний параметр функции SHGetSpecialFolderPath не равен false и запрашиваемая папка не существует, то она будет создана.

Пример использования функции GetSpesialDir, которая упоминалась в листинге 6.13, для составления списка некоторых системных путей (с записью их в элемент управления ListView) приведен в листинге 6.14. В этом же листинге указаны имена целочисленных констант, идентифицирующих некоторые папки.

**Листинг 6.14.** Использование функции GetSpecialDir

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  ...
  //Прочие системные пути
  struct _SpecDirInfo
  {
    string Name;  //Название папки
    string Path;  //Путь папки
    _SpecDirInfo( string name, string path ) : Name(name),
Path(path){}
  }spec_dirs[] =
  {
    _SpecDirInfo("Рабочий стол"    , GetSpecialDir(CSIDL_
DESKTOPDIRECTORY)),
    _SpecDirInfo("Избранное"         , GetSpecialDir(CSIDL_
FAVORITES)),
    _SpecDirInfo("Шрифты"            , GetSpecialDir(CSIDL_
FONTS)),
    _SpecDirInfo("Мои документы"     , GetSpecialDir(CSIDL_
PERSONAL)),
    _SpecDirInfo("Недавние документы", GetSpecialDir(CSIDL_
RECENT)),
    _SpecDirInfo("История"           , GetSpecialDir(CSIDL_
HISTORY)),
    _SpecDirInfo("Отправить"         , GetSpecialDir(CSIDL_
SENDTO)),
    _SpecDirInfo("Пуск"              , GetSpecialDir(CSIDL_
STARTMENU)),
    _SpecDirInfo("Программы"         , GetSpecialDir(CSIDL_
PROGRAMS)),
    _SpecDirInfo("Автозагрузка"      , GetSpecialDir(CSIDL_
STARTUP)),
    _SpecDirInfo("Шаблоны"           , GetSpecialDir(CSIDL_
TEMPLATES))
```

```
  };
  int dirs_count = sizeof(spec_dirs) / sizeof(_SpecDirInfo);
  for ( int i=0; i<dirs_count; i++ )
  {
    TListItem *pItem = lvwSpecialDirs->Items->Add();
    pItem->Caption = spec_dirs[i].Name.c_str();
    pItem->SubItems->Insert(0, spec_dirs[i].Path.c_str());
  }
}
```

Результат выполнения кода листинга 6.14 продемонстрирован на рис. 6.3.

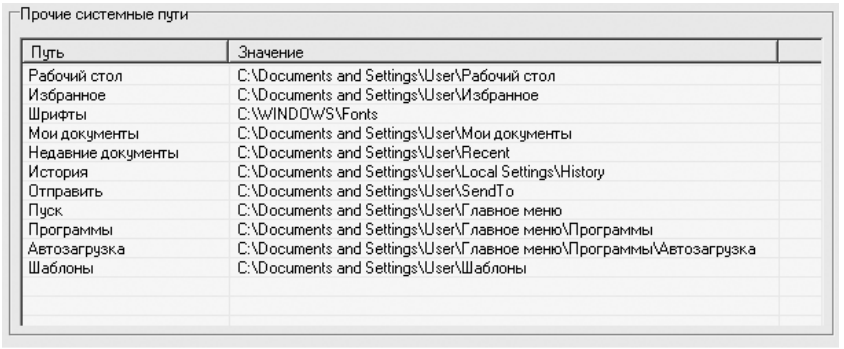


**Рис. 6.3.** Прочие системные пути Windows

В приведенном в листинге 6.14 фрагменте кода определены не все пути, которые могут быть получены с помощью функции `SHGetSpecialFolderPath`. Дело в том, что существует ряд виртуальных (не существующих реально на диске) папок: Мой компьютер, Принтеры, Сетевое окружение и т. д.

Вместе с тем для некоторых из упоминаемых в листинге 6.14 папок существуют аналогичные папки, содержимое которых доступно всем пользователям. Список идентификаторов таких общих папок приводится ниже:

- `CSIDL_COMMON_DESKTOPDIRECTORY` — содержимое этой папки отображается на Рабочем столе всех пользователей;
- `CSIDL_COMMON_DOCUMENTS` — папка 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, содержащая общие для всех пользователей шаблоны документов.

**ПРИМЕЧАНИЕ**

Большинство из перечисленных выше путей определяются в системах Windows **NT, но не определяются в Windows 95, 98 и ME.**

# Поиск

Поиск является неотъемлемой частью работы с файловой системой — даже обыкновенный просмотр содержимого любой папки связан с использованием хотя бы простейших, но все же поисковых средств (перебор и, возможно, отсеивание элементов папки). Поэтому далее будут рассмотрены возможные варианты реализации двух удобных функций поиска: поиска по маске и атрибутам файлов в пределах заданной папки и по всему дереву папок, начиная от заданной корневой папки. Коды всех рассмотренных далее функций поиска можно найти в модуле `SearchUtils` демонстрационного проекта, содержащегося на компакт-диске.

Сначала приведу немного сведений о масках, применяемых для поиска, и атрибутах файлов и папок.

## Маски и атрибуты

Маска имени файла или папки представляет собой строку, в которой неизвестный одиночный символ можно заменять символом `?`, а произвольное количество (ноль и более) неизвестных заранее символов — символом `*`. Обозначения остальных (допустимых в имени) символов не изменяются. Например, имена файлов `SomeFile.exe` и `Some.exe` удовлетворяют любой из следующих масок: `Some*` и `Some*.exe`.

Атрибуты определяют некоторые важные особенности файла. Так, например, при просмотре папки с использованием API-функций эта папка может отличаться от содержащегося в ней файла только наличием атрибута `FILE_ATTRIBURE_DIRECTORY`. Вообще содержимое папок записывается на диск в самые обычные файлы. Эти файлы отличает строго заданный формат записей, наличие атрибута (не изменяемого вручную), а также специальных функций, скрывающих все особенности работы с данными папки (открытия, поиска или удаления записей).

Ниже приведен перечень наиболее часто используемых атрибутов файлов и папок (идентификаторов целочисленных констант). Если не сказано иного, эти атрибуты можно изменять.

- `FILE_ATTRIBUTE_ARCHIVE` — атрибут архивного файла или папки (по опыту замечено, что этот атрибут присваивается практически всем файлам, находящимся на диске некоторое время);
- `FILE_ATTRIBUTE_DIRECTORY` — атрибут папки (этот атрибут нельзя самостоятельно снять или назначить);
- `FILE_ATTRIBUTE_HIDDEN` — скрытый файл или папка;

- FILE_ATTRIBUTE_NORMAL — означает отсутствие особых атрибутов у файла или папки (у последней, естественно, всегда установлен атрибут FILE_ATTRIBUTE_DIRECTORY);
- FILE_ATTRIBUTE_READONLY — файл или папка, предназначенный только для чтения;
- FILE_ATTRIBUTE_SYSTEM — атрибут системного файла или папки;
- FILE_ATTRIBUTE_TEMPORARY — атрибут временного файла (файловая система по возможности включает все содержимое открытого временного файла в память с целью ускорения доступа к хранящимся в нем данным).

Выше 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 папки. Так, атрибуты можно получить во время просмотра содержимого папки (как в рассмотренных ниже функциях поиска) или посредством использования API-функции GetFileAttributes. Функция эта принимает путь файла и возвращает значение типа DWORD, представляющее собой битовую маску. Если функция GetFileAttributes завершается неудачно, то она возвращает значение в виде константы INVALID_HANDLE_VALUE, то есть 0xFFFFFFFF.

Каждому из рассмотренных атрибутов соответствует бит в возвращаемом функцией GetFileAttributes значении. Отрывок программы, проверяющей наличие у файла системного атрибута, приведен ниже:

```
DWORD attrs;
attrs = ::GetFileAttribute("C:\boot.ini");
if (attrs & FILE_ATTRIBUTE_SYSTEM) /*Системный файл*/ ;
```

Установить атрибуты можно с помощью API-функции SetFileAttributes. Эта функция принимает два параметра — путь файла или папки и битовую маску атрибутов — и возвращает ненулевое значение, если атрибуты удалось применить, и 0 (false) в случае неудачи.

Поскольку в функцию SetFileAttributes передается маска, хранящая сведения сразу обо всех атрибутах, то изменять их нужно аккуратно (чтобы не удалить установленные ранее атрибуты). Пример (отрывок кода программы) одновременного назначения одного и снятия другого атрибута файла приведен в листинге 6.15 (проверка ошибок для упрощения не производится).

**Листинг 6.15.** Изменение атрибутов файла

```
DWORD attrs;
attrs = ::GetFileAttributes("C:\text.txt");
attrs |= FILE_ATTRIBUTE_HIDDEN;   //Установка атрибута
"скрытый"
attrs &= ~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 ее элементов с отбором тех из них, имена которых удовлетворяют маске и заданному набору атрибутов. В приведенном ниже примере алгоритма поиска (листинг 6.16) используется API-функция `FindFirstFile`, которая запускает просмотр заданной папки, автоматически отсеивая имена, не удовлетворяющие маске. Данная функция возвращает дескриптор (`HANDLE`) сеанса просмотра папки, используемый при продолжении поиска функцией `FindNextFile`.

После окончания просмотра папки вызывается функция `FindClose`, которая завершает просмотр папки. Процесс очень напоминает работу с обычным файлом (открытие, перебор записей, закрытие), не так ли?

**Листинг 6.16.** Поиск в заданной папке

```
bool SearchInFolder(string folder, string mask, DWORD flags,
                    bool addPath, list<string> &names)
{
  //Начинаем поиск
  string strSearchPath = folder + '\\' + mask;
  bool bRes = false;
  WIN32_FIND_DATA findData;
  HANDLE hSearch = ::FindFirstFile(strSearchPath.c_str(),
&findData);
  if ( hSearch != INVALID_HANDLE_VALUE )
  {
    //Ищем все похожие элементы (информация о первом элементе
    //уже
    //записана в findData функцией FindFirstFile)
    do
    {
      if ( string(findData.cFileName) != ".." &&
           string(findData.cFileName) != "." )  //Пропускаем .
и ..
      {
        if ( MatchAttrs(flags, findData.dwFileAttributes) )
        {
          //Нашли подходящий объект
          if ( addPath )
            names.push_back(folder + "\\" + findData.cFile-
Name);
          else
            names.push_back(findData.cFileName);
          bRes = true;
        }
      }
```

```
    } while ( ::FindNextFile(hSearch, &findData) );
    //Заканчиваем поиск
    ::FindClose(hSearch);
  }
  return bRes;
}
```

В результате работы функции `SearchInFolder` список `names` заполняется именами или, если значение параметра `addPath` равно `true`, полными путями найденных файлов и папок. Значение параметра `flags` (битовая маска атрибутов) для этой функции формируется так же, как и для функции `SetFileAttributes`, только здесь уже одновременно могут быть установлены любые интересующие атрибуты.

При нахождении хотя бы одного файла или папки функция `SearchInFolder` возвращает значение `true`.

В функции поиска проверка соответствия атрибутов найденных файлов и папок заданным в параметрах поиска атрибутам выполняется с использованием дополнительной функции `MatchAttrs`, код которой продемонстрирован в листинге 6.17.

**Листинг 6.17.** Фильтр атрибутов

```
bool MatchAttrs(DWORD flags, DWORD attrs)
{
  return (flags & attrs) == flags;
}
```

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

В листинге 6.17 приведен код реализации нестрогого фильтра: он принимает файл или папку, если они имеют все установленные в параметре `flags` атрибуты, независимо от наличия у файла или папки дополнительных атрибутов. Так, если задать `flags = FILE_ATTRIBUTE_READONLY`, будут найдены все скрытые, системные и прочие файлы и папки, имеющие атрибут `FILE_ATTRIBUTE_READONLY`. Для реализации строгого фильтра можно выражение в функции `MatchAttrs` заменить простым равенством: `flags == attrs`.

Возможный результат поиска с использованием функции `SearchInFolder` продемонстрирован на рис. 6.4.

Пример кода вызова функции `SearchInFolder` из программы (для показанного на рис. 6.4 приложения) приведен в листинге 6.18.

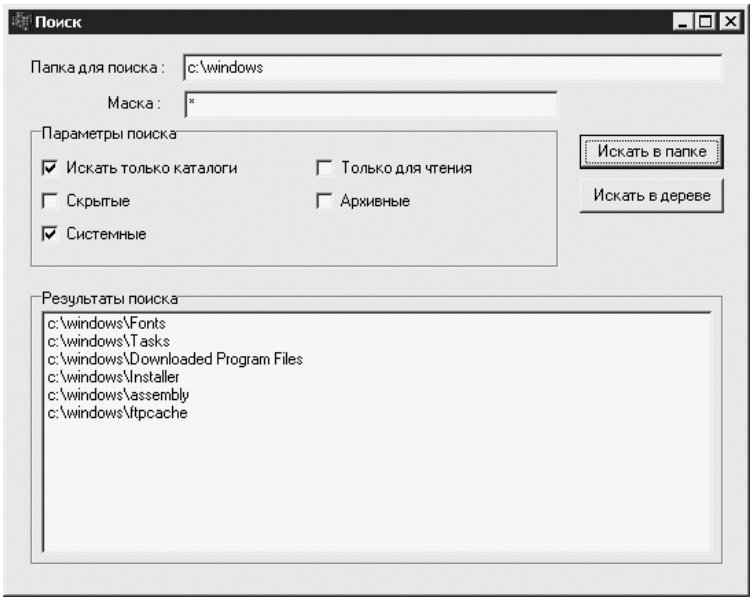


**Рис. 6.4.** Поиск в заданной папке

**Листинг 6.18.** Использование функции SearchInFolder

```
void __fastcall TForm1::cmbSearchInFolderClick(TObject *Send-
er)
{
  Screen->Cursor = crHourGlass;
  //Выполняем поиск в заданной папке
  string folder = txtFolder->Text.c_str();
  string mask = txtMask->Text.c_str();
  DWORD flags = 0;
  if (chkFolder->Checked) flags |= FILE_ATTRIBUTE_DIRECTORY;
  if (chkHidden->Checked) flags |= FILE_ATTRIBUTE_HIDDEN;
  if (chkSystem->Checked) flags |= FILE_ATTRIBUTE_SYSTEM;
  if (chkReadOnly->Checked) flags |= FILE_ATTRIBUTE_READONLY;
  if (chkArchive->Checked) flags |= FILE_ATTRIBUTE_ARCHIVE;
  list<string> result;
  lstResult->Clear();
  if ( SearchInFolder(folder, mask, flags, true, result) )
  {
    grResult->Caption.Format("Найдено %d объектов",
                             ARRAYOFCONST(((int)result.
size())));
    //Покажем список найденных файлов
    std::list<string>::const_iterator it;
    for ( it = result.begin(); it != result.end(); it++ )
```

```
      lstResult->AddItem(it->c_str(), NULL);
  }
  else
  {
    grResult->Caption = "Результаты поиска";
    lstResult->AddItem("Поиск не дал результатов", NULL);
  }
  Screen->Cursor = crDefault;
}
```

## Поиск по всему дереву папок

В листинге 6.19 приведен пример одной из возможных реализаций рекурсивного поиска по дереву папок. Алгоритм поиска работает в следующей последовательности.

1. Сначала выполняется поиск в папке `folder` (все найденные файлы или папки добавляются в список `names`).
2. Затем функция `SearchInTree` вызывается для каждой подпапки для продолжения поиска в поддереве, определяемом отдельной подпапкой.

**Листинг 6.19.** Поиск по дереву папок

```
bool SearchInTree(string folder, string mask, DWORD flags,
                  bool addPath, list<string> &names)
{
  //Осуществляем поиск в текущей папке
  bool bRes = SearchInFolder(folder, mask, flags, addPath,
names);
  //Продолжим поиск в каждой подпапке
  WIN32_FIND_DATA findData;
  HANDLE hSearch = ::FindFirstFile((folder + "\\*").c_str(),
&findData);
  if ( hSearch != INVALID_HANDLE_VALUE )
  {
    do
    {
      if ( string(findData.cFileName) != ".." &&

string(findData.cFileName) != "." )    //Пропускаем . и ..
      {
        if ( findData.dwFileAttributes & FILE_ATTRIBUTE_
DIRECTORY )
        {
          //Нашли подпапку — выполним в ней поиск
          if ( SearchInTree(folder + '\\' + string(findData.
cFileName),
                            mask, flags, addPath, names) )
          {
```

```
            bRes = true;
          }
        }
      }
    } while ( ::FindNextFile(hSearch, &findData) );
    //Завершаем поиск
    ::FindClose(hSearch);
  }
  return bRes;
}
```

В функции `SearchInTree` просмотр папки `folder` производится вручную (с помощью API-функций) из соображений эффективности. Если захотите, можете реализовать поиск подпапок с помощью функции `SearchInFolder`, правда, для этого нужно будет завести дополнительный список для сохранения найденных в текущей папке подпапок. Элементы списка будут использоваться всего один раз: для запуска поиска в подпапках.

Пример возможного результата поиска с использованием функции `SearchInTree` показан на рис. 6.5.

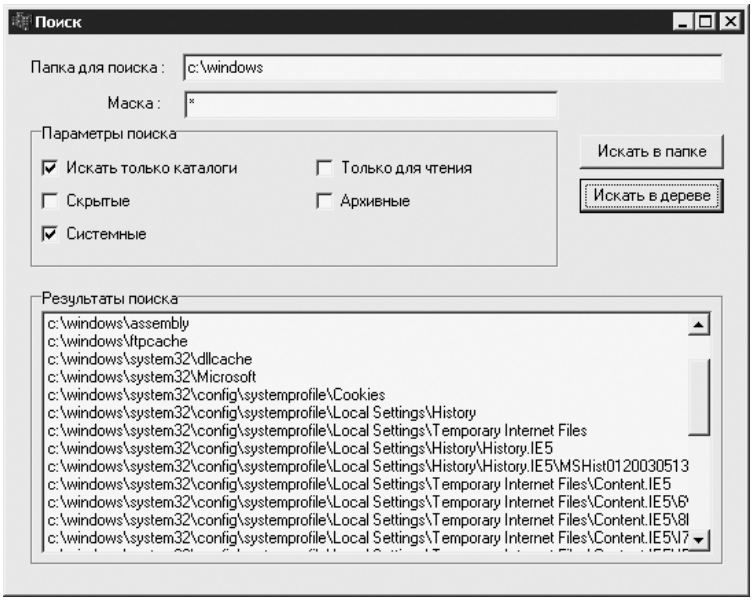


**Рис. 6.5.** Результат поиска в дереве папок

С небольшими модификациями алгоритм рекурсивного обхода дерева папок, реализованный в листинге 6.19, можно использовать и при операциях, которые отличаются от простого поиска: например, при копировании или удалении дерева папок. Собственно, таким операциям над деревом папок и посвящен следующий подраздел.

## Операции над деревом папок

Для выполнения различных операций над деревом папок может возникнуть необходимость обходить это дерево в различном порядке. Например, при копировании нужно обходить дерево папок сверху вниз, сначала создавая папки, а уже затем записывая в них содержимое; при удалении же дерева папок нужно, наоборот, сначала удалить все содержимое папки и только после этого саму папку — то есть обходить дерево снизу вверх; а при такой операции, как вычисление размера всех файлов, находящихся в папке, вообще не имеет значения применяемый для дерева подход.

Поскольку часто операции над деревом папок изначально предполагают определенный способ реализации того или иного обхода всего дерева, то предлагаю реализовать обобщенный алгоритм. В ходе работы этого алгоритма будет сохраняться возможность обрабатывать папку как до, так и после обхода всего ее содержимого.

Код функции обхода дерева приведен в листинге 6.20.

**Листинг 6.20.** Функция обхода дерева папок

```
void ProcessFolderTree(const char *folder, IFSNodeProcessor
*pProcessor)
{
  //Просматриваем заданную папку
  WIN32_FIND_DATA findData;
  HANDLE hSearch =
   ::FindFirstFile((string(folder) + "\\*").c_str(), &find-
Data);
  if ( hSearch == INVALID_HANDLE_VALUE )
    throw std::runtime_error((string("Не удалось открыть папку
") +
                              folder).c_str()) ;
  //Обработаем папку до обработки содержимого
  pProcessor->ProcessFolderBefore(folder);
  do
  {
    if ( string(findData.cFileName) != ".." &&
         string(findData.cFileName) != "." )    //Пропускаем .
                                                //и ..
    {
      if ( findData.dwFileAttributes & FILE_ATTRIBUTE_DIREC-
TORY )
      {
        //Обработаем содержимое подпапки
        ProcessFolderTree((string(folder) + "\\" +
                           findData.cFileName).c_str(), pPro-
cessor);
      }
```

```
      else
      {
        //Обработаем файл
        pProcessor->ProcessFile((string(folder) + '\\' +
                                 string(findData.cFileName)).c_
str());
      }
    }
  } while ( ::FindNextFile(hSearch, &findData) );
  //Завершаем просмотр папки
  ::FindClose(hSearch);
  //Обработаем папку после обработки содержимого
  pProcessor->ProcessFolderAfter(folder);
}
```

Как видите, функция обхода получилась рекурсивной. Что же она делает? Собственно, ничего сложного: она лишь просматривает папку, заданную параметром `folder`, и вызывает себя для каждой найденной подпапки, а при возникновении ошибки функция генерирует исключение `runtime_error`.

Процесс обработки найденных файлов и подпапок заключается в передаче их полных путей в методы объекта, скрытого за интерфейсом `IFSNodeProcessor` (второй параметр функции). Код объявления интерфейса `IFSNodeProcessor` приведен в листинге 6.21.

**Листинг 6.21.** Объявление интерфейса IFSNodeProcessor

```
interface IFSNodeProcessor
{
  virtual void ProcessFile(const char *fullPath) = 0;
  virtual void ProcessFolderBefore(const char *fullPath) = 0;
  virtual void ProcessFolderAfter(const char *fullPath) = 0;
};
```

На всякий случай поясню назначение методов интерфейса:

- `ProcessFile` — данный метод вызывается, когда функция обхода дерева находит очередной файл;
- `ProcessFolderBefore` — метод вызывается, когда функция обхода дерева находит подпапку, но прежде, чем обработает его содержимое;
- `ProcessFolderAfter` — данный метод вызывается уже после того, как функция обхода дерева обработает все содержимое подпапки.

Как видите, в самом обобщенном алгоритме обхода дерева папок ничего особо сложного нет, поэтому можно сразу переходить к рассмотрению нескольких примеров, показывающих возможность применения описанного здесь приема на практике.

Как ни странно, но разрушать гораздо проще, чем строить. Это справедливо и в данном случае — для удаления дерева папок достаточно написать совсем несложный класс (листинг 6.22).

**Листинг 6.22.** Удаление дерева папок

```
class TDeleteProcessor : public IFSNodeProcessor
{
public:
  virtual void ProcessFile(const char *fullPath)
  {
    //Удаление файла
    if ( !DeleteFile(fullPath) )
      throw std::runtime_error(string("Ошибка при удалении
файла ") +
                               fullPath);
  }
  virtual void ProcessFolderBefore(const char *fullPath){}
  virtual void ProcessFolderAfter(const char *fullPath)
  {
    //Удаление папки после того, как удалено все ее
    //содержимое
    if ( !RemoveDir(fullPath) )
      throw std::runtime_error(string("Ошибка при удалении
папки ") +
                               fullPath);
  }
};
```

Немного сложнее выглядит код реализации копирования дерева папок, приведенный в листинге 6.23.

**Листинг 6.23.** Копирование дерева папок

```
class TCopyProcessor : public IFSNodeProcessor
{
  string m_SrcPath;  //Путь копируемой папки
  string m_DestPath; //Путь, куда копируются данные
public:
  TCopyProcessor(const char *source, const char *destination)
  {
    m_SrcPath = source;
    m_DestPath = destination;
  }
  //IFSNodeProcessor
  virtual void ProcessFile(const char *fullPath)
  {
    //Копирование файла в папку назначения (попытка перезаписи
```

```
    //существующих файлов будет вызывать ошибку)
    string newPath = m_DestPath + "\\" +
       ExtractRelativePath((m_SrcPath + "\\").c_str(),
fullPath).c_str();
    if ( !CopyFile(fullPath, newPath.c_str(), true) )
      throw std::runtime_error("Не удалось записать файл " +
newPath);
  }
  virtual void ProcessFolderBefore(const char *fullPath)
  {
    //Создание папки в пункте назначения до обработки
    //содержимого папки
    string newPath = (m_SrcPath != fullPath)
                   ? m_DestPath + "\\" +
                     ExtractRelativePath((m_SrcPath + "\\").
c_str(),
                                         fullPath).c_str()
                   : m_DestPath;
    if ( !CreateDir(newPath.c_str()) )
      throw std::runtime_error("Не удалось создать папку " +
newPath);
  }
  virtual void ProcessFolderAfter(const char *fullPath){}
};
```

Правда, нельзя сказать, чтобы класс `TCopyProcessor`, реализующий копирование, был очень сложным — просто потребовалось перед началом копирования сохранить во внутренних переменных исходный путь и путь назначения, чтобы по мере копирования была возможность определять адрес каждого нового файла или папки. Новое положение определяется преобразованием сначала абсолютного пути в относительный (относительно исходного пути), а затем относительного пути в абсолютный, но уже исходя из пути папки назначения.

Наконец, описание класса, реализующего перемещение дерева папок, приведено в листинге 6.24.

**Листинг 6.24.** Перемещение дерева папок

```
class TMoveProcessor : public IFSNodeProcessor
{
  string m_SrcPath;  //Путь перемещаемой папки
  string m_DestPath; //Путь, куда перемещаются данные
public:
  TMoveProcessor(const char *source, const char *destination)
  {
    m_SrcPath = source;
    m_DestPath = destination;
```

```
  }
  //IFSNodeProcessor
  virtual void ProcessFile(const char *fullPath)
  {
    //Перемещение файла в папку назначения
    string newPath = m_DestPath + "\\" +
        ExtractRelativePath((m_SrcPath + "\\").c_str(),
fullPath).c_str();
    if ( !RenameFile(fullPath, newPath.c_str()) )
      throw std::runtime_error(string("Не удалось переместить
файл ") +
                               fullPath);
  }
  virtual void ProcessFolderBefore(const char *fullPath)
  {
    //Создание папки в пункте назначения до обработки
    //содержимого папки
    string newPath = (m_SrcPath != fullPath)
                   ? m_DestPath + "\\" +
                     ExtractRelativePath((m_SrcPath + "\\").
c_str(),
                                         fullPath).c_str()
                   : m_DestPath;
    if ( !CreateDir(newPath.c_str()) )
      throw std::runtime_error("Не удалось создать папку" +
newPath);
  }
  virtual void ProcessFolderAfter(const char *fullPath)
  {
    //Удаление папки после перемещения всего содержимого
    if ( !RemoveDir(fullPath) )
      throw std::runtime_error(string("Не удалось удалить
папку ") +
                               fullPath);
  }
};
```

По сравнению с копированием код реализации перемещения усложнился лишь процедурой удаления исходных подпапок после перемещения всех входящих в них файлов.

В действительности описанный здесь способ реализации перемещения может пригодиться только в том случае, если нужно переместить дерево папок на другой диск. В остальных случаях можно обойтись API-функцией `MoveFile`: эта функция принимает исходный и конечный пути и способна перемещать не только файлы, но и папки, причем она выполняет перемещение дерева папок очень быстро.

**ПРИМЕЧАНИЕ**

Стоит уточнить, что при реализации копирования и перемещения конечный путь означает не папку, внутрь которой копируется или перемещается дерево папок, а является путем, по которому будет создана корневая скопированная или перемещенная папка. То есть если при копировании указать исходную папку как C:\test, а конечную — C:\test_1, то скопированная корневая папка будет иметь адрес C:\test_1, а не C:\test_1\test. Это же справедливо и при перемещении дерева папок.

Выше были приведены коды реализаций лишь самих объектов, отвечающих за копирование, перемещение и удаление дерева папок. В листинге 6.25 приведен код примера использования этих операций на практике.

**Листинг 6.25.** Применение удаления, копирования и перемещения дерева папок

```
void TForm1::DoTreeProcessing(const char *folder,
                              IFSNodeProcessor *pProcessor)
{
  //Запуск обработки дерева папок с обработкой возможных
  //ошибок
  Screen->Cursor = crHourGlass;
  try
  {
    ProcessFolderTree(folder, pProcessor);
  }
  catch(std::exception &e)
  {
    Application->MessageBox(e.what(), Application->Title.c_
str(),
                            MB_ICONEXCLAMATION );
  }
  Screen->Cursor = crDefault;
}
void __fastcall TForm1::cmbDeleteClick(TObject *Sender)
{
  //Удаление дерева папок
  if ( Application->MessageBox("Удалить все содержимое
папки?",
                               Application->Title.c_str(),
                               MB_ICONQUESTION | MB_YESNO) ==
IDYES )
  {
    DoTreeProcessing(txtDeleteFolder->Text.c_str(),
&TDeleteProcessor());
  }
}
void __fastcall TForm1::cmbCopyClick(TObject *Sender)
```

```
{
  //Копирование дерева папок
  string from = txtFromFolder->Text.c_str();
  string to = txtToFolder->Text.c_str();
  DoTreeProcessing(from.c_str(), &TCopyProcessor(from.c_
str(), to.c_str()));
}
void __fastcall TForm1::cmbMoveClick(TObject *Sender)
{
  //Перемещение дерева папок
  string from = txtFromFolder->Text.c_str();
  string to = txtToFolder->Text.c_str();
  DoTreeProcessing(from.c_str(), &TMoveProcessor(from.c_
str(), to.c_str()));
}
```

Первая функция листинга 6.25 (метод `TForm1::DoTreeProcessing`) представляет собой лишь оболочку, скрывающую функцию обработки возможных ошибок. Три остальные функции представляют собой не что иное, как обработчики нажатий кнопок Удалить, Копировать и Переместить. Единственная задача этих обработчиков — создать и передать в метод `TForm1::DoTreeProcessing` нужные объекты для обработки дерева папок.

Теперь пора переходить к реализации сбора статистики о дереве папок. В листинге 6.26 приведен код примера, подсчитывающего количество файлов и папок (в том числе скрытых и системных отдельно), а также количество файлов, имеющих одинаковое расширение.

**Листинг 6.26.** Сбор статистики о дереве папок

```
class TStatisticsProcessor : public IFSNodeProcessor
{
public:
  //TStaticticsProcessor
  long SubFoldersCount;          //Общее количество подпапок
  long HiddenSubFoldersCount;    //Количество скрытых подпапок
  long SystemSubFoldersCount;    //Количество системных
                                 //подпапок
  long FilesCount;               //Общее количество файлов
  long HiddenFilesCount;         //Количество скрытых файлов
  long SystemFilesCount;         //Количество системных файлов
  map<string, long> CountByExt; //Количество файлов по
                                 //расширениям
  TStatisticsProcessor()
  {
    SubFoldersCount = -1; //Сама исследуемая папка не
                          //учитывается
```

```
    HiddenSubFoldersCount = 0;
    SystemSubFoldersCount = 0;
    FilesCount = 0;
    HiddenFilesCount = 0;
    SystemFilesCount = 0;
  }
  // IFSNodeProcessor
  virtual void ProcessFile(const char *fullPath)
  {
    //Сохраняем данные о файле
    FilesCount++;
    int attr = FileGetAttr(fullPath);
    if ( attr & faHidden ) HiddenFilesCount++;
    if ( attr & faSysFile ) SystemFilesCount++;
    //Обновляем статистику файлов по расширениям
    CountByExt[UpperCase(ExtractFileExt(fullPath)).c_
str()]++;
  }
  virtual void ProcessFolderBefore(const char *fullPath)
  {
    //Сохраняем данные о папке
    SubFoldersCount++;
    int attr = FileGetAttr(fullPath);
    if ( attr & faHidden ) HiddenSubFoldersCount++;
    if ( attr & faSysFile ) SystemSubFoldersCount++;
  }
  virtual void ProcessFolderAfter(const char *fullPath){}
};
```

Пример использования класса `TStatisticsProcessor` в приложении приведен в листинге 6.27.

**Листинг 6.27.** Сбор статистики о дереве папок

```
void __fastcall TForm1::cmbStatClick(TObject *Sender)
{
  lvwSummary->Clear();
  lvwCountByExt->Clear();
  //Получим статистические данные для папки
  TStatisticsProcessor stat;
  DoTreeProcessing(txtStatFolder->Text.c_str(), &stat);
  //Покажем результат
  //..общие сведения
  AddListViewItem(lvwSummary, "Файлов", stat.FilesCount);
  AddListViewItem(lvwSummary, "Папок", stat.SubFoldersCount);
  AddListViewItem(lvwSummary, "Скрытых файлов", stat.Hidden-
FilesCount);
```

```
  AddListViewItem(lvwSummary, "Системных файлов", stat.
SystemFilesCount);
  AddListViewItem(lvwSummary, "Скрытых папок", stat.Hidden-
SubFoldersCount);
  AddListViewItem(lvwSummary, "Системных папок",stat.Hidden-
SubFoldersCount);
  //..количество файлов по расширениям
  std::map<string, long>::const_iterator it;
  for ( it = stat.CountByExt.begin(); it != stat.CountByExt.
end(); it++ )
    AddListViewItem(lvwCountByExt, it->first.c_str(), it-
>second);
}
void TForm1::AddListViewItem(TListView *pList, const char
*text, long value)
{
  TListItem *pItem = pList->Items->Add();
  pItem->Caption = text;
  pItem->SubItems->Insert(0, IntToStr(value));
}
```

В этом примере собранные при обходе дерева папок данные помещаются в два компонента TListView. Возможный вид формы после сбора данных показан на рис. 6.6.

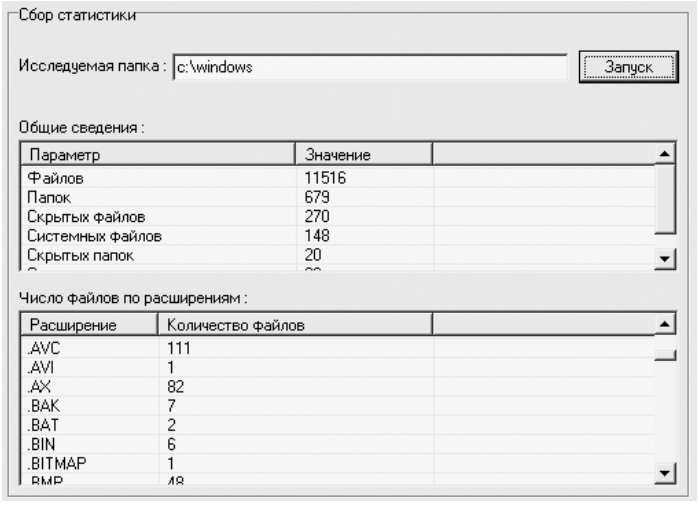


**Рис. 6.6.** Собранная статистика о папке

На этом, пожалуй, завершим рассмотрение простых операций над деревом папок. Однако к этой теме я еще вернусь в следующем разделе, где будет приведен довольно интересный алгоритм, позволяющий обнаружить изменения в дереве папок.

# Отслеживание изменений на диске

В этом подразделе рассмотрены два приложения, позволяющие отслеживать изменения, производимые с диском. Под этими изменениями имеются в виду операции по созданию, удалению файлов или папок, изменению их атрибутов и прочих характеристик.

Первый из рассматриваемых примеров довольно прост — главная его особенность состоит в применении специальных API-функций. Второй же пример алгоритмический. Он демонстрирует еще один способ применения описанного ранее алгоритма обхода дерева папок.

## Использование уведомлений об изменениях

Реализация методов, которые рассматриваются в данном примере, может пригодиться, если нужно установить лишь тот факт, что в наблюдаемой папке произошли изменения, но сделать это необходимо как можно скорее, например, если нужно написать программу, рассылающую каждый новый файл, создаваемый в определенной папке, всем пользователям сети.

В этом случае на помощь придут стандартные функции Windows API, позволяющие приложению получать уведомления об изменениях в отдельной папке или даже в целом дереве папок.

Чтобы иметь возможность следить за папкой, нужно с помощью функции `FindFirstChangeNotification` получить дескриптор уведомления. Прототип этой API-функции выглядит следующим образом:

```
HANDLE FindFirstChangeNotification(
  LPCTSTR lpPathName,     // Путь папки, за которой нужно
следить
  BOOL bWatchSubtree,     // Если true, то следить за
подпапками
  DWORD dwNotifyFilter    // Фильтр отслеживаемых изменений
);
```

Третьим параметром функции `FindFirstChangeNotification` является фильтр изменений, уведомления о которых требуется получать. Значение этого параметра составляется из набора предопределенных констант с помощью операции побитового ИЛИ, то есть представляет собой битовую маску. Константы, которые можно использовать для составления битовой маски, следующие:

- `FILE_NOTIFY_CHANGE_FILE_NAME` — устанавливает необходимость регистрации факта создания, удаления или переименования файла;
- `FILE_NOTIFY_CHANGE_DIR_NAME` — задает необходимость регистрации факта создания, удаления или переименования подпапки;
- `FILE_NOTIFY_CHANGE_ATTRIBUTES` — устанавливает необходимость регистрации любого изменения атрибутов файлов или подпапок;

- `FILE_NOTIFY_CHANGE_SIZE` — определяет необходимость регистрации факта изменения размера файла;
- `FILE_NOTIFY_CHANGE_LAST_WRITE` — задает необходимость регистрации факта изменения времени последней записи в файл;
- `FILE_NOTIFY_CHANGE_SECURITY` — устанавливает необходимость регистрации факта изменения атрибутов защиты файла или подпапки.

Представьте, что вы вызвали функцию `FindFirstChangeNotification` и получили дескриптор уведомления. Что же делать дальше? Теперь нужно в буквальном смысле ждать, пока в папке не произойдет необходимое событие. Для этого можно воспользоваться одной из API-функций, предназначенных для ожидания одного или нескольких событий (в частности, в приведенном примере используется функция `WaitForSingleObject`).

После того как вы дождетесь уведомления об изменении, можно выполнить необходимые действия. Если же предполагается и дальше получать уведомления об изменениях, достаточно вызвать API-функцию `FindNextChangeNotification`, передав ей в качестве параметра полученный дескриптор уведомления. После этого придется ожидать нового уведомления аналогично предыдущему (и так далее для всех последующих уведомлений).

Чтобы завершить получение уведомлений, достаточно вызвать функцию `FindCloseChangeNotification`, передав ей в качестве параметра дескриптор уведомления.

Вот, собственно, и вся схема работы с функциями уведомлений. Теперь можно перейти непосредственно к примеру. На рис. 6.7 показан внешний вид приложения, использующего уведомления об изменениях.

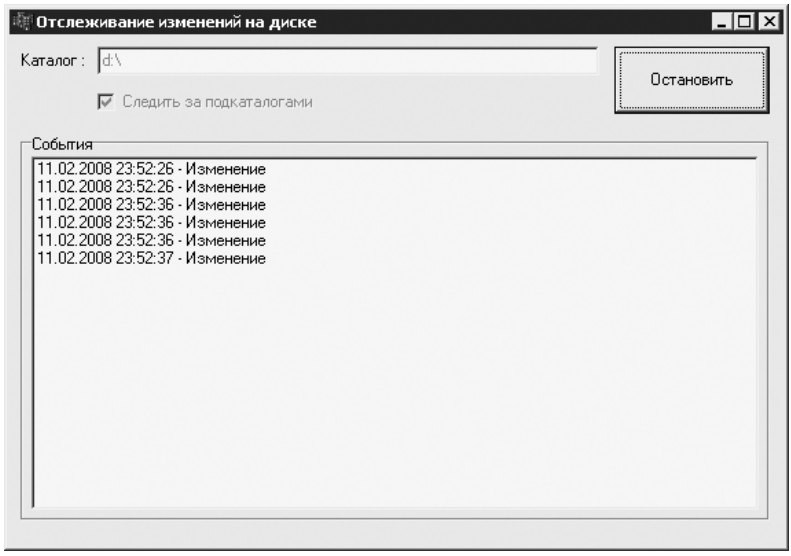


**Рис. 6.7.** Слежение за папкой с использованием уведомлений Windows

Работает пример следующим образом. В текстовом поле указывается адрес папки, за которой нужно следить. Флажок Следить за подкаталогами позволяет указать, нужно следить только за заданной папкой или же еще и за всеми вложенными в нее подпапками. После нажатия кнопки Начать слежение начинается регистрация изменений. Для завершения слежения нужно повторно нажать ту же кнопку (во время слежения название этой кнопки изменяется на Остановить). При получении уведомления об изменении время изменения регистрируется в списке События.

Раз рассмотрение работы программы началось с описания ее интерфейса, то и код будет приведен, начиная с обработчиков пользовательского ввода. По большому счету, такой обработчик в этом примере всего один — это обработчик нажатия кнопки cmbBeginWatch, код которого приведен в листинге 6.28.

**Листинг 6.28.** Обработчик кнопки запуска и остановки слежения

```
void __fastcall TForm1::cmbBeginWatchClick(TObject *Sender)
{
  if ( !m_bIsWatching )
  {
    if ( BeginWatchFolder(txtFolder->Text.c_str(), chkSub-
Folders->Checked) )
    {
      //Начинаем слежение
      lstNotifications->Clear();
      txtFolder->Enabled = false;
      chkSubFolders->Enabled = false;
      cmbBeginWatch->Caption = "Остановить";
      m_bIsWatching = true;
      while( m_bIsWatching )
      {
        WatchFolder(300);
        Application->ProcessMessages();
      }
    }
    else
    {
      Application->MessageBox("Не удалось начать слежение за
папкой",
                              Application->Title.c_str(), MB_
ICONINFORMATION);
    }
  }
  else
  {
    //Завершаем слежение
    txtFolder->Enabled = true;
    chkSubFolders->Enabled = true;
```

```
    cmbBeginWatch->Caption = "Начать слежение";
    m_bIsWatching = false;
  }
}
```

Объявления используемых в листинге 6.28 переменной `m_bIsWatching`, трех вспомогательных функций и еще одной внутренней переменной `m_hNotification`, хранящей дескриптор уведомления, приведены в листинге 6.29.

**Листинг 6.29.** Дополнительные объявления в классе формы

```
class TForm1 : public TForm
{
...
private:  // Пользовательские объявления
  bool m_bIsWatching;
  HANDLE m_hNotification;
  bool BeginWatchFolder( const char *folder, bool bWatchSub-
Tree );
  void WatchFolder( int timeout );
  void EndWatchFolder();
...
};
```

Код **реализации вспомогательных функций** `BeginWatchFolder`, `WatchFolder` и `EndWatchFolder` **приведен в листинге 6.30.**

**Листинг 6.30.** Функции для работы с уведомлениями

```
bool TForm1::BeginWatchFolder( const char *folder, bool
bWatchSubTree )
{
  //Начинаем слежение за папкой, получив дескриптор
  //соответствующего системного объекта
  DWORD filter=FILE_NOTIFY_CHANGE_FILE_NAME | FILE_NOTIFY_
CHANGE_DIR_NAME |
               FILE_NOTIFY_CHANGE_ATTRIBUTES | FILE_NOTIFY_
CHANGE_SIZE |
               FILE_NOTIFY_CHANGE_LAST_WRITE | FILE_NOTIFY_
CHANGE_SECURITY;
  m_hNotification =
   ::FindFirstChangeNotification(folder, bWatchSubTree, fil-
ter);
  return m_hNotification != INVALID_HANDLE_VALUE;
}
void TForm1::WatchFolder( int timeout )
{
  //Ожидаем изменения папки
```

```
  if ( ::WaitForSingleObject(m_hNotification, timeout) ==
WAIT_OBJECT_0 )
  {
    //..обработаем изменение
    lstNotifications->AddItem(Now().DateTimeString()+" -
Изменение", NULL);
    //..подготовимся к ожиданию следующего изменения
    ::FindNextChangeNotification(m_hNotification);
  }
}
void TForm1::EndWatchFolder()
{
  //Завершаем слежение за папкой
  ::FindCloseChangeNotification(m_hNotification);
  m_hNotification = INVALID_HANDLE_VALUE;
}
```

Первая из этих функций (то есть `BeginWatchFolder`) вызывается, чтобы начать слежение за папкой. Именно в этой функции данное приложение получает дескриптор уведомления, который сохраняется в переменной `m_hNotification`.

Функция `WatchFolder` вызывается для ожидания уведомления. При получении уведомления она записывает время получения в список и возвращается к ожиданию получения следующего уведомления.

Наконец, последняя приведенная в листинге 6.30 функция `EndWatchFolder` вызывается при завершении слежения за папкой. Она лишь закрывает дескриптор уведомления. Эта функция также вызывается и при закрытии приложения во время слежения за папкой.

## Сравнение снимков дерева папок

Предыдущий пример позволял только зарегистрировать сам факт того, что в папке, за которой устанавливалось слежение, произошли изменения. Сейчас же будет рассмотрен пример приложения, позволяющего определить, какие именно изменения произошли в наблюдаемой папке.

Принцип работы этой программы состоит в следующем. В начале наблюдения создается снимок дерева папок, то есть сохраняются все необходимые сведения о файлах и подпапках. В конце наблюдения делается новый снимок, который сравнивается с первоначальным. Правда, такой принцип работы накладывает ограничение: в отличие от предыдущего примера, рассматриваемая здесь программа не позволяет регистрировать изменения в реальном времени — но, несмотря на это, думается, что пример все же может пригодиться на практике.

По сути, так часто упоминаемый снимок дерева папок — всего лишь множество (`set`) структур `TFSNodeInfo`. Для начала, используя реализованную ранее функцию обхода дерева папок, нужно создать по одному экземпляру структуры для каждого найден-

ного файла и подпапки. Код объявления структуры TFSNodeInfo, из текста которого вполне понятно, какие сведения нужны программе, приведен в листинге 6.31.

**Листинг 6.31.** Структура для хранения сведений о файле или подпапке

```
struct TFSNodeInfo
{
  string Path;       //Путь файла или папки
  DWORD Attrs;       //Маска атрибутов
  __int64 Size;      //Размер файла
  __int64 DateTime; //Время и дата последнего изменения
  bool operator<(const TFSNodeInfo &second) const
  { //Для возможности сохранения во множестве (set) не
    //забываем
    //реализовать оператор сравнения
    return Path < second.Path;
  }
  TFSNodeInfo(const TFSNodeInfo &src)
  { //Нам также понадобится конструктор копирования
    Path = src.Path;
    Attrs = src.Attrs;
    Size = src.Size;
    DateTime = src.DateTime;
  };
  TFSNodeInfo(const char *path, DWORD attrs, __int64 size, __
int64 datetime)
  {
    Path = path;
    Attrs = attrs;
    Size = size;
    DateTime = datetime;
  }
};
typedef list<TFSChangeInfo> TFSChangeList;
```

В последней строке листинга 6.31 приводится также текст объявления контейнера, который используется для хранения всего снимка дерева папок.

Для сохранения информации о файлах и подпапках также пришлось реализовать класс TFolderItemsToSetSaver (листинг 6.32). Этот класс реализует ранее рассмотренный интерфейс IFSNodeProcessor, необходимый функции обхода дерева папок.

**Листинг 6.32.** Класс, используемый для построения снимка дерева папок

```
class TFolderItemsToSetSaver : public IFSNodeProcessor
{
  string m_RootFolder;
  TFSNodeSet *m_pSet;
```

```
public:
  TFolderItemsToSetSaver(const char *pszRootFolder, TFSNodeSet
*pSet)
  {
    m_RootFolder = pszRootFolder;
    m_pSet = pSet;
  }
  // IFSNodeProcessor
  void ProcessFile(const char *fullPath){__Store(fullPath,
true);}
  void ProcessFolderBefore(const char *fullPath){__
Store(fullPath, false);}
  void ProcessFolderAfter(const char *fullPath){}
private:
  void __Store(const char *fullPath, bool bFile)
  {
    if ( fullPath != m_RootFolder )
    {
      //Сохраняем данные найденного файла или папки
      WIN32_FILE_ATTRIBUTE_DATA attrs;
      ZeroMemory(&attrs, sizeof(attrs));
      if ( ::GetFileAttributesEx(fullPath,
GetFileExInfoStandard, &attrs) )
      {
        m_pSet->insert( TFSNodeInfo(
         ExtractRelativePath((m_RootFolder+"\\").c_str(),
fullPath).c_str(),
         attrs.dwFileAttributes,
         bFile ? ((__int64)attrs.nFileSizeHigh<<32)+attrs.
nFileSizeLow : 0,
         ((__int64)attrs.ftLastWriteTime.dwHighDateTime <<
32) +
                  attrs.ftLastWriteTime.dwHighDateTime ) );
      }
    }
  }
};
```

Для получения детальной информации о файлах и папках в листинге 6.32 используется API-функция `GetFileAttributesEx`. В отличие от рассмотренной ранее в данной главе функции `GetFileAttributes`, эта функция позволяет сразу как получить атрибуты файла или папки, так и определить размер, дату создания, последнего изменения и доступа. В рассматриваемом примере будут учитываться только отличия в атрибутах, размере файлов и дате последнего изменения.

Теперь, когда уже ясно, что представляет собой снимок дерева папок, можно обратиться к интерфейсу примера. Внешний вид программы показан на рис. 6.8.

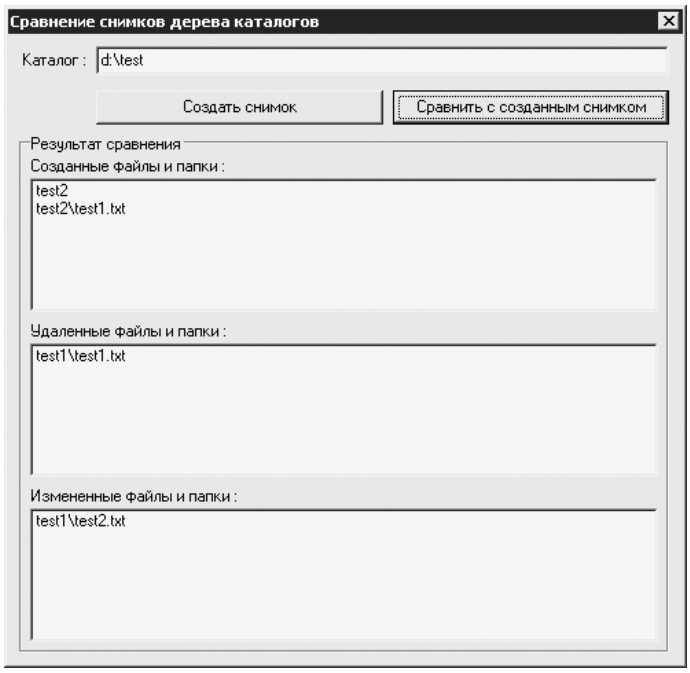


**Рис. 6.8.** Найденные с помощью сравнения снимков изменения папки

Способ работы с программой определяется используемым алгоритмом. Сначала путем нажатия кнопки Создать снимок создается исходный снимок дерева подпапок для папки, указанной в текстовом поле Каталог. Когда приходит время определить, какие изменения были произведены с заданной папкой, пользователь нажимает кнопку Сравнить с созданным снимком — при этом создается еще один снимок, отражающий текущее состояние дерева папок. После этого два снимка сравниваются, и в списках на форме выводятся пути созданных, удаленных и измененных файлов и папок.

Рассматриваемая здесь программа позволяет не только регистрировать изменения, совершенные в наблюдаемой папке, но и сравнивать два разных дерева папок. Именно с этой целью при создании снимка класс `TFolderItemsToSetSaver`, код которого приведен в листинге 6.32, сохраняет не абсолютные, а относительные пути файлов и подпапок. Так, если создать снимок одной папки, затем указать в текстовом поле адрес другой папки и нажать кнопку Сравнить с созданным снимком, то будет выполнено сравнение содержимого двух папок.

В листинге 6.33 приведены коды созданных объявлений в классе формы.

**Листинг 6.33.** Дополнительные объявления в классе формы

```
class TForm1 : public TForm
{
...
private:  // Пользовательские объявления
```

```
  TFSNodeSet *m_pShot; //Данные прошлого снимка дерева папок
  TFSNodeSet *MakeShot(const char *folder);
  TFSChangeList *CompareShots( TFSNodeSet *pBefore, TFSNode-
Set *pAfter );
...
};
```

Здесь переменная `m_pShot` используется для хранения первоначального снимка. Функции `MakeShot` и `CompareShots` позволяют, соответственно, создавать и сравнивать снимки. Код реализации функции `MakeShot` приведен в листинге 6.34.

**Листинг 6.34.** Создание снимка дерева папок

```
TFSNodeSet *TForm1::MakeShot(const char *folder)
{
  Screen->Cursor = crHourGlass;
  TFSNodeSet *pNewShot = new TFSNodeSet;
  try
  {
    //Пытаемся сделать снимок дерева папок
    ProcessFolderTree(folder, &TFolderItemsToSetSaver(folder,
pNewShot));
  }
  catch(std::exception &e)
  {
    //Не удалось получить снимок
    delete pNewShot;
    pNewShot = NULL;
    Application->MessageBox(e.what(), Application->Title.c_
str(),
                            MB_ICONEXCLAMATION );
  }
  Screen->Cursor = crDefault;
  return pNewShot;
}
```

Как видите, создать снимок совсем несложно. Если отвлечься от изменения указателя на время работы функции и от обработки ошибок, то, по большому счету, описание работы функции `MakeShot` сводится к одной строке:

```
ProcessFolderTree(folder, &TFolderItemsToSetSaver(folder,
pNewShot));
```

Самая интересная часть данного примера — собственно сравнение снимков. Ключевую роль играет то, что записи о файлах и подпапках отсортированы по строкам пути в алфавитном порядке. Это позволяет находить созданные или удаленные элементы с помощью очень простого алгоритма, код реализации которого приведен в листинге 6.35.

**Листинг 6.35.** Сравнение снимков дерева папок

```
TFSChangeList *TForm1::CompareShots( TFSNodeSet *pBefore,
                                     TFSNodeSet *pAfter )
{
  TFSChangeList *pChanges = new TFSChangeList;
  TFSNodeSet::const_iterator before = pBefore->begin();
  TFSNodeSet::const_iterator after = pAfter->begin();
  while ( before != pBefore->end() && after != pAfter->end()
)
  {
    int res = strcmp(before->Path.c_str(), after->Path.c_
str());
    if ( res < 0 )
    {
      //Нашли удаленный файл или папку
      pChanges->push_back(TFSChangeInfo(*before, ctDeleted));
      before++;
    }
    else if ( res > 0 )
    {
      //Нашли созданный файл или папку
      pChanges->push_back(TFSChangeInfo(*after, ctCreated));
      after++;
    }
    else
    {
      //Пути двух элементов совпадают. Проверим наличие
      //изменений
      if ( before->Attrs != after->Attrs )
      {
        //..атрибуты отличаются, то есть нашли измененный
        //файл или папку
        pChanges->push_back(TFSChangeInfo(*after,
ctChanged));
      }
      else if ( before->Size != after->Size )
      {
        //..изменился размер файла
        pChanges->push_back(TFSChangeInfo(*after,
ctChanged));
      }
      else if ( before->DateTime != after->DateTime )
      {
        //..изменилась дата последней модификации
        pChanges->push_back(TFSChangeInfo(*after,
ctChanged));
```

```
      }
      before++;
      after++;
    }
  }
  //Оставшиеся элементы снимка «до» (если остались) считаем
  //удаленными
  for ( ; before != pBefore->end(); before++ )
    pChanges->push_back(TFSChangeInfo(*before, ctDeleted));
  //Оставшиеся элементы снимка «после» (если остались)
  //считаем созданными
  for ( ; after != pAfter->end(); after++ )
    pChanges->push_back(TFSChangeInfo(*after, ctCreated));
  return pChanges;
}
```

Используемый алгоритм сравнения состоит из следующих операций.

1. Берутся два первых элемента снимка и сравниваются их пути (если помните, функция `strcmp` возвращает значение больше нуля, если первая строка в алфавитном порядке следует после второй, значение меньше нуля, если верно обратное, и ноль, если строки равны).
2. Если оказывается, что путь элемента из исходного снимка в алфавитном порядке располагается после пути того же элемента в конечном снимке, то делается вывод, что какого-то элемента из исходного снимка нет в конечном снимке. Так находятся удаленные элементы. Далее сравнивается тот же элемент конечного снимка со следующим элементом исходного снимка.
3. Если, наоборот, в алфавитном порядке элемент исходного снимка следует до элемента конечного снимка, то делается вывод, что какого-то элемента из конечного снимка нет в исходном снимке. Так находятся добавленные элементы. Далее следует сравнивать тот же элемент исходного снимка со следующим элементом конечного снимка.
4. Наконец, если пути равны, сравниваются атрибуты, размеры и даты модификации, сохраненные в начальном и конечном снимках. Это позволяет находить измененные элементы. Далее сравнивать нужно следующие элементы обоих снимков.

Если вы внимательно просматривали листинг 6.35, то наверняка заметили, что для каждого найденного изменения создается отдельный экземпляр структуры `TFSChangeInfo` (код ее объявления приведен в листинге 6.36), который помещается в список, возвращаемый функцией `CompareShots`.

**Листинг 6.36.** Структура для хранения данных о найденном изменении

```
enum TFSChangeType{ctDeleted, ctCreated, ctChanged};
struct TFSChangeInfo
{
```

```
  TFSNodeInfo FSNode; //Данные измененного элемента файловой
                      //системы
  TFSChangeType ChangeType; //Тип изменения
  TFSChangeInfo(const TFSChangeInfo &src)
   : FSNode(src.FSNode), ChangeType(src.ChangeType){}
  TFSChangeInfo(const TFSNodeInfo &node, TFSChangeType chan-
geType)
   : FSNode(node), ChangeType(changeType) {}
};
typedef list<TFSChangeInfo> TFSChangeList;
```

Наконец, последнее, без чего не будет работать программа, — обработчики для кнопок Создать снимок и Сравнить с созданным снимком, коды которых приведены в листинге 6.37.

**Листинг 6.37.** Обработчики кнопок создания и сравнения снимков

```
void __fastcall TForm1::cmbShotClick(TObject *Sender)
{
  //Создаем снимок папки (если надо, удаляем старую)
  if ( m_pShot ) delete m_pShot;
  m_pShot = MakeShot(txtFolder->Text.c_str());
  cmbCompare->Enabled = m_pShot != NULL;
}
void __fastcall TForm1::cmbCompareClick(TObject *Sender)
{
  lstCreated->Clear();
  lstDeleted->Clear();
  lstChanged->Clear();
  //Получаем снимок дерева папок в настоящее время
  TFSNodeSet *pNewShot = MakeShot(txtFolder->Text.c_str());
  if ( pNewShot )
  {
    //Сравниваем полученный снимок со снимком, сделанным
    //ранее
    TFSChangeList *pChanges = CompareShots(m_pShot, pNewS-
hot);
    //..покажем изменения
    TFSChangeList::const_iterator it;
    for ( it = pChanges->begin(); it != pChanges->end(); it++
)
    {
      if ( it->ChangeType == ctCreated )
        //Добавленный файл или папка
        lstCreated->AddItem(it->FSNode.Path.c_str(), NULL);
      else if ( it->ChangeType == ctDeleted )
        //Удаленный файл или папка
```

```
        lstDeleted->AddItem(it->FSNode.Path.c_str(), NULL);
      else
        //Измененный файл или папка
        lstChanged->AddItem(it->FSNode.Path.c_str(), NULL);
    }
    delete pChanges;
    delete pNewShot;
  }
}
```

В завершение стоит сказать, что приведенный здесь пример программы был специально испытан на довольно медленном компьютере (процессор Intel Celeron 633). Вопреки моим опасениям по поводу использования контейнера `set` для хранения данных о файлах и папках создание снимка дерева папок для всего жесткого диска (около 87 тыс. файлов) заняло всего 1,5 мин.

# Файлы и не только

В завершение будут рассмотрены три несложных примера, демонстрирующих работу с файлами: копирование больших файлов (с отображением хода копирования в элементе ProgressBar), определение значков, ассоциированных с файлами и папками, а также использование командной оболочки Windows для открытия и печати документов в ассоциированных с ними программах и просмотра содержимого папок в Проводнике Windows.

## Копирование файлов

Казалось бы, что может быть особенного в организации копирования большого файла, даже если нужно отображать ход копирования, — можно читать файл порциями, записывать прочитанные данные в файл назначения, попутно показывая в ProgressBar или где-либо еще отношение объема переписанной информации к размеру файла.

Однако зачем такие сложности? У API-функции `CopyFile`, осуществляющей простое копирование файла, есть расширенный вариант — функция `CopyFileEx`, в которую встроена поддержка отображения процесса копирования (и не только это). Ниже приведен прототип кода функции `CopyFileEx`:

```
BOOL CopyFileEx(
  LPCTSTR lpExistingFileName,//Путь копируемого файла
  LPCTSTR lpNewFileName,     //Путь создаваемого файла
  LPPROGRESS_ROUTINE lpProgressRoutine, // Функция обратного
                                        //вызова
  LPVOID lpData,             //Параметр для функциии обратного
вызова
  LPBOOL pbCancel,           //Адрес переменной, хранящей
                             //статус копирования
```

```
  DWORD dwCopyFlags           //Флаги для настройки
                              //копирования
);
```

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

Пример кода использования функции `CopyFileEx` в программе приведен в листинге 6.38. Здесь подразумевается, что кнопка `cmbCopy` используется для запуска, а кнопка `cmbCancel` — для прерывания копирования. Также на форме присутствуют следующие элементы управления:

- индикатор ProgressBar (`pbProgress`) с диапазоном значений от 0 до 100;
- текстовое поле `txtFromPath`, предназначенное для указания пути копируемого файла;
- текстовое поле `txtToPath`, предназначенное для указания пути файла назначения.

**Листинг 6.38.** Использование функции CopyFileEx

```
void __fastcall TForm1::cmbCopyClick(TObject *Sender)
{
  //Подготовка внешнего вида формы на время копирования
  pbProgress->Position = 0;
  pbProgress->Visible = true;
  cmbCopy->Enabled = false;
  cmbCancel->Enabled = true;
  Screen->Cursor = crAppStart;
  //Запускаем копирование
  m_bCancelCopy = false;
  if ( !::CopyFileEx(txtFromPath->Text.c_str(), txtToPath->
Text.c_str(),
                     ShowProgress, this, &m_bCancelCopy,
                     COPY_FILE_FAIL_IF_EXISTS) )
  {
    Application->MessageBox("Не удалось скопировать указанный
файл",
                            "Копирование", MB_
ICONINFORMATION);
  }
  //Делаем прежним внешний вид формы после копирования
  cmbCopy->Enabled = true;
  cmbCancel->Enabled = false;
```

```
  pbProgress->Visible = false;
  Screen->Cursor = crDefault;
}
void __fastcall TForm1::cmbCancelClick(TObject *Sender)
{
  m_bCancelCopy = true;
}
```

Из текста листинга 6.38 видно, что в качестве значения последнего параметра функции `CopyFileEx` можно передавать константу `COPY_FILE_FAIL_IF_EXISTS` (в этом случае функция вернет `false`, если файл назначения уже существует, и никакого копирования проводиться не будет). На самом деле значение параметра `dwCopyFlags` функции `CopyFileEx` может являться комбинацией значений `COPY_FILE_FAIL_IF_EXISTS` и `COPY_FILE_RESTARTABLE`, то есть представляет собой набор битовых флагов. Константа `COPY_FILE_RESTARTABLE` используется для того, чтобы копирование файла можно было возобновить после прерывания. В этом случае функция `CopyFileEx` сохраняет в файле назначения некоторую информацию, достаточную для возобновления копирования.

Из листинга 6.38 также видно, что в функцию обратного вызова передается указатель на экземпляр формы `TForm1`.

Переменная `m_bCancelCopy`, указатель на которую также передается в функцию `CopyFileEx`, является членом класса `TForm1`.Теперь, наконец, можно рассмотреть функцию обратного вызова, которая в данном случае отвечает за отображение индикатором хода копирования (листинг 6.39).

**Листинг 6.39.** Функция, показывающая ход копирования файла

```
DWORD CALLBACK TForm1::ShowProgress( LARGE_INTEGER TotalFile-
Size,
                              LARGE_INTEGER TotalBytesTrans-
ferred,
                              LARGE_INTEGER StreamSize,
                              LARGE_INTEGER StreamBytesTrans-
ferred,
                              DWORD dwStreamNumber,
                              DWORD dwCallbackReason,
                              HANDLE hSourceFile,
                              HANDLE hDestinationFile,
                              LPVOID lpData )
{
  TForm1 *pForm = (TForm1*)lpData;
  pForm->pbProgress->Position =
   100 * (*(__int64*)&TotalBytesTransferred) / (*(__
int64*)&TotalFileSize);
  Application->ProcessMessages();  //Чтобы не "зависал"
                                   //интерфейс приложения
```

```
  return PROGRESS_CONTINUE;
}
```

Пускай вас не смущает большое количество параметров функции `ShowProgress` — использовать все параметры совсем не обязательно. Функция, код которой представлен в листинге 6.39, реализует использование параметров, на мой взгляд, наиболее простым и очевидным образом: значения параметров `TotalBytesTransferred` и `TotalFileSize` используются для определения доли скопированной информации, а вызов метода `ProcessMessages` объекта `Application` используется потому, что функция `CopyFileEx` возвращает управление программе только после завершения (или прерывания) процесса копирования.

Ниже приведен список целочисленных констант, значения которых может возвращать функция `ShowProgress` (в приводимом примере используется только одно из четырех доступных значений):

- `PROGRESS_CONTINUE` — возврат этого значения указывает на необходимость продолжения копирования;
- `PROGRESS_CANCEL` — возврат этого значения указывает на необходимость отмены копирования;
- `PROGRESS_STOP` — возврат этого значения указывает на необходимость остановки копирования (с возможностью возобновления);
- `PROGRESS_QUIET` — при возврате этого значения система продолжает копирование и перестает вызывать функцию `ShowProgress`.

Внешний вид формы во время копирования большого файла показан на рис. 6.9.

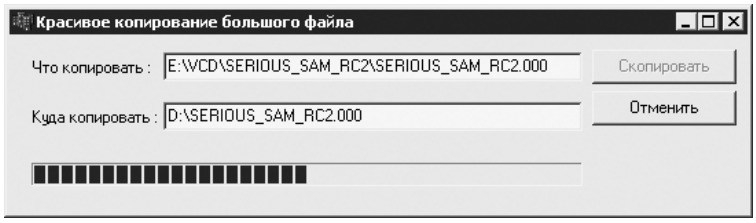


**Рис. 6.9.** Копирование большого файла

Не нужно также забывать о прерывании копирования при закрытии приложения и в прочих экстренных ситуациях. Так, если не предусмотреть обработку события `CloseQuery` для формы, закрыть ее обычным способом в ходе копирования не удастся, зато после завершения копирования (или после нажатия кнопки `Отмена`) форма сразу же исчезнет. Странно, не правда ли? Вариант кода реализации более-менее адекватной реакции формы на закрытие приведен в листинге 6.40.

**Листинг 6.40.** Остановка копирования при закрытии формы

```
void __fastcall TForm1::FormCloseQuery(TObject *Sender, bool
&CanClose)
{
```

```
  m_bCancelCopy = true;
}
```

В качестве варианта можно предложить запретить закрытие формы до завершения копирования (установить параметр `CanClose` в значение `false`).

Для варианта, когда копируется несколько файлов, можно ввести дополнительный элемент управления ProgressBar для отображения хода всего процесса копирования. Правда, при этом придется заранее определить общий размер копируемых файлов.

## Определение значков, ассоциированных с файлами и папками

Обратите внимание на еще один интересный пример, позволяющий получить значок файла или папки, показываемый, например, в Проводнике Windows. Функция `GetFSObjectIcon`, код которой приведен в листинге 6.41, принимает в качестве своих параметров путь файла или папки и флаг, указывающий, большой или малый значок нужно определить, и возвращает дескриптор экземпляра значка, ассоциированного с файлом или папкой.

**Листинг 6.41.** Определение значка файла или папки

```
HICON GetFSObjectIcon(const char *path, bool bLargeIcon)
{
  DWORD flags = SHGFI_ICON;
  if ( bLargeIcon )
    //Получение большого значка
    flags |= SHGFI_LARGEICON;
  else
    //Получение малого значка
    flags |= SHGFI_SMALLICON;
  SHFILEINFO info;
  ZeroMemory(&info, sizeof(info));
  //Получение значка
  SHGetFileInfo(path, 0, &info, sizeof(info), flags);
  return info.hIcon;
}
```

В листинге 6.42 приведен код примера использования функции `GetFSObjectIcon`. В данном подразделе описан пример реализации простейшей программы для просмотра содержимого папки. В нем для получения списка элементов папки используется рассмотренная в разделе «Папки и пути» функция поиска в пределах заданной папки `SearchInFolder`.

**Листинг 6.42.** Добавление в ListView элементов со значками файлов и папок

```
void __fastcall TForm1::cmbRefreshClick(TObject *Sender)
{
```

```
  //Очистка
  lvwLargeIcons->Clear();
  lvwSmallIcons->Clear();
  largeIcons->Clear();
  smallIcons->Clear();
  //Получение содержимого заданной папки
  list<string> folder_items;
  SearchInFolder(txtFolder->Text.c_str(), "*", 0, true,
folder_items);
  //Заполнение списков с одновременным определением значков
  //всех найденных файлов и подпапок
  list<string>::const_iterator it;
  for ( it = folder_items.begin(); it != folder_items.end();
it++ )
  {
    //..добавление названий найденных элементов в список
    //с большими значками
    TIcon *pLargeIcon = new TIcon;
    pLargeIcon->Handle = GetFSObjectIcon(it->c_str(), true);
    largeIcons->AddIcon(pLargeIcon);
    TListItem *pLargeItem = lvwLargeIcons->Items->Add();
    pLargeItem->Caption = ExtractFileName(it->c_str());
    pLargeItem->ImageIndex = largeIcons->Count-1;
    delete pLargeIcon;
    //..добавление названий найденных элементов в список
    //с маленькими
    //значками
    TIcon *pSmallIcon = new TIcon;
    pSmallIcon->Handle = GetFSObjectIcon(it->c_str(), false);
    smallIcons->AddIcon(pSmallIcon);
    TListItem *pSmallItem = lvwSmallIcons->Items->Add();
    pSmallItem->Caption = ExtractFileName(it->c_str());
    pSmallItem->ImageIndex = smallIcons->Count-1;
    delete pSmallIcon;
  }
}
```

Пример результата определения значков файлов и папок показан на рис. 6.10.

В заключение скажу несколько слов о прочих полезных возможностях API-функции `SHGetFileInfo`. Эта функция недаром так называется: она позволяет получить гораздо больше информации, чем просто значок файла. Количество и тип этой информации зависит от набора флагов, передаваемых функции в качестве последнего параметра. Но прежде чем рассматривать эти флаги, ознакомьтесь с назначением полей, из которых состоит структура `SHFILEINFO`, потому что результат (за редким исключением) помещается в поля именно этой структуры:

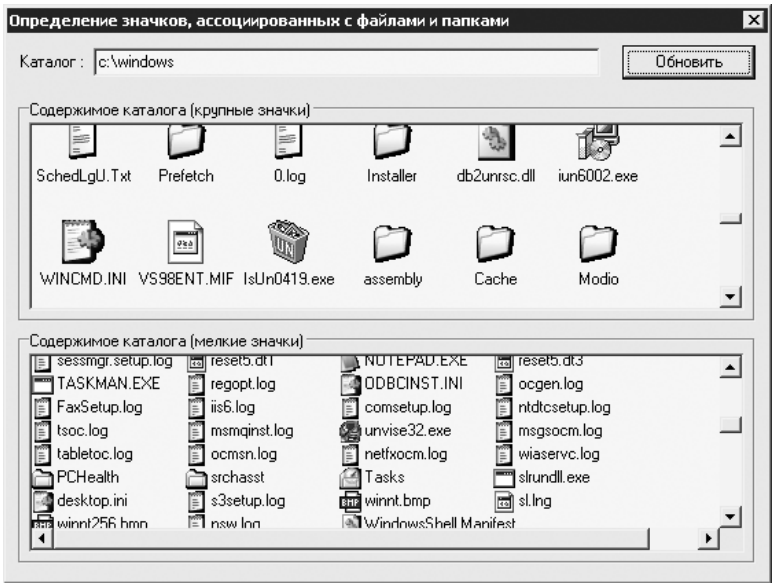


**Рис. 6.10.** Определение значков, ассоциированных с файлами и папками

- поле `hIcon` (тип `HICON`) — содержит дескриптор значка заданного объекта;
- поле `iIcon` (тип `Integer`) — содержит номер значка в системном компоненте ImageList;
- поле `dwAttributes` (тип `DWORD`) — содержит атрибуты заданного объекта;
- поле `szDisplayName` (массив `char`) — содержит буфер хранения для имени заданного пользователем элемента (например, сочетание имени и метки диска, отображаемое в Проводнике);
- поле `szTypeName` (массив `char`) — содержит буфер для хранения названия типа файла (например, документ Microsoft Word).

Подробно останавливаться на описании полей `dwAttributes` и `iIcon` не стоит, зато рассмотрим, как заставить функцию `SHGetFileInfo` заполнить остальные поля структуры. Для этого используются следующие флаги (имена целочисленных констант).

- `SHGFI_ICON` — заполняет поле `hIcon` дескриптором значка, ассоциированного с объектом файловой системы. Если при использовании дескриптор не сохраняется в каком-либо контейнере или другом объекте, автоматически удаляющем ненужные значки, то значок нужно вручную удалить после использования функции (с помощью API-функции `DestroyIcon`).
- `SHGFI_LARGEICON`, `SHGFI_SMALLICON` — применяются в комбинации с флагом `SHGFI_ICON` для получения большого или малого значка соответственно. Использование флагов вместе не имеет смысла — при этом будет получен малый значок.

- SHGFI_DISPLAYNAME — если установлен этот флаг, поле szDisplayName будет заполнено требуемым именем объекта (например, System(C:)).
- SHGFI_EXETYPE — если установлен это флаг, поле szTypeName будет заполнено текстовым описанием типа файла.

Значения в приведенном списке можно, если не сказано иного, комбинировать с помощью операции битового ИЛИ (or).

## Открытие и печать файлов. Открытие Проводника для папок

Наконец, в завершение приведу совсем несложный пример, демонстрирующий (правда, совсем поверхностно) использование командной оболочки Windows. В частности, с помощью API-функции ShellExecute можно открывать или распечатывать, например, документы Microsoft Word. В этом подразделе также будет показано, как открывать Проводник Windows для просмотра отдельных папок на диске.

Для начала кратко опишу саму функцию ShellExecute. Ее прототип выглядит следующим образом:

```
HINSTANCE ShellExecute(
    HWND hwnd,              //Дескриптор окна-владельца
    LPCTSTR lpVerb,         //Действие, которое нужно выполнить
    LPCTSTR lpFile,         //Путь файла или папки
    LPCTSTR lpParameters,   //Аргументы командной строки
    LPCTSTR lpDirectory,    //Рабочая папка для запущенного
                            //приложения
    INT nShowCmd            //Параметры показа окна запущенного
                            //приложения
);
```

Самыми общими словами работу функции ShellExecute можно пояснить так. Если в качестве параметра lpFile указан путь EXE-файла, то система попытается запустить данный файл на выполнение, при этом аргументы командной строки, заданные параметром lpParameters, передадутся запускаемому приложению. Если же файл, путь которого задан параметром lpFile, окажется документом какого-либо приложения, то запустится соответствующее приложение. Запущенное приложение будет выполнять действие, заданное параметром lpVerb. Для этого параметра могут использоваться следующие строковые значения (разные приложения могут не поддерживать некоторые из перечисленных значений):

- edit — открывает файл для редактирования;
- explore — открывает Проводник Windows для просмотра папки, заданной параметром lpFile;
- find — открывает стандартную программу поиска Windows;

- `open` — открывает заданный файл в соответствующем приложении, запускает исполняемый файл или открывает папку в Проводнике Windows;
- `print` — выполняет печать заданного файла;
- `properties` — открывает стандартное окно свойств заданного файла или папки.

Значения параметра `nShowCmd` — комбинация флагов, определяющая внешний вид окна запущенного приложения. В рассматриваемом здесь примере используется значение `SW_NORMAL` (указывает на нормальный размер окна, то есть окно не свернуто и не развернуто на весь экран).

Наконец, первый параметр функции `ShellExecute` может использоваться хотя бы для того, чтобы было к чему «привязать» окно с сообщением об ошибке, которое может быть выведено на экран автоматически при ее возникновении.

Как следует из всего сказанного выше, для открытия файла достаточно вызвать функцию `ShellExecute` с параметром `lpVerb`, равным `open`; чтобы распечатать файл — с параметром `lpVerb`, равным `print`; и, наконец, чтобы открыть Проводник для просмотра папки — с параметром `lpVerb`, равным `explore`.

В случае успешного выполнения команды функция `ShellExecute` возвратит дескриптор запущенного приложения, значение которого больше 32, — меньшие значения свидетельствуют о возникновении ошибки.

Далее будет показано, как внедрить описанные выше возможности в рассмотренную в предыдущем подразделе программу просмотра содержимого папки (рис. 6.11).


**Рис. 6.11.** Новый вид программы просмотра папок

В этом примере при формировании списка файлов и папок с каждым элементом списка будут ассоциироваться дополнительные данные, а именно: указатели на объекты класса `TFolderItem`, код которого приведен в листинге 6.43.

**Листинг 6.43.** Класс, описывающий элемент папки

```
class TFolderItem : public TObject
{
  string m_Path; //Путь файла или папки на диске
  bool m_IsFile; //Если true, то файл, если false, то папка
public:
  TFolderItem(const char *path)
  {
    m_Path = path;
    //Определим, путь папки или файла сюда передан
    DWORD attrs = ::GetFileAttributes(path);
    if ( attrs == INVALID_FILE_ATTRIBUTES)
      m_IsFile = true;
    else
      m_IsFile = attrs & FILE_ATTRIBUTE_DIRECTORY ? false :
true;
  }
  bool IsFile(){ return m_IsFile; }
  bool Explore()
  {
    //Открытие папки в Проводнике Windows
    if ( IsFile() ) return false;
    return (int)::ShellExecute( Application->Handle,
"explore",
                                m_Path.c_str(), NULL, NULL,
SW_SHOWNORMAL )
                                > 32;
  }
  bool Open()
  {
    //Открытие файла
    if ( !IsFile() ) return false;
    return (int)::ShellExecute( Application->Handle, "open",
                                m_Path.c_str(), NULL, NULL,
SW_SHOWNORMAL )
                                > 32;
  }
  bool Print()
  {
    //Печать файла
    if ( !IsFile() ) return false;
```

```
    return (int)::ShellExecute( Application->Handle, "print",
                                m_Path.c_str(), NULL, NULL,
SW_SHOWNORMAL )
                                > 32;
  }
};
```

Как видно из листинга 6.43, в классе `TFolderItem`, кроме хранения полного пути, реализовано определение, является ли элемент, путь которого передан, файлом. Кроме того, класс `TFolderItem` включает элементы использования функции `ShellExecute`.

Код заполнения списка элементов папки, учитывающих произведенные изменения, приведен в листинге 6.44. Стоит отметить, что в этом примере владельцем созданных объектов считается не элемент ListView на форме, а список `m_FolderItems`, в который помещаются все созданные объекты `TFolderItem`.

**Листинг 6.44.** Новое заполнение списка элементов папки

```
void __fastcall TForm1::cmbRefreshClick(TObject *Sender)
{
  //Очистка
  lvwLargeIcons->Clear();
  largeIcons->Clear();
  DeleteFolderItems();
  //Получение списка подпапок и файлов
  list<string> folder_items;
  SearchInFolder(txtFolder->Text.c_str(), "*", 0, true,
folder_items);
  //Заполнение списков с одновременным определением значков
  //всех найденных файлов и подпапок
  list<string>::const_iterator it;
  for ( it = folder_items.begin(); it != folder_items.end();
it++ )
  {
    TIcon *pLargeIcon = new TIcon;
    pLargeIcon->Handle = GetFSObjectIcon(it->c_str(), true);
    largeIcons->AddIcon(pLargeIcon);
    delete pLargeIcon;
    TListItem *pLargeItem = lvwLargeIcons->Items->Add();
    pLargeItem->Caption = ExtractFileName(it->c_str());
    pLargeItem->ImageIndex = largeIcons->Count-1;
    TFolderItem *pFolderItem = new TFolderItem(it->c_str());
    m_FolderItems.push_back(pFolderItem);
    pLargeItem->Data = pFolderItem;
  }
}
```

Код функции DeleteFolderItems, вызов которой осуществляется кодом, представленным в листинге 6.44, и которая вызывается при закрытии формы, приведен в листинге 6.45.

**Листинг 6.45.** Удаление данных со сведениями об элементах папки

```
void TForm1::DeleteFolderItems()
{
  //Удаление объектов с информацией об элементах папки
  list<TFolderItem*>::iterator it;
  for ( it = m_FolderItems.begin(); it != m_FolderItems.end();
it++ )
    delete *it;
  m_FolderItems.clear();
}
```

Наконец, чтобы создать меню, показанное на рис. 6.11, нужно поместить компонент PopupMenu на форму, назначить его в качестве открывающегося щелчком правой кнопкой мыши меню для списка lvwLargeImages (свойство PopupMenu), создать в этом меню пункты Проводник, Открыть и Печать и написать обработчики, код которых приведен в листинге 6.46.

**Листинг 6.46.** Последний штрих — меню

```
void __fastcall TForm1::menuPopup(TObject *Sender)
{
  if ( lvwLargeIcons->ItemIndex == -1 )
  {
    //Не выделен ни файл, ни папка
    mnuExplore->Enabled = false;
    mnuOpen->Enabled = false;
    mnuPrint->Enabled = false;
    menu->Tag = 0;
  }
  else
  {
    //Активируем пункты меню в зависимости от того,
    //выделен файл или папка
    TListItem *pItem = lvwLargeIcons->Items-
>Item[lvwLargeIcons->ItemIndex];
    TFolderItem *pFolderItem = (TFolderItem*)pItem->Data;
    mnuExplore->Enabled = !pFolderItem->IsFile();
    mnuOpen->Enabled = pFolderItem->IsFile();
    mnuPrint->Enabled = pFolderItem->IsFile();
    menu->Tag = (int)pFolderItem;
  }
}
```

```
void __fastcall TForm1::mnuOpenClick(TObject *Sender)
{
  ((TFolderItem*)menu->Tag)->Open();
}
void __fastcall TForm1::mnuPrintClick(TObject *Sender)
{
  ((TFolderItem*)menu->Tag)->Print();
}
void __fastcall TForm1::mnuExploreClick(TObject *Sender)
{
  ((TFolderItem*)menu->Tag)->Explore();
}
```

Теперь пример полностью готов. Хочется верить, что он довольно неплохо демонстрирует простоту способов внедрения в приложение таких возможностей файловых менеджеров, как печать файлов, просмотр и редактирование файлов и папок.

# Глава 7

## Ресурсы

- Общие вопросы работы с ресурсами
- Использование ресурсов в приложениях
- Ресурсы других приложений

Ресурсы — особый вид данных, которые Windows позволяет сохранять в модулях с исполняемым кодом приложения, например в EXE- или DLL-файлах. В качестве ресурсов могут выступать данные как встроенных в Windows (системных), так и пользовательских типов. Примерами первых являются растровые изображения, значки, курсоры, строки и прочие данные, для которых в Windows API предусмотрены средства обработки. К категории ресурсов, тип которых определяется пользователем (или, точнее, программистом), можно отнести любые данные, для которых нужно самостоятельно реализовывать средства обработки.

Что же дают ресурсы, чего не может дать использование внешних файлов, ведь вряд ли может быть что-то особо сложное в том, чтобы вместо ресурсов создать нужное количество файлов с данными на диске? Но не зря механизм ресурсов встроен в саму Windows — использование ресурсов вместо внешних файлов с данными предоставляет как минимум следующие преимущества.

- Сохранение целостности данных приложения — это означает, что ресурс из файла приложения никуда пропасть не может, как это бывает с файлами на диске (особенно у неопытных пользователей), а это, в свою очередь, означает, что нет необходимости предусматривать специальные средства обработки ошибок, которые могут возникнуть при неудачном обращении к данным приложения.
- Простота разворачивания и обновления приложения — если данные приложения собраны в ресурсах одного или нескольких исполняемых файлов, то после установки или обновления приложения достаточно лишь скопировать или заменить эти файлы; нет необходимости заботиться о написании сложных сценариев распаковки и копирования файлов данных по нужным адресам, а также об их удалении.
- Простота обращения к данным, хранимым в ресурсах, — для загрузки нужного ресурса часто достаточно вызвать всего одну функцию, передав ей в качестве параметра идентификатор этого ресурса.
- Не слишком надежная, но тем не менее защита данных приложения от копирования — неопытный пользователь, который по крайней мере не сможет изменить содержимое данных программ. Насколько надежна предоставляемая защита ресурсов от опытных пользователей и программистов, вам станет понятно по мере изучения примеров главы.
- Возможность интернационализации приложения — с помощью ресурсов можно легко изменять данные, специфичные для разных языков.

Естественно, использование ресурсов не является панацеей, которая избавляет от всех трудностей, возникающих при работе с данными приложения, — более того, использование ресурсов налагает ряд ограничений и требует более дисциплинированного подхода к программированию (об этом будет рассказано подробно при рассмотрении строковых ресурсов).

# Общие вопросы работы с ресурсами

Сейчас предлагаю перейти к более подробному рассмотрению некоторых общих вопросов, которые могут возникнуть при работе с ресурсами Windows.

## Виды ресурсов

Для работы с системными ресурсами предназначены специальные API-функции. Основные распространенные типы этих ресурсов приведены ниже:

- растровые изображения (bitmaps) — обычные растровые изображения формата BMP;
- метафайлы (metafiles) — векторные изображения WMF;
- значки (icons);
- указатели (cursors);
- окна (dialogs) — шаблоны окон; в Windows API предусмотрен целый ряд функций для работы с окнами, которые создают их по записанному в ресурсах шаблону;
- шрифты (fonts);
- меню (menus) — записи о структуре меню и параметрах элементов меню;
- таблицы строк (string tables) — группы, в которые объединяются строки при сохранении в ресурсах;
- данные о версии приложения (version information).

Приведенный здесь список, конечно, не является полным и ограничен лишь теми ресурсами, которые наиболее часто используются при написании программ в среде Borland C++ Builder.

## Windows API для работы с ресурсами

В этом подразделе вкратце рассказывается об основных возможностях работы с ресурсами, предусмотренными Windows API, и перечисляются этапы выполнения операций с ресурсами. Примеры, демонстрирующие способы использования большинства описанных в этом подразделе функций, приведены далее в тексте главы.

### Общая схема работы с ресурсами

Так как ресурсы хранятся в исполняемом файле, то прежде всего нужно получить дескриптор модуля этого файла. Если речь идет о ресурсах, хранимых в EXE-файле приложения, то для хранения этого дескриптора в программе используется глобальная переменная `HInstance`. Большинство функций, предназначенных для работы с ресурсами, в качестве дескриптора модуля могут принимать значение `NULL`, которое означает, что обращение идет именно к ресурсам EXE-файла самого приложения — в противном случае потребуется загрузить исполняемый файл

в память с помощью одной из API-функций, например LoadLibrary, имеющей следующий прототип:

```
HMODULE LoadLibrary(LPCTSTR lpFileName);
```

Единственный аргумент функции LoadLibrary используется для указания пути исполняемого файла. При успешном завершении функция возвращает дескриптор загруженного в память модуля и NULL — в противном случае.

После завершения загрузки модуля в память становится возможным использовать его ресурсы. Для некоторых типов ресурсов, перечисленных в предыдущем подразделе, доступны следующие стандартные функции загрузки:

```
HBITMAP LoadBitmap(HINSTANCE hInstance, LPCTSTR lpBitmap-
Name);
HCURSOR LoadCursor(HINSTANCE hInstance, LPCTSTR lpCursor-
Name);
HICON LoadIcon(HINSTANCE hInstance, LPCTSTR lpIconName);
HMENU LoadMenu(HINSTANCE hInstance, LPCTSTR lpMenuName);
int LoadString(HINSTANCE hInstance, UINT uID,
               LPTSTR lpBuffer, int nBufferMax);
```

Первые четыре приведенные функции предназначены для загрузки растровых изображений, курсоров, значков и меню соответственно. Каждая из этих функций принимает в качестве параметров дескриптор модуля и идентификатор ресурса и возвращает дескриптор объекта, созданного в памяти на основе данных ресурса. Последняя указанная функция используется для загрузки строк. Помимо идентификатора, который для строк может быть только целочисленным, и дескриптора модуля функция принимает адрес и размер буфера, в который предполагается скопировать текст из ресурса.

## Загрузка бинарных ресурсов

Естественно, кроме загрузки системных типов ресурсов существует также возможность загружать ресурсы других типов. Для этого предназначены функции FindResource и LoadResource, имеющие следующие прототипы:

```
HRSRC FindResource(HMODULE hModule, LPCTSTR lpName, LPCTSTR
lpType);
HGLOBAL LoadResource(HMODULE hModule, HRSRC hResInfo);
```

Первая из приведенных функций используется для поиска ресурса типа lpType с идентификатором lpName в модуле hModule. В случае успешного выполнения команды функция FindResource возвращает дескриптор найденного ресурса и значение NULL — при неудаче. Вторая функция загружает найденный ресурс в память.

Правда, с использованием функции LoadResource связана одна особенность: как видите, она возвращает не указатель на участок памяти, а значение типа HGLOBAL, являющееся дескриптором данных ресурса, размещенных в памяти. Для получения

указателя на участок памяти, в котором хранятся данные ресурса, придется вызвать API-функцию LockResource, имеющую следующий прототип:

```
LPVOID LockResource(HGLOBAL hResData);
```

С помощью описанных выше трех функций можно, в частности, работать с бинарными ресурсами, а также с любыми стандартными ресурсами, которые в данном случае будут рассматриваться как бинарные.

## Перечисление ресурсов в исполняемом файле

Набор функций Windows API предусматривает возможности перечисления ресурсов, которые могут оказаться полезными при проверке исполняемых файлов на предмет наличия тех или иных ресурсов. Так, для перечисления типов ресурсов, хранящихся в исполняемом модуле, пригодится функция EnumReourceTypes:

```
BOOL EnumResourceTypes(HMODULE hModule,
                       ENUMRESTYPEPROC lpEnumFunc, LONG_PTR
lParam);
```

Функция EnumResourceTypes, помимо дескриптора модуля, принимает указатель на функцию обратного вызова, которую Windows будет вызывать для каждого найденного типа ресурса, а также пользовательское значение, дополнительно передаваемое в функцию обратного вызова. Код объявления функции обратного вызова выглядит следующим образом:

```
BOOL CALLBACK EnumTypesFunc(HANDLE hModule, LPTSTR lpType,
LONG lParam);
```

Как видите, функция обратного вызова получает дескриптор модуля, имя типа ресурса и параметр hModule, значение которого уже было передано в API-функцию EnumResourceTypes. Эта функция обратного вызова возвращает значение true, если нужно продолжать перечисление типов ресурсов, и false, если перечисление требуется завершить. Вместе с тем в качестве параметра hModule в функцию обратного вызова системой может передаваться как строковое значение, так и целое число, приведенное к указателю LPTSTR. В частности, системные типы ресурсов имеют целочисленные идентификаторы. Определить, какой именно параметр получила функция обратного вызова, можно с помощью макроса IS_INTRESOURCE:

```
#define IS_INTRESOURCE(_r) (((ULONG_PTR)(_r) >> 16) == 0)
```

Следующая функция, которую можно применять уже к ресурсам конкретного типа, имеет имя EnumResourceNames. Она позволяет получить идентификаторы всех ресурсов любого заданного типа. Прототип функции EnumResourceNames выглядит следующим образом:

```
BOOL EnumResourceNames(HMODULE hModule, LPCTSTR lpszType,
                       ENUMRESNAMEPROC lpEnumFunc, LONG_PTR
lParam);
```

Как видите, ничего особо сложного в ней нет: функция принимает дескриптор модуля, имя (или целочисленный идентификатор) типа ресурса, адрес функции обратного вызова и параметр, передаваемый в функцию обратного вызова. Код определения функции обратного вызова, используемой в этом случае, имеет следующий вид:

```
BOOL CALLBACK EnumResNameProc(HMODULE hModule, LPCTSTR lpsz-
Type,
                              LPTSTR lpszName, LONG_PTR
lParam);
```

Все сказанное о функции `EnumResourceNames` действительно и для этой функции, только параметр `lpszName` тоже может быть как указателем на строку, так и целочисленным идентификатором. Для определения содержания параметра `lpszName` также используется макрос `IS_INTRESOURCE`.

Ниже приведен список предопределенных целочисленных идентификаторов типов ресурсов, доступных для передачи в функцию `EnumResourceNames`, для которых в документации удалось найти осмысленные пояснения:

- `RT_ACCELERATOR` — таблица «быстрых клавиш»;
- `RT_ANICURSOR` — анимированный указатель мыши;
- `RT_ANIICON` — анимированный значок;
- `RT_BITMAP` — растровый рисунок (BMP);
- `RT_CURSOR` — указатель мыши;
- `RT_DIALOG` — шаблон окна;
- `RT_FONT` — шрифт;
- `RT_FONTDIR` — каталог шрифтов;
- `RT_GROUP_CURSOR` — аппаратно-независимый указатель мыши;
- `RT_GROUP_ICON` — аппаратно-независимый значок;
- `RT_HTML` — HTML-документ;
- `RT_ICON` — значок;
- `RT_MANIFEST` — XML-манифест (в Windows XP с его помощью можно изменять настройки запускаемой программы, например цвет и шрифт окон);
- `RT_MENU` — меню;
- `RT_MESSAGETABLE` — элементы карты сообщений;
- `RT_RCDATA` — бинарный ресурс;
- `RT_STRING` — строка;
- `RT_VERSION` — версия приложения.

## Изменение и обновление ресурсов

Помимо считывания данных из ресурсов, Windows API предоставляет ряд функций для изменения данных в секции ресурсов исполняемого файла. К сожалению, тема обновления данных ресурсов выходит за рамки темы данной главы, но тем не менее я считаю своим долгом хотя бы вкратце рассказать об особенностях этого процесса.

Итак, для изменения содержимого секции ресурсов исполняемого файла предназначены три функции. Функцией, которая должна быть вызвана первой, является функция `BeginUpdateResource`:

```
HANDLE BeginUpdateResource(LPCTSTR pFileName,BOOL bDeleteExis
tingResources);
```

Эта функция принимает в качестве своих параметров имя файла и флаг, значение `true` которого указывает на необходимость удалять из файла все первоначально присутствующие в нем ресурсы, и возвращает дескриптор, который передается в две оставшиеся функции.

После получения дескриптора обновляемого файла можно использовать функцию `UpdateResource`, код объявления которой выглядит следующим образом:

```
BOOL UpdateResource(HANDLE hUpdate, LPCTSTR lpType,
                    LPCTSTR lpName,
                    WORD wLanguage, LPVOID lpData,
                    DWORD cbData);
```

Параметры этой функции имеют следующие значения:

- `hUpdate` — указывает дескриптор файла, ресурсы которого обновляются;
- `lpType` — задает тип обновляемого ресурса, строку или целочисленное значение, преобразованное в строковый тип макросом `MAKEINTRESOURCE`;
- `lpName` — определяет идентификатор ресурса, строку или целочисленное значение, преобразованное в строковый тип макросом `MAKEINTRESOURCE`;
- `wLanguage` — указывает идентификатор языка; если ресурс предназначен для всех языков, то можно использовать выражение;
- `lpData` — принимает в качестве своего значения данные ресурса или `NULL`, если нужно удалить старый ресурс;
- `cbData` — определяет размер данных ресурса.

Наконец, после выполнения с помощью функции `UpdateResource` всех требуемых обновлений нужно завершить работу с исполняемым файлом, вызвав API-функцию `EndUpdateResource`, имеющую следующий вид:

```
BOOL EndUpdateResource(HANDLE hUpdate, BOOL fDiscard);
```

Первым параметром функции EndUpdateResource является дескриптор обновляемого файла. Параметр fDiscard позволяет указать, нужно ли фиксировать, то есть сохранять, в файл изменения, совершенные с помощью функции UpdateResource. Чтобы сохранить все изменения, значение параметра fDiscard нужно установить в false, и, соответственно, чтобы отменить изменения, параметру fDiscard нужно передать значение true.

## Создание файла ресурсов

Ниже кратко рассмотрен способ добавления ресурсов в приложение. Речь идет о создании файла, имеющего расширение RC и представляющего собой обычный текстовый файл, в котором с помощью записей специального формата описываются ресурсы. Обобщенный (и упрощенный) формат записи о ресурсе выглядит следующим образом:

```
идентификатор_ресурса тип_ресурса данные_ресурса
```

Здесь идентификатор_ресурса — численное значение (для некоторых типов ресурсов допускается строковое значение), уникальное в пределах исполняемого модуля, в котором хранится ресурс; тип_ресурса — имя секции ресурсов в исполняемом файле (в секции хранятся ресурсы только одного типа); данные_ресурса — собственно содержимое ресурса.

В книге подробно будет рассмотрен формат записей лишь для некоторых типов ресурсов: строк, изображений, значков, аудио- и видеоданных. Стоит отметить, что для таких ресурсов, как строки, формат записи отличается от приведенного выше, но принцип перечисления отдельных строк остается тем же: запись дает сведения о принадлежности ресурса к типу строковых ресурсов, задает идентификатор и значение строки.

Ниже приведены примеры описания строковых ресурсов и ресурсов растровых изображений BMP (листинг 7.1).

**Листинг 7.1.** Пример файла ресурсов

```
//Строковые ресурсы
STRINGTABLE
BEGIN
  0001 "String1"
  0002 "String2"
END
//Ресурсы растровых изображений
0003      BITMAP "image.bmp"
res_image BITMAP "image1.bmp"
```

В приведенном фрагменте RC-файла два строковых ресурса и первый ресурс изображения получают целочисленные идентификаторы, а последний ресурс изображения — идентификатор-строку.

Если говорить честно, то у меня существуют большие сомнения по поводу того, что кто-то решится описывать ресурсы с помощью «голых» чисел, как это показано в листинге 7.1. Во всяком случае, строковые идентификаторы гораздо информативнее. Но и в этом случае не все гладко. Как, например, проверить, что при написании строкового идентификатора изображения или другого элемента не была допущена ошибка? До начала выполнения фрагмента программы никаких признаков ошибки проявляться не будет, зато потом... Конечно, последствия такой ошибки зависят от наличия в программе механизма обработки ошибок при обращениях к ресурсам, но не лучше ли регистрировать ошибки в идентификаторах ресурсов еще на этапе компиляции?

В реальных программах гораздо удобнее использовать директиву `#define`, которая, наряду с некоторыми другими директивами препроцессора C, поддерживается компилятором ресурсов. С помощью нее можно определить имена, которые позволяет проверить компилятор программы, — остается лишь вынести идентификаторы ресурсов в отдельный заголовочный файл и включить (с помощью директивы `#include`) определения идентификаторов как в файл ресурсов, так и в файлы программы.

Код измененного согласно изложенным соображениям файла ресурсов приведен в листинге 7.2.

**Листинг 7.2.** Пример файла ресурсов (идентификаторы в H-файле)

```
#include "resource.h"
//Строковые ресурсы
STRINGTABLE
BEGIN
  IDS_String1 "String1"
  IDS_String2 "String2"
END
//Ресурсы растровых изображений
IDB_Bitmap1  BITMAP "image.bmp"
IDB_Bitmap2  BITMAP "image1.bmp"
```

Текст заголовочного файла, используемого приведенным в листинге 7.2 RC-файлом, выглядит очень примитивно, но тем не менее для наглядности все же приведен в листинге 7.3.

**Листинг 7.3.** Определение идентификаторов ресурсов в H-файле

```
//Определения идентификаторов ресурсов
#define IDS_String1   1
#define IDS_String2   2
#define IDB_Bitmap1   3
#define IDB_Bitmap2   4
```

В завершение скажу несколько слов о том, как подключить файл ресурсов в проект. Для этого существуют как минимум два решения. Во-первых, можно использовать директиву `#pragma` (применяется для ресурсов форм), поместив в CPP-файл строку вроде этой:

```
#pragma resource "resource.rc"
```

Правда, скомпилированный бинарный файл ресурсов в этом случае придется создавать самостоятельно, подав описание, содержащееся в RC-файле, на вход компилятора ресурсов, например, консольной командой `bcc32.exe resource.rc`.

Во-вторых, можно просто добавить файл ресурсов в проект. При сборке проекта Borland C++ Builder замечательно идентифицирует файлы ресурсов и файлы с исходным кодом и, соответственно, запускает нужный компилятор. Именно такой способ, как наиболее простой и очевидный, используется в приводимых в этой главе примерах.

# Использование ресурсов в приложениях

Пожалуй, хватит отступлений. Пора переходить к практическим примерам использования ресурсов в приложениях Borland C++ Builder.

## Строковые ресурсы

Возможно, первое, что просится быть вынесенным в ресурсы приложения, — это используемые в нем строковые константы. Ими могут быть как тексты сообщений для пользователя, так и заголовки элементов управления форм приложения.

Вынесение строк в ресурсы действительно значительно усложняет код, но позволяет отделить логику работы программы от пользовательского интерфейса. К тому же благодаря использованию ресурсов заметно упрощается проверка и исправление грамматических ошибок и опечаток, которыми могут грешить программисты, да и перевести приложение, использующее ресурсы, на другой язык гораздо проще, чем приложение, их не использующее.

Пример, предлагаемый к рассмотрению в этом подразделе, весьма примитивен, но, думаю, и довольно показателен. В данном случае имеется форма приложения (рис. 7.1), надписи которой нужно вынести в ресурсы.

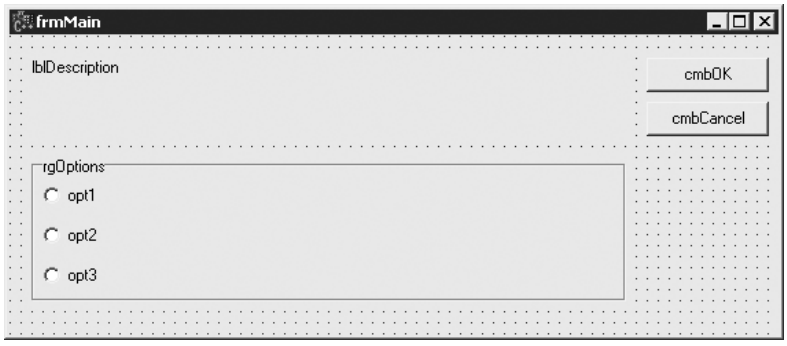


**Рис. 7.1.** Форма на этапе разработки

Всем компонентам формы в этом примере можно присвоить идентификаторы, отражающие роль компонента, а при создании формы выполнить приведенный в листинге 7.4 код.

**Листинг 7.4.** Загрузка надписей для компонентов формы из ресурсов

```
void __fastcall TfrmMain::FormCreate(TObject *Sender)
{
  //Установка заголовков формы и компонентов
  frmMain->Caption = AnsiString::LoadStr(IDS_frmMain);
  lblDescription->Caption = AnsiString::LoadStr(IDS_frmMain_
lblDescription);
  cmbOK->Caption = AnsiString::LoadStr(IDS_frmMain_cmbOK);
  cmbCancel->Caption = AnsiString::LoadStr(IDS_frmMain_
cmbCancel);
  rgOptions->Caption = AnsiString::LoadStr(IDS_frmMain_
rgOptions);
  rgOptions->Items->Strings[0] = AnsiString::LoadStr(IDS_
frmMain_opt1);
  rgOptions->Items->Strings[1] = AnsiString::LoadStr(IDS_
frmMain_opt2);
  rgOptions->Items->Strings[2] = AnsiString::LoadStr(IDS_
frmMain_opt3);
}
```

После этого форма обретет вид, показанный на рис. 7.2.

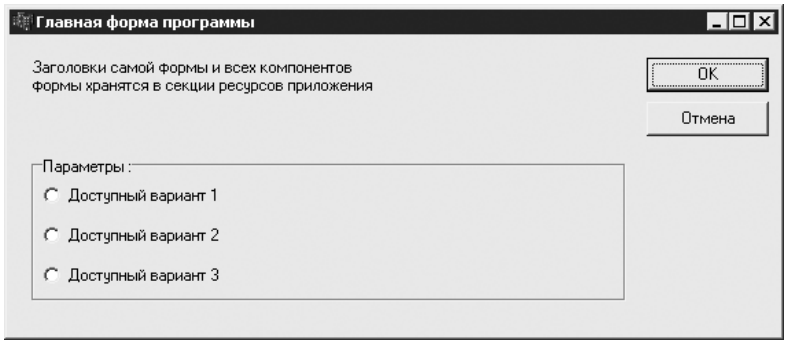


**Рис. 7.2.** Форма с загруженными из ресурса надписями

Обратите внимание, что в листинге 7.4 при загрузке текста из ресурсов используется не API-функция `LoadResource`, а статический метод `LoadStr` класса `AnsiString`, но можно, конечно, использовать и API-функцию, если это нужно.

Естественно, для работы примера пришлось написать и файл ресурсов, текст которого приведен в листинге 7.5. При его написании предполагалось, что строки, относящиеся к разным формам, для удобства будут группироваться в отдельные таблицы строк.

**Листинг 7.5.** Определение строковых ресурсов для приложения

```
#include "StringResource.h"
//Строки для формы frmMain
STRINGTABLE
BEGIN
  IDS_frmMain            "Главная форма программы"
  IDS_frmMain_lblDescription "Заголовки самой формы и всех
компонентов \
формы хранятся в секции ресурсов приложения"
  IDS_frmMain_cmbOK      "OK"
  IDS_frmMain_cmbCancel  "Отмена"
  IDS_frmMain_rgOptions  "Параметры :"
  IDS_frmMain_opt1       "Доступный вариант 1"
  IDS_frmMain_opt2       "Доступный вариант 2"
  IDS_frmMain_opt3       "Доступный вариант 3"
END
```

Приведенный в листинге 7.5 файл ресурсов использует идентификаторы, определенные в файле `StringResource.h`, текст которого показан в листинге 7.6.

**Листинг 7.6.** Файл с определением идентификаторов строковых ресурсов

```
#ifndef __STRING_RESOURCE_H__INCLUDED__
#define __STRING_RESOURCE_H__INCLUDED__
//Идентификаторы ресурсов для формы frmMain
#define IDS_frmMain                    1
#define IDS_frmMain_lblDescription     2
#define IDS_frmMain_cmbOK              3
#define IDS_frmMain_cmbCancel          4
#define IDS_frmMain_rgOptions          5
#define IDS_frmMain_opt1               6
#define IDS_frmMain_opt2               7
#define IDS_frmMain_opt3               8
#endif //__STRING_RESOURCE_H__INCLUDED__
```

Далее в примерах главы тексты H-файлов, содержащие описания идентификаторов ресурсов, приводиться не будут, поскольку, как мне кажется, принцип построения списка идентификаторов ресурсов уже полностью изложен.

Кстати, даже на таком примитивном примере, который был только что рассмотрен, можно показать не менее примитивный, но все же способ перевода приложения на другие языки.

Взгляните на листинг 7.7. В нем показано, как с помощью поддерживаемых компилятором ресурсов директив препроцессора можно задать различные варианты строк, зависящие от языка, для которого компилируется приложение.

**Листинг 7.7.** Строковые ресурсы для различных языков

```
#include "StringResource.h"
#if defined ( __LANG_RUS__ )
//Строки для русскоязычной версии приложения
STRINGTABLE
BEGIN
  IDS_frmMain            "Главная форма программы"
  IDS_frmMain_lblDescription "Заголовки самой формы и всех
компонентов \
формы хранятся в секции ресурсов приложения"
  IDS_frmMain_cmbOK      "OK"
  IDS_frmMain_cmbCancel  "Отмена"
  IDS_frmMain_rgOptions  "Параметры :"
  IDS_frmMain_opt1       "Доступный вариант 1"
  IDS_frmMain_opt2       "Доступный вариант 2"
  IDS_frmMain_opt3       "Доступный вариант 3"
END
#elif defined ( __LANG_ENG__ )
//Строки для англоязычной версии приложения
STRINGTABLE
BEGIN
  IDS_frmMain            "Main application form"
  IDS_frmMain_lblDescription "All component's captions, that
are at \
the form, stored in the resources section of the application"
  IDS_frmMain_cmbOK      "OK"
  IDS_frmMain_cmbCancel  "Cancel"
  IDS_frmMain_rgOptions  "Options :"
  IDS_frmMain_opt1       "Available choise 1"
  IDS_frmMain_opt2       "Available choise 2"
  IDS_frmMain_opt3       "Available choise 3"
END
#else
  #error Укажите язык, используемый для строковых ресурсов
#endif
```

В данном случае должен быть определен или макрос __LANG_RUS__, или макрос __LANG_ENG__. В зависимости от того, какой макрос определен, выбирается язык надписей приложения.

При таком подходе остается открытым вопрос: где же определять макрос, задающий используемый язык? Наверное, одним из наиболее простых и в то же время универсальных способов является задание нужного макроса в конфигурации проекта. Так, например, в среде разработки Borland C++ Builder 6 при компиляции всех файлов проекта макросы задаются в окне, показанном на рис. 7.3.

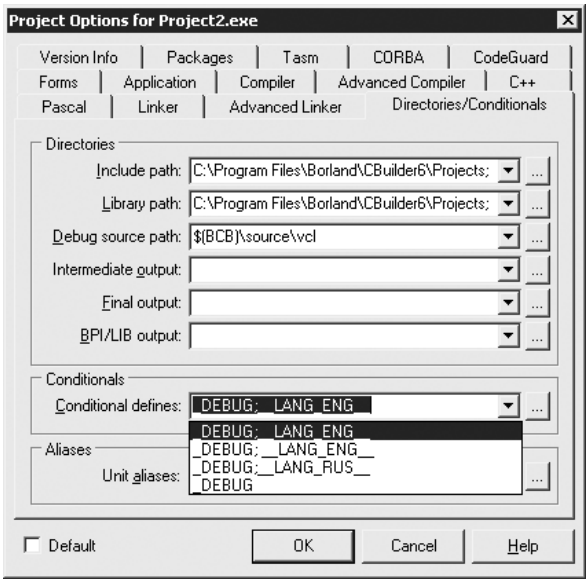


**Рис. 7.3.** Задание предопределенных макросов

На вкладке Directories/Conditionals в области настроек Conditions окна, показанного на рис. 7.3, отображается комбинация из ранее заданных макросов. Перечень самих доступных макросов задается в окне, вызываемом нажатием кнопки ... в этой группе.

Теперь остается лишь выбрать нужную конфигурацию макросов, и после сборки получится приложение на нужном нам языке.

## Изображения и значки в ресурсах

В качестве примера использования ресурсов в приложениях можно привести программу, которая основана на применении ресурсов, хранящих значки и растровые изображения в формате BMP. Так, на рис. 7.4 приведена форма приложения, значок и показанное в компоненте Image изображение которой загружены из ресурсов.

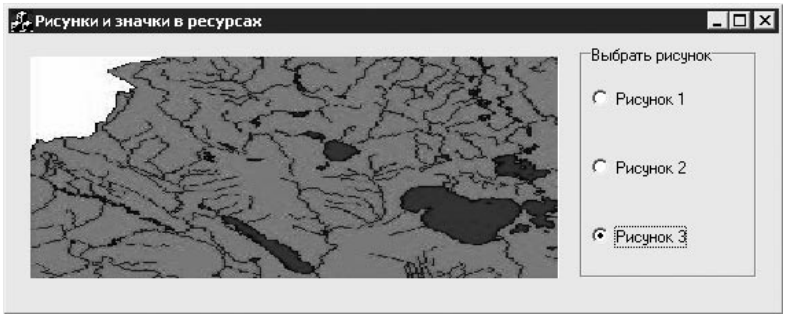


**Рис. 7.4.** Форма со значком и изображениями, полученными из ресурсов

В данном случае для описания ресурсов используется RC-файл, текст соодержания которого приведен в листинге 7.8.

**Листинг 7.8.** Описание ресурсов значка и растровых изображений

```
#include "ImageResource.h"
//Значок для главной формы приложения
IDI_frmMain          ICON     "images\\frmMain.ico"
//Рисунки, выводимые на главной форме приложения
IDB_frmMain_Picture1 BITMAP   "images\\frmMain_Picture1.bmp"
IDB_frmMain_Picture2 BITMAP   "images\\frmMain_Picture2.bmp"
IDB_frmMain_Picture3 BITMAP   "images\\frmMain_Picture3.bmp"
```

Описание способа загрузки значка, хранящегося в ресурсах, при создании формы представлено в листинге 7.9. К сожалению, класс `TIcon` не имеет методов (по крайней мере мне их не удалось найти), позволяющих загружать объекты из ресурсов, поэтому пришлось использовать API-функцию `LoadIcon`.

**Листинг 7.9.** Загрузка значка из ресурса

```
void __fastcall TfrmMain::FormCreate(TObject *Sender)
{
  //Загрузка из ресурсов значка формы
  HICON hIcon = (HICON)::LoadIcon(HInstance,
MAKEINTRESOURCE(IDI_frmMain));
  frmMain->Icon->Handle = hIcon;
}
```

Макрос `MAKEINTRESOURCE`, используемый в вызове функции `LoadIcon` в листинге 7.9, выполняет преобразование целочисленного значения идентификатора в указатель на символ. Команда определения макроса для ANSI-версии приложения имеет следующий вид:

```
#define MAKEINTRESOURCE(i) (LPSTR)((DWORD)((WORD)(i)))
```

Для загрузки растровых изображений из ресурсов класс `TBitmap` реализует метод `LoadFromResourceID`, избавляющий от необходимости работать с дескрипторами изображений (`HBITMAP`) и использовать макрос `MAKEINTRESOURCE`. Код реализации загрузки изображений в данном примере показан в листинге 7.10.

**Листинг 7.10.** Загрузка растровых изображений из ресурсов

```
void __fastcall TfrmMain::rgPictureClick(TObject *Sender)
{
  //Загрузка из ресурсов выбранного растрового изображения
  if ( rgPicture->ItemIndex == -1 ) return;
  UINT nImage = 0;
  switch ( rgPicture->ItemIndex )
  {
  case 0: nImage = IDB_frmMain_Picture1; break;
```

```
  case 1: nImage = IDB_frmMain_Picture2; break;
  case 2: nImage = IDB_frmMain_Picture3; break;
  }
  image->Picture->Bitmap->LoadFromResourceID((UINT)HInstance,
nImage);
}
```

На всякий случай напомню, что при необходимости вы всегда можете воспользоваться API-функцией `LoadBitmap`, которая аналогична функции `LoadIcon`.

## Видео- и аудиоданные в ресурсах

Теперь можно перейти к рассмотрению алгоритма работы с видео- и аудиоданными, хранящимися в секции ресурсов приложения. Еще в гл. 4 при описании компонента Animate, а также API-функции `PlaySound` я упоминал, что:

- компонент Animate имеет свойства `ResHandle` и `ResId`, которые обеспечивают загрузку видеоданных из ресурса с идентификатором `ResId` модуля с дескриптором `ResHandle`;
- функция `PlaySound` может иметь флаг `SND_RESOURCE`, означающий, что функции передается идентификатор ресурса, и параметр `hmod`, который в этом случае должен равняться дескриптору модуля, хранящего ресурс.

Ниже приведены конкретные примеры, которые, используя упомянутую функцию, демонстрируют способы реализации воспроизведения видео- и аудиоданных, хранящихся в ресурсах. Для начала взгляните на построение части RC-файла, ответственной за описание рассматриваемых ресурсов мультимедиа (листинг 7.11).

**Листинг 7.11.** Импорт видео- и аудиоданных в ресурсы

```
#include "MediaResource.h"
//Видеофайлы
IDR_Video1    AVI    "video\\video1.avi"
IDR_Video2    AVI    "video\\video2.avi"
//Аудиофайлы
IDR_Sound1    SOUND    "sound\\sound1.wav"
IDR_Sound2    SOUND    "sound\\sound2.wav"
```

В данном случае для корректной работы компонента Animate требуется, чтобы видеоданные находились в секции ресурсов с названием `AVI`. Для функции же `PlaySound` необходимо, чтобы передаваемые ей ресурсы находились в секции `SOUND`.

Код, реализующий использование компонента Animate для воспроизведения одного-двух видеофайлов из ресурсов, приведен в листинге 7.12.

**Листинг 7.12.** Воспроизведение видеоданных из ресурсов в компоненте Animate

```
void __fastcall TForm1::rgVideoClick(TObject *Sender)
{
```

```
  if ( rgVideo->ItemIndex == 0 )
  {
    animate->ResId = IDR_Video1;
    animate->Active = true;
  }
  else if ( rgVideo->ItemIndex == 1 )
  {
    animate->ResId = IDR_Video2;
    animate->Active = true;
  }
}
```

Обратите внимание, что в приведенном примере не используется свойство `ResHandle`, — это означает, что ресурсы загружаются из основного исполняемого модуля приложения (другими словами, из EXE-файла).

В свою очередь, нет ничего сложного в использовании функции `PlaySound` и для воспроизведения аудиоданных из ресурсов. Пример кода, демонстрирующего способ реализации проигрывания двух разных звуков, приведен в листинге 7.13.

**Листинг 7.13.** Воспроизведение аудиоданных из ресурсов

```
void __fastcall TForm1::cmbPlay1Click(TObject *Sender)
{
  ::PlaySound(MAKEINTRESOURCE(IDR_Sound1), HInstance,
              SND_SYNC | SND_NODEFAULT |  SND_RESOURCE);
}
void __fastcall TForm1::cmbPlay2Click(TObject *Sender)
{
  ::PlaySound(MAKEINTRESOURCE(IDR_Sound2), HInstance,
              SND_SYNC | SND_NODEFAULT |  SND_RESOURCE);
}
```

В листинге 7.13 используются ресурсы, хранящиеся в модуле EXE-файла приложения, дескриптор которого хранится в глобальной переменной `HInstance`.

## Бинарные ресурсы

Если по каким-то причинам вам не хватает возможностей, предоставляемых стандартными типами ресурсов Windows, ничто не мешает хранить данные в ресурсах в «сыром», то есть двоичном, виде.

Существуют как минимум два подхода для описания двоичных данных в RC-файле:

- первый подход подразумевает создание для каждого типа двоичного ресурса отдельной секции в файле ресурсов (по сути, именно этот подход был реализован в примерах с видео- и аудиоданными);
- второй подход предполагает хранение двоичных данных в специально предусмотренной для этого секции `RCDATA`.

В приведенном ниже примере используется второй подход (листинг 7.14).

**Листинг 7.14.** Описание двоичных ресурсов в RC-файле

```
#include "Resource.h"
//Данные элементов списка
IDR_List_Item_001 RCDATA
{
  1L,              //Целое число DWORD
  "Элемент 1\0",  //Строка длиной в 10 символов
}
IDR_List_Item_002 RCDATA
{
  2L,
  "Элемент 2\0",
}
IDR_List_Item_003 RCDATA
{
  3L,
  "Элемент 3\0",
}
//Аудиоданные, сохраненные как обычные бинарные данные
IDR_Sound1 RCDATA "sound1.wav"
IDR_Sound2 RCDATA "sound2.wav"
```

В первой части RC-файла, приведенного в листинге 7.14, представлены данные, которые будут загружены в список формы. Обратите внимание, что данные имеют фиксированный формат: число `DWORD` и строка длиной 10 символов (с учетом терминирующего нулевого символа, который нужно задавать для строк самостоятельно). Именно на предположении, что запись элемента списка имеет строго постоянный размер, и основана работа приведенного далее примера. В более сложных случаях, когда размер данных не может оставаться неизменным, может понадобиться предусмотреть собственные средства правильного разбора ресурсов по аналогии с составными файлами типа WAV, принцип устройства которых рассматривался еще в гл. 4 при генерации звуков.

Во второй части RC-файла описываются двоичные ресурсы, содержимое которых импортируется из файлов `sound1.wav` и `sound2.wav`. Эти записи имеют очень знакомый формат, не правда ли?

Теперь об использовании описанных в листинге 7.14 ресурсов. Код, отвечающий за формирование списка на основе данных, сохраненных в ресурсах, приведен в листинге 7.15.

**Листинг 7.15.** Формирование списка на основе двоичных данных из ресурсов

```
void *LoadResourceToMemory(UINT uResId)
{
  //Загружаем бинарный ресурс с переданным идентификатором
```

```
  //и передаем адрес данных ресурса в памяти
  HRSRC hResDesc = ::FindResource(NULL,
MAKEINTRESOURCE(uResId), RT_RCDATA);
  HGLOBAL hResGlobal = ::LoadResource(NULL, hResDesc);
  return ::LockResource(hResGlobal);
}
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  //Загрузим содержимое списка из ресурсов. Помещать данные
  //будем
  //в структуру, формат которой в точности повторяет формат
  //ресурсов
  //элементов списка
  struct _ListItem
  {
    DWORD Id;      //Идентификатор элемента
    char Text[10]; //Текст элемента
  }*pItem;
  for ( UINT uItemResId = First_List_Item;
        uItemResId <= Last_List_Item;
        uItemResId++ )
  {
    pItem = (_ListItem*)LoadResourceToMemory(uItemResId);
    List1->Items->Add(IntToStr(pItem->Id) + " - " + pItem->
Text);
  }
}
```

Вся хитрость кода, приведенного в листинге 7.15, состоит в том, что формат структуры `_ListItem` в точности соответствует формату записи ресурса с данными элементов списка. Всю работу по поиску ресурса и получению его адреса в оперативной памяти выполняет приведенная в начале листинга 7.15 функция `LoadResourceToMemory`.

Результат выполнения кода, приведенного в листинге 7.15, продемонстрирован на рис. 7.5.

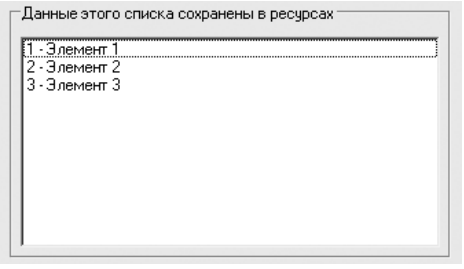


**Рис. 7.5.** Список, заполненный данными из ресурсов

Теперь, когда есть функция LoadResourceToMemory, ничего не стоит задействовать функцию PlaySound для воспроизведения аудиосодержимого, сохраненного как «сырой» (RAW) бинарный ресурс, а не как ресурс типа SOUND (листинг 7.16).

**Листинг 7.16.** Воспроизведение аудиоданных

```
void PlayResource(UINT uResId)
{
  //Загружаем бинарный ресурс с переданным идентификатором
  //и передаем его функции PlaySound
  void *pResData = LoadResourceToMemory(uResId);
  ::PlaySound((LPCSTR)pResData, NULL, SND_SYNC|SND_
NODEFAULT|SND_MEMORY);
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  //Воспроизведение первого звука
  PlayResource(IDR_Sound1);
}
```

В завершение стоит отметить некоторые удобные способы задания бинарных ресурсов, предоставляемые компилятором ресурсов. Думаю, вряд ли стоит заострять внимание на том, что хранение ресурсов допускается во внешних файлах, — это, собственно, не раз использовалось в примерах данной главы. Лучше остановиться на рассмотрении типов значений, из которых можно составлять содержимое ресурса, применяя конструкцию {}.

В листинге 7.14 задействовались значения двух типов: DWORD (беззнаковое 32-битное целое) и строковое (в данном случае это строка в кодировке ANSI). Еще раз заострю внимание на том, что при задании в ресурсах строк в том виде, в каком их принято использовать в C/C++, последний нулевой символ нужно задавать самостоятельно — терминирующий нулевой символ для строк не генерируется компилятором ресурсов.

При описании ресурсов также доступны значения типа WORD (беззнаковые 16-битные целые), шестнадцатеричные и восьмеричные целочисленные значения и строки в кодировке Unicode. Ниже приведен код, демонстрирующий использование всех перечисленных типов значений для задания ресурса:

```
IDR_Res_1
{
  123    //Значение типа WORD
  123L   //Значение типа DWORD
  0xAAAA     //Шестнадцатеричное значение (2 байта)
  0xBBBBBBBB //Шестнадцатеричное значение (4 байта)
  0o2222     //Восьмеричное значение (1 байт)
  0o11111111 //Восьмеричное значение (2 байта)
```

```
  "Строка в кодировке ANSI\0"
  L"Строка в кодировке UNICODE\0"
}
```

В завершение также стоит упомянуть о функции `SizeOfResource`, которая позволяет получать точный размер данных ресурса, а не полагаться на различные допущения и предположения:

```
DWORD SizeofResource(HMODULE hModule, HRSRC hResInfo);
```

Первый параметр функции — уже много раз упоминавшийся в главе дескриптор исполняемого модуля, из которого загружаются ресурсы. Второй параметр — описатель ресурса, возвращенный API-функцией поиска ресурса `FindResource` или `FindResourceEx` (см. функцию `LoadResourceToMemory` в листинге 7.15).

# Ресурсы других приложений

Думаю, некоторые читатели согласятся, что иногда, особенно на начальных стадиях освоения какого-либо нового предмета (в данном случае — использования ресурсов), бывает интересно заглянуть внутрь приложений других разработчиков. Приведенные в этом разделе примеры наглядно демонстрируют, каким образом можно заглянуть в секции ресурсов произвольных исполняемых файлов.

## Извлечение значков из EXE- и DLL-файлов

Для начала хотелось бы обратить ваше внимание на совсем несложную программу, позволяющую просматривать значки, хранящиеся в исполняемых файлах приложения. В этом примере косвенно используется работа с ресурсами (операции с ресурсами скрыты внутри используемой API-функции).

Значки, хранимые в исполняемых файлах, используются не только внутри приложения в качестве картинок, например для кнопок панелей инструментов или команд меню, значки из исполняемых файлов также отображаются Проводником Windows как значки файлов, ассоциированных с приложением.

Пускай имеются путь файла, а также два списка (`ImageList`) для больших и малых значков, тогда код функции, заполняющей списки значками, извлеченными из файла, может выглядеть следующим образом (листинг 7.17).

**Листинг 7.17.** Составление списков значков, хранимых в файле

```
void LoadIcons(const char *filename, TImageList *pLargeIcons,
               TImageList *pSmallIcons)
{
  //Загрузка всех значков, хранимых в файле
  //..определение количества значков
  UINT uImageCount = ExtractIconEx(filename, -1, NULL, NULL,
0);
```

```
  if ( uImageCount )
  {
    //..получение всех значков за один прием
    HICON *phLargeIcons = new HICON[uImageCount];
    HICON *phSmallIcons = new HICON[uImageCount];
    ExtractIconEx(filename, 0, phLargeIcons, phSmallIcons,
uImageCount);
    //..сохранение значков в списках
    for ( UINT i=0; i<uImageCount; i++ )
    {
      //...большой значок
      TIcon *pIcon = new TIcon;
      pIcon->Handle = phLargeIcons[i];
      pLargeIcons->AddIcon(pIcon);
      delete pIcon;
      //...маленький значок
      pIcon = new TIcon;
      pIcon->Handle = phSmallIcons[i];
      pSmallIcons->AddIcon(pIcon);
      delete pIcon;
    }
    //..не забываем освобождать память, выделенную под массивы
    delete []phLargeIcons;
    delete []phSmallIcons;
  }
}
```

Здесь для извлечения значков из файла используется функция `ExtractIconEx`. Прототип данной функции выглядит следующим образом:

```
UINT ExtractIconEx(LPCTSTR lpszFile, int nIconIndex, HICON
*phiconLarge,
                   HICON *phiconSmall, UINT nIcons);
```

Функция `ExtractIconEx` принимает следующие параметры:

- `lpszFile` — задает путь файла, из которого извлекаются значки;
- `nIconIndex` — указывает номер первого извлекаемого значка; нумерация начинается с нуля (если номер равен –1 и параметры `piconLarge` и `piconSmall` нулевые, то функция возвращает общее количество значков в файле);
- `piconLarge`, `piconSmall` — указатели на массивы `HICON`, используемые для помещения в них дескрипторов больших и малых значков соответственно;
- `nIcons` — указывает количество извлекаемых значков (по сути, может указывать на количество элементов в передаваемых в функцию массивах: лишние элементы заполнены не будут).

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

На рис. 7.6 показан внешний вид формы приложения после извлечения значков из файла `Explorer.exe`.

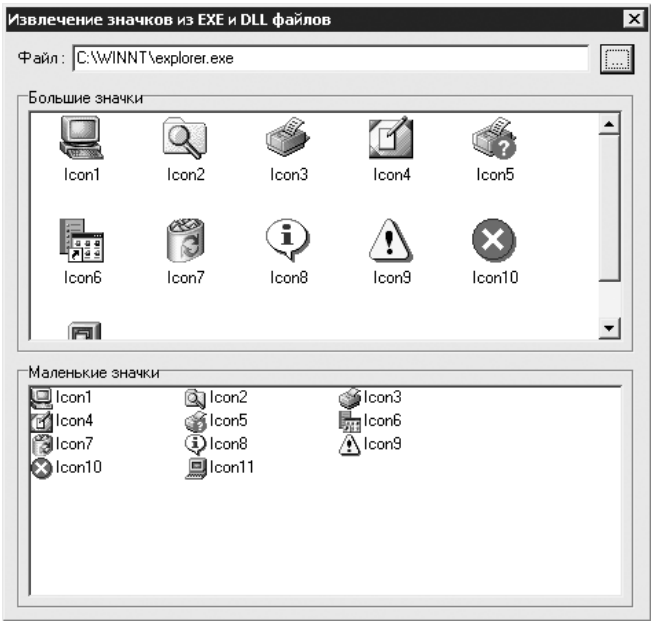


**Рис. 7.6.** Извлеченные из EXE-файла значки

Код обработчика нажатия кнопки выбора файла, инициирующего загрузку списка значков, приведен в листинге 7.18.

**Листинг 7.18.** Составление списков значков и их отображение

```
void __fastcall TForm1::cmbBrowseClick(TObject *Sender)
{
  //Выберем файл, из которого нужно извлечь значки
  if ( dlgOpen->Execute() )
  {
    txtFile->Text = dlgOpen->FileName;
    lvwLargeIcons->Clear();
    lvwSmallIcons->Clear();
    //Загрузка значков в ImageList
    largeImgList->Clear();
    smallImgList->Clear();
    LoadIcons(dlgOpen->FileName.c_str(), largeImgList, smal-
lImgList);
    //Создание элементов в ListView с большими и малыми
    //значками
```

```
    for ( long i=0; i<largeImgList->Count; i++ )
    {
      //..элемент списка с большими значками
      TListItem *pItem = lvwLargeIcons->Items->Add();
      pItem->Caption = "Icon" + IntToStr(i+1);
      pItem->ImageIndex = i;
      //..элемент списка с маленькими значками
      pItem = lvwSmallIcons->Items->Add();
      pItem->Caption = "Icon" + IntToStr(i+1);
      pItem->ImageIndex = i;
    }
  }
}
```

Здесь подразумевается, что элементы `lvwLargeIcons` и `lvwSmallIcons` компонента ListView используются для отображения больших и малых значков соответственно. Также на форме предусмотрены два элемента компонента ImageList: `largeImgList` и `smallImgList` — для хранения больших и малых значков соответственно.

С помощью окна Object Inspector список `largeImgList` назначен в качестве источника больших изображений (свойство `LargeImages`) для `lvwLargeIcons`, а список `smallImgList`, соответственно, — в качестве источника малых изображений (свойство `SmallImages`) для `lvwSmallIcons`.

## Программа для поиска значков

Интересным развитием только что рассмотренного примера программы извлечения значков из исполняемых файлов является его доработка таким образом, чтобы можно было составлять список всех значков, находящихся во всех исполняемых файлах в заданном дереве папок.

Провести такую доработку совсем несложно, так как практически все необходимые функции уже реализованы в предыдущих примерах. Сейчас самое время вспомнить о функции `ProcessFolderTree`, обходящей заданное дерево папок (она была рассмотрена в гл. 6). Имея в своем распоряжении эту функцию, остается лишь создать класс для обработки каждого найденного исполняемого файла. Код возможной реализации такого класса приведен в листинге 7.19.

**Листинг 7.19.** Класс для сбора значков, хранимых в файлах

```
class TIconCollectioner : public IFSNodeProcessor
{
  TImageList *m_pLargeIcons;
  TImageList *m_pSmallIcons;
  TListView *m_pIcons;
  long m_IconCounter;
public:
```

```
  TIconCollectioner(TImageList *pLargeIcons, TImageList
*pSmallIcons,
                    TListView *pIcons)
  {
    m_pLargeIcons = pLargeIcons;
    m_pSmallIcons = pSmallIcons;
    m_pIcons = pIcons;
    m_IconCounter = 0;
  }
  //IFSNodeProcessor
  virtual void ProcessFile(const char *fullPath)
  {
    if ( IsExecutableFile(fullPath) )
    {
      LoadIcons(fullPath);
    }
  }
  virtual void ProcessFolderBefore(const char *fullPath){}
  virtual void ProcessFolderAfter(const char *fullPath){}
private:
  bool IsExecutableFile(const char *filename)
  {
    //По расширению файла определяем, может ли файл содержать
    //значки
    String strExt = UpperCase(ExtractFileExt(filename));
    return strExt == ".EXE" || strExt == ".DLL";
  }
  void LoadIcons(const char *filename)
  {
    //Загрузка всех значков, хранимых в файле
    //..определение количества значков
    UINT uImageCount = ExtractIconEx(filename, -1, NULL, NULL,
0);
    if ( uImageCount )
    {
      //..получение всех значков за один прием
      HICON *phLargeIcons = new HICON[uImageCount];
      HICON *phSmallIcons = new HICON[uImageCount];
      ExtractIconEx(filename, 0, phLargeIcons, phSmallIcons,
uImageCount);
      for ( UINT i=0; i<uImageCount; i++ )
      {
        //..сохранение большого значка
        TIcon *pIcon = new TIcon;
        pIcon->Handle = phLargeIcons[i];
        m_pLargeIcons->AddIcon(pIcon);
```

```
        delete pIcon;
        //...сохранение маленького значка
        pIcon = new TIcon;
        pIcon->Handle = phSmallIcons[i];
        m_pSmallIcons->AddIcon(pIcon);
        delete pIcon;
        //..добавление элемента в ListView
        TListItem *pItem = m_pIcons->Items->Add();
        pItem->Caption = "Icon" + IntToStr(m_IconCounter++);
        pItem->ImageIndex = i;
      }
      //..не забываем освобождать память, выделенную под
      //массивы
      delete []phLargeIcons;
      delete []phSmallIcons;
    }
  }
};
```

Думаю, принцип работы приведенного в листинге 7.19 класса прост и очевиден: объект класса принимает указатели на компоненты ImageList и заполняет эти компоненты значками по мере их нахождения в исполняемых файлах. При нахождении каждого значка создается элемент в списке ListView (указатель на который требуется также и конструктором класса), с которым связываются извлеченные большой и малый значки.

Пример результата работы программы показан на рис. 7.7.

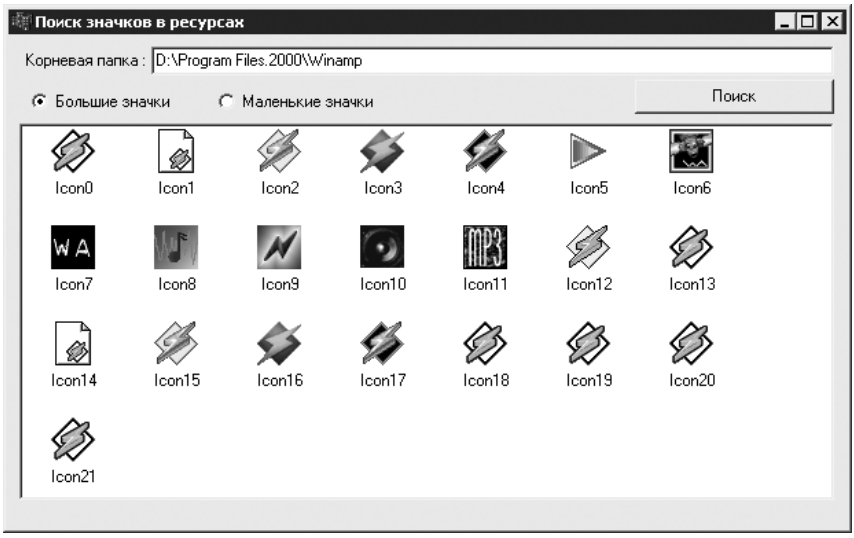


**Рис. 7.7.** Результат поиска значков

Как видно из рис. 7.7, программа, приведенная в данном примере, позволяет отображать только отдельно большие или маленькие значки. Думаю, реализация переключения между режимами работы компонента ListView ни у кого трудностей не вызовет, а вот код реализации запуска процесса поиска значков на всякий случай приведен в листинге 7.20.

**Листинг 7.20.** Запуск процесса поиска значков

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  if ( !DirectoryExists(txtRootFolder->Text) )
  {
    Application->MessageBox("Введите путь существующей
папки",
                            Application->Title.c_str(), MB_
ICONINFORMATION);
    return;
  }
  //Очистка перед поиском
  lvwIcons->Clear();
  largeIcons->Clear();
  smallIcons->Clear();
  //Поиск значков в ресурсах
  Screen->Cursor = crHourGlass;
  ProcessFolderTree(txtRootFolder->Text.c_str(),
                    &TIconCollectioner(largeIcons, smallIcons,
lvwIcons));
  Screen->Cursor = crDefault;
}
```

Последнее, что хотелось бы сказать о приведенной в этом примере программе, так это то, что не стоит запускать ее для анализа больших деревьев папок с большим количеством исполняемых файлов и, соответственно, большим количеством значков. При написании этого примера производительность программы считалась не столь важной — основной задачей являлось добиться максимальной наглядности и простоты примера, поэтому я ограничился штатными возможностями компонентов ListView и ImageList.

# Глава 8

## Системная информация и реестр Windows

- Системная информация
- Системное время
- Реестр

Возникала ли когда-нибудь у вас необходимость программно определить текущее состояние компьютера или получить какие-нибудь сведения об операционной системе? Часто при столкновении с необходимостью решения подобной задачи можно только удивляться, как близко, практически «под носом» у программиста, находятся средства получения системной информации и как сложно иногда об этих средствах узнать. Речь идет о средствах, которые всегда доступны при программировании для Windows, — функции Windows API.

В данной главе будут рассмотрены некоторые способы, с помощью которых можно получить информацию, касающуюся операционной системы. Они могут пригодиться, например, если вы используете в своих приложениях возможности, различающиеся на 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 по программированию в таких средах, как Borland C++ Builder или Borland Delphi.

В примерах представленной вашему вниманию главы, кроме способов получения информации о самой Windows и некотором оборудовании компьютера, рассмотрена работа с системным реестром Windows — своего рода базой данных, в которой хранится много полезной информации: от параметров операционной системы, настроек приложений до сведений о работе компьютера, поступающих и изменяющихся в реальном времени.

# Системная информация

Сначала будут рассмотрены несколько несложных примеров, позволяющих получить информацию об операционной системе, установленном на компьютере оборудовании и некоторые сведения реального времени, такие как загрузка памяти компьютера, состояние питания и др.

## Версия операционной системы

Получение сведений об операционной системе хотя и не является повседневной необходимостью, но все же в некоторых специфичных случаях может потребоваться: например, когда программа работает по-разному с различными установленными обновлениями Windows либо вы самостоятельно пишете инсталлятор, который способен устанавливать версии программы, скомпилированные под Windows ME (95, 98) и Windows NT (2000, XP).

Одним из способов узнать версию Windows является использование API-функции `GetVersionEx`. Данная функция принимает в качестве параметра структуру `OSVERSIONINFO` (или `OSVERSIONINFOEX`, но об этом позже), заполняет поля этой структуры и возвращает ненулевое значение в случае успешного выполнения задачи.

Код объявления ANSI-версии структуры OSVERSIONINFO в библиотеке Borland C++ Builder выглядит следующим образом:

```
typedef struct _OSVERSIONINFO
{
    DWORD dwOSVersionInfoSize;   //Размер структуры
                                 //OSVERSIONINFO
    DWORD dwMajorVersion;        //Старший номер версии ОС
    DWORD dwMinorVersion;        //Младший номер версии ОС
    DWORD dwBuildNumber;         //Порядковый номер сборки
    DWORD dwPlatformId;          //Идентификатор платформы
                                 //Windows
    CHAR   szCSDVersion[ 128 ]; //Строка с дополнительной
                                 //информацией
} OSVERSIONINFO;
```

Не стоит сейчас вдаваться в подробное описание возможных значений полей этой структуры: практически все они будут понятны из приведенного далее примера — напомню лишь не забывать заполнять поле dwOSVersionInfoSize перед вызовом функции GetVersionEx.

Итак, код примера обработки данных, помещаемых в структуру OSVERSIONINFO, приведен в листинге 8.1. При загрузке формы элемент управления lvwVerInfo компонента ListView заполняется сведениями о версии системы, представленными в удобной для чтения форме.

**Листинг 8.1.** Получение и отображение сведений о Windows

```
void __fastcall TfrmWinMEVersion::FormCreate(TObject *Sender)
{
  //Получаем информацию о версии операционной системы
  OSVERSIONINFO info;
  ZeroMemory(&info, sizeof(info));
  info.dwOSVersionInfoSize = sizeof(info);
  ::GetVersionEx(&info);
  //Заполняем список информацией об операционной системе
  //..версия операционной системы
  TListItem *item = lvwVerInfo->Items->Add();
  item->Caption = "Версия системы";
  item->SubItems->Insert(0, IntToStr(info.dwMajorVersion) +
'.' +
                         IntToStr(info.dwMinorVersion));
  //..номер сборки
  item = lvwVerInfo->Items->Add();
  item->Caption = "Сборка";
  item->SubItems->Insert(0, IntToStr(info.dwBuildNumber));
  //..платформа
```

```
  item = lvwVerInfo->Items->Add();
  item->Caption = "Платформа";
  switch ( info.dwPlatformId )
  {
  case VER_PLATFORM_WIN32s:
    //Эмуляция Win32 или Win16
    item->SubItems->Insert(0, "Win16");
    break;
  case VER_PLATFORM_WIN32_WINDOWS:
    //"Классическая" Win32: 95, 98 или ME
    item->SubItems->Insert(0, "Win32");
    break;
  case VER_PLATFORM_WIN32_NT:
    //Ядро NT
    item->SubItems->Insert(0, "WinNT");
    break;
  }
  //..дополнительная информация (например, ServicePack)
  item = lvwVerInfo->Items->Add();
  item->Caption = "Дополнительные сведения";
  item->SubItems->Insert(0, info.szCSDVersion);
}
```

Возможный результат работы программы (для Windows XP SP1) показан на рис. 8.1.

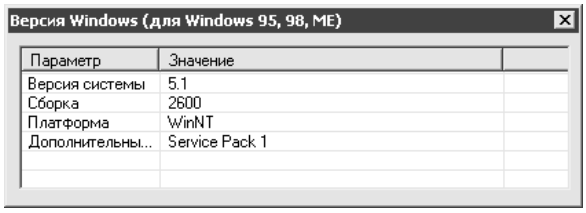


**Рис. 8.1.** Информация о версии Windows

В качестве параметра в функцию `GetVersionEx` вместо структуры `OSVERSION-INFO` может передаваться структура `OSVERSIONINFOEX`. Код объявления ANSI-версии этой структуры с комментариями к тем полям, которых нет в описанной выше структуре, имеет следующий вид:

```
typedef struct _OSVERSIONINFOEXA {
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    CHAR   szCSDVersion[ 128 ];
```

```
    WORD   wServicePackMajor; //Старшая цифра версии
                              //ServicePack
    WORD   wServicePackMinor; //Младшая цифра версии
                              //ServicePack
    WORD   wSuiteMask;        //Комплектация системы
    BYTE  wProductType;       //Дополнительная информация
                              //об операционной системе
    BYTE  wReserved;
} OSVERSIONINFOEX;
```

Дополнительные (по сравнению со структурой `OSVERSIONINFO`) поля структуры могут заполнять операционные системы Windows NT 4.0 SP6 и более поздние версии Windows NT (в том числе 2000 и XP).

Значение поля `wSuiteMask` (является битовой маской) может быть представлено комбинацией значений следующих констант:

- `VER_SUITE_BACKOFFICE` — означает, что на компьютере установлен Microsoft BackOffice;
- `VER_SUITE_DATACENTER` — указывает, что на компьютере установлен Microsoft Data Center;
- `VER_SUITE_ENTERPRISE` — означает, что на компьютере установлена операционная система Windows 2000 Advanced Server;
- `VER_SUITE_SMALLBUSINESS` — указывает, что на компьютере установлен Microsoft Windows Small Business Server;
- `VER_SUITE_SMALLBUSINESS_RESTRICTED` — означает, что на компьютере установлена ограниченная версия Microsoft Windows Small Business Server;
- `VER_SUITE_TERMINAL` — определяет, что на компьютере установлены терминальные службы;
- `VER_SUITE_PERSONAL` — означает, что на компьютере установлена персональная версия операционной системы (типичный набор функций; меньше, чем в Professional).

Значение поля `wProductType` может быть задано одним из приведенных ниже значений (определяет тип сетевой операционной системы и, соответственно, роль, которую компьютер с данной операционной системой может выполнять при подключении в сети):

- `VER_NT_WORKSTATION` — рабочая станция;
- `VER_NT_DOMAIN_CONTROLLER` — контроллер домена;
- `VER_NT_SERVER` — сервер.

Код определения полной информации о версии операционной системы для случая Windows на платформе NT (выше NT 4.0 SP6) может выглядеть следующим образом (листинг 8.2).

**Листинг 8.2.** Определение версии операционной системы (NT, 2000, XP)

```
void __fastcall TfrmWinNTVersion::FormCreate(TObject *Sender)
{
  //Получаем информацию о версии операционной системы
  OSVERSIONINFOEX info;
  ZeroMemory(&info, sizeof(info));
  info.dwOSVersionInfoSize = sizeof(info);
  ::GetVersionEx((OSVERSIONINFO*)&info);
  //Заполняем список информацией об операционной системе
  //..версия операционной системы
  TListItem *item = lvwVerInfo->Items->Add();
  item->Caption = "Версия системы";
  item->SubItems->Insert(0, IntToStr(info.dwMajorVersion) +
                                                    '.' +
                         IntToStr(info.dwMinorVersion));
  //..номер сборки
  item = lvwVerInfo->Items->Add();
  item->Caption = "Сборка";
  item->SubItems->Insert(0, IntToStr(info.dwBuildNumber));
  //..платформа
  item = lvwVerInfo->Items->Add();
  item->Caption = "Платформа";
  switch ( info.dwPlatformId )
  {
  case VER_PLATFORM_WIN32s:
    //Эмуляция Win32 или Win16
    item->SubItems->Insert(0, "Win16");
    break;
  case VER_PLATFORM_WIN32_WINDOWS:
    //"Классическая" Win32: 95, 98 или ME
    item->SubItems->Insert(0, "Win32");
    break;
  case VER_PLATFORM_WIN32_NT:
    //Ядро NT
    item->SubItems->Insert(0, "WinNT");
    break;
  }
  //..дополнительная информация (например, ServicePack)
  item = lvwVerInfo->Items->Add();
  item->Caption = "Дополнительные сведения";
  item->SubItems->Insert(0, info.szCSDVersion);
  //..версия ServicePack
  item = lvwVerInfo->Items->Add();
  item->Caption = "Версия ServicePack";
  item->SubItems->Insert(0, IntToStr(info.wServicePackMajor)
+ '.' +
```

```
                                IntToStr(info.wServicePackMinor));
  //..комплектация операционной системы
  String suite;
  if ( info.wSuiteMask & VER_SUITE_BACKOFFICE )
    suite += "[Установлен BackOffice] ";
  if ( info.wSuiteMask & VER_SUITE_DATACENTER )
    suite += "[Microsoft Data Center] ";
  if ( info.wSuiteMask & VER_SUITE_ENTERPRISE )
    suite += "[Windows 2000 Advanced Server] ";
  if ( info.wSuiteMask & VER_SUITE_SMALLBUSINESS )
    suite += "[Small Business Server] ";
  if ( info.wSuiteMask & VER_SUITE_SMALLBUSINESS_RESTRICTED )
    suite += "[Small Business Server, ограниченная версия] ";
  if ( info.wSuiteMask & VER_SUITE_TERMINAL )
    suite += "[Terminal Service] ";
  if ( info.wSuiteMask & VER_SUITE_PERSONAL )
   suite += "[Workstation Personal (не Professional)] ";
  item = lvwVerInfo->Items->Add();
  item->Caption = "Комплектация";
  item->SubItems->Add(suite);
  //..дополнительные сведения
  String additional;
  switch ( info.wProductType )
  {
  case VER_NT_WORKSTATION:
    additional = "[Рабочая станция] ";
    break;
  case VER_NT_DOMAIN_CONTROLLER:
    additional = "[Контроллер домена] ";
    break;
  case VER_NT_SERVER:
    additional = "[Сервер] ";
    break;
  }
  item = lvwVerInfo->Items->Add();
  item->Caption = "Дополнительно";
  item->SubItems->Add(additional);
}
```

## Имя компьютера

Следующий простой пример (листинг 8.3) демонстрирует способ определения сетевого имени компьютера. Приведенная здесь функция `GetComputerName` скрывает нюансы работы со строковым буфером, который нужно передавать в API-функцию `GetComputerName`.

**Листинг 8.3.** Определение сетевого имени компьютера

```
String GetComputerName()
{
  //Получение имени компьютера
  char buffer[MAX_COMPUTERNAME_LENGTH+1] = "";
  unsigned long size = MAX_COMPUTERNAME_LENGTH+1;
  ::GetComputerName(buffer, &size);
  return String(buffer);
}
```

## Имя пользователя

Определить имя учетной записи пользователя, из-под которой запущена программа (точнее, вызывающий функцию поток), можно с использованием функции GetUserName, код которой представлен в листинге 8.4.

**Листинг 8.4.** Определение имени пользователя

```
String GetUserName()
{
  //Получение имени пользователя
  char buffer[100] = "";
  unsigned long size = 100;
  ::GetUserName(buffer, &size);
  return String(buffer);
}
```

Чаще всего функция GetUserName определяет пользователя, выполнившего вход в систему, но если приложение было запущенно от имени другого пользователя (например, Admin, несмотря на то что вход был осуществлен через учетную запись User), то, соответственно, определено будет имя пользователя Admin.

## Состояние системы питания компьютера

Следующий пример может быть интересным для обладателей компьютеров, подключенных к резервным источникам питания (батарея в ноутбуке или источник бесперебойного питания).

Для определения состояния системы питания компьютера предназначена API-функция GetSystemPowerStatus. Данная функция заполняет структуру TSystemPowerStatus и возвращает ненулевое значение в случае успешного выполнения задачи. Упомянутая структура имеет следующий вид:

```
typedef struct _SYSTEM_POWER_STATUS {
    BYTE ACLineStatus;          //Подключение к сети переменного
                                //тока
    BYTE BatteryFlag;           //Состояние батареи (уровень
                                //заряда и пр.)
```

```
    BYTE BatteryLifePercent;    //Оставшийся ресурс батареи
                                //(в процентах)
    BYTE Reserved1;
    DWORD BatteryLifeTime;      //Оставшееся время (в сек.)
                                //работы батареи
    DWORD BatteryFullLifeTime; //Полное время (в сек.) работы
                                //батареи
}TSystemPowerStatus;
```

Если значения приведенных полей BatteryLifePercent, BatteryLifeTime и BatteryFullLifeTime вполне понятны, то способ извлечения информации из полей ACLineStatus и BatteryFlag можно подсмотреть в листинге 8.5.

**Листинг 8.5.** Определение состояния системы питания

```
void TfrmPowerStatus::ShowPowerStatus()
{
  lvwPowerStatus->Clear();
  //Получаем информацию о состоянии питания
  TSystemPowerStatus status;
  ZeroMemory(&status, sizeof(status));
  ::GetSystemPowerStatus(&status);
  //Заполняем список информации о состоянии питания
  //..подключение к сети
  switch ( status.ACLineStatus )
  {
  case 0: AddParam("Подключение в сети", "Отключен"); break;
  case 1: AddParam("Подключение в сети", "Подключен"); break;
  default: AddParam("Подключение в сети", "Неизвестно");
  }
  //..заряд батареи (битовая маска)
  String batFlags;
  if ( status.BatteryFlag & 1 ) batFlags = "Высокий ";
  if ( status.BatteryFlag & 2 ) batFlags += "Низкий ";
  if ( status.BatteryFlag & 4 ) batFlags += "Критический ";
  if ( status.BatteryFlag & 8 ) batFlags += "(Идет зарядка)";
  if ( status.BatteryFlag & 128 ) batFlags += "Батарея не
установлена";
  if ( status.BatteryFlag == 255 ) batFlags += "Неизвестно";
  AddParam("Заряд батареи", batFlags);
  //..числовые характеристики батареи
  if ( status.BatteryLifePercent != 255 )
    AddParam("Остаток заряда батареи", IntToStr(status.
BatteryLifePercent));
  else
    AddParam("Остаток заряда батареи", "Неизвестно");
  if ( status.BatteryLifeTime != (DWORD)-1 )
```

```
    AddParam("Время работы батареи (остаток, сек.)",
             IntToStr(status.BatteryLifeTime));
  else
    AddParam("Время работы батареи (остаток, сек.)",
"Неизвестно");
  if ( status.BatteryFullLifeTime != (DWORD)-1 )
    AddParam("Полное время работы батареи, сек.",
             IntToStr(status.BatteryFullLifeTime));
  else
    AddParam("Полное время работы батареи, сек.",
"Неизвестно");
}
```

В листинге 8.5 для отображения каждого параметра системы питания вызывается функция `AddParam`, добавляющая в элемент управления формы название параметра и его значение. Этим элементом управления может быть, например, компонент ListView. Возможный результат выполнения функции `ShowPowerStatus` показан на рис. 8.2.

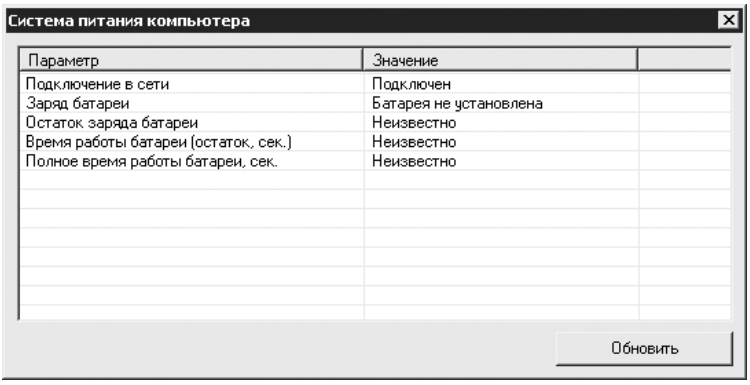


**Рис. 8.2.** Собранная информация о системе питания

Скажу еще несколько слов о сфере реального применения рассмотренного примера: он может пригодиться в случае, если ваше приложение оперирует большим объемом важных данных, на сохранение которых требуется длительное время и потеря которых может вызвать большие неприятности. Так, при обнаружении низкого уровня заряда батареи приложение может сохранить (точнее, длительное время сохранять) данные на диск, например, до тех пор, пока питание вновь не будет восстановлено, а заряд батареи не достигнет требуемого значения.

## Состояние памяти компьютера

Получить сводку о текущем состоянии памяти компьютера также не представляет сложной задачи. Недаром эту информацию выводят в окне О программе многие приложения, в том числе и Блокнот.

Итак, получить информацию о состоянии памяти компьютера можно с помощью API-функции `GlobalMemoryStatus`. Она принимает в качестве параметра структуру `TMemoryStatus`, заполняет ее поля значениями и возвращает отличное от нуля число в случае успешного выполнения задачи. Объявление структуры `TMemoryStatus` с комментариями роли ее полей имеет следующий вид:

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength;          //Размер структуры (байт)
    DWORD dwMemoryLoad;      //Процент загрузки физической
                             //памяти
    SIZE_T dwTotalPhys;      //Полный объем физической памяти
    SIZE_T dwAvailPhys;      //Объем свободной оперативной
                             //памяти
    SIZE_T dwTotalPageFile; //Полный объем файла подкачки
    SIZE_T dwAvailPageFile; //Объем свободного пространства
                             //в файле подкачки
    SIZE_T dwTotalVirtual;  //Полный объем виртуальной памяти
    SIZE_T dwAvailVirtual;  //Объем свободной виртуальной
                             //памяти
} TMemoryStatus;
```

Два последних поля структуры `TMemoryStatus` относятся к вызывающему функцию `GlobalMemoryStatus` приложению. Их смысл (для тех, кто не знает) раскрывается ниже. Пример использования функции `GlobalMemoryStatus` показан в листинге 8.6.

**Листинг 8.6.** Определение состояния памяти

```
void TfrmMemoryUsage::ShowMemoryUsage()
{
  //Получение информации о загрузке памяти
  TMemoryStatus memStat;
  ZeroMemory(&memStat, sizeof(memStat));
  memStat.dwLength = sizeof(memStat);
  ::GlobalMemoryStatus(&memStat);
  //Заполнение формы
  long percent;
  //..использование оперативной памяти
  percent = 100.0 - 100.0 * memStat.dwAvailPhys / memStat.
dwTotalPhys;
  pbPhMem->Position = percent;
  lblPhMemPercent->Caption = IntToStr(percent) + "%";
  txtPhMemAll->Text = IntToStr(memStat.dwTotalPhys / 1024);
  txtPhMemAvail->Text = IntToStr(memStat.dwAvailPhys / 1024);
  //..использование файла подкачки
  percent = 100.0 - 100.0*memStat.dwAvailPageFile / memStat.
dwTotalPageFile;
```

```
  pbPageFile->Position = percent;
  lblPageFilePercent->Caption = IntToStr(percent) + "%";
  txtPageFileAll->Text = IntToStr(memStat.dwTotalPageFile /
1024);
  txtPageFileAvail->Text = IntToStr(memStat.dwAvailPageFile /
1024);
  //..использование виртуальной памяти
  percent = 100.0 - 100.0 * memStat.dwAvailVirtual / memStat.
dwTotalVirtual;
  pbVMem->Position = percent;
  lblVMemPercent->Caption = IntToStr(percent) + "%";
  txtVMemAll->Text = IntToStr(memStat.dwTotalVirtual / 1024);
  txtVMemAvail->Text = IntToStr(memStat.dwAvailVirtual /
1024);
}
```

Внешний вид формы, элементы управления которой заполняются в результате выполнения кода, представленного в листинге 8.6, показан на рис. 8.3.

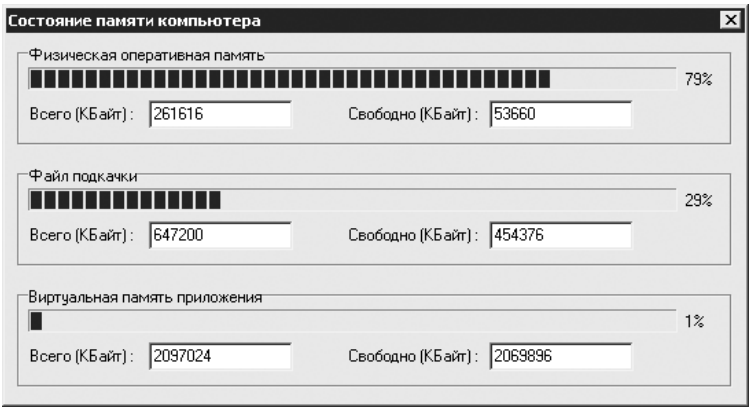


**Рис. 8.3.** Определение состояния памяти компьютера

Напоследок поясню (несколько упрощенно) результаты, выводимые в текстовых полях формы, для тех, кто немного не в курсе, как организовано управление памятью в Windows.

Каждому процессу Windows предоставляет адресное пространство (правда, не настоящее, а виртуальное) размером чуть меньше 2 Гбайт (в 64-битных и серверных версиях Windows значительно больше). В отличие от 16-битных предшественниц, в 32-битных Windows адресные пространства различных процессов являются закрытыми: приложение использует память (точнее, младшие 2 Гбайт адресного пространства) от собственного имени и не может без дополнительных привилегий манипулировать данными других процессов. Именно динамику использования приложением представляемого ему адресного пространства и отражают значения двух последних полей структуры `TMemoryStatus` (нижняя группа элементов управления на рис. 8.3).

Механизм выделения виртуальной памяти является достаточно удобной надстройкой, скрывающей ограниченность аппаратных ресурсов компьютера. Ограниченный объем оперативной памяти компенсируется использованием физического места на диске (файла подкачки, страничного файла). В этот файл записываются для временного хранения неиспользуемые страницы памяти (блоки данных по несколько килобайт), давая возможность помещать другие, нужные приложению, данные в оперативную память.

Теперь можно вернуться к форме, показанной на рис. 8.3. Группа элементов управления в области настроек Физическая оперативная память показывает полный и свободный объем реально установленной на компьютере оперативной памяти (за вычетом памяти, используемой для системных нужд). Использование этого вида памяти также отражает индикатор ProgressBar на форме. По аналогичному принципу показывается динамика использования файла подкачки (область Файл подкачки) и виртуальной памяти (область Виртуальная память приложения).

# Системное время

Этот раздел посвящен способам отнюдь не простого получения информации о текущем времени или дате (благо эти функции можно найти и в библиотеке Borland C++ Builder). Здесь будет раскрыта несколько более интересная тема — использование системных средств измерения малых промежутков времени.

Все рассмотренные далее способы определения времени основаны на подсчете количества срабатываний таймера. Для сохранения показаний таймера система использует соответствующие счетчики, а для определения временного интервала вычисляется разница между показаниями счетчика в начале и в конце промежутка времени, и если период таймера не соответствует требуемой единице измерения (например, миллисекунды), полученное значение делится на частоту таймера, выраженную в соответствующих единицах.

## Время работы операционной системы

В момент своего запуска операционная система Windows запускает специальный счетчик, показывающий количество срабатываний, произошедших с момента запуска системы.

Этот системный счетчик можно использовать как для определения времени работы системы, так и для измерения временных интервалов. Для доступа к упомянутому счетчику можно использовать API-функцию `GetTickCount`. Эта функция не имеет параметров и возвращает целочисленное 32-битное значение.

Код приведенной в листинге 8.7 функции `GetSystemWorkTime` наглядно демонстрирует способ использования описанного счетчика для определения времени работы системы, выраженного в часах, минутах и секундах.

**Листинг 8.7.** Определение времени работы системы

```
String GetSystemWorkTime()
{
  //Получаем количество миллисекунд с момента старта системы
  //и сразу переводим в секунды
  DWORD ticks = ::GetTickCount()  / 1000;
  //Получаем количество часов, минут, секунд
  int hh = ticks / 3600;
  ticks -= hh * 3600;
  int mm = ticks / 60;
  int ss = ticks - mm * 60;
  return IntToStr(hh) + ":" + IntToStr(mm) + ":" +
IntToStr(ss);
}
```

Из-за относительно малой разрядности значение счетчика обнуляется приблизительно каждые 49,7 суток, что следует учитывать при измерении длительных интервалов или при запуске измерения времени после длительной работы системы (например, 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 срабатываниями этого таймера может быть значительно меньше 1 мс, что позволяет производить очень точные измерения. Для сохранения количества срабатываний аппаратного таймера используется 64-битный счетчик.

Код получения значения счетчика аппаратного таймера приведен в листинге 8.8. Частота, возвращаемая функцией `hwTimerGetCounter`, измеряется в герцах, то есть означает количество срабатываний таймера за 1 с.

**Листинг 8.8.** Получение значения счетчика аппаратного таймера

```
__int64 hwTimerGetCounter()
{
  LARGE_INTEGER freq;
  return ::QueryPerformanceCounter(&freq) ? freq.QuadPart :
0;
}
```

Чтобы перевести количество срабатываний аппаратного таймера в привычные единицы измерения, нужно знать частоту аппаратного таймера. Для этого может использоваться функция, код которой приведен в листинге 8.9.

**Листинг 8.9.** Определение частоты аппаратного таймера

```
__int64 hwTimerGetFreq()
{
```

```
  LARGE_INTEGER freq;
  return ::QueryPerformanceFrequency(&freq) ? freq.QuadPart :
0;
}
```

Если известна разность (`count`) между значениями счетчика в начале и в конце измерения длительности промежутка времени, то перевести это значение в секунды можно следующим образом:

```
time = count / hwTimerGetFreq();
```

Результат определения характеристик аппаратного таймера показан на рис. 8.4.

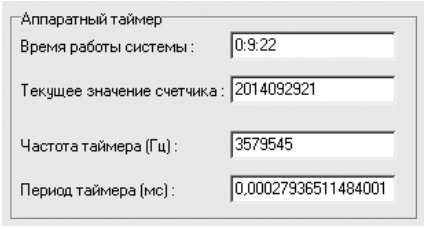


**Рис. 8.4.** Характеристики аппаратного таймера

Заполнение приведенных на рис. 8.4 текстовых полей производится чрезвычайно просто, а поэтому выполняющий это код в тексте книги не приводится. При желании вы можете найти его в демонстрационном проекте.

## Мультимедийный таймер

Обратите внимание на еще один способ измерения времени, основанный на использовании так называемого мультимедийного таймера. Удобство использования этого таймера определятся возможностью указания его точности. Группа API-функций работы с мультимедийным таймером позволяет не только измерять временные интервалы, но и создавать программные таймеры (аналогичные компоненту Timer), срабатывающие через предельно короткие промежутки времени.

Для получения текущего значения счетчика мультимедийного таймера можно использовать функцию `timeGetTime`. Эта функция возвращает значения, аналогичные значениям, возвращаемым функцией `GetTickCount`. Счетчик является 32-битным, обнуляемым приблизительно каждые 49,7 суток. Прототип функции `timeGetTime` имеет следующий вид:

```
DWORD timeGetTime();
```

Пример использования этой функции приведен ниже, а пока скажу несколько слов о том, как получить значения минимальной и максимальной точности рассматриваемого таймера. Для получения этих данных можно использовать функцию `timeGetDevCaps`, которая принимает в качестве параметра структуру `TTimeCaps`

и заполняет два ее поля соответствующими значениями. В листинге 8.10 приведен пример кода реализации функции определения характеристик мультимедийного таймера.

**Листинг 8.10.** Определение характеристик мультимедийного таймера

```
//Получение максимального периода таймера (мс)
UINT timeGetMaxPeriod()
{
  TIMECAPS time;
  ::timeGetDevCaps(&time, sizeof(time));
  return time.wPeriodMax;
}
//Получение минимального периода таймера (мс)
UINT timeGetMinPeriod()
{
  TIMECAPS time;
  ::timeGetDevCaps(&time, sizeof(time));
  return time.wPeriodMin;
}
```

Уже было сказано, как получать параметры таймера, но также было сказано, что точность таймера можно регулировать. Сделать это можно с помощью соответствующих функций `timeBeginPeriod` и `timeEndPeriod`.

Первая функция вызывается для установки минимальной точности таймера. Она принимает значение требуемой точности таймера в миллисекундах и возвращает значение `TIMERR_NOERROR` в случае успешного выполнения задачи и `TIMERR_NOCANDO`, если требуемая точность не может быть обеспечена.

Вторая функция восстанавливает заданную до вызова функции `timeBeginPeriod` точность таймера. В функцию `timeEndPeriod` должно передаваться то же значение, что и в функцию `timeBeginPeriod`.

В листинге 8.11 показан пример использования функций `timeBeginPeriod` и `timeEndPeriod` (реализованы функции-оболочки). При применении описанных ниже функций нужно помнить, что после вызова функции `timeSetTimerPeriod` и проведения измерения должна быть обязательно вызвана функция `timeRestoreTimerPeriod`. Функция `timeSetTimerPeriod` сохраняет значение установленной точности таймера в глобальной переменной `lastPeriod`, чтобы можно было не заботиться о сохранении этого значения в использующем таймер коде.

**Листинг 8.11.** Функции изменения точности таймера

```
static UINT lastPeriod = 0;
//Установка периода таймера (мс) перед началом измерения
bool timeSetTimerPeriod(UINT period)
{
  if ( ::timeBeginPeriod(period) == TIMERR_NOERROR )
```

```
  {
    //Сохраним значение для восстановления состояния таймера
    lastPeriod == period;
    return true;
  }
  else
  {
    //Неудача
    return false;
  }
}
//Восстановление периода таймера (обязательно)
bool timeRestoreTimerPeriod()
{
  return ::timeEndPeriod(lastPeriod) == TIMERR_NOERROR;
}
```

Теперь, после завершения рассмотрения особенностей настройки мультимедийного таймера, приведу пример его использования для измерения времени выполнения отрезка программы (листинг 8.12).

**Листинг 8.12.** Измерение времени выполнения отрезка программы

```
void __fastcall TForm1::cmbSummClick(TObject *Sender)
{
  txtMMSummTime->Text = "Измерение...";
  Refresh();
  int maxVal = StrToInt(txtMMSummMax->Text);
  //Устанавливаем максимальную точность таймера
  ::timeSetTimerPeriod(::timeGetMinPeriod());
  //Суммирование с замером времени
  UINT startTime = ::timeGetTime(); //Начальный момент времени
  __int64 summ = 0;
  for ( int arg = 1; arg <= maxVal; ++arg ) summ += arg;
  UINT endTime = ::timeGetTime(); //Конечный момент времени
  //Восстанавливаем период таймера
  ::timeRestoreTimerPeriod();
  //Время выполнения операций
  txtMMSummTime->Text = IntToStr(endTime - startTime);
}
```

## Создание программного таймера высокой точности

В самом начале рассмотрения возможностей мультимедийного таймера было сказано, что в его API заложена возможность создания программных таймеров, которые отличаются от доступного набора компонентов Timer. Это действительно так,

причем максимальная точность такого таймера может оказаться довольно высокой: на современных компьютерах создание программного таймера с периодом срабатывания 1 мс не представляет проблемы. Правда, использовать максимальную частоту таймера вряд ли стоит: слишком велика вероятность ошибки, по меньшей мере на 1 мс.

Теперь нужно уяснить, что за программный таймер предполагается создать и чем он отличается от компонента Timer, помещаемого на форму. Этот таймер отличается тем (кроме более высокой предоставляемой точности), что его не нужно привязывать к окну (форме): при срабатывании компонента Timer окну, за которым закреплен таймер, посылается сообщение WM_TIMER. Создаваемый же здесь таймер работает несколько иначе, что удобнее рассмотреть на примере. Код создания таймера может выглядеть следующим образом:

```
timerID = ::timeSetEvent(StrToInt(txtMM2Period->Text),
                         ::timeGetMinPeriod(),
                         TimerProc, 0,
                         TIME_CALLBACK_FUNCTION | TIME_
PERIODIC);
```

В приведенном выше отрывке программы с помощью функции timeSetEvent производится регистрация, запоминание адреса, функции TimerProc, периодически вызываемой Windows при срабатываниях таймера. При успешном создании таймера функция timeSetEvent возвращает ненулевое значение — идентификатор созданного таймера. Это значение может использоваться в дальнейшем для определения откликнувшегося таймера. Значение, возвращенное функцией timeSetEvent, также необходимо при удалении таймера, которое выполняется с помощью следующей команды:

```
timeKillEvent(timerID);
```

Функция timeKillEvent возвращает целочисленное значение TIMERR_NOERROR, если ее вызов завершился успешно, и MMSYSERR_INVALPARAM, если таймера, заданного параметром функции, не существует.

Теперь о функции, адрес которой передается в функцию timeSetEvent. В данном примере она выглядит следующим образом (листинг 8.13).

**Листинг 8.13.** Функция, вызываемая при срабатывании таймера

```
void __stdcall TForm1::TimerProc(UINT uTimerID, UINT uMessage,

DWORD dwUser, DWORD dw1, DWORD dw2)
{
  //Добавляем текущее значение времени в список (чтобы была
  //видна разница между моментами вызова этой функции)
  Form1->lstMM2Times->Items->Add(IntToStr(::timeGetTime()));
}
```

Естественно, функция TimerProc может выполнять совершенно разные действия. В данном случае с ее помощью заполняется список (компонент List) значениями счетчика срабатываний таймера на момент вызова процедуры (рис 8.5).

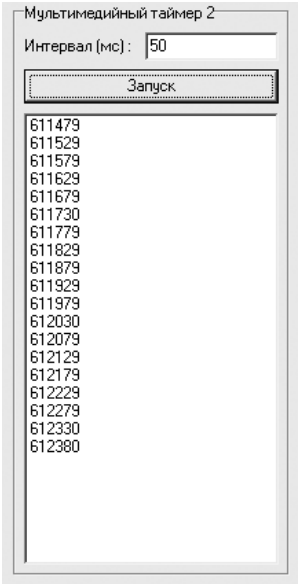


**Рис. 8.5.** Результат работы таймера

В завершение раздела вновь вернусь к функции timeSetEvent: кратко перечислю возможности, которые эта функция предоставляет, но которые не использовались в приведенном выше примере.

Как вы могли заметить, последний параметр функции timeSetEvent является битовой маской. Флаги этой маски задают количество срабатываний таймера и тип действия, которое требуется выполнять при срабатывании.

Количество срабатываний таймера определяется двумя значениями:

- TIME_ONESHOT — вызывает однократное срабатывание таймера; для таких таймеров вызывать функцию timeKillEvent после срабатывания не нужно;
- TIME_PERIODIC — вызывает периодические срабатывания таймера через заданные промежутки времени; такие таймеры нужно обязательно удалять вызовом функции timeKillEvent.

Тип действия, выполняемого таймером, задается с помощью следующих констант:

- TIME_CALLBACK_FUNCTION — вызывает при срабатывании таймера функцию, адрес которой был передан третьим параметром;
- TIME_CALLBACK_EVENT_SET — активизирует функцию SetEvent для объекта синхронизации «событие», дескриптор которого передан третьим параметром;

- TIME_CALLBACK_EVENT_PULSE — вызывает функцию PulseEvent для объекта синхронизации «событие», дескриптор которого передан третьим параметром.

К сожалению, использование объектов синхронизации хоть и является интересной темой для разговора, но выходит далеко за рамки этой главы. Поэтому, упомянув о соответствующих возможностях таймера, больше распространяться на эту тему не буду.

# Реестр

Далее будет рассмотрено несколько примеров, демонстрирующих способы использования в программах одного из важнейших хранилищ информации операционной системы Windows. Речь идет о системном реестре.

## Краткие сведения о реестре Windows

Что же представляет собой системный реестр и для чего он предназначен? Реестр состоит из нескольких файлов с достаточно сложной организацией записей, формирующих иерархическую структуру (родитель — потомки), а точнее, несколько веток структуры. Благодаря наличию специальных функций работа с реестром организуется именно как с иерархической структурой, а не как с набором разрозненных записей в файле.

Реестр Windows является превосходным примером организации централизованного хранения данных (в основном настроек программ). Кроме того, реестр представляет собой хорошую альтернативу большим INI-файлам, доставшимся в наследство от 16-разрядных версий Windows, главным образом из-за предоставляемой возможности лучше структурировать информацию (ведь секции параметров в реестре могут быть многократно вложенными). В реестре хранятся данные, которые могут быть использованы сразу несколькими программами: например, расположения COM-серверов, пути приложений, ассоциированных с различными типами файлов.

В реестре могут быть представлены объекты двух типов: ключи (во многом аналогичны папкам файловой системы) и параметры (кроме параметров по умолчанию, имеют имя, тип и значение).

Данные реестра сгруппированы в несколько ветвей (рис. 8.6). Для запуска показанной на рисунке программы Редактор реестра достаточно выполнить команду Regedit из командной строки либо отыскать файл Regedit.exe в папке Windows.

Информация, помещаемая в различных ключах реестра, группируется по следующим признакам:

- HKEY_CURRENT_USER — в этом ключе реестра содержится информация, применяемая текущим пользователем, то есть пользователем, вошедшим в систему: например, значения переменных окружения, фон Рабочего стола, вид меню Пуск;

- HKEY_USERS — этот ключ содержит настройки системы различных пользователей, а также настройки, используемые по умолчанию для нового пользователя;
- HKEY_LOCAL_MACHINE — этот ключ реестра представляет собой самую большую и главную ветвь реестра, содержащую параметры Windows, приложений, оборудования, ассоциации расширений файлов, расположения серверов COM и много другой информации;
- HKEY_CURRENT_CONFIG — в этом ключе хранятся значения параметров Windows, отличающиеся от стандартных; ключ является псевдонимом для ключа HKEY_LOCAL_MACHINE\System\CurrentControlSet\Hardware Profiles\Current;
- HKEY_CLASSES_ROOT — в системах Windows 95/98 (и, видимо, ME), NT 4.0 и более ранних этот ключ является псевдонимом ключа HKEY_LOCAL_MACHINE\Software\Classes; в Windows 2000 и XP содержимое данного ключа составляется из содержимого соответствующих ключей ветвей HKEY_LOCAL_MACHINE и HKEY_CURRENT_USER.

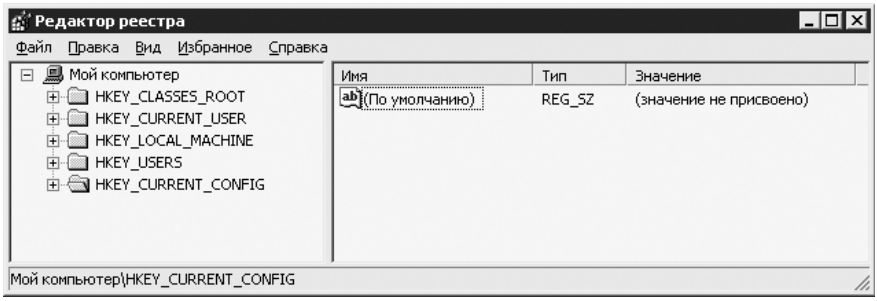


**Рис. 8.6.** Корневые ключи реестра

Доступ к ключам реестра осуществляется по дескрипторам. Дескриптор ключа можно получить при создании или открытии этого ключа, указав дескриптор одного из рассмотренных выше корневых ключей, а также путь требуемого ключа. Для хранения дескрипторов корневых ключей реестра в заголовочных файлах Windows API определены одноименные константы.

## Средства работы с реестром

Для работы с реестром предусмотрена целая группа API-функций, однако зачем изобретать велосипед, испытывая на себе «удобства» работы с этими функциями, если среда Borland предоставляет в распоряжение замечательный по своей простоте класс TRegistry? Описанию использования этого класса и посвящено несколько следующих абзацев.

Класс TRegistry находится в файле Registry.hpp. Если кому-то станет интересно использование API для работы с реестром, можете заглянуть в этот файл и там посмотреть, как реализованы методы класса TRegistry.

**ПРИМЕЧАНИЕ**

Помимо класса TRegistry, в модуле Registry можно найти такие классы, как TRegIniFile и TRegistryIniFile, позволяющие работать с реестром, будто это INI-файл. В ряде случаев использование этих классов вместо класса TRegistry позволяет сократить размер программы и значительно ее упростить.

В табл. 8.1 приведены свойства класса `TRegistry`.

**Таблица 8.1.** Свойства класса TRegistry

| Свойство | Тип | Назначение |
|---|---|---|
| CurrentKey | HKEY, только чтение | Дескриптор текущего ключа реестра |
| CurrentPath | String, только чтение | Путь текущего ключа |
| RootKey | HKEY | Дескриптор корневого ключа иерархии. Устанавливается до открытия и создания ключа. По умолчанию имеет значение HKEY_CURRENT_USER |
| LazyWrite | Boolean | Если имеет значение false (по умолчанию значение true), то при закрытии текущего ключа происходит немедленная запись изменений в файл реестра на диске. Использование немедленной записи данных реестра на диск представляется довольно расточительным в отношении использования ресурсов компьютера, но все же применимо, если нужно гарантировать сохранение внесенных в реестр сведений. Если данный режим записи выключен, то сохранение данных может произойти позже, например во время простоя системы либо перед выключением или перезагрузкой компьютера |
| Access | LongWord | Битовая маска, хранящая режим доступа к реестру. По умолчанию имеет значение KEY_ALL_ACCESS |

Список констант, которые могут объединяться операцией побитового ИЛИ для формирования значения свойства `Access`, следующий:

- `KEY_QUERY_VALUE` — указывает на необходимость получения значений параметров ключа;
- `KEY_ENUMERATE_SUB_KEYS` — указывает на возможность составления списка подключей;
- `KEY_SET_VALUE` — определяет необходимость задания значений, создания параметров в ключе;
- `KEY_CREATE_SUB_KEY` — указывает на необходимость создания подключей;
- `KEY_CREATE_LINK` — устанавливает необходимость создания символических ссылок (здесь не рассматривается);
- `KEY_NOTIFY` — указывает на необходимость получения уведомлений об изменениях ключа и его подключей (здесь не рассматривается);

- KEY_READ — является комбинацией значений KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS и KEY_NOTIFY;
- KEY_WRITE — представляет собой комбинацию значений KEY_SET_VALUE и KEY_CREATE_SUB_KEY;
- KEY_ALL_ACCESS — является комбинацией значений KEY_READ, KEY_WRITE и KEY_CREATE_LINK.

Приводить список всех методов класса TRegistry в книге нерационально и не имеет смысла — благо, названия методов говорят сами за себя, да и Borland C++ Builder поставляется с неплохой справочной системой. Здесь будет рассмотрены некоторые особенности работы с методами класса TRegistry.

При работе с ключами реестра важно (в общем случае) соблюдать такую последовательность действий.

1. Установить значение свойства RootKey, если корневой ключ отличен от HKEY_CURENT_USER; установить значение свойства Access, если нет необходимости устанавливать полный доступ.
2. Открыть методом OpenKey или создать методом CreateKey ключ реестра; если использовать метод OpenKeyReadOnly, то необходимо также задать соответствующее значение свойства Access.
3. Произвести нужные операции с элементами ключа.
4. Если вы собираетесь использовать один и тот же объект TRegistry для последовательной работы с несколькими ключами, то закрыть ключ (метод OpenKey не закрывает ранее открытый ключ).

Осталось сказать еще несколько слов о проверке результата работы методов класса TRegistry. Большинство методов этого класса, осуществляющих доступ к ключам реестра, реализованы в качестве функций, возвращающих значение true в случае успешного выполнения задачи и значение false при возникновении ошибки.

Для чтения и записи параметров разного типа в классе TRegistry предусмотрены пары read- и write-методов соответственно. Использовать их крайне просто, в чем вы скоро убедитесь. Главное при использовании этих методов — не забывать определять тип значений параметров, если он заранее вам неизвестен, с помощью метода GetDataType. Следует также помнить, что методы работы с параметрами генерируют исключение ERegistryException при возникновении ошибок.

Напоследок немного о параметре (По умолчанию) — этот безымянный параметр может присутствовать в каждом ключе. Для обращения к нему необходимо использовать пустую строку в качестве имени ключа (только нужно учитывать, что в Windows 2000/XP, в отличие от более ранних версий Windows, параметр (По умолчанию) автоматически не создается).

## Хранение настроек программы в реестре

Первый простой пример демонстрирует способ использования реестра для хранения небольшого объема данных между запусками приложения.

Пусть нужно, чтобы формы приложений запоминали свое расположение, размер, введенные и выбранные в элементах управления данные. В таком случае пользователя не будет раздражать необходимость в очередной раз перетаскивать часто открываемую форму на удобное место, а если форма требует постоянного ввода однотипных данных, то восстановление значений, выбранных и введенных в прошлый раз пользователем, его только обрадуют.

Итак, есть форма для фильтрации запроса к базе данных, показанная на рис. 8.7.

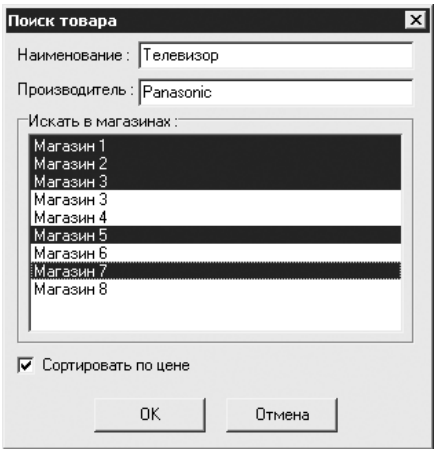


**Рис. 8.7.** Форма фильтра для поиска товара

Содержимое формы в данном случае не имеет значения — важно то, что при нажатии на кнопку OK положение, размер формы, а также данные, введенные пользователем, сохраняются в реестре с помощью функции `SaveFilter` (листинг 8.14).

**Листинг 8.14.** Сохранение параметров формы в реестре

```
void TForm1::SaveFilter()
{
  //Открываем или создаем ключ, в котором будут
  //сохранены параметры формы
  TRegistry *reg = new TRegistry();
  reg->RootKey = HKEY_CURRENT_USER;
  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 и производитель
  reg->WriteString("Name", txtName->Text);
  reg->WriteString("Producer", txtProducer->Text);
  //3. Выбранные магазины
  String strShops;
  for ( int i = 0; i < lstShops->Count; i++ )
  {
    if ( lstShops->Selected[i] )
      strShops += lstShops->Items->Strings[i] + ",";
  }
  reg->WriteString("Shops", strShops);
  //4. Применение сортировки по цене
  reg->WriteBool("SortByPrice", chkSort->Checked);
  reg->CloseKey();
  delete reg;
}
```

В рассматриваемом примере константа `strBaseKey`, определяющая положение ключа, используемого для сохранения настроек, задана следующим образом:

```
const String strBaseKey = "Software\\TricksCpp\\Settings";
```

Открыв Редактор реестра, можно удостовериться в правильном сохранении требуемых нами параметров (рис. 8.8).

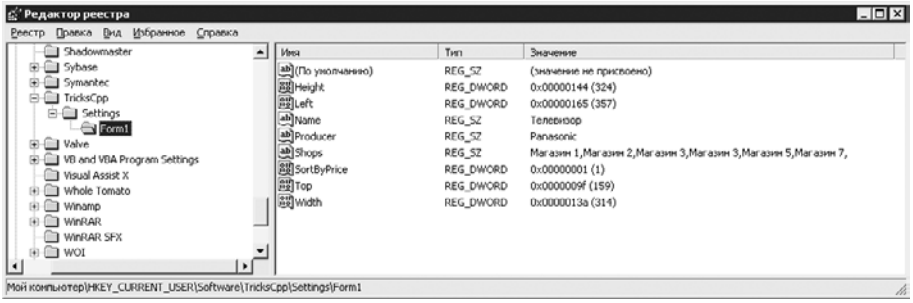


**Рис. 8.8.** Параметры формы, записанные в реестр

Считывание параметров формы можно выполнять, например, во время ее создания, тогда в обработчик события `Create` достаточно поместить вызов функции `LoadFilter` (листинг 8.15).

**Листинг 8.15.** Загрузка параметров формы из реестра

```
void TForm1::LoadFilter()
{
  //Пытаемся открыть ключ реестра с сохраненными ранее
  //данными формы
```

```
  TRegistry *reg = new TRegistry();
  reg->RootKey = HKEY_CURRENT_USER;
  if ( reg->KeyExists(strBaseKey + "\\Form1" ) )
  {
    reg->OpenKey(strBaseKey + "\\Form1", false);
    //Считываем значения из реестра
    //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 и производитель
    txtName->Text = reg->ReadString("Name");
    txtProducer->Text = reg->ReadString("Producer");
    //3. Выбранные магазины
    String strShops = reg->ReadString("Shops");
    int shopStart = 1;
    for ( int shopEnd = 1; shopEnd <= strShops.Length();
shopEnd++ )
    {
      if ( strShops[shopEnd] == ',' )
      {
        //Получение имени магазина и выделение его в списке
        SelectShop(strShops.SubString(shopStart, shopEnd -
shopStart));
        shopStart = shopEnd + 1;
      }
    }
    //4. Применение сортировки по цене
    chkSort->Checked = reg->ReadInteger("SortByPrice");
  }
  delete reg;
}
void TForm1::SelectShop(const String &name)
{
  //Выделяем в списке магазин с заданным обозначением
  for ( int i=0; i<lstShops->Items->Count; i++ )
  {
    if ( lstShops->Items->Strings[i] == name )
    {
      lstShops->Selected[i] = true;
      return;
    }
  }
}
```

Некоторая сложность алгоритма загрузки списка выбранных магазинов обусловлена желанием добиться того, чтобы при изменении списка не выделялись не выбранные ранее магазины (иначе можно было бы просто сохранять индексы, а не названия магазинов).

## Автозапуск программ

Так уж повелось, что, рассматривая работу с реестром, редко удается рассказать о способах организации автоматического запуска приложений, минуя меню Автозагрузка. Здесь эта тема также будет затронута: будут рассмотрены наиболее простые способы автоматического запуска программ, относящихся к несервисному типу.

Итак, в секциях реестра HKEY_CURRENT_USER и HKEY_LOCAL_MACHINE находятся ключи Software\Microsoft\Windows\CurrentVersion\Run и Software\Microsoft\Windows\CurrentVersion\RunOnce. В первом ключе сохраняются пути приложений, запускаемых при каждом запуске Windows. Во втором ключе обычно регистрируются приложения типа инсталляторов, которые запускаются при первой с момента регистрации перезагрузке Windows, но до запуска программы Проводника. При запуске приложения, зарегистрированного в ключе RunOnce, соответствующая запись из этого ключа автоматически удаляется.

От выбора секции реестра (HKEY_LOCAL_MACHINE или HKEY_CURRENT_USER) зависит порядок запуска приложения: в сеансе всех пользователей или только для вошедшего в данный момент в систему пользователя.

Ниже рассмотрено создание простейшей программы, позволяющей указать режим ее загрузки и, если выбран автозапуск, способ загрузки. Программа также способна создавать и удалять параметры в предусмотренных ключах реестра для задания нужного режима запуска.

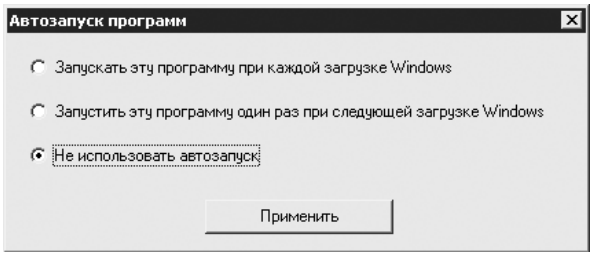


**Рис. 8.9.** Форма после определения варианта режима запуска приложения

Пусть на форме приложения расположен переключатель, имеющий три положения (рис. 8.9). Функция, код которой приведен в листинге 8.16, устанавливает положение переключателя в зависимости от того, в каком ключе секции HKEY_LOCAL_MACHINE расположен параметр с именем, совпадающим с именем программы (это условность, которая нужна для работы данного примера).

**Листинг 8.16.** Определение режима запуска приложения

```
void TForm1::GetRunMode()
{
  const String autoRunRoot = "Software\\Microsoft\\Windows\\
CurrentVersion\\";
  TRegistry *reg = new TRegistry();
  reg->RootKey = HKEY_LOCAL_MACHINE;
  //Определение режима запуска программы (по наличию значений
  //в соответствующих ключах)
  if ( reg->OpenKey(autoRunRoot + "Run", false) )
  {
    if ( reg->ValueExists(Application->Title) )
    {
      //Программа есть в секции Run - запускается при каждой
      //загрузке
      //Windows
      optRunAuto->Checked = true;
      reg->CloseKey();
      delete reg;
      return;
    }
    reg->CloseKey();
  }
  if ( reg->OpenKey(autoRunRoot + "RunOnce", false) )
  {
    if ( reg->ValueExists(Application->Title) )
    {
      //Программа есть в секции RunOnce — запускается один раз
      //при старте
      //Windows
      optRunOnce->Checked = true;
      reg->CloseKey();
      delete reg;
      return;
    }
    reg->CloseKey();
  }
  //Автозапуск программы (рассматриваемым способом) не
  //включен
  optRunNone->Checked = true;
  delete reg;
}
```

Изменение параметров режима запуска выполняется при нажатии кнопки Применить (листинг 8.17).

**Листинг 8.17.** Применение режима запуска

```
void __fastcall TForm1::cmbApplyClick(TObject *Sender)
{
  const String autoRunRoot = "Software\\Microsoft\\Windows\\
CurrentVersion\\";
  TRegistry *reg = new TRegistry();
  reg->RootKey = HKEY_LOCAL_MACHINE;
  //Отмена прошлого режима
  //..удаление параметра из секции Run
  if ( !optRunAuto->Checked )
  {
    if ( reg->OpenKey(autoRunRoot + "Run", false) )
    {
      reg->DeleteValue(Application->Title);
      reg->CloseKey();
    }
  }
  //..удаление параметра из секции RunOnce
  if ( !optRunOnce->Checked )
  {
    if ( reg->OpenKey(autoRunRoot + "RunOnce", false) )
    {
      reg->DeleteValue(Application->Title);
      reg->CloseKey();
    }
  }
  //Установка нового режима (создание параметра
  //в соответствующей секции)
  if ( optRunAuto->Checked )
  {
    //..добавление параметра в секцию Run
    if ( reg->OpenKey(autoRunRoot + "Run",  true) )
    {
      reg->WriteString(Application->Title, Application->
ExeName);
      reg->CloseKey();
    }
  }
  else if ( optRunOnce->Checked )
  {
    //..добавление параметра в секцию RunOnce
    if ( reg->OpenKey(autoRunRoot + "RunOnce", true) )
    {
      reg->WriteString(Application->Title, Application->
ExeName);
```

```
      reg->CloseKey();
    }
  }
  delete reg;
  //Для верности обновим показания на форме по данным из
  //реестра
  GetRunMode();
}
```

При желании вы можете изменить используемую здесь секцию реестра на HKEY_CURRENT_USER, если нужно, чтобы приложение, которое вы собираетесь делать, запускалось только для определенных пользователей.

## Запуск приложения из командной строки

Сразу стоит оговориться, что из командной строки можно запустить любое приложение: достаточно указать его полный или относительный (относительно рабочей папки) путь. Однако вы, возможно, замечали, что некоторые приложения можно запускать простым вводом в командную строку имени приложения, например calc (Калькулятор), msaccess (Microsoft Access) или winword (Microsoft Word). Как раз обеспечение возможности запуска приложений таким ускоренным способом будет сейчас рассмотрено.

Простым способом зарегистрировать приложение для быстрого запуска является внесение его пути в ключ реестра SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths в секции HKEY_CURRENT_USER или HKEY_LOCAL_MACHINE. Путь EXE-файла приложения должен быть записан в параметр (По умолчанию) подключа, имеющего такое же имя, как и EXE-файл приложения (включая его расширение).

Пример кода функции, регистрирующей приложение для быстрого запуска, приведен в листинге 8.18.

**Листинг 8.18.** Регистрация приложения для запуска из командной строки

```
void RegisterQuickStart()
{
  TRegistry *reg = new TRegistry();
  reg->RootKey = HKEY_LOCAL_MACHINE;
  //Регистрируем программу для запуска по имени из командной
  //строки
  if ( reg->OpenKey(paths + "\\" + Application->Title + ".
exe", true) )
  {
    reg->WriteString("", Application->ExeName);
    reg->CloseKey();
  }
```

```
  delete reg;
}
```

Для отмены быстрого запуска приложения из командной строки можно воспользоваться функцией, описание которой приведено в листинге 8.19.

**Листинг 8.19.** Отмена быстрого запуска приложения

```
void UnregisterQuickStart()
{
  TRegistry *reg = new TRegistry();
  reg->RootKey = HKEY_LOCAL_MACHINE;
  //Удаляем сведения о программе из реестра
  reg->DeleteKey(paths + "\\" + Application->Title + ".exe");
  delete reg;
}
```

В приведенных выше листингах значение константы `paths` равно:

```
const String
   paths = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\App
Paths";
```

## Регистрация расширений файлов

Далее будет рассмотрен вопрос, нередко интересующий программистов, приложения которых должны позволять сохранять и загружать данные из файлов.

Открытие файлов приложения (документов приложения) из самого приложения организовать несложно: достаточно воспользоваться окном открытия файла, вызываемым, например, специальной командой меню. Но как заставить, например, Проводник автоматически запускать приложение при выборе соответствующего файла? Сделать это несложно: достаточно внести небольшие изменения в секцию реестра `HKEY_CLASSES_ROOT`.

Перечень операций, которые нужно произвести для регистрации собственного типа файла (к примеру, `*.mydoc`), следующий.

1. Создать ключ `HKEY_CLASSES_ROOT\.mydoc`, в параметр `(По умолчанию)` которого записать имя типа файла, например `TricksCpp.DocumentSample`.
2. Создать ключ `HKEY_CLASSES_ROOT\имя_типа`, например `HKEY_CLASSES_ROOT\TricksCpp.DocumentSample`; если в параметр `(По умолчанию)` этого ключа записать строку, то она будет отображаться в качестве описания типа файла.
3. Если нужно, чтобы для отображения документа в Проводнике использовался определенный значок, то создать ключ `HKEY_CLASSES_ROOT\имя_типа\DefaultIcon`, в параметр `(По умолчанию)` которого записать полный путь

EXE- или DLL-файла, из которого предполагается использовать значок, и через запятую номер значка (см. гл. 7).

4. Наконец, для автоматического запуска приложения при выборе файла заданного типа создать ключ `HKEY_CLASSES_ROOT\имя_типа\Shell\Open\Command`, в параметр `(По умолчанию)` которого записать строку вида `путь_приложения %1` для передачи имени документа через командную строку.

Пример кода функции, производящей все перечисленные выше манипуляции, приведен ниже (листинг 8.20).

**Листинг 8.20.** Регистрация расширения файла

```
void RegisterAppDocuments()
{
  TRegistry *reg = new TRegistry();
  reg->RootKey = HKEY_CLASSES_ROOT;
  //Вносим информацию о типе файлов в реестр
  //..само расширение
  if ( reg->OpenKey(".mydoc", true) )
  {
    reg->WriteString("", "TricksCpp.DocumentSample");
    reg->CloseKey();
  }
  //..описание типа файла
  if ( reg->OpenKey("TricksCpp.DocumentSample", true) )
  {
    reg->WriteString("", "Документ TricksCpp.DocumentSam-
ple");
    reg->CloseKey();
  }
  //..значок для файлов типа *.mydoc
  if ( reg->OpenKey("TricksCpp.DocumentSample\\DefaultIcon",
true) )
  {
    reg->WriteString("", Application->ExeName + ", 1");
    reg->CloseKey();
  }
  //..приложение, открывающее документ *.mydoc
  if (reg->OpenKey("TricksCpp.DocumentSample\\Shell\\Open\\
Command", true))
  {
    reg->WriteString("", Application->ExeName + " %1");
    reg->CloseKey();
  }
  delete reg;
}
```

Результат работы этой функции показан на рис. 8.10.


**Рис. 8.10.** Результат регистрации типа файла

Теперь в файловой оболочке данное приложение запускается с путем выбранного файла (правда, в формате 8.3) в качестве аргумента командной строки (о способе перевода пути из короткой формы в длинную (если в этом вообще может быть смысл) рассказывалось в гл. 6). Читатели, которые не знакомы с тем, как получать доступ к аргументам командной строки, могут ознакомиться с листингом 8.21 (здесь имя открываемого файла помещается в текстовое поле на форме).

**Листинг 8.21.** Определение имени открываемого файла

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  GetRegisteredState();
  if ( ParamCount() > 0 )
  {
    //Обрабатываем данные в командной строке...
    txtPath->Text = ParamStr(1);
  }
}
```

Удаление сведений о типе файла из регистра производится простым удалением созданных ранее ключей, например способом, приведенным в листинге 8.22.

**Листинг 8.22.** Удаление сведений о типе файла из реестра

```
void UnregisterAppDocuments()
{
  //Удаление информации о типе файла из реестра
  TRegistry *reg = new TRegistry();
  reg->RootKey = HKEY_CLASSES_ROOT;
```

```
  reg->DeleteKey(".mydoc");
  reg->DeleteKey("TricksCpp.DocumentSample");
  delete reg;
}
```

## Программа просмотра реестра

Для демонстрации некоторых других приемов работы с реестром: например, перемещения по иерархии ключей реестра, определения списка параметров, их типа и значений — ниже рассмотрен способ реализации приложения, предоставляющего соответствующие возможности. В результате получится хоть и жалкая, но альтернатива программе Редактор реестра (правда, пригодная только для просмотра, а не для редактирования реестра).

Вид главной формы программы показан на рис. 8.11.

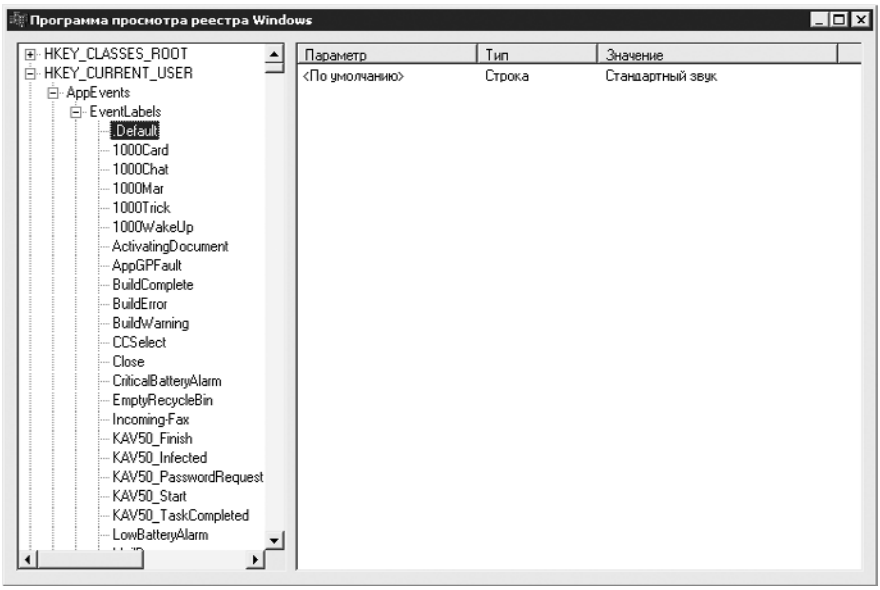


**Рис. 8.11.** Программа просмотра реестра

Функции, формирующие основу этого приложения, рассмотрены ниже в порядке их использования.

При запуске формы составляется список корневых ключей реестра (листинг 8.23).

**Листинг 8.23.** Первоначальная инициализация дерева ключей реестра

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  TTreeNode *pItem;
  //Формирование списка корневых разделов реестра
```

```
  pItem = tvwKeys->Items->AddChild(NULL, "HKEY_CLASSES_
ROOT");
  pItem->Data = (void*)HKEY_CLASSES_ROOT;
  CheckSubKeys(pItem);
  pItem = tvwKeys->Items->AddChild(NULL, "HKEY_CURRENT_
USER");
  pItem->Data = (void*)HKEY_CURRENT_USER;
  CheckSubKeys(pItem);
  pItem = tvwKeys->Items->AddChild(NULL, "HKEY_LOCAL_MA-
CHINE");
  pItem->Data = (void*)HKEY_LOCAL_MACHINE;
  CheckSubKeys(pItem);
  pItem = tvwKeys->Items->AddChild(NULL, "HKEY_USERS");
  pItem->Data = (void*)HKEY_USERS;
  CheckSubKeys(pItem);
  pItem = tvwKeys->Items->AddChild(NULL, "HKEY_CURRENT_
CONFIG");
  pItem->Data = (void*)HKEY_CURRENT_CONFIG;
  CheckSubKeys(pItem);
}
```

Вызываемая для каждого нового элемента дерева функция `CheckSubKeys` из листинга 8.23 реализована так, как показано в листинге 8.24.

**Листинг 8.24.** Оформление элемента дерева в зависимости от наличия вложенных ключей

```
void TForm1::CheckSubKeys(TTreeNode *pItem)
{
  //Проверка, есть ли в ключе реестра вложенные ключи
  TRegistry *pReg = new TRegistry;
  pReg->RootKey = GetRootKey(pItem);
  if ( pReg->OpenKeyReadOnly(GetKeyPath(pItem)) )
  {
    if ( pReg->HasSubKeys() )
    {
      //Добавляем фиктивный элемент (чтобы показывался «+»
      //для
      //разворачивания ключа). Одновременно помечаем фиктивный
      //элемент,
      //чтобы зачитать его содержимое при первом раскрытии
      //ключа
      tvwKeys->Items->AddChild(pItem, "")->Data = (void*)-1;
    }
    pReg->CloseKey();
  }
  delete pReg;
}
```

По сравнению с примером, рассмотренным в гл. 6 (дерево папок), проверка наличия дочерних ключей реестра — относительно легкая операция, поэтому она будет проведена сразу при составлении списка подключей. Как и в примере, рассмотренном ранее в гл. 6, здесь в дерево необходимо добавить фиктивный дочерний элемент для тех элементов дерева, для которых соответствующие им ключи реестра содержат подключи.

Нужно заметить, что фиктивному элементу присваивается значение -1. Именно по наличию дочернего элемента с полем Data, имеющим значение -1, можно определить, зачитывалось ли содержимое ключа, соответствующего определенному элементу дерева. Зачитывается содержимое ключа при разворачивании элемента дерева (листинг 8.25).

**Листинг 8.25.** Составление списка дочерних ключей

```
void __fastcall TForm1::tvwKeysExpanding(TObject *Sender,
TTreeNode *Node,
                                          bool &AllowExpan-
sion)
{
  if ( (int)Node->getFirstChild()->Data != -1 )
  {
    //Список подключей ранее был зачитан
    return;
  }
  Node->DeleteChildren(); //Удаление фиктивного элемента
                          //дерева
  //Загрузка списка подключей выбранного ключа
  Screen->Cursor = crHourGlass;
  TRegistry *pReg = new TRegistry;
  pReg->RootKey = GetRootKey(Node);
  if ( pReg->OpenKey(GetKeyPath(Node), false) )
  {
    //Получение списка подключей
    TStrings *pSubkeys = new TStringList;
    pReg->GetKeyNames(pSubkeys);
    for ( int i = 0; i < pSubkeys->Count; i++ )
    {
      //Добавление элемента для дочернего ключа (не забываем
      //проверять наличие подключей у каждого дочернего
      //ключа)
      CheckSubKeys(tvwKeys->Items->AddChild(Node, pSubkeys-
>Strings[i]));
    }
    delete pSubkeys;
    pReg->CloseKey();
  }
```

```
  pReg->Free();
  Screen->Cursor = crDefault;
}
```

В листинге 8.25 используются две дополнительные функции для определения полного пути ключа, соответствующего элементу дерева (без имени корневого ключа), и для получения дескриптора корневого ключа, который хранится в поле `Data` корневого элемента каждой ветви дерева.

Определение пути ключа выполняется несложно: нужно просто подняться к корню соответствующей ветви дерева, собирая по пути имена элементов дерева (листинг 8.26).

**Листинг 8.26.** Определение пути ключа в дереве

```
String GetKeyPath(TTreeNode *pItem)
{
  TTreeNode *pTemp = pItem;
  String strPath;
  while ( pTemp->Parent )
  {
    strPath = pTemp->Text + "\\" + strPath;
    pTemp = pTemp->Parent;
  }
  return strPath;
}
```

Аналогично или даже проще определяется дескриптор корневого ключа ветви реестра: для этого достаточно перейти к корню ветви дерева и прочитать значение поля `Data` корневого элемента (листинг 8.27).

**Листинг 8.27.** Определение дескриптора корневого ключа ветви

```
HKEY GetRootKey(TTreeNode *pItem)
{
  TTreeNode *pTemp = pItem;
  while ( pTemp->Parent ) pTemp = pTemp->Parent;
  return (HKEY)pTemp->Data;
}
```

При выделении элемента дерева происходит отображение параметров соответствующего ключа в списке в правой части формы. Список заполняется следующим образом (листинг 8.28).

**Листинг 8.28.** Составление списка параметров ключа реестра

```
void __fastcall TForm1::tvwKeysChange(TObject *Sender, TTree-
Node *Node)
{
  TTreeNode *pItem = tvwKeys->Selected;
```

```
if ( pItem )
{
  //Зачитаем содержимое выбранного ключа в ListView
  lvwValues->Clear();
  TRegistry *pReg = new TRegistry;
  pReg->RootKey = GetRootKey(pItem);
  if ( pReg->OpenKeyReadOnly(GetKeyPath(pItem)) )
  {
    TStrings *pValueNames = new TStringList;
    //Получение списка названий параметров
    pReg->GetValueNames(pValueNames);
    //Добавление каждого параметра в список
    for ( int i = 0; i < pValueNames->Count; i++ )
    {
      TListItem *pValueItem = lvwValues->Items->Add();
      if ( pValueNames->Strings[i] == "" )
        pValueItem->Caption = "<По умолчанию>";
      else
        pValueItem->Caption = pValueNames->Strings[i];
      //Получение типа и значения параметра
      switch ( pReg->GetDataType(pValueNames->Strings[i]) )
      {
      case rdUnknown:
        pValueItem->SubItems->Add("Неизвестно");
        break;
      case rdString:
      case rdExpandString:
        pValueItem->SubItems->Add("Строка");
        pValueItem->SubItems->
         Add(pReg->ReadString(pValueNames->Strings[i]));
        break;
      case rdInteger:
        pValueItem->SubItems->Add("Число");
        pValueItem->SubItems->Add(IntToStr
                                  (pReg->ReadInteger(
                                  pValueNames->
                                  Strings[i])));
      case rdBinary:
        pValueItem->SubItems->Add("Двоичные данные");
        break;
      }
    }
    delete pValueNames;
    pReg->CloseKey();
  }
```

```
    delete pReg;
  }
}
```

Функция, код которой приведен в листинге 8.28, не считывает значения двоичных параметров. Это сделано для упрощения и без того громоздкого фрагмента кода. В считывании значений двоичных параметров на самом деле нет ничего сложного: нужно лишь заранее определить размер данных (методом `GetDataSize`) и создать буфер соответствующего размера.

# Заключение

Вот и закончилась книга. К сожалению, рассмотреть все нюансы и подробности программирования в операционной системе Windows практически невозможно (особенно в издании такого объема). Да, собственно, и не в этом заключалась цель написания книги, ведь она не является подробным руководством по программированию в Windows.

Тем не менее хочется надеяться, что описанные в книге алгоритмы, приемы и примеры использования возможностей Windows помогли вам разобраться или хотя бы пролили свет на некоторые механизмы работы этой операционной системы.

При написании книги я старался минимизировать количество примеров, которым сложно найти практическое применение. Насколько мне это удалось, судить только вам. Мне остается лишь пожелать вам, уважаемый читатель, успехов в вашей программистской практике (неважно, с использованием C++ 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 несимвольных клавиш.

**Таблица П1.1.** Коды, обозначения и описания клавиш

| Константа | Код (десятичный) | Описание |
|---|---|---|
| VK_BACK | 8 | Клавиша Backspace |
| VK_TAB | 9 | Клавиша Tab |
| VK_RETURN | 13 | Клавиша Enter |
| VK_SHIFT | 16 | Клавиша Shift (левая или правая) |
| VK_CONTROL | 17 | Клавиша Control (левая или правая) |
| VK_MENU | 18 | Клавиша Alt (левая или правая) |
| VK_PAUSE | 19 | Клавиша Pause |
| VK_CAPITAL | 20 | Клавиша Caps Lock |
| VK_ESCAPE | 27 | Клавиша Esc (Escape) |
| VK_SPACE | 32 | Пробел |
| VK_PRIOR | 33 | Клавиша Page Up |
| VK_NEXT | 34 | Клавиша Page Down |
| VK_END | 35 | Клавиша End |
| VK_HOME | 36 | Клавиша Home |
| VK_LEFT | 37 | Клавиша «влево» (указатель) |
| VK_UP | 38 | Клавиша «вверх» (указатель) |
| VK_RIGHT | 39 | Клавиша «вправо» (указатель) |
| VK_DOWN | 40 | Клавиша «вниз» (указатель) |
| VK_SNAPSHOT | 44 | Клавиша Print Screen |
| VK_INSERT | 45 | Клавиша Insert |
| VK_DELETE | 46 | Клавиша Delete |
| VK_LWIN | 91 | Левая клавиша Win |
| VK_RWIN | 92 | Правая клавиша Win |
| VK_APPS | 93 | Клавиша для вызова контекстного меню (обычно рядом с правой клавишей Windows на клавиатуре для правши) |
| VK_NUMPAD0 | 96 | Клавиша 0 на цифровой клавиатуре |

| Константа | Код (десятичный) | Описание |
|---|---|---|
| VK_NUMPAD1 | 97 | Клавиша 1 на цифровой клавиатуре |
| VK_NUMPAD2 | 98 | Клавиша 2 на цифровой клавиатуре |
| VK_NUMPAD3 | 99 | Клавиша 3 на цифровой клавиатуре |
| VK_NUMPAD4 | 100 | Клавиша 4 на цифровой клавиатуре |
| VK_NUMPAD5 | 101 | Клавиша 5 на цифровой клавиатуре |
| VK_NUMPAD6 | 102 | Клавиша 6 на цифровой клавиатуре |
| VK_NUMPAD7 | 103 | Клавиша 7 на цифровой клавиатуре |
| VK_NUMPAD8 | 104 | Клавиша 8 на цифровой клавиатуре |
| VK_NUMPAD9 | 105 | Клавиша 9 на цифровой клавиатуре |
| VK_MULTIPLY | 106 | Клавиша * на цифровой клавиатуре |
| VK_ADD | 107 | Клавиша + на цифровой клавиатуре |
| VK_SUBTRACT | 109 | Клавиша – на цифровой клавиатуре |
| VK_DECIMAL | 110 | Разделитель дробной части на цифровой клавиатуре |
| VK_DIVIDE | 111 | Клавиша / на цифровой клавиатуре |
| VK_F1 | 112 | Функциональная клавиша F1 |
| VK_F2 | 113 | Функциональная клавиша F2 |
| VK_F3 | 114 | Функциональная клавиша F3 |
| VK_F4 | 115 | Функциональная клавиша F4 |
| VK_F5 | 116 | Функциональная клавиша F5 |
| VK_F6 | 117 | Функциональная клавиша F6 |
| VK_F7 | 118 | Функциональная клавиша F7 |
| VK_F8 | 119 | Функциональная клавиша F8 |
| VK_F9 | 120 | Функциональная клавиша F9 |
| VK_F10 | 121 | Функциональная клавиша F10 |
| VK_F11 | 122 | Функциональная клавиша F11 |
| VK_F12 | 123 | Функциональная клавиша F12 |
| VK_F13 | 124 | Функциональная клавиша F13 |
| VK_F14 | 125 | Функциональная клавиша F14 |
| VK_F15 | 126 | Функциональная клавиша F15 |
| VK_F16 | 127 | Функциональная клавиша F16 |
| VK_F17 | 128 | Функциональная клавиша F17 |
| VK_F18 | 129 | Функциональная клавиша F18 |
| VK_F19 | 130 | Функциональная клавиша F19 |
| VK_F20 | 131 | Функциональная клавиша F20 |
| VK_F21 | 132 | Функциональная клавиша F21 |
| VK_F22 | 133 | Функциональная клавиша F22 |
| VK_F23 | 134 | Функциональная клавиша F23 |
| VK_F24 | 135 | Функциональная клавиша F24 |
| VK_NUMLOCK | 144 | Клавиша Num Lock |

*Продолжение ⇩*

**Таблица П1.1** *(продолжение)*

| Константа | Код (десятичный) | Описание |
|---|---|---|
| VK_SCROLL | 145 | Клавиша Scroll Lock |
| VK_LSHIFT | 160 | Левая клавиша Shift |
| VK_RSHIFT | 161 | Правая клавиша Shift |
| VK_LCONTROL | 162 | Левая клавиша Control |
| VK_RCONTROL | 163 | Правая клавиша Control |
| VK_LMENU | 164 | Левая клавиша Alt |
| VK_RMENU | 165 | Правая клавиша Alt |

# Приложение 2. Цветовые константы

В табл. П2.1 приведены идентификаторы, числовые значения основных цветовых констант, а также названия цветов, скрывающихся за каждой константой.

**Таблица П2.1.** Основные цветовые константы

| Константа | Значение | Название цвета |
|---|---|---|
| clBlack | 0x00000000 | Черный |
| clMaroon | 0x00000080 | Темно-пурпурный |
| clGreen | 0x00008000 | Зеленый |
| clOlive | 0x00008080 | Оливковый |
| clNavy | 0x00800000 | Морской волны |
| clPurple | 0x00800080 | Пурпурный |
| clTeal | 0x00808000 | Темно-зеленый |
| clGray | 0x00808080 | Серый |
| clSilver | 0x00C0C0C0 | Серебристый |
| clRed | 0x000000FF | Красный |
| clLime | 0x0000FF00 | Зеленый |
| clYellow | 0x0000FFFF | Желтый |
| clBlue | 0x00FF0000 | Синий |
| clFuchsia | 0x00FF00FF | Фисташковый |
| clAqua | 0x00FFFF00 | Ярко-голубой |
| clLtGray | 0x00C0C0C0 | Светло-серый |
| clDkGray | 0x00808080 | Темно-серый |
| clWhite | 0x00FFFFFF | Белый |

Операционная система Windows обладает настраиваемым пользовательским интерфейсом, причем предусмотрены настройки даже цвета окон. В свою очередь, в программах можно сослаться на цвета, заданные пользователем: например, чтобы изменить цвет какого-либо места окна на цвет фона заголовка окна, используется константа `clActiveCaption`, а не цветовая константа `clBlue` (из табл. П2.1).

Полный список констант, ссылающихся на цвета элементов пользовательского интерфейса Windows, приведен в табл. П2.2.

**Таблица П2.2.** Константы, соответствующие цветам стандартных элементов интерфейса Windows

| Константа | Значение | Элемент интерфейса, цвет |
|---|---|---|
| clScrollBar | 0x80000000 | Полосы прокрутки |
| clBackground | 0x80000001 | Фон Рабочего стола |
| clActiveCaption | 0x80000002 | Заголовок активного окна |
| clInactiveCaption | 0x80000003 | Заголовок неактивного окна |
| clMenu | 0x80000004 | Фон меню |
| clWindow | 0x80000005 | Фон окон |
| clWindowFrame | 0x80000006 | Рамки окон |
| clMenuText | 0x80000007 | Текст меню |
| clWindowText | 0x80000008 | Текст окон |
| clCaptionText | 0x80000009 | Текст заголовка неактивного окна |
| clActiveBorder | 0x8000000A | Обрамление активного окна |
| clInactiveBorder | 0x8000000B | Обрамление неактивного окна |
| clAppWorkSpace | 0x8000000C | Рабочая область компонента |
| clHighlight | 0x8000000D | Фон выделенного текста |
| clHightlightText | 0x8000000E | Выделенный текст |
| clBtnFace | 0x8000000F | Цвет поверхности кнопок |
| clBtnShadow | 0x80000010 | Тени кнопок |
| clGrayText | 0x80000011 | Текст недоступных элементов |
| clBtnText | 0x80000012 | Текст кнопок |
| clInactiveCaptionText | 0x80000013 | Текст заголовка неактивного окна |
| clBtnHighlight | 0x80000014 | Выделенная кнопка |
| cl3DDkShadow | 0x80000015 | Темные тени трехмерных компонентов |
| cl3DLight | 0x80000016 | Светлые части трехмерного компонента |
| clInfoText | 0x80000017 | Текст всплывающих подсказок |
| clInfoBk | 0x80000018 | Фон всплывающих подсказок |
| clGradientActiveCaption | 0x8000001B | Правая сторона градиентной заливки названия активного окна |
| clGradientInactiveCaption | 0x8000001C | Правая сторона градиентной заливки названия неактивного окна |

# Приложение 3. Описание компакт-диска

На компакт-диске, прилагаемом к книге, находятся примеры почти всех программ, которые здесь описаны. Примеры находятся в папках, соответствующих главам в книге.

- Глава 1. Окна — примеры для работы с окнами: привлечение внимания к окну, растягиваемые формы, окна нестандартной формы и пр.
- Глава 2. Графика — работа с графикой: рисование на форме, создание простого графического редактора, преобразование изображений.
- Глава 3. Меню и компоненты — примеры программ для создания меню и графических списков.
- Глава 4. Мультимедиа — создание мультимедийных приложений: универсального проигрывателя, проигрывателя компакт-дисков, простого синтезатора и пр.
- Глава 5. Мышь и клавиатура — примеры программ для работы с мышью и клавиатурой: захват указателя мыши, ограничение перемещения указателя, вычисление расстояния, пройденного мышью, опрос клавиатуры, имитация нажатия клавиш и многое другое.
- Глава 6. Папки, файлы и диски — управление папками, файлами и дисками: поиск, операции над деревом папок, отслеживание изменений, копирование файлов, просмотр свойств диска и пр.
- Глава 7. Ресурсы — примеры программ для управления ресурсами.
- Глава 8. Системная информация и реестр — работа с системной информацией: использование имени компьютера, имени пользователя, настройки системного времени, просмотр и редактирование реестра.

*Чиртик Александр Анатольевич*

**Программирование на C++. Трюки и эффекты (+CD)**


| | |
|---|---|
| Заведующий редакцией | *А. Громаковский* |
| Ведущий редактор | *Н. Гринчик* |
| Литературный редактор | *Д. Романов* |
| Художник | *А. Смага* |
| Корректоры | *Н. Терех, Ю. Цеханович* |
| Верстка | *А. Барцевич* |

Подписано в печать 22.01.10. Формат 70×100/16. Усл. п. л. 28,38. Тираж 3000. Заказ 0000.

ООО «Лидер», 194044, Санкт-Петербург, Б. Сампсониевский пр., 29а.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.

# ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

***У Вас есть свой сайт?***
***Вы ведете блог?***
***Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?***
***ЭТО ВПОЛНЕ РЕАЛЬНО!***

## СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!

- *Зарегистрируйтесь на нашем сайте в качестве партнера по адресу* ***www.piter.com/ePartners***
- *Получите свой персональный уникальный номер партнера*
- *Выбирайте книги на сайте* ***www.piter.com,*** *размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт* ***www.piter.com****)*

**ВНИМАНИЕ!** *В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.*

**С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером.** А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

**Пример партнерской ссылки:**
http://www.piter.com/book.phtml?978538800282 – обычная ссылка
http://www.piter.com/book.phtml?978538800282**&refer=0000** – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

**Подробно о Партнерской программе ИД «Питер» читайте на сайте WWW.PITER.COM**

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

**ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»**
**предлагают эксклюзивный ассортимент компьютерной, медицинской, психологической, экономической и популярной литературы**

## РОССИЯ

**Санкт-Петербург** м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

**Москва** м. «Электрозаводская», Семеновская наб., д. 2/1, корп. 1, 6-й этаж
тел./факс: (495) 234-38-15, 974-34-50; e-mail: sales@msk.piter.com

**Воронеж** Ленинский пр., д. 169; тел./факс: (4732) 39-61-70
e-mail: piterctr@comch.ru

**Екатеринбург** ул. Бебеля, д. 11а; тел./факс: (343) 378-98-41, 378-98-42
e-mail: office@ekat.piter.com

**Нижний Новгород** ул. Совхозная, д. 13; тел.: (8312) 41-27-31
e-mail: office@nnov.piter.com

**Новосибирск** ул. Станционная, д. 36; тел.: (383) 363-01-14
факс: (383) 350-19-79; e-mail: sib@nsk.piter.com

**Ростов-на-Дону** ул. Ульяновская, д. 26; тел.: (863) 269-91-22, 269-91-30
e-mail: piter-ug@rostov.piter.com

**Самара** ул. Молодогвардейская, д. 33а; офис 223; тел.: (846) 277-89-79
e-mail: pitvolga@samtel.ru

## УКРАИНА

**Харьков** ул. Суздальские ряды, д. 12, офис 10; тел.: (1038057) 751-10-02
758-41-45; факс: (1038057) 712-27-05; e-mail: piter@kharkov.piter.com

**Киев** Московский пр., д. 6, корп. 1, офис 33; тел.: (1038044) 490-35-69
факс: (1038044) 490-35-68; e-mail: office@kiev.piter.com

## БЕЛАРУСЬ

**Минск** ул. Притыцкого, д. 34, офис 2; тел./факс: (1037517) 201-48-77
e-mail: gv@minsk.piter.com

---

Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 703-73-73**. **E-mail:** fuganov@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 по тел.: (812) 703-73-73.
Специальное предложение – e-mail: kozin@piter.com

---

Заказ книг по почте: на сайте **www.piter.com**; по тел.: (812) 703-73-74
по ICQ 413763617

---

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

**УВАЖАЕМЫЕ ГОСПОДА!
КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
ВЫ МОЖЕТЕ ПРИОБРЕСТИ
ОПТОМ И В РОЗНИЦУ У НАШИХ
РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.**

## ДАЛЬНИЙ ВОСТОК

**Владивосток**
«Приморский торговый дом книги»
тел./факс: (4232) 23-82-12
e-mail: bookbase@mail.primorye.ru

**Хабаровск**, «Деловая книга», ул. Путевая, д. 1а
тел.: (4212) 36-06-65, 33-95-31
e-mail: dkniga@mail.kht.ru

**Хабаровск**, «Книжный мир»
тел.: (4212) 32-85-51, факс: (4212) 32-82-50
e-mail: postmaster@worldbooks.kht.ru

**Хабаровск**, «Мирс»
тел.: (4212) 39-49-60
e-mail: zakaz@booksmirs.ru

## ЕВРОПЕЙСКИЕ РЕГИОНЫ РОССИИ

**Архангельск**, «Дом книги», пл. Ленина, д. 3
тел.: (8182) 65-41-34, 65-38-79
e-mail: marketing@avfkniga.ru

**Воронеж**, «Амиталь», пл. Ленина, д. 4
тел.: (4732) 26-77-77
http://www.amital.ru

**Калининград**, «Вестер»,
сеть магазинов «Книги и книжечки»
тел./факс: (4012) 21-56-28, 6 5-65-68
e-mail: nshibkova@vester.ru
http://www.vester.ru

**Самара**, «Чакона», ТЦ «Фрегат»
Московское шоссе, д.15
тел.: (846) 331-22-33
e-mail: chaconne@chaccone.ru

**Саратов**, «Читающий Саратов»
пр. Революции, д. 58
тел.: (4732) 51-28-93, 47-00-81
e-mail: manager@kmsvrn.ru

## СЕВЕРНЫЙ КАВКАЗ

**Ессентуки**, «Россы», ул. Октябрьская, 424
тел./факс: (87934) 6-93-09
e-mail: rossy@kmw.ru

## СИБИРЬ

**Иркутск**, «ПродаЛитЪ»
тел.: (3952) 20-09-17, 24-17-77
e-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

**Иркутск**, «Светлана»
тел./факс: (3952) 25-25-90
e-mail: kkcbooks@bk.ru
http://www.kkcbooks.ru

**Красноярск**, «Книжный мир»
пр. Мира, д. 86
тел./факс: (3912) 27-39-71
e-mail: book-world@public.krasnet.ru

**Новосибирск**, «Топ-книга»
тел.: (383) 336-10-26
факс: (383) 336-10-27
e-mail: office@top-kniga.ru
http://www.top-kniga.ru

## ТАТАРСТАН

**Казань**, «Таис»,
сеть магазинов «Дом книги»
тел.: (843) 272-34-55
e-mail: tais@bancorp.ru

## УРАЛ

**Екатеринбург**, ООО «Дом книги»
ул. Антона Валека, д. 12
тел./факс: (343) 358-18-98, 358-14-84
e-mail: domknigi@k66.ru

**Екатеринбург**, ТЦ «Люмна»
ул. Студенческая, д. 1в
тел./факс: (343) 228-10-70
e-mail: igm@lumna.ru
http://www.lumna.ru

**Челябинск**, ООО «ИнтерСервис ЛТД»
ул. Артиллерийская, д. 124
тел.: (351) 247-74-03, 247-74-09,
247-74-16
e-mail: zakup@intser.ru
http://www.fkniga.ru, www.intser.ru