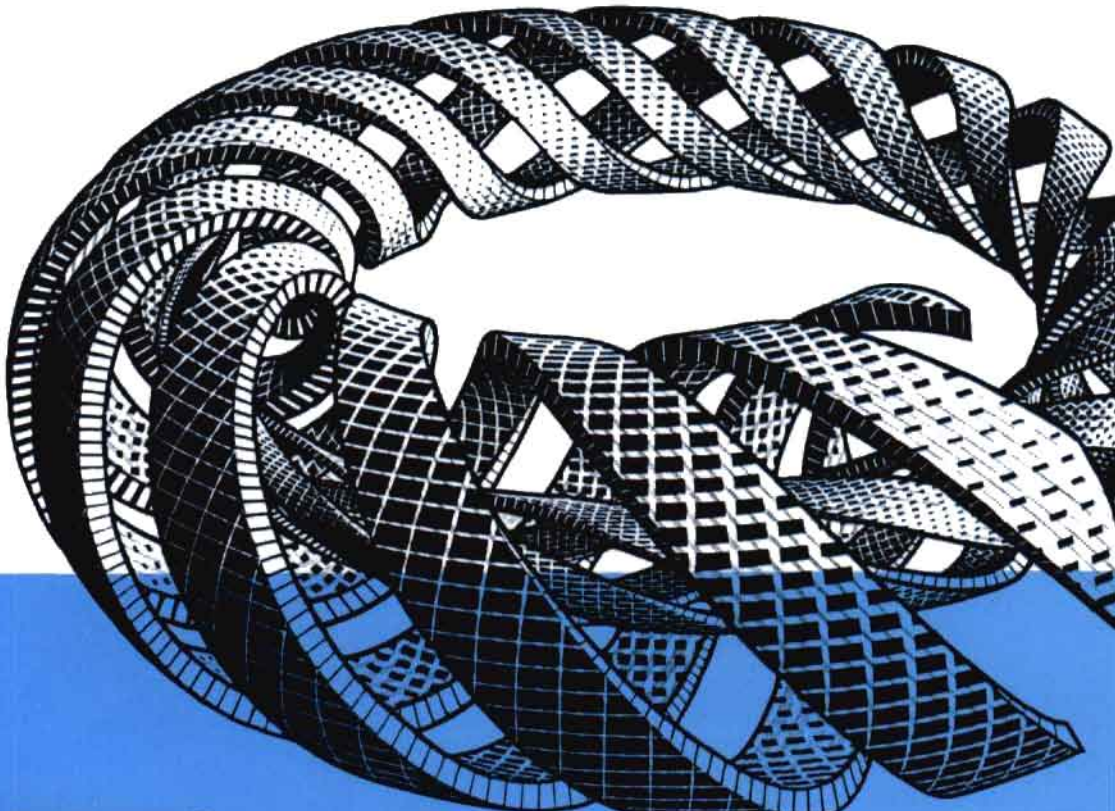
В. Ермолаев, Т. Сорока

C++ BUILDER: КНИГА РЕЦЕПТОВ



КУДИЦ-ОБРАЗ

Вячеслав Ермолаев, Тарас Сорока

C++ Builder:

Книга рецептов

КУДИЦ-ОБРАЗ
Москва • 2006

ББК 32.973.-018.2
Вячеслав Ермолаев, Тарас Сорока
C++ Builder: Книга рецептов
М.: КУДИЦ-ОБРАЗ, 2006. – 208 с.

Данная книга написана специалистами в области разработки ПО по материалам дискуссий на самом известном российском сайте, посвященном C++Builder: *http://bcbdev.ru*. В книге, построенной как справочник, даются примеры решения типичных задач, встающих в процесс разработки приложения на C++Builder. Это позволяет разработчикам сконцентрироваться на предметной области, экономя время и не отвлекаясь на частности.

Кроме основной массы вопросов, касающихся создания пользовательского интерфейса, также затрагивается работа с файлами, реестром и рядом внутренних классов VCL. Издание сопровождается компакт-диском с полным кодом всех рассмотренных проектов.

Для профессиональных разработчиков. Также книга может быть полезна студентам и аспирантам соответствующих специальностей.

Вячеслав Ермолаев, Тарас Сорока

C++ Builder: Книга рецептов
Учебно-справочное издание

Корректор В. Клименко
Макет О. Горкина

ISBN 5-9579-0091-5

«ИД КУДИЦ-ОБРАЗ»
119049, Москва, Ленинский пр-т., д. 4, стр. 1А. Тел.: 333-82-11, ok@kudits.ru

Подписано в печать 11.07.05.
Формат 70×90/16.
Печать офс. Бумага газ.
Усл. печ. л. 15,21. Тираж 3000. Заказ 1534

Отпечатано с готовых диапозитивов
в ОАО «Щербинская типография»
117623, Москва, ул. Типографская, д. 10

Как устроена эта книга

Данная книга устроена в виде сборника вопросов и ответов. Под вопросом понимается формулировка проблемы, часто обсуждавшейся на форумах *http://bcbdev.ru*, а также на других форумах и конференциях, участниками которых являются авторы; под ответом – развернутое решение данной проблемы вместе с исходным кодом. В ответе объясняется не только как решается данная проблема, но и почему она так решается, а также для чего служит тот или иной этап в ее решении. Почти для каждого рассмотренного в книге вопроса предоставлен тестовый проект.

Данная книга состоит из трех разделов: первый, небольшой, посвящен C++Builder еще пятой версии. Во-первых, до сих пор еще значительное число разработчиков использует в своей работе эту версию средства разработки, и при ее использовании возникают некие трудности, которые мы не смогли обойти стороной. Во-вторых, тем, кто использует C++Builder последней, шестой версии, также рекомендуем не пропускать данный раздел. Приведенные в нем примеры будут полезны и вам, тем более что в шестой версии их можно использовать без каких бы то ни было изменений: просто в новой версии для решения описываемых проблем появился стандартный путь от Borland.

Второй раздел состоит из подробного описания нескольких классов библиотеки VCL, которые либо недостаточно известны, либо по каким-то непонятным причинам вызывают трудности в использовании и типичные ошибки у некоторого круга разработчиков.

И наконец, третий раздел. Собственно описание заинтересовавшего авторов вопроса и его подробное, пошаговое решение с комментариями ко всему тому, что комментировать можно и нельзя.

Сразу признаем, что книга ни в коей мере не претендует на полноту охвата материала. Вопросы, рассматриваемые в данной книге, отбирались авторами по своему разумению. Если вы считаете, что мы забыли о чем-то рассказать, о чем рассказать были должны, или просто считаете, что есть интересные темы, не освещенные в данной книге, пишите на *tnsoroka@bcbdev.ru*. Мы постараемся учесть ваши пожелания в последующих изданиях данной или в новых книгах.

Обо всех замеченных в книге ошибках также сообщайте на *tnsoroka@bcbdev.ru*.

Авторы.

Глава 1

О версии прошлой замолвите слово...

Прозрачность в W2K/XP с использованием SetLayeredWindowAttributes

Я думаю, вы видели в некоторых приложениях прозрачные окна и наверняка задавали себе вопрос о реализации подобной прозрачности. А реализована прозрачность может быть двумя способами. Один из них – сложный, неудобный и достаточно нетривиальный – применялся программистами в системах Windows NT 4.0, Windows 95–Windows Me включительно, но его мы рассматривать не будем в связи с неактуальностью на сегодняшний день данных систем и со сложностью реализации данного способа, а вот о втором способе, достаточно приятном и не очень трудном, мы поговорим.

Начиная с Windows 2000, в Windows API включена новая функция, ***SetLayeredWindowAttributes***. Данная функция объявлена в Winuser.h как

```
BOOL SetLayeredWindowAttributes(
    HWND hwnd,
    COLORREF crKey,
    BYTE bAlpha,
    DWORD dwFlags
);
```

и позволяет в операционных системах, начиная с Windows 2000 и далее, сделать окно прозрачным, причем интенсивность прозрачности может регулироваться.

Первый параметр функции, ***hwnd***, – это дескриптор окна, которое мы желаем сделать прозрачным. К окну предъявляются особые требования: у него должен быть установлен стиль WS_EX_LAYERED. Стиль может быть установлен либо во время создания окна функцией ***CreateWindowEx***, либо функцией ***SetWindowLong*** после создания окна. Поскольку мы пишем в C++Builder, первый способ для нас, я считаю, не очень актуален, и мы будем использовать установку стиля с помощью ***SetWindowLong***.

Второй параметр функции ***SetLayeredWindowAttributes***, ***crKey***, представляет собой значение типа ***COLORREF***, которое определяет цвет прозрачности для окна (об этом будет сказано ниже).

Третий параметр, ***bAlpha***, определяет степень прозрачности окна при использовании альфа-прозрачности (разъяснения также будут приведены ниже). Может принимать значения из диапазона 0–255. При 0 окно является полностью прозрачным, при 255 – полностью непрозрачным.

И наконец, четвертый параметр, ***dwFlags***, определяет тип прозрачности окна. Может принимать одно из следующих значений:

- LWA_COLORKEY
- LWA_ALPHA

Значение LWA_COLORKEY реализует так называемую "прозрачность цветового ключа". То есть, говоря иными словами, при указании LWA_COLORKEY как значения параметра ***dwFlags*** в качестве прозрачного цвета будет использоваться цвет, определенный значением параметра ***crKey***. В этом случае все в окне, что закрашено данным цветом, будет прозрачным.

Значение LWA_ALPHA реализует альфа-прозрачность. Альфа-прозрачность – это прозрачность всего окна целиком, вне зависимости от того, в какой цвет раскрашена та или иная его часть. Интенсивность прозрачности регулируется значением параметра ***bAlpha***, о чем уже упоминалось выше.

Теперь, собственно, можно приступить к практике. Загрузите тестовый проект в C++Builder. На мониторе у вас должно быть нечто вроде этого.

Группой радиокнопок мы выбираем вид прозрачности окна, а ползунком можно регулировать степень прозрачности окна при отмеченной радиокнопке "Альфа-прозрачность".

Прежде чем запускать проект, давайте рассмотрим, наконец, как устанавливается стиль WS_EX_LAYERED. Вот код для установления стиля.

```
SetWindowLong(MainForm->Handle, GWL_EXSTYLE,GetWindowLong
(MainForm->Handle, GWL_EXSTYLE) |WS_EX_LAYERED);
```

В функции ***SetWindowLong*** первый параметр является дескриптором окна, для которого устанавливается стиль. В нашем случае стиль устанавливается для главной (и единственной) формы проекта – ***MainForm***. GWL_EXSTYLE во втором параметре означает, что мы устанавливаем новое значение для расширенных стилей окна. Третий параметр в функции ***SetWindowLong*** определяет новое значение для второго параметра. В нашем случае мы в качестве значения третьего параметра использовали результат вызова функции ***GetWindowLong***.

```
GetWindowLong(MainForm->Handle, GWL_EXSTYLE) | WS_EX_LAYERED).
```

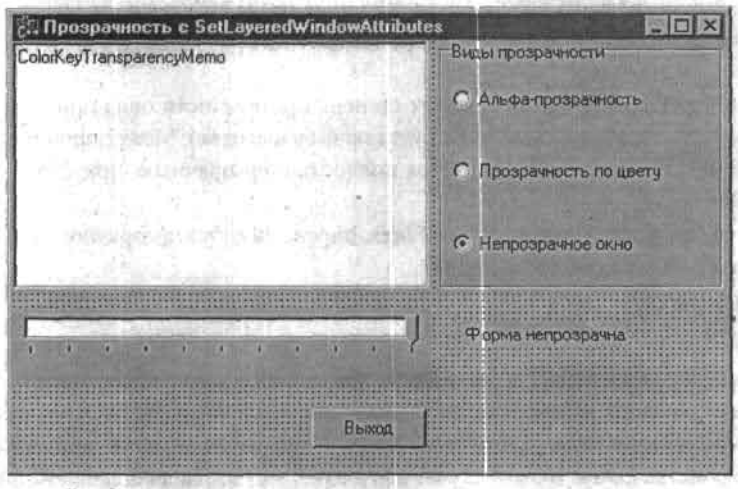


Рис. 1.1. *Внешний вид тестового проекта*

Данная строка возвращает комбинацию текущих стилей окна ***MainForm*** и стиля WS_EX_LAYERED. Иными словами, можно считать, что этот код просто добавляет стиль WS_EX_LAYERED в расширенные стили окна.

Теперь запустите проект и отметьте радиокнопку "Прозрачность по цвету". У вас будет нечто похожее на изображенное на рис. 1.2.

Такой внешний вид формы обусловлен тем, что в обработчике события ***OnClick*** компонента ***TransparencySelectionRadioGroup*** (***TRadioGroup***) для кнопки "Прозрачность по цвету" присутствует следующий код.

```
SetLayeredWindowAttributes(MainForm->Handle, clWhite, 0, LWA_COLORKEY);
```

То есть все, что было на форме белого цвета, стало прозрачным. При этом виде прозрачности появляется еще один интересный эффект: попробуйте мышью передать фокус в ***ColorKeyTransparencyMemo*** – у вас ничего не получится. Фокус будет передан в то окно, которое лежит непосредственно под ***ColorKeyTransparencyMemo*** и которое сквозь него "просвечивает". Также обратите внимание, что стали прозрачными все дочерние элементы управления, которые имели белый цвет.

Теперь отметьте радиокнопку "Альфа-прозрачность" и попробуйте подвигать ползунок. У вас будет на экране следующая картина.

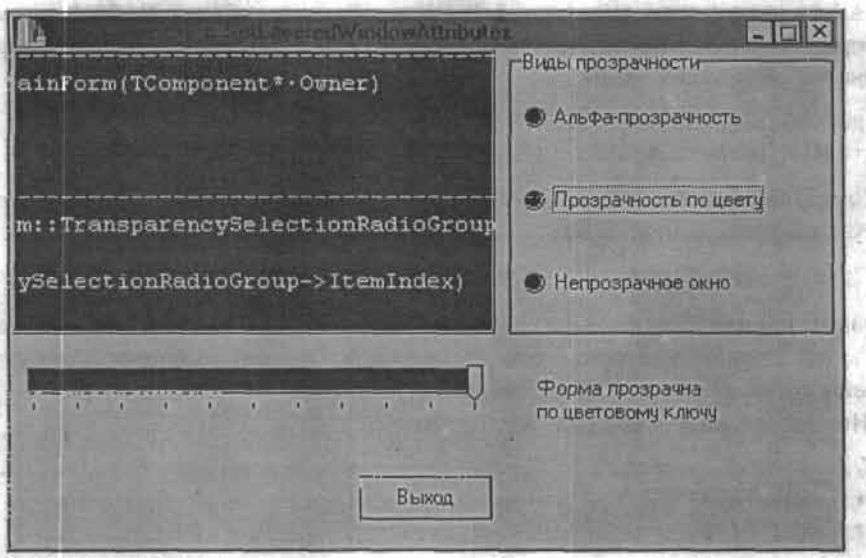


Рис. 1.2. *Прозрачность окна по цвету*

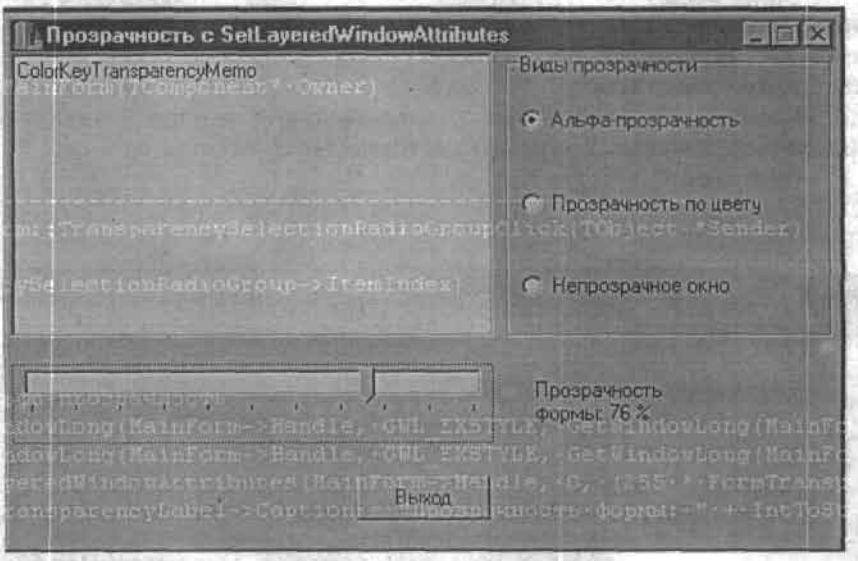


Рис. 1.3. *Вид тестового проекта с альфа-прозрачностью*

Альфа-прозрачность устанавливается в обработчике события ***OnClick*** компонента ***TransparencySelectionRadioGroup*** (***TRadioGroup***) при отметке радиокнопки "Альфа-прозрачность" следующим кодом.

```
SetLayeredWindowAttributes(MainForm->Handle, 0,
(255 * FormTransparencyTrackBar->Position) / 100, LWA_ALPHA);
```

Значение параметра ***crKey*** установлено в 0, так как мы используем не прозрачность по цвету, а альфа-прозрачность. Код

```
(255 * FormTransparencyTrackBar->Position) / 100
```

переводит прозрачность формы из диапазона изменений от 0 до 255 в диапазон значений от 0 до 100. В процентное соотношение. Для удобства. Сама же интенсивность прозрачности задается позицией ползунка ***FormTransparencyTrackBar***.

Ниже приведен весь код тестового проекта.

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "TransparencyUnit.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
   : TForm(Owner)
{
}
//---------------------------------------------------------------------
void __fastcall TMainForm::TransparencySelectionRadioGroupClick
(TObject *Sender)
   {
      switch(TransparencySelectionRadioGroup->ItemIndex)
         {
            case 0:
               {
                  // альфа-прозрачность
                  SetWindowLong(MainForm->Handle, GWL_EXSTYLE,
                  GetWindowLong(MainForm->Handle, GWL_EXSTYLE) &
                  ~WS_EX_LAYERED);
```

```
                SetWindowLong(MainForm->Handle, GWL_EXSTYLE,
                GetWindowLong(MainForm->Handle, GWL_EXSTYLE) |
                WS_EX_LAYERED);
              SetLayeredWindowAttributes(MainForm->Handle, 0, (255 *
               FormTransparencyTrackBar->Position) / 100, LWA_ALPHA);
               FormTransparencyLabel->Caption = "Прозрачность формы: " +
               IntToStr(FormTransparencyTrackBar->Position) + " %";
            }
          break;
          case 1:
            {
              // прозрачность цветового ключа
              SetWindowLong(MainForm->Handle, GWL_EXSTYLE,
              GetWindowLong(MainForm->Handle, GWL_EXSTYLE) &
              WS_EX_LAYERED);
              SetWindowLong(MainForm->Handle, GWL_EXSTYLE,
              GetWindowLong(MainForm->Handle, GWL_EXSTYLE) |
              WS_EX_LAYERED);
             SetLayeredWindowAttributes(MainForm->Handle, clWhite, 0,
              LWA_COLORKEY);
             FormTransparencyLabel->Caption = "Форма прозрачна по
              цветовому ключу";
            }
          break;
          case 2:
            {
              // непрозрачное окно
              SetWindowLong(MainForm->Handle, GWL_EXSTYLE,
              GetWindowLong(MainForm->Handle, GWL_EXSTYLE) &
              ~WS_EX_LAYERED);
             FormTransparencyLabel->Caption = "Форма непрозрачна";
            }
          break;
        }
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::FormTransparencyTrackBarChange(TObject *Sender)
    {
      if(!TransparencySelectionRadioGroup->ItemIndex)
        {
          SetLayeredWindowAttributes(MainForm->Handle, 0, (255 *
```

```
            FormTransparencyTrackBar->Position) / 100, LWA_ALPHA);
          FormTransparencyLabel->Caption = "Прозрачность формы: " +
            IntToStr(FormTransparencyTrackBar->Position) + " %";
        }
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::ExitButtonClick(TObject *Sender)
    {
      Application->Terminate();
    }
//---------------------------------------------------------------------
```

Вот, собственно, и все, что можно рассказать о прозрачности с использованием функции ***SetLayeredWindowAttributes***. Хотелось бы, правда, напоследок заострить ваше внимание на двух аспектах.

Первый аспект: после работы с прозрачным окном не забудьте эту прозрачность снять следующим кодом.

```
SetWindowLong(MainForm->Handle, GWL_EXSTYLE, GetWindowLong
(MainForm->Handle, GWL_EXSTYLE) & ~WS_EX_LAYERED);
```

Непрозрачная форма и прозрачная, но с установленной в 255 прозрачностью – не одно и то же.

Второй аспект: возможно, как в тестовом проекте, сделать окно полупрозрачным вместе со всеми его дочерними окнами, но невозможно использовать альфа-прозрачность для отдельного элемента управления в окне.

Стиль csOwnerDrawFixed в TComboBox

Один из распространенных вопросов, которые во множестве задаются в форумах и конференциях от начинающих (и не очень) программистов: как использовать собственную отрисовку в компонентах VCL для кастомизации их внешнего вида? Есть ответ на этот вопрос. Ниже будет рассмотрена собственная отрисовка в компоненте ***TComboBox*** с использованием стиля ***csOwnerDrawFixed***.

Немного теории. Для того чтобы выполнять собственную отрисовку в ***TComboBox*** (и в ***TListBox*** тоже), необходимо значение свойства ***Style*** установить в ***csOwnerDrawFixed*** или ***csOwnerDrawVariable***. Установка ***Style*** в ***csOwnerDrawFixed*** применяется, когда все элементы списка имеют одинаковую высоту – высота каждого элемента списка будет определяться значением свойства ***ItemHeight***. При установке значения свойства ***Style*** в ***csOwnerDrawVariable*** высота каждого элемента может быть различной. Для измерения высоты отдельного элемента в данном случае необходимо писать обработчик события

OnMeasureItem. Данный путь несколько сложнее, поэтому для начала рассмотрим работу с ***TComboBox*** со стилем ***csOwnerDrawFixed***, а потом уже рассмотрим ***TListBox*** со стилем ***csOwnerDrawVariable***.

Итак, задача. Давайте создадим выпадающие списки (***TComboBox***) с элементами разного цвета. Подобные списки вы наверняка видели во многих программах. Мы создадим два списка. Первый – для выбора цвета шрифта, второй – для выбора цвета фона или цвета собственно элемента управления (см. рис. 1.4 и 1.5 соответственно).

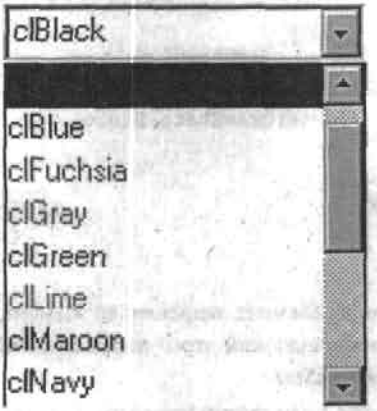


Рис. 1.4. *Выпадающий список для задания цвета шрифта элемента управления*

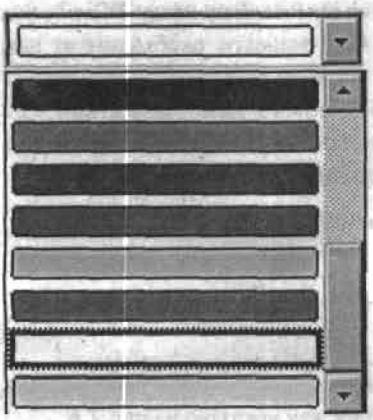


Рис. 1.5. *Выпадающий список для задания цвета фона или цвета самого элемента управления*

При использовании стиля ***csOwnerDrawFixed*** в ***TComboBox*** основным и единственным инструментом для кастомизации внешнего вида является обработка события ***OnDrawItem***.

```
__property TDrawItemEvent OnDrawItem = {read=FOnDrawItem, write=FOnDrawItem};
```

Данное событие возникает всякий раз, когда необходимо отобразить на экране элемент списка. Событие ***OnDrawItem*** происходит только при установке значения свойства ***Style*** в ***csOwnerDrawFixed*** или ***csOwnerDrawVariable*** – при других значениях свойства ***Style*** событие ***OnDrawItem*** не происходит.

Тип ***TDrawItemEvent***

```
typedef void __fastcall (__closure *TDrawItemEvent)(Controls::TWinControl*
Control, int Index, const Windows::TRect &Rect, TOwnerDrawState State);
```

включает в себя следующие параметры:

Control – компонент, в котором происходит событие;

Index – индекс элемента в свойстве ***Items***;

Rect – координаты элемента на канве компонента;

State – состояние элемента, которое указывает, выбран ли элемент, запрещен ли элемент, активен ли он в настоящий момент и т. д. Полный список значений этого параметра вы можете посмотреть в справке по VCL для типа ***TOwnerDrawState***.

Как видно из вышеприведенного, путь решения нашей с вами задачи пролегает следующим образом: мы должны получить в обработчике события ***OnDrawItem*** канву ***TComboBox*** и затем на этой канве в пределах прямоугольника ***Rect*** вывести необходимую нам информацию. Приступим.

Начнем с первого выпадающего списка – с разноцветными текстовыми элементами. Код, который нам потребуется для этого, не слишком сложен и не слишком большой по размерам.

Сперва нам необходимо получить доступ к канве выпадающего списка.

```
TComboBox *pComboBox = static_cast <TComboBox *> (Control);
TCanvas *pCanvas = pComboBox->Canvas;
```

Затем мы должны очистить прямоугольник, определяемый значением параметра ***Rect***.

```
pCanvas->FillRect(Rect);
```

Если этого не сделать, то при отображении элемента при его раскрытии и перемещении над ним курсора мыши будут видны артефакты изображения, как показано на рис. 1.6.

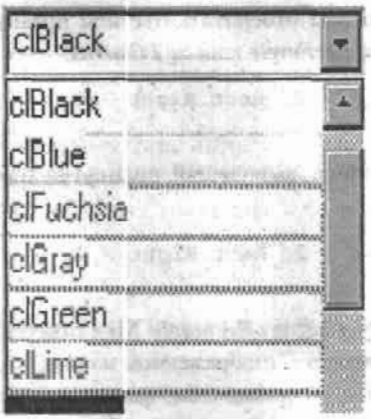


Рис. 1.6. *Артефакты изображения при отсутствии FillRect*

Затем нам необходимо установить цвет каждого элемента в списке. Учитывая, что в списке должны отображаться названия цветов, самым простым и удобным способом для индивидуального задания цвета будет следующий.

```
pCanvas->Font->Color = StringToColor(pComboBox->Items->Strings[Index]);
```

Здесь мы воспользовались функцией ***StringToColor***, которая преобразует строку "clBlack" в цветовое значение ***clBlack***.

И наконец, мы должны вывести на экран каждый элемент списка. Сделаем это с помощью метода ***TextOut*** класса ***TCanvas***.

```
pCanvas->TextOut(Rect.Left, Rect.Top, pComboBox->Items->Strings[Index]);
```

Теперь перейдем ко второму выпадающему списку. Он не должен отображать текстовых элементов, а вместо них в списке должны быть цветные прямоугольники.

В обработчике события мы точно так же получаем канву ***TComboBox*** и очищаем прямоугольник.

```
TComboBox *pComboBox = static_cast <TComboBox *> (Control);
TCanvas *pCanvas = pComboBox->Canvas;
pCanvas->FillRect(Rect);
```

Поскольку нам требуется вместо текста выводить геометрические фигуры, мы устанавливаем соответствующий цвет для свойства ***Brush*** класса ***TCanvas***, а не для свойства ***Font***.

```
pCanvas->Brush->Color = StringToColor(pComboBox->Items->Strings[Index]);
```

После всех вышеописанных приготовлений мы готовы отобразить цветной прямоугольник в качестве элемента списка с помощью метода ***Rectangle*** класса ***TCanvas***.

```
pCanvas->Rectangle(Rect.Left + 2, Rect.Top + 2, Rect.Right - 2,
Rect.Bottom - 2);
```

Также можно использовать метод ***RoundRect TCanvas***, выводящий прямоугольник с закругленными краями.

```
pCanvas->RoundRect(Rect.Left + 2, Rect.Top + 2, Rect.Right - 2,
Rect.Bottom - 2, 2, 2);
```

Честно говоря, я предпочитаю именно метод ***RoundRect*** методу ***Rectangle***. Мне кажется, что так список выглядит симпатичнее, и в выпадающем списке, изображенном на рис. 1.5, был использован именно этот метод. И еще одна маленькая, но чрезвычайно важная деталь: элементы в списках будут отображаться корректно, если в их свойствах Items будут находиться 16 строк, представляющих 16 основных цветов:

- clAqua
- clBlack
- clBlue
- clFuchsia
- clGray
- clGreen
- clLime
- clMaroon
- clNavy
- clOlive
- clPurple
- clRed
- clSilver
- clTeal
- clWhite
- clYellow

Теперь загрузим тестовое приложение и испытаем наши списки в действии. Внешний вид формы тестового приложения изображен на рис. 1.7.

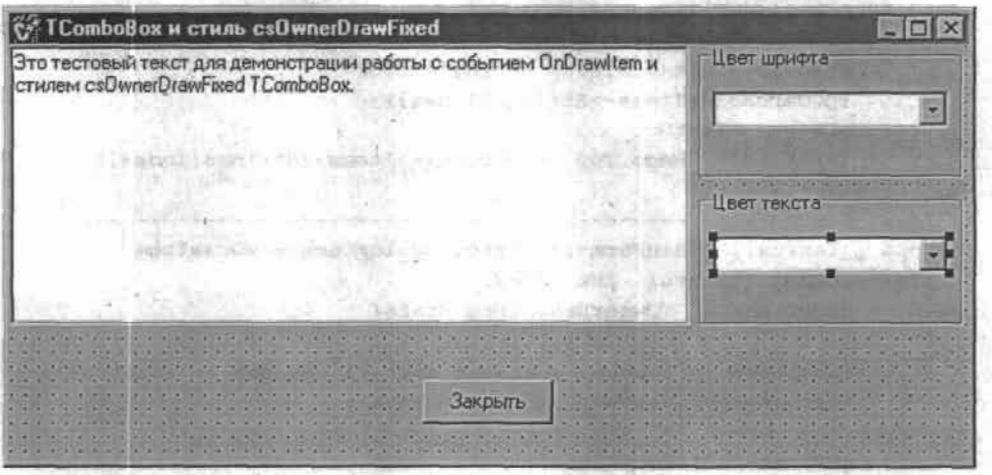


Рис. 1.7. *Внешний вид формы тестового проекта*

Код cpp-файла формы тестового проекта приведен ниже.

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "OwnerDrawFixedUnit.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------
void __fastcall TMainForm::FontColorComboBoxDrawItem
      (TWinControl *Control, int Index, TRect &Rect, TOwnerDrawState State)
   {
      TComboBox *pComboBox = static_cast <TComboBox *> (Control);
      TCanvas *pCanvas = pComboBox->Canvas;
```

```
        pCanvas->FillRect(Rect);

        pCanvas->Font->Color = StringToColor
        (pComboBox->Items->Strings[Index]);
        pCanvas->TextOut
        (Rect.Left, Rect.Top, pComboBox->Items->Strings[Index]);
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::BackgroundColorComboBoxDrawItem
(TWinControl *Control, int Index,
      TRect &Rect, TOwnerDrawState State)
    {
        TComboBox *pComboBox = static_cast <TComboBox *> (Control);
        TCanvas *pCanvas = pComboBox->Canvas;

        pCanvas->FillRect(Rect);
        pCanvas->Brush->Color =
        StringToColor(pComboBox->Items->Strings[Index]);
       pCanvas->RoundRect(Rect.Left + 2, Rect.Top + 2, Rect.Right - 2,
        Rect.Bottom - 2, 2, 2);
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::FormCreate(TObject *Sender)
    {
        FontColorComboBox->ItemIndex = 1;
        BackgroundColorComboBox->ItemIndex = 14;
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::CloseButtonClick(TObject *Sender)
    {
        Application->Terminate();
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::FontColorComboBoxChange(TObject *Sender)
    {
        TestMemo->Font->Color = StringToColor(FontColorComboBox->Text);
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::BackgroundColorComboBoxChange(TObject *Sender)
```

```
    {
      TestMemo->Color = StringToColor(BackgroundColorComboBox->Text);
    }
//---------------------------------------------------------------------
```

Обработчики события *OnDrawItem* рассмотрены выше, поэтому перейдем к остальному коду проекта.

В обработчике события *OnCreate* формы.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::FormCreate(TObject *Sender)
    {
      FontColorComboBox->ItemIndex = 1;
      BackgroundColorComboBox->ItemIndex = 14;
    }
//---------------------------------------------------------------------
```

мы указываем, какие элементы будут изначально выбраны в выпадающих списках. Заметьте, поскольку у выпадающих списков установлен стиль *csOwnerDrawFixed*, простым присваиванием желаемого значения свойству *Text* установить начальные значения списков невозможно.

В обработчиках события *OnChange* компонентов мы задаем новый цвет шрифта и фона *TestMemo* в соответствии с выбранным элементом выпадающего списка.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::FontColorComboBoxChange(TObject *Sender)
    {
      TestMemo->Font->Color = StringToColor(FontColorComboBox->Text);
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::BackgroundColorComboBoxChange
(TObject *Sender)
    {
      TestMemo->Color = StringToColor(BackgroundColorComboBox->Text);
    }
//---------------------------------------------------------------------
```

В них вновь для преобразования текста выбранного элемента списка в цветовое значение используется функция *StringToColor*.

Запустите тестовый проект (внешний вид приложения показан на рис. 1.8).

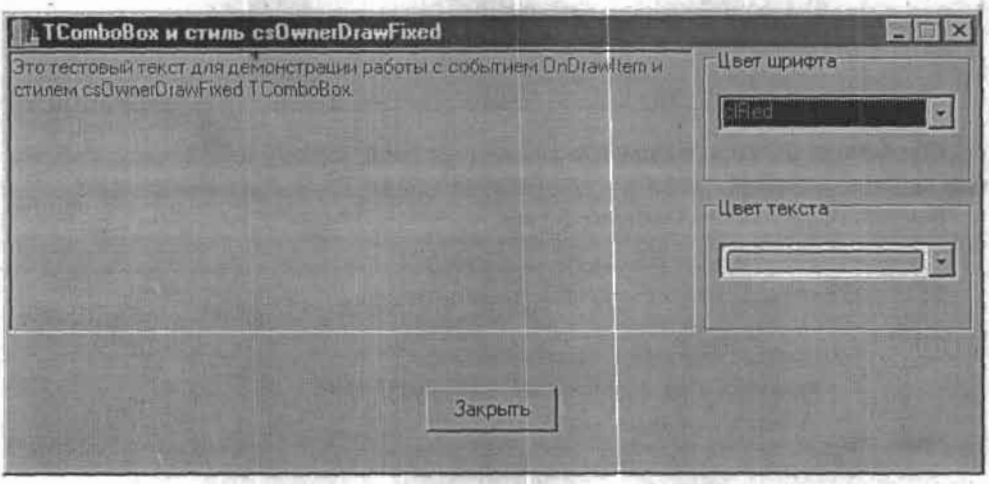


Рис. 1.8. *Внешний вид загруженного приложения с цветом TestMemo clSilver и цветом шрифта clRed*

Поиграйтесь с выпадающими списками цветами ***TestMemo***, убедитесь, что все функционирует именно так, как хотелось и задумывалось. Думаю, что неясностей с использованием стиля ***csOwnerDrawFixed*** у вас уже не осталось. Тогда – вперед! Следующая проблема, которую нам предстоит решить, – стиль ***csOwnerDrawVariable*** у ***TListBox***. Об этом – далее в нашей книге.

Глава 2

Некоторые из классов VCL

Сравнение строк по маске и использование TMask

Как можно было предположить, фирма Borland не обошла стороной и такой вопрос, как сравнение строк по маске. Для этих целей в VCL существует класс ***TMask***.

Класс ***TMask*** – это очень простой класс. Он унаследован непосредственно от ***TObject***. ***TMask*** не имеет свойств, а методов у него всего три.

Конструктор:

```
__fastcall TMask(const AnsiString MaskValue);
```

деструктор:

```
__fastcall virtual ~TMask(void);
```

и метод ***Matches***:

```
bool __fastcall Matches(const AnsiString Filename);
```

В конструкторе задается маска, с которой будет происходить сравнение строки, переданной в качестве параметра в метод ***Matches***. В маске допустимы буквенные символы, групповые символы (wildcards) и множества.

Буквенному символу в маске соответствует такой же символ в сравниваемой строке. То есть если в маске указан символ "7", то для соответствия данному символу в строке должен находиться символ "7" в той же позиции.

Теперь рассмотрим множества.

Каждое множество должно начинаться с открывающей прямоугольной скобки "[" (без кавычек, разумеется. И все далее встречаемые в тексте символы в кавычках в маске должны быть указаны без кавычек) и заканчиваться закрывающей прямоугольной скобкой "]". Внутри скобок находятся элементы множества, которые могут быть либо простым буквенным символом, либо диапазоном символов. Диапазоны символов задаются начальным значением, конечным значением и разделителем "-" (знак "минус" на клавиатуре) между ними.

Не нужно разделять элементы множества пробелами или запятыми. Исходя из вышесказанного, мы можем задать множество следующим образом.

[a-f18]

Множеству должен соответствовать один символ в задаваемой в методе ***Matches*** строке. Символ совпадает со множеством, если он лежит либо в пределах одного из указанных во множестве диапазонов, либо совпадает с любым из других символов множества. Для множества

[a-f18]

совпадающим с ним будут символы

a, b, c, d, e, f, 1, 8.

Граничные символы диапазона, как видно из примера, также входят в диапазон значений. Если первый символ во множестве, после открывающей прямоугольной скобки, является восклицательным знаком "!", то множеству в проверяемой строке соответствует любой символ, не перечисленный во множестве. Для множества

[!a-f18]

совпадающим с ним будет любой символ, кроме "a", "b", "c", "d", "e", "f", "1", "8".

Групповыми символами (wildcards) являются символы звездочки "*" и вопросительного знака "?". Правила соответствия для этих символов такие же, как и для имен файлов. Символу "*" соответствует любое количество любых символов. Символу "?" соответствует произвольный единичный символ.

Сравнение строки с маской является регистронезависимым. Если в маске указан символ "a", например, то в строке ему будет соответствовать как символ "a", так и "A".

В методе ***Matches*** указывается строка, для которой будет производиться сравнение с маской. ***Matches*** возвращает ***true***, если строка соответствует маске, заданной в конструкторе, и ***false***, если указанная строка маске не соответствует. В случае, если маска не соответствует синтаксису, ***Matches*** выбрасывает исключение.

И последнее замечание. Поскольку ***TMask*** унаследован от ***TObject***, его экземпляр необходимо создавать посредством оператора ***new***.

Теперь взгляните на код тестового проекта.

```
//---------------------------------------------------------------------
#include <Masks.hpp>
#include <vcl.h>
#pragma hdrstop

#include "TMaskUnit.h"
//---------------------------------------------------------------------
```

```
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
    : TForm(Owner)
    {
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::ExitButtonClick(TObject *Sender)
    {
      Application->Terminate();
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::CompareButtonClick(TObject *Sender)
    {
      TMask *Mask = new TMask(MaskEdit->Text);
      if(Mask->Matches(StringEdit->Text))
        Application->MessageBox("Строка совпадает с маской.",
"Информация",
        MB_OK | MB_ICONINFORMATION);
      else
        Application->MessageBox("Строка не совпадает с маской!",
        "Внимание!", MB_OK | MB_ICONWARNING);
    }
//---------------------------------------------------------------------
```

Данный код настолько прост, что не требует комментариев. Единственное, на что следует обратить внимание, – это строка

```
#include <Masks.hpp>
```

Не забывайте подключать модуль Masks.hpp, когда вы хотите использовать ***TMask***.

Screen и его использование

Наряду с визуальными и невизуальными компонентами фирма Borland предоставила в распоряжение программистов классы, работа которых "и опасна, и трудна, и на первый взгляд как будто не видна". Экземпляры таких классов вы никогда не будете создавать вручную самостоятельно – они всегда создаются автоматически при старте приложения. К таким классам относятся классы ***TScreen***, ***TApplication*** и ряд других.

Рассмотрим работу с ***TScreen*** – классом, который, как можно понять из названия, инкапсулирует в себе поведение экрана, или, иначе, рабочего стола компьютера.

Класс ***TScreen*** наследуется непосредственно от ***TComponent*** и имеет следующие свойства, методы и события.

Свойства

- ActiveControl
- ActiveCustomForm
- ActiveForm
- Cursor
- Cursors
- CustomFormCount
- CustomForms
- DataModuleCount
- DataModules
- DefaultIme
- DefaultKbLayout
- DesktopHeight
- DesktopLeft
- DesktopTop
- DesktopWidth
- Fonts
- FormCount
- Forms
- Height
- HintFont
- IconFont
- Imes
- MenuFont
- MonitorCount
- Monitors
- PixelsPerInch
- Width

Методы

- ~TScreen
- DisableAlign
- EnableAlign

- Realign
- ResetFonts
- TScreen

События

- OnActiveControlChange
- OnActiveFormChange

Большинство свойств класса ***TScreen*** являются свойствами только для чтения.

Вкратце о методах, ибо используют их довольно редко, да и использование не представляет никакой сложности.

Сперва необходимо сказать о конструкторе

```
__fastcall virtual TScreen(Classes::TComponent* AOwner);
```

и деструкторе

```
__fastcall virtual ~TScreen(void);
```

класса.

Вам необходимо помнить, что вы никогда, ни в каком случае не должны сами создавать объект класса ***TScreen***. Для доступа к свойствам и методам класса в каждом VCL-приложении есть глобальная переменная ***Screen***. Также никогда не должны разрушать глобальный объект ***Screen*** – он автоматически будет разрушен при завершении работы приложения.

Далее. Как вам известно, у форм в C++Builder есть свойство ***Align***, которое определяет, как форма будет выравниваться внутри экрана. Так вот, метод ***EnableAlign***

```
void __fastcall EnableAlign(void);
```

разрешает формам быть выровненными внутри экрана.

Метод ***DisableAlign***

```
void __fastcall DisableAlign(void);
```

запрещает формам быть выровненными внутри экрана. После вызова этого метода значения свойств ***Align*** у всех форм игнорируются до тех пор, пока не будет вызван метод ***EnableAlign***.

А метод ***Realign***

```
void __fastcall Realign(void);
```

заново позиционирует формы на экране согласно значениям их свойств ***Align***.

Метод ***ResetFonts***

```
void __fastcall Realign(void);
```

вызывается для обновления списка шрифтов, перечисленных в свойстве ***Fonts***. Вы не должны вызывать этот метод самостоятельно – он вызывается внутри класса ***TScreen*** в ответ на сообщения Windows об изменении списка доступных шрифтов.

Переходим к рассмотрению свойств и событий.

Свойство ***ActiveControl***

```
__property Controls::TWinControl* ActiveControl = {read=FActiveControl};
```

позволяет определить, какой оконный элемент управления в активной форме получает в настоящий момент фокус ввода. Данное свойство необходимо рассматривать вместе с событием ***OnActiveControlChange***

```
__property Classes::TNotifyEvent OnActiveControlChange =
{read=FOnActiveControlChange, write =FOnActiveControlChange};
```

Данное событие происходит, когда фокус ввода передается другому элементу управления. То есть, если вам необходимо отследить переход фокуса с одного элемента управления на другой, вы должны написать обработчик события ***OnActiveControlChange***. Как это делается, демонстрирует первый тестовый проект, внешний вид формы которого приведен на рис. 2.1.

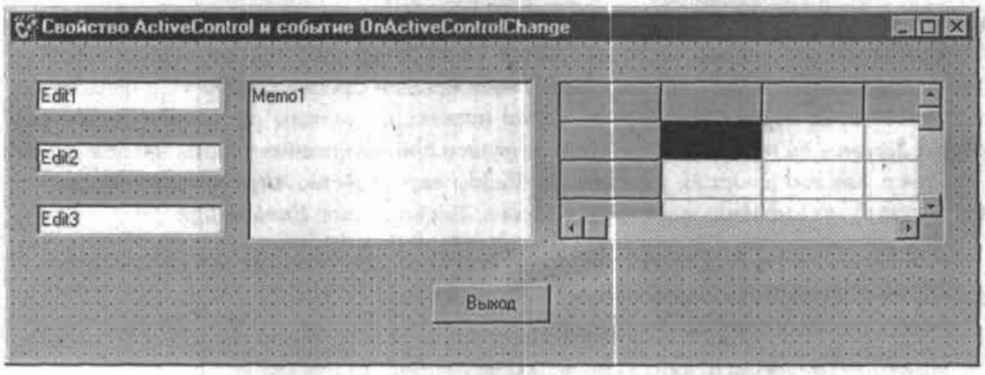


Рис. 2.1. *Внешний вид формы тестового проекта*

Запустите проект. При передаче фокуса другому элементу управления с помощью мыши или с помощью клавиши табуляции в заголовке формы будет отображено имя элемента управления, в котором в настоящий момент находится фокус ввода (см. рис. 2.2).

Это реализуется буквально парой строчек. В заголовочном файле в секцию ***private*** класса формы добавлено объявление функции, которая будет обработчиком события ***OnActiveControlChange***.

```
void __fastcall ActiveControlChange(TObject *Sender);
```

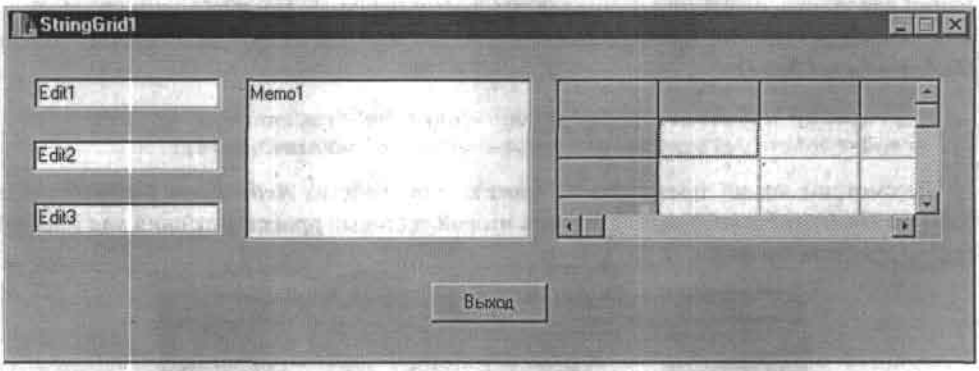


Рис. 2.2. *Имя элемента управления, в котором находится фокус, отображается в заголовке формы*

В обработчике события ***OnCreate*** формы мы назначаем данную функцию обработчиком события ***OnActiveControlChange***.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::FormCreate(TObject *Sender)
   {
      Screen->OnActiveControlChange = ActiveControlChange;
   }
//---------------------------------------------------------------------
```

Собственно, сама функция ***ActiveControlChange***.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::ActiveControlChange(TObject *Sender)
   {
      TWinControl *WinControl = Screen->ActiveControl;
      if(WinControl)
         Caption = WinControl->Name;
   }
//---------------------------------------------------------------------
```

Подобным же образом осуществляется работа со свойством ***ActiveForm*** и событием ***OnActiveFormChange***. Свойство ***ActiveForm***

```
__property TForm* ActiveForm = {read=FActiveForm};
```

определяет, какая из форм приложения в настоящий момент активна, то есть имеет фокус. Если само приложение в настоящий момент неактивно, то значение свойства определяет, какая форма станет активной при активации приложения. Обратите внимание, что ***ActiveForm*** – свойство

только для чтения, и; для того чтобы сделать форму активной, вам необходимо воспользоваться методом ***SetFocus***. При переходе фокуса с одной формы на другую возникает событие ***OnActiveFormChange***.

```
__property Classes::TNotifyEvent OnActiveFormChange =
{read=FOnActiveFormChange, write =FOnActiveFormChange};
```

Рассмотрим аналог предыдущего проекта для свойства ***ActiveForm*** и обработчика события ***OnActiveFormChange***. Загрузите второй тестовый проект. Внешний вид главной формы представлен на рис. 2.3.

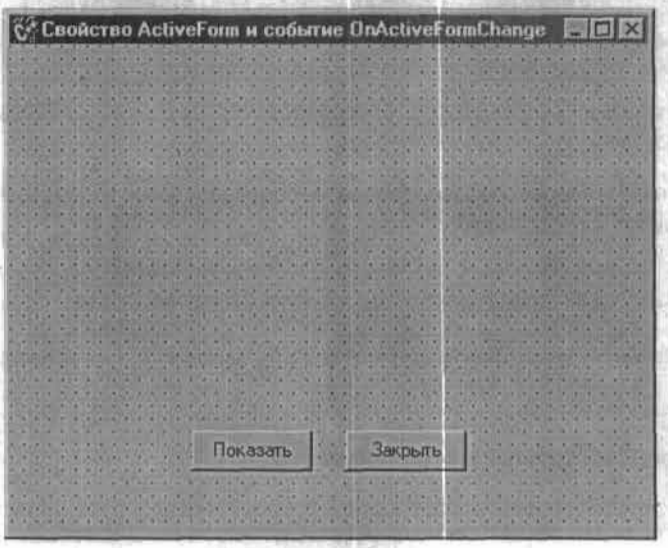


Рис. 2.3. *Внешний вид главной формы тестового проекта*

Как видно, сам проект чрезвычайно прост. При нажатии кнопки ***ShowFormsButton*** (с заголовком "Показать") отображаются остальные две формы проекта, и при переходе фокуса на другую форму на экран выводится сообщение с именем формы, которая в настоящий момент активна (см. рис. 2.4).

Код этого проекта очень похож на код предыдущего проекта. В секцию ***private*** класса главной формы приложения добавлено объявление функции ***ActiveFormChange***, которая будет обработчиком события ***OnActiveFormChange***.

```
private:        // User declarations
   void __fastcall ActiveFormChange(TObject *Sender);
```

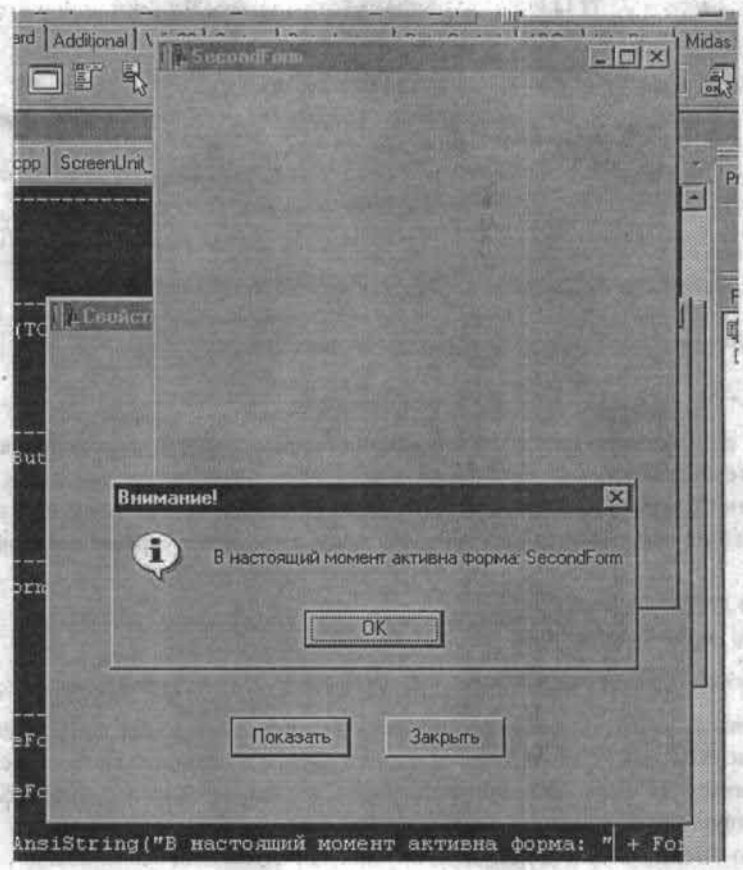


Рис. 2.4. *Внешний вид второго тестового проекта, иллюстрирующего работу со свойством ActiveForm и событием OnActiveFormChange*

А сам код главной формы, представляющий непосредственный интерес, выглядит следующим образом.

```
//---------------------------------------------------------------------
void __fastcall TFirstForm::ShowFormsButtonClick(TObject *Sender)
{
    SecondForm->Show();
    ThirdForm->Show();
}
//---------------------------------------------------------------------
```

```
void __fastcall TFirstForm::ActiveFormChange(TObject *Sender)
   {
      TForm *Form = Screen->ActiveForm;
      if(Form)
         Application->MessageBox(AnsiString("В настоящий момент
активна
         форма: " + Form->Name).c_str(), "Внимание!",
         MB_OK | MB_ICONINFORMATION);
   }
//---------------------------------------------------------------------
void __fastcall TFirstForm::FormCreate(TObject *Sender)
   {
      Screen->OnActiveFormChange = ActiveFormChange;
   }
//---------------------------------------------------------------------
```

Данный код настолько прост, что его даже нет необходимости комментировать.

Вот и все о событиях класса ***TScreen***. Сделаю лишь небольшое замечание. Событие ***OnActiveFormChange*** произойдет и в том случае, когда форма неактивна, но в ней вызывается метод ***SetFocusedControl*** для передачи фокуса в элемент управления этой неактивной формы.

Теперь о других свойствах.

Свойство ***ActiveCustomForm***

```
__property TCustomForm* ActiveCustomForm = {read=FActiveCustomForm};
```

позволяет узнать, какой из наследников класса ***TCustomForm*** имеет фокус в настоящий момент. К наследникам ***TCustomForm*** относятся формы и страницы свойств. Если активным наследником ***TCustomForm*** является форма, то данное свойство полностью аналогично свойству ***ActiveForm***.

Свойство ***Cursor***.

```
__property Controls::TCursor Cursor =
{read=FCursor, write=SetCursor, nodefault};
```

позволяет работать с курсором мыши на уровне приложения, а не на уровне отдельных компонентов. Если значение свойства ***Cursor*** равно ***crDefault***, то вид курсора мыши определяется индивидуально для каждого компонента формы согласно значению свойства ***Cursor*** данного компонента. Если же значение свойства ***Cursor*** отлично от ***crDefault***, то данный курсор будет отображаться для всех окон и элементов управления данного приложения, вне зависимости от значения свойства ***Cursor*** отдельных элементов управления. Установленный через свойство ***Cursor*** курсор будет таковым до тех пор, пока значение свойства вновь не будет установлено в ***crDefault***. Возможно, что изменение значения

свойства ***Cursor*** не будет показано немедленно. В этом случае необходимо вызвать метод ***Application::ProcessMessages***, для того чтобы приложение откликнулось на изменение курсора. Чаще всего свойство ***Cursor*** используют следующим образом.

```
TCursor OldCursor = Screen->Cursor;
Screen->Cursor = crHourGlass;  // отображаем курсор "песочные часы"
try
  {
    // продолжительные по времени операции
  }
__finally
  {
    Screen->Cursor = OldCursor; // восстанавливаем исходный курсор
  }
```

Восстановление исходного курсора обязательно.

Свойство ***Cursors***

```
__property HICON Cursors[int Index] = {read=GetCursors, write=SetCursors};
```

представляет собой индексированный список курсоров, которые вы можете использовать либо для всего приложения, либо для отдельного элемента управления в приложении. Класс ***TScreen*** содержит ряд встроенных курсоров, которые проиндексированы символическими константами.

Табл. 2.1. *Поименованные константы и их значения*

Константа	Значение
crDefault	0
crNone	-1
crArrow	-2
crCross	-3
crIBeam	-4
crSizeNESW	-6
crSizeNS	-7
crSizeNWSE	-8
crSizeWE	-9
crUpArrow	-10
crHourGlass	-11
crDrag	-12
crNoDrop	-13

Табл. 2.1. Поименованные константы и их значения (Продолжение)

Константа	Значение
crHSplit	-14
crVSplit	-15
crMultiDrag	-16
crSQLWait	17
crNo	18
crAppStart	-19
crHelp	-20
crHandPoint	-21
crSize (устаревшая)	-22
crSizeAll	-22

Также вы можете с помощью функции Windows API ***LoadCursor*** загрузить свой собственный курсор для использования в приложении. В этом случае по окончании использования данного курсора вы не должны вызывать API-функцию ***DestroyCursor***. C++Builder сделает это автоматически.

Свойство ***CustomFormCount***

```
__property int CustomFormCount = {read=GetCustomFormCount, nodefault};
```

позволяет вам узнать количество форм и страниц свойств в приложении. Чаще всего это свойство используется совместно с индексированным свойством ***CustomForms***

```
__property TCustomForm* CustomForms[int Index] = {read=GetCustomForms};
```

которое позволяет получить доступ ко всем формам и страницам свойств в приложении по индексу. Например, список всех наследников ***TCustomForm*** в приложении получают следующим образом.

```
TStringList *FormList = new TStringList();

for (int i = 0; i < Screen->CustomFormCount; i++)
    FormList ->Add(Screen->CustomForms[i]->Name);
```

После выполнения данного кода объект ***FormList*** класса ***TStringList*** будет содержать в себе список всех форм и страниц свойств в приложении.

Абсолютно аналогичны свойствам ***CustomFormCount*** и ***CustomForms*** свойства ***DataModuleCount*** и ***DataModules***.

```
__property int DataModuleCount = {read=GetDataModuleCount, nodefault};
__property TDataModule* DataModules[int Index] = {read=GetDataModule};
```

Свойство ***DataModuleCount*** позволяет узнать количество модулей данных (экземпляров класса ***TDataModule***) в приложении, а свойство ***DataModules*** – получить доступ к конкретному экземпляру по индексу.

Свойство ***DefaultKbLayout***

```
__property HKL DefaultKbLayout = {read=FDefaultKbLayout, nodefault};
```

позволяет получить дескриптор раскладки клавиатуры, которая была активна в момент старта приложения. Данный дескриптор нужен в вызовах функций Windows API, работающих с раскладкой клавиатуры.

Два свойства, ***DesktopHeight*** и ***DesktopWidth***,

```
__property int DesktopHeight = {read=GetDesktopHeight, nodefault};
__property int DesktopWidth = {read=GetDesktopWidth, nodefault};
```

позволяют узнать высоту и ширину рабочего стола по отношению к верхнему левому углу основного монитора (не забываем о многомониторных конфигурациях). Высота и ширина рабочего стола измеряются в пикселах. В одномониторных конфигурациях данные свойства аналогичны свойствам ***Height*** и ***Width*** соответственно.

Пара свойств, ***DesktopLeft*** и ***DesktopTop***,

```
__property int DesktopLeft = {read=GetDesktopLeft, nodefault};
__property int DesktopTop = {read=GetDesktopTop, nodefault};
```

позволяет узнать x- и y-координаты левой и верхней границы рабочего стола относительно верхнего левого угла основного монитора соответственно. В одномониторных конфигурациях данные свойства аналогичны свойствам ***Left*** и ***Top***.

Свойство ***Fonts***

```
__property Classes::TStrings* Fonts = {read=GetFonts};
```

одно из наиболее часто используемых свойств класса ***TScreen***. Оно позволяет получить список имен шрифтов, установленных в системе. Заметьте, что свойство ***Fonts*** представляет только экранные шрифты – не шрифты принтера.

Свойство ***FormCount***

```
__property int FormCount = {read=GetFormCount, nodefault};
```

указывает количество форм приложения, отображаемых в настоящий момент на экране. Совместно со свойством ***Forms***

```
__property TForm* Forms[int Index] = {read=GetForm};
```

позволяет пройтись по всему списку отображаемых в приложении форм.

```
TStringList *FormList = new TStringList();

for (int i = 0; i < Screen->FormCount; i++)
    FormList ->Add(Screen->Forms[i]->Name);
```

Свойство ***Height***

```
__property int Height = {read=GetHeight, nodefault};
```

определяет высоту экрана в пикселах.

Свойство ***Width***

```
__property int Width = {read=GetWidth, nodefault};
```

определяет ширину экрана в пикселах. Таким образом, комбинация свойств ***Width*** и ***Height*** определяет, по сути, текущее разрешение экрана.

В дополнение к этим двум свойствам есть еще свойство ***PixelsPerInch***

```
__property int PixelsPerInch = {read=FPixelsPerInch, nodefault};
```

которое позволяет узнать количество пикселов на дюйм монитора. Также данное свойство может использоваться при различных измерениях и преобразованиях между логическими пикселами и логическими единицами длины, но надо помнить, что значение данного свойства точно только для вертикальных измерений и преобразований, так как разные мониторы имеют разный масштабный коэффициент по горизонтали.

Свойство ***MonitorCount***

```
__property int MonitorCount = {read=GetMonitorCount, nodefault};
```

и свойство

```
__property TMonitor* Monitors[int Index] = {read=GetMonitor};
```

полезны в многомониторной конфигурации. ***MonitorCount*** позволяет получить количество мониторов, которые составляют рабочий стол, а свойство ***Monitors*** позволяет обратиться к каждому монитору по индексу. Основной монитор имеет индекс, равный нулю: Screen- > Monitors[0].

Вкратце о свойствах все. В принципе этой информации должно быть достаточно, чтобы при работе с ***TScreen*** не испытывать затруднений. Перейдем теперь к последнему, третьему тестовому проекту, где будет на практике продемонстрировано использование свойств ***TScreen***.

Внешний вид главной формы третьего тестового проекта показан на рис. 2.5.

Рис. 2.5. *Главная форма третьего тестового проекта*

Запустив приложение и нажав кнопку "Старт", вы увидите примерно следующую картину.

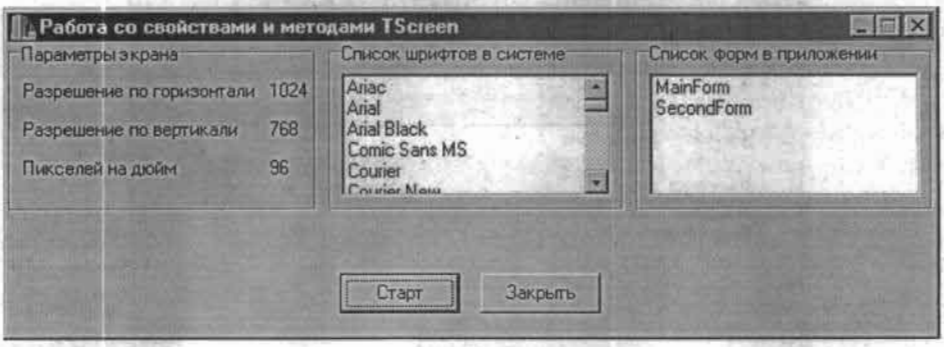


Рис. 2.6. *Вид запущенного приложения после нажатия кнопки "Старт"*

Все происходит в обработчике события ***OnClick*** кнопки "Старт".

```
//---------------------------------------------------------------------
void __fastcall TMainForm::StartButtonClick(TObject *Sender)
   {
      // получаем параметры экрана
      HorizResLabel->Caption = IntToStr(Screen->Width);
      VertResLabel->Caption = IntToStr(Screen->Height);
      PixelsPerInchLabel->Caption = IntToStr(Screen->PixelsPerInch);
```

```
    //получаем список шрифтов, установленных в системе
    FontListBox->Items->Assign(Screen->Fonts);

    // получаем список открытых форм в приложении
    FormListBox->Items->Clear();
    for(int i = 0; i < Screen->FormCount; i++)
       FormListBox->Items->Add(Screen->Forms[i]->Name);
}
//---------------------------------------------------------------------
```

Как видите, обработчик донельзя прост.

И напоследок необходимо отметить одну важную деталь. Обратите внимание на содержимое списка форм в проекте. Там кроме главной формы с именем ***MainForm*** отображена и другая форма проекта – ***SecondForm***. И она находится в этом списке, несмотря на то что не отображается на экране. А происходит это из-за того, что данная форма находится в списке автосоздаваемых форм проекта.

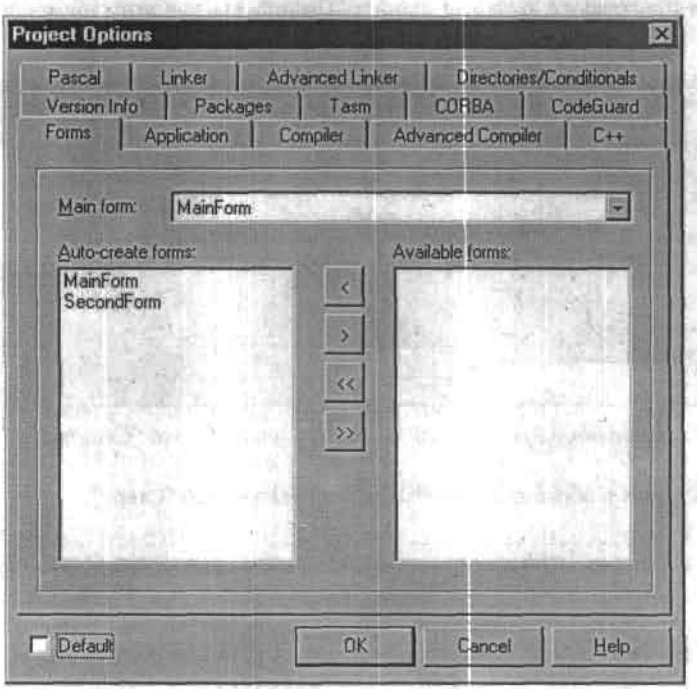


Рис. 2.7. *Список автосоздаваемых форм проекта*

То есть форма ***SecondForm*** создается при старте приложения, а не отображается на экране, поскольку после создания ее свойство ***Visible*** установлено в ***false***. И любая форма, создаваемая автоматически, будет перечисляться в свойстве ***Forms*** (и ***CustomForms***). Если переместить ***SecondForm*** из списка автосоздаваемых форм в список доступных форм, то форма при старте приложения создаваться не будет и в списке форм нашего приложения она также не будет фигурировать. Но для того чтобы отобразить такую форму, вам ее будет необходимо создать вручную.

Немного о TMonitor

В C++Builder есть класс, который большинству программистов даже не известен. Это класс ***TMonitor***. Столкнуться с ним вы могли, используя свойство ***Monitors*** класса ***TScreen***.

```
__property TMonitor* Monitors[int Index] = {read=GetMonitor};
```

Данное индексированное свойство позволяет получить доступ к любому из установленных в системе мониторов по индексу.

Класс ***TMonitor*** имеет следующие методы:

- ~TMonitor
- TMonitor

и свойства:

- Handle
- Height
- Left
- MonitorNum
- Top
- Width

Конструктор и деструкторы ***TMonitor*** не представляют собой никакого интереса.

```
inline __fastcall TMonitor(void) : System::TObject() { }
__fastcall virtual ~TMonitor(void);
```

Вы никогда не должны создавать экземпляр ***TMonitor*** самостоятельно и никогда не должны разрушать объект ***TMonitor***. Для получения доступа к конкретному монитору вам необходимо пользоваться свойством ***Monitors*** класса ***TScreen***.

Выражение Screen->Monitors[0] определяет основной монитор.

Свойство ***Width***

```
__property int Width = {read=GetWidth, nodefault};
```

определяет ширину монитора в пикселах.

Свойство ***Height***

```
__property int Height = {read=GetHeight, nodefault};
```

определяет высоту монитора в пикселах.

Таким образом, совокупность свойств ***Width*** и ***Height*** определяет текущее разрешение монитора. Также эти свойства полезны при использовании в многомониторных конфигурациях в совокупности со свойствами ***Left*** и ***Top***.

```
__property int Left = {read=GetLeft, nodefault};
__property int Top = {read=GetTop, nodefault};
```

Свойство ***Left*** определяет координаты левой границы монитора на общем для нескольких мониторов рабочем столе. Свойство ***Top*** определяет координаты верхней границы монитора на общем для нескольких мониторов рабочем столе. Таким образом, использование свойств ***Left***, ***Top***, ***Width*** и ***Height*** позволяет точно позиционировать элемент управления на рабочем столе в многомониторных системах. В системе с одним монитором, где рабочий стол распространен только один монитор, значения свойств ***Left*** и ***Top*** равны нулю.

Свойство ***MonitorNum***

```
__property int MonitorNum = {read=FMonitorNum, nodefault};
```

определяет номер монитора в системе. Нумерация мониторов начинается с нуля. Для первого монитора значение свойства равно нулю, для второго – единица и т. д. В системах с одним монитором значение этого свойства всегда равно нулю.

И наконец, последнее свойство класса ***TMonitor*** – свойство ***Handle***.

```
__property HMONITOR Handle = {read=FHandle, nodefault};
```

Это свойство определяет дескриптор монитора. Это может показаться необычным, но каждый монитор в Windows имеет свой дескриптор. И дескриптор этот можно использовать в вызовах функций Windows API, например в функции ***GetMonitorInfo***.

Функция ***GetMonitorInfo*** объявлена в Winuser.h следующим образом:

```
BOOL GetMonitorInfo(
  HMONITOR hMonitor, // дескриптор монитора
  LPMONITORINFO lpmi // информация о мониторе
);
```

и может использоваться в версиях Windows 98/2000 и выше.

В функцию первым параметром передается дескриптор монитора, а вторым – указатель на структуру типа ***MONITORINFO*** или ***MONITORINFOEX***.

К сожалению, никакой дополнительной информации о мониторе с помощью данной функции и структуры ***MONITORINFO*** по сравнению с использованием свойств класса ***TMonitor*** мы не получим. Но если мы воспользуемся структурой ***MONITORINFOEX***, то, прочитав значение ее члена данных ***szDevice***, сможем еще получить название монитора. Код будет приведен ниже для всего проекта.

Теперь давайте рассмотрим тестовое приложение, в котором демонстрируется работа с ***TMonitor***. Внешний вид формы проекта приведен на рис. 2.8.

Рис. 2.8. *Форма тестового приложения в design-time*

После запуска приложения и нажатия кнопки "Старт" у вас на экране должно быть изображение, похожее на рис. 2.9.

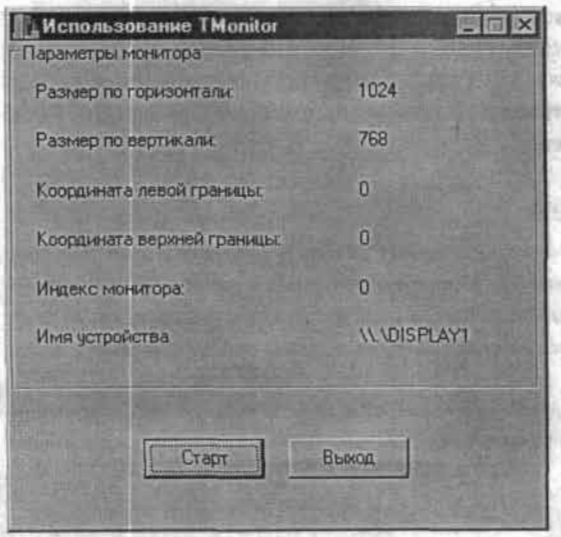


Рис. 2.9. *Вид запущенного тестового приложения*

Код, выполняющий всю работу с ***TMonitor***, расположен в обработчике события ***OnClick*** кнопки ***StartButton***.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::StartButtonClick(TObject *Sender)
   {
      WidthLabel->Caption = IntToStr(Screen->Monitors[0]->Width);
      HeightLabel->Caption = IntToStr(Screen->Monitors[0]->Height);
      LeftLabel->Caption = IntToStr(Screen->Monitors[0]->Left);
      TopLabel->Caption = IntToStr(Screen->Monitors[0]->Top);
      MonitorNumLabel1->Caption =
      IntToStr(Screen->Monitors[0]->MonitorNum);

      // используем функцию GetMonitorInfo
      MONITORINFOEX MonitorInfo;
      MonitorInfo.cbSize = sizeof(MONITORINFOEX);
      GetMonitorInfo(Screen->Monitors[0]->Handle, &MonitorInfo);
      MonitorNameLabel->Caption = MonitorInfo.szDevice;
   }
//---------------------------------------------------------------------
```

Вам необходимо обратить внимание на то, что для компьютера с одним монитором (именно такой ситуации соответствует рис. 2.9) использование свойств ***Width*** и ***Height*** класса ***TMonitor*** аналогично использованию свойств ***Width*** и ***Height*** класса ***TScreen***. Также заметьте, что ***MonitorInfo.szDevice*** определяет не название монитора как таковое (типа Samsung 755 NF), а название монитора как устройства: \\.\DISPLAY1.

И напоследок. ***GetMonitorInfo*** не единственная функция для работы с мониторами в Windows API, разумеется. В Windows API есть целый раздел, посвященный работе с мониторами и многомониторными системами. Для дополнительной информации смотрите раздел "Multiple Display Monitors Reference".

Неизвестный TLanguages

Модуль Sysutils.hpp богат на различные полезные и удобные функции и классы. Этот модуль, пожалуй, одна из наиболее удавшихся Borland и полезных в работе программиста на C++Builder вещей. Однако, как показывает опыт общения с разработчиками, многим из них известно в этом модуле далеко не все. Так давайте восполним пробелы и рассмотрим малоизвестный класс ***TLanguages***.

Говоря откровенно, данный класс может пригодиться в работе отнюдь не всем разработчикам, что, однако, никак не противоречит тому, что об этом классе надо знать, благо это знание не обременит.

Класс ***TLanguages*** предназначен для получения списка локалей, доступных в системе. Сразу надо оговориться, что "локаль" (locale) и "раскладка клавиатуры" (keyboard layout) – вещи разные и не взаимозаменяемые. Понятие "локаль" шире понятия "раскладка клавиатуры" и включает в себя собственно раскладку клавиатуры, а также, кроме нее, настройки даты, времени, десятичного разделителя, денежного знака и еще ряд других специальных параметров. Так вот, ***TLanguages*** получает именно список доступных локалей. Данная информация берется непосредственно из операционной системы.

TLanguages унаследован непосредственно от ***TObject*** и имеет следующие свойства:

- Count
- Ext
- ID
- LocaleID
- Name
- NameFromLCID
- NameFromLocaleID

Свойство ***Count*** представляет собой количество доступных локалей.

Свойство ***Ext*** представляет собой индексированное свойство для получения стандартного трехбуквенного расширения для локали. Для английского языка (американского), например, расширение будет представлять собой сочетание "ENU", для русского "RUS", для итальянского "ITA" и т. д.

Свойство ***ID*** представляет собой индексированное свойство для получения идентификатора локали в виде строки.

Свойство ***LocaleID*** представляет собой индексированное свойство для получения идентификатора локали в виде целого числа.

Свойство ***Name*** – это опять же индексированное свойство, возвращающее стандартное имя Windows для локали.

Свойства ***NameFromLCID*** и ***NameFromLocaleID*** менее полезны, но тоже интересны. ***NameFromLCID*** предназначено для получения имени локали по строковому представлению ее идентификатора (это то, что мы получаем, читая свойство ID). ***NameFromLocaleID*** делает то же, что и ***NameFromLCID***, но для числового идентификатора локали (это то, что мы получаем, читая свойство ***LocaleID***).

Обратите внимание, что все свойства класса ***TLanguages*** имеют доступ только для чтения (read-only). Также обратите внимание, что все свойства, за исключением ***NameFromLCID*** и ***NameFromLocaleID***, являются индексированными.

Теперь о методах. За исключением конструктора и деструктора, у ***TLanguages*** всего лишь один метод: ***IndexOf***. Он возвращает индекс локали по ее идентификатору. То есть предположим, что итальянская локаль перечислена третьей в ***TLanguages***. Тогда вызов

```
IndexOf(0x0410)
```

вернет 2. (Индексация начинается с нуля, об этом не забываем.)

При отсутствии локали с указанным числовым идентификатором метод ***IndexOf*** вернет –1.

Перед собственно написанием кода осталось сказать последнее: нет необходимости создавать экземпляр ***TLanguages*** через оператор ***new***. Для этого есть специальная функция, ***Languages()***, также объявленная в Sysutils.hpp и возвращающая указатель на объект ***TLanguages***.

Теперь попробуем применить полученные знания на практике. Загрузите тестовый проект в C++Builder. На экране должно появиться нечто похожее на приведенное изображение.

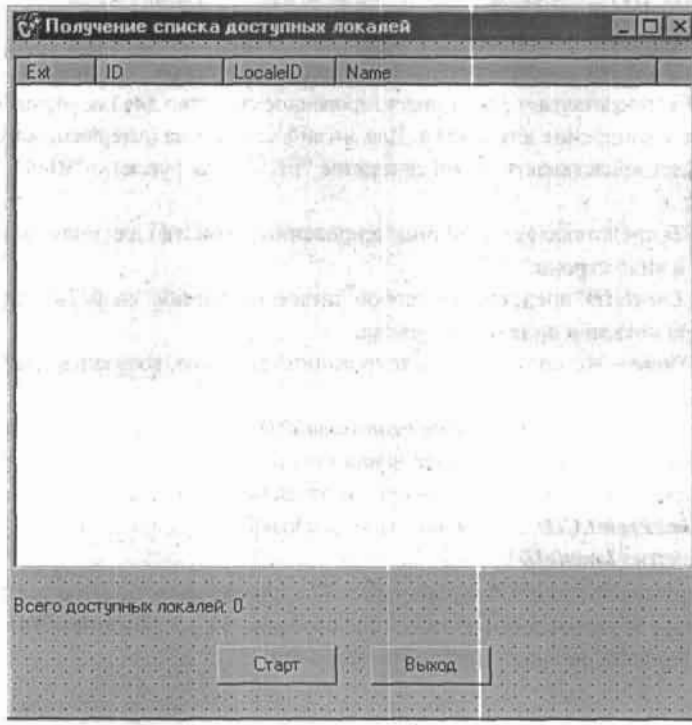


Рис. 2.10. *C++Builder с загруженным в него тестовым проектом*

Запустите проект и нажмите кнопку "Старт". Вот то, что у вас должно получиться.

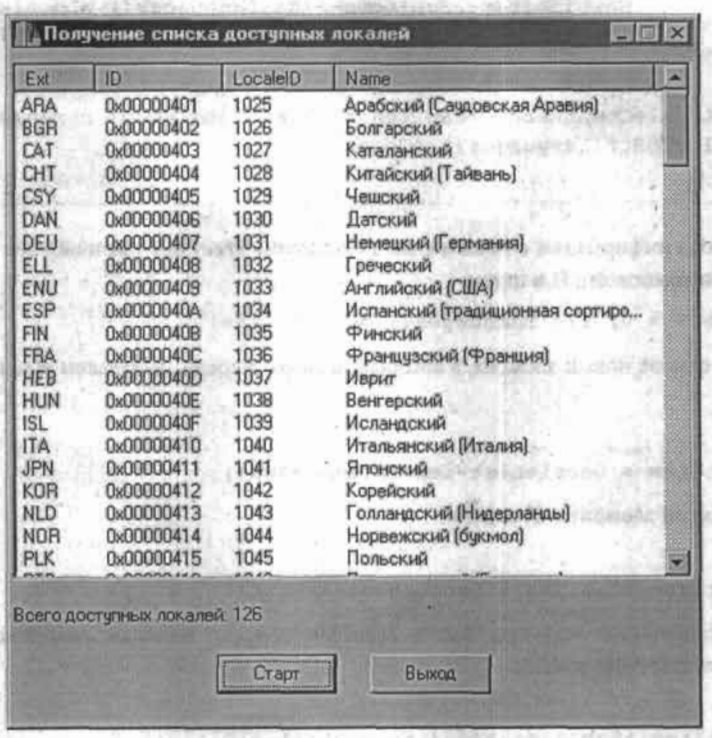


Рис. 2.11. *Запущенный проект*

Теперь разберемся, как это сделано. Обработчик события ***OnClick*** кнопки ***StartButton*** представляет собой следующее.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::StartButtonClick(TObject *Sender)
   {
      TListItem *NewListItem;

      LocaleListView->Items->Clear();

      for(int i = 0; i < Languages()->Count; i++)
         {
            NewListItem = LocaleListView->Items->Add();
            NewListItem->Caption = Languages()->Ext[i];
```

```
            NewListItem->SubItems->Add(Languages()->ID[i]);
            NewListItem->SubItems->Add(Languages()->LocaleID[i]);
            NewListItem->SubItems->Add(Languages()->Name[i]);
        }

    LocaleCountLabel->Caption = "Всего доступных локалей: " +
    IntToStr(Languages()->Count);
}
//---------------------------------------------------------------------
```

Для вывода информации о локалях мы используем *TreeView* как наиболее подходящий для этой цели компонент. И в цикле

```
for(int i = 0; i < Languages()->Count; i++)
```

добавляем в список новый элемент, в который, в свою очередь, добавляем всю информацию о локалях.

Строка

```
NewListItem = LocaleListView->Items->Add();
```

добавляет новый элемент в *TreeView*.

Строка

```
NewListItem->Caption = Languages()->Ext[i];
```

формирует содержимое колонки "Ext" в *TreeView*: заносит в новый элемент списка трехбуквенное расширение локали.

Строка

```
NewListItem->SubItems->Add(Languages()->ID[i]);
```

формирует содержимое колонки "ID" в *TreeView*: заносит в новый элемент списка строковый идентификатор локали.

Строка

```
NewListItem->SubItems->Add(Languages()->LocaleID[i]);
```

формирует содержимое колонки "LocaleID" в *TreeView*: заносит в новый элемент списка числовой идентификатор локали. Обратите внимание, что строковый идентификатор представлен в шестнадцатеричном виде, а числовой – в десятичном. Для представления числового идентификатора также в шестнадцатеричном виде строку

```
NewListItem->SubItems->Add(Languages()->LocaleID[i]);
```

надо заменить на

```
NewListItem->SubItems->Add(IntToHex(int(Languages()->LocaleID[i]), 4));
```

И наконец, последняя строка в цикле

```
NewListItem->SubItems->Add(Languages()->Name[i]);
```

заполняет четвертую колонку, "Name", названием данной локали, занося данное название в новый элемент списка.

Вот, собственно, и все, что надо знать о классе ***TLanguages***. Как говорится, "простенько и со вкусом", ибо незачем изобретать велосипед там, где он уже изобретен ранее. Теперь, по крайней мере, когда вам понадобится получить список локалей и краткую информацию о них, вы сделаете это в течение нескольких минут.

Использование TAction в C++Builder

Одним из эффективных инструментов для централизованного управления кодом является действие (***TAction***). Обоснованное применение ***TAction*** значительно упрощает программирование пользовательского интерфейса.

Судите сами. Современные концепции пользовательского интерфейса предполагают, что пользователь может выполнять одно и то же действие различными способами: через пункт меню (главного и/или всплывающего), кнопку на инструментальной панели, нажатие комбинации клавиш и т. д. В зависимости от состояния программы, это действие в каждый момент времени может быть доступно или недоступно. Естественно, элементы интерфейса (пункты меню, кнопки) должны адекватно отражать доступность или недоступность действия. Класс ***TAction***, обеспечивая связь между действием и элементами интерфейса, сводит весь процесс управления доступностью/недоступностью к строкам:

```
MyAction->Enabled = true;
```

или

```
MyAction->Enabled = false
```

Для того чтобы привязать конкретное действие конкретному элементу или группе элементов управления, достаточно свойству ***Action*** каждого элемента присвоить требуемое действие. Данное свойство присутствует у всех компонентов, порожденных от ***TControl***. Причем одно и то же действие можно сопоставить одновременно нескольким элементам управления. Операцию сопоставления можно произвести как в дизайн-режиме, так и в режиме выполнения посредством внесения соответствующего кода. Например, это может выглядеть следующим образом.

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    TAction* Action = new TAction(this);
    ExitBtn->Action =  Action;
    ExitMenu->Action = Action;
}
```

Однако если попытаться оттранслировать и запустить приложение в таком виде, то ни выполнить пункт меню, ни нажать кнопку не удастся, так как меню и кнопка будут заблокированы. Более того, на меню и кнопке будут отсутствовать надписи, даже если свойства ***Caption*** и были заполнены у этих элементов управления в дизайн-режиме. Это связано с тем, что свойство ***Caption*** не было заполнено у самого экземпляра ***TAction***. Стоит нам только заполнить это свойство у экземпляра ***TAction***, оно автоматически обновится и у всех элементов управления. Это касается не только ***Caption***. Автоматически также синхронизируются свойства ***Checked***, ***Enabled***, ***HelpContext***, ***Hint***, ***ImageIndex***, ***ShortCut*** и ***Visible***, если они, конечно, присутствуют у конкретного элемента управления. Все дальнейшие изменения, произведенные с этими свойствами, автоматически будут отражаться и на всех элементах управления, ассоциированных с данным экземпляром ***TAction***. Таким образом, в нашем распоряжении оказывается механизм централизованного управления не только действиями, но состоянием и содержанием свойств элементов управления. Теперь нет необходимости по отдельности присваивать соответствующее содержание ***Caption*** для пункта меню и кнопки. Вполне достаточно выполнить присвоение только для экземпляра ***TAction***.

```
Action->Caption = "Выполнить";
```

Теперь при выполнении программы и кнопка, и пункт меню будут иметь названия, но по-прежнему будут недоступны. Дело в том, что, связав эти элементы управления с ***TAction***, мы не задали того, что, собственно, должно выполняться. У ***TAction***, кроме свойств, имеются еще и три события: ***OnExecute***, ***OnHint***, ***OnUpdate***. Из этих трех событий именно ***OnExecute*** определяет обработчик, который будет выполняться при задействовании элемента управления. Зададим этот обработчик. Пускать при его выполнении будет выводиться сообщение "Выполнено".

```
void __fastcall TForm1::ExitActionExecute(TObject *Sender)
  {
    ShowMessage("Выполнено");
}
```

Модифицируем код тела конструктора.

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
  {
    TAction* Action = new TAction(this);
    ExitBtn->Action = Action;
    ExitMenu->Action = Action;
    Action->Caption = "Выполнить";
    Action->OnExecute = ExecActionExecute;
  }
```

После этих изменений и пункт меню, и кнопка при работе программы будут доступны для выполнения. Немного усложним задачу и наложим дополнительные ограничения, при которых пользователь может воспользоваться меню и кнопкой. Например, сообщение "Выполнено" будет появляться, если пользователь заполнил поля Login и Password, которые являются экземплярами компонента *TEdit*. Наиболее простое и напрашивающееся решение – вставить в обработчик проверку и вывести поясняющее сообщение в случае, если необходимые условия не были выполнены.

```
void __fastcall TForm1::ExecActionExecute(TObject *Sender)
  {
    if(Login->Text.IsEmpty() || Password->Text.IsEmpty())
      {
        ShowMessage("Логин и пароль должны быть заполнены");
        return;
      }
    ShowMessage("Выполнено");
  }
```

В этом случае пользователь, чтобы выяснить, что он сделал что-то неправильно, должен сначала попытаться выполнить действие и затем получить разъяснение о том, что он сделал не так. *TAction* предоставляет возможность реализовать другой подход к решению данной задачи: запретить возможность выполнить пункт меню или нажать кнопку, пока оба поля не будут заполнены. Для этого необходимо воспользоваться другим событием – *TAction::OnUpdate*. Это событие запускается либо во время "холостого кода" (idle) программы, либо при обновлении списка *TActionList*. Обработчику этого события можно вполне предоставить слежение за доступностью действия в конкретный момент времени. Код обработчика достаточно прост.

```
void __fastcall TForm1::ExecActionUpdate(TObject *Sender)
  {
    ((TAction*)Sender)->Enabled =
    !Login->Text.IsEmpty() && !Password->Text.IsEmpty();
  }
```

Добавим в конструктор еще одну строчку.

```
Action->OnExecute = ExecActionUpdate;
```

И опять запустим программу. Пока хотя бы одно из полей остается незаполненным, пункт меню и кнопка будут недоступными. Но как только пользователь внесет информацию в оба поля, меню и кнопка будут тотчас же разблокированы и пользователь сможет воспользоваться ими.

Пока все действия по программированию ***TAction*** были выполнены посредством написания кода, хотя выше было упомянуто, что определенные манипуляции можно выполнить и в дизайн-режиме. Сам ***TAction*** является дочерним классом ***TComponent***, но разработчики VCL-библиотеки решили не присваивать ему своей иконки, поэтому в палитре компонентов вы его не найдете. Работа с ***TAction*** в дизайн-режиме осуществляется не непосредственно, а через компонент ***TActionList***. Этот компонент представляет собой контейнер, содержащий список действий, и предназначен для организации работы с действиями в дизайн-режиме и для организации централизованного управления действиями. Например, для выполнения групповых операций типа

```
for (int i = ActionList->ActionCount; i--; )
  {
    TAction* Action = (TAction*) ActionList->Actions[i];
    if (Action->Category == AnsiString("Category1"))
      Action->Enabled = false;
}
```

Компонент ***TActionList*** расположен на вкладке "Standard" в палитре компонентов, и если им воспользоваться, то весь код, расположенный в теле конструктора, становится ненужным. Для этого разместим компонент ***TActionList*** на форме. Через всплывающее меню вызовем Action List Edtior и опять же через всплывающее меню создадим новый экземпляр TAction (рис. 2.12).

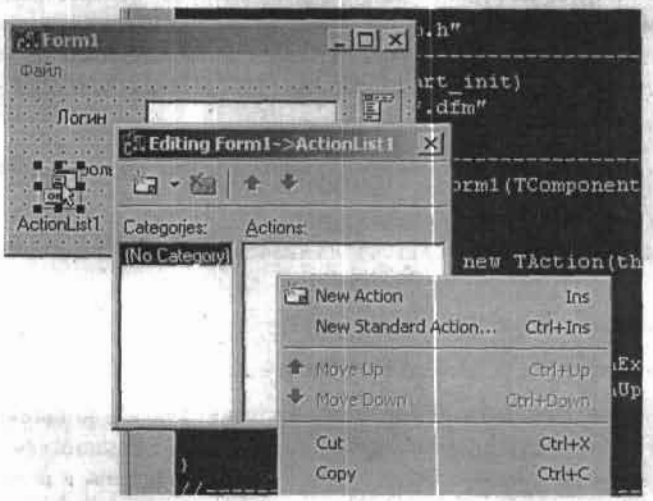


Рис. 2.12. *Создание экземпляра TAction в дизайн-режиме*

Затем, используя Object Inspector, отредактируем свойства и события ***Action***: заполним свойство ***Caption*** и присвоим событиям ***OnExecute*** и ***OnUpdate*** соответствующие обработчики. Также через Object Inspector у кнопки и пункта меню заполним свойство ***Action***, выбрав имя из выпадающего списка. После этого можно удалить весь код, набранный в теле конструктора, при этом ничего не потеряв в работоспособности и функциональности программы.

Остановимся подробнее на механизме взаимодействия ***Action*** и элементов управления. Внешне логика действий выглядит довольно простой: когда срабатывает элемент управления (нажатие кнопки, выбор пункта меню), вызывается метод ***TAction::Execute()***, который возбуждает событие ***TAction::OnExecute***, где и выполняется пользовательский код. В большинстве случаев, когда программист использует для задания действия класс ***TAction*** и ограничивается только определением события ***OnExecute***, это соответствует конечному результату. В действительности разработчики VCL и CLX (поскольку все сказанное относится и к новому пакету межплатформенной разработки, появившемуся в 6-й версии C++Builder) создали более гибкий механизм, порождая не одно, а целую цепочку событий. Кроме самого ***TAction*** в ней могут принять участие список действий ***TActionList***, к которому принадлежит действие, приложение ***TApplication***, активная форма, главная форма приложения (если активная форма не является главной) и активный элемент управления. Рассмотрим, каким же образом и в каком порядке обеспечивается совместная работа всех задействованных объектов. Для начала ознакомимся с некоторыми особенностями реализации ***TAction***. Иерархия классов, реализующих действия, показана на рис. 2.13.

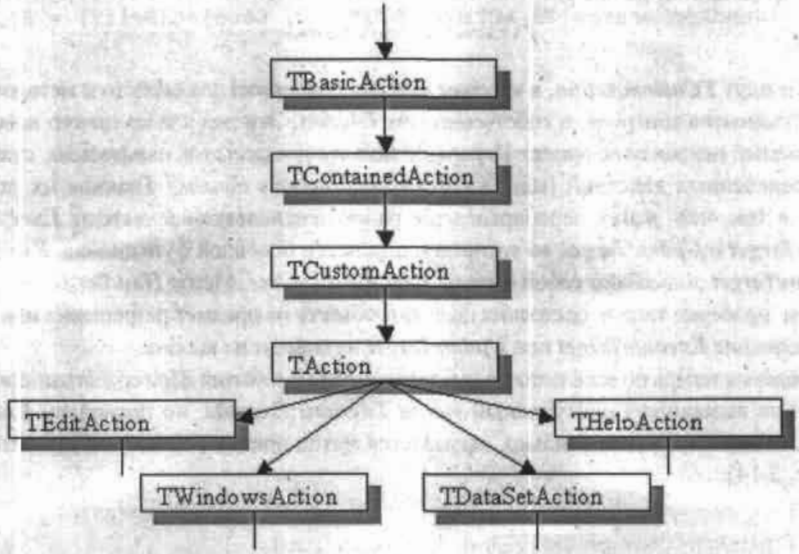


Рис. 2.13. *Иерархия классов, реализующих действия (Actions)*

В основе иерархии лежит ***TBasicAction***, который является базовым классом для всех объектов, реализующих действия. Более подробно с ним можно ознакомиться по справочной системе. Отметим следующее.

1. Довольно прозрачную реализацию виртуальных методов ***Execute()*** и ***Update()***, которые, как и ожидалось, просто вызывают соответствующие события ***OnExecute*** и ***OnUpdate***, если они определены.
2. Наличие дополнительных виртуальных методов ***ExecuteTarget(TObject* Target)***, ***HandlesTarget(TObject* Target)*** и ***UpdateTarget(TObject* Target)***, которым в качестве аргумента передается некий целевой (судя по названию) объект и которые пока ничего не делают. В качестве целевого объекта передается указатель на активный элемент контроля или на компонент. Что конкретно будет передаваться, определяется реализацией виртуального метода ***TComponent::ExecuteAction(TBasicAction* Action)***.

Следующим по иерархии является класс ***TContainedAction***, в котором добавлены свойства (в частности, свойство ***ActionList***) и методы, необходимые для работы в составе списка действий ***TActionList***, и, самое главное, переопределен метод ***Execute***. Теперь этот метод запускает цепочку вызовов, которая в нотации Object Pascal выглядит следующим образом.

```
function TContainedAction.Execute: Boolean;
begin
  Result := (ActionList <> nil) and ActionList.ExecuteAction(Self) or
    Application.ExecuteAction(Self) or inherited Execute or
    (SendAppMessage(CM_ACTIONEXECUTE, 0, Longint(Self)) = 1);
end;
```

Затем идут ***TCustomAction***, в котором введена поддержка для свойств и методов пунктов меню и элементов контроля, и собственно сам ***TAction***. Эти два класса ничего нового в рассматриваемый механизм не вносят. Иерархию замыкают классы так называемых стандартных предопределенных действий (standart pre-defined actions classes). Главное их отличие от ***TAction*** в том, что в них переопределены ранее неиспользуемые методы ***ExecuteTarget***, ***HandlesTarget*** и ***UpdateTarget***, на которые и перенесен основной функционал. ***ExecuteTarget*** и ***UpadateTarget*** подменяют собой методы ***Execute*** и ***Update***. Метод ***HandlesAction*** реализует механизм проверки типа и состояния целевого объекта на предмет разрешения или запрещения выполнения ***ExecuteTarget*** или ***UpdateTarget*** на момент из вызова.

Пройдемся теперь по всей цепочке вызовов методов и событий. При срабатывании элемента управления вызывается виртуальный метод ***TAction::Execute***, но поскольку ***Execute*** для ***TAction*** не переопределен, реально вызывается метод предка ***TContainedAction::Execute()*** (см. рис. 2.14).

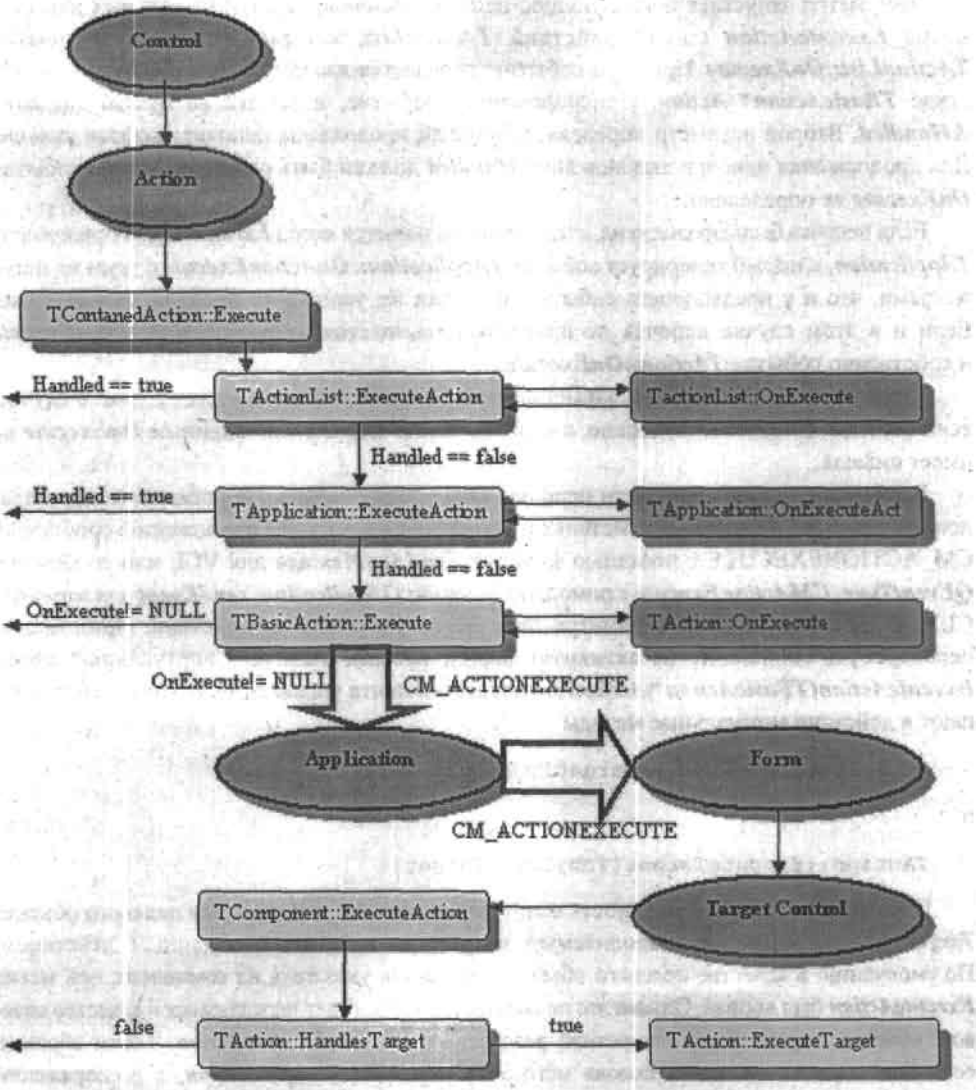


Рис. 2.14. *Порядок вызовов методов при выполнении действия (Action)*

Этот метод запускает последующую цепочку вызовов, первым из которых является метод ***ExecuteAction*** списка действий ***TActionList***, который генерирует событие ***TActionList::OnExecute***. При этом событию передается два параметра: указатель на действие ***TBasicAction* Action***, сгенерировавшее событие, и ссылка на булево значение ***&Handled***. Второй параметр определяет, будет ли продолжена цепочка вызовов дальше. Для продолжения цепочки вызовов либо ***Handled*** должен быть равен ***false***, либо событие ***OnExecute*** не определено.

Если цепочка была продолжена, следующим вызывается метод ***ExecuteAction*** приложения ***TApplication***, который генерирует событие ***TApplication::OnActionExecute*** с теми же параметрами, что и у предыдущего события, и с теми же условиями продолжения цепочки. Если и в этом случае цепочка не прервалась, вызовется метод ***TBasicAction::Execute*** и собственно событие ***TAction::OnExecute***.

Здесь цепочка, как правило, заканчивается. Продолжиться она может только в случае, если событие ***OnExecute*** не задано, а использование ***TAction*** с незаданным ***OnExecute*** не имеет смысла.

Вернемся к оставшейся части цепочки. Она предназначена для использования в предопределенных стандартных действиях и заключается в посылке приложению сообщения CM_ACTIONEXECUTE с помощью функции ***SendAppMessage*** для VCL или сообщения ***QEventType_CMActionExecute*** с помощью функции ***QApplication_sendEvent*** для варианта CLX. В качестве второго параметра передается указатель на действие. Приложение переадресует сообщение на активную форму, которая вызывает виртуальный метод ***ExecuteAction(TBasicAction* Action)*** активного элемента управления. В этом случае вступают в действие виртуальные методы

```
bool TAction::HandlesTarget(TObject* Target);
```

и

```
TAction::ExecuteTarget(TObject* Target);
```

Первый определяет возможность выполнения данного действия для целевого объекта ***Target***, второй содержит выполняемый код, ассоциированный с данным действием. По умолчанию в качестве целевого объекта передается указатель на компонент, чей метод ***ExecuteAction*** был вызван. Однако это не обязательно. Что будет передаваться в качестве целевого объекта, определяется конкретной реализацией метода ***ExecuteAction***. Таким образом, если использовать механизм вызова метода ***TComponent::ExecuteAction***, в распоряжении программиста окажется более гибкий и мощный метод, чем при использовании просто ***TAction***.

Как уже было указано выше, этот механизм применяется в предопределенных стандартных действиях, набор которых существенно расширен в шестой версии C++Builder. К ранее существовавшим уже группам Edit Actions, Window Actions, Help Actions и DataSet Actions были добавлены группы Format actions, File actions, Search actions, Tab (page control) actions, List actions, Dialog actions, Internet actions, Tools actions. Возможности, предоставляемые стандартными действиями, можно продемонстрировать на примере программы, реализующей относительно сложный алгоритм манипулирования данными двух списков (копирование, перемещение, удаление). При этом весь этот функционал будет создан только средствами дизайна. Создадим проект, на форме расположим два компонента ***TListBox***, пять кнопок ***TSpeedButton*** и список действий ***TActionList***. В результате у формы должен получиться примерно такой вид (см. рис. 2.15).

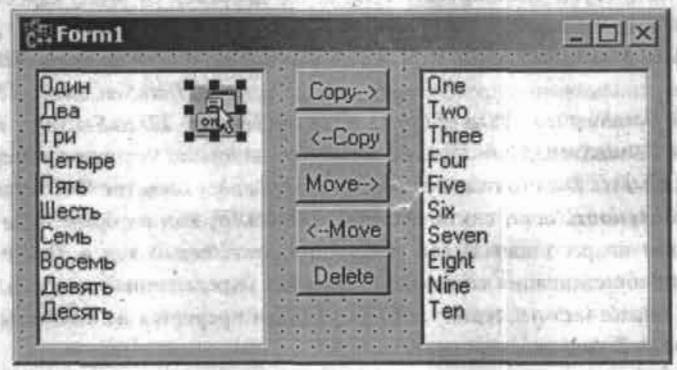


Рис. 2.15. *Форма с двумя списками, демонстрирующая возможности стандартных действий*

Откроем редактор списка действий (всплывающее меню->пункт ***"Action List Editor"***) и добавим в него (Ctrl-Insert) пять стандартных действий, выбрав их из окна ***"Standard Action Classes"***. В данном случае это будут стандартные действия из группы ***List***: ***TListContolCopySelection*** (2 экземпляра), ***TListContolMoveSelection*** (2 экземпляра) и ***TListContolDeleteSelection***. У ***TListContolCopySelection*** и ***TListContolMoveSelection*** имеются свойства ***ListControl*** и ***Destination***. Эти свойства имеют тип ***TCustomListControl**** и должны быть заполнены указателем на список-источник и на список-цель соответственно. Для одной пары ***TListContolCopySelection*** и ***TListContolMoveSelection*** свойство ***ListContol*** заполняем указателем на левый ***ListBox***, а свойство ***Destination*** – указателем на правый ***ListBox***. Аналогичную операцию повторяем и для второй пары, поменяв местами источник и цель. Для ***TListContolDeleteSelection*** свойство ***ListBox*** оставляем незаполненным. Заполняем свойства ***Caption*** в соответствии с предназначением каждого действия и сопос-

тавляем действия кнопкам посредством заполнения у них свойства ***Action***. Проект закончен. Готовая программа сможет перемещать, копировать выделенные данные из одного списка в другой или удалять их по нажатию соответствующей кнопки, а также следить за состоянием этих кнопок и управлять их доступностью, исходя из конкретной ситуации.

Удобство использования стандартных действий наводит на мысль о создании своих собственных стандартных действий. Попытаемся это проделать. В качестве базового класса воспользуемся ***TDataSetAction***, который является базовым для группы ***DataSet Actions***. В отличие от обычного ***TAction***, у них появилось свойство ***DataSource***. Назначение его понятно из названия. Отмечу лишь одну интересную особенность: если явно не указать ***DataSource***, то действие будет производиться над активным в данный момент ***TDataSet***. Кроме того, автоматически отслеживается доступность действия в зависимости от состояния ***TDataSet***. Попробуем унаследовать свой класс действий от этого класса. Для начала выясним, чем нас не устраивает стандартный ***TDataSetAction***.

К сожалению, в дизайн-режиме нельзя непосредственно использовать ***TDataSetAction***. Возможно лишь использование предопределенных действий ***TDataSetCancel***, ***TDataSetDelete***, ***TDataSetEdit***, ***TDataSetFirst***, ***TDataSetInsert***, ***TDataSetLast***, ***TDataSetNext***, ***TDataSetPost***, ***TDataSetPrior*** и ***TDataSetRefresh***. Это несколько ограничивает условия применения. Кроме того, если не указывать ***DataSource***, невозможно задать свое событие ***OnExecute***, поскольку отсутствует возможность определить активный ***TDataSet***, над которым надо производить действия. Нежелательно также и вставлять дополнительный код в ***OnUpdate***. В этом случае механизм отслеживания состояния ***TDataSet***, определенный для каждого события в ***UpdateTarget***, перестанет работать. Цепочка вызовов прервется на ***TBasicAction::Update***, и до вызова ***UpdateTarget*** дело просто не дойдет.

В связи с этими ограничениями хотелось бы иметь некий расширенный класс, назовем его ***TExtDataSetAction***, в котором в дизайн-режиме можно было бы задавать состояния ***TDataSet***, при которых это действие разрешено, а также прописывать требуемый код, используя события ***OnExecute*** и ***OnUpdate***. При этом, естественно, должны сохраниться все преимущества базового класса. Определим, какими дополнительными свойствами должен обладать ***TextDataSetAction***, чтобы удовлетворять предъявляемым требованиям.

Первое – набор состояний ***DataSetStates***, который определяет те состояния ***DataSet***, при которых действие разрешено, и который имеет тип ***Set<TDataSetState, dsInactive, dsOpening>***.

Второе – набор состояний курсора ***CursorStates***, имеющий тип ***TCursorStates***

```
typedef enum  { csBof, csEof, csEmpty } TCursorState;
typedef Set<TCursorState,csBof, csEmpty> TcursorStates;
```

и определяющий состояния курсора, при которых действие также разрешено. Здесь ***csBof*** – действие разрешено, если курсор указывает в начало таблицы, ***csEof*** – в конец и ***csEmpty*** – если таблица пуста.

Третье свойство – ***IsModify*** – признак того, что данное действие может изменять содержимое таблицы. Имеет тип ***bool***.

Четвертое дополнительное свойство – ***DataSetTarget*** типа ***TDataSet****, в котором хранится указатель на текущий активный ***TDataSet***. Будем использовать это свойство, если свойство ***TDataSetAction::DataSource*** не задано.

Для того чтобы дать возможность программисту запрещать действие вне зависимости от состояния ***TDataSet***, переопределим свойство ***Enabled***, а также переопределим события ***OnExecute*** и ***OnUpdate*** с целью переноса их из методов ***Execute*** и ***Update*** в методы ***ExecuteTarget*** и ***UpdateTarget*** соответственно. В событие ***OnUpdate*** добавим параметр ***bool & Allow***, с помощью которого можно определять дополнительные условия доступности действия. Для того чтобы правильно реализовать методы ***ExecuteTarget*** и ***UpdateTarget***, необходимо выяснить, что им передается в качестве целевого объекта. Анализ исходных кодов показал,что все Data Controls компоненты в своих методах ***ExecuteAction*** и ***ExecuteUpdate*** вызывают методы ***ExecuteAction*** и ***ExecuteUpdate*** объекта типа ***TDataLink***, который в качестве целевого объекта передает указатель на объект ***TDataSource***. Текст описания интерфейса и реализации методов приведен ниже.

ExtDataSetAction.h:

```
#ifndef ExtDataSetActionH
#define ExtDataSetActionH
//---------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
#include <ActnList.hpp>
#include <DBActns.hpp>
//---------------------------------------------------------------------
typedef Set<TDataSetState, dsInactive, dsOpening> TDataSetStates;
typedef enum {csBof, csEof, csEmpty}  TCursorState;
typedef Set<TCursorState, csBof, csEmpty> TCursorStates;
typedef void __fastcall (__closure *TUpdateActionEvent)(System::TObject*
Sender, bool& Allow);
class PACKAGE TExtDataSetAction : public TDataSetAction
{
private:
        TDataSetStates FDataSetStates;
        TCursorStates FCursorStates;
        bool FEnabled;
        TNotifyEvent FOnExecute;
```

```
        TUpdateActionEvent FOnUpdate;
        TDataSet* FTarget;
        bool FIsModify;
        void __fastcall SetDataSetStates(TDataSetStates value);
        void __fastcall SetCursorStates(TCursorStates value);
        void __fastcall SetEnabled(bool value);
        void __fastcall SetTarget(TDataSet* value);
        void __fastcall SetIsModified(bool value);
        void __fastcall SetIsModify(bool value);
protected:
        virtual void __fastcall ExecuteTarget(TObject* Target);
        void __fastcall UpdateTarget(TObject* Target);
public:
        __fastcall TExtDataSetAction(TComponent* Owner);
__published:
        __property TDataSetStates DataSetStates =
{ read=FDataSetStates, write=SetDataSetStates, default=0x0E};
        __property TCursorStates CursorStates =
{ read=FCursorStates, write=SetCursorStates, default=0 };
        __property bool IsModify =
{ read=FIsModify, write=SetIsModify, default=false };
        __property bool Enabled =
{ read=FEnabled, write=SetEnabled, default=true };
        __property TNotifyEvent OnExecute  =
{ read=FOnExecute, write=FOnExecute };
        __property TUpdateActionEvent OnUpdate  =
{ read=FOnUpdate, write=FOnUpdate };
        __property TDataSet* TargetDataSet  =
{ read=FTarget, write=SetTarget };
};
//---------------------------------------------------------------------
#endif
```

ExtDataSetAction.cpp:

```
#include <vcl.h>
#pragma hdrstop
#include "ExtDataSetAction.h"
#pragma package(smart_init)
//---------------------------------------------------------------------
__fastcall TExtDataSetAction::TExtDataSetAction(TComponent* Owner)
        : TDataSetAction(Owner),
```

```
        FEnabled(true),
        FOnExecute(NULL),
        FOnUpdate(NULL),
        FTarget(NULL),
        FIsModify(false)
{
  FDataSetStates << dsBrowse << dsEdit << dsInsert;
}
//---------------------------------------------------------------------
void __fastcall TExtDataSetAction::SetDataSetStates(TDataSetStates value)
{
        if(FDataSetStates != value) {
                FDataSetStates = value;
        }
}
//---------------------------------------------------------------------
void __fastcall TExtDataSetAction::SetCursorStates(TCursorStates
value)
{
        if(FCursorStates != value) {
                FCursorStates = value;
        }
}
//---------------------------------------------------------------------
void __fastcall TExtDataSetAction::SetIsModify(bool value)
{
        if(FIsModify != value) {
                FIsModify = value;
        }
}
//---------------------------------------------------------------------
void __fastcall TExtDataSetAction::SetEnabled(bool value)
{
        if(FEnabled != value) {
                FEnabled = value;
                if (!FEnabled)
                  TDataSetAction::Enabled = false;
        }
}
//---------------------------------------------------------------------
void __fastcall TExtDataSetAction::SetTarget(TDataSet* value)
```

```
{
        if(FTarget != value) {
                FTarget = value;
        }
}
//---------------------------------------------------------------------
void __fastcall TExtDataSetAction::ExecuteTarget(TObject* Target)
{
   FTarget = GetDataSet(Target);
   if (FOnExecute) FOnExecute(this);
}
//---------------------------------------------------------------------
void __fastcall TExtDataSetAction::UpdateTarget(TObject* Target)
{
   if (!FEnabled) return;
   FTarget =  GetDataSet(Target);
   bool bAllow = false;
   if(FTarget && FOnExecute)
   {
    bAllow = FDataSetStates.Contains(FTarget->State);
    if(bAllow && FTarget->Active)
        bAllow = (FIsModify?FTarget->CanModify:true) &&
                 (FCursorStates.Contains(csBof)?true:!FTarget->Bof) &&
                 (FCursorStates.Contains(csEof)?true:!FTarget->Eof) &&
                 (FCursorStates.Contains(csEmpty)?true:!FTarget->IsEmpty());
    if (FOnUpdate)
         FOnUpdate(this,bAllow);
   }
   TDataSetAction::Enabled = bAllow;
}
```

Кроме того, о существовании этого класса надо каким-то образом известить среду разработки. Для этого разработчики C++ Builder предоставили функцию регистрации действий.

```
extern PACKAGE void __fastcall RegisterActions(constAnsiString
CategoryName, TMetaClass* * AClasses, const int AClasses_Size,
TMetaClass* Resource);
```

В качестве параметров ей передаются имя категории действия, массив указателей на информацию о классах регистрируемых действий, индекс последнего элемента массива. Последний параметр – это ресурсный параметр, в котором хранятся значения по

умолчанию для каждого действия. В нашем случае он не нужен, и ему можно присвоить значение NULL. Добавляем вызов этой функции для регистрации нашего класса в файл реализации класса.

```
namespace Extdatasetaction
{
  void __fastcall PACKAGE Register()
  {
    RegisterActions("ExtDataSet", &__classid(TExtDataSetAction), 0, NULL);
  }
}
```

Результат нашей работы вставляем в проект нового пакета (package) или добавляем в уже существующий. После компиляции и линковки инсталлируем созданный пакет. Внешне в палитре компонентов после инсталляции ничего не изменится, но при работе с *TActionList* при выборе пункта *New Standard Action* во всплывающем меню откроется диалоговое окно со списком стандартных действий, среди которых мы можем найти и *TExtDataSetAction* (см. рис. 2.16).

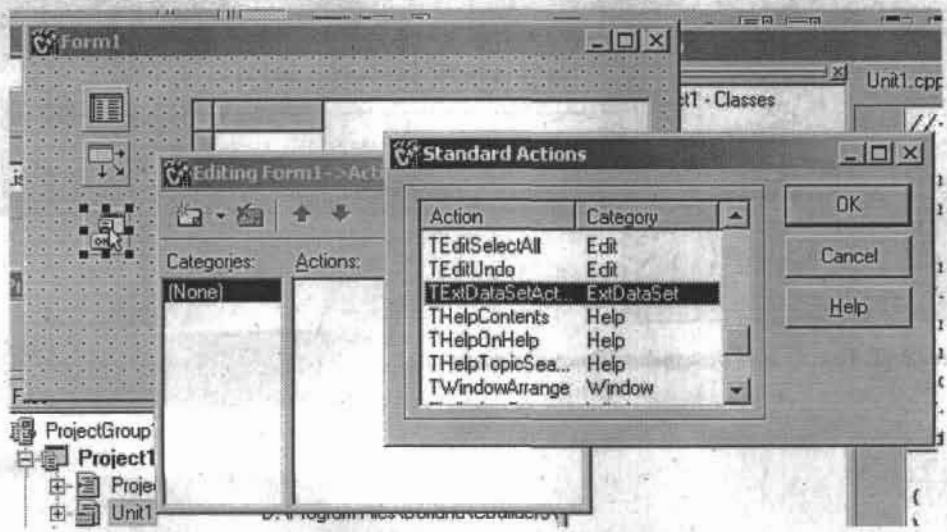


Рис. 2.16. *Список стандартных действий*

Теперь о том, как этим пользоваться. Попробуем определить действие "Переход в начало таблицы", которое не может быть выполнено, если таблица находится в режиме редактирования/вставки. В инспекторе для ***CursorStates*** устанавливаем ***csEof*** в ***true***, а ***csBof*** и ***csEmpty*** в ***false***, что будет означать, что действие запрещено, если таблица пуста и курсор позиционирован на первую запись. Также сбрасываем все флаги ***DataSetStates*** в ***false***, за исключением ***dsBrowse*** – действие разрешено, когда таблица находится в состоянии просмотра.

Определим событие ***OnExecute***.

```
void __fastcall TForm1::ExtDataSetFirstExecute(TObject *Sender)
{
  ((TExtDataSetAction*)Sender)->TargetDataSet->First();
}
```

Вот и все. Событие будет отслеживать заданные условия автоматически и либо разрешать, либо запрещать действие в соответствии с заданными ограничениями (см. рис. 2.17). На рис. 2.18 показана реализация действия Post с наложением дополнительного условия.

Рис. 2.17. *Реализация действия "В начало таблицы"*

Рис. 2.18. *Реализация действия "Сохранить" (Post)*

Работа с датой и временем в VCL: TDateTime

Для работы с датами и временем C++Builder предоставляет нам, программистам, ряд функций, пожалуй, из самого богатого по функциональности модуля – Sysutils.hpp, а также класс ***TDateTime***, объявленный в модуле systdate.h. Вот о нем сейчас и пойдет речь.

Первое, что надо отметить при начале работы с данным классом, – это то, что он не унаследован от ***TObject***, поэтому совершенно необязательно создавать экземпляр класса через оператор new.

Второе: у этого класса нет свойств и событий, а есть только методы.

И третье: у ***TDateTime*** существует большое количество перегруженных арифметических операторов.

Теперь, после этого беглого знакомства, сразу перейдем к тестовому проекту, ибо использование ***TDateTime*** совершенно несложно и что как работает – мы разберемся по ходу дела.

Внешний вид тестового проекта, загруженного в C++Builder, представлен на рис. 2.19.

Как видно из рисунка, наш тестовый проект охватывает почти все аспекты работы с ***TDateTime***. Запустите проект на выполнение и убедитесь в этом сами (рис. 2.20).

Теперь давайте разбираться, что, где и как. Ниже полностью приведен код тестового проекта.

Листинг 1. Код проекта

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "DateTimeUnit.h"
```

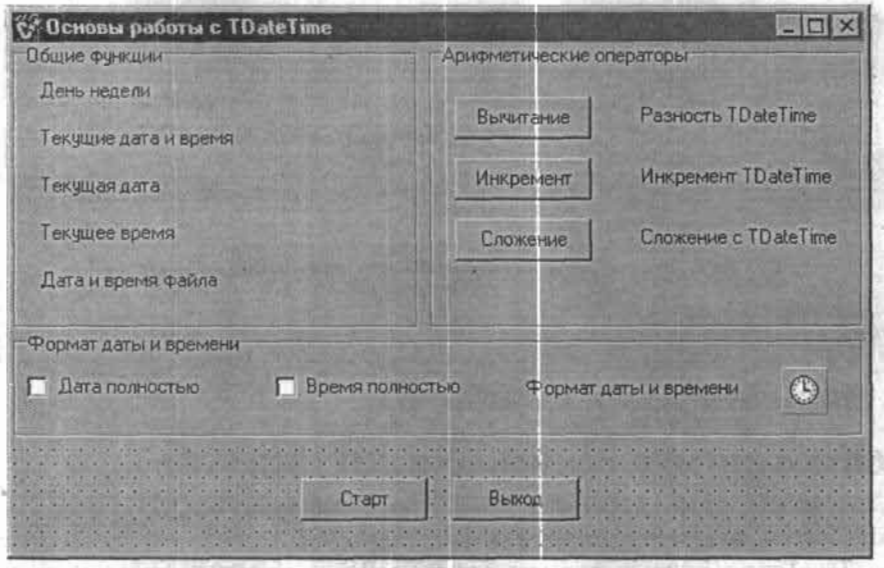


Рис. 2.19. *Форма тестового проекта в среде разработки*

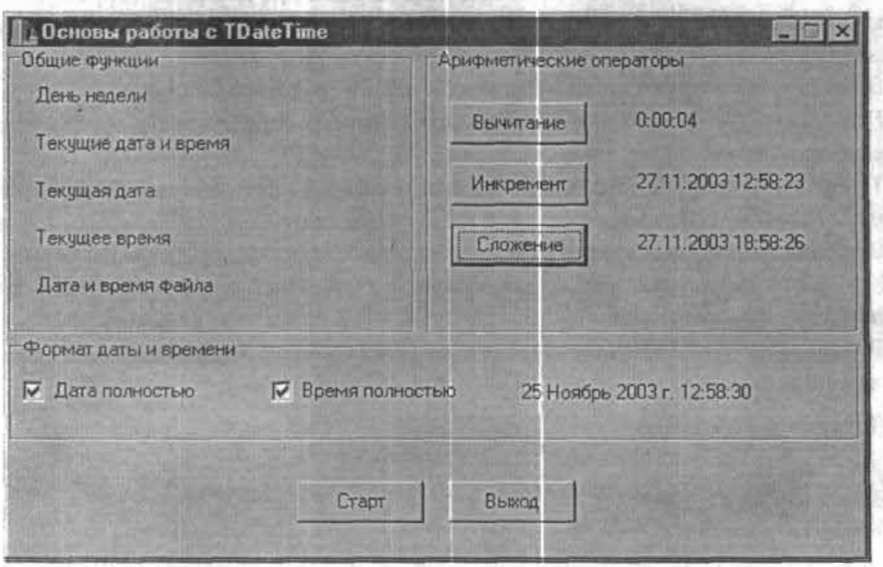


Рис. 2.20. *Запущенный проект*

```
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
   : TForm(Owner)
   {
      DateFormat = "dd.mm.yy ";
      TimeFormat = "hh:nn";
   }
//---------------------------------------------------------------------
void __fastcall TMainForm::OKButtonClick(TObject *Sender)
   {
      TDateTime DateTime = TDateTime::CurrentDateTime();

      CurrenDatetTimeLabel->Caption = DateTime.DateTimeString();
      CurrentDateLabel->Caption = DateTime.DateString();
      CurrentTimeLabel->Caption = DateTime.TimeString();

      int DayOfWeek = DateTime.DayOfWeek();
      switch(DayOfWeek)
         {
            case 1:
               DayOfWeekLabel->Caption = "сейчас воскресенье, " +
               AnsiString(DateTime);
            break;
            case 2:
               DayOfWeekLabel->Caption = "сейчас понедельник, " +
               AnsiString(DateTime);
            break;
            case 3:
               DayOfWeekLabel->Caption = "сейчас вторник, " +
               AnsiString(DateTime);
            break;
            case 4:
               DayOfWeekLabel->Caption = "сейчас среда, " +
               AnsiString(DateTime);
            break;
            case 5:
               DayOfWeekLabel->Caption = "сейчас четверг, " +
               AnsiString(DateTime);
```

```
            break;
            case 6:
               DayOfWeekLabel->Caption = "сейчас пятница, " +
               AnsiString(DateTime);
            break;
            case 7:
               DayOfWeekLabel->Caption = "сейчас суббота, " +
               AnsiString(DateTime);
            break;
         }
      DateTime = TDateTime::FileDateToDateTime(FileAge(Application->ExeName));
      FileDatetTimeLabel->Caption = "дата и время файла: " +
      DateTime.DateTimeString();
   }
//---------------------------------------------------------------------
void __fastcall TMainForm::ExitButtonClick(TObject *Sender)
   {
      Application->Terminate();
   }
//---------------------------------------------------------------------
void __fastcall TMainForm::DateDifferenceButtonClick(TObject *Sender)
   {
      TCursor OldCursor = Screen->Cursor;
      Screen->Cursor = crHourGlass;

      TDateTime StartDateTime = TDateTime::CurrentDateTime();

      for(int i = 0; i < 500000; i++)
         {
            TDateTime(TDateTime::CurrentDateTime().DateTimeString());
            Application->ProcessMessages();
         }

      TDateTime EndDateTime = TDateTime::CurrentDateTime();
      Screen->Cursor = OldCursor;

      DateDifferenceLabel->Caption = (EndDateTime - StartDateTime).TimeString();
   }
//---------------------------------------------------------------------
void __fastcall TMainForm::IncrementDateButtonClick(TObject *Sender)
   {
      static TDateTime DateTime = TDateTime::CurrentDateTime();
```

```
        IncrementDateLabel->Caption = (DateTime++).DateTimeString();
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::DateAdditionButtonClick(TObject *Sender)
    {
        static TDateTime DateTime = TDateTime::CurrentDateTime();
        DateAdditionLabel->Caption=(DateTime+=1.125).DateTimeString();
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::TimerTimer(TObject *Sender)
    {
        FormatDateTimeLabel->Caption=TDateTime::CurrentDateTime().
        FormatString(DateFormat
        + TimeFormat);
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::FullDateCheckBoxClick(TObject *Sender)
    {
        if(FullDateCheckBox->Checked)
            DateFormat = "dddddd ";
        else
            DateFormat = "ddddd ";

        FormatDateTimeLabel->Caption = TDateTime::CurrentDateTime().
        FormatString(DateFormat
        + TimeFormat);
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::FullTimeCheckBoxClick(TObject *Sender)
    {
        if(FullTimeCheckBox->Checked)
            TimeFormat = "hh:nn:ss";
        else
            TimeFormat = "hh:nn";

        FormatDateTimeLabel->Caption = TDateTime::CurrentDateTime().
        FormatString(DateFormat
        + TimeFormat);
    }
//---------------------------------------------------------------------
```

Начнем мы с обработчика события ***OnClick*** кнопки ***OKButton***. Первой строкой в этом обработчике мы получаем текущую дату и время с помощью метода ***CurrentDateTime()*** класса ***TDateTime***. Метод ***CurrentDateTime()*** объявлен статическим методом.

```
static TDateTime __fastcall CurrentDateTime();
```

поэтому вызываем его как ***TDateTime::CurrentDateTime()***. Возвращаемое им значение присваиваем объекту ***DateTime***.

Далее, мы используем группу схожих методов для получения строкового представления даты и времени, отдельно даты и отдельно времени.

```
AnsiString __fastcall DateTimeString() const;
AnsiString __fastcall DateString() const;
AnsiString __fastcall DateString() const;
```

Строкой

```
int DayOfWeek = DateTime.DayOfWeek();
```

мы получаем текущий день недели. Метод ***DayOfWeek()*** объявлен следующим образом.

```
int __fastcall DayOfWeek() const;
```

Метод возвращает значение типа ***int***, которое и определяет день недели. Только надо помнить, что неделя начинается не с понедельника, а с воскресенья, поэтому возвращенной методом ***DayOfWeek()*** единице будет соответствовать воскресенье, двойке – понедельник, и т. д., вплоть до субботы, которой будет соответствовать семерка.

Оператор ***switch(DayOfWeek)***, "преобразующий", так сказать, возвращенное ***DayOfWeek()*** числовое значение в нормальный строковый заголовок метки ***DayOfWeekLabel***, не был бы интересен, если бы не одно "но" – строка

```
DayOfWeekLabel->Caption = "сейчас воскресенье, " +AnsiString(DateTime);
```

и ей подобные. В данных строках для получения строкового представления даты и времени используется не уже знакомый нам метод ***DateTimeString()***, а оператор ***AnsiString()*** класса ***TDateTime***.

```
__fastcall operator AnsiString() const;
```

Данный оператор преобразования возвращает строковое представление объекта ***TDateTime***. Преобразование использует формат даты и времени, определяемый значением глобальной переменной ***LongTimeFormat*** (об этом ниже). У ***TDateTime*** есть также еще операторы преобразования его в числовое представление.

```
__fastcall operator int() const;
__fastcall operator double() const;
```

Оператор ***int()*** возвращает целочисленное представление даты объекта ***TDateTime*** – число дней, прошедших с 30.12.1899.

Оператор ***double()*** возвращает числовое представление объекта ***TDateTime*** (всего объекта целиком, а не только даты, как в случае с оператором ***int()***). Целая часть возвращенного ***double()*** значения представляет собой число дней, прошедших с 30.12.1899, а дробная часть – время суток, представляющее собой часть от 24 часов. Но мы отвлеклись...

В строке

```
DateTime = TDateTime::FileDateToDateTime(FileAge(Application->ExeName));
```

мы получаем время и дату исполняемого файла нашего тестового проекта с помощью функции ***FileAge***, а затем преобразуем полученное значение в объект ***TDateTime***, используя метод ***FileDateToDateTime***. Обратите внимание, что данный метод является статическим методом класса.

```
static TDateTime __fastcall FileDateToDateTime(int fileDate);
```

Обратное преобразование объекта ***TDateTime*** в дату и время файла выполняет другой метод класса ***TDateTime***, ***FileDate***.

```
int __fastcall FileDate() const;
```

Метод ***FileDate*** статическим, в отличие от метода ***FileDateToDateTime***, не является.

В обработчике события ***OnClick*** кнопки ***DateDifferenceButton*** демонстрируется работа с оператором вычитания, перегруженным, как и ряд других арифметических операторов, в классе ***TDateTime***. Ключевой код в данном обработчике – это строка

```
DateDifferenceLabel->Caption = (EndDateTime - StartDateTime).TimeString();
```

в которой из текущего времени после окончания длительного цикла вычитается время, которое было текущим непосредственно перед началом цикла. Таким образом, мы сможем определить длительность цикла. Сразу хочу предупредить, что данный метод годится только для определения достаточно больших интервалов времени – от нескольких секунд и выше. На меньших интервалах будет очень низкая точность, а совсем короткие временные интервалы данным методом вообще не удастся измерить.

Обработчик события ***OnClick*** кнопки ***IncrementDateButton*** демонстрирует нам возможность оператора инкремента в ***TDateTime***.

```
static TDateTime DateTime = TDateTime::CurrentDateTime();
IncrementDateLabel->Caption = (DateTime++).DateTimeString();
```

Операция инкремента увеличивает дату на единицу. Противоположная по смыслу операция, декремент, уменьшает дату на единицу.

Далее, рассмотрим оператор сложения. Пример его использования мы можем наблюдать в обработчике события ***OnClick*** кнопки ***DateAdditionButton***.

```
static TDateTime DateTime = TDateTime::CurrentDateTime();
DateAdditionLabel->Caption =(DateTime += 1.125).DateTimeString();
```

При сложении числа с объектом ***TDateTime*** целая часть числа прибавляется к дате, а дробная часть – ко времени экземпляра ***TDateTime***. Таким образом, код

```
DateTime += 1.125
```

увеличит значение даты объекта ***DateTime*** на единицу, а значение времени – на три часа (0.125, одна восьмая часть суток).

В завершение разбора кода тестового проекта поговорим о методе ***FormatString*** класса ***TDateTime***. Метод объявлен следующим образом.

```
AnsiString __fastcall FormatString(const AnsiString& format);
```

В секции ***private*** класса формы объявлены две переменные: ***DateFormat*** для формата даты; ***TimeFormat*** для формата времени. Код:

```
private:
   AnsiString DateFormat;
   AnsiString TimeFormat;
```

В конструкторе формы задаются начальные значения этих переменных.

```
DateFormat = "dd.mm.yy ";
TimeFormat = "hh:nn";
```

Таким образом, мы задаем следующий формат вывода даты для метода ***FormatString***:

дд.мм.гг

и формат времени:

чч.мм.

При запуске приложения в обработчике события ***OnTimer*** таймера сформированные по заданному формату текущие дата и время задаются в качестве заголовка метки ***FormatDateTimeLabel***.

```
FormatDateTimeLabel->Caption = TDateTime::CurrentDateTime().FormatString
(DateFormat
+ TimeFormat);
```

В обработчиках события ***OnClick*** флажков (check-box) мы просто меняем значения переменных ***DateFormat*** и ***TimeFormat*** для изменения формата даты и времени, возвращаемых методом ***FormatString***.

Список значений форматирования приведен в табл. 2.2.

Табл. 2.2. *Список значений форматирования значений даты и времени*

Значение	Пояснение
c	Отображает дату, используя формат, определяемый значением глобальной переменной ***ShortDateFormat***, и время, следующее за значением даты, используя формат, определяемый значением глобальной переменной ***LongTimeFormat***. Время не отображается, если дробная часть значения ***DateTime*** равна нулю
d	Отображает день как число без лидирующего нуля (1-31)
dd	Отображает день как число с лидирующим нулем (01-31)
ddd	Отображает день как сокращение (Вск-Сб), используя строки, определяемые значением глобальной переменной ***ShortDayNames***
dddd	Отображает день как полное имя (Воскресенье-Суббота), используя строки, определяемые значением глобальной переменной ***LongDayNames***
ddddd	Отображает дату, используя формат, определяемый значением глобальной переменной ***ShortDateFormat***
dddddd	Отображает дату, используя формат, определяемый значением глобальной переменной ***LongDateFormat***
m	Отображает месяц как число без лидирующего нуля (1-12). Если сразу за спецификатором "m" следует спецификатор "h" или "hh", то вместо месяца отображаются минуты
mm	Отображает месяц как число с лидирующим нулем (01-12). Если сразу за спецификатором "mm" следует спецификатор "h" или "hh", то вместо месяца отображаются минуты
mmm	Отображает месяц как сокращение (Янв-Дек), используя строки, определяемые значением глобальной переменной ***ShortMonthNames***
mmmm	Отображает месяц как полное имя (Январь-Декабрь), используя строки, определяемые значением глобальной переменной ***LongMonthNames***
yy	Отображает год как двузначное число (00-99)
yyyy	Отображает год как четырехзначное число (0000-9999)
h	Отображает час как число без лидирующего нуля (0-23)
hh	Отображает час как число с лидирующим нулем (00-23)
n	Отображает минуту как число без лидирующего нуля (0-59)

Табл. 2.2. Список значений форматирования значений даты и времени (Продолжение)

Значение	Пояснение
nn	Отображает минуту как число с лидирующим нулем (00-59)
s	Отображает секунду как число без лидирующего нуля (0-59)
ss	Отображает секунду как число с лидирующим нулем (00-59)
t	Отображает время, используя формат, определяемый значением глобальной переменной ***ShortTimeFormat***
tt	Отображает время, используя формат, определяемый значением глобальной переменной ***LongTimeFormat***
am/pm	Для предшествующего спецификатора "h" или "hh" использует 12-часовой формат отображения времени, добавляя "am" к любому часу до полудня и "pm" к любому часу после полудня. Спецификатор "am/pm" может использовать нижний, верхний или смешанный регистр, результат отображается соответственно
a/p	Для предшествующего спецификатора "h" или "hh" использует 12-часовой формат отображения времени, добавляя "a" к любому часу до полудня и "p" к любому часу после полудня. Спецификатор "a/p" может использовать нижний, верхний или смешанный регистр, результат отображается соответственно
/	Отображает разделитель даты, определяемый значением глобальной переменной ***DateSeparator***
:	Отображает разделитель времени, определяемый значением глобальной переменной ***TimeSeparator***
'xx'/"xx"	Символы, заключенные в одинарные или двойные кавычки, отображаются "как есть", форматирование на них не действует

Если строка, определяемая значением параметра ***format*** метода ***FormatString***, пуста, то выполняется форматирование, как если бы в ***FormatString*** была передана строка "c".

Следует отметить, что в метод ***FormatString*** можно передавать не только строки, указанные в таблице, но и строки вида:

"Сегодня dd.mm.yyyy. Сейчас hh:nn:ss".

При передаче такой строки, например, в нижеприведенный вызов

```
Application->MessageBox(TDateTime::CurrentDateTime().FormatString
("Сегодня dd.mm.yyyy.
 Сейчас hh:nn:ss").c_str(), "Текущие дата и время", MB_OK |
MB_ICONINFORMATION);
```

на экране мы увидим следующее изображение (рис. 2.21).

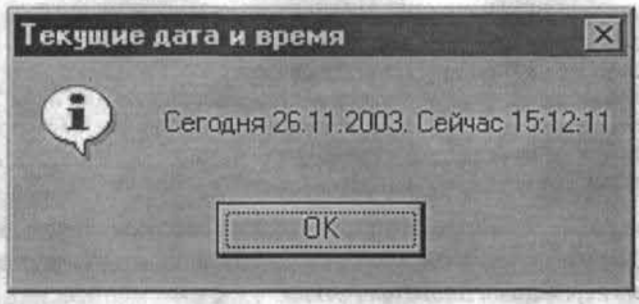


Рис. 2.21. *Пример форматирования значений даты и времени с помощью метода FormatString*

Почти все. Код проекта мы разобрали. Мы, правда, не упомянули о перегруженных операторах сравнения (<, <=, >, >= и т. д.), но смею вас уверить, что работают они так, как от них и ожидается, в чем вы можете убедиться сами. Также остались нерассмотренными два полезных метода класса ***TDateTime***: ***DecodeDate*** и ***DecodeTime***. Данные методы объявлены следующим образом:

```
void __fastcall DecodeDate(unsigned short* year, unsigned short* month,
unsigned short* day) const;
void __fastcall DecodeTime(unsigned short* hour, unsigned short* min,
unsigned short* sec,
unsigned short* msec) const;
```

и разбивают соответственно дату и время на составные части. Следующий код:

```
unsigned short Year;
unsigned short Month;
unsigned short Day;
TDateTime::CurrentDateTime().DecodeDate(&Year, &Month, &Day);
```

занесет текущее значение года в переменную ***Year***, текущее значение месяца – в переменную ***Month***, текущее значение числа месяца – в переменную ***Day***. Для метода ***DecodeTime*** все аналогично.

И наконец, поговорим о глобальных переменных, упомянутых ранее. Данные переменные объявлены в Sysutils.hpp следующим образом.

```
extern PACKAGE char DateSeparator;
extern PACKAGE AnsiString ShortDateFormat;
extern PACKAGE AnsiString LongDateFormat;
extern PACKAGE char TimeSeparator;
extern PACKAGE AnsiString TimeAMString;
```

```
extern PACKAGE AnsiString TimePMString;
extern PACKAGE AnsiString ShortTimeFormat;
extern PACKAGE AnsiString LongTimeFormat;
extern PACKAGE AnsiString ShortMonthNames[12];
extern PACKAGE AnsiString LongMonthNames[12];
extern PACKAGE AnsiString ShortDayNames[7];
extern PACKAGE AnsiString LongDayNames[7];
```

Начальные значения этих переменных берутся из настроек операционной системы. Чтобы изменить, например, настройки отображения даты в коротком формате, мы должны изменить значение переменной ***ShortDateFormat***.

```
ShortDateFormat = "yyyy-mm-dd";
```

Если добавить данную строчку, например, в конструктор формы нашего тестового проекта, то все даты, на отображение которых влияет эта переменная, будут выводиться в формате "гггг-мм-дд" (рис. 2.22).

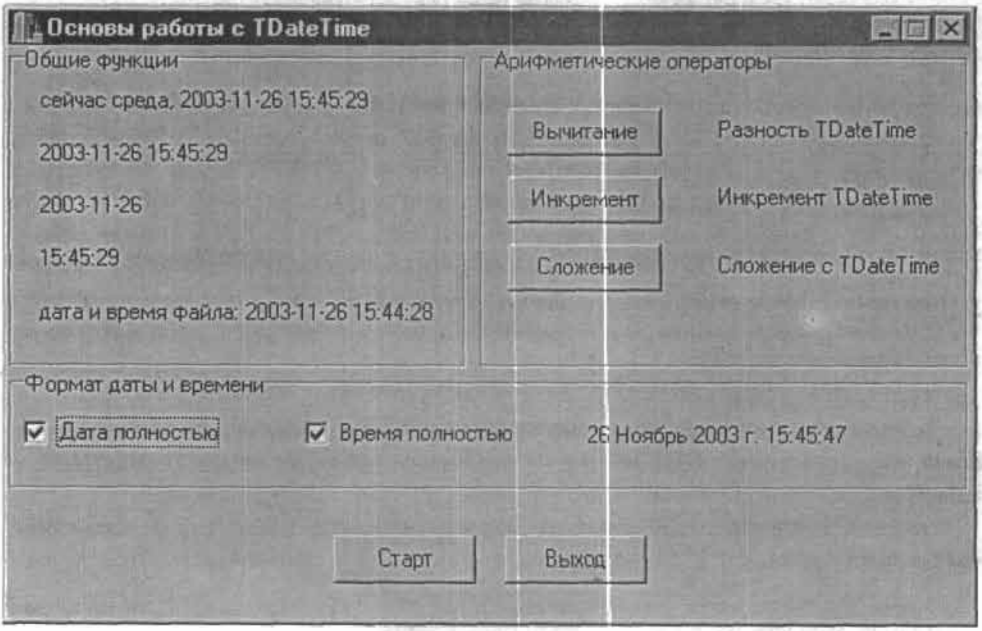


Рис. 2.22. *Изменение формата отображения даты через переменную ShortDateFormat*

С остальными переменными аналогично.

Это, наверное, все, что можно было рассмотреть в классе ***TDateTime*** и его "окрестностях". Однако это не единственный инструмент для работы с датами и временем в C++Builder. Соответствующий инструментарий есть и в модуле Sysutils.hpp. Но об этом в другой раз.

TMouse

Список полезных модулей в C++Builder не ограничивается, конечно же, одним лишь Sysutils.hpp. Все модули по-своему полезны, но некоторые, к сожалению, малоизвестны многим программистам на C++Builder. Одним из таких модулей является Controls.hpp, который содержит востребованный в повседневной работе класс ***TMouse***.

Казалось бы, зачем нужен ***TMouse***, если у потомков ***TControl*** имеются свои события, связанные с мышью, и в них предоставляется достаточно информации для пользователя? Но дело в том, что данная информация предоставляется в контексте конкретного элемента управления. Например, событие ***OnMouseMove TControl***, объявленное следующим образом.

```
typedef void __fastcall (__closure *TMouseMoveEvent)(System::TObject*
Sender, Classes:: TShiftState Shift, int X, int Y);
```

позволяет получить текущие координаты мыши в пределах элемента управления, в котором это событие произошло, т. наз. клиентские координаты, ничего не сообщая о том, где в данный момент мышь находится на экране.

Таким образом, можно считать (да так оно и есть на самом деле), что ***TMouse*** предназначен для получения глобальной информации о мыши. Для работы с ***TMouse*** нам нет необходимости создавать экземпляр класса. В Borland об этом уже позаботились и предоставили нам глобальную переменную ***Mouse***, имеющую тип "указатель на объект класса TMouse" – ***TMouse*** *. Переменная объявлена в модуле Controls.hpp следующим образом:

```
extern PACKAGE TMouse* Mouse;
```

Теперь приступим к рассмотрению свойств и методов класса. Класс ***TMouse*** унаследован непосредственно от ***TObject*** и имеет единственный метод, не считая конструктора и деструктора, ***SettingChanged***.

```
void __fastcall SettingChanged(int Setting);
```

Вы никогда не должны вызывать этот метод – он автоматически вызывается в ответ на изменения в настройках мыши, сделанные через Панель управления Windows, или каким-либо иным образом. Данный метод обновляет переменную ***Mouse*** в ответ на произведенные изменения.

Переходим к свойствам. В классе ***TMouse*** объявлены следующие свойства:

- Capture
- CursorPos
- DragImmediate
- DragThreshold
- MousePresent
- RegWheelMessage
- WheelPresent
- WheelScrollLines

Свойство ***Capture***

```
__property HWND Capture = {read=GetCapture, write=SetCapture, nodefault};
```

представляет собой дескриптор окна, получающего ввод мыши, или, как говорят, имеющего захват мыши (mouse capture). Можно прочитать значение этого свойства, чтобы узнать, какое окно в настоящий момент захватило мышь, и установить значение этого свойства, чтобы передать захват мыши другому окну. Обычно этого не требуется, но мало ли...

Свойство ***CursorPos***:

```
__property Windows::TPoint CursorPos = {read=GetCursorPos,
write=SetCursorPos};
```

Свойство представляет собой экранные, не зависящие от элемента управления координаты мыши. В принципе их можно получить, используя функции API ***GetCursorPos*** и ***SetCursorPos***, но через свойство, на мой взгляд, удобнее. Данное свойство может быть полезно при размещении элементов пользовательского интерфейса относительно курсора мыши, без привязки к каким-либо событиям типа ***OnMouseMove*** и прочим. Данная возможность часто используется в позиционировании всплывающих подсказок или, например, размещении всплывающего меню.

Свойства ***MousePresent*** и ***WheelPresent***

```
__property bool MousePresent = {read=FMousePresent, nodefault};
__property bool WheelPresent = {read=FWheelPresent, nodefault};
```

позволяют определить наличие мыши в системе и колесика у мыши соответственно.

Для чего предназначено свойство ***RegWheelMessage***

```
__property unsigned RegWheelMessage = {read=FWheelMessage, nodefault};
```

мы даже рассматривать не будем. Оно применимо только для Windows 95, а много ли вы написали в последнее время программ, рассчитанных только на Windows 95?

Свойство ***WheelScrollLines***

```
__property int WheelScrollLines = {read=FScrollLines, nodefault};
```

позволяет узнать, сколько строк прокручивается при вращении колесика мыши на один шаг. Свойство отображает настройку операционной системы, которую можно изменить в апплете "Мышь" Панели управления Windows.

И у нас для рассмотрения остались свойства ***DragImmediate*** и ***DragThreshold***.

```
__property bool DragImmediate = {read=FDragImmediate, write=FDragImmediate,
default=1};
__property int DragThreshold = {read=FDragThreshold, write=FDragThreshold,
default=5};
```

Данные свойства полезны для работы с Drag'n'Drop. Свойство ***DragImmediate*** определяет, начинает ли щелчок левой кнопкой мыши операцию Drag'n'Drop немедленно. Если значение этого свойства установлено в ***false***, то операция Drag'n'Drop начнется только после того, как курсор мыши переместится на количество пикселов, определяемых значением свойства ***DragThreshold***.

Теперь можно перейти к рассмотрению тестового проекта. Код тестового проекта приведен ниже.

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "TMouseUnit.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
    : TForm(Owner)
    {
    }
//---------------------------------------------------------------------
```

```
void __fastcall TMainForm::FormShow(TObject *Sender)
   {
      ScreenCursorPosLabel->Caption = "x-координата: " +
      IntToStr(Mouse->CursorPos.x) + ", y-координата: " +
      IntToStr(Mouse->CursorPos.x);

      FormCursorPosLabel->Caption = "ждем события OnMouseMove";

      if(Mouse->DragImmediate)
         DragImmediateLabel->Caption = "да";
      else
         DragImmediateLabel->Caption = "нет";

      DragThresholdLabel->Caption = IntToStr(Mouse->DragThreshold);

      if(Mouse->MousePresent)
         MousePresentLabel->Caption = "да";
      else
         MousePresentLabel->Caption = "нет";

      if(Mouse->WheelPresent)
         WheelPresentLabel->Caption = "да";
      else
         WheelPresentLabel->Caption = "нет";

      WheelScrollLinesLabel->Caption = IntToStr(Mouse->WheelScrollLines);
   }
//---------------------------------------------------------------------
void __fastcall TMainForm::FormMouseMove(TObject *Sender, TShiftState
Shift, int X, int Y)
   {
      ScreenCursorPosLabel->Caption = "x-координата: " +
      IntToStr(Mouse->CursorPos.x) + ", y-координата: " +
      IntToStr(Mouse->CursorPos.x);

      FormCursorPosLabel->Caption = "x-координата: " +
      IntToStr(X) + ", y-координата: " + IntToStr(Y);
   }
//---------------------------------------------------------------------
```

```
void __fastcall TMainForm::ExitButtonClick(TObject *Sender)
  {
    Application->Terminate();
  }
//---------------------------------------------------------------------
```

Внешний вид приложения вы можете посмотреть, загрузив тестовый проект.

Глава 3

Все, что вы хотели реализовать в C++Builder, но не знали как

Использование стиля csOwnerDrawVariable в TListBox

После рассмотрения собственной отрисовки в ***TComboBox*** с использованием стиля ***csOwnerDrawFixed*** пришло время поговорить об отрисовке со стилем ***csOwnerDrawVariable***. Для некоего разнообразия разберем собственную отрисовку с этим стилем на примере компонента ***TListBox***. Использование данного стиля позволяет каждому элементу списка иметь разную высоту. Кстати, следует отметить, что все, что говорилось о кастомизации внешнего вида относительно ***TComboBox***, абсолютно так же применимо и к ***TListBox***. Справедливо и обратное: все, что сейчас будет рассмотрено в применении к ***TListBox***, полностью аналогично может быть использовано при работе с ***TComboBox***.

Создадим список, в котором будут перечислены все экранные шрифты, установленные в системе. Каждый элемент будет отображен своим собственным шрифтом: элемент, соответствующий шрифту Arial, будет выведен на дисплей шрифтом Arial; элемент списка, соответствующий шрифту Courier, – шрифтом Courier и т. д. Конечный результат нашей работы должен будет выглядеть следующим образом (рис. 3.1).

Рис. 3.1. *Список с элементами, изображенными соответствующими шрифтами*

Нам с самого начала понадобится ссылаться на тестовый проект, поэтому загрузите его в C++Builder. Внешний вид формы тестового проекта приведен на рис. 3.2.

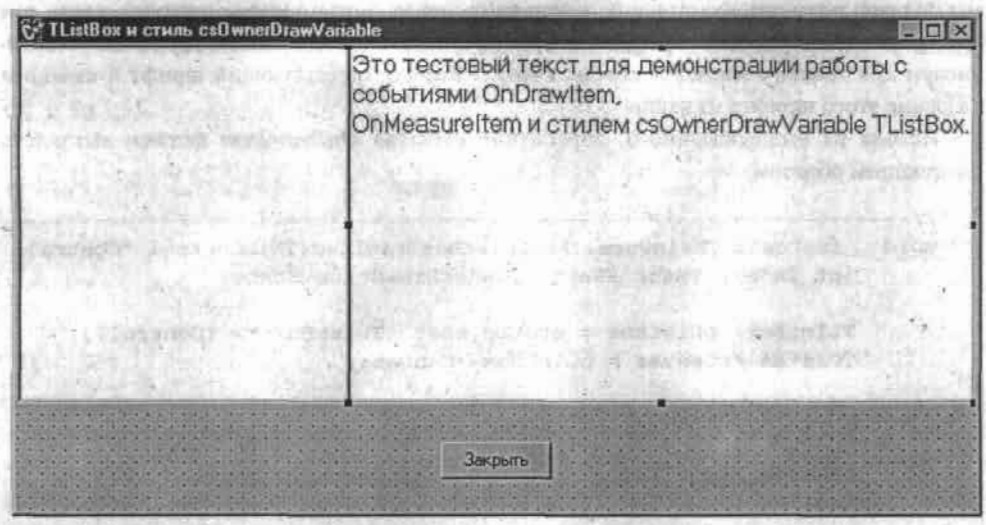


Рис. 3.2. *Внешний вид формы тестового проекта*

На форме ***MainForm*** тестового приложения расположен компонент ***TListBox*** с именем ***FontListBox***. В нем мы будем отображать список экранных шрифтов. Значение свойства ***Style FontListBox*** установлено в ***csOwnerDrawVariable***. В ***TestMemo*** (***TMemo***) будет меняться шрифт текста в соответствии с выбранным элементом в ***FontListBox***. Ну и по кнопке ***CloseButton*** осуществляется выход из приложения.

Начнем с того, что нам уже немного знакомо, – с написания обработчика события ***OnDrawItem*** для ***FontListBox***. Но сначала мы должны загрузить в него список доступных экранных шрифтов, установленных в системе. Это естественно сделать при инициализации нашего тестового приложения – в обработчике события ***OnCreate*** формы. Для получения списка шрифтов использовано свойство ***Fonts*** глобальной переменной ***Screen***.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::FormCreate(TObject *Sender)
    {
      FontListBox->Items->Assign(Screen->Fonts);
    }
//---------------------------------------------------------------------
```

Теперь рассмотрим обработчик события ***OnDrawItem*** списка. Почти все полностью аналогично рассмотренному ранее обработчику этого же события для ***TComboBox***. Сначала мы должны получить указатель на канву компонента, затем должны очистить канву для предотвращения появления артефактов при перерисовке нашего списка. Далее мы устанавливаем для каждого элемента списка ***FontListBox*** соответствующий шрифт и выводим название этого шрифта на канву.

Исходя из вышесказанного, обработчик события ***OnDrawItem*** должен выглядеть следующим образом.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::FontListBoxDrawItem(TWinControl *Control,
      int Index, TRect &Rect, TOwnerDrawState State)
   {
      TListBox *pListBox = static_cast <TListBox *> (Control);
      TCanvas *pCanvas = pListBox->Canvas;

      pCanvas->FillRect(Rect);

      pCanvas->Font->Name = pListBox->Items->Strings[Index];
      pCanvas->TextOut(Rect.Left, Rect.Top, pListBox->Items->Strings[Index]);

   }
//---------------------------------------------------------------------
```

В данных строках:

```
TListBox *pListBox = static_cast <TListBox *> (Control);
TCanvas *pCanvas = pListBox->Canvas;
```

мы получаем указатель ***pCanvas*** на канву компонента ***TListBox***.

Строкой

```
pCanvas->FillRect(Rect);
```

мы закрашиваем прямоугольник ***Rect***, в котором будет выводиться конкретный элемент ***TListBox***, текущей кистью для предотвращения появления артефактов изображения при выделении элемента списка. Если этого не сделать, то при выделении элемента мышью мы увидим следующую картину.

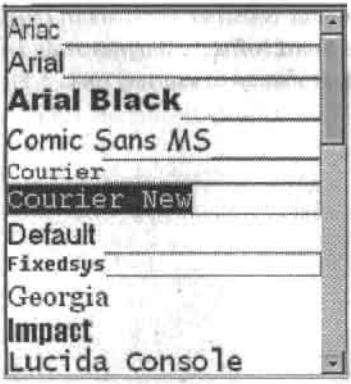


Рис. 3.3. *Артефакты при отсутствии строки с FillRect*

Строкой

```
pCanvas->Font->Name = pListBox->Items->Strings[Index];
```

мы устанавливаем шрифт, которым на канве компонента будут выводиться названия элементов списка. Мы не устанавливаем размер шрифта для каждого элемента, поскольку он задается в свойстве ***Font*** компонента ***TListBox*** для всех элементов сразу.

И наконец, мы выводим название элемента установленным нами шрифтом.

```
pCanvas->TextOut(Rect.Left, Rect.Top, pListBox->Items->Strings[Index]);
```

Теперь давайте посмотрим, что у нас получилось. Запустите тестовый проект, временно изменив значение свойства ***Style*** с ***csOwnerDrawVariable*** на ***csOwnerDrawFixed*** либо отменив назначение обработчика события ***OnMeasureItem***, что в принципе эквивалентно. Мы изменяем стиль списка, чтобы увидеть, к чему приводит отсутствие обработки события ***OnMeasureItem***. Теперь, если вы запустите проект, вы увидите, что список выглядит так, как изображено на рис. 3.4.

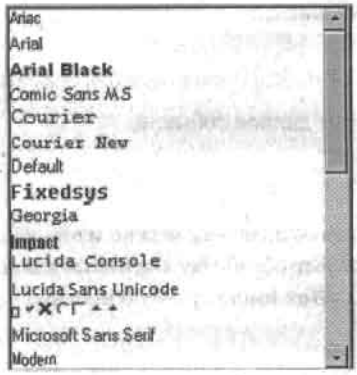


Рис. 3.4. *Внешний вид списка без обработки события OnMeasureItem (со стилем csOwnerDrawFixed)*

Казалось бы, все в порядке! Зачем еще нужна обработка события ***OnMeasureItem***? Но увеличьте размер шрифта (через свойство ***Font*** списка ***FontListBox***) с 8 пунктов до 14. И что вы увидите, запустив приложение? Увидите грустную в общем-то картину (рис. 3.5).

Рис. 3.5. *FontListBox со стилем csOwnerDrawFixed и увеличенным размером шрифта*

Элементы нашего списка налезли друг на друга, и картина совершенно неприглядная. Это происходит из-за того, что у разных шрифтов даже при одинаковых размерах в пунктах может быть разная высота, и именно это их свойство приводит к наблюдаемым нами несуразностям. И смысл обработки события ***OnMeasureItem*** – как раз избежание этих несуразностей и приведение списка со шрифтами в нормальное состояние, изображенное на рис. 3.1. Теперь переходим к написанию обработчика события ***OnMeasureItem***.

Событие ***OnMeasureItem*** объявлено следующим образом.

```
__property TMeasureItemEvent OnMeasureItem = {read=FOnMeasureItem,
write=FOnMeasureItem};
```

Тип ***TMeasureItemEvent*** объявлен так.

```
typedef void __fastcall (__closure *TMeasureItemEvent)
(Controls::TWinControl* Control, int Index,
int &Height);
```

Control – это элемент управления, в котором происходит данное событие.

Index – это индекс элемента в списке.

Height – это высота элемента в списке в пикселах.

Обратите внимание, что высота элемента передается по ссылке – ее можно изменить в обработчике события соответствующим образом, что и делает обработку ***OnMeasureItem*** незаменимым инструментом в случае, когда элементы ***TListBox*** имеют разную высоту.

При установке стиля в ***csOwnerDrawVariable*** событие ***OnMeasureItem*** вызывается перед событием ***OnDrawItem*** (при значении свойства ***Style csOwnerDrawFixed*** событие ***OnMeasureItem*** не происходит), и если мы установим соответствующее значение для параметра ***Height***, то в обработчике события ***OnDrawItem*** каждый элемент будет выводиться с той высотой, какую мы установили в обработчике ***OnMeasureItem***. Таким образом, мы уже знаем путь для решения нашей проблемы. Но теперь перед нами встает вопрос чисто технического характера – измерение высоты каждого отдельного элемента в списке.

Первой мыслью будет использовать свойство ***Height*** класса ***TFont*** по следующей формуле (нижеприведенная строка не является программным кодом. Это просто удобное изображение формулы подсчета высоты шрифта в пикселах).

```
Font->Height = -Font->Size * Font->PixelsPerInch / 72;
```

Но проблема в том, что с полученным нами таким образом значением корректной работы не будет – у всех элементов компонента ***TListBox*** (или, говоря по-другому, у всех шрифтов) по этой формуле получится одинаковая высота, так как размер всех шрифтов в списке будет одинаковым, определяемым значением свойства ***Size*** свойства ***Font*** списка. В качестве другого варианта можно воспользоваться функциями Win32 API для работы со шрифтами, но у них чрезвычайно сложный синтаксис. Однако есть и третий вариант – простой, надежный и удобный в использовании. Нам ведь, по сути дела, необходимо узнать, сколько пикселов займет каждый элемент списка на канве компонента. А в классе ***TCanvas*** есть метод ***TextHeight***, который для текущего шрифта определяет высоту в пикселах строки текста, переданной ему в качестве параметра.

```
int __fastcall TextHeight(const AnsiString Text);
```

Таким образом, если мы в обработчике события установим шрифт канвы в тот шрифт, которым должен быть изображен текущий элемент в списке, и воспользуемся методом ***TextHeight***, мы получим интересующую нас высоту элемента списка в пикселах.

Исходя из этого, обработчик события ***OnMeasureItem*** должен выглядеть следующим образом.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::FontListBoxMeasureItem(TWinControl *Control,
      int Index, int &Height)
   {
      TListBox *pListBox = static_cast <TListBox *> (Control);
      TCanvas *pCanvas = pListBox->Canvas;

      pCanvas->Font->Size = pListBox->Font->Size;
```

```
        pCanvas->Font->Name = pListBox->Items->Strings[Index];
        Height = pCanvas->TextHeight(pCanvas->Font->Name);
    }
//---------------------------------------------------------------------
```

Пояснения заслуживают только три последние строчки.

Строками

```
pCanvas->Font->Size = pListBox->Font->Size;
pCanvas->Font->Name = pListBox->Items->Strings[Index];
```

мы устанавливаем шрифт для канвы ***TListBox***. Размер шрифта берем равным значению свойства ***Size*** шрифта списка, а имя шрифта – текст того элемента, для которого высчитывается высота.

Строкой

```
Height = pCanvas->TextHeight(pCanvas->Font->Name);
```

мы вычисляем высоту элемента в пикселах и присваиваем ее параметру ***Height*** обработчика. Таким образом, будет вычислена высота каждого элемента списка в зависимости от того, название какого шрифта он содержит.

Остальной код проекта не представляет интереса. В обработчике события ***OnClick*** списка изменяется шрифт текста в ***TestMemo*** соответственно с выбранным элементом.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::FontListBoxClick(TObject *Sender)
    {
        TestMemo->Font->Name = FontListBox->Items->Strings[FontListBox-
        >ItemIndex];
    }
//---------------------------------------------------------------------
```

И в обработчике нажатия кнопки реализован выход из приложения.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::CloseButtonClick(TObject *Sender)
    {
        Application->Terminate();
    }
//---------------------------------------------------------------------
```

Разумеется, собственная отрисовка в ***TComboBox*** и ***TListBox*** не ограничивается только рассмотренными нами примерами. Вы можете, например, в обработчике события ***OnDrawItem*** выводить битовые изображения для каждого элемента. Это не представляет никаких трудностей, поэтому оставляется на самостоятельное упражнение.

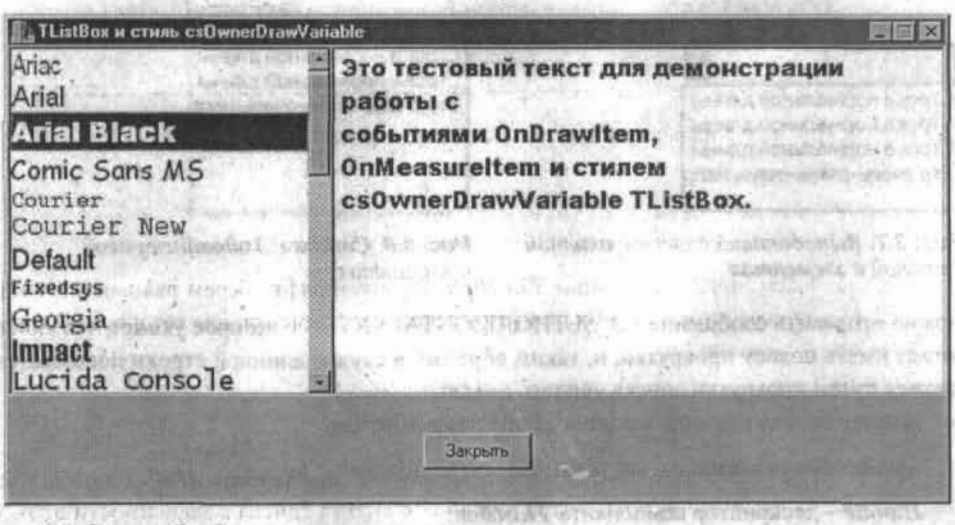


Рис. 3.6. *Внешний вид запущенного приложения*

Улучшение интерфейса TListBox и TComboBox

TListBox, ***TComboBox***. VCL-обертки стандартных элементов управления Windows. Просты настолько, что, кажется, проще них может быть только ***TLabel*** и ***TButton***. И это в общем-то так, но вам никогда не приходило в голову, что эти два компонента списков можно сделать более удобными для использования? Если не приходило, то сейчас я вам покажу то, на что вы, вероятно, не обращали внимания. А именно: крайнюю неудобность использования стандартных списков при работе с длинными строками, которые не помещаются в элементы управления. Откройте C++Builder, разместите на форме компоненты ***TListBox*** и ***TComboBox*** и занесите в Инспекторе объектов в свойства ***Items*** этих компонентов следующие строки:

Строка нормальной длины

Строка нормальной длины

Строка нормальной длины

Это очень-очень-очень непропорционально длинная строка для тестирования

При запуске проекта списки будут выглядеть следующим образом (рис. 3.7 и 3.8).

Представьте себя на месте пользователя программы с такими списками (да вы наверняка и сами встречали программы с длинными строками в данных элементах управления). Вряд ли у вас найдется хоть одно хорошее слово в адрес разработчиков такой программы. Но выход есть. Элементу управления "список" (оберткой которого является компонент ***TListBox***)

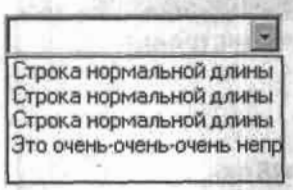


Рис. 3.7. *Выпадающий список с длинной строкой в элементах*

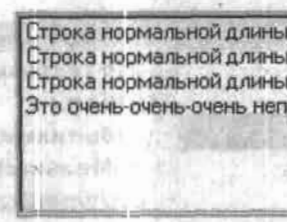


Рис. 3.8. *Список с длинной строкой в элементах*

можно отправить сообщение LB_SETHORIZONTALEXTENT, которое укажет, что список может иметь полосу прокрутки, и, таким образом, в случае длинной строки пользователь сможет путем прокрутки списка увидеть ее всю.

Данное сообщение отправляется следующим образом.

```
SendMessage(Handle, LB_SETHORIZONTALEXTENT, Length, 0);
```

Handle – дескриптор компонента ***TListBox***;

Length – количество пикселов, на которые будет прокручиваться список по горизонтали.

Если присвоить этому параметру длину наибольшей строки в списке в пикселах, то мы получим как раз тот эффект, что нам необходим.

Таким образом, задача разделяется на несколько этапов.

- Нахождение самой длинной строки в ***TListBox***.
- Определение ее длины в пикселах.
- Отправка сообщения списку.

Первые два этапа реализуются следующим кодом.

```
int Length = 0;
TCanvas *pCanvas = new TCanvas();
Canvas->Handle = GetDC(ListBox->Handle);

  for(int i = 0; i < pListBox->Items->Count; i++)
    {
      AnsiString Text = pListBox->Items->Strings[i];
      int TempLength = pCanvas->TextWidth(Text);
      if(TempLength > Length)
      Length = TempLength;
    }

delete pCanvas;
```

Строками

```
TCanvas *pCanvas = new TCanvas();
pCanvas->Handle = GetDC(ListBox->Handle);
```

мы создаем канву и назначаем ее списку (***ListBox*** – компонент ***TListBox***, для которого реализуется горизонтальная прокрутка).

Далее в цикле мы последовательно проходим по всем строкам списка. Внутри цикла определяем длину каждой строки в пикселах с помощью метода ***TextWidth*** канвы.

```
int TempLength = pCanvas->TextWidth(Text);
```

К моменту завершения цикла размер в пикселах наиболее длинной строки будет находиться в переменной ***Length***.

И напоследок не забываем об удалении канвы.

```
delete pCanvas;
```

После этого мы можем уже отправлять сообщение.

```
SendMessage(ListBox->Handle, LB_SETHORIZONTALEXTENT, Length, 0);
```

Теперь перейдем к компоненту ***TComboBox***.

Для элемента управления "выпадающий список" нет сообщения, которое бы у списка включало полосу прокрутки, но есть сообщение CB_SETDROPPEDWIDTH, которое позволяет данному элементу управления расширить его выпадающую часть до нужных размеров.

CB_SETDROPPEDWIDTH отправляется аналогичным LB_SETHORIZONTALEXTENT образом.

```
SendMessage(Handle, CB_SETDROPPEDWIDTH, Length, 0);
```

Handle – дескриптор выпадающего списка;

Length – ширина его выпадающей части в пикселах.

Задача увеличения ширины выпадающей части списка делится на те же три этапа, что и при установке линейки прокрутки для обычного списка. Поэтому для вычисления размера самой длинной строки в пикселах можно воспользоваться приведенным выше кодом, после чего отправить сообщение

```
SendMessage(ComboBox->Handle, CB_SETDROPPEDWIDTH, Length, 0);
```

После теории переходим к практике.

Загрузите в C++Builder тестовый проект. Вид главной формы его представлен на рис. 3.9.

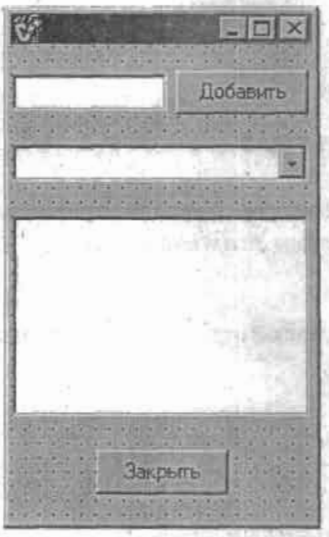


Рис. 3.9. *Форма тестового проекта во время разработки*

Как вы видите, на форме расположен компонент ***TEdit***, кнопка (***TButton***), по нажатии на которую текст из ***TEdit*** заносится в выпадающий список (***TComboBox***) и в обычный список (***TListBox***).

Поскольку код, определяющий размер (в пикселах) самой длинной строки, для обоих списков практически идентичен, я оформил его в виде функции ***GetTextLength***, принимающей в качестве параметра указатель на экземпляр компонента, а возвращающей число пикселов самой его длинной строки (объявление данной функции смотрите в заголовочном файле формы).

```
//---------------------------------------------------------------------
int TMainForm::GetTextLength(TWinControl *AControl)
  {
    TComboBox *pComboBox = dynamic_cast <TComboBox *> (AControl);
    TListBox *pListBox = dynamic_cast <TListBox *> (AControl);

    int Length = 0;

    if(pComboBox)
      {
        TCanvas *pCanvas = new TCanvas();
        pCanvas->Handle = GetDC(pComboBox->Handle);
        for(int i = 0; i < pComboBox->Items->Count; i++)
          {
            AnsiString Text = pComboBox->Items->Strings[i];
```

```
            int TempLength = pCanvas->TextWidth(Text);
            if(TempLength > Length)
              Length = TempLength;
          }
        delete pCanvas;
        return Length;
      }
    else if(pListBox)
      {
        TCanvas *pCanvas = new TCanvas();
        pCanvas->Handle = GetDC(pListBox->Handle);
        for(int i = 0; i < pListBox->Items->Count; i++)
          {
            AnsiString Text = pListBox->Items->Strings[i];

            int TempLength = pCanvas->TextWidth(Text);
            if(TempLength > Length)
              Length = TempLength;
          }
        delete pCanvas;
        return Length;
      }
    else
      return 0;
  }
//---------------------------------------------------------------------------
```

Теперь давайте определимся, где мы должны отправлять соответствующие сообщения спискам. Компоненту ***TListBox*** мы должны пересчитать длину строк и отправить сообщение сразу после добавления в него новых элементов, то есть в обработчике события ***OnClick*** кнопки.

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::AddButtonClick(TObject *Sender)
  {
    ComboBox->Items->Add(Edit->Text);
    ListBox->Items->Add(Edit->Text);
    SendMessage(ListBox->Handle, LB_SETHORIZONTALEXTENT,
    GetTextLength(ListBox) + 10, 0);
  }
//---------------------------------------------------------------------------
```

Выпадающему списку нет необходимости отправлять сообщение сразу после добавления в него новых элементов – это необходимо делать в тот момент, когда список раскрывается и отображается его выпадающая часть. Для этих целей подходит событие ***OnDropDown***.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::ComboBoxDropDown(TObject *Sender)
  {
    SendMessage(ComboBox->Handle, CB_SETDROPPEDWIDTH,
    GetTextLength(ComboBox) + 10, 0);
  }
//---------------------------------------------------------------------
```

Вы, наверное, обратили внимание, что к возвращаемому функцией ***GetTextLength*** размеру самой длинной строки прибавлено еще 10 пикселов. Это сделано для более красивого внешнего вида. Попробуйте не прибавлять это значение – увидите, что без него хуже.

Естественно, что если вы заполняете списки строками через Инспектор объектов во время разработки, вы должны будете в конструктор формы, например, также добавить отправку сообщения компоненту ***TListBox***. Для ***TComboBox*** при использовании события ***OnDropDown*** это не критично, а вот обычному списку просто необходимо.

На этом практическая часть закончена. Запустите тестовый проект и добавьте несколько очень длинных строк в оба списка. После этого посмотрите, как стал выглядеть выпадающий список.

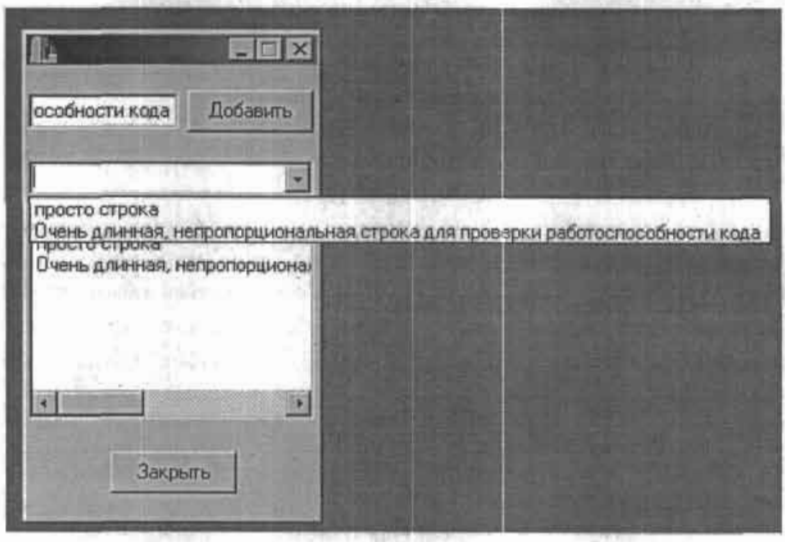


Рис. 3.10. *Выпадающий список во время выполнения*

И обратите внимание на список, представленный компонентом ***TListBox***. На рисунке он изображен с полностью прокрученным направо содержимым (рис. 3.11).

Рис. 3.11. *Использование полосы прокрутки у TListBox во время выполнения*

Ну что? Мне кажется, пользователи ваших программ с такими списками вряд ли скажут в ваш, как разработчика, адрес, нехорошие слова, если вы сможете, а теперь вы наверняка сможете, осчастливить их такими улучшенными элементами управления. Кстати, чуть не забыл – использовать сообщение CB_SETDROPPEDWIDTH вы можете только с теми компонентами ***TComboBox***, у которых свойство ***Style*** установлено в любое значение, отличное от ***csSimple***.

Реализация заставки (splash screen) в C++Builder

Итак, splash screen. Он же – окно-заставка, или просто заставка. Это окно, которое отображается на экране при старте запущенного приложения и демонстрирует пользователю процесс инициализации и загрузки приложения. Наглядный пример, который видит каждый программист, использующий C++Builder, – это окно, выводимое при старте среды разработки. Для C++Builder пятой версии данное окно выглядит следующим образом.

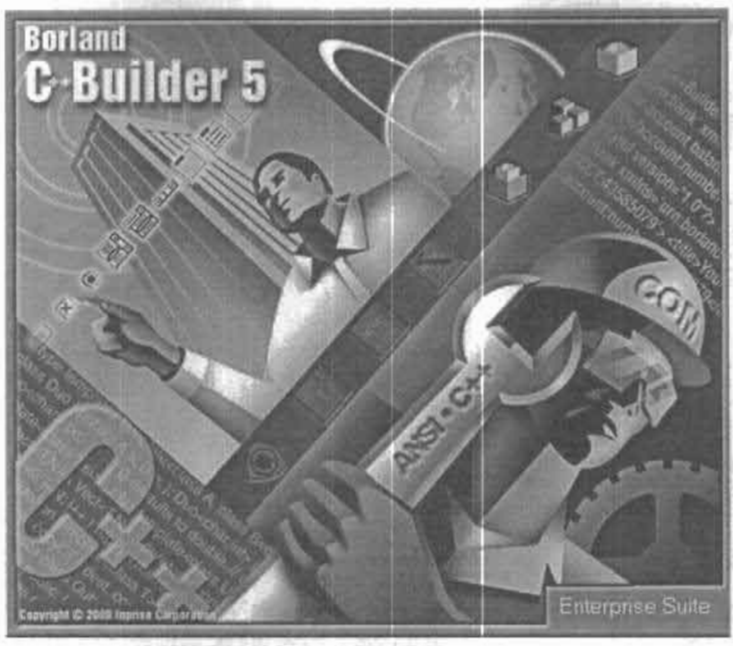


Рис. 3.12. *Splash screen при запуске C++Builder 5*

Реализация таких окон-заставок – дело десяти минут для знающего программиста. Превращением вас в "знающих программистов" мы сейчас и займемся.

Загрузите тестовый проект в C++Builder. Внешний вид тестового проекта совершенно не интересен, там все формы, за исключением формы для заставки, пустые и добавлены в проект только для демонстрации самого подхода, в связи с чем здесь и не приводится. А вот внешний вид одной из форм, той, которая и является собственно заставкой, стоит посмотреть.

Стандартно: рисунок на форме (использован компонент ***TImage***), надписи, поясняющие, что в данный момент грузится (у нас заменено просто меткой с заголовком "Загрузка"), и индикатор процесса. Значение свойства ***BorderStyle*** формы установлено в ***bsNone***.

Теперь запустите тестовый проект. При запуске проекта у вас сначала появится заставка с бегущим индикатором и лишь потом формы проекта.

Рис. 3.13. *Форма-заставка во время разработки*

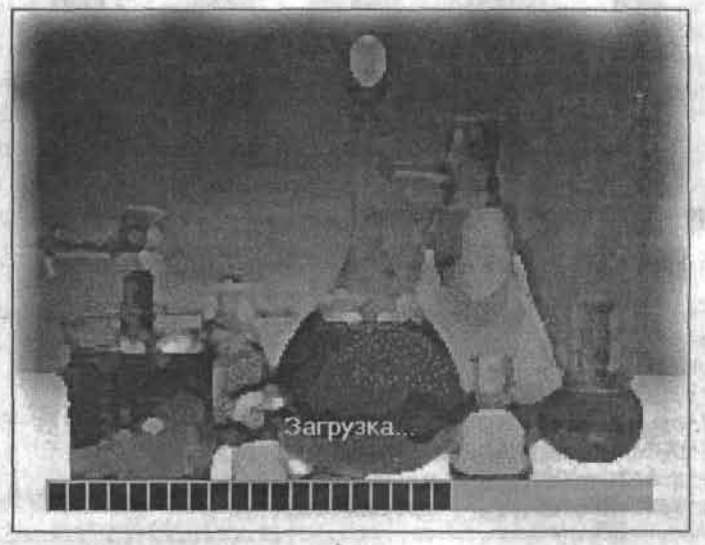


Рис. 3.14. *Форма-заставка во время выполнения*

Как вы видите, все "по-взрослому": при запуске приложения показывается окно-заставка, в котором отображается ход инициализации приложения... и для этого надо буквально всего лишь несколько строк. Правда там, куда вы, вполне возможно, еще не заглядывали – непосредственно в файл проекта.

Но перед тем как смотреть код файла проекта, надо упомянуть о следующем. Форма заставки (***SplashScreenForm***) должна быть исключена из автосоздаваемых форм проекта, и список форм тестового проекта выглядит следующим образом.

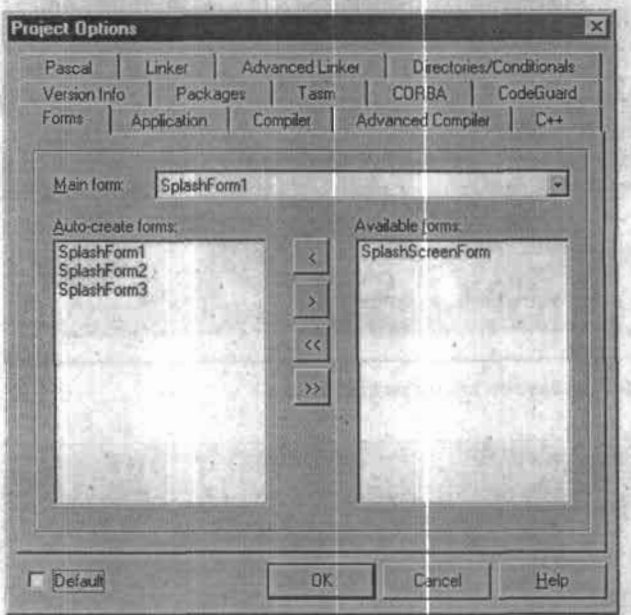


Рис. 3.15. *Список форм тестового проекта*

Теперь переходим собственно к коду файла проекта – SplashProject.cpp.

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
USERES("SplashProject.res");
USEFORM("SpalshUnit1.cpp", SplashForm1);
USEFORM("SplashFormUnit.cpp", SplashScreenForm);
USEFORM("SpalshUnit2.cpp", SplashForm2);
USEFORM("SpalshUnit3.cpp", SplashForm3);
//---------------------------------------------------------------------
#include "SplashFormUnit.h"
```

```
//---------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
        {
            SplashScreenForm = new TSplashScreenForm(Application);
            SplashScreenForm->Show();
            SplashScreenForm->Update();

            Application->Initialize();
           Application->CreateForm(__classid(TSplashForm1), &SplashForm1);
            SplashScreenForm->SplashScreenProgressBar->StepIt();
           Application->CreateForm(__classid(TSplashForm2), &SplashForm2);
            SplashScreenForm->SplashScreenProgressBar->StepIt();
           Application->CreateForm(__classid(TSplashForm3), &SplashForm3);
            SplashScreenForm->SplashScreenProgressBar->StepIt();

            Sleep(500);
            SplashScreenForm->Hide();
            SplashScreenForm->Close();

            Application->Run();
        }
    catch (Exception &exception)
        {
            Application->ShowException(&exception);
        }
    return 0;
}
//---------------------------------------------------------------------
```

Первое, что должно привлечь ваше внимание: мы создаем форму заставки вручную.

```
SplashScreenForm = new TSplashScreenForm(Application);
```

Именно для этого мы убирали ее из списка автосоздаваемых при старте приложения форм.

Второе: для создания ***SplashScreenForm*** мы подключили в файл проекта заголовочный файл этой формы.

```
#include "SplashFormUnit.h"
```

Без этой строки компилятор будет выдавать ошибку

[C++ Error] SplashProject.cpp(16): E2450 Undefined structure 'TSplashScreenForm'

и ряд других. И будет прав, ибо без заголовочного файла тип ***TSplashScreenForm*** в файле проекта просто неизвестен.

Далее, после создания формы мы ее отображаем на экране.

```
SplashScreenForm->Show();
SplashScreenForm->Update();
```

Что будет без строки с ***Show()***, я думаю, вам ясно. А строка с ***Update()*** нужна для отрисовки изображения в ***SplashScreenImage***. Без ***Update()*** наша заставка будет без картинки.

Строка

```
SplashScreenForm->SplashScreenProgressBar->StepIt();
```

увеличивает позицию индикатора после создания и инициализации одной из остальных форм проекта, а последовательность строк

```
Application->CreateForm(__classid(TSplashForm1), &SplashForm1);
SplashScreenForm->SplashScreenProgressBar->StepIt();
Application->CreateForm(__classid(TSplashForm2), &SplashForm2);
SplashScreenForm->SplashScreenProgressBar->StepIt();
Application->CreateForm(__classid(TSplashForm3), &SplashForm3);
SplashScreenForm->SplashScreenProgressBar->StepIt();
```

собственно и дает визуализацию процесса загрузки приложения. Отмечу, что у индикатора на форме (***SplashScreenProgressBar***) для корректного отображения процесса загрузки значение свойства ***Min*** должно быть установлено в 0, значение свойства ***Max*** – в число форм проекта минус единица (форму заставки не учитываем). Значение свойства ***Position*** должно быть установлено в 1.

И наконец, строки

```
SplashScreenForm->Hide();
SplashScreenForm->Close();
```

просто убирают заставку с экрана и закрывают форму, в которой уже отпала необходимость.

Вы наверняка обратили внимание на строку

```
Sleep(500);
```

Это сделано для того, чтобы пользователь мог увидеть окончание процесса инициализации приложения. Без данной строки последнего увеличения индикатора пользователь не увидит. Такие же строки есть и в конструкторах всех остальных форм проекта.

```
//---------------------------------------------------------------------
__fastcall TSplashForm1::TSplashForm1(TComponent* Owner)
    : TForm(Owner)
    {
      Sleep(500);
    }
//---------------------------------------------------------------------
```

Дело в том, что формы у нас пустые и их создание происходит буквально за мгновения, – вы на мощной машине даже не успеете увидеть splash screen. И вот для того чтобы процесс отображения заставки занимал разумное время и был вообще наблюдаем, в конструкторы форм введена задержка в полсекунды. Разумеется, в больших серьезных приложениях это просто лишнее – создание сложных форм со множеством компонентов занимает и так достаточно долгое время.

Ну вот, собственно, и все, что можно рассказать о создании заставок. Выдумывайте, пробуйте, творите. А рисунок взят из подкаталога Images каталога Borland Shared.

TEdit и OnKeyPress

Кажется, что компонент ***TEdit*** настолько прост, что ничего нового про работу с ним рассказать уже нельзя. Но это не так. И сейчас мы разберем некоторые аспекты работы с ним, которые довольно часто обсуждаются в различных эхоконференциях о C++Builder.

Одним из таких аспектов является ограничение пользователя в символах, которые он может ввести в ***TEdit***. Речь не идет о максимальной длине текста – для этого у компонента ***TEdit*** есть свойство ***MaxLength***, значение которого указывает, какое максимальное количество символов пользователь может ввести. Речь идет об ограничении другого рода – например, пользователю необходимо предоставить возможность вводить только определенные символы, или необходимо корректировать пользовательский ввод. Также может понадобиться сообщать пользователю о недопустимом для ввода символе. Именно такие случаи мы сейчас рассмотрим.

Простейший случай ограничения пользовательского ввода: в компонент ***TEdit*** пользователь должен иметь возможность вводить только цифры. Так как ***TEdit*** – это всего лишь обертка библиотеки VCL над стандартным элементом управления "строка ввода" Windows, то решение, которое напрашивается само собой, – воспользоваться стилем ES_NUMBER следующим образом:

```
SetWindowLong(TestEdit->Handle, GWL_STYLE,
GetWindowLong(TestEdit->Handle, GWL_STYLE) | ES_NUMBER);
```

где ***TestEdit*** – имя вашего компонента ***TEdit***. (Здесь и далее в коде ***TestEdit*** – имя компонента ***TEdit***, размещенного на форме с именем ***MainForm***.)

После этого при непосредственном вводе символа в компонент пользователь сможет ввести только цифры. К сожалению, у данного способа есть недостаток – из буфера обмена в компонент по-прежнему можно ввести любой алфавитно-цифровой символ. К сожалению, стандартного способа обработки вставки из буфера обмена нет, а ниже мы рассмотрим предлагаемый мною способ обработки данной ситуации.

Способ через изменение стиля компонента, к сожалению, имеет очень ограниченное применение. Он позволяет сделать компонент восприимчивым только к вводу цифр, и ничего больше. Если же необходимо другое ограничение ввода для ***TEdit***, то необходимо воспользоваться универсальным способом – через обработку события ***OnKeyPress*** компонента.

TEdit наследует событие ***OnKeyPress*** от компонента ***TWinControl***. Событие ***OnKeyPress*** объявлено в ***TWinControl*** следующим образом.

```
typedef void __fastcall (__closure *TKeyPressEvent)(System::TObject* Sender,
char &Key);
```

Оно происходит, когда пользователь нажимает на клавиатуре клавишу, соответствующую одиночному символу. Значение параметра ***Sender*** определяет компонент, в котором произошло событие, а значение параметра ***Key*** – нажатую клавишу. Параметр ***Key*** передается по ссылке, что позволяет корректировать ввод пользователя в обработчике события.

Теперь посмотрим, как решить нашу задачу о вводе только цифр в компонент ***TEdit*** с помощью обработчика события ***OnKeyPress***.

Разумеется, непосредственно перечислять в обработчике события все цифры, чтобы ограничить ими ввод пользователя, будет не очень разумно. Для подобных целей Win32 API предоставляет нам ряд функций для работы со строками и символами. Данные функции объявлены в файле winuser.h, который подключается в windows.h, который в свою очередь автоматически подключается к каждому VCL-проекту, так что вы можете использовать данные функции без каких-либо дополнительных действий. В данном случае нам подойдет функция ***IsCharAlpha***.

```
BOOL IsCharAlpha(TCHAR ch);
```

Данная функция возвращает ***true***, если символ, определяемый значением параметра ***ch***, является алфавитным символом, и ***false*** – в противном случае.

Тогда обработчик события ***OnkeyPress*** компонента ***TEdit*** можно написать следующим образом.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::TestEditKeyPress(TObject *Sender, char
&Key)
   {
      if(IsCharAlpha(Key))
         Key = 0;
   }
//---------------------------------------------------------------------
```

Строка

```
Key = 0;
```

предотвращает ввод алфавитного символа.

К сожалению, кроме цифр будут вводиться и другие символы, такие, как, например, "?", "*" и т. п. Чтобы ограничить ввод этих символов, нам придется перечислять их в обработчике события (ниже мы рассмотрим способ, позволяющий избежать перечисления).

```
//---------------------------------------------------------------------
void __fastcall TMainForm::TestEditKeyPress(TObject *Sender, char &Key)
   {
      if(IsCharAlpha(Key) || Key == '*' || Key == '&')
         Key = 0;
   }
//---------------------------------------------------------------------
```

Разумеется, здесь перечислены не все символы, которые недопустимы к вводу, а только "*" и "&". Остальные символы должны быть перечислены в операторе ***if*** аналогичным образом. Перечисляя символы, не сильно увлекайтесь – пробел и Backspace, например, тоже относятся к алфавитным символам, так что вам необходимо быть внимательными, чтобы случайно не запретить их ввод.

А что же у нас с буфером обмена? С буфером обмена та же грустная история – через него вставить можно любой символ. Нам также необходимо либо запрещать для данного элемента управления использование буфера обмена вовсе, либо обрабатывать содержимое буфера обмена на предмет наличия в нем запрещенных символов.

Напишем функцию, которая бы проверяла корректность содержимого буфера обмена. Если в содержимом буфера присутствует хотя бы один алфавитный символ, то функция будет возвращать ***false***. Если же содержимое буфера обмена состоит целиком из цифр, то функция будет возвращать ***true***.

Ниже приведен текст данной функции.

```
//---------------------------------------------------------------------
bool __fastcall TMainForm::ClipboardCheck(void)
   {
      if(Clipboard()->HasFormat(CF_TEXT))
         {
            bool ReturnFlag = true;
            int BufferSize = Clipboard()->AsText.Length() + 1;

            char *Buffer = new char[BufferSize];

            ZeroMemory(Buffer, BufferSize);
            Clipboard()->GetTextBuf(Buffer, BufferSize);

            for(int i = 0; i < BufferSize - 1; i++)
               if(IsCharAlpha(Buffer[i]))
                  {
```

```
                ReturnFlag = false;
                break;
            }

        delete [] Buffer;

        return ReturnFlag;
      }
    // конец оператора if(Clipboard()->HasFormat(CF_TEXT))

    return false;
  }
//---------------------------------------------------------------------
```

Сначала мы проверяем, что вообще содержится в буфере обмена. Если данные в буфере обмена не в текстовом формате, то мы сразу выходим из функции, возвращая ***false***. Если данные в текстовом формате, то сначала мы устанавливаем флаг возврата в ***true***.

```
bool ReturnFlag = true;
```

Затем узнаем длину текста в буфере обмена и увеличиваем ее на единицу. Это необходимо для завершающего нулевого символа.

```
int BufferSize = Clipboard()->AsText.Length() + 1;
```

После создания временного буфера, в котором будет находиться содержимое буфера обмена:

```
char *Buffer = new char[BufferSize];
```

обнуляем временный буфер:

```
ZeroMemory(Buffer, BufferSize);
```

и помещаем содержимое буфера обмена в созданный нами буфер:

```
Clipboard()->GetTextBuf(Buffer, BufferSize);
```

Далее проходим посимвольно по всему содержимому буфера обмена, используя переменную ***Buffer***, за исключением завершающего нулевого символа:

```
for(int i = 0; i < BufferSize - 1; i++)
```

и каждый символ мы проверяем на соответствие:

```
if(IsCharAlpha(Buffer[i]))
```

Если символ не является алфавитным символом (является цифрой), то переходим к следующей итерации цикла. Если символ является алфавитным символом, который, согласно условиям нашей задачи, не должен вводиться в элемент управления, то мы устанавливаем значение флага возврата в ***false*** и досрочно выходим из цикла ***for***.

```
ReturnFlag = false;
break;
```

Напоследок не забываем освобождать память:

```
delete [] Buffer;
```

и возвращаем значение:

```
return ReturnFlag;
```

Теперь нам необходимо вызывать эту функцию всякий раз, когда мы выполняем вставку из буфера обмена в компонент. И тут нас подстерегают трудности. Как я уже говорил выше, стандартного способа отследить момент вставки из буфера обмена не существует. Я предлагаю следующий путь. Хотя и весьма трудоемкий, но работающий.

На форме, где расположен нужный нам компонент ***TEdit***, разместите компонент ***TActionList*** и компонент ***TPopupMenu***. Назначьте (через свойство ***PopupMenu*** компонента ***TEdit***) всплывающее меню вашему ***TEdit***. В списке действий (***TActionList***) создайте новое стандартное действие ***TEditPaste***, во всплывающем меню создайте пункт меню, для которого укажите данную акцию (через свойство ***Action*** пункта меню). Теперь останется написать обработчик события ***OnExecute*** стандартного действия.

Обработчик должен выглядеть следующим образом.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::EditPaste1Execute(TObject *Sender)
   {
      if(ClipboardCheck())
         TestEdit->PasteFromClipboard();
   }
//---------------------------------------------------------------------
```

Если содержимое буфера обмена проходит проверку функцией ***ClipboardCheck***, то мы выполняем вставку из буфера методом ***PasteFromClipboard***. Если же функция ***ClipboardCheck*** вернула ***false***, то вставки содержимого буфера обмена не происходит.

Конечно, мы рассмотрели самый простой случай с цифрами. Теперь рассмотрим другой случай. Случай в общем-то теоретический, но он нужен для демонстрации еще одного способа проверки содержимого буфера обмена на корректность символов. После рассмотрения данной ситуации вы легко справитесь с решением конкретной задачи. Итак, пусть нам необходимо разрешить в ***TEdit*** вводить любые символы, кроме тех, которые представляют гласные звуки латинского алфавита, а именно кроме "a", "e", "i", "o", "u".

Обработчик события ***OnKeyPress*** для такого случая может выглядеть аналогично рассмотренному выше, но перечислять все доступные для ввода символы неудобно – их слишком много. Поэтому перепишем обработчик следующим образом.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::TestEditKeyPress(TObject *Sender, char &Key)
   {
      AnsiString ForbiddenChars = "aeiou";

      if(ForbiddenChars.Pos(Key))
         Key = 0;
   }
//---------------------------------------------------------------------
```

В переменной ***ForbiddenChars*** мы перечисляем символы, ввод которых запрещен. Если введен запрещенный символ, то метод ***Pos*** класса ***AnsiString*** вернет позицию данного символа в строке ***ForbiddenChars***. Это значение будет отлично от нуля, и такой символ будет пропущен при вводе. Разумеется, такой способ обработки вводимых символов можно использовать и для цифр, и вообще для чего угодно – он удобнее, чем тот, что мы рассмотрели ранее, и я чаще всего использую именно этот способ. Но если мне надо запретить к вводу один-два символа, я использую приведенный выше вариант с перечислением.

Можно аналогичным образом переписать и функцию проверки содержимого буфера обмена. Выглядеть она будет следующим образом.

```
//---------------------------------------------------------------------
bool __fastcall TMainForm::ClipboardCheck(void)
   {
      if(Clipboard()->HasFormat(CF_TEXT))
         {
            AnsiString ForbiddenChars = "aeiou";
            bool ReturnFlag = true;
            int BufferSize = Clipboard()->AsText.Length() + 1;

            char *Buffer = new char[BufferSize];

            ZeroMemory(Buffer, BufferSize);
            Clipboard()->GetTextBuf(Buffer, BufferSize);

            for(int i = 0; i < BufferSize - 1; i++)
               if(ForbiddenChars.Pos(Buffer[i]))
                  {
                     ReturnFlag = false;
                     break;
                  }
```

```
            delete [] Buffer;
            return ReturnFlag;
        }
      // конец оператора if(Clipboard()->HasFormat(CF_TEXT))
      return false;
  }
//---------------------------------------------------------------------
```

Дальнейшие шаги для того, чтобы "неправильные" символы не попали в элемент управления при вставке из буфера обмена, те же: создаем в ***TActionList*** стандартное действие ***TEditPaste***, в обработчике события ***OnExecute*** которого пишем:

```
//---------------------------------------------------------------------
void __fastcall TMainForm::EditPaste1Execute(TObject *Sender)
  {
    if(ClipboardCheck())
        TestEdit->PasteFromClipboard();
  }
//---------------------------------------------------------------------
```

Затем создаем на форме всплывающее (popup) меню, данное меню присваиваем ***TestEdit*** через свойство ***PopupMenu***, а для одного из пунктов данного меню назначаем созданное стандартное действие через свойство ***Action*** пункта меню.

По кастомизации ввода в компонент ***TEdit*** почти все. Но только почти. У многих пользователей может возникнуть непонимание относительно того, почему они не могут ввести ряд символов в элемент управления. Хорошим тоном считается сообщить пользователю о причинах невозможности ввода. Вы можете реализовать информирование пользователя через метод ***MessageBox*** класса ***TApplication*** примерно так:

```
//---------------------------------------------------------------------
void __fastcall TMainForm::TestEditKeyPress (TObject *Sender, char &Key)
   {
      if(Key == 'a')
         {
            Key = 0;
            Application->MessageBox("Ввод символа 'a' недопустим!",
"Ошибка!",
            MB_OK | MB_ICONERROR);
         }
   }
//---------------------------------------------------------------------
```

Чем неудобно такое предупреждение? Тем, что на экран выводится дополнительное окно и пользователю необходимо его закрывать. Но можно предупредить пользователя звуковым сигналом, тогда никаких дополнительных окон ему закрывать не придется.

Для вывода звукового сигнала в Win32 API существует функция ***PlaySound***.

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

Данная функция объявлена в файле Mmsystem.h, так что перед ее использованием не забудьте подключить данный файл к проекту. Обратите внимание, что данный файл должен быть подключен после строки.

```
#include <vcl.h>
```

Значение параметра ***pszSound*** определяет проигрываемый звук, значение параметра ***hmod*** должно быть равно NULL, если будет проигрываться звук, не находящийся в ресурсах исполняемого файла, а значением параметра ***fdwSound*** устанавливаются флаги воспроизведения.

Вообще говоря, использование функции ***PlaySound*** – материал для отдельного рассмотрения, ибо число флагов и условий, при которых действует тот или иной флаг, достаточно велико. Для информирования пользователя нам необходимо вспомнить, что для сообщения об ошибках обычно используется воспроизведение стандартного звука через динамик компьютера (на тот случай, если компьютер не оборудован звуковой картой). Данному звуку соответствует в системном реестре операционной системы ассоциация "Beep". Поэтому, для того чтобы сообщить пользователю о допущенной им ошибке ввода, использование функции ***PlaySound*** выглядит следующим образом.

```
PlaySound("Beep", NULL, SND_ALIAS);
```

Тогда информирование пользователя об ошибке ввода может быть реализовано так, как показано ниже.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::TestEditKeyPress(TObject *Sender, char &Key)
   {
      if(Key == 'a')
         {
            Key = 0;
            PlaySound("Beep", NULL, SND_ALIAS);
         }
   }
//---------------------------------------------------------------------
```

Теперь пользователь при вводе символа "а" услышит звуковое предупреждение. Оба способа сообщения об ошибках ввода пользователя допустимы и имеют право на существование. Выберите один из них по вашему усмотрению.

Манипуляции с методами классов, или Как вызвать функцию по ее символьному имени

В качестве отрицательных сторон библиотеки VCL часто называется ее дельфийское происхождение, что приводит к тому, что по некоторым моментам поведение объектов VCL-классов не соответствует стандарту C++. К таким моментам, в частности, можно отнести своеобразный порядок вызова конструкторов базовых "дельфийных" классов, поведение виртуальной функции при вызове ее в теле конструктора, ограничения, накладываемые при использовании множественного наследования (до появления C++Builder 6 разговор велся не просто об ограничении, а о недопустимости применения множественного наследования для VCL-классов).

Кроме того, в компилятор для поддержки VCL-библиотеки были добавлены расширения, что тоже не приветствуется сторонниками чистоты C++. К таким расширениям относится введение ключевого слова ***__closure*** – указателя на метод класса. В отличие от предусмотренного стандартом C++ указателя на метод класса, ***__closure*** кроме самого адреса метода, хранит еще и адрес экземпляра класса и физически представляет собой структуру, состоящую из двух указателей: на экземпляр класса и на метод класса. Таким образом, ***__closure*** практически является указателем не просто на метод класса, а на метод объекта (экземпляра) класса.

Не стоит думать, что применение указателя на метод объекта узко ограничено лишь областью VCL-классов. Подобные указатели, хотя и не часто, встречаются в программистской практике и оказываются довольно полезны. Имеются реализации таких указателей с использованием стандартного C++. (*Александреску А.* Современное проектирование на C++. М.: Издательский дом "Вильямс", 2002.) В C++Builder программист получает эти возможности даром, в качестве своеобразной компенсации за "моральный ущерб" в результате потери совместимости со стандартом.

Для иллюстрации этих возможностей создадим простой проект, состоящий из одной формы. На форму положим пять кнопок и отредактируем им свойство ***Caption*** в соответствии с задачами, решение которых попытаемся продемонстрировать.

Вот программа-минимум, которую мы должны выполнить:

- вызов обычной функции как метода в качестве обработчика события;
- вызов метода как обычной функции;
- вызов опубликованного (***published***) метода по его символьному имени;
- получение имени опубликованного метода.

В соответствии с этими задачами наша форма примет следующий вид.

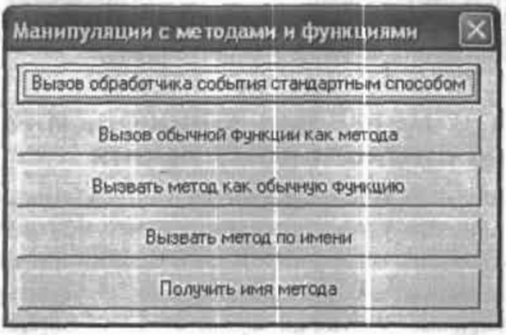


Рис. 3.16. *Запущенный тестовый проект*

При нажатии самой верхней кнопки будет вызван метод ***Button1Click*** обычным стандартным способом, при нажатии остальных четырех кнопок будут выполняться действия, соответствующие поставленным задачам.

Прежде всего определим метод и функцию, с которыми будем экспериментировать.

Для определения метода просто зададим обработчик ***OnClick*** для верхней кнопки ***Button1*** и в сгенерированном шаблоне наберем код тела метода.

```
//---------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
  {
    ShowMessage(AnsiString("Метод:")+
    this->Name + "->" + ((TComponent*)Sender)->Name);
  }
//---------------------------------------------------------------------
```

Данный метод при вызове просто будет выдавать сообщение, что вызван метод класса, и сообщать имя кнопки, которая была нажата в момент вызова.

Теперь определим обычную функцию ***GlobalClick***.

```
//---------------------------------------------------------------------
void __fastcall GlobalClick(void* This, TObject *Sender)
  {
    ShowMessage(AnsiString("Функция:")+
    ((TComponent*)This)->Name + "->" +
    ((TComponent*)Sender)->Name);
  }
//---------------------------------------------------------------------
```

Функция будет выдавать признак вызова функции и аналогично методу сообщать имя кнопки, которая была нажата в момент вызова.

Отметим следующую особенность. У функции, в отличие от метода, добавился еще один параметр типа ***void****. Через этот параметр будет передаваться указатель на объект класса. В нашем случае через него будет передаваться указатель на ***Form1***.

Теперь попытаемся организовать вызов обычной функции в качестве обработчика события ***OnClick*** кнопки ***Button2***.

Замечание. События (events) реализуются в C++Builder посредством указателей на метод объекта (***__closure***), то есть события являются по сути указателями на метод объекта.

Эту операцию выполним в теле конструктора формы.

```
//---------------------------------------------------------------------
_fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
  {
    TMethod Method;
    Method.Data = this;
    Method.Code= GlobalClick;
    Button2->OnClick = *(TNotifyEvent*)&Method;
}
//---------------------------------------------------------------------
```

При выполнении этой операции задействована структура ***TMethod***. Структура довольно проста, содержит всего лишь два поля – ***Data*** и ***Code***, которые имеют тип ***void****:

```
struct TMethod
  {
    void *Code;
    void *Data;
  };
```

и по размеру соответствует указателю на метод объекта, состоящего, как уже упоминалось выше, также из двух указателей. Следовательно, посредством этой структуры с помощью преобразования можно инициализировать указатель на метод нужным значением. Для этого предварительно в поле ***Data*** заносится адрес объекта (в данном случае это форма), в поле ***Code*** – адрес функции. После присвоения содержимого ***Method*** событию ***OnClick*** данное событие будет связано с функцией, и при нажатии на кнопку ***Button2*** в процессе работы программы механизм выполнения событий вызовет функцию ***GlobalClick*** и передаст ей в качестве параметров адрес на ***Form1*** и адрес на объект, запустивший событие. В нашем случае этим объектом будет кнопка ***Button2***. Результат работы программы показан на рис. 3.17.

Рис. 3.17. *Результат работы программы при нажатии на вторую кнопку*

Вызов метода как обычной функции выполним в теле обработчика кнопки ***Button3***.

```
//---------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
  {
    //вызвать метод как обычную функцию
    TNotifyEvent Click = &Button1Click;
    TMethod Method = *(TMethod*)&Click;
    //через первый скрытый параметр передаем this
    typedef void (__fastcall *Func)(void*, TObject *);
    Func func;
    func = (Func)Method.Code;
    func(this, Sender);
  }
//---------------------------------------------------------------------
```

Данная операция выполняется в обратном порядке. Переменной ***Method*** присваивается содержимое: указатель на метод. Затем поле ***Code*** приводим к типу: указатель на функцию параметрами типа ***void****, ***TObject**** и выполняем вызов функции, передав ей в качестве параметров ***this*** и ***Sender***. Результат работы программы можно увидеть на рис. 3.18.

Для выполнения двух остальных задач потребуется задействовать дополнительные возможности расширенного RTTI, которые достались по наследству от ***Delphi*** и реализованы в виде методов класса ***TObject***. Естественно, эта дополнительная информация доступна только для VCL-классов, то есть только тех классов, которые являются производными от ***TObject***. Нам потребуются пока только два метода:

```
void * __fastcall MethodAddress(const ShortString &Name);
```

и

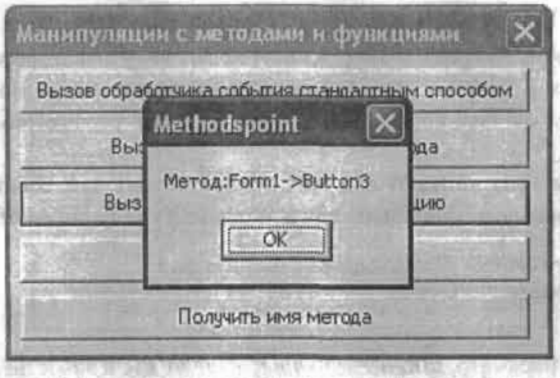


Рис. 3.18. *Результат работы программы при нажатии на третью кнопку*

С помощью первого из них можно получить адрес метода по его символьному имени, с помощью второго – получить имя, зная адрес. При этом надо учитывать, что использовать эти функции можно только для опубликованных методов, то есть методов, которые размещены в секции ***__published***.

Вызов метода по имени выполним в теле обработчика кнопки ***Button4***, а получение имени метода – в теле обработчика ***Button5***.

```
//---------------------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
  {
    ShortString ProcName = "Button1Click";
    TMethod Method = { MethodAddress(ProcName), this } ;
    if (Method.Code)
      {
        TNotifyEvent Click = *(TNotifyEvent*)&Method;
        Click(Sender);
      }
  }
//---------------------------------------------------------------------
void __fastcall TForm1::Button5Click(TObject *Sender)
  {
    TMethod Method = *(TMethod*)&(Button1->OnClick);
    ShowMessage( MethodName(Method.Code));
  }
//---------------------------------------------------------------------
```

Как видно, код достаточно прост и с учетом сделанных выше разъяснений не представляет трудностей для понимания.

Получение типа диска

Довольно часто перед программистом стоит задача получить информацию о каком-либо из дисков, стоящих в компьютере. Информация, которую можно получить, достаточно обширна, и об этом мы еще поговорим позднее, а сейчас рассмотрим вопрос получения типа диска.

Для получения типа диска Win32 API предоставляет нам специальную функцию, ***GetDriveType***. Данная функция объявлена в Winbase.h следующим образом.

```
UINT GetDriveType(
  LPCTSTR lpRootPathName // корневая директория
);
```

Единственный параметр, ***lpRootPathName***, ждет от нас в качестве значения корневую директорию указанного диска, то есть для диска "C" корневой директорией будет "C:\", для диска "D" корневой директорией будет "D:\" и т. д.

Возвращает функция значение типа ***int***, которое и определяет тип диска. Список возможных значений приведен в табл. 3.1.

Табл. 3.1. *Список возможных значений, возвращаемых функцией* ***GetDriveType***

Значение	Пояснение
DRIVE_UNKNOWN	Функция не может определить тип диска
DRIVE_NO_ROOT_DIR	Неверен путь к корневой директории. Такое происходит в том случае, когда по указанному пути не смонтирован диск
DRIVE_REMOVABLE	Сменный диск
DRIVE_FIXED	Жесткий диск
DRIVE_REMOTE	Диск является удаленным (сетевым) диском
DRIVE_CDROM	CD-ROM-диск
DRIVE_RAMDISK	RAM-диск

Теперь, после теоретической подготовки, можно приступать к практике. Загрузите тестовый проект и запустите его. Внешний вид запущенного тестового приложения представлен на рис. 3.19.

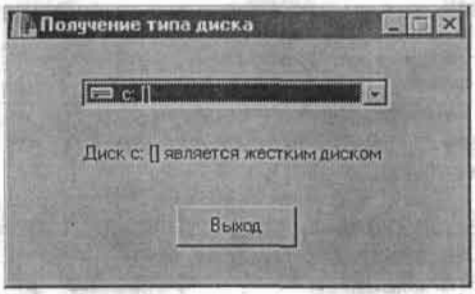


Рис. 3.19. *Внешний вид работающей программы*

Код данного проекта предельно прост и почти не требует комментариев. Все действия происходят в обработчике события ***OnChange*** выпадающего списка ***DriveComboBox***, который представляет собой компонент ***TDriveComboBox*** со страницы Win 3.1 палитры компонентов.

```
//---------------------------------------------------------------------
void __fastcall TDiskTypeForm::DriveComboBoxChange(TObject *Sender)
   {
      AnsiString DiskName = DriveComboBox->Text.SubString(1, 2) + "\\";
      int DriveType = GetDriveType(DiskName.c_str());

      switch(DriveType)
         {
            case DRIVE_UNKNOWN:
               DriveTypeLabel->Caption = "Неизвестный тип диска";
            break;
            case DRIVE_NO_ROOT_DIR:
              DriveTypeLabel->Caption = "По данному пути диск отсутствует";
            break;
            case DRIVE_REMOVABLE:
               //DiskType = dtRemovable;
             DriveTypeLabel->Caption = "Диск " + DriveComboBox->Text + "
               является сменным диском";
            break;
            case DRIVE_FIXED:
               //DiskType = dtFixed;
             DriveTypeLabel->Caption = "Диск " + DriveComboBox->Text + "
               является жестким диском";
            break;
            case DRIVE_REMOTE:
               //DiskType = dtRemoted;
              DriveTypeLabel->Caption = "Диск " + DriveComboBox->Text + "
                является удаленным диском";
```

```
                break;
                case DRIVE_CDROM:
                   //DiskType = dtCdrom;
                  DriveTypeLabel->Caption = "Диск " + DriveComboBox->Text + "
                    является CD-ROM-диском";
                break;
                case DRIVE_RAMDISK:
                   //DiskType = dtRamdisk;
                  DriveTypeLabel->Caption = "Диск " + DriveComboBox->Text + "
                    является RAM-диском";
                break;
                default:
                  Application->MessageBox("Ошибка при определении типа диска!",
                    "Внимание!", MB_OK | MB_ICONWARNING);
            }
    }
//---------------------------------------------------------------------------
```

Строкой

```
AnsiString DiskName = DriveComboBox->Text.SubString(1, 2) + "\\";
```

мы формируем корневую директорию ("C:\") для выбранного в списке диска.

В строке

```
int DriveType = GetDriveType(DiskName.c_str());
```

собственно, заносим тип диска в переменную ***DriveType***, а далее оператором ***switch*** производим анализ полученного значения. Вообще, на мой взгляд, надо сказать спасибо Microsoft, что они не объявили для типа диска какой-нибудь новый тип данных, а ограничились старым добрым ***int***. Просто и удобно.

В принципе еще имеет смысл заострить внимание на двух фрагментах кода. Первый:

```
default:
  Application->MessageBox("Ошибка при определении типа диска!",
  "Внимание!", MB_OK | MB_ICONWARNING);
break;
```

Казалось бы, зачем этот код, когда в него никогда не должно перейти управление? Вообще, правила хорошего тона всегда рекомендуют писать секцию ***default*** в ***switch*** – мало ли что... Второй фрагмент, на который следует обратить внимание:

```
case DRIVE_NO_ROOT_DIR:
  DriveTypeLabel->Caption = "По данному пути диск отсутствует";
break;
```

Этот код можно использовать не совсем по его прямому назначению. Функция ***GetDriveType*** вернет DRIVE_NO_ROOT_DIR не только в том случае, если неверен путь к корневой директории (вместо "C:\" указан "C:]", например), но и в том случае, если путь верен, а физически этого диска в системе нет. То есть мы можем использовать проверку на DRIVE_NO_ROOT_DIR для определения того, есть ли в системе указанный диск или нет.

Получение списка дисков в системе

Не является чем-то необычным необходимость получить список всех доступных в системе дисков. К сожалению, VCL в этом деле нам не очень хороший помощник. Она не предоставляет нам никаких методов и компонентов для этой цели, поэтому нам придется действовать самостоятельно, используя Windows API.

В Windows API за получение доступных в системе дисков ответственны две функции: ***GetLogicalDrives()*** и ***GetLogicalDriveStrings()***. То есть получить список всех дисков можно двумя способами, и я в данном материале последовательно рассмотрю оба этих метода.

Способ первый: GetLogicalDrives()

Функция ***GetLogicalDrives()*** возвращает значение типа ***DWORD*** (оно же ***int***), которое представляет собой битовую маску всех дисков в системе. То есть младший, нулевой бит возвращаемого значения отображает наличие или отсутствие диска "A", бит под номером 1 отображает наличие или отсутствие диска "B", бит под номером 2 – диска "C" и т. д., в соответствии с табл. 3.2.

Табл. 3.2. *Соответствие битов в маске логическим дискам*

Бит	...	3	2	1	0
Диск	...	D:\	C:\	B:\	A:\

Из таблицы видно, что узнать о наличии в системе диска "A" можно, применив операцию поразрядного "И" над маской, возвращенной функцией ***GetLogicalDrives()***, и числом, в нулевом бите которого содержится единица, а во всех остальных битах – нули, то есть с единицей. Узнать о наличии в системе диска "B" можно аналогичным образом, применив операцию поразрядного "И" над маской и двойкой. Диска "C" – над маской и четверкой и т. д., по тому же принципу. Число, с которым мы побитно складываем маску дисков, может быть получено путем поразрядного сдвига единицы влево. Учитывая тот факт, что дисков в системе не может быть больше 26, по числу букв латинского алфавита, мы можем написать следующий код для извлечения всех наличествующих в системе дисков.

```
DWORD Drives = GetLogicalDrives();

for(int i = 0; i < 26; i++)
   {
      if(Drives & (1 << i))
         {
            …
         }
   }
```

Эта заготовка кода реализует описанный выше алгоритм разбора битовой маски, возвращенной функцией ***GetLogicalDrives()***, но, собственно, никакого списка дисков она еще, разумеется, не заполняет. Для заполнения списка мы должны в оператор ***if*** добавить собственно код для преобразования результата разбора маски в буквенные обозначения и добавления полученных названий дисков в список. Результирующий код может выглядеть следующим образом.

```
TStringList *DiskList = new TStringList();
int Increment = 0;
AnsiString DiskName;

DWORD Drives = GetLogicalDrives();

for(int i = 0; i < 26; i++)
   {
      if(Drives & (1 << i))
         {
            DiskName = AnsiString(char('A' + Increment)) + ":\\";
            int OldErrorMode = SetErrorMode(SEM_FAILCRITICALERRORS);
            DiskList->Add(DiskName);
            SetErrorMode(OldErrorMode);
         }
      Increment++;
   }

delete DiskList;
```

Я думаю, код прост и пояснений не требует, кроме строк

```
int OldErrorMode = SetErrorMode(SEM_FAILCRITICALERRORS);
```

и

```
SetErrorMode(OldErrorMode);
```

Дело в том, что при наличии диска "A" в системе и отсутствии в нем сменного носителя без первой строки на экран будет выдано сообщение об ошибке. Поэтому мы указываем новый режим обработки ошибок системой, при котором вся обработка ошибки ложится на вызывающий процесс, а сообщение об ошибке на экран не выводится, сохраняя старый режим в переменной ***OldErrorMode***. После заполнения списка именами дисков мы восстанавливаем прежний режим обработки ошибок.

Теперь давайте перейдем к рассмотрению получения списка дисков с использованием функции ***GetLogicalDriveStrings()***.

Способ второй: GetLogicalDriveStrings()

Данный способ несколько проще, на мой взгляд. Функция ***GetLogicalDriveStrings()*** возвращает строку, уже состоящую из имен дисков, разделенных нулевым символом. Сама строка также завершается нулевым символом. Рассмотрим пример. Предположим, в вашей системе есть диски "C", "D" и "E". Тогда строка, возвращенная функцией ***GetLogicalDriveStrings()***, будет иметь следующую структуру (табл. 3.3).

Табл. 3.3. *Посимвольная структура строки при наличии в системе трех логических дисков*

C	:	\	\0	D	:	\	\0	E	:	\	\0	\0

Исходя из вышесказанного, получение списка дисков при использовании этой функции должно проходить по следующей схеме:

- выделение для строки буфера соответствующего размера;
- разбор содержимого буфера;
- занесение дисков в экземпляр класса ***TStringList***.

Для выполнения первого пункта вызовите ***GetLogicalDriveStrings()*** со следующими значениями параметров.

```
int BufferSize = GetLogicalDriveStrings(0, NULL);\
```

В этом случае функция вернет необходимый размер буфера с учетом всех нулевых символов, и он будет занесен в переменную ***BufferSize***.

А для выполнения второго и третьего пунктов плана напишите следующий код.

```
char *Buffer = new char[BufferSize];
TStringList *DiskList = new TStringList;

GetLogicalDriveStrings(BufferSize, Buffer);

// получаем количество дисков в системе
int DiskCount = (BufferSize - 1) / 4;
```

```
for(int i = 0; i < DiskCount; i++)
  {
      char DiskString[4];
      CopyMemory(DiskString, (Buffer + i * 4), 4);
      DiskList->Add(DiskString);
   }

delete [] Buffer;
delete DiskList;
```

На мой взгляд, необходимо прокомментировать следующий участок кода.

```
int DiskCount = (BufferSize - 1) / 4;

  for(int i = 0; i < DiskCount; i++)
    {
        char DiskString[4];
        CopyMemory(DiskString, (Buffer + i * 4), 4);
        DiskList->Add(DiskString);
     }
```

Первой строкой мы получаем количество дисков в системе (см. табл. 3.3, поясняющую структуру возвращаемой функцией ***GetLogicalDriveStrings()*** строки). В цикле ***for*** мы просто копируем из результирующего буфера по четыре байта и заносим имя диска в список. Мы повторяем эту операцию столько раз, сколько дисков в системе. В итоге, как и в первом случае с функцией ***GetLogicalDrives()***, у нас есть указатель на объект класса ***TStringList***, представляющий содержащий список всех логических дисков в системе.

Для демонстрации вышеизложенного создадим тестовый проект, внешний вид которого представлен на рис. 3.20.

При нажатии на кнопку "Старт1" (***GLDSButton***) вызывается код получения списка логических дисков через ***GetLogicalDriveStrings***, при нажатии на кнопку "Старт2" (***GLDSButton***) вызывается код получения списка логических дисков через ***GetLogicalDrives***. В обоих случаях в соответствующий список (***DiskListBox1*** или ***DiskListBox2***) заносятся полученные имена дисков (см. рис. 3.21).

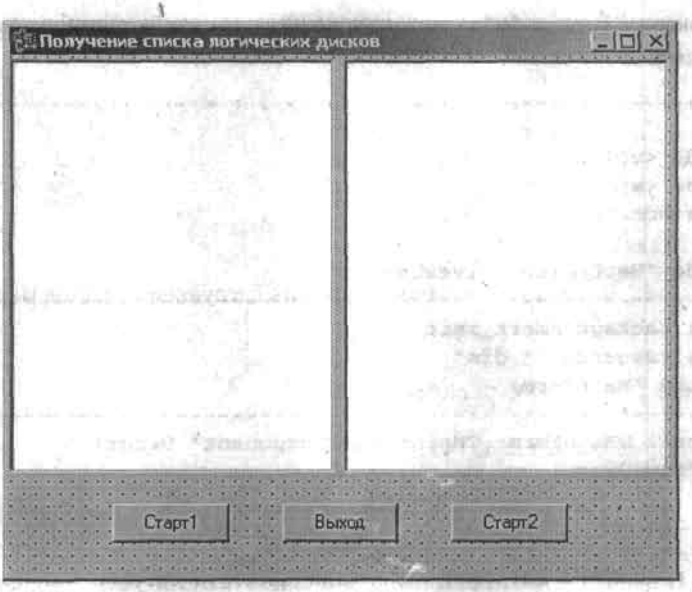


Рис. 3.20. *Внешний вид тестового приложения во время разработки*

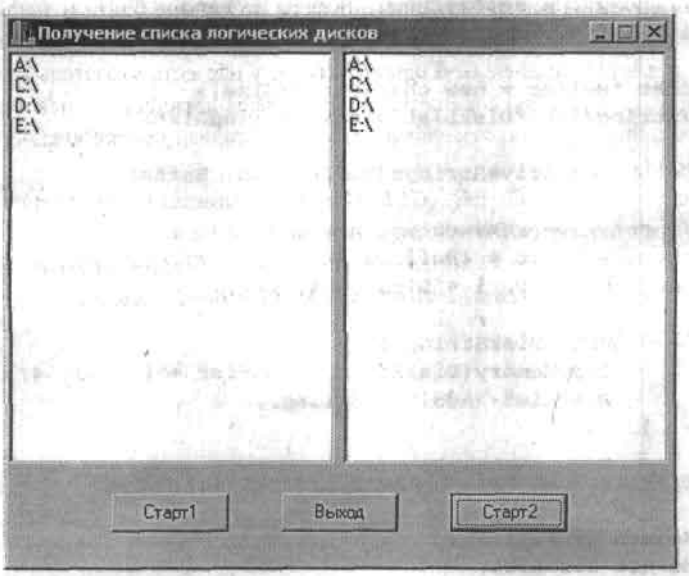


Рис. 3.21. *Результаты выполнения программы*

Ниже приведен код cpp-файла тестового проекта, так как заголовочный файл интереса не представляет.

```
//---------------------------------------------------------------------

#include <vcl.h>
#include <winbase.h>
#pragma hdrstop

#include "GetLogicalDrivesUnit.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------
void __fastcall TMainForm::GLDSButtonClick(TObject *Sender)
    {
        // получаем необходимый размер буфера
        int BufferSize = GetLogicalDriveStrings(0, NULL);

        char *Buffer = new char[BufferSize];
        TStringList *DiskList = new TStringList;

        GetLogicalDriveStrings(BufferSize, Buffer);

        // получаем количество дисков
        int DiskCount = (BufferSize - 1) / 4;
        for(int i = 0; i < DiskCount; i++)
            {
                char DiskString[4];
                CopyMemory(DiskString, (Buffer + i * 4), 4);
                DiskList->Add(DiskString);
            }

        DiskListBox1->Items->Assign(DiskList);

        delete [] Buffer;
        delete DiskList;
    }
```

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::GLDButtonClick(TObject *Sender)
{
      TStringList *DiskList = new TStringList();

      DWORD Drives = GetLogicalDrives();
      int Increment = 0;
      AnsiString DiskName;

      for(int i = 0; i < 26; i++)
        {
          if(Drives & (1 << i))
            {
              DiskName = AnsiString(char('A' + Increment)) + ":\\";
             int OldErrorMode = SetErrorMode(SEM_FAILCRITICALERRORS);

              DiskList->Add(DiskName);

              SetErrorMode(OldErrorMode);
            }
          // конец оператора if(Drives & (1 << i))
          Increment++;
        }
      // конец оператора for(int i = 0; i < 26; i++)

   DiskListBox2->Items->Assign(DiskList);

   delete DiskList;
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::ExitButtonClick(TObject *Sender)
  {
    Application->Terminate();
  }
//---------------------------------------------------------------------------
```

Применение шаблонов при динамическом связывании DLL с основным приложением

Процесс динамического связывания модуля DLL с основным приложением состоит из следующих операций:

- загрузка модуля DLL и получение указателя на этот модуль;
- получение указателя на импортируемую функцию;
- вызов функции;
- выгрузка модуля DLL.

Обычно код, выполняющий данные операции, может выглядеть следующим образом.

```
const AnsiString cDllName1 =  "modalform.dll ";
const AnsiString cProcName1 = "_ShowModal "

void __fastcall TForm1::ToolButton3Click(TObject *Sender)
{
    static int Number = 0;
    typedef TModalResult (*TShowModal)(const AnsiString&);
    TShowModal ShowModal;
    HINSTANCE DllInstance = NULL;
    try
    {
#pragma warn -pia
        if (!(DllInstance = LoadLibrary(cDllName1.c_str())))
        {
            throw(Exception("Ошибка загрузки  библиотеки:" +
cDllName1));
        }
        if(!(ShowModal =
              (TShowModal)GetProcAddress(DllIn-
stance,cProcName1.c_str())))
        {
            throw(Exception("Ошибка получения адреса функции:"+
                    cProcName1));
        }
       if (mrOk == ShowModal(AnsiString("Form №") + AnsiString(Number++)))
             ShowMessage("Нажата кнопка Ok");
        else ShowMessage("Нажата кнопка Cancel");
    }
#pragma warn .pia
    catch(Exception& excp)
    {
```

```
        Application->ShowException(&excp);
      }
      if (DllInstance)
      {
        FreeLibrary(DllInstance);
      }
}
```

Поскольку импортирование функции из модулей DLL является довольно распространенным приемом при проектировании модульных приложений, то вполне естественным становится желание упростить механизм вызова функций, сделать код удобочитаемым и лаконичным. Решения этих задач можно добиться, задействовав механизм шаблонов. В файле UTILCLS.H, поставляемом в составе C++Builder, имеется реализация классов, инкапсулирующих работу с модулем DLL и импортируемыми функциями. К сожалению, они даны в сильно урезанном варианте и несколько бессистемно, что объясняется, по всей видимости, тем, что они предназначались разработчиками C++Builder для своих внутренних нужд. Поэтому предлагается использовать их не напрямую, а заново переписать классы и собрать их вместе во вновь созданном для этого файле Dll.h, немного адаптировав и расширив набор для более удобного использования. Прежде всего определим классы для исключений (exception classes), которые будут использоваться в качестве параметров при генерации исключительных ситуаций.

```
__declspec(selectany) AnsiString  ErrDllMsg = "Ошибка загрузки
библиотеки:%s";
__declspec(selectany) AnsiString  ErrProcMsg = "Ошибка получения
адреса функции:%s";
class EDllError: public Exception
{
private:
    AnsiString m_Name;
public:
    __fastcall EDllError(const AnsiString a_Fmt ,const AnsiString a_Name)
              : Exception(Format(a_Fmt,ARRAYOFCONST((a_Name)))),
                m_Name(a_Name){ };
    __property AnsiString Name  = { read=m_Name, write=m_Name };
};
class EDllProcError: public EDllError
{
public:
    __fastcall EDllProcError(const AnsiString a_Fmt ,const AnsiString a_Name)
             : EDllError(a_Fmt ,a_Name){};
};
```

В переменных ***ErrDlgMsg*** и ***ErrProcMsg*** помещены строки форматирования, которые используются для формирования сообщения об ошибке. Они специально не объявлены как константы, для того чтобы при необходимости можно было легко подменить строку форматирования. Класс ***EDllError*** будет применяться при генерации исключительной ситуации, если процесс по загрузке модуля DLL завершится неудачно, а ***EDllProcError*** – если невозможно будет получить указатель на функцию.

Теперь определим классы-обертки для двух понятий: модуль DLL и импортируемая функция.

```
class TDll {
    HINSTANCE m_DLLInstance;
public:
    TDll(const char* a_Name) {
#pragma option push -w-pia
     if (!(m_DLLInstance = LoadLibrary(a_Name)))
#pragma option pop
       throw(EDllError(ErrDllMsg, a_Name));
    }
    ~TDll(){
     if (m_DLLInstance) FreeLibrary(m_DLLInstance);
    }
    operator HINSTANCE() const { return m_DLLInstance; }
};
class TDllProc {
public:
  TDllProc(const TDll& a_Dll, const char*  a_Name)
  {
#pragma option push -w-pia
   if(!(m_Proc = GetProcAddress(HINSTANCE(a_Dll),a_Name)))
#pragma option pop
    throw(EDllProcError(ErrProcMsg, a_Name));
  }
public:
 FARPROC m_Proc;
};
```

Основное отличие от тех классов, которые предлагает Borland, заключается в генерации исключительной ситуации в конструкторе класса при неудачном выполнении операции (загрузка библиотеки или получение адреса функции). В этом случае экземпляр класса просто не будет создан, а естественный ход выполнения будет просто прерван. Дело вкуса, но такой подход, на наш взгляд, выглядит более предпочтительным, чем вариант от Borland с последующей проверкой на правильность выполнения операции. Теперь на основе класса TDllProc создаем шаблонные классы для функций с различным числом параметров. Прежде всего договоримся о соглашениях в наименовании классов.

Имя класса для функции, которая не возвращает результат, будет иметь вид TDllProcVX.

Имя класса для функции, которая возвращает результат, будет иметь вид TDllProcX.

В каждом конкретном случае вместо X в имени класса подставляется число параметров. Тогда класс для функции без параметров и не возвращающей результат будет выглядеть следующим образом.

```
class TDllProcV0 : public TDllProc {
public:
  TDllProcV0(const TDll& a_Dll, const char*  a_Name)
  :TDllProc(a_Dll,a_Name) {}
  void operator ()()
    {
    typedef void (* TProc)();
        ((TProc)m_Proc)();
  }
};
```

Так как функция, как уже упоминалось, не имеет параметров и ничего не возвращает, то шаблоны здесь не потребовались. Но уже при определении класса для следующей функции, которая также не имеет параметров, но возвращает результат, без шаблонов не обойтись.

```
template <class R>
class TDllProc0 : public TDllProc {
public:
  TDllProc0(const TDll& a_Dll, const char*  a_Name)
  : TDllProc(a_Dll,a_Name) {}
  R operator ()()
  {
    typedef R (  * TProc)();
    return ((TProc)m_Proc)();
  }
};
```

Аналогично надо создать реализации для функций с одним, двумя аргументами и далее до необходимого максимального числа. Например, в библиотеке OWL, в которой тоже использовался подобный прием, ограничились 13 параметрами. В качестве примера покажем реализацию класса для функции с двумя параметрами.

```
template <class P1, class P2>
class TDllProcV2 : public TDllProc {
public:
  TDllProcV0(const TDll& a_Dll, const char*  a_Name)
  :TDllProc(a_Dll,a_Name) {}
  void operator ()(P1 p1, P2 p2)
```

```
    {
    typedef void (__cdecl* TProc)(P1, P2 );
        ((TProc)m_Proc)(p1, p2);
  }
};

template <class R, class P1, class P2>
class TDllProc0 : public TDllProc {
public:
  TDllProc0(const TDll& a_Dll, const char*  a_Name)
  : TDllProc(a_Dll,a_Name) {}
  R operator ()(P1 p1, P2 p1)
  {
    typedef R ( __cdecl  *TProc)(P1, P2);
    return ((TProc)m_Proc)(p1, p2);
  }
};
```

При вызове импортируемых функций требуется учитывать еще способ передачи параметров. В C++Builder установлено, что если с помощью спецификатора явно не определен другой способ, то по умолчанию используется передача параметров в стиле C (***__cdecl***). Для того чтобы не возникли неясности в связи с возможной перенастройкой опций IDE, данный спецификатор лучше указать явно. Кроме ***__cdecl*** очень часто используется стандартный способ передачи параметров (***__stdcall***), поэтому напишем реализацию и для ***__stdcal***.

```
template <class P1, class P2>
class TDllStdProcV2 : public TDllProc {
public:
  TDllStdProcV2(const TDll& a_Dll, const char*  a_Name)
  : TDllProc(a_Dll,a_Name) {}
  void operator ()(P1 p1,P2 p2)
  {
    typedef void ( __stdcall* TProc)(P1,P2);
    ((TProc)m_Proc)(p1,p2);
  }
};

template <class R,class P1,class P2>

class TDllStdProc2 : public TDllProc {
public:
  TDllStdProc2(const TDll& a_Dll, const char*  a_Name)
  : TDllProc(a_Dll,a_Name) {}
```

```
    R operator ()(P1 p1,P2 p2)
    {
      typedef R ( __stdcall* TProc)(P1,P2 );
      return ((TProc)m_Proc)(p1,p2);
    }
};
```

Во избежание конфликтов имен рекомендуется все эти классы обрамить в namespace.

```
namespace  Dll
{
…
}
using namespace Dll;
```

Теперь посмотрим, как будет выглядеть код по вызову функции из DLL с использованием шаблонов.

```
void __fastcall TForm1::ToolButton2Click(TObject *Sender)
{
    static int Number = 0;
    try
    {
        TDll dll(cDllName1.c_str());
        TDllProc1<TModalResult,const AnsiString&>
                                    ShowModal(dll,cProcName1.c_str());
      if (mrOk== ShowModal(AnsiString("Form №") + AnsiString(Number++)))
            ShowMessage("Нажата кнопка Ok");
        else ShowMessage("Нажата кнопка Cancel");
    }
    catch(Exception& except)
    {
        ShowMessage(except.Message);
    }
}
```

Вся процедура вызова функции из динамической библиотеки уложилась в три строки: два объявления и вызов. Все достаточно прозрачно и лаконично.

Подмена оконной процедуры компонента и обработка сообщений

Бывают ситуации, когда вам необходимо обработать сообщение, которое каким-либо компонентом VCL не обрабатывается по умолчанию. Ну нет для данного сообщения готового обработчика. Что делать в таких случаях? Не паниковать и вспомнить, что VCL предоставляет, как минимум, два метода решения данной проблемы. Первый, восходящий своими корнями в незапамятные времена – это использование карты сообщений. В VCL это означает использование макросов BEGIN_MESSAGE_MAP, MESSAGE_HANDLER и END_MESSAGE_MAP. Второй способ – замена оконной процедуры компонента через свойство ***WindowProc***.

При всем том что эти два способа направлены на решение одной и той же задачи, их использование не всегда эквивалентно. При написании собственного компонента – да, не имеет значения, какой способ обработки сообщений вы используете. Но в случае когда вы по *какой-либо причине не хотите создавать свой компонент, данные способы не взаимозаменяемы.* Без создания компонента карта сообщений позволяет обработать сообщения только для того класса, в котором она объявлена. То есть, если вы объявили карту сообщений в классе формы, вы сможете обработать только те сообщения, которые отправляются форме. А если ваша цель – отлов и отклик на необрабатываемые по умолчанию сообщения в каком-либо отдельно взятом компоненте, расположенном на форме, -- необходимо пользоваться подменой оконной процедуры компонента, а не картой сообщений. Ибо поместить карту сообщений внутрь класса, какого-нибудь ***TMemo***, например, без написания компонента-наследника ***TMemo*** – невозможно. Поэтому мы сейчас поговорим об обработке сообщений без написания компонента – о подмене оконной процедуры компонента.

Все визуальные (соответственно имеющие окно в понятии ОС Windows) компоненты наследуются в VCL от класса ***TControl***. В данном классе есть метод ***WndProc*** и свойство ***WindowProc***. Они-то нам и понадобятся.

Метод ***WndProc***

```
virtual void __fastcall WndProc(Messages::TMessage &Message);
```

по сути дела и есть оконная процедура компонента, занимающаяся обработкой сообщений.

Свойство ***WindowProc***

```
__property TWndMethod WindowProc = {read=FWindowProc,
write=FWindowProc};
```

определяет, какая функция используется компонентом для обработки сообщений. Изначально значение ***WindowProc*** установлено в ***WndProc***, то есть по умолчанию обработкой сообщений в компоненте занимается метод ***WndProc***. Написав свою функцию обработки сообщений и присвоив ее адрес свойству ***WindowProc***, мы сможем обработать те сообщения, которые изначально компонентом не обрабатываются.

Итак, приступаем. В качестве задачи поставим себе следующее: пусть нам будет необходимо определить попадание курсора мыши в пределы элемента управления и выход курсора мыши за пределы элемента управления. В качестве такого элемента управления воспользуемся компонентом ***TLabel***, размещенным на форме.

Загрузите тестовый проект. Внешний вид формы тестового проекта представлен на рис. 3.22.

Рис. 3.22. *Форма тестового проекта*

После нажатия кнопки ***ProcessButton*** (с заголовком "Обрабатывать") метка ***TestLabel*** ("Тестовая метка") начинает обрабатывать вхождение курсора мыши в свои пределы и покидание пределов курсором мыши, о чем свидетельствует изменение цвета метки на красный и обратно и изменение заголовка формы (рис. 3.23 и 3.24).

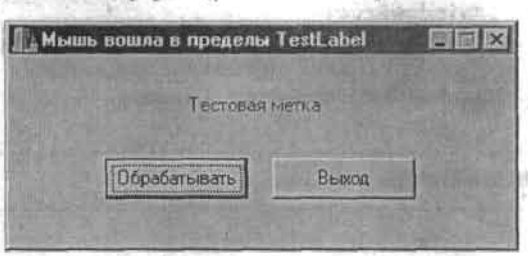


Рис. 3.23. *Курсор мыши в пределах метки*

Рис. 3.24. *Курсор мыши в пределах метки*

Код тестового проекта приведен ниже.

Заголовочный файл модуля формы.

```
//---------------------------------------------------------------------
#ifndef WindowProcUnitH
#define WindowProcUnitH
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------
class TMainForm : public TForm
{
__published:            // IDE-managed Components
   TLabel *TestLabel;
   TButton *ProcessButton;
   TButton *ExitButton;
   void __fastcall ProcessButtonClick(TObject *Sender);
   void __fastcall ExitButtonClick(TObject *Sender);
   void __fastcall FormCreate(TObject *Sender);
private:        // User declarations
   TColor OldLabelFontColor;
   TWndMethod OldWindowProc;
   void __fastcall NewWindowProc(Messages::TMessage &Message);
public:                 // User declarations
   __fastcall TMainForm(TComponent* Owner);
};
//---------------------------------------------------------------------
extern PACKAGE TMainForm *MainForm;
//---------------------------------------------------------------------
#endif
```

Все сделанные нами объявления в заголовочном файле модуля находятся в секции private класса формы.

Строкой

```
TWndMethod OldWindowProc;
```

мы объявляем переменную ***OldWindowProc***. Переменная ***OldWindowProc*** имеет тот же тип, что и свойство ***WindowProc*** метки, – ***TWndMethod***. Данная переменная необходима нам для сохранения старой оконной функции компонента ***TLabel***.

Строкой

```
void __fastcall NewWindowProc(Messages::TMessage &Message);
```

мы объявляем новую оконную функцию для нашей тестовой метки. Данная функция будет обрабатывать все сообщения, направляемые метке.

Строка

```
TColor OldLabelFontColor;
```

необходима для временного хранения исходного цвета шрифта метки при его изменении во время вхождения курсора мыши в пределы компонента.

CPP-файл модуля формы.

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "WindowProcUnit.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
    : TForm(Owner)
    {
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::NewWindowProc(Messages::TMessage &Message)
    {

        if(Message.Msg == CM_MOUSEENTER)
            {
                Caption = "Мышь вошла в пределы TestLabel";
                TestLabel->Font->Color = clRed;
            }
        else if(Message.Msg == CM_MOUSELEAVE)
            {
                Caption = "Мышь покинула пределы TestLabel";
                TestLabel->Font->Color = OldLabelFontColor;
            }
        OldWindowProc(Message);
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::ProcessButtonClick(TObject *Sender)
```

```
    {
        OldWindowProc = TestLabel->WindowProc;
        TestLabel->WindowProc = NewWindowProc;
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::ExitButtonClick(TObject *Sender)
    {
        Application->Terminate();
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::FormCreate(TObject *Sender)
    {
        OldLabelFontColor = TestLabel->Font->Color;
    }
//---------------------------------------------------------------------
```

Теперь переходим к основному файлу модуля. Рассмотрим сначала обработчик события ***OnClick*** кнопки ***ProcessButton***.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::ProcessButtonClick(TObject *Sender)
    {
        OldWindowProc = TestLabel->WindowProc;
        TestLabel->WindowProc = NewWindowProc;
    }
//---------------------------------------------------------------------
```

В данном обработчике мы сохраняем старый адрес оконной процедуры ***WndProc*** компонента в члене данных формы ***OldWindowProc*** (для этого в секцию ***private*** мы и добавляли строку с объявлением переменной ***OldWindowProc***).

Второй строкой:

```
TestLabel->WindowProc = NewWindowProc;
```

мы устанавливаем значение свойства ***WindowProc*** в адрес новой оконной функции, в которой теперь будут обрабатываться все сообщения для нашей тестовой метки. Теперь оконной функцией для компонента будет функция ***NewWindowProc(Messages::TMessage &Message)***.

Рассмотрим определение новой оконной функции для метки.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::NewWindowProc(Messages::TMessage &Message)
    {
        if(Message.Msg == CM_MOUSEENTER)
            {
                Caption = "Мышь вошла в пределы TestLabel";
                TestLabel->Font->Color = clRed;
```

```
    }
  else if(Message.Msg == CM_MOUSELEAVE)
    {
      Caption = "Мышь покинула пределы TestLabel";
      TestLabel->Font->Color = OldLabelFontColor;
    }
  OldWindowProc(Message);
}
//---------------------------------------------------------------------------
```

Во-первых, необходимо знать, что при попадании курсора мыши в пределы элемента управления VCL отправляет элементу управления сообщение CM_MOUSEENTER, а при покидании курсором мыши пределов элемента управления – сообщение CM_MOUSELEAVE. Далее, структура ***TMessage*** в VCL имеет член данных ***Msg***, который позволяет идентифицировать поступившее на обработку в оконную функцию сообщение. При получении функцией сообщений CM_MOUSEENTER или CM_MOUSELEAVE изменяются цвет шрифта метки и заголовок формы. При получении всех остальных сообщений вызывается исходный, неизменный метод ***WndProc***. Обратите на это внимание! В переопределенных оконных функциях после обработки интересующих вас сообщений обязателен вызов исходного метода ***WndProc***. Адрес ***WndProc*** метки у нас сохранен в ***OldWindowProc***, поэтому ***WndProc*** и вызывается как

```
OldWindowProc(Message)
```

Осталось сказать немногое. Во-первых, повторю еще раз, что в переопределенном методе ***WndProc*** обязательно должен присутствовать вызов сохраненного исходного метода ***WndProc***. Во-вторых, у рассмотренного нами метода есть существенный недостаток. Представим, что на форме у вас не одна такая метка, а пять. И даже если код отклика на события должен быть одинаковым, использовать одну и ту же оконную процедуру для всех меток не получится, как это ни грустно. Поскольку в каждой оконной функции используется дескриптор окна только одной конкретной метки. Иными словами, работает правило: у каждого компонента – своя оконная функция. Поэтому данный способ хорош, когда отклик на события не надо "тиражировать" для хотя бы нескольких компонентов. Если же вам постоянно в каких-то компонентах необходимо иметь отклик на события, обработка которых в данных компонентах отсутствует, вам необходимо писать свои собственные компоненты, внутри которых делать обработку интересующих вас событий.

Вы можете поинтересоваться, почему данный материал вообще рассмотрен в данной книге, когда у компонента ***TLabel*** в C++Builder 6 обработчики событий ***OnMouseEnter*** и ***OnMouseLeave***? Дело в том, что ***TLabel*** здесь используется просто в качестве одного из самых простых примеров, чтобы понять собственно принцип. А во-вторых, все остальные сообщения обрабатываются абсолютно аналогичным способом: вместо CM_MOUSEENTER и CM_MOUSELEAVE вы используете символьные константы других сообщений – и эти сообщения обрабатываются безо всяких проблем.

Окна нестандартной формы

Есть ряд вопросов, посвященных программированию вообще и в среде C++Builder в частности, которые с завидной регулярностью выходят на свет в различных местах общения программистов. К ним относятся вопросы типа помещения иконки в трей (есть даже особое название трея – "туда, где часы"), ограничение количества запущенных экземпляров приложения и т. д. Среди этих вопросов, успевших порядком поднадоесть, выделяется, вероятно из-за своей бесполезной привлекательности, вопрос об окне нестандартного вида: круглого, произвольной формы, с дырками... Выделяется бесполезностью, потому что использование таких окон в серьезных программах крайне ограничено (вероятно, все подобные серьезные программы, где такие окна можно применить, уже написаны до вас), а писать несерьезные... А есть ли смысл? Тем не менее считаю необходимым объяснить процесс построения окон произвольной формы, чтобы, по крайней мере, если вдруг вам понадобится программная реализация для какой-нибудь серьезной программы, где крайне необходимо наличие такого вот окна, не изобретать велосипед самому, а воспользоваться готовым кодом, ибо он несложен, вплоть до примитива.

Итак, окна произвольной формы. Суть их создания состоит в размещении на форме битового изображения, один из цветов которого необходимо сделать прозрачным, чтобы сквозь области, закрашенные данным цветом, просвечивали нижележащие окна. Таким образом, мы сможем смоделировать окно практически любых, самых изощренных форм. Процесс обеспечения прозрачности отдельных частей окна несколько различается в зависимости от того, для каких операционных систем предназначается программа с нестандартными окнами. Если конкретнее, то для операционных систем Windows 2000 и Windows XP можно воспользоваться реализуемой ими на уровне систем прозрачностью окон, а для всех остальных операционных систем существует способ с использованием функций Windows API ***CreateRgn***, ***CombineRgn*** и ***DeleteObject***. Второй способ является более универсальным, так как, помимо всего прочего, он также применим и для Windows 2000/XP, но все-таки я советую вам писать две версии кода и во время работы программы вызывать ту из них, которая предназначена для той операционной системы, под управлением которой работает ваша программа. Реализуемая на уровне ОС прозрачность окна работает заметно быстрее функций для работы с областями (или, иначе, регионами), да и будущее именно за развитием линейки XP, поэтому не вижу смысла в качестве основного использовать код, предназначенный для работы на неподдерживаемых более ОС. Более того, в пользу метода с прозрачностью окон есть один аргумент, который можно назвать решающим: данный метод заметно проще. С него и начнем.

Реализация окон нестандартной формы в Windows 2000/XP

Нет, чем все-таки он хорош – тем, что вам почти ничего не надо делать. Все, что вам нужно:

- Подготовить картинку. В качестве картинки я предлагаю следующее изображение.

Рис. 3.25. *Исходное изображение для окна*

Картинка, конечно, страшная. Очень страшная. Но не надо переживать – в итоге у нас останется только окно стандартного серого цвета, той формы, как серая область на этой картинке, а весь этот жуткий сиреневый цвет мы объявим как прозрачный.

- Разместить данную картинку на форме нового, созданного вами в C++Builder проекта.
- В Инспекторе объектов установить соответствующие свойства, определяющие прозрачный цвет и еще ряд иных атрибутов окна.
- Навести внешний лоск путем добавления нескольких строчек кода в проект.
- Запустить приложение и насладиться полученным результатом.

Это вкратце. Теперь более подробно. Откройте в C++Builder наш тестовый проект, предназначенный для Windows 2000/XP. У вас на экране должно быть нечто вроде этого.

Рис. 3.26. *Внешний вид тестового проекта во время разработки*

На форме (***fMainForm***) расположен компонент ***TImage*** (***iMainImage***) с загруженным изображением, кнопка (***bClose***), по нажатии на которую программа завершает свою работу.

Теперь установим в Инспекторе объектов соответствующие значения свойств.

Сначала установите значение свойства ***BorderStyle*** формы в ***bsNone***. Если этого не сделать, то впоследствии при запуске приложения у вас будет следующая картина.

Рис. 3.27. *Неустановленное свойство BorderStyle у формы приведет к таким последствиям*

Не думаю, что это входит в ваши планы, так что не забудьте про свойство ***BorderStyle***.

Далее. Установите значение свойства ***AlphaBlend*** формы в ***true***. Именно свойство ***AlphaBlend*** ответственно за включение прозрачности у окна. Установите значение свойства ***TransparentColor*** в ***true***. Это укажет, что один из цветов, который есть на форме, будет прозрачным, и останется только указать, какой именно цвет будет обозначать прозрачность. Для этого установите значение свойства ***TransparentColorValue*** в "этот ужасный сиреневый цвет" – ***clFuchsia***. Все, со свойствами закончено (я же предупреждал, что делать почти ничего не надо!).

Теперь о том самом загадочном лоске. Поскольку у нашего окна не будет заголовка (значение свойства ***BorderStyle*** формы установлено в ***bsNone***, вы помните), то необходимо предоставить пользователю механизм закрытия окна, во-первых, и перемещения данного окна – во-вторых. И если с первым все понятно – надо всего лишь на обработчик события ***OnClick*** кнопки повесить вызов метода ***Close()***:

```
//---------------------------------------------------------------------
void __fastcall TfMainForm::bCloseClick(TObject *Sender)
  {
    Close();
  }
//---------------
```

то для перемещения формы вам придется воспользоваться следующим кодом:

```
//---------------------------------------------------------------------
void __fastcall TfMainForm::iMainImageMouseDown(TObject *Sender,
     TMouseButton Button, TShiftState Shift, int X, int Y)
{
  long SC_DRAGMOVE = 0xF012;
  if(Button == mbLeft)
  {
    ReleaseCapture();
    SendMessage(Handle, WM_SYSCOMMAND, SC_DRAGMOVE, 0);
  }
}
//---------------------------------------------------------------------
```

Как видно, в обработчик события ***OnMouseDown*** компонента ***TImage*** надо всего лишь добавить пару строк. В качестве бонуса пользователь получит возможность перемещать форму за любую ее точку, а не только за заголовок, как это принято по умолчанию. Более того, чтобы организовать заголовок, его придется в такой форме специально эмулировать. Как? Оставляю вам на самостоятельную проработку – не только же код вам копировать из данного материала.

Запустите тестовый проект. На экране будет окно вашей мечты (рис. 3.28).

Рис. 3.28. *То самое окно нестандартной формы*

Теперь мы готовы рассмотреть, как создать такое же окно, но под старыми операционными системами, которые не поддерживают прозрачность окон: это вся линейка Win9x/ME плюс Windows NT 4.0.

Реализация окон нестандартной формы в Windows NT 4.0, Windows 95/98/ME

Исходный материал для подобных окон в данных системах тот же самый, только тестовый проект должен быть немного другим и свойства, отвечающие за прозрачность формы, устанавливать не надо. Внешний вид тестового проекта абсолютно такой же, как и на рис. 3.26. Основное отличие тестового проекта для данных операционных систем – это наличие в нем дополнительной, но очень важной функции – ***BitmapToRegion***. Именно данная функция выполняет всю тяжелую и грязную работу по "превращению" битового изображения в окно соответствующей формы.

Функция ***BitmapToRegion*** основана на использовании функций Win32 API для работы с регионами, а именно функций ***CreateRectRgn*** и ***CombineRgn***. Подробное описание этих функций вы можете найти в MSDN, а мы непосредственно перейдем к функции ***BitmapToRegion***.

Код ее приведен ниже.

```
//---------------------------------------------------------------------
HRGN TfMainForm::BitmapToRegion(TPicture *APicture,
TColor ATransparentColor)
  {
    HRGN Result = 0;

    for(int y = 0; y < APicture->Height; y++)
      {
        int x = 0, start = 0, end = 0;

        // Пропускаем прозрачные точки
        while((x < APicture->Width) &&
        (APicture->Bitmap->Canvas->Pixels[x][y] == ATransparentColor))
          x++;

        start = x; // начало региона по горизонтали

        // Пропускаем непрозрачные точки
        while((x < APicture->Width) &&
        (APicture->Bitmap->Canvas->Pixels[x][y] != ATransparentColor))
          x++;

        end = x; // конец региона по горизонтали

        if(!Result)
          Result = CreateRectRgn(start, y, end, y + 1);
        else
          {
```

```
            HRGN TempRgn = CreateRectRgn(start, y, end, y + 1);
            CombineRgn(Result, Result, TempRgn, RGN_OR);
            DeleteObject(TempRgn);
          }
        // конец оператора if(!Result)
      }
    // конец оператора for(int y = 0; y < APicture->Height; y++)

    return Result;
  }
//---------------------------------------------------------------------------
```

Код простой, поэтому даны лишь краткие пояснения.

Цикл

```
for(int y = 0; y < APicture->Height; y++)
```

обеспечивает итерацию по всем линиям битового изображения от нулевой (самой верхней) до последней (самой нижней).

Внутри цикла ***for*** операторы

```
while((x < APicture->Width) &&
(APicture->Bitmap->Canvas->Pixels[x][y] != ATransparentColor))
  x++;
```

ищут первую непрозрачную точку в битовом изображении, а оператор

```
start = x; // начало региона по горизонтали
```

запоминает ее x-координату.

Аналогичным образом функционируют операторы

```
// Пропускаем непрозрачные точки
while((x < APicture->Width) &&
(APicture->Bitmap->Canvas->Pixels[x][y] != ATransparentColor))
  x++;
```

и

```
end = x; // конец региона по горизонтали
```

с той лишь разницей, что ищется и запоминается координата не первой, а последней непрозрачной точки битового изображения в данной линии пикселов, определяемой значением переменной *y*.

Затем создается область на основе полученных координат и на строках пикселов, отличных от первой, складывается с ранее созданной областью.

```
if(!Result)
  Result = CreateRectRgn(start, y, end, y + 1);
else
    {
      HRGN TempRgn = CreateRectRgn(start, y, end, y + 1);
      CombineRgn(Result, Result, TempRgn, RGN_OR);
      DeleteObject(TempRgn);
    }
```

На что следует обратить внимание, так это на оператор

```
DeleteObject(TempRgn);
```

Созданная область должна быть освобождена, когда в ней отсутствует необходимость. В противном случае будет происходить утечка ресурсов.

Остальной код в данном проекте аналогичен коду из предыдущего проекта, а при запуске нашего тестового приложения вы получите ту же форму, что и при запуске тестового приложения, использующего особенности операционных систем Windows 2000 и выше.

Передача параметров командной строки в приложение

Передача параметров командной строки в приложение. Казалось бы, что может быть интересного в этом для программиста на C++Builder в частности и для Windows-программиста в общем? Ведь эпоха ДОС уже давно ушла и приложениям с графическим интерфейсом пользователя не принято передавать входные параметры через командную строку? Однако тут кроется заблуждение. И дело не только в том, что через командную строку приложению можно указать ряд каких-то опций, которые бывают полезны опытным пользователям данного приложения (например, если вы не знаете, использование параметра –ns, переданного в командной строке C++Builder, заставит последний загружаться без отображения заставки). Дело в том, что любая ассоциация файлов с приложением, столь необходимая в современных программах для Windows, использует ту самую, вроде бы несовременную и устаревшую передачу параметров через командную строку. Поэтому списывать ее еще рано, и ниже мы рассмотрим реализацию данного способа передачи параметров.

Разумеется, приложение должно реагировать на передачу параметров до появления на экране главной его формы. Поэтому придется опять заглянуть в святая святых каждого проекта на C++Builder – в код проекта.

Вот стандартный код проекта, который создает среда разработки.

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
USERES("Project1.res");
```

```
USEFORM("Unit1.cpp", Form1);
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  try
  {
     Application->Initialize();
     Application->CreateForm(__classid(TForm1), &Form1);
     Application->Run();
  }
  catch (Exception &exception)
  {
     Application->ShowException(&exception);
  }
  return 0;
}
//---------------------------------------------------------------------------
```

Мы должны вставить наш код отклика на переданный параметр командной строки до строки

```
Application->Run();
```

Но код должен быть также вставлен после процесса создания всех форм, в противном случае вы не сможете обратиться к компонентам на формах в этом коде, если вам понадобится. Попытка обращения к ним вызовет AccessViolation – формы еще не будут созданы.

Теперь давайте посмотрим, что Borland предоставляет для облегчения нашей нелегкой жизни программистов. Как обычно, выручает модуль Sysutls.hpp, в котором объявлены две функции, ***ParamStr*** и ***ParamCount***.

```
extern PACKAGE AnsiString __fastcall ParamStr(int Index);
extern PACKAGE int __fastcall ParamCount(void);
```

Функция ***ParamCount*** возвращает число переданных в строке параметров, а функция ***ParamStr*** позволяет получить к ним доступ по их индексу – порядковому номеру параметра в командной строке. Отсчет параметров начинается с единицы, то есть если передано три параметра, то первый будет иметь индекс, равный единице, второй – двойке и т. д. Под нулевым индексом идет полный путь к выполняющейся программе.

Создадим небольшое тестовое приложение. Сымитируем примитивнейший текстовый редактор, который будет открывать файл, переданный ему в качестве параметра, или, если приложению не передано ни одного параметра, редактор будет просто запускаться.

Откройте тестовый проект. Вид его главной формы приведен на рис. 3.29.

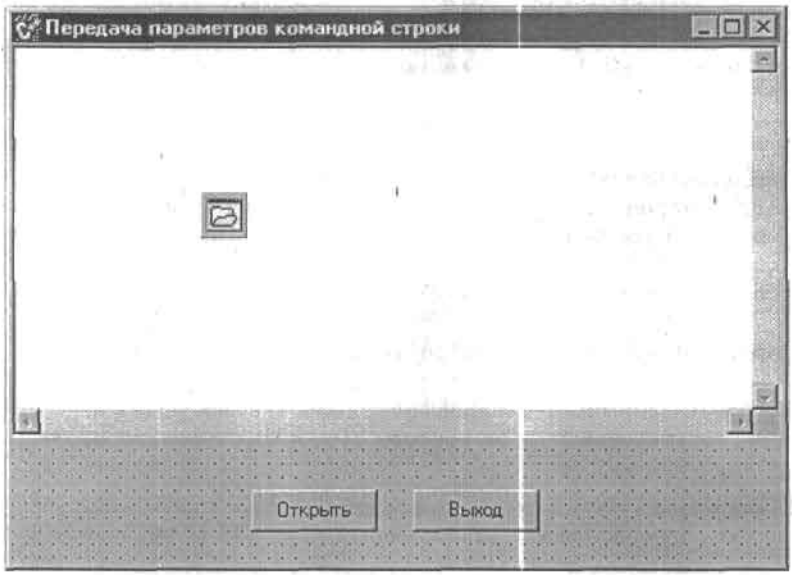


Рис. 3.29. *Внешний вид формы тестового проекта во время разработки*

Компонент ***TOpenDialog*** нужен только лишь как демонстрация того, что работа с командной строкой нисколько не мешает открывать файлы обычным образом.

Перейдите теперь в код проекта и добавьте в него код, исходя из высказанных выше положений. Код проекта должен выглядеть следующим образом.

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "CommandLineParametersUnit.h"

USERES("CommandLineParametersProject.res");
USEFORM("CommandLineParametersUnit.cpp", MainForm);
//---------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
  {
    try
      {
        Application->Initialize();
        Application->CreateForm(__classid(TMainForm), &MainForm);
```

```
        // проверяем количество переданных параметров в командной строке
        if(ParamCount())
          {
           if(FileExists(ParamStr(1))) // если переданный файл существует
              MainForm->MainMemo->Lines->LoadFromFile(ParamStr(1));
          }
        // конец оператора if(ParamCount())


        Application->Run();
      }
    catch (Exception &exception)
      {
        Application->ShowException(&exception);
      }
    return 0;
  }
//---------------------------------------------------------------------
```

Если приложению передан параметр и он представляет собой имя файла, то приложение откроет его и отобразит содержимое в компоненте ***TMemo*** формы. Если не передано ни одного параметра или данный параметр не является именем файла, приложение просто запустится, оставив ***MainMemo*** пустым.

Хочу также обратить ваше особое внимание на включение в код файла проекта строки

```
#include "CommandLineParametersUnit.h"
```

Эта строка необходима для того, чтобы была возможность обращаться к компонентам, а также к методам и данным класса ***TMainForm***. Не забывайте про нее.

Теперь запустите проект, передав ему в качестве параметра какой-либо файл. Я передал ему один из файлов проекта:

```
CommandLineParametersProject.exe CommandLineParametersUnit.h
```

Вот как выглядит приложение, запущенное с данным параметром (рис. 3.30).

Теперь вы знаете, что передача параметров в приложение через командную строку – дело очень простое и полезное. Удостоверьтесь, что и без параметра приложение запускается как ни в чем не бывало. Остальной код проекта не представляет никакого интереса, и вы сможете его посмотреть сами, загрузив среду разработки.

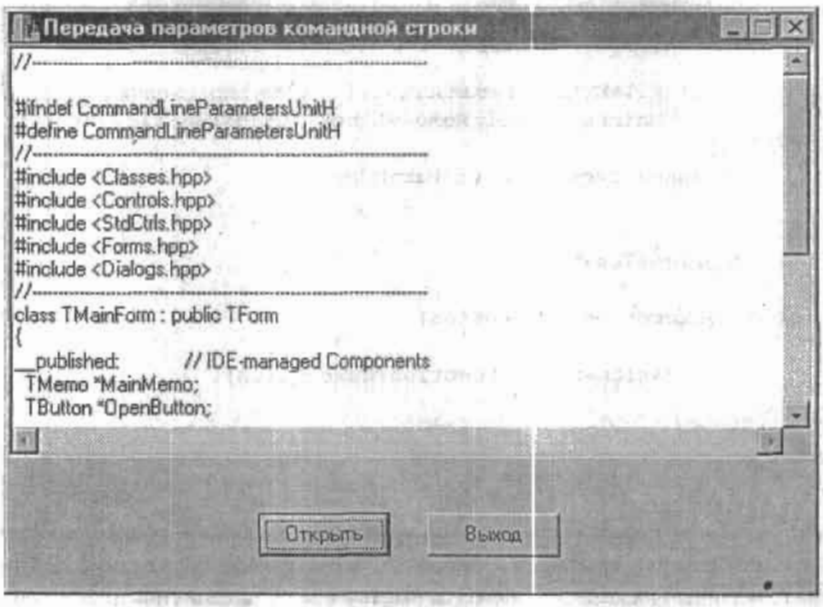


Рис. 3.30. *Вид приложения с прошедшей успешно передачей параметра*

Работа с реестром и создание файловой ассоциации

Ранее было показано, как можно реализовать передачу в приложение параметров командной строки и их обработку. Воспользуемся полученными ранее знаниями и разовьем данную тему: рассмотрим, каким образом можно создать файловую ассоциацию в Windows. Вы можете спросить, при чем тут обработка параметров командной строки? Дело в том, что в реализации одного из способов создания файловой ассоциации как раз и используется обработка приложением параметров командной строки. Более того, создание ассоциации требует достаточно интенсивной работы с реестром, так что мы можем одновременно совместить приятное с полезным: и файловую ассоциацию создать, и работу с реестром изучить.

В качестве основы для нашего тестового проекта возьмем проект из главы "Параметры командной строки" и несколько переделаем его (внешний вид формы тестового проекта приведен на рис. 3.31).

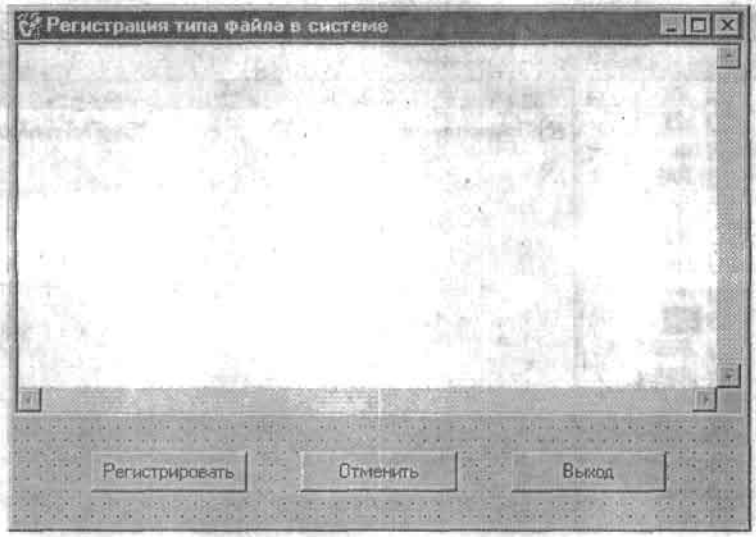


Рис. 3.31. *Внешний вид формы тестового проекта во время разработки*

Как видно, мы убрали кнопку "Открыть" и добавили кнопки "Регистрировать" (***RegFileTypeButton***) и "Отменить" (***UnregFileTypeButton***). При нажатии кнопки "Регистрировать" происходит регистрация файловой ассоциации в системе, а при нажатии пользователем кнопки "Отменить" происходит удаление ранее созданной файловой ассоциации из системы.

На всякий случай хочу пояснить, что я имею в виду под созданием файловой ассоциации. Под термином "создание" я подразумеваю выполнение такой последовательности действий, в результате которой файлы определенного нами типа (с указанным расширением имени файла) станут открываться неким конкретным приложением по двойному щелчку мыши на файле данного типа в Проводнике Windows. То есть ничего необычного. Просто краткое пояснение для полной ясности.

В качестве расширения имени файла для создания ассоциации возьмем расширение "rfte" (сокращение от "register file type example" – "пример регистрации типа файла"). Все файлы в системе с таким расширением должны будут открываться нашим тестовым приложением. (Мы, конечно, не будем придумывать свой формат файлов. Это будут обычные текстовые файлы, просто вместо расширения "txt" у них будет расширение "rfte".)

Теперь я расскажу о той самой последовательности действий, выполнение которой необходимо для создания файловой ассоциации.

Первым шагом в этой последовательности является создание подраздела с именем ".rfte" (без кавычек, но обязательно с точкой) в разделе реестра HKEY_CLASSES_ROOT.

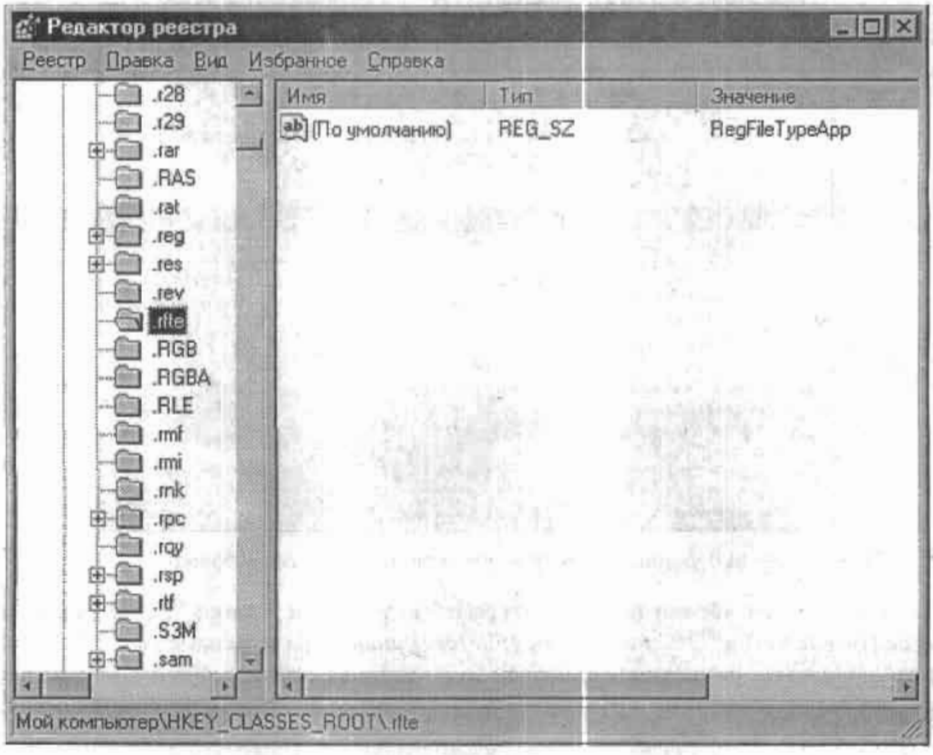


Рис. 3.32. *Подраздел ".rfte" в разделе HKEY_CLASSES_ROOT*

Опять необходимо сделать небольшое отступление. Насчет используемой в данном тексте терминологии. Дело в том, что раньше, лет так пять назад, термины, используемые для описания реестра в книгах и статьях, были вполне устоявшимися. Однако в последнее время каких терминов только не встречается: разделы, подразделы, ключи, ветви и даже ульи (буквальный перевод английского слова "hive", применяемого в англоязычной литературе для описания раздела реестра). Я использую термины "раздел реестра" и "подраздел реестра" для обозначения той части реестра, которая отображается в regedit как открытая папка на рис. 3.2 и имеет имя ".rfte". В принципе термины "раздел" и "подраздел" обозначают одно и то же – раздел реестра, но "подраздел" я использую для обозначения дочернего раздела родительского раздела реестра для большей благозвучности, для избежания чрезмерного употребления слова "раздел" в одном предложении. Иногда я еще использую термин "ветвь". То, что на рис. 3.32 имеет имя "По умолчанию", я называю параметром раздела.

В разделе ".rfte" существует один параметр, параметр по умолчанию, который определяет приложение, открывающее данный тип файлов. Данный параметр имеет строковый тип, а значение его – это название того раздела реестра, который описывает обрабатывающее файлы данного типа приложение.

Вторым шагом будет создание этого раздела реестра. Подраздел "RegFileTypeApp" также создается в ветви HKEY_CLASSES_ROOT.

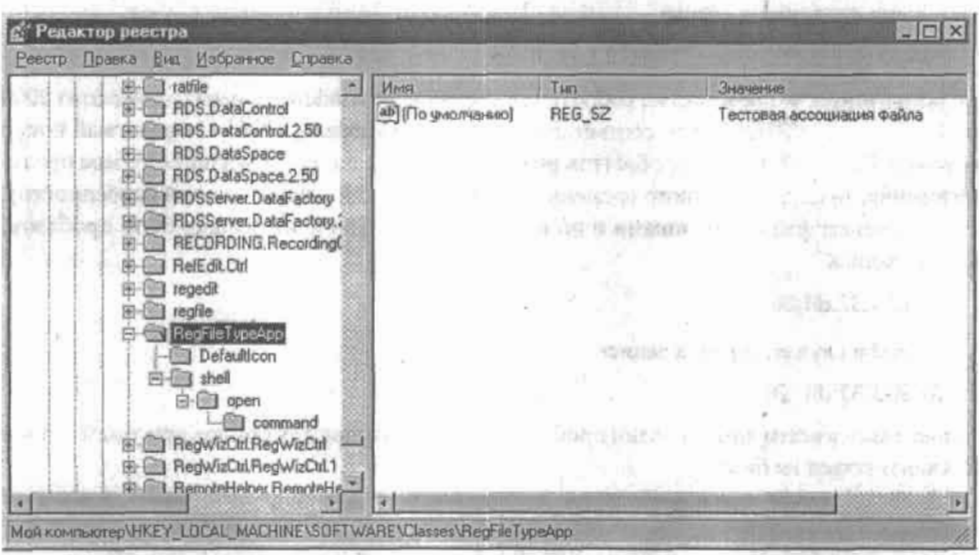


Рис. 3.33. *Подраздел "RegFileTypeApp" в разделе HKEY_CLASSES_ROOT*

Как видно из рисунка, в разделе "RegFileTypeApp", в свою очередь, содержится еще ряд подразделов: "DefaultIcon" и "shell", последний из которых содержит вложенный раздел "open", а раздел "open" содержит вложенный в него раздел "command".

Объясню, какую функцию выполняет каждый из подразделов раздела "RegFileTypeApp". Подраздел "DefaultIcon" определяет иконку, которая будет связана с файлами "*.rfte". Если этот параметр будет отсутствовать, то будет выводиться иконка по умолчанию, которая вряд ли будет радовать пользователей вашего приложения (рис. 3.34).

Рис. 3.34. *Стандартная иконка, которая будет связана с файлами "*.rfte" при отсутствии раздела "DefaultIcon"*

Но одного создания раздела "DefaultIcon" недостаточно. Значением параметра по умолчанию этого раздела должен быть путь к отображаемой иконке. Допускается использовать не только отдельные файлы иконок ("*.ico"), но и иконки, находящиеся как ресурсы в файлах других типов, – и это даже более распространено, чем указывание отдельных ico-файлов. В качестве примера я буду использовать библиотеку Windows SHELL32.dll, где расположено множество стандартных иконок для файлов. Значение параметра по умолчанию составляет следующая строка:

%SystemRoot%\system32\SHELL32.dll,26

где %SystemRoot%\ представляет собой путь к директории Windows (у меня под Windows 2000 это "C:\WINNT", SHELL32.dll соответственно файл с иконками, а "26" – порядковый номер иконки в SHELL32.dll). Хочу обратить ваше внимание (если вы не воспримете мое предупреждение, то потратите много времени на выявление этой специфической особенности): между именем файла с иконками и номером иконки в файле не должно быть пробелов. То есть запись

SHELL32.dll,26

как в нашем случае, верна, а запись

SHELL32.dll, 26

отличающаяся всего лишь на один пробел, – неверна. Иконка для файлов при такой записи устанавливаться не будет.

Далее. Для всех команд оболочки должен быть создан подраздел "shell", в котором необходимо перечислить названия команд. Имя команды может быть любым, но существует также ряд предопределенных команд Windows. Команда "open" – одна из них. Именно она ответственна за открытие файла связанным с ним приложением по двойному щелчку на файле в Проводнике. Команда "open" содержит подраздел "command", который и определяет, собственно, какое приложение будет открывать файлы данного типа.

Строковому параметру подраздела "command" в качестве значения должен быть установлен путь к приложению вместе с параметром командной строки, который соответствует выбранному файлу. У меня на машине для нашего тестового приложения он выглядит вот таким образом.

"C:\Documents and Settings\tnsoroka\Мои документы\Книга\Регистрация типа файла в системе\Project\RegisterFileTypeProject.exe" "%1"

И перед тем как мы приступим к рассмотрению собственно кода программы, последняя деталь: значение параметра по умолчанию раздела "RegFileTypeApp" установлено в "Тестовая ассоциация файла". Данная строка является описанием файловой ассоциации, и система представляет это пользователю, например, в следующем виде (см. столбец "Тип" на рис. 3.35).

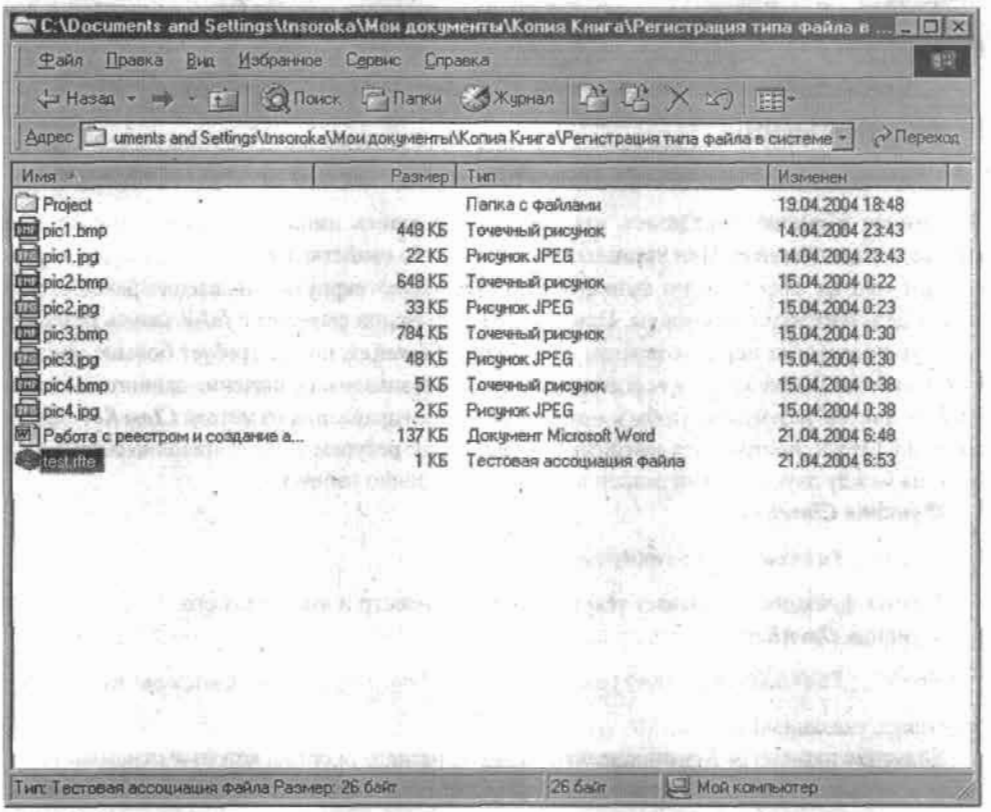


Рис. 3.35. *Описание файловой ассоциации (тип файла) в окне проводника Windows*

Вам необходимо это иметь в виду и давать вашим ассоциациям значимые, осмысленные описания, которые бы сразу давали понять пользователю, какое приложение ответственно за эти файлы и что вообще эти файлы собой представляют.

Теперь вплотную перейдем к рассмотрению процесса программного создания файловых ассоциаций.

Для работы с реестром в VCL существует класс ***TRegistry***, объявленный в файле Registry.hpp. Я рассмотрю некоторые из свойств и методов данного класса, которые нам понадобятся при создании файловой ассоциации в системе.

Свойство ***RootKey***, определяющее, с каким из разделов реестра будет происходить вся последующая работа:

```
__property HKEY RootKey = {read=FRootKey, write=SetRootKey, nodefault};
```

Свойство ***LazyWrite***:

```
__property bool LazyWrite = {read=FLazyWrite, write=FLazyWrite, nodefault};
```

Данное свойство определяет, как будет происходить запись разделов в реестр при вызове метода ***CloseKey***. При установке значения этого свойства в ***true*** запись разделов происходит при их закрытии, но функция ***CloseKey*** может вернуть управление раньше, чем запись действительно произошла. При установке значения свойства в ***false*** запись разделов в реестр происходит перед возвратом из функции ***CloseKey***, но это требует больше системных ресурсов. Тем не менее я всегда предпочитаю устанавливать значения данного свойства в ***false*** – считаю разумным, чтобы к моменту возврата управления из метода ***CloseKey*** запись данных в раздел реестра была завершена. А системные ресурсы... На современных машинах разница между двумя типами записи в реестр совершенно неощутима.

Функция ***CloseKey***:

```
void __fastcall CloseKey(void);
```

Данная функция записывает текущий раздел в реестр и закрывает его.

Функция ***OpenKey***

```
bool __fastcall OpenKey(const AnsiString Key, bool CanCreate);
```

открывает указанный раздел.

Значение параметра ***Key*** определяет название раздела реестра, который открывает или создает данная функция, а значение параметра ***CanCreate*** определяет, будет ли создан раздел, если на момент вызова функции его не существует. Установив значение этого параметра в ***true***, функция будет создавать несуществующий раздел. В случае успешного открытия или создания ключа ***OpenKey*** возвращает ***true***.

Функция ***DeleteKey***

```
bool __fastcall DeleteKey(const AnsiString Key);
```

удаляет раздел и все связанные с ним данные из реестра. В Windows NT/2000/XP, если раздел содержит вложенные разделы, перед удалением родительского раздела все дочерние должны быть явно удалены. В Windows 9X/ME явного удаления подразделов не требуется – при удалении родительского раздела все его дочерние подразделы будут также удалены. В случае успешного удаления ***DeleteKey*** возвращает ***true***.

И наконец, последняя функция класса ***TRegistry***, необходимая при создании файловой ассоциации, ***WriteString***:

```
void __fastcall WriteString(const AnsiString Name, const AnsiString Value);
```

Данная функция сохраняет строку, определяемую значением параметра ***Value***, в указанном параметре текущего раздела реестра. Имя связанного с текущим разделом реестра параметра определяется значением параметра ***Name*** функции. При сбое в записи функция выбрасывает исключение, и запись не производится.

На этом завершим краткий обзор свойств и методов ***TRegistry***, которые используются в создании файловой ассоциации. Описание остальных его свойств и методов вы можете посмотреть в справке по VCL, поставляемой вместе со средой разработки.

Теперь приступим к рассмотрению кода, который будет создавать и удалять файловую ассоциацию согласно рассмотренному выше алгоритму.

В нашем приложении мы работаем исключительно с разделом HKEY_CLASSES_ROOT, поэтому в обработчике события ***OnCreate*** формы находится следующая строка:

```
pRegistry->RootKey = HKEY_CLASSES_ROOT;
```

где ***pRegistry*** – это указатель на экземпляр класса ***TRegistry***, через который мы работаем с реестром в нашем тестовом приложении.

Хочу обратить ваше внимание, что HKEY_CLASSES_ROOT не является строковым значением, а является значением типа HKEY, поэтому должно использоваться без кавычек. Также в обработчике события ***OnCreate*** формы установим значение свойства ***LazyWrite*** в ***false***.

```
pRegistry->LazyWrite = false;
```

Код, который будет создавать в реестре соответствующие разделы и присваивать параметрам созданных разделов необходимые значения, находится в обработчике события ***OnClick*** компонента ***RegFileTypeButton*** (кнопка с заголовком "Регистрировать").

Создаем раздел в реестре, определяющий название расширения имени файла.

```
pRegistry->OpenKey("\\.rfte", true);
```

Значением параметра по умолчанию в данном разделе является "RegFileTypeApp" (в реестр оно будет записано без кавычек).

```
pRegistry->WriteString("", "RegFileTypeApp");
```

Первый параметр функции ***WriteString*** – пустая строка. Это означает, что мы записываем значение в параметр по умолчанию данного раздела реестра. Значение "RegFileTypeApp" определяет название другого раздела в HKEY_CLASSES_ROOT, подразделы которого и их параметры будут определять уже все остальные характеристики файловой ассоциации.

После записи строкового параметра для сохранения в реестре записанных значений и закрытия раздела вызываем функцию ***CloseKey***.

```
pRegistry->CloseKey();
```

Далее создаем соответствующие подразделы в созданном ранее разделе "RegFileTypeApp".

```
pRegistry->OpenKey("\\RegFileTypeApp", true);
pRegistry->WriteString("", "Тестовая ассоциация файла");
pRegistry->CloseKey();
```

Аналогичным образом:

```
pRegistry->OpenKey("\\RegFileTypeApp\\shell\\open\\command", true);
pRegistry->WriteString("", "\"" + Application->ExeName + "\"" + " \"%1\"");
pRegistry->CloseKey();
```

и:

```
pRegistry->OpenKey("\\RegFileTypeApp\\DefaultIcon", true);
pRegistry->WriteString("", "%SystemRoot%\\system32\\SHELL32.dll,26");
pRegistry->CloseKey();
```

На что в этом коде надо обратить внимание, так это на то, что пути к разделам реестра указываются абсолютные, а не относительные (подробнее об этих двух типах путей к разделам реестра можно прочитать в онлайновом справочнике, поставляемом с C++Builder). Я сам использую и вам рекомендую использовать абсолютные пути; с относительными иногда бывают сбои, да и в них проще запутаться.

Код удаления файловой ассоциации находится в обработчике события ***OnClick*** компонента ***UnregFileTypeButton*** (кнопки с заголовком "Отменить"). Он представляет собой обычное удаление созданных ранее разделов из реестра.

```
pRegistry->DeleteKey("\\.rfte");
```

Все остальные подразделы из раздела "RegFileTypeApp" удаляются аналогично. После вызовов метода ***DeleteKey*** метод ***CloseKey***, разумеется, вызывать не надо.

Запустите программу и создайте файловую ассоциацию. Что вы видите? Что ассоциация не работает. Если вы откроете редактор реестра, то вы обнаружите, что в реестре все разделы созданы и значения их параметров записаны. Так в чем же дело? Дело в том, что мы не сообщили операционной системе о том, что была создана новая файловая ассоциация и данные из реестра необходимо перечитать. Сообщить об этом факте системе можно с помощью вызова функции ***SHChangeNotify***, которая объявлена в файле shlobj.h, со следующими параметрами.

```
SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_PATH, NULL, NULL);
```

Данный вызов необходимо сделать после создания файловой ассоциации и ее удаления в соответствующих обработчиках событий ***OnClick*** кнопок. После того как вы добавили вызов функции в обработчики, удалите ассоциацию и создайте ее еще раз – и убедитесь в том, что система обновила свой список файловых ассоциаций. На рис. 3.36 показано, как должно выглядеть описание вновь созданной ассоциации.

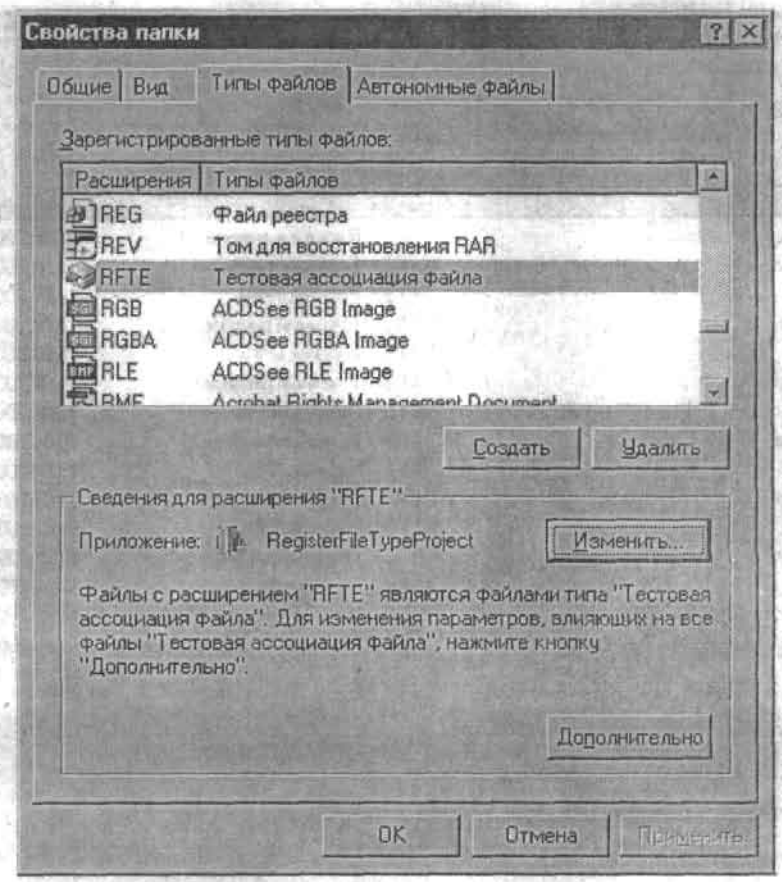


Рис. 3.36. *Отображение созданной файловой ассоциации в окне "Свойства папки"*

Ну и самая главная проверка – это открытие файла с расширением "rfte" по двойному щелчку на нем в Проводнике. Создайте в блокноте текстовый файл и сохраните его с расширением "rfte". Затем дважды щелкните по нему мышью – запустится наш тестовый проект, он загрузит в себя выбранный вами файл (рис. 3.37 и 3.38) по двойному щелчку на файле в Проводнике Windows.

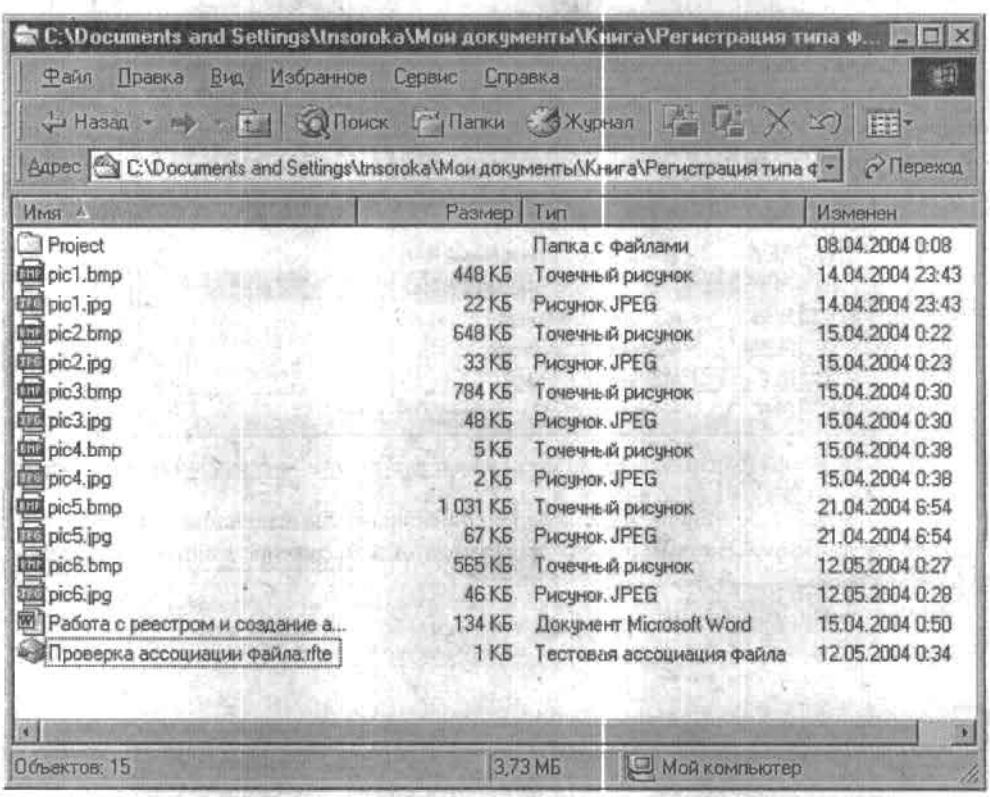


Рис. 3.37. *Файл с расширением "rfte" в Проводнике*

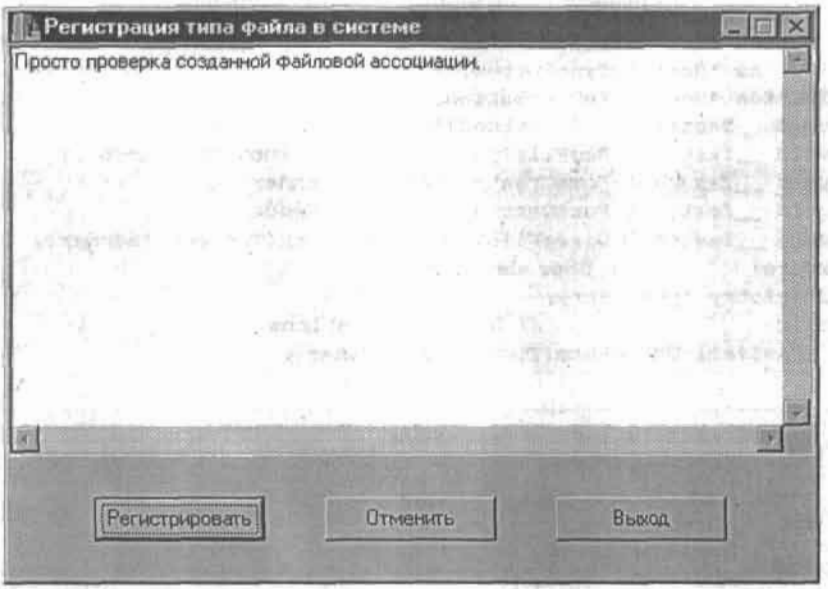


Рис. 3.38. *Открытие файла "Проверка ассоциации файла "rfte" тестовым приложением*

Теперь вы знаете достаточно о файловой ассоциации, чтобы использовать это полезное свойство операционной системы в своих программах. А в завершение данного материала – полный код проекта.

Заголовочный файл формы:

```
//---------------------------------------------------------------------

#ifndef RegisterFileTypeUnitH
#define RegisterFileTypeUnitH
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Dialogs.hpp>
#include <Registry.hpp>
//---------------------------------------------------------------------
class TMainForm : public TForm
{
__published:            // IDE-managed Components
```

```
  TMemo *MainMemo;
  TButton *ExitButton;
  TButton *RegFileTypeButton;
  TButton *UnregFileTypeButton;
  void __fastcall ExitButtonClick(TObject *Sender);
  void __fastcall RegFileTypeButtonClick(TObject *Sender);
  void __fastcall FormCreate(TObject *Sender);
  void __fastcall FormDestroy(TObject *Sender);
  void __fastcall UnregFileTypeButtonClick(TObject *Sender);
private:        // User declarations
  TRegistry *pRegistry;
public:                 // User declarations
  __fastcall TMainForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TMainForm *MainForm;
//---------------------------------------------------------------------------

#endif
```

Cpp-файл:

```
//---------------------------------------------------------------------
#define NO_WIN32_LEAN_AND_MEAN
#include <shlobj.h>
#include <vcl.h>
#pragma hdrstop

#include "RegisterFileTypeUnit.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
  : TForm(Owner)
    {
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::ExitButtonClick(TObject *Sender)
  {
    Application->Terminate();
  }
//-
```

```
void __fastcall TMainForm::RegFileTypeButtonClick(TObject *Sender)
  {
    // регистрируем в системе файлы с расширением rfte
    // от слов "register file type example"

    pRegistry->OpenKey("\\.rfte", true);
    // пустая строка, так как пишем в значение по умолчанию
    pRegistry->WriteString("", "RegFileTypeApp");
    // записали содержимое в реестр
    pRegistry->CloseKey();

    pRegistry->OpenKey("\\RegFileTypeApp", true);
    pRegistry->WriteString("", "Тестовая ассоциация файла");
    pRegistry->CloseKey();

    pRegistry->OpenKey("\\RegFileTypeApp\\shell\\open\\command", true);
    pRegistry->WriteString("", "\"" + Application->ExeName + "\" " \"%1\"");
    pRegistry->CloseKey();

    pRegistry->OpenKey("\\RegFileTypeApp\\DefaultIcon", true);
    pRegistry->WriteString("", "%SystemRoot%\\system32\\SHELL32.dll,26");
    pRegistry->CloseKey();

    // уведомляем систему о смене ассоциации файла
    SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_PATH, NULL,
    NULL);
  }
//---------------------------------------------------------------------
void __fastcall TMainForm::FormCreate(TObject *Sender)
  {
    pRegistry = new TRegistry();

    // соединяемся с разделом реестра HKEY_CLASSES_ROOT
    pRegistry->RootKey = HKEY_CLASSES_ROOT;

    pRegistry->LazyWrite = false;
  }
//---------------------------------------------------------------------
void __fastcall TMainForm::FormDestroy(TObject *Sender)
  {
    if(pRegistry != NULL)
      delete pRegistry;
  }
```

```
//---------------------------------------------------------------------
void __fastcall TMainForm::UnregFileTypeButtonClick(TObject *Sender)
  {
    pRegistry->DeleteKey("\\.rfte");

    // удаляем сначала все подразделы раздела RegFileTypeApp - для линейки
NT/W2K/XP
    pRegistry->DeleteKey("\\RegFileTypeApp\\shell\\open\\command");
    pRegistry->DeleteKey("\\RegFileTypeApp\\shell\\open");
    pRegistry->DeleteKey("\\RegFileTypeApp\\shell");
    pRegistry->DeleteKey("\\RegFileTypeApp\\DefaultIcon");

    // удаляем сам раздел RegFileTypeApp
    pRegistry->DeleteKey("\\RegFileTypeApp");

    // уведомляем систему о смене ассоциации файла
    SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_PATH, NULL,
    NULL);
  }
//---------------------------------------------------------------------
```

Программа работы со сканером

При разработке программ, связанных с обработкой документооборота, возникает необходимость организации ввода графического образа документов посредством сканера. На первый взгляд наиболее простым решением данной проблемы является вызов из программы соответствующей утилиты, поставляемой со сканером, и последующее чтение файла, полученного в результате этого сканирования. Но не намного сложнее, а может быть и проще, оказывается возможность организации непосредственного взаимодействия программы со сканером. Справедливость этого утверждения мы сейчас продемонстрируем на примере создания простейшего приложения, в котором попытаемся реализовать следующий функционал. Наше приложение должно:

- предоставлять возможность сканирования изображения с выбранного источника;
- выводить результат сканирования в окне;
- предоставлять возможность масштабирования изображения;
- сохранять изображение в файле с использованием JPEG-формата.

Для взаимодействия приложения со сканером используется интерфейс, имеющий аббревиатуру TWAIN, – индустриальный стандарт на программный интерфейс, предназначенный для работы со сканирующими устройствами. В настоящее время действует версия 1.9. О текущем состоянии стандарта можно узнать на официальном сайте *www.twain.org*. Текущая версия была выпущена в 2000 г. Сейчас ведется подготовка к выпуску версии 1.92.

Для упрощения работы с интерфейсом TWAIN разработано довольно много библиотек. Одной из них мы и воспользуемся для построения приложения. Свой выбор остановим на EZTwain, так как, во-первых, это наиболее распространенная и, во-вторых, открытая библиотека. Точнее, открытой является упрощенная версия EZTwain Classic, которая используется как Dll и поставляется с открытым кодом. Ознакомиться с ней, а также скачать ее можно с сайта *www.dosadi.com*. Последняя версия библиотеки датируется сентябрем 1999 г. и имеет номер 1.13.

Для начала создадим заготовку приложения, состоящую из одной формы. На форму "накидаем" компоненты ***TScrollBox***, ***TToolBar***, ***TActionList***, ***TSavePictureDialog*** и ***TImageList***. Дадим имя форме – ***PreviewForm*** и переименуем файлы формы Unit1.cpp и Unit1.h на UPreviewForm.cpp и UPreviewForm.h соответственно. На ***TScrollBox*** выкладываем ***TImage*** и на инструментальной панели (***TToolBar***) создаем пять кнопок. После соответствующих манипуляций со свойством ***Align*** у компонентов ***TScrollBox***, ***TToolBar*** и ***TImageList*** получаем примерно вот такую картинку.

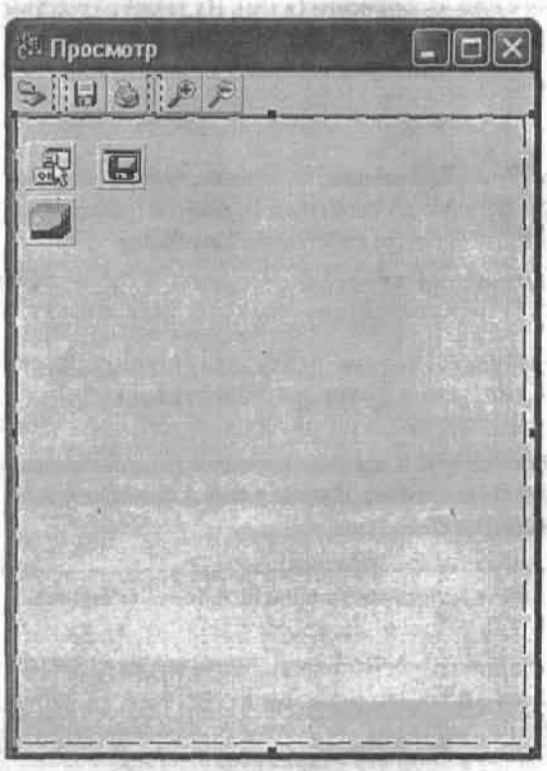


Рис. 3.39. *Внешний вид тестового проекта*

Подключаем библиотеку EZTwain посредством динамической подгрузки с использованием приемов, описанных в разделе "Применение шаблонов при динамическом связывании DLL с основным приложением". Для этого в файл UPreviewForm.h добавляем строчку

```
#include "Dll.h"
```

и в класс ***TPreviewForm*** в секцию ***private*** добавляем:

```
TDll* m_EzTwDll;
```

– указатель на объект "динамическая библиотека". Также нам понадобятся дополнительные члены в классе ***TPreviewForm***:

```
int m_Scale;
```

– масштабный множитель, который может изменяться в диапазоне от 25 до 800;

```
int m_Width;
int m_Height;
```

– фактическая ширина и высота сканированного изображения (в мм). Их также разместим в ***private***-секции.

В файле UPreviewForm.cpp зададим пределы изменения ***m_Scale*** с помощью констант:

```
const int cMaxScale = 800;
const int cMinScale = 25;
```

В конструкторе формы выполним загрузку библиотеки, инициализируем значением "100" член данных класса ***m_Scale*** и присвоим значения свойствам ***DefaultExt*** (расширение по умолчанию для сохраняемого файла) и ***Filter*** (фильтр) компонента ***SaveDialog***.

```
__fastcall TPreviewForm::TPreviewForm(TComponent* Owner)
    : TForm(Owner),m_Scale(100), m_EzTwDll(new TDll("EZTW32.dll"))
{
    SaveDialog->DefaultExt = GraphicExtension(__classid(TJPEGImage));
    SaveDialog->Filter = GraphicFilter(__classid(TJPEGImage));
}
```

Откроем окно для редактирования списка действий, выполнив двойное нажатие мышью на компоненте ***ActionList***, и создадим два действия (***Action***): ***ZoomInAction*** и ***ZoomOutAction***. Для этих действий зададим обработчики событий ***OnExecute***.

```
//---------------------------------------------------------------------
void __fastcall TPreviewForm::ZoomInActionExecute(TObject *Sender)
{
        m_Scale *= 2;
        Image->Height *= 2;
        Image->Width  *= 2;
}
//
```

```
void __fastcall TPreviewForm::ZoomOutActionExecute(TObject *Sender)
{
    m_Scale /= 2;
    Image->Height /= 2;
    Image->Width  /= 2;
}
//---------------------------------------------------------------------
```

При вызове обработчиков будут выполняться очень простые действия: ***m_Scale***, ***Image->Height*** и ***Image->Width*** будут либо увеличиваться в два раза (для ***ZoomInAction***), либо уменьшаться (для ***ZoomOutAction***).

Для того чтобы запретить возможность использования этих действий при отсутствии сосканированного образа в окне, а также при достижении минимального (для ***ZoomOutAction***) и максимального (для ***ZoomInAction***) значения масштаба определим обработчики для событий ***OnUpdate***.

```
//---------------------------------------------------------------------
void __fastcall TPreviewForm::ZoomInActionUpdate(TObject *Sender)
{
    ZoomInAction->Enabled = !Image->Picture->Bitmap->Empty &&
    m_Scale < cMaxScale;
}
//---------------------------------------------------------------------
void __fastcall TPreviewForm::ZoomOutActionUpdate(TObject *Sender)
{
    ZoomOutAction->Enabled = !Image->Picture->Bitmap->Empty &&
    m_Scale > cMinScale;
}
//---------------------------------------------------------------------
```

Теперь приступим непосредственно к написанию процедуры сканирования документа. Для этого нам потребуются следующие функции из библиотеки EZTwain.

Функция

```
int _stdcall TWAIN_ SelectImageSource(HWND);
```

предназначена для выбора источника получения данных из списка TWAIN-совместимых устройств. Возвращает 0, если выбор был сделан.

Функция

```
HANDLE __stdcall TWAIN_AcquireNative(HWND, int);
```

предназначена для получения изображения посредством вызова диалогового окна соответствующего устройства и последующей передачи образа в программу. Второй параметр определяет режим сканирования и при вызове всегда должен быть равен 0. Функция возвращает указатель на область памяти, содержащей полученные данные в DIB-формате.

Функция

```
HPALLETE __stdcall  TWAIN_ CreateDibPalette(HANDLE);
```

получает цветовую палитру образца. В качестве параметра передается значение, возвращенное функцией ***TWAIN_AcquireNative***.

Функция

```
void _stdcall TWAIN_DrawDibToDC(HDC, int, int, int, int, HANDLE, int, int);
```

передает данные в формате, совместимом с указанным контекстом устройства.

Функция

```
void __stdcall TWAIN_FreeNative(HANDLE);
```

освобождает память, выделенную под DIB-данные, с помощью функции ***TWAIN_AcquireNative***.

Функция

```
int void __stdcall TWAIN_IsAvailable(void);
```

проверяет наличие на компьютере TWAIN-менеджера.

Вот и весь набор функций из библиотеки EZTwain, которые будут задействованы в приложении. Естественно, сама библиотека этим набором не ограничивается. Создадим новое действие (***ScanAction***) в списке ***ActionList*** и зададим обработчик ***OnExecute*** для него. Листинг обработчика с комментариями приводится ниже.

```
//---------------------------------------------------------------
void __fastcall TPreviewForm::ScanActionExecute(TObject *Sender)
{
   // создаем объекты-обертки для организации получения и хранения адресов
   // импортируемых функций и последующего вызова их
   // объявление объектов static обеспечивает "одноразовость" выполнения
   // процедуры получения адреса
   static TDllStdProc1<int,HWND>
   SelectImageSource(*m_EzTwDll,"TWAIN_SelectImageSource");

    static TDllStdProc2<HANDLE, HWND, int>
   AcquireNative(*m_EzTwDll,"TWAIN_AcquireNative");

    static TDllStdProc1<HPALETTE,HANDLE>
   CreateDibPalette(*m_EzTwDll,"TWAIN_CreateDibPalette");

    static TDllStdProcV8<HDC, int, int, int, int,HANDLE, int, int>
   DrawDibToDC(*m_EzTwDll,"TWAIN_DrawDibToDC");

    static TDllStdProcV1<HANDLE>
FreeNative(*m_EzTwDll,"TWAIN_FreeNative");

// вызов функции для выбора источника
```

```
// если  выбор не был сделан (нажата кнопка "Cancel")
//  выполнение обработчика прекращается
    if (!SelectImageSource(Handle))
        return;
    // вызов диалогового окна сканирования
    // если образ не был возвращен, выполнение обработчика прекращается
    if (HANDLE BMHandle = AcquireNative(Handle,0))
    {
        try
        {
            Graphics::TBitmap* Bitmap =  Image->Picture->Bitmap;
            //получаем адрес на  структуру BITMAPINFOHEADER
            PBITMAPINFOHEADER Info =
            (PBITMAPINFOHEADER)GlobalLock(BMHandle);
           //получаем размер образа (в мм)
            m_Width  = 1000 * Info->biWidth/Info->biXPelsPerMeter;
            m_Height = 1000 * Info->biHeight/Info->biYPelsPerMeter;
          // заполняем палитру для Bitmap
            Bitmap->Palette =  CreateDibPalette(BMHandle);
            // задаем размеры Bitmap в пикселах
            Bitmap->Width  = Info->biWidth;
            Bitmap->Height = Info->biHeight;
            //копируем образ в Bitmap
            DrawDibToDC(Bitmap->Canvas->Handle,0,0,Bitmap->Width,
            Bitmap->Height,BMHandle,0,0);
        }
        __finally
        {
             //освобождаем  память, выделенную при сканировании
             FreeNative(BMHandle);
        }
    }
}
//---------------------------------------------------------------------------
```

Для управления доступностью ***ScanAction*** создадим обработчик ***OnUpdate*** с вызовом функции, проверяющей наличие установленного на компьютере TWAIN-менеджера.

```
//---------------------------------------------------------------------------
void __fastcall TPreviewForm::ScanActionUpdate(TObject *Sender)
{
    static TDllStdProc0<int> IsAvailable(*m_EzTwDll,"TWAIN_IsAvailable");
    ScanAction->Enabled =  IsAvailable();
}
//---------------------------------------------------------------------------
```

Осталось реализовать функции по сохранению образа в файле формата JPEG и вывода образа на печать. Для этого создадим в списке ***ActionList*** еще два действия: ***SaveImageAction*** и ***PrintAction*** и напишем обработчики ***OnExecute*** для них.

Код обработчика для ***SaveImageAction*** достаточно прост. Он состоит из вызова диалогового окна для ввода пути и имени файла, создания экземпляра класса ***TJPEGImage***, копирования в него графического образа с последующим сохранением образа в файле. Первое действие (вызов диалогового окна) выполняется посредством вызова метода ***Execute*** компонента ***TSavePictureDialog*** с именем ***SaveDialog***, копирование и сохранение – с помощью методов ***Assign*** и ***SaveToFile*** созданного экземпляра класса ***TJPEGImage***.

```
//---------------------------------------------------------------------
void __fastcall TPreviewForm::SaveImageActionExecute(TObject *Sender)
{
  if(SaveDialog->Execute())
  {
   std::auto_ptr<TJPEGImage> Jpeg(new TJPEGImage());
   Jpeg->Assign(Image->Picture->Bitmap);
   Jpeg->SaveToFile(SaveDialog->FileName);
  }
}
//---------------------------------------------------------------------
```

Теперь займемся распечаткой графического образа. Для этого используем ***TPrinter***, класс-обертку VCL вокруг Windows-интерфейса, обеспечивающего работу с принтером. Указатель на глобальный экземпляр получаем с помощью функции ***Printer***.

```
TPrinter* Printer  =  Printers::Printer();
```

Затем нам понадобятся размеры устройства вывода (в мм). Их мы можем получить с помощью функции ***GetDeviceCaps***.

```
int PageWidth = GetDeviceCaps(Printer->Handle,HORZSIZE);
int PageHeight = GetDeviceCaps(Printer->Handle,VERTSIZE);
```

Для того чтобы при выводе на печать не произошло искажение масштаба, необходимо обеспечить вывод не на всю страницу, а только на область, равную по размеру отсканированному графическому изображению. Напомним, что размер изображения в миллиметрах был получен из DIB-данных и сохранен в ***m_Width*** и ***m_Height***. Но для вывода нужно указать размеры области вывода не в миллиметрах, а в пикселах. Пересчет миллиметров в пикселы может быть легко выполнен, если вспомнить, что у ***TPrinter*** имеются свойства ***PageWidth*** и ***PageHeight***, в которых указан размер страницы принтера в пикселах. Зная размеры страницы в миллиметрах и пикселах, а также размеры графического образа в миллиметрах, можно легко получить размеры этого же образа в пикселах устройства вывода.

```
    TRect Rect(0,0, m_Width  * Printer->PageWidth / PageWidth,
                    m_Height * Printer->PageHeight / PageHeight);
```

Теперь осталось осуществить непосредственно сам вывод.

```
Printer->BeginDoc();
Printer->Canvas->StretchDraw(Rect, Image->Picture->Bitmap);
Printer->EndDoc();
```

Полный листинг обработчика приведен ниже.

```
//---------------------------------------------------------------------
void __fastcall TPreviewForm::PrintActionExecute(TObject *Sender)
{
   TPrinter* Printer  =  Printers::Printer();
   int PageWidth   = GetDeviceCaps(Printer->Handle,HORZSIZE);
   int PageHeight  = GetDeviceCaps(Printer->Handle,VERTSIZE);
   TRect Rect(0,0, m_Width  * Printer->PageWidth / PageWidth,
                   m_Height * Printer->PageHeight / PageHeight);
   Printer->BeginDoc();
   Printer->Canvas->StretchDraw(Rect, Image->Picture->Bitmap);
   Printer->EndDoc();
}
//---------------------------------------------------------------------
```

Для того чтобы ограничить доступность выполнения действий только теми моментами, когда имеется графический образ (нет смысла сохранять или выводить на печать пустой лист), создадим для этих действий обработчик события ***OnUpdate***. Код его будет одинаков для обоих действий.

```
//---------------------------------------------------------------------
void __fastcall TPreviewForm::ActionUpdate(TObject *Sender)
  {
    ((TAction*)Sender)->Enabled = !Image->Picture->Bitmap->Empty;
  }
//---------------------------------------------------------------------
```

Теперь осталось ввести в ***TImageList*** заранее подобранные иконки, сопоставить их соответствующим действиям, а сами действия сопоставить с кнопками на инструментальной панели. И программу можно компилировать и запускать.

Drag'n'Drop внутри элемента управления

Почему-то для многих программистов в C++Builder, и не только для начинающих, а и для достаточно опытных разработчиков, drag'n'drop является "больным вопросом", несмотря на то что в Borland сделали достаточно для того, чтобы реализация drag'n'drop не вызывала особых трудностей. Тем не менее трудности есть, а раз так, то нам придется их преодолеть. Правда, на мой взгляд, понятие "drag'n'drop" достаточно широко, и я разделяю работу с drag'n'drop на три направления.

Первое: drag'n'drop внутри одного элемента управления. Например, перетаскивание мышью элементов внутри одного списка (***TListBox***), узлов дерева в пределах древовидного списка (***TTreeView***) и прочие подобные действия я отношу к первому виду drag'n'drop.

Ко второму виду drag'n'drop я отношу перетаскивание мышью элементов управления и их частей в пределах одной формы (рис. 3.40).

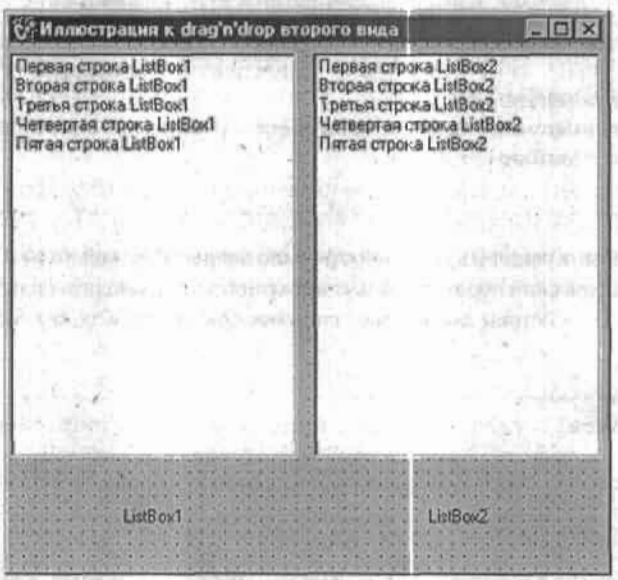


Рис. 3.40. *Drag'n'drop второго типа*

Перетаскивание мышью "первой строки ListBox1" в ***ListBox2*** как раз и будет являться тем, что я отношу к drag'n'drop второго типа. Также к нему относится, по моей классификации, перетаскивание второй метки (***TLabel*** с заголовком "ListBox2") на первую (с заголовком "ListBox1") или любой из меток в любой из списков.

И наконец, drag'n'drop третьего типа: перетаскивание чего-либо в окно вашего приложения извне его или, наоборот, из окна вашего приложения в окно другого приложения, или даже непосредственно в Windows (например, перетаскивание изображения из графического редактора на Рабочий стол для размещения этого изображения на Рабочем столе в качестве обоев). Примером drag'n'drop третьего, по моей классификации, типа является перетаскивание мышью документа Word в открытое окно Word. В принципе к данному виду drag'n'drop относится и перетаскивание чего-либо между разными окнами вашего собственного приложения, но это, по сути, лишь частный случай перетаскивания между разными приложениями.

Второй вид drag'n'drop мы рассматривать не будем, так как после данного материала на основе полученных знаний вы сможете его реализовать в ваших приложениях самостоятельно, а третий вид drag'n'drop мы рассмотрим отдельно и несколько позднее. Сейчас же мы будем двигаться по порядку и рассмотрим первый вид: перетаскивание мышью в пределах одного элемента управления.

В качестве рабочей задачи поставим себе перемещение элементов внутри ***TListView***. Это будет немного сложнее, чем аналогичная задача по перемещению элементов внутри ***TListBox*** и ***TComboBox***, и чуть легче, чем в ***TTreeView***, то есть как раз та сложность, что необходима для понимания и обучения.

Теперь немного теории. Давайте посмотрим, что в Borland сделали для того, чтобы наша задача была решена без затрат больших усилий. Справка по классу ***TListView*** сообщает нам, что для работы с drag'n'drop у ***TListView*** есть следующие свойства:

- DragCursor
- DragKind
- DragMode
- DropTarget

и события:

- OnDragDrop
- OnDragOver
- OnEndDrag
- OnStartDrag.

Нам понадобятся не все эти свойства и события. Мы обойдемся для наших целей свойствами ***DragKind***, ***DragMode*** и событиями ***OnDragDrop***, ***OnDragOver***.

Прежде чем заниматься реализацией drag'n'drop, его просто необходимо разрешить для данного элемента управления. Для этого и служит свойство ***DragMode***. Установка его в ***dmAutomatic*** разрешает операции drag'n'drop; установка его значения в ***dmManual*** переводит управление операциями перетаскивания в ручной режим, как и следует из названия значения свойства.

Значение свойства ***DragKind*** должно быть установлено в ***dkDrag***. Установка его в ***dkDock*** приведет к тому, что с элементом управления будут происходить не операции перетаскивания (drag'n'drop), а операции перемещения и встраивания (drag'n'dock).

Теперь о событиях.

Событие ***OnDragOver*** происходит во время перемещения объекта над элементом управления. Это событие унаследовано ***TListView*** еще от ***TControl*** и объявлено следующим образом.

```
typedef void __fastcall (__closure *TDragOverEvent)(System::TObject* Sender,
System::TObject * Source, int X, int Y, TDragState State, bool &Accept);
```

Значение параметра ***Sender*** определяет объект, в который пользователь перетаскивает другой объект, определяемый значением параметра ***Source***.

Значения параметров ***X*** и ***Y*** представляют собой координаты мыши в пикселах во время перетаскивания. Значения являются координатами элемента управления (определяемого параметром ***Sender***), а не экранными координатами (то есть точка (0, 0) находится в верхнем левом углу элемента управления, а не в левом верхнем углу экрана). Значение параметра ***State*** определяет, как объект перемещается над элементом управления. Может принимать одно из следующих значений.

Табл. 3.4. *Возможные значения параметра* ***State***

Значение	Пояснение
dsDragEnter	Мышь входит в элемент управления
dsDragLeave	Мышь покидает элемент управления
dsDragMove	Мышь перемещается над элементом управления

И наконец, параметр ***Accept***, который, обратите внимание, передается по ссылке. Это означает, что его можно изменить непосредственно в обработчике. Установка значения параметра ***Accept*** в ***false*** отклоняет принятие перемещенного объекта элементом управления. По умолчанию значение ***Accept*** установлено в ***true***.

Теперь о событии ***OnDragDrop***.

Это событие происходит, когда пользователь отпускает перемещаемый мышью объект. Говоря иными словами, оно происходит по завершении операции перетаскивания. Событие также унаследовано от ***TControl*** и объявлено следующим образом.

```
typedef void __fastcall (__closure *TDragDropEvent)(System::TObject* Sender,
System::TObject * Source, int X, int Y);
```

Параметры ***Sender***, ***Source***, ***X*** и ***Y*** имеют тот же смысл, что и в событии ***OnDragOver***.

Теперь можно приступать непосредственно к реализации нашей задачи. Итак, загрузите проект в C++Builder. На экране у вас должно быть нечто похожее на то, что изображено на рис. 3.41.

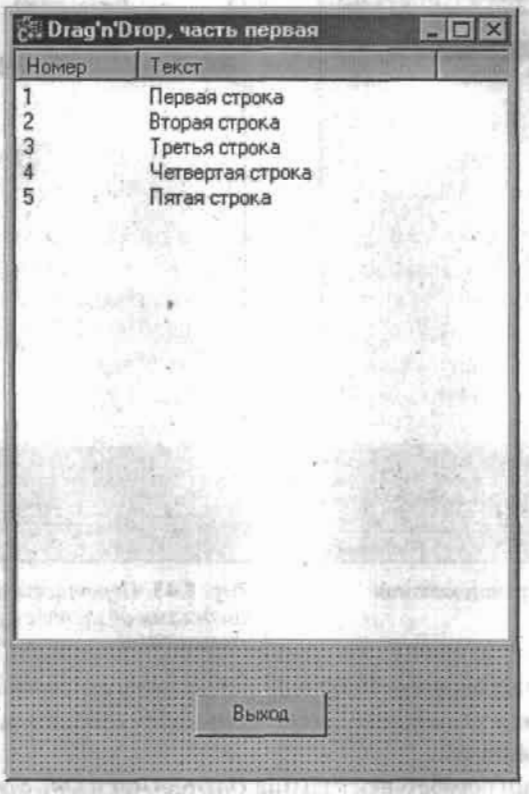


Рис. 3.41. *Тестовый проект, загруженный в C++Builder*

Запустите проект, выделите мышью какой-либо элемент в списке и попробуйте его переместить в пределах существующих элементов в ***TListView*** (см. рис. 3.42).

А теперь за пределами ***TListView*** или даже за пределами существующих элементов в списке (см. рис. 3.43).

Как вы можете видеть, все работает именно так, как ожидалось: элемент списка перемещается мышью куда надо и не перемещается куда не надо. Давайте разбираться, как это реализовано.

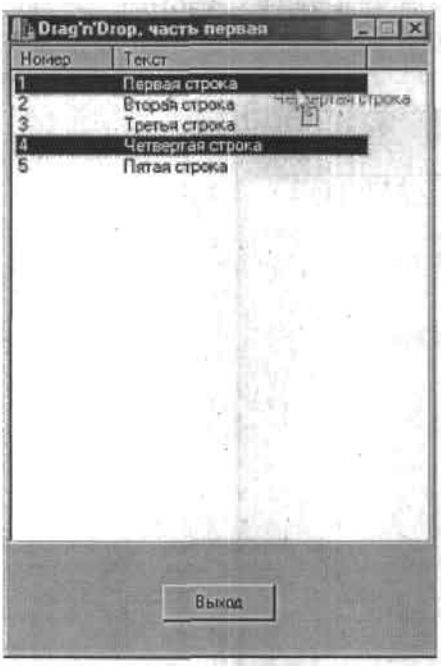


Рис. 3.42. *Пример перетаскивания элемента TListView*

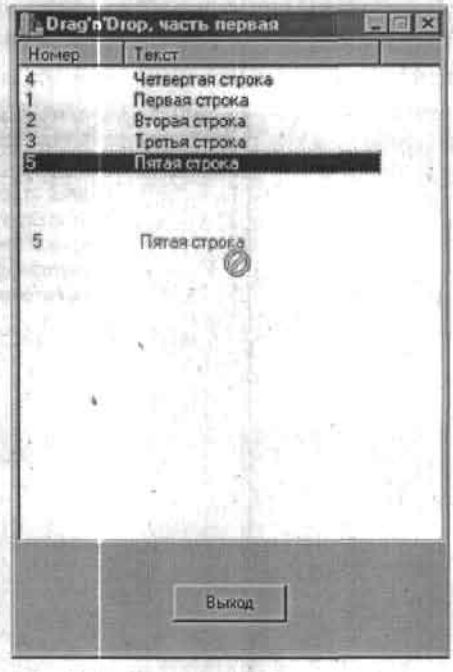


Рис. 3.43. *Перетаскивание элемента за пределами области с существующими элементами*

Во-первых, у списка свойство ***DragMode*** выставлено в ***dmAutomatic***, а ***DragKind*** установлено в ***dkDrag***, как и упоминалось ранее. Попробуйте изменить значения этих свойств и посмотрите, что получится.

Теперь рассмотрим обработчики событий ***OnDragOver*** и ***OnDragDrop***.

Обработчик события ***OnDragOver***.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::FirstListViewDragOver(TObject *Sender,
      TObject *Source, int X, int Y, TDragState State, bool &Accept)
   {
      if(FirstListView->GetItemAt(X, Y))
         Accept = true;
      else
         Accept = false;
   }
//---
```

Казалось бы, зачем вообще этот обработчик? Что можно обрабатывать во время перемещения элемента списка мышью? А попробуйте закомментировать этот код, и вы увидите, что при перетаскивании мышью элемента за пределы существующих элементов в списке курсор не меняется, и, судя по нему, перетаскивание на пустые области списка возможно (рис. 3.44).

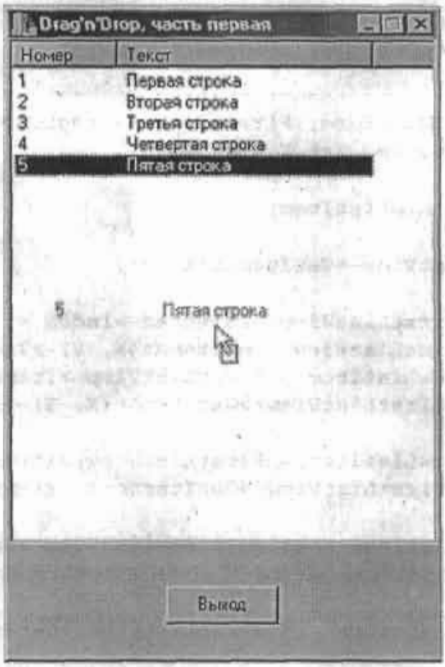


Рис. 3.44. *Неверное поведение перемещаемого элемента списка*

Но ведь это неверно! Нелогично перемещать элемент на пустое пространство (разве что если вы хотите сделать его копию). Поэтому в обработчик перемещения и добавлен этот код.

Строка

```
if(FirstListView->GetItemAt(X, Y))
```

как раз и определяет, есть ли в указанной точке с координатами ***X*** и ***Y*** элемент (***TListItem***) списка или нет, так как метод ***GetItemAt()*** возвращает ***TListItem****, если в указанной точке есть элемент списка, и NULL, если в указанной точке нет элемента списка.

Если в указанной точке не пусто, то строкой

```
Accept = true;
```

мы разрешаем перетаскивание в эту точку; иначе строкой

```
Accept = false;
```

мы его запрещаем.

Теперь переходим к обработчику события ***OnDragDrop***.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::FirstListViewDragDrop(TObject *Sender,
      TObject *Source, int X, int Y)
   {
      TListItem *NewListItem;

      if(FirstListView->GetItemAt(X, Y))
         {
            if(FirstListView->Selected->Index <
              FirstListView->GetItemAt(X, Y)->Index)
               NewListItem = FirstListView->Items->Insert
               (FirstListView->GetItemAt(X, Y)->Index + 1);
            else
               NewListItem = FirstListView->Items->Insert
               (FirstListView->GetItemAt(X, Y)->Index);

            NewListItem->Assign(FirstListView->Selected);
           FirstListView->Items->Delete(FirstListView->Selected->Index);
         }
      // конец оператора  if(FirstListView->GetItemAt(X, Y))
   }
//---------------------------------------------------------------------
```

Строкой

```
if(FirstListView->GetItemAt(X, Y))
```

мы опять ограничиваем область перемещения элемента уже существующими элементами. Если этого не сделать, то код

```
FirstListView->GetItemAt(X, Y)->Index
```

в случае, если в точке с координатами ***X*** и ***Y*** отсутствует элемент списка, вызовет Access Violation.

Строки

```
if(FirstListView->Selected->Index< FirstListView->GetItemAt(X, Y)->Index)
    NewListItem = FirstListView->Items->Insert
    (FirstListView->GetItemAt(X, Y)->Index + 1);
else
    NewListItem = FirstListView->Items->Insert
    (FirstListView->GetItemAt(X, Y)->Index);
```

обеспечивают корректную вставку элемента при перемещении его мышью как вверх по списку, так и вниз. Обратите внимание, кстати, что перемещаемый элемент списка мы получаем через

```
FirstListView->Selected
```

Другого способа его получить просто нет. А как же параметры ***Source*** и ***Sender***, спросите вы. Дело в том, что все они будут указателями на наш с вами ***TListView*** (***FirstListView***). Вот так. Можете проверить это самостоятельно.

Строка

```
NewListItem->Assign(FirstListView->Selected);
```

присваивает вставленному методом ***Insert*** элементу наш, выбранный мышью и перемещенный элемент, а строка

```
FirstListView->Items->Delete(FirstListView->Selected->Index);
```

удаляет его с предыдущей позиции. Приходится делать так, поскольку, к сожалению, у ***TListView*** и у объекта ***TListItems*** отсутствует метод ***Move*** – метод перемещения элемента списка вверх или вниз. И цепочка вызовов ***Insert*** и ***Delete*** как раз эмулирует этот метод.

Ну вот, собственно, и все. Поиграйтесь с тестовым проектом, оцените приобретенные знания и напишите для тренировки аналогичный код для ***TListBox***. Теперь это для вас не должно представлять никаких трудностей.

Внешний Drag'n'Drop

Теперь вы познакомитесь с еще одним из видов drag'n'drop – "внешним" drag'n'drop, иначе – drag'n'drop третьего типа по моей, приведенной ранее классификации. В качестве конкретной задачи поставим себе целью сделать приложение, реализующее одну из функций современного текстового редактора – прием перемещаемого мышью файла и его открытие. Проект для демонстрации реализации данного механизма состоит из компонента ***TRichEdit*** (***RichEdit***), размещенного на форме (***MainForm***), и кнопки закрытия приложения (***ExitButton***). Его внешний вид во время разработки представлен на рис. 3.45.

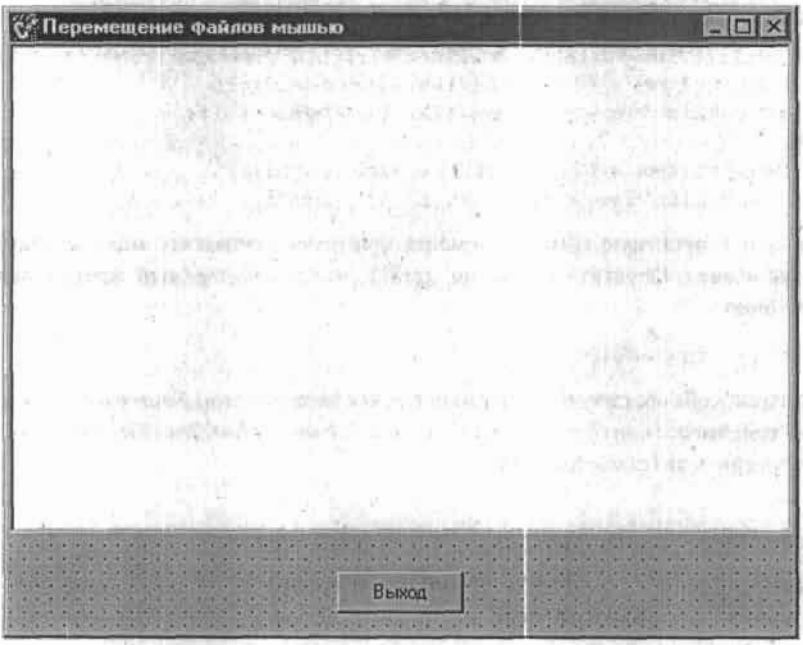


Рис. 3.45. *Форма проекта во время разработки*

Прежде чем начать, примем некоторые допущения.

- Приложение работает только с файлами определенных типов: "rtf" и "txt".
- Приложение не работает с перемещенными файлами, если их количество превышает единицу.

Эти допущения не скроют основного механизма реализации, но позволят не загромождать код излишними деталями. В случае необходимости вы без проблем добавите в ваши приложения работу с разными типами и количеством файлов.

Сам процесс открытия перемещенного файла состоит из следующих этапов.

1. Регистрация RichEdit как окна, способного принимать перемещенные файлы.
2. Создание обработчика события приема файлов для RichEdit.
3. Получение количества файлов.
4. Получение имени файла, если перемещен один файл.
5. Проверка типа файла на предмет того, работает ли с данным типом наша программа.
6. Загрузка содержимого файла в RichEdit при успешном выполнении пунктов 4 и 5.

C++Builder не предоставляет нам встроенных средств для реализации ряда намеченных этапов, поэтому придется воспользоваться Win32 API. Для обработки перемещенных мышью файлов в Win32 API предусмотрено сообщение WM_DROPFILES. В обработчике данного сообщения мы как раз и сможем получить все необходимые нам данные. Поскольку в компоненте ***TRichEdit*** нет никакого события, соответствующего сообщению WM_DROPFILES, то необходимо обрабатывать сообщение самостоятельно. Это можно сделать путем подмены оконной процедуры компонента ***TRichEdit***. Этот механизм довольно подробно рассмотрен ранее, и полагается, что читатель с ним уже знаком.

Я рассмотрю обработку сообщения WM_DROPFILES довольно кратко, акцентируя ваше внимание на наиболее ответственных моментах, а ниже приведу полный код проекта. Он несложен.

Для регистрации окна, как способного принимать перемещенные в него мышью файлы, служит функция ***DragAcceptFiles***. Данная функция описана в файле shellapi.h следующим образом:

```
VOID DragAcceptFiles(HWND hWnd, BOOL fAccept);
```

где ***hWnd*** – дескриптор окна, которому необходимо разрешить прием файлов в случае, если значение параметра ***fAccept*** равно ***true***, либо запретить его, если значение параметра ***fAccept*** равно ***false***.

Для того чтобы наш ***RichEdit*** смог принимать файлы, мы должны в конструкторе формы написать:

```
DragAcceptFiles(RichEdit->Handle, true);
```

После того как ***RichEdit*** получил возможность принимать файлы, при перемещении файлов ему будет отправляться сообщение WM_DROPFILES. Данное сообщение объявлено в файле winuser.h.

```
WM_DROPFILES
hDrop = (HANDLE) wParam;
```

В структуре, дескриптор на которую находится в параметре ***wParam*** сообщения, содержится информация, описывающая перемещенные файлы. Нам она понадобится в дальнейшем, а сейчас пока вернемся к тому факту, что в компоненте ***TRichEdit*** не предусмотрена обработка события WM_DROPFILES, и для подмены оконной процедуры ***RichEdit*** добавим в конструктор формы следующие строки для сохранения старой и назначения новой оконной процедуры.

```
OldWindowProc = RichEdit->WindowProc;
RichEdit->WindowProc = NewWindowProc;
```

Далее, мы должны в новой оконной процедуре обработать соответствующим образом полученное сообщение WM_DROPFILES, чтобы извлечь всю информацию, находящуюся в структуре. Дескриптор структуры передается в функцию ***DragQueryFile***:

```
UINT DragQueryFile(HDROP hDrop, UINT iFile, LPTSTR lpszFile, UINT cch);
```

которая в зависимости от значений параметров возвращает либо число перемещенных файлов, либо размер буфера для имени конкретного файла, либо само имя файла. Для более детальной информации смотрите описание функции в MSDN.

Итак, определяем количество перемещенных файлов.

```
FilesCount = DragQueryFile(hDrop, 0xFFFFFFFF, NULL, 0);
```

где ***FileCount*** – переменная типа ***int*** (определение всех переменных, упоминающихся ниже, смотрите в приведенном полном коде проекта).

Если значение этой переменной не больше единицы, то продолжаем работу и определяем размер буфера, необходимый для имени файла в байтах.

```
BufferSize = DragQueryFile(hDrop, 0, NULL, 0) + 1;
```

BufferSize также переменная целого типа. Значение второго параметра, равное нулю, – индекс первого файла в списке перемещенных файлов (индексация начинается с нуля). К возвращаемому функцией ***DragQueryFile*** размеру буфера прибавляем единицу – для завершающегося нулевого символа.

Мы уже готовы для решения задачи, поставленной в качестве четвертого этапа. Получаем имя файла:

```
DragQueryFile(hDrop, 0, FileName, BufferSize);
```

где ***FileName*** – динамический массив типа ***char*** размером ***BufferSize***.

Проверяя расширение имени файла, мы определяем, необходимо нам его загружать или нет. Если файл имеет расширение, отличное от "rtf" или "txt", – выводим на экран сообщение о том, почему данный файл не может быть загружен приложением. Иначе – загружаем этот файл в ***RichEdit***.

```
if((FileExt == ".txt") || (FileExt == ".rtf"))
  RichEdit->Lines->LoadFromFile(FileName);
else
  MessageDlg("К сожалению, данный тип файлов не поддерживается
\nприложением.", mtInformation, TMsgDlgButtons() << mbOK, 0);
```

Краткий обзор завершен, ниже приведен полный исходный код тестового проекта.

Заголовочный файл формы:

```
//---------------------------------------------------------------------
#ifndef DragDrop3UnitH
#define DragDrop3UnitH
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ComCtrls.hpp>
//---------------------------------------------------------------------
class TMainForm : public TForm
  {
  __published:          // IDE-managed Components
    TRichEdit *RichEdit;
    TButton *ExitButton;
    void __fastcall FormCreate(TObject *Sender);
    void __fastcall ExitButtonClick(TObject *Sender);
  private:       // User declarations
    TWndMethod OldWindowProc;
    void __fastcall NewWindowProc(Messages::TMessage &Message);
  public:                // User declarations
    __fastcall TMainForm(TComponent* Owner);
  };
//---------------------------------------------------------------------
extern PACKAGE TMainForm *MainForm;
//---------------------------------------------------------------------
#endif
```

Cpp-файл формы:

```
//---------------------------------------------------------------------
#include <vcl.h>
#include <shellapi.h>
#pragma hdrstop

#include "DragDrop3Unit.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
  : TForm(Owner)
```

```
{
}
//---------------------------------------------------------------------
void __fastcall TMainForm::FormCreate(TObject *Sender)
  {
    DragAcceptFiles(RichEdit->Handle, true);
    OldWindowProc = RichEdit->WindowProc;
    RichEdit->WindowProc = NewWindowProc;
  }
//---------------------------------------------------------------------
void __fastcall TMainForm::NewWindowProc(Messages::TMessage &Message)
  {
    if(Message.Msg == WM_DROPFILES)
      {
        HDROP hDrop; // структура с информацией о файлах
        int BufferSize; // размер буфера для имени файла
        char *FileName; // буфер для имени файла
        int FilesCount; // количество перемещенных файлов
        AnsiString FileExt; // расширение имени файла

        hDrop = HANDLE(Message.WParam);
        // получаем количество перемещенных файлов
        FilesCount = DragQueryFile(hDrop, 0xFFFFFFFF, NULL, 0);

        if(FilesCount <= 1)
          {
           // получаем размер буфера в байтах и увеличиваем его на единицу
            BufferSize = DragQueryFile(hDrop, 0, NULL, 0) + 1;
            FileName = new char[BufferSize];
            ZeroMemory(FileName, BufferSize);

            // получаем полное имя файла
            DragQueryFile(hDrop, 0, FileName, BufferSize);
            // получаем расширение имени файла
            FileExt = ExtractFileExt(FileName);

            if((FileExt == ".txt") || (FileExt == ".rtf"))
              RichEdit->Lines->LoadFromFile(FileName);
            else
             MessageDlg("К сожалению, данный тип файлов не поддерживается
\nприложением.", mtInformation, TMsgDlgButtons() << mbOK, 0);

            delete [] FileName;
```

```
        }
      else
        MessageDlg("Работа с несколькими файлами не поддерживается.",
                                  mtWarning, TMsgDlgButtons() << mbOK,
0);

      // конец оператора if(FilesCount = 1)
    }
  // конец оператора if(Message.Msg == WM_DROPFILES)
  OldWindowProc(Message);
 }
//---------------------------------------------------------------------
void __fastcall TMainForm::ExitButtonClick(TObject *Sender)
 {
  Application->Terminate();
 }
//---------------------------------------------------------------------
```

Перемещение элемента управления во время разработки

Ранее мы уже рассмотрели реализацию некоторых видов операции drag'n'drop. Ниже мы рассмотрим действие, не являющееся непосредственно drag'n'drop, но весьма близкое к нему по визуальным, так сказать, признакам и не очень далеко отстоящее по реализации. Это операция перемещения мышью непосредственно самих элементов управления в пределах одной формы.

Самый яркий пример реализации этой возможности – дизайнер форм в C++Builder. Также возможность перемещения элемента управления реализована, например, во всех редакторах векторной графики: вы берете элемент курсором мыши и перемещаете его мышью в пределах рабочей области. Разумеется, никакого подобия векторного редактора мы разрабатывать не будем, но вот основной принцип рассмотрим.

Загрузите в C++Builder тестовый проект. Вид его главной и единственной формы представлен на рис. 3.46.

На форме расположен единственный компонент – метка ***DragTestLabel***, который мы будем перемещать мышью во время выполнения приложения, точно так же как вы перемещаете его во время разработки приложения.

Что необходимо для перемещения компонента во время выполнения программы?

Во-первых, поскольку компонент должен перемещаться только после кликанья на нем мышью и последующего удерживания кнопки мыши в нажатом состоянии, необходимо как-то зафиксировать сам факт нажатия и удержания. Для этого воспользуемся обработчиком события ***OnMouseDown*** компонента ***TLabel***.

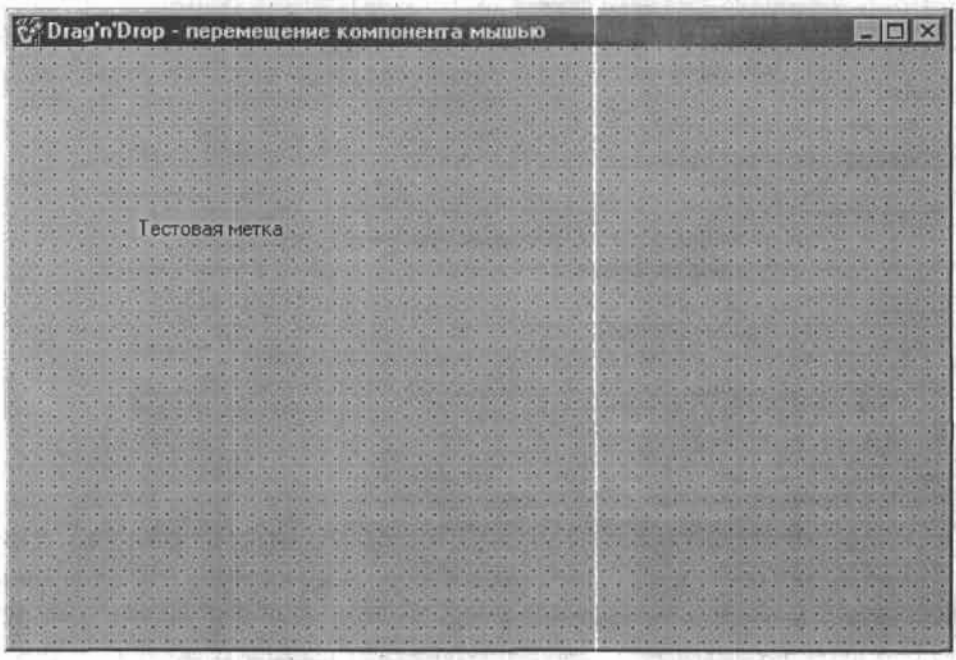


Рис. 3.46. *Внешний вид формы тестового проекта в design-time*

Заведем в классе формы в ее private-секции переменную ***DragDropFlag***:

```
private:
   bool DragDropFlag;
```

которая будет фиксировать, удержана кнопка мыши на метке или нет. Для этого в обработчике обработчиком события ***OnMouseDown*** напишем следующий код.

```
DragDropFlag = true;
```

До тех пор, пока не отпущена кнопка мыши, значение ***DragDropFlag*** будет равно ***true***.

Для фиксации отпускания кнопки мыши воспользуемся обработчиком события ***OnMouseUp*** компонента ***TLabel***. Добавьте в обработчик строку

```
DragDropFlag = false;
```

Таким образом, мы отследили нажатие и удерживание нажатой кнопки мыши на компоненте, а также ее отпускание и теперь можем перейти к реализации перемещения метки.

Может возникнуть вопрос: в каком же событии (и есть ли такое) можно реализовать перемещение метки во время выполнения? Такое событие есть. Это, как ни странно, событие ***OnMouseMove*** метки. Перемещение метки должно осуществляться лишь при удерживании нажатой левой кнопки мыши – для этого нам и понадобится переменная ***DragDropFlag***. Напишите следующий код в обработчике события ***OnMouseMove***.

```
if(DragDropFlag)
  {
    DragTestLabel->Left = DragTestLabel->Left + X;
    DragTestLabel->Top = DragTestLabel->Top + Y;

    Caption = "Drag'n'Drop - перемещение компонента мышью " +
              IntToStr(DragTestLabel->Left)
              + ": " + IntToStr(DragTestLabel->Top);
  }
```

Запустите проект и попробуйте подвигать метку по форме (рис. 3.47).

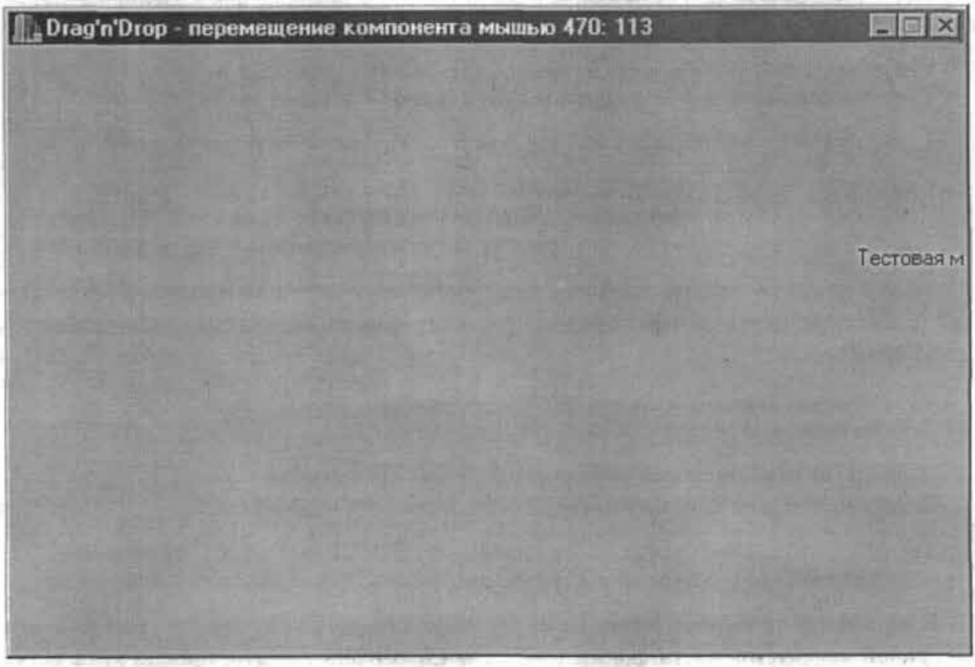


Рис. 3.47. *Перемещение метки мышью во время выполнения*

Двигается. Но сразу обнаруживается ряд недостатков.

Во-первых, на какой бы части метки вы ни остановили курсор мыши, при движении он всегда перескакивает на верхний левый угол метки. Во-вторых, метка свободно перемещается за пределы формы, и если вы туда ее уведете, то вернуть ее назад, в пределы окна, будет уже невозможно.

Для решения первой проблемы необходимо запомнить координаты того положения курсора мыши, когда он был зафиксирован при щелканье на метке. Для этого необходимо объявить две переменные для координат курсора мыши в секции private класса формы:

```
int DeltaX, DeltaY;
```

и присвоить им значения координат курсора в момент нажатия кнопки мыши на компоненте в обработчике события ***OnMouseDown*** метки.

```
DeltaX = X;
DeltaY = Y;
```

Затем необходимо ввести на эти величины поправку для координат курсора мыши во время перемещения компонента.

```
DragTestLabel->Left = DragTestLabel->Left - DeltaX + X;
DragTestLabel->Top = DragTestLabel->Top - DeltaY + Y;
```

Также нужно скорректировать координаты курсора, отображаемые в заголовке формы.

```
Caption = "Drag'n'Drop - перемещение компонента мышью " +
                  IntToStr(DragTestLabel->Left + DeltaX)
                  + ": " + IntToStr(DragTestLabel->Top + DeltaY);
```

Теперь осталось решить проблему с перемещением метки за пределы формы. Для того чтобы ограничить ее перемещение пределами формы, необходимо воспользоваться следующим кодом.

```
if(DragTestLabel->Left <= 0)
  DragTestLabel->Left = 0;
```

Данный код предотвращает уход метки за левый край формы.

Следующий код не позволит метке уйти за правый край формы.

```
if((DragTestLabel->Left + DragTestLabel->Width) >= ClientWidth)
  DragTestLabel->Left = ClientWidth - DragTestLabel->Width;
```

В первой строке кода мы вычисляем координаты правого края метки и сравниваем их с шириной клиентской части формы (свойство ***ClientWidth***), и если правый край метки вдруг ушел за пределы формы, то мы возвращаем метку к краю формы.

Аналогичным образом делается контроль координат перемещения метки по высоте – чтобы ее нельзя было переместить за нижний и верхний край формы.

Ниже приведен полный код проекта со всеми необходимыми изменениями.

В файле DragDrop2Unit.h:

```
private:
   bool DragDropFlag;
   int DeltaX, DeltaY;
```

В файле DragDrop2Unit.cpp:

```
//---------------------------------------------------------------------
void __fastcall TMainForm::DragTestLabelMouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
   {
      DragDropFlag = true;
      DeltaX = X;
      DeltaY = Y;
   }
//---------------------------------------------------------------------
void __fastcall TMainForm::DragTestLabelMouseMove(TObject *Sender,
TShiftState Shift,
      int X, int Y)
   {
      if(DragDropFlag)
         {
            DragTestLabel->Left = DragTestLabel->Left - DeltaX + X;
            DragTestLabel->Top = DragTestLabel->Top - DeltaY + Y;

           Caption = "Drag'n'Drop - перемещение компонента мышью " +
                     IntToStr(DragTestLabel->Left + DeltaX)
                      + ": " +
                      IntToStr(DragTestLabel->Top + DeltaY);

            // ограничиваем перемещение метки в пределах формы
            if(DragTestLabel->Left <= 0)
               DragTestLabel->Left = 0;
           if((DragTestLabel->Left + DragTestLabel->Width) >= ClientWidth)
             DragTestLabel->Left = ClientWidth - DragTestLabel->Width;
            if(DragTestLabel->Top <= 0)
               DragTestLabel->Top = 0;
           if((DragTestLabel->Top + DragTestLabel->Height) >= ClientHeight)
             DragTestLabel->Top = ClientHeight - DragTestLabel->Height;
         }
```

```
        // конец оператора if(DragDropFlag)
    }
//---------------------------------------------------------------------
void __fastcall TMainForm::DragTestLabelMouseUp(TObject *Sender,
TMouseButton Button,
      TShiftState Shift, int X, int Y)
    {
      DragDropFlag = false;
    }
//---------------------------------------------------------------------
```

Попробуйте теперь запустить проект и подвигать метку мышью. Вы увидите, что сейчас процесс перемещения происходит куда лучше, чем раньше, до внесения изменений (рис. 3.48).

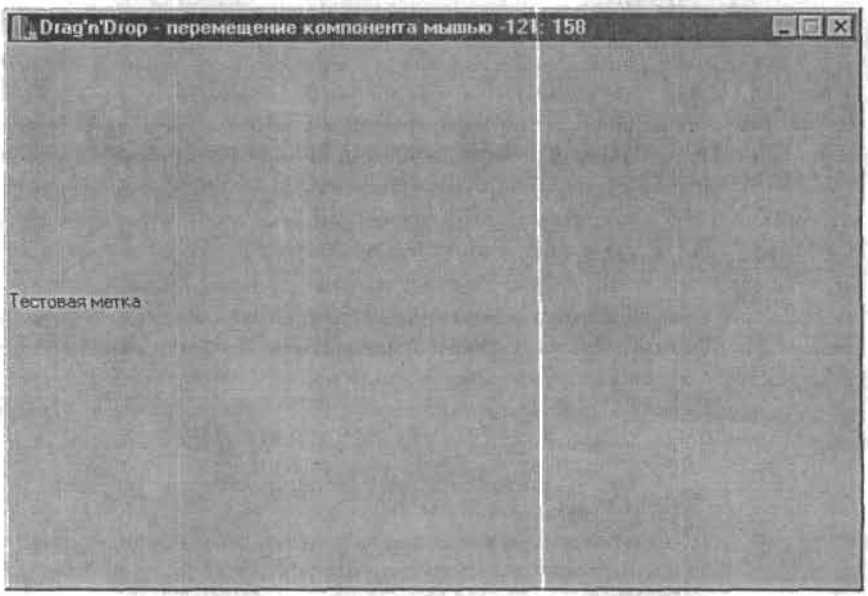


Рис. 3.48. *Внешний вид тестового проекта после внесения в код всех необходимых изменений*

Обратите внимание на заголовок формы. X-координате курсора мыши соответствует отрицательное значение – мышь ушла далеко за пределы левого края формы, но метка осталась у края и дальше не перемещается.

С дальнейшим развитием данной темы, если она вас заинтересовала, вы справитесь самостоятельно.

Работа с компонентами ActionManager и ActionMainMenuBar в дизайне и рантайме

Одним из нововведений, появившихся в C++Builder 6, является дальнейшее развитие механизма централизованного управления *Действиями* (*TAction*). Это развитие выразилось в появлении в палитре компонентов на вкладке Additional новых четырех компонентов.

Рис. 3.49. *Рассматриваемые компоненты на Панели компонентов C++Builder 6*

Первый из них – компонент ***TActionManager*** (Менеджер действий) – предназначен для выполнения примерно тех же задач, для которых ранее уже использовался хорошо известный компонент ***TActionList*** (Список действий). Об их общности говорит хотя бы тот факт, что оба компонента имеют один базовый класс – ***TCustomActionList***. Естественно, функционал нового компонента существенно расширен. Он теперь не только служит для создания, управления и группировки по категориям Списков действий, но и дает возможность использовать их в компонентах, являющихся производными от ***TCustomActionBar***. Менеджер действий позволяет подключить к себе "старые" Списки действий и использовать Действия из них в компонентах ***ActionBar***. Кроме того, Менеджер действий берет на себя заботу о сохранении пользовательских настроек внешнего вида меню и инструментальных панелей в файле настроек. Для задействования этой возможности достаточно указать имя файла в свойстве ***FileName***. Два следующих компонента – ***TActionMainMenuBar*** и ***TActionToolBar*** – как раз те контейнеры, с помощью которых можно отобразить пункты Действий, которые входят в список Менеджера действий. Первый компонент позволяет реализовать панель главного меню окна, второй – кнопочную панель инструментов. Четвертый компонент – ***TCustomizeDlg*** – компонентная обертка вокруг вызова окна настройки компонентов ***ActionBar***.

Что касается работы с ними в дизайн-режиме, то использование компонентов не представляет трудностей и практически все сводятся к принципу drag-and-drop. Продемонстрируем это на практике. Создадим новый проект. На главную форму проекта поместим компоненты ***TActionManager***, ***TActionMainMenuBar***, ***TActionToolBar***, ***TCustomizeDlg*** и ***TImageList***.

Для свойства ***Images*** компонента ***ActionManager1*** в инспекторе выберем из выпадающего списка значение ***ImageList1***, а напротив свойства ***FileName*** наберем значение ***SimpleEditor***. Вызовем редактор компонента ***ActionManager1*** через всплывающее меню (пункт меню ***Customize...***). В редакторе компонента опять же через всплывающее меню (пункт ***New Standart Action...***) вызовем окно списка классов стандартных действий (Standard Action Classes). В окне выберем все классы группы ***Edit***, ***Format***, ***Search***, а также классы ***TFileOpen*** и ***TFileSaveAs***.

Завершим выбор классов кнопкой "Ok". После этой операции на вкладке Actions редактора компонента в левой части появится список доступных категорий, в правой – Список действий, соответствующий выбранной категории. С помощью мыши "перенесем" последовательно категории на панель компонента ***ActionMainMenuBar1***. При выполнении этого действия для каждой группы сформируется ниспадающее меню. Теперь "перебросим" таким же образом группу "Search" на ***ActionToolBar***. Здесь проявится внешнее видимое отличие в поведении компонента ***ActionMainMenuBar*** от ***ActionToolBar***. Если в первом случае при выполнении этой операции создавалось меню, то во втором на панели ***ActionToolBar1*** будут созданы кнопки для каждого Действия, входящего в группу. Впрочем, при необходимости можно создать кнопку в стиле DropDown. Для этого опять же с помощью редактора компонента ***ActionManager1*** добавим новое действие, нажав кнопку ***New Action*** на вкладке Actions. Переименуем вновь созданное действие в ***AlignAction*** и "перенесем" его на ***ActionToolBar***. После выполнения этих манипуляций на тулбаре появится дополнительная кнопка ***AlignAction***. Именно ее создание являлось нашей целью. Создание Действия ***AlignAction*** являлось лишь вспомогательной операцией, и его можно удалить. Выберем кнопку ***AlignAction*** с помощью мыши и вызовем инспектор. Здесь нас ожидает сюрприз. Вместо инспектора класса, являющегося производным от ***TControl***, как можно было ожидать, откроется инспектор класса ***TActionClientItem***, который непосредственно никакого отношения к контролам не имеет. Экземпляры этого класса автоматически были созданы для каждого Действия, "перенесенного" на ***ActionToolBar*** или ***ActionMenuBar***. Присвоим свойству ***Action*** класса ***TActionClientItem*** значение ***RichEditAlignCenter1*** и, дважды щелкнув на свойстве ***Items***, откроем редактор коллекции и создадим в нем три экземпляра ***TActionClientItem***. Свойствам ***Action*** этих созданных ***ActionClientItem*** присвоим соответственно значения ***RichEditAlignLeft1***, ***RichEditAlignRight1***. В результате наша обычная кнопка превратится в DropDown-кнопку, состоящую из двух соединенных кнопок. При нажатии на левую ее часть будет выполняться действие, на которое указывает свойство ***Action***. При нажатии на правую часть появится ниспадающее меню, состоящее в нашем конкретном случае из трех пунктов.

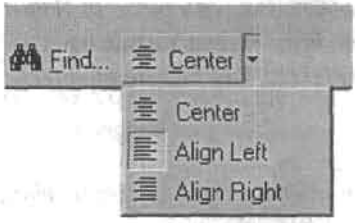


Рис. 3.50. *Кнопка с ниспадающим меню*

В качестве завершающего штриха добавим обработчик для события ***OnExecute*** компонента ***ActionManager1***.

```
//---------------------------------------------------------------------
void __fastcall TForm1::ActionManager1Execute(TBasicAction *Action,
      bool &Handled)
{
    class TProxyActionBar : public TActionToolBar
    {
    public:
        __property Items;
    };
    if (Action == RichEditAlignLeft1 ||
        Action == RichEditAlignRight1 ||
        Action == RichEditAlignCenter1)
    {
      TProxyActionBar* ActionBar = (TProxyActionBar*)ActionToolBar1;
      ((TActionClientItem*)ActionBar->Items->Items[4])->Action =
      (TContainedAction*)Action;
    }
}
//---------------------------------------------------------------------
```

Данный обработчик будет присваивать DropDown-кнопке последнее вызванное пользователем Действие из тех, которые перечислены в ниспадающем меню.

Дадим небольшой комментарий к тексту обработчика. Разработчики VCL по какой-то понятной только им причине ограничили доступ к свойству ***Items***, которое хранит коллекцию TActionClientItem, разместив это свойство в ***protected***-секции. Поэтому для доступа к нему нам понадобится производный промежуточный класс, у которого это свойство будет открыто. Далее приводим ***ActionToolBar*** к этому классу и получаем доступ к свойству. Наша кнопка ассоциирована с ***ActionClientItem*** с индексом 4; если пользователь вызвал одно из следующих действий: ***RichEditAlignLeft1***, ***RichEditAlignRight1***, ***RichEditAlignCenter1***, присваиваем указатель это Действие свойству ***Action***.

Рис. 3.51. *Результаты работы обработчика ActionManager1Execute*

Для того чтобы придать нашему приложению законченный вид, добавим возможность настраивать внешний вид меню и инструментальной панели. Эту задачу можно выполнить двумя способами. Первый – создать предопределенное стандартное Действие типа ***TCustomizeActionBars***, выбрав его из списка стандартных действий. Второй способ – задействовать компонент ***TCustomizeDlg***. Для этого нужно создать еще один Action и определить для него обработчик типа:

```
//---------------------------------------------------------------------
void __fastcall TForm1::CustomizeDialogExecute(TObject *Sender)
  {
    CustomizeDlg1->Show();
  }
//---------------------------------------------------------------------
```

В результате мы получили текстовый редактор, используя в основном возможности дизайн-режима. Но довольно часто использование только возможностей дизайн-режима недостаточно для выполнения поставленной задачи, и требуется выполнить аналогичные операции и в режиме рантайма. К сожалению, работа с Менеджером действий и с контейнерами типа ***ActionMainMenuBar*** и ***ActionToolBar*** не столь прозрачна. Попытаемся продемонстрировать ее на небольшом примере.

В настоящее время вследствие усложнения задач, которые ставятся перед разработчиком программного обеспечения, широкое распространение получил модульный принцип построения приложений. В этом случае приложение разбивается на основной exe-модуль и на ряд отдельных подгружаемых модулей. Основной модуль выполняется максимально простым. Как правило, его функционал ограничивается минимальными возможностями. К ним относятся простейшие сервисные функции общего назначения типа: установка принтера, обработка и журналирование возможных ошибок, проверка версионности и т. д. Основное же назначение главного модуля – подгрузка модулей и вызов функций из них по

требованию пользователя. Прикладной функционал размещают соответственно в подгружаемых модулях. Такое построение приложения позволяет достаточно легко наращивать функционал по мере необходимости, обеспечить простой механизм смены версий и разграничения доступа к функционалу, а также комфортно организовать разработку приложения при работе в команде.

Попробуем создать простейший пример подобного приложения, состоящий из основного и одного подгружаемого модуля. Создадим новый проект с одной главной формой. Вновь созданный проект переименуем в ***AppConsole***. Это будет наше главное приложение. Форму переименуем в ***MainForm***, а файл формы – в UmainForm.cpp. Свойству формы ***FormStyle*** присвоим значение ***fsMDIForm***, а на саму форму поместим компоненты ***ApplicationEvents***, ***ActionManager*** и ***ActionMainMenuBar***. В Менеджер действий (***ActionManager***) добавим стандартное действие ***TFileExit***. Полученную при этом категорию File "перенесем" на ***ActionMainMenuBar***. В результате на ***ActionMainMenuBar*** появится меню ***File*** с одним пунктом ***Exit***. На этом набор всех тех операций, которые можно выполнить, используя принцип "drag-and-drop", закончился. Дальше только ручное кодирование.

Прежде всего позаботимся об обработке ошибок, которые могут оказаться в нашем приложении. Для этого достаточно задать обработчик для события ***OnException*** компонента ***ApplicationEvents1***. В более сложном случае, используя это событие, можно выдавать довольно полную информацию об ошибке, записывать ее в журнал и даже, например, оповещать разработчика по электронной почте о произошедшей ошибке. Мы же просто ограничимся выводом информации, используя функцию ***ShowException***. В результате наш обработчик будет выглядеть следующим образом.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::ApplicationEvents1Exception(TObject *Sender,
      Exception *E)
  {
   Application->ShowException(E);
  }
//---------------------------------------------------------------------
```

Информацию о функциях, которые должны запускаться из подгружаемых модулей, мы будем держать в файле Modules.ini в следующем виде.

```
[Новое Окно]
Menu=Доп. функции
Module=Module.dll
Function=NewChildForm
```

Где имя секции – наименования пункта вертикального меню. Ключи Menu, Module и Function задают наименование горизонтального меню, наименование динамической библиотеки и имя функции соответственно.

Теперь реализуем функцию ***LoadIniFile***, которая будет считывать информацию из файла и формирования меню. Расшифровка кода дана в комментариях.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::LoadIniFile(void)
{
   //открытие ini-файла
   std::auto_ptr<TIniFile> IniFile(new TIniFile(GetCurrentDir() + "\\" +
  cIniFileName));
    // создание списка строк, в котором будут храниться наименования
    //секций
    std::auto_ptr<TStringList> SectionList(new TStringList());
    //считывание имен секций
    IniFile->ReadSections(SectionList.get());
    for(int i = 0; i < SectionList->Count; i++ )
    {
        // считывание наименования пункта главного меню
        AnsiString Menu = IniFile->ReadString(SectionList->Strings[i],
                                               cMenu, EmptyStr);
        // формирование меню
        if (!Menu.IsEmpty())
            CreateMenu(Menu, SectionList->Strings[i]);
    }
}
//---------------------------------------------------------------------
```

Непосредственно пункт меню формируется с помощью функции ***CreateMenu***. В качестве параметров функция получает две строки: наименование (***Caption***) горизонтального меню и наименование пункта меню. Функция создает экземпляр ***Action***, заполняет свойства ***Caption***, ***Category*** соответствующими значениями, событию ***OnExecute*** присваивает указатель на метод ***RunFunction***. Затем с помощью функции ***CreateMenuItem*** получает вновь созданный экземпляр класса ***TActionClientItem*** и свойству ***Action*** этого экземпляра присваивает указатель на ***Action***.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::CreateMenu(const AnsiString& HMenuName,
                                      const AnsiString& VMenuName)
{
    TAction* Action = new TAction(this);
```

```
    Action->Caption =VMenuName;
    Action->Category =  HMenuName;
    Action->OnExecute =  RunFunction;
    TActionClientItem* ClientItem = CreateMenuItem(HMenuName);
    ClientItem->Action = Action;
}
//---------------------------------------------------------------------
```

В задачу функции ***CreateClientItem***, как следует из названия, входит создание экземпляра класса ***TActionClientItem***. Текст этой функции размещен ниже.

```
//---------------------------------------------------------------------
TActionClientItem* TMainForm::CreateClientItem(const AnsiString& Name)
{
    class TProxyActionBar : public TCustomActionBar
    {
      public:
        __property Items;
        using TCustomActionBar::CreateControl;
    };

    TProxyActionBar* MenuBar =  (TProxyActionBar*)ActionMainMenuBar1;
    TActionClientItem* Item =  FindMenu(MenuBar->Items, Name);
    if (Item)
    {
      Item = Item->Items->Add();
    }
    else
    {
        Item = MenuBar->Items->Add();
        Item->Caption = Name;
        Item->Items->Add();
         MenuBar->CreateControl(Item);
        Item = (TActionClientItem*)Item->Items->Items[0];
    }
    return Item;
}
//---------------------------------------------------------------------
```

Прежде всего отметим следующий момент. Разработчики библиотеки VCL, по всей видимости, не очень рассчитывали, что кому-то придет в голову формировать меню в рантайме, и поэтому практически не предоставили никаких средств для выполнения этой задачи. Более того, свойство ***Items*** компонента ***TActionMainMenuBar*** размещено в секции ***protected*** и недоступно для непосредственного использования. Для того чтобы получить доступ к нему, а также к функции ***CreateControl***, мы воспользовались промежуточным классом ***TProxyActionBar***, в котором свойство и метод объявлены как ***public***. После приведения ***ActionMainMenuBar*** к этому классу мы получили доступ к необходимым для выполнения задачи элементам. Поскольку этот класс нигде, кроме этой функции, не задействован, вполне логичным выглядит объявление его в теле функции. Далее с помощью функции ***FindMenu*** производится поиск экземпляра ***TActionClientItem***, который соответствует пункту горизонтального меню, в которое будет произведена вставка. Если такой пункт горизонтального меню существует, то к его коллекции подпунктов добавляется очередной подпункт с помощью метода ***Add()*** и указатель на добавленный экземпляр ***TActionClientItem*** возвращается в качестве результата.

Немного сложнее дело обстоит в том случае, если экземпляр ***TActionClientItem*** для пункта горизонтального меню еще не был создан. В таком случае он предварительно создается с помощью метода ***Add*** коллекции ***Items*** компонента ***ActionMainMenuBar***, затем его свойству ***Caption*** присваивается наименование горизонтального меню, а уж затем к нему добавляется подпункт меню. Но дело этим не заканчивается. Одновременно с созданием экземпляра ***TActionClientItem*** компонент ***ActionMenuBar*** создает ему соответствующий контрол ***TStandardMenuButton***, и поскольку его создание происходило до того, как было присвоено значение свойству ***Caption*** и был добавлен подпункт, эти изменения на внешнем виде и поведении контрола никак не отразятся. Одним из возможных решений этой проблемы является принудительное пересоздание контрола с помощью метода ***CreateContol***. Вот и все тонкости работы с ***ActionMainMenuBar*** в рантайм-режиме. Аналогично происходит работа с ***ActionToolBar***.

Если в процессе работы возникает необходимость включения созданного динамически Действия (***TAction***) в Менеджер действий (***ActionManager***), то эта операция выполняется посредством присвоения указателя на Менеджер действий (***ActionManager***) свойству ***ActionList***.

```
Action->ActionList = ActionManager1;
```

Для полноты картины осталось дать текст функции, осуществляющей поиск пункта горизонтального меню FindMenu, и функции загрузки динамических библиотек и вызова импортируемых функций. Код первой функции показан ниже.

```
//---------------------------------------------------------------------
TActionClientItem* __fastcall TMainForm::FindMenu(TActionClients* Items,
const AnsiString& Name)
{
```

```
    for(int i = 0; i < Items->Count; i++)
    {
        TActionClientItem* Item = (TActionClientItem*)Items->Items[i];
        if (Item->Caption == Name)
        {
            return Item;
        }
    }
    return NULL;
}
//---------------------------------------------------------------------
```

Он достаточно прозрачен, и его оставим без комментариев. Реализации вызова импортируемых функций могут быть достаточно разнообразны. Мы же воспользуемся уже описанными ранее функциями работы с динамическими библиотеками, при использовании которых код функции может выглядеть примерно таким образом.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::RunFunction(TObject* Sender)
{
    TAction* Action = (TAction*)Sender;
    std::auto_ptr<TIniFile> IniFile(new TIniFile(GetCurrentDir() + "\\" +
    cIniFileName));
    AnsiString ModuleName = IniFile->ReadString(Action->Caption,
                                        cModule, EmptyStr);
    AnsiString FunctionName = IniFile->ReadString(Action->Caption,
                                        cFunction, EmptyStr);
    Dll::TDll Dll(ModuleName.c_str(), false);
    Dll::TDllProcV0 Proc(Dll, FunctionName.c_str());
    Proc();
}
//---------------------------------------------------------------------
```

Вот в принципе и все. Приложение, которое динамически формирует свое меню, готово.

Получение списка файлов в каталоге

Довольно часто возникает необходимость получить список файлов в каком-то из каталогов. Давайте рассмотрим, какие инструменты нам предоставляет VCL для нашей задачи.

А инструменты практически все те же, со времен DOS еще, с проверенными временем названиями: ***FindFirst***, ***FindNext***, ***FindClose***. И в дополнение к ним вспомогательная структура ***TSearchRec***.

Рассмотрим назначение этих функций и структуры.

FindFirst объявлена в модуле Sysutils.hpp как

```
extern PACKAGE int __fastcall FindFirst(const AnsiString Path, int Attr,
TSearchRec &F);
```

Функция ищет в каталоге первый файл с указанным именем и атрибутами и результат возвращает в параметре F. В случае успешного поиска возвращается нуль, в случае неудачи – код ошибки Windows.

Параметр ***Path*** задает каталог, в котором происходит поиск файла, и непосредственно само имя файла. То есть значение параметра должно быть полным, квалифицированным именем файла, например "C:\\My Documents\\Myfile.doc". В данном случае функция будет искать файл "Myfile.doc" в каталоге "C:\My Documents". Также в значении параметра допустимо указывать маски файлов: символы "?" и "*". То есть вполне допускается такое значение параметра Path: C:\\My Documents\\Myf*.do?". И именно этот случай нам наиболее интересен, как будет показано в дальнейшем.

Параметром ***Attr*** задаются файлы, которые необходимо включать в поиск согласно значениям их атрибутов. Список значений параметра ***Attr*** приведен в таблице.

Табл. 3.5. *Список возможных значений параметра* ***Attr***

Значение	Пояснение
faReadOnly	Файлы только для чтения
faHidden	Скрытые файлы
faSysFile	Системные файлы
faVolumeID	Идентификаторы томов
faDirectory	Директории
faArchive	Архивные файлы
faAnyFile	Любые файлы

Разумеется, у каждого параметра есть еще и его числовое значение, но это нас не интересует, так как в подавляющем большинстве случаев все эти атрибуты используются по именам, а не по значениям. Эти значения можно комбинировать операцией "логического ИЛИ", то есть для поиска системных и архивных файлов можно будет в параметр ***Attr*** передать:

```
(faSysFile | faArchive)
```

К использованию значений параметра ***Attr*** мы еще вернемся и рассмотрим их поподробнее, а сейчас поговорим о последнем параметре.

Последний параметр представляет собой ссылку на структуру ***TSearchRec***, объявленную также в Sysutils.hpp как

```
struct TSearchRec
  {
    int Time;
    int Size;
    int Attr;
    AnsiString Name;
    int ExcludeAttr;
    int FindHandle;
    _WIN32_FIND_DATA FindData;
  };
```

В данной структуре наиболее интересны следующие поля: ***Name***, которое содержит имя найденного файла с расширением; ***Size*** – размер найденного файла; ***Attr*** – его атрибуты (те же значения, что и у параметра ***Attr*** функции ***FindFirst***) и ***FindData***. Последний параметр позволяет нам получить просто море полезной информации о найденном файле, включая все Windows-атрибуты файла, получить которые нельзя через поле ***Attr*** структуры, время создания файла, время последнего доступа к нему и ряд других атрибутов. Отсылаем читателя к соответствующему разделу MSDN.

Таким образом, все, что касается функции ***FindFirst***, мы разобрали. Осталось уяснить назначение функций ***FindNext*** и ***FindClose*** – и можно приступать к написанию кода.

FindNext объявлена в модуле Sysutils.hpp как

```
extern PACKAGE int __fastcall FindNext(TSearchRec &F);
```

FindNext может использоваться только после вызова функции ***FindFirst***. Функция находит следующий файл с теми же параметрами, что были указаны при вызове ***FindFirst***, и в случае успешного поиска возвращается нуль, в случае неудачи – код ошибки Windows.

Функция ***FindClose*** объявлена там же, где и предыдущие функции, следующим образом:

```
extern PACKAGE void __fastcall FindClose(TSearchRec &F);
```

и ее единственное назначение – освобождение памяти, выделенной функцией ***FindFirst***. ***FindClose*** закрывает всю цепочку вызовов ***FindFirst/FindNext***, и ее надо обязательно вызывать после вызовов ***FindFirst/FindNext***.

Теперь, после столь мощной теоретической подготовки, можно писать код, который найдет нам все файлы в каталоге. Создайте в C++Builder новый проект, разместите на форме (***MainForm***) ***TListBox*** (***FileListBox***), ***TEdit*** для маски файлов (***MaskEdit***), для выбора директории разместите компонент ***TDirectoryListBox*** (***DirectoryListBox***) и две кнопки: ***StartButton*** и ***ExitButton***. Еще необходимо разместить ***TGroupBox*** (***AttributesGroupBox***) с флажками и ***TLabel***, так что будет лучше, если вы воспользуетесь уже созданным, работающим проектом с компакт-диска, приложенного к книге.

Приложение выглядит следующим образом.

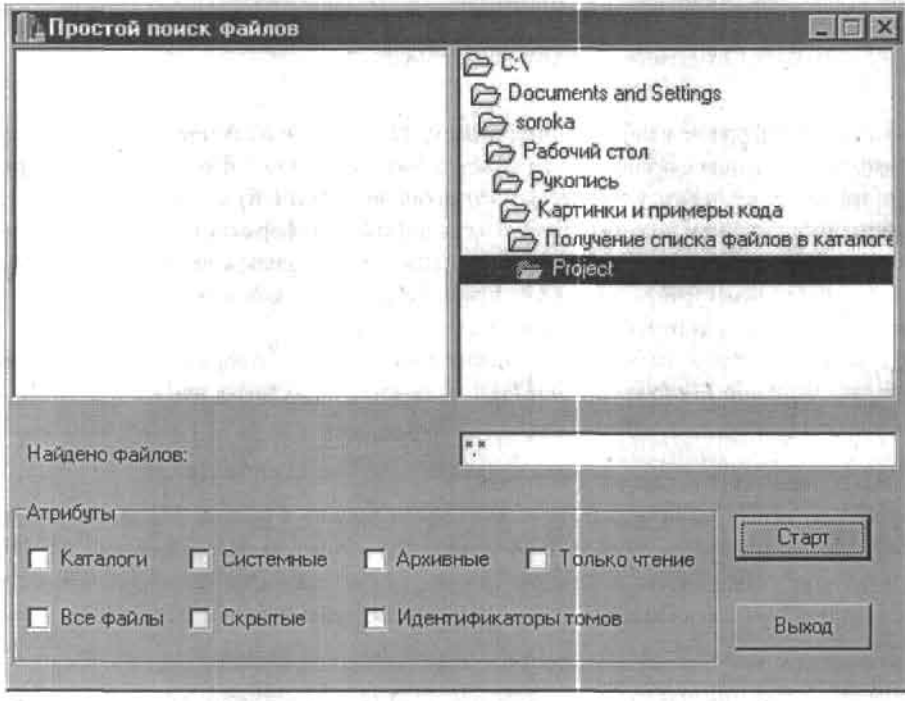


Рис. 3.52. *Внешний вид демонстрационного приложения*

Вот код, который "повешен" на обработчик события *OnClick* кнопки *StartButton*.

```
TSearchRec sr;
AnsiString PathForSearch;

int iAttributes = 0;

iAttributes |= faDirectory * DirectoriesCheckBox->Checked;
iAttributes |= faReadOnly * ReadOnlyCheckBox->Checked;
iAttributes |= faHidden * HiddenCheckBox->Checked;
iAttributes |= faVolumeID * VolumeIDCheckBox->Checked;
iAttributes |= faSysFile * SystemCheckBox->Checked;
iAttributes |= faArchive * ArchiveCheckBox->Checked;
iAttributes |= faAnyFile * AnyFilesCheckBox->Checked;

FileListBox->Items->Clear();

PathForSearch = DirectoryListBox->Directory + "\\" + MaskEdit->Text;

if(FindFirst(PathForSearch, iAttributes, sr) == 0)
  {
    do
      if((sr.Attr & iAttributes) == sr.Attr)
        FileListBox->Items->Add(sr.Name);
    while(FindNext(sr) == 0);

    FindClose(sr);
  }

FileCountLabel->Caption = AnsiString("Найдено ") +
IntToStr(FileListBox->Items->Count) + AnsiString(" файлов");
```

Как видно, размер кода на самом деле не соответствует размеру теории, изложенной выше. Но тем не менее некоторые комментарии по коду надо дать.

Строка

```
PathForSearch = DirectoryListBox->Directory + "\\" + MaskEdit->Text;
```

задает полный путь к искомым файлам.

Строка

```
iAttributes |= faDirectory * DirectoriesCheckBox->Checked;
```

и ей подобные добавляют в атрибуты файла, на соответствие которым ведется поиск, соответствующий атрибут. Данная строка включает поиск в директории.

Строка

```
if(FindFirst(PathForSearch, iAttributes, sr) == 0);
```

начинает собственно поиск, если есть хотя бы один файл, удовлетворяющий условию.

В цикле ***do ... while*** поиск файлов производится с помощью функции ***FindNext***.

Собственно проверка на совпадение атрибутов:

```
if((sr.Attr & iAttributes) == sr.Attr);
```

И обратите внимание на присутствие вызова ***FindClose(sr)*** по завершении поиска (когда ***FindNext(sr)*** возвращает не нуль, а код ошибки).

Вот и все. Осталось рассказать о некоторых подводных камнях данного подхода и на всякий случай дать тот же код поиска файлов по маске на чистом API (мы же пишем все-таки под Windows, и пользоваться API надо уметь).

Итак, подводный камень номер один: возьмем файл, у которого стоят атрибуты "только для чтения" и "архивный". В терминах VCL это означает установленные атрибуты ***faArchive*** и ***faReadOnly***. Теперь попробуйте поискать этот файл нашей тестовой программой, установив флажок "Архивные". Файл не будет найден. Попробуйте поискать этот же файл, установив флажок "Только чтение". Опять же файл найден не будет. И лишь установив эти два флажка, программа найдет данный файл.

Все дело в строке

```
if((sr.Attr & iAttributes) == sr.Attr);
```

которая задает точное, строгое соответствие атрибутов у искомых файлов. Именно эта строка при установке атрибута ***faReadOnly*** для поиска не позволит найти файлы, у которых кроме ***faReadOnly*** будет установлен еще какой-нибудь атрибут.

Достаточно изменить условие выбора на

```
if(sr.Attr & iAttributes);
```

и будут выбираться файлы, имеющие в числе прочих указанный атрибут, но, возможно, и любые другие атрибуты.

Второй подводный камень: в документации от Borland часто упоминается такое понятие, как "нормальные файлы" (normal files). Якобы указанные атрибуты позволяют искать файлы в дополнение к "нормальным", обычным файлам. И здравый смысл, и MSDN о таком понятии умалчивают, да вы и сами можете попробовать запустить наш проект, не указав ни одного атрибута в условии поиска. Ни один файл найден не будет. Так что этот кусок документации от Borland остается на их совести.

Теперь, собственно, код почти той же функциональности на WinAPI: данный код ищет все файлы (с маской *.*) в корневом каталоге диска C, без ограничения по атрибутам (то есть, по сути, как с установленным атрибутом ***faAnyFile*** в вышеприведенном коде).

```
WIN32_FIND_DATA FileData;
HANDLE hSearch;
BOOL fFinished = FALSE;

hSearch = FindFirstFile("C:\\*.*", &FileData);

while(!fFinished)
  {
    FileListBox->Items->Add(FileData.cFileName);
    if(!FindNextFile(hSearch, &FileData))
      if(GetLastError() == ERROR_NO_MORE_FILES)
        fFinished = TRUE;
  }

FindClose(hSearch);
FileCountLabel->Caption = FileListBox->Items->Count;
```

Этот код уже в комментариях не нуждается. Остается заметить, что код поиска файлов имеет смысл оформить в виде компонента. Для удобства будущего использования.

Рекурсивный поиск файлов

Мы уже знакомы с организацией поиска файлов в каталоге по маске и атрибутам, но иногда необходимо организовать поиск по группе вложенных директорий, и предложенное ранее решение для такой задачи не подходит. Однако его несложно переделать, для того чтобы ранее приведенный код искал файлы не только в текущей директории.

Чтобы понять, что нужно переделать в ранее приведенном коде для соответствия его новым реалиям, загрузите тестовый проект, относящийся к материалу "Получение списка файлов в каталоге", и посмотрите на код, который, собственно, искал файлы в текущей директории.

```
TSearchRec sr;
AnsiString PathForSearch;

int iAttributes = 0;

iAttributes |= faDirectory * DirectoriesCheckBox->Checked;
iAttributes |= faReadOnly * ReadOnlyCheckBox->Checked;
iAttributes |= faHidden * HiddenCheckBox->Checked;
iAttributes |= faVolumeID * VolumeIDCheckBox->Checked;
iAttributes |= faSysFile * SystemCheckBox->Checked;
iAttributes |= faArchive * ArchiveCheckBox->Checked;
iAttributes |= faAnyFile * AnyFilesCheckBox->Checked;

FileListBox->Items->Clear();

PathForSearch = DirectoryListBox->Directory + "\\" + MaskEdit->Text;

if(FindFirst(PathForSearch, iAttributes, sr) == 0)
  {
    do
      if((sr.Attr & iAttributes) == sr.Attr)
        FileListBox->Items->Add(sr.Name);
    while(FindNext(sr) == 0);

    FindClose(sr);
  }
```

Путь поиска и маска файла определялись значением ***PathForSearch***, а атрибуты поиска – значением ***iAttributes***. Результат поиска заносился в структуру ***sr***.

Найдя все файлы в текущей директории, цикл ***do ... while*** завершался. Для того чтобы поиск происходил в папках, лежащих уровнями ниже, необходимо изменять путь поиска, добавляя в него найденную директорию, и повторять поиск уже для нее. И так по всем встреченным на всех уровнях директориям. Иначе говоря, код поиска необходимо вызывать рекурсивно.

Чтобы оформить рекурсивный вызов кода, для начала давайте упростим задачу, чтобы сосредоточиться на наиболее существенном и не отвлекаться на детали. Будем искать только файлы, директории в поиск включать не будем. Поиск будем осуществлять по всем файлам, вне зависимости от их атрибутов. Иначе говоря, в качестве атрибутов поиска укажем ***faAnyFile***. Упрощенная основа кода для рекурсивного поиска будет выглядеть следующим образом.

```
TSearchRec sr;
AnsiString PathForSearch;

FileListBox->Items->Clear();

PathForSearch = ExcludeTrailingBackslash(DirectoryListBox->Directory) + "\\";

if(FindFirst(PathForSearch + "*.*", faAnyFile, sr) == 0)
   {
      do
         if((sr.Attr & faDirectory) != faDirectory)
               {
               if(((sr.Attr & faAnyFile) == sr.Attr) && (MatchesMask(sr.Name,
                     MaskEdit->Text)))
                     {
                        FileListBox->Items->Add(sr.Name);
                     }
                  // конец оператора if(((sr.Attr & AAttributes) == sr.Attr) &&
                     // MatchesMask(sr.Name, MaskEdit->Text)))
               }
            else
               {
                  if(sr.Name != "." && sr.Name!= "..")
                     {
                        // здесь необходимо спуститься на уровень ниже
                           и осуществить
                           // поиск там
                     }
               }
            // конец оператора if((sr.Attr & faDirectory) != faDirectory)
```

```
        while(FindNext(sr) == 0);

        FindClose(sr);
    }
// конец оператора if(FindFirst(PathForSearch + "*.*", faAnyFile, sr) == 0)
```

Во-первых, обратите внимание, что мы разделили путь поиска и маску (имя) файлов. Маска теперь задается непосредственно значением ***MaskEdit->Text***. Так нам впоследствии будет намного удобнее. Второе, на что надо обратить внимание: в функциях ***FindFirst/FindNext*** ищем теперь файлы не по указанной маске, а по маске "*.*". Соответствие маске, определяемой значением ***MaskEdit->Text***, проверяет функция ***MatchesMask***. Данная функция объявлена в модуле Masks.hpp следующим образом.

```
extern PACKAGE bool __fastcall MatchesMask(const AnsiString Filename,
const AnsiString Mask);
```

MatchesMask проверяет файлы на соответствие указанной маске. Если имя файла соответствует маске, то возвращается ***true***, иначе – ***false***.

Почему мы ищем файлы не сразу по указанной маске (например, "*.doc")? Дело в том, что возможна такая ситуация: в текущем каталоге файлов с указанной маской нет, но они есть в нижележащих каталогах. Осуществляя поиск не по "*.*", мы не сможем добраться до нижележащих каталогов с файлами: ***FindFirst*** определит, что в текущем каталоге файлов с заданной маской нет, и дальнейший поиск осуществляться не будет.

Теперь самое главное. Оператор

```
if((sr.Attr & faDirectory) != faDirectory)
   {
      if(((sr.Attr & faAnyFile) == sr.Attr) && (MatchesMask(sr.Name,
      MaskEdit->Text)))
         {
            FileListBox->Items->Add(sr.Name);
         }
      // конец оператора if(((sr.Attr & AAttributes) == sr.Attr) &&
       (MatchesMask(sr.Name, MaskEdit->Text)))
   }
else
   {
      if(sr.Name != "." && sr.Name!= "..")
         {
```

```
                // здесь необходимо спуститься на уровень ниже и осуществить
                // поиск там
            }
        }
    // конец оператора if((sr.Attr & faDirectory) != faDirectory)
```

Если файл не является директорией, то мы заносим его в список. Если же файл является директорией и не является служебными директориями с именами "." и "..", то нам необходимо добавить имя этой директории к ***PathForSearch*** и начать поиск уже по новому пути... Но в данном случае ничто, кроме оператора goto, не вернет нас к началу поиска... Что же делать? Можно (и разумно) оформить поиск в текущей директории как функцию и вызывать ее рекурсивно вот в этом месте:

```
if(sr.Name != "." && sr.Name!= "..")
  {
     // здесь необходимо спуститься на уровень ниже и осуществить поиск там
  }
```

Такая функция приведена в тестовом проекте к данному материалу. Загрузите тестовый проект, вид которого приведен ниже.

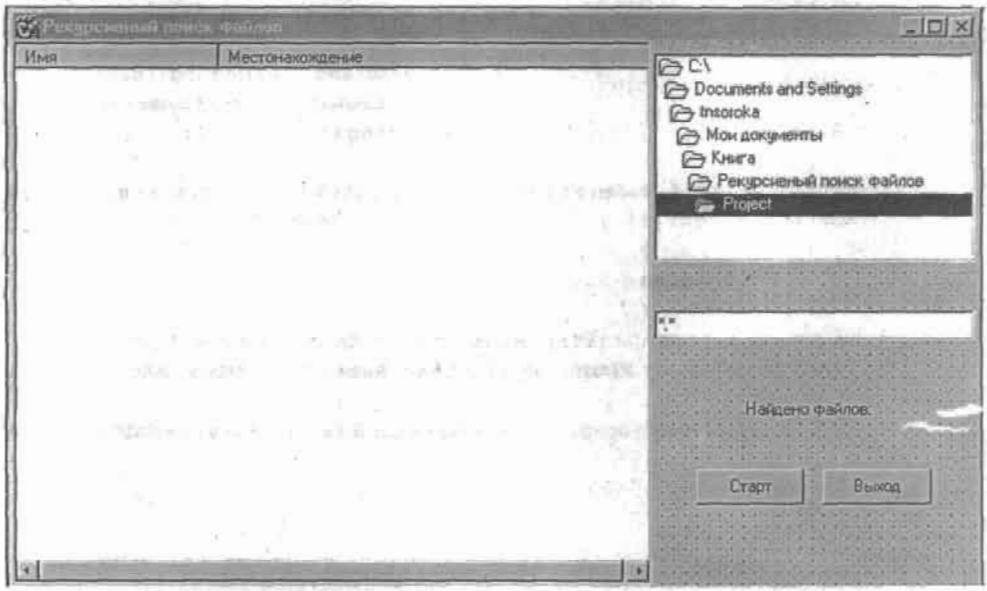


Рис. 3.53. *Внешний вид тестового проекта*

Поиск файлов, оформленный в функцию, выглядит следующим образом.

```
//---------------------------------------------------------------------
void __fastcall TMainForm::FindFiles(AnsiString APath,
 const AnsiString AMask, int AAttributes)
   {
      TSearchRec sr;

      if(FindFirst(APath + "*.*", AAttributes, sr) == 0)
         {
            do
               {
                  if((sr.Attr & faDirectory) != faDirectory)
                     {
                        if(((sr.Attr & AAttributes) == sr.Attr) &&
                        (MatchesMask(sr.Name, AMask)))
                           {
                              TListItem *NewItem;
                              NewItem = FileListView->Items->Add();
                              NewItem->Caption = sr.Name;
                              NewItem->SubItems->Add(APath);

                        FileCountLabel->Caption=AnsiString("Найдено ")+
                              IntToStr(FileListView->Items->Count)
                               + AnsiString(" файлов");
                           }
                     // конец оператора if(((sr.Attr&AAttributes)==sr.Attr)&&
                        // (MatchesMask(sr.Name, AMask)))
                     }
                  else
                     {
                        if(sr.Name != "." && sr.Name!= "..")
                     FindFiles(APath+sr.Name+"\\", AMask, AAttributes);
                     }
               // конец оператора if((sr.Attr&faDirectory) != faDirectory)
```

```
                Application->ProcessMessages();
              }
            while(FindNext(sr) == 0);

            FindClose(sr);
          }
    }
//---------------------------------------------------------------------------
```

Вот, собственно, та строчка, в которой происходит рекурсивный вызов функции.

```
if(sr.Name != "." && sr.Name!= "..")
  FindFiles(APath + sr.Name + "\\", AMask, AAttributes);
```

Если мы в текущем каталоге нашли директорию и ее имя не "." и "..", то формируем новый путь поиска, прибавляя к текущему пути имя этой директории, и начинаем поиск в ней. И так до тех пор, пока мы не обойдем все нижележащие директории. Если в качестве начальной директории указать имя диска "C:\", например, и задать для поиска маску "*.*", то данный код найдет все файлы на диске "C:\".

Поскольку поиск уже ведется не в одной директории, то вместо простого списка (***TListBox***) мы используем расширенный список (***TListView***), в одной колонке которого отображаем имя файла, а в другой – путь к нему.

Строка

```
Application->ProcessMessages();
```

нужна для того, чтобы во время поиска приложение реагировало на действия пользователя.

Результаты работы программы приведены на рис. 3.54.

У данного способа поиска файлов есть недостаток: он выполняется в основном потоке программы, и ничего другого во время поиска ваша программа делать уже не сможет. Для того чтобы устранить данный недостаток, необходимо выполнять поиск в отдельном потоке.

Плюс у нашего кода есть еще один недостаток: поиск невозможно прервать по требованию пользователя. Но так как это совсем несложно сделать, думаю, вы справитесь с этим самостоятельно.

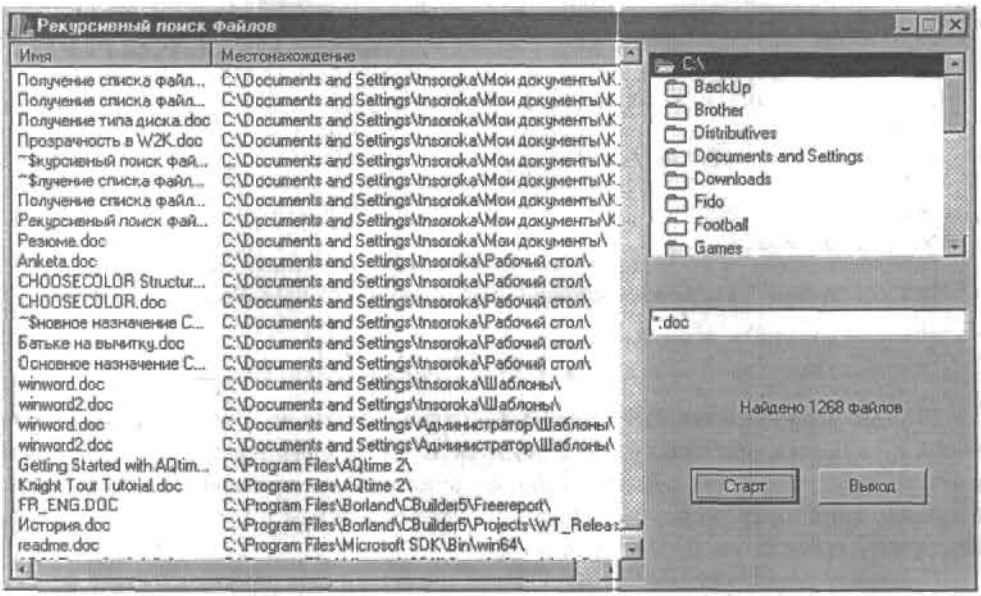


Рис. 3.54. *Список всех *.doc-файлов на диске C*

Содержание

ИЗДАТЕЛЬСТВО
«КУДИЦ-ОБРАЗ»

Тел.: (095) 333-82-11; ok@kudits.ru, http://www.kudits.ru/publish

КНИГИ В ПРОДАЖЕ

ArchiCAD 8.0. Справочник с примерами. Титов С.
480 с. 2003 г. Опт. цена 209 р.

ArchiCAD 7.0. Титов С.
400 с. 2003 г. Опт. цена 154 р.

ArchiCAD: полезные рецепты. Комплект. Титов С.
272 с. 2003 г. Опт. цена 171,6 р.

AutoCAD 2002/2002 LT/2000. Справочник команд. Россоловский А.
720 с. 2002 г. Опт. цена 220 р.

1С: практика настройки оперативного учета. Ражиков М.Ю.
256 с. 2003 г. Опт. цена 121 р.

AES – стандарт криптографической защиты. Конечные поля. Зензин О.С., Иванов М.А.
176 с. 2002 г. Опт. цена 55 р.

C++ & Visual Studio. NET. Самоучитель программиста. Баженова И.Ю.
448 с., 2003 г. Опт.цена 132 р.

C#. Визуальное проектирование приложений. Фролов А., Фролов Г.
512 с. 2003 г. Опт.цена 220 р.

Delphi 7. Самоучитель. Климова Л.М.
480 с. 2004 г. Опт.цена 124,3 р.

Delphi 7. Самоучитель программиста. Баженова И.Ю.
432 с. 2002 г. Опт.цена 110 р.

Dreamweaver MX. Базовый курс Божко А.Н.

Flash & XML. Руководство разработчика. Джекобсон Д. Пер. с англ.
352 с. 2003 г. Опт.цена 111 р.

IT-безопасность. Стоит ли рисковать корпорацией? Маккарти Л. Пер. с англ.
208 с. 2004 г. Опт.цена 110 р.

Jamagic: программирование игр и симуляторов. Перес Серхио. Пер. с англ.
288 с. 2004 г. Опт.цена 132 р.

Java: основы Web-служб. Тост Андре. Пер. с англ.
464 с. 2004 г. Опт.цена 143 р.

Linux: создание виртуальных частных сетей (VPN). Колесников О., Хетч Брайан
464 с. 2004 г. Опт.цена 154 р.

ИЗДАТЕЛЬСТВО
«КУДИЦ-ОБРАЗ»

Тел.: (095) 333-82-11; ok@kudits.ru, http://www.kudits.ru/publish

КНИГИ В ПРОДАЖЕ

.Net Framework: Библиотека классов. Темплман Дж., Виттер Д.
672 с. 2003 г. Опт.цена 298,1 р.

Office XP. Афанасьев Д., Баричев С., Плотников О.
356 с. 2002 г. Опт.цена 84,7 р.

PageMaker 6.5/7.0. Самоучитель. Вовк Е.Т.
352 с. 2002 г. Опт.цена 121 р.

Pascal 7.0. Основы практического программирования. Решение типовых задач. Климова Л.М.
528 с. 2000 г. Опт.цена 105,6 р.

Photoshop CS: технология работы. Сканирование, ретушь. Божко А.Н.
624 с. 2004 г. Опт.цена 176 р.

Sendmail: настройка и оптимизация. Кристенсон Ник. Пер. с англ.
272 с. 2004 г. Опт.цена 110 р.

QuarkXPress 5.0. Самоучитель. Вовк Е.Т.
288 с. 2002 г. Опт.цена 88 р.

Visual Basic.NET, Visual Basic 6.0, Visual Basic for Applications 6.0. Король В.И.
496 с. 2002 г. Опт.цена 143 р.

Web-дизайн: Photoshop & Dreamweaver. 3 ключевых этапа. Смит Колин. Пер. с англ.
264 с. 2004 г. Опт.цена 99 р.

Windows XP Professional. Проффит Б.
416 с. 2002 г. Опт.цена 112,2 р.

Администрирование баз данных. Крейг С.Маллинс
752 с. 2003 г. Опт.цена 253 р.

Ассемблер в задачах защиты информации. Иванов М.А.
320 с. 2002 г. Опт.цена 96,8 р.

Безопасность: технологии, средства, услуги. Барсуков В.С.
496 с. 2001 г. Опт.цена 99 р.

Брендинг – Дорога к мировому рынку. Анхолд Симон. Пер. с англ.
272 с. 2004 г. Опт.цена 99 р.

Информационная архитектура. Чертежи для сайта. Уодтке К. Пер. с англ.
320 с. 2004 г. Опт.цена 110 р.

Искусство дизайна с компьютером и без... Пер. с англ.
208 с. 2004 г. Опт.цена 88 р.

ИЗДАТЕЛЬСТВО
«КУДИЦ-ОБРАЗ»

Тел.: (095) 333-82-11; ok@kudits.ru, http://www.kudits.ru/publish

КНИГИ В ПРОДАЖЕ

Исследуем Maya 4: 30 уроков в 3D. Шонхер М. Пер. с англ.
288 с. 2002 г. Опт.цена 99 р.

Как преподнести себя на рынке труда. Хангерленд Бафф. Пер. с англ.
224 с. 2003 г. Опт.цена 67,10 р.

Как успешно руководить проектами. Серебряная пуля. Фергус О'Коннелл
288 с., 2003 г. Опт.цена 121 р.

Коммутаторы CISCO. Одом Ш., Ноттингем Х.
528 с., 2003г. Опт.цена 231 р.

Компьютерная анимация. Рик Пэрент. Пер. с англ.
560 с., 2004г. Опт.цена 242 р.

Компьютерные игры: секреты бизнеса. Пер. с англ.
416 с. 2004 г. Опт.цена 169,4 р.

Компьютерные презентации: от риторики до слайд-шоу. Елизаветина Т.М. Пер. с англ.
240 с. 2004 г. Опт.цена 77 р.

Лечение псориаза – естественный путь. Пегано Дж. Пер. с англ.
288 с. 2001 г. Опт.цена 132 р.

Маршрутизаторы CISCO для IP-сетей. Руденко И., Tsunami Computing.
656 с. 2003 г. Опт.цена 242 р.

Мир InterBase. Архитектура, администрирование и разработка приложений баз данных в InterBase/FireBird/Yaffil. Изд-е 2-е, дополн. Ковязин А., Востриков С.
496 с. 2003 г. Опт.цена 220 р.

Наука отладки. Тэллес М., Хсих Ю.
560 с. 2003 г. Опт.цена 187 р.

Объектно-ориентированное программирование на ActionScript. Пер. с англ.
416 с. 2003 г. Опт.цена 125,4 р.

Основы пространственных баз данных. Шаши Ш., Санжей Ч. Пер. с англ.
336 с. 2004 г. Опт.цена 121 р.

Персональная защита от хакеров. Руководство для начинающих. Форд Дж.
272 с. 2002 г. Опт.цена 92,4 р.

Платформа программирования J2ME для портативных устройств. Пирумян В.
352 с. 2003 г. Опт.цена 132 р.

Популярные Web-приложения на Flash MX. Чанг Тим К., Кларк Шон
272 с. 2003 г. Опт.цена 129,8 р.