

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

**Методические указания**  
по выполнению лабораторных работ по дисциплине  
**«РАЗРАБОТКА ЗАЩИЩЕННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ»**  
для студентов специальности 10.05.01 Компьютерная безопасность  
специализация «Информационно-аналитическая и техническая экспертиза  
компьютерных систем»

**Ставрополь**  
**2026**

## Содержание

|                                                              |           |
|--------------------------------------------------------------|-----------|
| <b>Содержание .....</b>                                      | <b>2</b>  |
| <b>1. Приложение для Windows .....</b>                       | <b>4</b>  |
| Дескрипторы или особенность доступа к объектам Windows ..... | 4         |
| Описание класса главного окна .....                          | 5         |
| Регистрация класса окна .....                                | 6         |
| Создание экземпляра окна .....                               | 6         |
| Отображение окна .....                                       | 7         |
| Обновление заданной области окна .....                       | 8         |
| Организация цикла обработки сообщений .....                  | 8         |
| Сообщение WM_PAINT .....                                     | 10        |
| Оконная процедура .....                                      | 10        |
| Отправка сообщений .....                                     | 10        |
| Листинг программы .....                                      | 11        |
| Дочерние окна .....                                          | 13        |
| Субклассирование .....                                       | 14        |
| <b>2. Процессы .....</b>                                     | <b>17</b> |
| Управление процессом .....                                   | 17        |
| Доступ к процессу .....                                      | 19        |
| Приоритет процесса .....                                     | 20        |
| Завершение процесса .....                                    | 21        |
| Запуск дочерних процессов .....                              | 21        |
| Сбор информации о процессах Windows .....                    | 22        |
| Время выполнения процесса .....                              | 24        |
| Получение сведений о версии ОС .....                         | 25        |
| <b>3. Многопоточные приложения .....</b>                     | <b>27</b> |
| Поток <i>TThread</i> .....                                   | 27        |
| Пример многопоточного приложения .....                       | 32        |
| Синхронизация потоков .....                                  | 34        |
| Библиотека параллельных вычислений .....                     | 41        |
| Распараллеливание, класс <i>TParallel</i> .....              | 41        |
| Класс-задача, <i>TTask</i> .....                             | 45        |
| Блокирование операций, класс <i>TInterlocked</i> .....       | 48        |
| <b>4. Динамически подключаемые библиотеки .....</b>          | <b>49</b> |
| Создание проекта DLL .....                                   | 49        |
| Объявление и экспорт функций в DLL .....                     | 50        |
| Пример экспорта функций .....                                | 51        |
| Пример хранения форм в библиотеке .....                      | 51        |
| Вызов библиотеки из приложения .....                         | 53        |
| Неявное подключение DLL .....                                | 53        |
| Явное подключение DLL .....                                  | 55        |
| <b>5. Взаимодействие процессов .....</b>                     | <b>57</b> |
| Обмен данными через буфер обмена .....                       | 57        |
| Регистрация пользовательского формата буфера обмена .....    | 60        |
| Обмен сообщениями .....                                      | 63        |
| Поиск окна .....                                             | 64        |
| Регистрация пользовательских сообщений .....                 | 65        |
| Пример обмена сообщениями между процессами .....             | 66        |
| Файловое отображение .....                                   | 70        |
| <b>6. Взаимодействие в сети .....</b>                        | <b>74</b> |
| Почтовые слоты .....                                         | 75        |
| Именованные каналы .....                                     | 79        |
| Сокеты .....                                                 | 89        |

|                                                                 |            |
|-----------------------------------------------------------------|------------|
| <b>7. Сервисы и апплеты .....</b>                               | <b>96</b>  |
| Менеджер управления сервисами .....                             | 96         |
| Управление сервисом .....                                       | 99         |
| Состояние службы .....                                          | 100        |
| Конфигурирование службы .....                                   | 100        |
| Удаление службы .....                                           | 101        |
| Сервис в VCL, класс TService .....                              | 101        |
| Идентификация .....                                             | 101        |
| Тип сервиса .....                                               | 101        |
| Определение прав на управление сервисом .....                   | 102        |
| Загрузка и запуск службы .....                                  | 102        |
| Статус службы .....                                             | 103        |
| Остановка и возобновление службы .....                          | 104        |
| Инсталляция и деинсталляция сервиса .....                       | 104        |
| Выполнение службы, поток <i>TServiceThread</i> .....            | 105        |
| Ведение протокола службы .....                                  | 105        |
| Приложение-сервис TServiceApplication .....                     | 106        |
| Пример .....                                                    | 107        |
| Регистрация сервиса в ручном режиме .....                       | 109        |
| Апплеты Панели управления .....                                 | 110        |
| Апплет Панели управления, класс <i>TAppletModule</i> .....      | 111        |
| Приложение Панели управления <i>TAppletApplication</i> .....    | 112        |
| Пример апплета управления сервисом Windows .....                | 112        |
| <b>8. Многокомпонентная модель COM .....</b>                    | <b>116</b> |
| COM-объект .....                                                | 116        |
| Понятие интерфейса .....                                        | 117        |
| Базовый интерфейс <i>IUnknown</i> .....                         | 118        |
| Реализация интерфейса .....                                     | 119        |
| Порядок вызова сервера COM .....                                | 120        |
| Интерфейс <i>IClassFactory</i> и библиотека COM .....           | 121        |
| Реализация фабрики класса, класс <i>TComObjectFactory</i> ..... | 123        |
| Реализация COM-объекта в Delphi .....                           | 125        |
| Класс <i>TComObject</i> .....                                   | 126        |
| Класс <i>TTypedComObject</i> .....                              | 126        |
| Класс TComServer .....                                          | 126        |
| Пример COM-проекта .....                                        | 128        |
| COM-сервер .....                                                | 128        |
| COM-клиент .....                                                | 137        |
| Обращение к COM-объекту .....                                   | 138        |
| <b>9. Графика .....</b>                                         | <b>140</b> |
| Представление цвета .....                                       | 140        |
| Кисть TBrush .....                                              | 142        |
| Перо TPen .....                                                 | 144        |
| Шрифт TFont .....                                               | 147        |
| Холст TCanvas в VCL .....                                       | 150        |
| Закраска области .....                                          | 151        |
| Градиентная заливка .....                                       | 152        |
| Графические примитивы .....                                     | 152        |
| Копирование части холста .....                                  | 157        |
| Сложные графические задачи .....                                | 158        |
| Растровые операции .....                                        | 158        |
| Управление прозрачностью .....                                  | 161        |
| Системы координат и режимы отображения .....                    | 163        |
| <b>Список использованной литературы .....</b>                   | <b>172</b> |

# 1. Приложение для Windows

*Вид занятия – практическая работа*

*Цель – исследование порядка создания и особенностей функционирования приложения для ОС Windows*

*Продолжительность – 2 часа*

На протяжении всего периода работы приложения под управлением Windows оно находится под неусыпным контролем со стороны операционной системы. Для этого с момента создания до момента разрушения **главного окна** приложения операционная система поддерживает с ним активный диалог. Беседа ведётся посредством отправки сообщений оконной процедуре (*window procedure*) главного окна приложения. Что понимается под словосочетанием “отправить сообщение”? Это значит, что операционная система осуществит запись о каком-то событии в область памяти доступную оконной процедуре. Указанная область памяти называется **очередью программы** или **очередью сообщений**. Очередь может хранить несколько сообщений, для этого она устроена по принципу стека первым пришёл – первым вышел (*FIFO, First Input First Output*). Программа также в праве отправить сообщение Windows. В этом случае сообщение от программы помещается в другую, общесистемную очередь Windows. Заметьте, что общесистемная очередь существует в одном единственном экземпляре и в неё помещаются сообщения от всех окон.

Для того, чтобы приложение не прозевало предназначенное ей послание оно периодически просматривает свой «почтовый ящик» на предмет новых поступлений. Для этого в любой программе (кстати, и в самой Windows) организуется **цикл обработки сообщений** (*message loop*). И если по какой-то причине у приложения остановится этот цикл, то оно безнадежно “зависнет”, а если остановится цикл операционной системы, то нас спасёт только перезагрузка.

Обозначим этапы, через которые надо пройти программисту для того, чтобы создать простейшее жизнеспособное приложение для Windows. А для этого необходимо:

- Описать макет будущего главного окна приложения.
- Зарегистрировать этот класс в Windows.
- Создать экземпляр главного окна приложения.
- Отобразить окно на экране компьютера.
- Организовать цикл обработки сообщений Windows.
- Описать оконную процедуру.

## **Дескрипторы или особенность доступа к объектам Windows**

При программировании на Win API мы будем пользоваться **дескрипторами** объектов. Для начинающего программиста, работающего исключительно с компонентами среды программирования Delphi это не вполне понятный термин.

Операционная система Windows тщательно охраняет свои ключевые объекты, она не допускает получение доступа к ним непосредственного доступа. При создании объекта ему назначается уникальное целочисленное значение – его идентификатор. Работа с такими объектами возможна только при передаче указателя на них в соответствующие функции. В терминах Windows такие указатели называются дескрипторами. Хотя физически дескриптор представляет собой обычное целое число, но для того, чтобы исходный код программы стал более читабельным установлен ряд правил об именах дескрипторов. Название типа дескриптора всегда начинается с символа «H» (первый символ английского слова *handle*). Опять же по существующим соглашениям для ключевых дескрипторов закреплены постоянные имена, так, например, при чтении документации SDK можно быть твёрдо уверенным, что:

```
Instance:HINSTANCE; //дескриптор экземпляра приложения
Wnd: HWND;          //дескриптор окна
DC:HDC;              //дескриптор контекста устройства
Brush:HBRUSH;        //дескриптор кисти
Pen:HPEN;            //дескриптор пера
Font:HFONT;          //дескриптор шрифта
GDIObj:HGDIOBJ;      //дескриптор GDI объекта
```

Это далеко не полный перечень всех существующих дескрипторов, но думаю, что вы уже заметили созвучие их с именами классов из комплекта VCL Delphi, и, столкнувшись с новым названием дескриптора, легко ассоциируете его со знакомым объектом.

## Описание класса главного окна

В классическом приложении для Windows всегда существует главное окно, в котором осуществляется обработка сообщений поступающих приложению. Главное окно всегда создаётся на основе специальной структуры называемой **классом окна**. В заголовочных файлах C++ из пакета Microsoft Visual Studio (и соответственно в SDK) упомянутая структура называется `WNDCLASSEX`. В Delphi имеется аналог указанной структуры, это запись `TWndClass`. Запись содержит десять полей:

```
type TWndClass = packed record
    style: UINT;                //стиль (комбинация стилей) окна
    lpfnWndProc: TFWndProc;    //адрес оконной процедуры
    cbClsExtra: Integer;       //выделенная память
    cbWndExtra: Integer;       //выделенная память
    hInstance: HINST;          //дескриптор экземпляра программы
    hIcon: HICON;               //дескриптор иконки окна
    hCursor: HCURSOR;           //курсор окна
    hbrBackground: HBRUSH;     //кисть, закрашивающая окно
    lpszMenuName: PAnsiChar;   //имя меню
    lpszClassName: PAnsiChar;  //имя класса окна
end;
```

Стиль создаваемого окна определяется **первым** по счёту полем структуры `Style`. Сюда может быть передано одно значение или комбинация значений, определяющих особенности внешнего вида и поведения окна. Так как большинство стилей окна оказывают весьма существенное влияние на порядок вывода изображений рекомендую изучить таблицу 1.1.

Таблица 1.1. Стили окна, поле `style` структуры `TWndClass`

| Стиль                           | Описание                                                                                                                                                                                              |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>CS_BYTEALIGNCLIENT</code> | Автоматическое выравнивание клиентской области окна по горизонтали по границе байта. Этот флаг позволяет несколько повысить производительность вывода изображения.                                    |
| <code>CS_BYTEALIGNWINDOW</code> | Тоже самое, что и <code>CS_BYTEALIGNCLIENT</code> , только теперь речь идёт обо всём окне.                                                                                                            |
| <code>CS_CLASSDC</code>         | Позволяет сформировать единый стиль для вывода в многооконном приложении, так как при создании нескольких окон одного класса они станут разделять один единственный контекст графического устройства. |
| <code>CS_DBLCLKS</code>         | Позволяет окну реагировать на двойной щелчок.                                                                                                                                                         |
| <code>CS_DROPShadow</code>      | Стиль появился в Windows XP. При перетаскивании окна включается эффект тени.                                                                                                                          |
| <code>CS_GLOBALCLASS</code>     | Создаваемый класс окна станет доступен всем приложениям. Этот флаг часто применяется при хранении окон в динамических библиотеках.                                                                    |
| <code>CS_HREDRAW</code>         | При изменении горизонтального размера окна автоматически поступает команда на перерисовку этого окна.                                                                                                 |
| <code>CS_NOCLOSE</code>         | Отключается системная кнопка закрытия окна в его заголовке.                                                                                                                                           |
| <code>CS_OWNDC</code>           | Для каждого из окон одного класса создаётся уникальный контекст графического устройства.                                                                                                              |
| <code>CS_PARENTDC</code>        | Каждое дочернее окно создано на базе этого класса будет использовать контекст устройства родительского окна.                                                                                          |
| <code>CS_SAVEBITS</code>        | Позволяет быстро восстанавливать часть окна скрытую под другими окнами. Для этого затенённая область сохраняется в виде растрового образа и копируется в окно в момент его перерисовки.               |
| <code>CS_VREDRAW</code>         | При изменении вертикального размера окна автоматически поступает команда на перерисовку этого окна.                                                                                                   |

Особо подчеркну то, что если программист рассчитывает использовать комбинацию стилей, то между константами стиля необходимо ставить оператор `OR` (логическое “ИЛИ”). Если приложение не будет обладать дочерними окнами то, в самом общем случае стилем окна

объявляют комбинацию: CS\_VREDRAW и CS\_HREDRAW. Это значит, что окно должно перерисовываться при изменении вертикального или горизонтального размера.

Возвращаемся к структуре TWndClass. Её **второе** поле (lpfnWndProc) хранит адрес оконной процедуры, в которой описывается реакция приложения на поступающие извне сообщения. **Третье и четвёртое** поля определяют количество байт памяти дополнительно выделяемых для хранения структуры окна и экземпляра окна в памяти. Всегда устанавливайте эти поля в ноль, тогда задачу распределения памяти решит Windows. **Пятое** поле hInstance желает знать какое приложение готовит к регистрации класс окна. **Шестое, седьмое и восьмое** поля соответственно хранят дескрипторы иконки, курсора и кисти (заливающей клиентскую часть) окна. **Девятое** поле lpszMenuName задаёт меню класса окна. И последнее, **десятое** поле содержит имя класса.

В случае построения простейшего главного окна приложения потребуется следующее описание класса:

```
with WindowClass do
begin
  Style := CS_HREDRAW or CS_VREDRAW; {Стиль окна класса}
  lpfnWndProc := @WindowProc;        {Указатель на оконную процедуру}
  cbClsExtra := 0; {Выделенная память, самостоятельно используемая программой}
  cbWndExtra := 0; {Выделенная память, самостоятельно используемая программой}
  hInstance := 0; {Ссылка на экземпляр программы}
  hIcon := LoadIcon (0, idi_Application); {Дескриптор маленькой иконки окна}
  hCursor := LoadCursor (0, idc_Arrow); {Дескриптор курсора}
  hbrBackground := GetStockObject(White_Brush); {Дескриптор кисти окна}
  lpszMenuName := ''; {Ссылка на строку имени меню}
  lpszClassName := AppName; {Имя класса}
end;
```

---

В Windows существует достаточно большой спектр готовых описаний классов окон. Это кнопки, переключатели, строки ввода и другие стандартные элементы управления.

---

## Регистрация класса окна

Справившись со всеми десятью полями класса окна, переходим к следующему этапу – регистрации класса будущего окна в Windows. В качестве крупного специалиста по данному вопросу рекомендую функцию:

```
Function RegisterClass(WndClass:TWndClass):Integer;
```

В единственный параметр функции направляется заполненная ранее структура TWndClass. В случае успеха функция возвращает уникальный идентификатор зарегистрированной структуры (в терминах Windows – атомарное значение, атом). Если регистрация завершилась неудачей, то получаем ноль. На такое неуважительное к себе отношение программа просто обязана отреагировать, например следующим образом:

```
If RegisterClass(WindowClass) = 0 then Halt;
```

Если попытка регистрации нового класса завершается неудачно, то выполнение программы прекращается – выполняется суровая директива Halt(), моментально останавливающая приложение.

---

Если на момент обращения к функции RegisterClass() в системе уже был зарегистрирован (например, другим приложением) идентичный класс окна, то функция возвратит атомарное значение оставшееся после предыдущей регистрации.

---

## Создание экземпляра окна

Факт успешной регистрации класса окна говорит о том, что операционная система не против того, чтобы на основе зарегистрированного класса был создан экземпляр окна. Теперь наступает черёд вводить в бой тяжёлую кавалерию – очередную системную функцию Windows:

```
Function CreateWindow(
  ClassName : PChar; // имя зарегистрированного класса
  lpWindowName : PChar; // заголовок окна
  Style : Longword; // стиль окна
  x : Integer; // горизонтальная позиция окна
```

```

y : Integer;           // вертикальная позиция окна
Width : Integer;       // ширина окна
Height : Integer;      // высота окна
WndParent : HWND;      // дескриптор родительского окна
Menu : HMENU;          // дескриптор меню окна
Instance : HINSTANCE;  // дескриптор экземпляра приложения
lpParam : pointer      // указатель на дополнительную информацию
) : HWND;

```

Самый первый параметр `ClassName` представляет собой имя только что зарегистрированного класса окна. В Windows предусмотрен целый ряд заранее описанных заготовок классов. Так для создания стандартной кнопки имя класса “`BUTTON`”, для создания комбинированного списка – “`COMBOBOX`”, строки ввода – “`EDIT`”, метки текста – “`STATIC`”, и т.д. В нашем случае мы намерены создать эксклюзивное главное окно приложения на основе описанного и зарегистрированного именно нами класса окна. Поэтому значение параметра `ClassName` должно совпадать со значением последнего поля в классе окна (структуре `TWndClass`). Вторым параметром соответствует тексту заголовка окна. Индивидуальные характеристики или, как чаще говорят стиль создаваемого экземпляра окна, следует указывать в третьем параметре `Style`. Перечень стилей настолько богат, что я лучше вновь отправлю вас к справочной системе. Только отмечу, что для описания стандартного окна вполне приемлемо значение `WS_OVERLAPPEDWINDOW`. Это комбинация стилей: `WS_OVERLAPPED`, `WS_CAPTION`, `WS_SYSMENU`, `WS_THICKFRAME`, `WS_MINIMIZEBOX` и `WS_MAXIMIZEBOX` указывающая функции на то, что мы рассчитываем получить окно с заголовком, рамкой и системными кнопками в правой части заголовка. Следующие четыре аргумента связаны с местом вывода и размерами будущего окна и поэтому в особых комментариях не нуждаются, а вот восьмой параметр `WndParent` весьма важен. Это дескриптор родительского окна или элемента управления. У главного окна программы он нулевой. Но если вы создаёте дочерние окна (не забывайте, что и кнопка, панель, строка ввода и много чего другого также являются окнами), то в этом параметре следует указывать ссылку на окно-владелец. Следующий параметр `Instance` является дескриптором приложения, в котором создаётся окно. Это глобальная переменная, за её объявление отвечает среда программирования. Последний необязательный параметр `lpParam` это указатель, на дополнительные данные окна. При удачном выполнении функция возвратит уникальный дескриптор окна созданного окна.

---

Для работы с дескриптором приложения не стоит создавать свою собственную переменную `HInstance`: `LongWord`, так как она уже объявлена в модуле `SysInit` среды Delphi.

---

```

var Wnd : HWND;
...
Wnd := CreateWindow           // создает окно и возвращает его дескриптор
(AppName,                    // имя класса
'Demo',                      // заголовок окна
ws_OverlappedWindow,        // стиль окна
cw_UseDefault,               // позиция верхнего левого угла окна
cw_UseDefault,               // позиция верхнего левого угла окна
cw_UseDefault,               // ширина окна
cw_UseDefault,               // высота окна
0,                            // родительское окно
0,                            // меню
HInstance,                   // указывает на экземпляр программы
nil);

```

---

 В распоряжении программистов имеется ещё более совершенная функция `CreateWindowEx()`. Она позволяет создавать окна с расширенными возможностями.

---

## Отображение окна

После создания экземпляра окна осуществляем его вывод на экран. Эту почётную миссию выполнит функция:

```

Function ShowWindow(Wnd : HWND; CmdShow : Integer;) : Boolean;

```

С первым параметром всё просто, это дескриптор только что созданного окна (результат работы функции `CreateWindow`). Вторым параметром устанавливает режим отображения окна. Например,

для того чтобы вывести окно нужно использовать константу SW\_SHOW. Наиболее часто применяемые константы перечислены в таблице 1.2.

Таблица 1.2. Некоторые значения параметра CmdShow

| Значение    | Описание                                   |
|-------------|--------------------------------------------|
| SW_HIDE     | Скрыть окно                                |
| SW_MAXIMIZE | Развернуть окно                            |
| SW_MINIMIZE | Свернуть окно                              |
| SW_RESTORE  | Активизировать и восстановить размеры окна |
| SW_SHOW     | Показать окно                              |

## Обновление заданной области окна

Для перерисовки всей клиентской области окна обычно используют функцию:

```
Function UpdateWindow(Wnd : HWND) :Boolean;
```

Единственный аргумент функции – дескриптор обновляемого окна. Функция информирует Windows о необходимости отправки окну сообщения WM\_PAINT, заставляющего перерисовать клиентскую область окна.

## Организация цикла обработки сообщений

Работа приложений в операционной системе Microsoft Windows основывается на обработке сообщений. Сообщение – это реакция Windows на события, происходящие в системе. События подразделяются на аппаратные (нажатие клавиш на клавиатуре, движение мышки по коврику, и т.п.) и программные (передаваемые приложениями с помощью функций SendMessage и PostMessage). Ни одно сообщение не должно (да, в общем-то, и не может) отправляться в обход операционной системы. Сформированное сообщение помещается в очередь сообщений (*message queue*), затем планировщик системы распределяет их очереди соответствующих процессов. Задача программиста – научить своё приложение обрабатывать сообщения, отправляемые ему операционной системой. Задача приложения – извлекать сообщений из своей очереди, осуществлять их обработку и при этом желательно не “зависнуть”. Общепринятым названием блока команд, извлекающего сообщения из очереди называть **циклом обработки сообщений**.

В цикле обработки сообщений применяется пара функций Windows API: GetMessage() и DispatchMessage(). Определение первой функции выглядит следующим образом:

```
Function GetMessage(
    Msg : TMsg;           // указатель на структуру сообщения
    Wnd : HWND;           // дескриптор окна, созданного программой
    wMsgFilterMin: Cardinal; // нижняя граница фильтра
    wMsgFilterMax : Cardinal; // верхняя граница фильтра
) : boolean;
```

Функция должна осуществить выбор из очереди сообщений сообщения предназначенного окну, идентифицируемому дескриптором Wnd. С помощью GetMessage() допускается обрабатывать и сообщения для всех созданных программой окон, но в этом случае вместо дескриптора окна передавайте нулевое значение. Третий и четвёртый параметры, играют роль фильтров, ограничивающих поток сообщений. Минимальный номер принимаемого сообщения указываем в параметре wMsgFilterMin, максимальный в wMsgFilterMax. Если оба параметра установлены в ноль, то фильтры отключаются и обслуживаются все сообщения. Ключевым аргументом выступает запись, содержащая принимаемое сообщение – Msg. Эта запись соответствует структуре Windows – tagMSG. В Delphi предусмотрен аналог данной структуры – запись TMsg.

```
type
    TMsg = packed record
        Wnd: HWND;
        message: Cardinal;
        wParam: WPARAM;
        lParam: LPARAM;
        time: DWORD;
        pt: TPoint;
```

**end;**

Здесь:

`Wnd` – дескриптор окна, для которого предназначено сообщение, окном может быть любой экранный объект (кнопки, группы переключателей, строки ввода и т.д.);

`message` – число, описывающее сообщение. Для удобства в Windows для каждого числа объявлен идентификатор, начинающийся с префикса `WM` (*window message*). Если перевести дословно – **оконное сообщение**. Например: при перерисовке окна поступает сообщение `WM_PAINT`, при закрытии окна – `WM_CLOSE`, при нажатии клавиши на клавиатуре – `WM_KEYDOWN`;

`wParam` – в зависимости от типа сообщения в поле окажется дополнительная информация. Кроме того, здесь может находиться идентификатор окна или элемента управления, связанного с данным сообщением;

`lParam` – параметр хранит индекс или указатель на некоторые вспомогательные данные в памяти;

`time` – время помещения сообщения в очередь;

`pt` – экранные координаты курсора мышки в момент помещения сообщения в очередь.

Возвращаемся к `GetMessage()`. Цикл обработки сообщений будет продолжаться до тех пор, пока функция не возвратит значение `false`, это знаменательное событие произойдет после получения сообщения `WM_QUIT` – завершение работы программы.

---

Программист имеет право регистрировать в системе свои собственные пользовательские сообщения. Для этого применяется функция `RegisterWindowMessage()`.

---

В программах Delphi классический цикл обработки сообщения выглядит примерно так:

```
while GetMessage(Msg, 0, 0, 0)=true do
begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
end;
```

Здесь инструкции `TranslateMessage()` и `DispatchMessage()` передают сообщение `Msg` обратно операционной системе. Функция `TranslateMessage()` предназначена для преобразования сообщений виртуальных клавиш в соответствующий код `WM_CHAR`. Функция `DispatchMessage()` при посредничестве Windows отправляет сообщение соответствующей оконной процедуре. В таблице 1.3 приведены некоторые типы оконных сообщений Windows, они пригодятся в процессе описания оконной процедуры.

Таблица 1.3. Ключевые оконные сообщения Windows

| Идентификатор               | Описание                                                                                            |
|-----------------------------|-----------------------------------------------------------------------------------------------------|
| <code>WM_ACTIVATE</code>    | Активизация (деактивизация) окна.                                                                   |
| <code>WM_CLOSE</code>       | Закрытие окна.                                                                                      |
| <code>WM_KEYDOWN</code>     | Нажатие клавиши на клавиатуре.                                                                      |
| <code>WM_CHAR</code>        | Для некоторой клавиши было отправлено сообщение <code>WM_KEYDOWN</code> или <code>WM_KEYUP</code> . |
| <code>WM_KEYUP</code>       | Клавиша на клавиатуре отпущена.                                                                     |
| <code>WM_LBUTTONDOWN</code> | Нажата левая кнопка мышки.                                                                          |
| <code>WM_MOUSEMOVE</code>   | Переместился указатель мышки.                                                                       |
| <code>WM_PAINT</code>       | Команда на перерисовку клиентской области окна.                                                     |
| <code>WM_TIMER</code>       | Произошло событие таймера.                                                                          |
| <code>WM_COMMAND</code>     | Щелчок по пункту меню, кнопке.                                                                      |
| <code>WM_QUIT</code>        | Программа должна быть завершена.                                                                    |

## Сообщение WM\_PAINT

Сообщение WM\_PAINT связано с необходимостью перерисовки окна или его части. Указанное сообщение может быть отправлено как операционной системой, так и приложением. В первом случае можно быть уверенным, что сообщение не было сгенерировано зря – Windows никогда не отправит WM\_PAINT в адрес не нуждающегося в перерисовке окна. ОС предварительно проверит состояние региона окна, подлежащего обновлению.

Если в ходе выполнения программы возникает необходимость внеочередной перерисовки, то для генерации WM\_PAINT из кода программы программисты обычно применяют функции UpdateWindow() и RedrawWindow(). Позднее мы ещё не раз обратимся к этим функциям, а сейчас рассмотрим фрагмент кода, демонстрирующий порядок обработки сообщения WM\_PAINT.

```
WM_PAINT :
begin
  DC:=BeginPaint(Window, PAINTSTRUCT);
  {другие команды}
  EndPaint(Window, PAINTSTRUCT);
end;
```

Стандартная обработка сообщения перерисовки окна включает три этапа:

1. Получение дескриптора контекста окна путём вызова функции BeginPaint().
2. Собственно процесс перерисовки.
3. Обращение к функции EndPaint() для освобождения дескриптора контекста.

## Оконная процедура

Основная задача оконной процедуры – организация обработки сообщений, поступающих в адрес окна, поэтому **оконная процедура вызывается в момент получения окном сообщения**. К счастью это не обязывает программиста запереться на пару лет в келье и описать поведение окна при поступлении всех мыслимых и немыслимых сообщений начинающихся с приставки “WM\_”. В Windows предусмотрена обработка сообщений по умолчанию при помощи процедуры DefWindowProc(). Но, если при получении определённого сообщения поведение окна должно чем то отличаться от стандартного, то будьте добры – садитесь за клавиатуру. Мы об этом ещё поговорим, в частности в подразделе посвящённом субклассированию.

Заголовок классической оконной процедуры определяется описанной в Windows функцией WindowProc():

```
Function WindowProc (
  Wnd : HWND;           // дескриптор окна
  uMsg : UINT;          // идентификатор сообщения
  wParam : WPARAM;      // первый (старший) параметр сообщения
  lParam : LPARAM       // второй (младший) параметр сообщения
) : longint;
```

Возвращаемое значение зависит от обработанного сообщения.

---

Для дочерних окон (окон созданных с флагом WS\_CHILD) сообщения отправляются индивидуально (в обход главного окна). Это замечание следует учитывать при разработке приложений, содержащих “цепочки” дочерних окон, например, когда на поверхности окна находится панель группировки, а ей в свою очередь принадлежат другие дочерние окна (списки, кнопки, и т.д).

---

## Отправка сообщений

Приложение должно уметь не только получать, но и отправлять сообщения. Для этого применяются функции SendMessage() и PostMessage(). Они обладают идентичным набором параметров:

```
Function SendMessage (
  Wnd : HWND;           // дескриптор окна-получателя
  Msg : UINT;           // отправляемое сообщение
  wParam : WPARAM;      // дополнительный старший параметр сообщения
  lParam : LPARAM       // дополнительный младший параметр сообщения
) : LRESULT;
```

```

Function PostMessage (
    Wnd : HWND;           // дескриптор окна-получателя
    Msg : UINT;           // отправляемое сообщение
    wParam : WPARAM;      // дополнительный старший параметр сообщения
    lParam : LPARAM       // дополнительный младший параметр сообщения
) : boolean;

```

Однако, не смотря на внешнюю схожесть, поведение функций существенно отличается друг от друга. Функция `SendMessage()` работает в синхронном режиме, она отправляет сообщение непосредственно процедуре окна и ожидает окончания его обработки. В отличие от неё асинхронная функция `PostMessage()` стандартным образом помещает сообщение в очередь Windows и сразу возвращает управление вызвавшей её программе, не дожидаясь результатов обработки сообщения.

Синхронная функция `SendMessage()` относится к разряду потенциально опасных, так как если окно-получатель по каким-то причинам не сможет ответить на сообщение, то вызвавшее функцию приложение может даже потерять работоспособность. Поэтому вместо `SendMessage()` зачастую используют `SendMessageTimeout()`. Указанная функция, ожидает ответ в течение установленного таймаута, и если за этот промежуток времени ответа получено не было она гарантированно возвратит управление своей программе.

---

Исходя из разницы в работе функций, различаются и возвращаемые ими значения. `SendMessage()` возвращает значение, полученное в результате обработки сообщения, `PostMessage()` всего лишь информирует – удалось ли ей поставить сообщение в очередь (значение `true`) или нет (`false`).

---

## Листинг программы

Поскольку мы решили поработать без VCL, то уже самый первый шаг, связанный с созданием нового проекта будет иметь свои особенности. Нам надо создать пустой проект – проект без формы. Самый простой путь для этого – сгенерировать шаблон для нового консольного приложения. Для этого находим пункт меню `File – New... – Other` и в выпорхнувшем на ваш рабочий стол окне “New Items” выбираем иконку “Console Application”. В появившемся листинге удаляем ненужную директиву компилятора `{$APPTYPE CONSOLE}`. В строке `Uses` вместо модуля `SysUtils` установим ссылки на файлы `Windows` и `Messages`. Сохраняем проект под именем `ex01_01.dpr`. Если вы внимательно следовали моим инструкциям, то в редакторе кода останутся лишь эти строки:

```

program ex01_01;
uses Windows, Messages;

begin

end.

```

Ещё раз обращаю внимание читателя на то, что мы явным образом задействовали лишь две библиотеки подпрограмм Delphi. Модуль `Windows` содержит вызовы ключевых функций прикладного интерфейса Windows API, а модуль `Messages` хранит описания основных сообщений. За бортом нашего проекта остались такие монстры визуальной библиотеки компонентов Delphi как `Forms`, `Classes`, `Graphics`, `Controls`, без которых не обходится не одно из стандартных приложений VCL.

Теперь подготовим ряд переменных и записей. Для этого перед словом `begin` набираем следующие строки:

```

const AppName = ' ex01_01'; //константа с именем приложения

var
    Wnd : HWND;           {Дескриптор окна}
    Msg : TMsg;           {Структура для хранения сообщений}
    WindowClass : TWndClass; {Структура класса окна}
    DC:HDC;               {Дескриптор контекста графического устройства}
    PAINTSTRUCT : TPaintStruct; {Структура для рисования}

```

Сразу после объявления переменных, но до слова `begin` описываем оконную процедуру программы.

```

function WindowProc (Window: HWND; Msg: UINT;
                    wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
    WindowProc := 0;
    case Msg of
        {действия при поступлении сообщения WM_DESTROY}
        WM_DESTROY : begin
            PostQuitMessage (0); {завершение выполнения программы}
            Exit;
            end;
        {действия при поступлении сообщения WM_PAINT}
        WM_PAINT : begin {управление прорисовкой окна}
            DC:=BeginPaint(Window, PAINTSTRUCT);
            DrawText(DC,
                    PChar('Hello, Word!'), -1,
                    PAINTSTRUCT.rcPaint,
                    DT_SINGLELINE or DT_CENTER or DT_VCENTER);
            EndPaint(Window, PAINTSTRUCT);
            end;
    end;
    {обработка сообщений по умолчанию осуществляется функцией Windows
    DefWindowProc}
    WindowProc := DefWindowProc(Window, Msg, wParam, lParam);
end;

```

Обратите внимание на последние строки оконной процедуры. В ней мы обращаемся за помощью к функции DefWindowProc() – это функция Windows API осуществляющая обработку сообщений по умолчанию, т.е. всех тех сообщений, которые не были явно описаны в нашем, не постесняемся этого слова выдающемся коде.

Нам осталось заполнить пустое пространство между ключевыми словами begin и end.

```

Begin
//описание класса создаваемого окна
with WindowClass do
begin
    Style := cs_HRedraw or cs_VRedraw;
    lpfnWndProc := @WindowProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    hInstance := 0;
    hIcon := LoadIcon (0, idi_Application);
    hCursor := LoadCursor (0, idc_Arrow);
    hbrBackground := GetStockObject(WHITE_BRUSH);
    lpzMenuName := '';
    lpzClassName := AppName;
end;
//регистрация класса окна
If RegisterClass (WindowClass) = 0 then Halt;
//создание окна
Wnd := CreateWindow(AppName, 'Demo', ws_OverlappedWindow,
                    cw_UseDefault, cw_UseDefault,
                    cw_UseDefault, cw_UseDefault, 0, 0, HInstance, nil);

ShowWindow (Wnd, SW_SHOW); //показ окна
UpdateWindow (Wnd); //перерисовка окна
//цикл обработки сообщений
while GetMessage (Msg, 0, 0, 0) do
begin
    TranslateMessage (Msg);
    DispatchMessage (Msg);
end;
Halt (Msg.wParam); //Завершение работы
программы
end.

```

Теперь простейшее приложение готово – проверяем его работоспособность (см. рис. 1.1). Предвижу возмущение читателя: “Столько работы, для того,

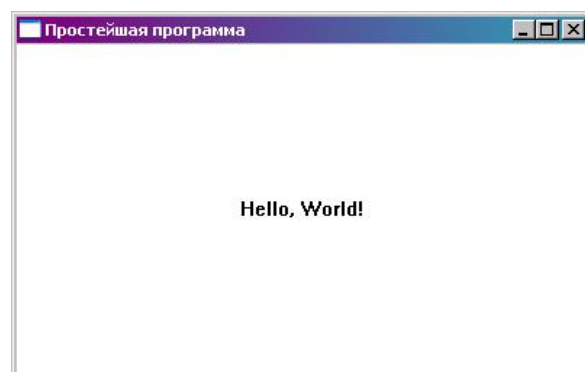


Рисунок 1.1. Hello, World!

чтобы создать элементарный проект!”. Предлагаю взглянуть на это с другой стороны. Например, размер полученного исполняемого файла минимален – он не более 16 килобайт! Создание аналогичной программы на Delphi с применением VCL действительно занимает не более минуты нашего драгоценного времени, но из-за этого размер exe-файла составит не менее 300 килобайт, вот такая арифметика! Но это далеко не все преимущества нашего проекта. В сравнении со стандартным приложением Delphi только что написанная программа менее требовательна к оперативной памяти, задействует вдвое меньшее число объектов GDI, обладает большей производительностью. Подтверждением моих слов может стать рисунок 1.2 на котором представлен экранный снимок диспетчера задач Windows. Диспетчер задач контролирует перечисленные выше показатели проекта написанного на Windows API и проекта построенного при поддержке VCL. Процесс ex01\_01.exe по всем параметрам превосходит своего неповоротливого стандартного собрата Project1.exe.

Приложение на Windows API

Стандартный проект на VCL

| Имя образа    | ЦП | Память    | Дескр. | Объекты GDI |
|---------------|----|-----------|--------|-------------|
| delphi32.exe  | 00 | 64 196 КБ | 330    | 586         |
| dmaster.exe   | 00 | 8 532 КБ  | 206    | 235         |
| ex01_01.exe   | 00 | 1 536 КБ  | 23     | 10          |
| explorer.exe  | 01 | 18 232 КБ | 610    | 425         |
| IAAnotif.exe  | 00 | 1 868 КБ  | 28     | 28          |
| IAANTMon.exe  | 00 | 1 420 КБ  | 94     | 4           |
| ieexplore.exe | 00 | 28 340 КБ | 491    | 469         |
| kavmm.exe     | 00 | 54 616 КБ | 751    | 4           |
| KW5Prod.exe   | 00 | 3 352 КБ  | 255    | 13          |
| lsass.exe     | 00 | 452 КБ    | 357    | 6           |
| LvAgent.exe   | 00 | 1 812 КБ  | 23     | 9           |
| mdm.exe       | 00 | 2 884 КБ  | 87     | 4           |
| nysvc32.exe   | 00 | 3 140 КБ  | 95     | 4           |
| outpost.exe   | 00 | 30 992 КБ | 877    | 163         |
| PDVDServ.exe  | 00 | 2 780 КБ  | 75     | 5           |
| Project1.exe  | 00 | 2 464 КБ  | 27     | 28          |
| RichVideo.exe | 00 | 3 072 КБ  | 80     | 4           |
| RTHDCPL.EXE   | 00 | 21 316 КБ | 139    | 617         |
| rundll32.exe  | 00 | 2 060 КБ  | 25     | 13          |

Процессов: 43    Загрузка ЦП: 4%    Выделение памяти: 402МБ / 6

Рисунок 1.2. Экранный снимок диспетчера задач

## Дочерние окна

Для уяснения ещё некоторых особенностей программирования на Windows API предлагаю продолжить работу над проектом. Несколько дополнив наш код, мы получим опыт размещения на окне элемента управления – кнопки.

1. В разделе объявления переменных добавьте дескриптор будущей кнопки btn:

```
var Wnd, Btn : HWND;
```

2. Сразу после создания экземпляра окна Wnd создайте экземпляр кнопки:

```
Btn:=CreateWindow('BUTTON',
    'Закреть',
    WS_CHILDWINDOW,
    10,
    10,
    75,
    22,
    Wnd, //определяем дескриптор владельца кнопки
    0,
    HInstance,
    nil);
```

3. Покажите экземпляр кнопки:

```
ShowWindow(Wnd, SW_SHOW);
```

```
UpdateWindow(Wnd);
ShowWindow(BTN, SW_SHOW); //показ экземпляра кнопки
```

4. Дополните Case-селектор оконной процедуры ещё одним сообщением:

```
CASE MESSAGE OF
. .
{Поступление сообщения WM_COMMAND}
WM_COMMAND: BEGIN
    if lParam=Btn then {если источник наша кнопка}
    begin
        if MessageBox(WND,
            PChar('Заккрыть окно?'),
            PChar('Подтвердите'),
            MB_OKCANCEL+MB_ICONINFORMATION)=idOK
        then SendMessage(Wnd,WM_DESTROY,0,0); {сами себе отправляем WM_DESTROY}
    END;
END;
```

## Субклассирование

Допустим, что наше приложение содержит цепочки окон, в которых одно окно принадлежит другому. Например, программная логика программы предполагает размещение на главном окне проекта двух групп переключателей. В таком случае при формировании интерфейса проекта разумно применить две группирующие панели с принадлежащими им кнопками (см. рис. 1.3).

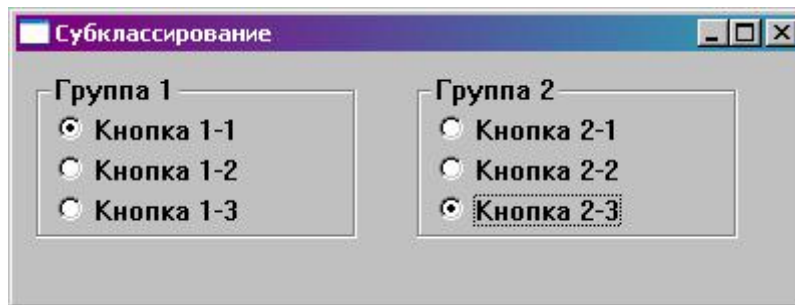


Рисунок 1.3. Внешний вид окна с двумя группами переключателей

Для воплощения нашей идеи на практике целесообразно воспользоваться сообщением WM\_CREATE, вызываемого сразу после создания главного окна приложения. В рамках обработки сообщения создаём панели группировки (описываемые

дескрипторами grBox1 и grBox2) и шесть переключателей (rBtn1... rBtn6), причём первые три кнопки принадлежат первой, а вторые три – второй панели. Ниже представлен фрагмент листинга главной оконной процедуры, претворяющий в жизнь нашу идею.

```
var
    grBox1, grBox2, {дескрипторы панелей группировки}
    rBtn1,rBtn2,rBtn3, rBtn4,rBtn5,rBtn6 : HWND; // дескрипторы кнопок
//...
function WindowProc (Window: HWND; Msg: UINT;
    wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
    case Msg of
        WM_CREATE :
            begin
                //===== Группа переключателей 1 =====
                {создание группирующей панели № 1}
                grBox1:= CreateWindow('BUTTON', 'Группа 1',
                    WS_CHILD OR WS_VISIBLE OR BS_GROUPBOX,
                    10, 10, 160, 82, Window, 0, HInstance, nil);
                {первая кнопка, принадлежит группирующей панели № 1}
                rBtn1 := CreateWindow('BUTTON','Кнопка 1-1',
                    WS_CHILD OR WS_VISIBLE OR BS_AUTORADIOBUTTON OR BS_NOTIFY
                    OR WS_GROUP, 10, 20, 120, 16, grBox1, 0, HInstance, nil);
                //... аналогичный код для кнопок rBtn2 и rBtn3
                //===== Группа переключателей 2 =====
                grBox2:= CreateWindow('BUTTON','Группа 2',
                    WS_CHILD OR WS_VISIBLE OR BS_GROUPBOX,
                    200, 10, 160, 82, Window, 0, HInstance, nil);
                rBtn4 := CreateWindow('BUTTON','Кнопка 2-1',
```

```

        WS_CHILD OR WS_VISIBLE OR BS_AUTORADIOBUTTON OR BS_NOTIFY
        OR WS_GROUP, 10, 20, 120, 16, grBox2, 0, HInstance, nil);
//... аналогичный код для кнопок rBtn5 и rBtn6

    end;
//... обработка остальных сообщений
end;

```

После старта проекта мы столкнёмся с определённой сложностью – мы не сможем оперативно реагировать на действия пользователя, выбирающего ту или иную кнопку. Все наши попытки “отловить” сообщения кнопок в разделе секции WM\_COMMAND главной оконной процедуры обречены на провал. Это объясняется тем, что владельцами кнопок выступают панели группировки, а не как не главное окно приложения. Поэтому все исходящие от кнопок сигналы Windows станет направлять в адрес оконных процедур панелей grBox2 и grBox2, а не в процедуру WindowProc. Наша проблема в том, что в проекте нет оконных процедур для панелей группировки кнопок-переключателей...

Выйти из создавшегося затруднительного положения нам поможет **субклассирование** (*subclassing*). Идея субклассирования заключается в замене оконной процедуры по умолчанию другой – нестандартной.

Хакерский приём подмены оконной процедуры без особых проблем осуществляется функцией:

```
function SetWindowLong(Wnd: HWND; Index: Integer; NewLong: Longint): Longint;
```

Нам лишь требуется передать в функцию три параметра. Первый из них (параметр Wnd) – дескриптор окна, которое мы собираемся обмануть, а нашем случае это панели группировки. Второй параметр Index уточняет задачу функции, например для смены адреса оконной процедуры в него направляется константа GWL\_WNDPROC. Кроме фокуса с субклассированием функция владеет ещё рядом силовых приёмов, например, она позволяет изменить стиль окна (см. таблицу 1.4). Третий параметр NewLong определяет подменяемое значение. В случае успешного выполнения функция возвращает предыдущее 32-х разрядное значение (если речь идёт о субклассировании, то мы получим адрес старой оконной процедуры). Это значение стоит запомнить в отдельной переменной, оно может пригодиться для возврата окна к родной оконной процедуре, но это уже сфера задач функции CallWindowProc(), о ней мы поговорим позднее.

Таблица 1.4. Значения параметра Index функции SetWindowLong()

| Константа     | Описание                                                                      |
|---------------|-------------------------------------------------------------------------------|
| GWL_EXSTYLE   | Установка расширенного стиля окна (совместимого с процедурой CreateWindowEx). |
| GWL_STYLE     | Установка нового стиля окна.                                                  |
| GWL_WNDPROC   | Установка адреса процедуры окна.                                              |
| GWL_HINSTANCE | Установка нового дескриптора приложения.                                      |
| GWL_ID        | Смена идентификатора окна.                                                    |
| GWL_USERDATA  | Установка данных связанных с окном.                                           |
| DWL_DLGPROC   | Новый адрес процедуры диалогового окна.                                       |
| DWL_MSGRESULT | Изменение значения возвращаемого диалоговым окном.                            |
| DWL_USER      | Работа с дополнительной информацией.                                          |

Допустим, что в нашем проекте уже объявлена нестандартная оконная процедура для панели группировки GroupBoxProc(), тогда её вызов будет выглядеть следующим образом.

```

{переменная для хранения адреса стандартной оконной процедуры}
Var OldProc : TFNWndProc;
//...
OldProc:=TFNWndProc(SetWindowLong(grBox1, GWL_WNDPROC,
Longint(@GroupBoxProc)));

```

Последнюю строку примера можно разместить сразу после создания окна панели группировки grBox1. С этого момента все сообщения предназначенные для панели станут направляться в адрес оконной процедуры GroupBoxProc().

---

По умолчанию Delphi использует классическое для языка Паскаль соглашение о вызовах pascal. Такой подход недопустим для функций Windows API, они работают в рамках стандартного соглашения stdcall.

Одно из коренных отличий между соглашениями – порядок обработки параметров функций. Стандартное соглашение обслуживает параметры справа налево, а соглашение Паскаль наоборот – слева направо.

К структуре подставной оконной процедуры предъявляется ряд жёстких требований. Перечень её параметров обязан полностью соответствовать списку параметров обычной оконной процедуры. Также нельзя путать последовательность аргументов. На первом месте всегда идёт дескриптор окна, затем сообщение, старший и младший параметр. После описания заголовка процедуры необходимо указать, что функция опирается на стандартное соглашение о вызовах `stdcall`.

```
function GroupBoxProc (Window: HWND; Msg: UINT;
                      wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
  case Msg of
    WM_COMMAND : begin
      //...
    end;
  end;
  {вызов стандартной процедуры окна}
  CallWindowProc(OldProc, Window, Msg, wParam, lParam);
end;
```

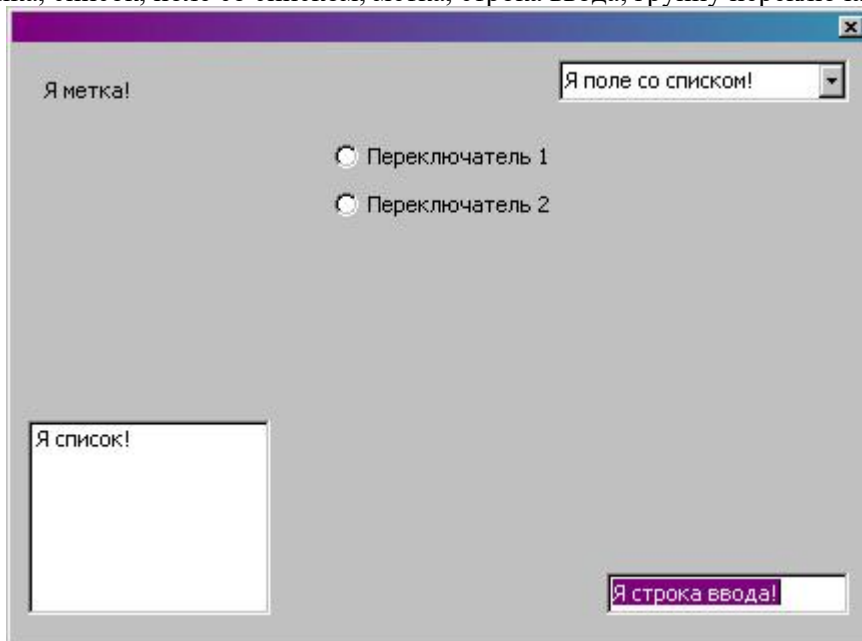
Как правило, код нестандартной оконной процедуры завершается обращением к функции:

```
function CallWindowProc (PrevWndFunc: TFWndProc; Wnd: HWND;
                        Msg: UINT; wParam: WPARAM; lParam: LPARAM): LRESULT;
```

Функция позволяет вернуть обработку сообщений окна в стандартное русло – в “объятия” родной оконной процедуры. Указатель на эту процедуру передаётся в первый параметр функции, в нашем примере это ничто иное, как возвращаемое функцией `SetWindowLong()` значение.

## Задание

1. Напишите исходный код приложения для Windows на Win API
2. Научитесь располагать на поверхности главного окна следующие элементы управления (кнопка, список, поле со списком, метка, строка ввода, группу переключателей).



3. Напишите код, позволяющий сохранять местоположение элементов управления относительно границ окна при изменении его размеров.

## 2. Процессы

*Вид занятия – практическая работа*

*Цель – управление процессами в Windows*

*Продолжительность – 2 часа*

На протяжении всех предшествующих глав мы с вами учились разрабатывать программы для Windows, и, надеюсь, стали неплохими специалистами в области программирования на языке C++ Builder. Однако до настоящих профессионалов нам еще далеко. Создавая приложения, мы опустили весьма важную область знаний о том, как ведет себя программа, выполняющаяся под управлением Windows.

Как бы мы высоко не ценили созданные нами программы, по сути они представляют собой просто статический набор команд. Лишь операционная система превращает программу в полноценное приложение. С точки зрения Windows термин "*приложение*" несколько размыт, вместо него специалисты предпочитают применять термин "*процесс*". Процесс состоит из:

- ☐ *адресного пространства*, представляющего собой набор адресов виртуальной памяти, доступных для процесса;
- ☐ исполняемой программы с ее кодом и данными;
- ☐ списка дескрипторов системных ресурсов, выделяемых процессу;
- ☐ как минимум одного потока, именуемого *потоком управления* (thread of execution), или главным потоком процесса. Именно поток управления и требует от операционной системы времени на обслуживание процесса;
- ☐ *контекста защиты* (security context), определяющего привилегии процесса.

В момент запуска процесса операционная система создает дескриптор — специальную структуру, описывающую стартующий процесс. Благодаря *дескриптору процесса* легко получить информацию, позволяющую системе управлять процессом. В перечень входят:

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

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

### Управление процессом

Главную роль в формировании нового процесса в Windows выполняет функция `CreateProcess()`, создающая процесс и поток управления. Сама по себе функция достойна отдельной главы, т. к. обладает широчайшим спектром настроек и, кроме того, может использоваться для синхронизации работы нескольких процессов и потоков. К сожалению, в рамках этой книги, такую роскошь мы себе позволить не можем, поэтому ограничимся лишь вводным описанием.

**BOOL WINAPI** `CreateProcess(`

```

    __in          LPCTSTR lpApplicationName,
    __in_out      LPTSTR lpCommandLine,
    __in          LPSECURITY_ATTRIBUTES lpProcessAttributes,
    __in          LPSECURITY_ATTRIBUTES lpThreadAttributes,
    __in          BOOL bInheritHandles,
    __in          DWORD dwCreationFlags,
    __in          LPVOID lpEnvironment,
    __in          LPCTSTR lpCurrentDirectory,
    __in          LPSTARTUPINFO lpStartupInfo,
    __out         LPPROCESS_INFORMATION lpProcessInformation
);

```

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

Третий и четвертый параметры определяют атрибуты безопасности создаваемого процесса и главного потока, порожденного этим процессом. Физически данные, передаваемые в аргументы, представляют собой указатели на специальную структуру (листинг 2.1) атрибутов защиты.

#### Листинг 2.1. Структура безопасности `_SECURITY_ATTRIBUTES`

```

typedef struct _SECURITY_ATTRIBUTES {
    DWORD nLength;           //размер структуры в байтах
    LPVOID lpSecurityDescriptor; //указатель на дескриптор защиты процесса
    BOOL bInheritHandle;     //признак наследования возвращаемого дескриптора
} SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES, *LPSECURITY_ATTRIBUTES;

```

Защита по умолчанию подразумевает, что полный доступ к процессу получают лишь его создатель и любой член группы администраторов. Все прочие пользователи к процессу не допускаются.

Одной из особенностей Windows является возможность разграничения доступа пользователей к различным ее объектам. Так, например, не обладающий правами администратора пользователь не имеет возможности просматривать системные каталоги или работать с каким-то ресурсом. Более того, обычный пользователь не имеет права запускать процессы или управлять некоторыми процессам, имеющими высокие, недоступные для него атрибуты безопасности. Если вместо указателя этой функции передается `NULL`, то включается защита по умолчанию и дескриптор создаваемого процесса наследоваться не сможет.

Особый интерес в структуре `_SECURITY_ATTRIBUTES` представляет второе поле — указатель на дескриптор защиты. В этом дескрипторе сосредоточена информация безопасности объекта. Для инициализации дескриптора используется функция `InitializeSecurityDescriptor()`. В параметры безопасности допускается передача неопределенных указателей `NULL`, в таком случае ОС присвоит процессу установленный для текущего сеанса уровень прав.

Пятый по счету параметр `bInheritHandles` разрешает или запрещает создаваемому процессу наследовать дескрипторы из вызывающего процесса. Обычно при создании процесса в этом параметре передается `false`, это означает, что дочерний процесс не должен наследовать наследуемые описатели, принадлежащие родительскому процессу. При передаче в параметр `true` дочерний процесс копирует все описатели родительского. Таким образом, дескрипторы будут идентичны как в родительском, так и в дочернем процессах. С этого момента все объекты родительского процесса становятся доступны и дочернему. И теперь, чтобы уничтожить какой-то объект, от него должны отказаться оба процесса (например, вызвав функцию `CloseHandle`).

Набор флагов `dwCreationFlags` определяет особенности создания процесса (табл. 2.1) и уровень приоритета. Шесть возможных значений приоритета приведены в следующем

подразделе. Параметр `lpEnvironment` — указатель на настройки окружения нового процесса. Путь к текущему каталогу нового процесса — `lpCurrentDirectory`. Параметр `lpStartupInfo` — это специальная структура, описывающая характеристики окна создаваемого приложения. `lpProcessInformation` — указатель на структуру, используемую созданным процессом. Если функция выполнена успешно, то методом возвращается значение `true`.

Таблица 2.1. Основные флаги параметра *CreationFlags*

| Флаг                                 | Описание                                                                                                              |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <code>CREATE_NEW_CONSOLE</code>      | Для нового процесса создается новая консоль, наследующая все параметры родительской консоли                           |
| <code>CREATE_SUSPENDED</code>        | Процесс создается в приостановленном виде, для его запуска необходимо обращение к функции <code>ResumeThread()</code> |
| <code>DEBUG_PROCESS</code>           | Флаг указывает на то, что данный процесс и все его дочерние процессы запущены в режиме отладки                        |
| <code>DEBUG_ONLY_THIS_PROCESS</code> | Процесс запущен в режиме отладки                                                                                      |
| <code>INHERIT_CALLER_PRIORITY</code> | Процесс наследует приоритет запустившего его процесса                                                                 |

В листинге 2.2 приведен пример запуска процесса Блокнота (файл `notepad.exe`), в командной строке процессу передается указание открыть файл `demo.txt`.

#### Листинг 2.2. Создание процесса

```

_SECURITY_DESCRIPTOR SD;
InitializeSecurityDescriptor(&SD, SECURITY_DESCRIPTOR_REVISION);

TSecurityAttributes SA;
SA.nLength = sizeof(TSecurityAttributes);
SA.lpSecurityDescriptor = &SD;
SA.bInheritHandle = true;

STARTUPINFO SI;
ZeroMemory(&SI, sizeof(STARTUPINFO));

PROCESS_INFORMATION PI;

CreateProcess(L"c:\\windows\\notepad.exe", //запускаемое приложение
             L"open c:\\demo.txt", //командная строка
             &SA, &SA, false,
             NORMAL_PRIORITY_CLASS, //нормальный приоритет
             NULL, NULL, &SI, &PI);

```

## Доступ к процессу

Для получения доступа к определенному процессу, исполняющемуся в ОС, следует воспользоваться функцией

```

HANDLE WINAPI OpenProcess( __in DWORD dwDesiredAccess,
                          __in BOOL bInheritHandle,
                          __in DWORD dwProcessId);

```

Первый параметр функции контролирует права вызвавшего функцию приложения. Наиболее универсальный аргумент — константа `PROCESS_ALL_ACCESS`, которая сигнализирует, что мы должны обладать всеми возможными правами доступа. Второй параметр `bInheritHandle` вам уже знаком по функции `CreateProcess()`, он определяет особенности наследования дескрипторов. Последний параметр `dwProcessId` должен содержать идентификатор процесса. В случае успешного выполнения функция возвратит дескриптор запрашиваемого процесса.

**Внимание!**

Для получения идентификатора процесса `ProcessId` можно воспользоваться услугами функций из состава библиотеки `Tool Help Library`, предоставляющей информацию о протекающих в системе процессах.

Для выяснения дескриптора процесса, созданного нашим приложением, обращаемся к функции

```
HANDLE WINAPI GetCurrentProcess(void);
```

Функция в аргументах не нуждается и возвращает дескриптор процесса, в рамках которого она была вызвана.

## Приоритет процесса

Количество квантов процессорного времени, выделяемого процессу (точнее, потоку управления), определяется его приоритетом. Зависимость прямо пропорциональная — чем выше приоритет, тем большего внимания его удостоит процессор (или процессоры) вашего компьютера.

Система Windows поддерживает шесть категорий приоритетов процессов:

- ☐ фоновый (`IDLE_PRIORITY_CLASS`);
- ☐ нормальный заднего плана (`BELOW_NORMAL_PRIORITY_CLASS`);
- ☐ нормальный (`NORMAL_PRIORITY_CLASS`);
- ☐ нормальный переднего плана (`ABOVE_NORMAL_PRIORITY_CLASS`);
- ☐ высокий (`HIGH_PRIORITY_CLASS`);
- ☐ процесс реального времени (`REALTIME_PRIORITY_CLASS`).

По умолчанию, при создании нового процесса ему присваивается нормальный приоритет.

Для системного программиста большой практический интерес предоставляет возможность ускорения или замедления выполняющегося процесса. Для достижения этой цели достаточно сделать два шага: во-первых, получить дескриптор процесса (с помощью функции `OpenProcess` или `GetCurrentProcess`) и, во-вторых, изменить его приоритет.

Тем или иным способом приобретя дескриптор интересующего нас процесса, приступаем к главному — меняем приоритет задачи. Для этого вызовем функцию

```
BOOL WINAPI SetPriorityClass(__in HANDLE hProcess,
                             __in DWORD dwPriorityClass);
```

Первый аргумент требует дескриптор процесса, второй — константа приоритета.

Если планируется выяснить приоритет процесса — воспользуйтесь функцией

```
DWORD WINAPI GetPriorityClass(__in HANDLE hProcess);
```

В зависимости от приоритета функция возвратит одну из констант приоритета процесса. Представленный в листинге 2.3 код демонстрирует способ определения приоритета выполняющейся задачи.

### Листинг 2.3. Выяснение приоритета текущего процесса

```
String msg;
HANDLE hProcess = GetCurrentProcess(); //получаем дескриптор процесса
switch (GetPriorityClass(hProcess)) {
    case IDLE_PRIORITY_CLASS      : msg="Фоновый"; break;
    case NORMAL_PRIORITY_CLASS    : msg="Нормальный"; break;
    case HIGH_PRIORITY_CLASS      : msg="Высокий"; break;
    case REALTIME_PRIORITY_CLASS  : msg="Реального времени"; break;
}
ShowMessage(msg);
```

## Завершение процесса

Наилучший вариант завершения процесса — это окончание выполнения процесса только после возврата управления входной функцией потока управления. Только в этом случае можно утверждать, что процесс корректно освободит все свои объекты, а ОС очистит занимаемую стеком главного потока память и уменьшит счетчик на единицу.

Процесс может прекратить свое существование самостоятельно, для этого он должен вызвать функцию

```
VOID WINAPI ExitProcess( UINT uExitCode);
```

Но если процесс не желает уходить из жизни добровольно, то применяют функцию

```
BOOL WINAPI TerminateProcess(HANDLE hProcess,  UINT uExitCode);
```

Функция может быть вызвана родительским процессом или любым другим процессом, знающим дескриптор разрушаемого процесса. В обеих функциях параметр `uExitCode` содержит код завершения процесса. Этот код можно получить с помощью функции `GetExitCodeProcess()`.

## Запуск дочерних процессов

В распоряжении C++ Builder есть заголовочный модуль `process.h` в котором описаны прототипы целого набора функций (`exec1`, `execle`, `exec1p`, `exec1pe`, `execv`, `execve`, `execvp`, `execvpe`, `_wexec1`, `_wexecle`, `_wexec1p`, `_wexec1pe`, `_wexecv`, `_wexecve`, `_wexecvp`, `_wexecvpe`) способных запускать дочерние процессы.

Названия функций построены по следующему правилу:

- ☐ Префикс `_w` указывает на то, что параметры функций способны работать с кодировкой Unicode.
- ☐ Слово `exec` является сокращением от англ. `execute` (выполнить).
- ☐ Символы “l”, “v” и “p” в конце имени конкретизируют возможности функции.
  - Функция с символом “l” — применяется в тех случаях, когда программисту заранее известно число аргументов, которые следует передать дочернему процессу.
  - Символ “v” — предполагает, что дочернему процессу следует отправить целый массив аргументов.
  - Символ “p” — указывает на то, что исполняемый файл с запускаемым процессом может находиться в каталогах, описанных в переменной окружения `PATH`, иначе будет проанализирован лишь текущий каталог.
  - Символ “e” — говорит о том, что в адрес дочернего процесса могут быть переданы дополнительные сведения о параметрах окружения, иначе они будут унаследованы от родительского процесса.

Основная положительная черта перечисленных функций заключается в том, что они позволяют передавать в дочерний процесс некоторый набор аргументов, тем самым конкретизируя порядок их работы. Это как нельзя лучше подходит для управления консольными приложениями, которые призваны самостоятельно решать небольшие задачи, например, связанные с очисткой диска от мусора, архивацией данных, и т.п.

Функции наделены схожими параметрами

```
int _wexec1(const wchar_t * __path, const wchar_t * __arg0, ...);
int _wexecle(const wchar_t * __path, const wchar_t * __arg0, ...);
int _wexec1p(const wchar_t * __path, const wchar_t * __arg0, ...);
int _wexec1pe(const wchar_t * __path, const wchar_t * __arg0, ...);
int _wexecv(const wchar_t * __path, wchar_t * const * __argv);
int _wexecve(const wchar_t * __path, wchar_t * const * __argv,
             wchar_t * const * __env);
int _wexecvp(const wchar_t * __path, wchar_t * const * __argv);
int _wexecvpe(const wchar_t * __path,
              wchar_t * const * __argv,   wchar_t * const * __env);
```

Первый параметр должен содержать путь и имя подлежащего запуску исполняемого .exe файла или командного файла .com. Затем следует перечень аргументов, который мы намерены передать дочернему процессу. Этот перечень должен завершаться NULL.

## Сбор информации о процессах Windows

Для исследования выполняющихся в Windows задач применяются информационные функции из библиотеки Tool Help Library. Этих функций немногим больше десятка, они нацелены на предоставление нам данных о выполняющихся на момент опроса процессах. Особо подчеркну словосочетание "на момент опроса", и вот почему: Windows — это динамическая система, в которой задачи вправе рождаться и умирать ежесекундно. Поэтому функции Tool Help Library делают мгновенный снимок состояния системы в указанный программистом момент времени и затем "препарируют" полученные данные, извлекая из снимка требуемую информацию. Если есть необходимость в непрерывном отслеживании состояния системы, то применяйте таймер, "фотографирующий" ОС с определенной периодичностью. Роль фотографа отводится функции

```
HANDLE WINAPI CreateToolhelp32Snapshot (DWORD dwFlags,
                                           DWORD th32ProcessID);
```

Параметр `Flags` определяет, какую информацию мы планируем включить в снимок (табл. 2.2). Если мы исследуем конкретный процесс, то во второй параметр направляется дескриптор этого процесса `th32ProcessID`. При успешном выполнении метод возвращает указатель на структуру в памяти, называемую снимком.

Таблица 2.2. Возможные значения параметра `Flags` функции `CreateToolhelp32Snapshot()`

| Значение            | Описание                                                             |
|---------------------|----------------------------------------------------------------------|
| TH32CS_INHERIT      | Наследовать дескриптор снимка                                        |
| TH32CS_SNAPALL      | Включить в снимок всю информацию                                     |
| TH32CS_SNAPHEAPLIST | Включить в список данные о кучах в процессе <code>ProcessID</code>   |
| TH32CS_SNAPMODULE   | Включить в список данные о модулях в процессе <code>ProcessID</code> |
| TH32CS_SNAPPROCESS  | Включить в снимок данные о процессах                                 |
| TH32CS_SNAPTHREAD   | Включить в снимок данные о потоках                                   |

Для исследования имеющегося снимка на предмет получения информации о процессах применяют работающие в паре функции:

```
BOOL WINAPI Process32First (HANDLE hSnapshot, LPPROCESSENTRY32 lppe);
BOOL WINAPI Process32Next (HANDLE hSnapshot, LPPROCESSENTRY32 lppe);
```

Параметры методов идентичны. Первый из них — дескриптор снимка, второй — указатель на структуру `PROCESSENTRY32` (ее аналог в VCL — `TProcessEntry32`), в которую они помещают результаты исследований.

Как мы уже отмечали, сбор данных производится совместно. Первоначально вызывается метод `Process32First()`, проверяющий наличие интересующей нас информации в снимке. Если такая информация имеется, то метод возвращает `true`. Затем наступает черед функции `Process32Next()`. Этот метод вызывается в цикле до тех пор, пока им не будет возвращено значение `false`. Это признак того, что все данные собраны.

Переключим внимание на применяемую при исследовании задач структуру `tagPROCESSENTRY32` (листинг 2.4).

**Листинг 2.4. Объявление записи `PROCESSENTRY32`**

```
typedef struct tagPROCESSENTRY32 {DWORD dwSize; //размер записи
    DWORD cntUsage; //счетчик ссылок на процесс
    DWORD th32ProcessID; //идентификатор процесса
    ULONG_PTR th32DefaultHeapID; //идентификатор кучи процесса
    DWORD th32ModuleID; //модуль связанный с процессом
    DWORD cntThreads; //количество дочерних потоков
    DWORD th32ParentProcessID; //родительский процесс
    LONG pcPriClassBase; //базовое значение приоритета процесса
    DWORD dwFlags; //не используется
    TCHAR szExeFile[MAX_PATH]; //имя исполняемого файла
} PROCESSENTRY32, *PPROCESSENTRY32;
```

Все поля записи снабжены комментариями и в дополнительном обсуждении не нуждаются, но за небольшим исключением. Перед первым вызовом структура типа tagPROCESSENTRY32 должна быть проинициализирована — полю dwSize передается значение размера структуры sizeof (PROCESSENTRY32).

Рассмотрим упрощенный пример сбора данных о процессах системы (листинг 34.8). Для этого нам понадобится создать новое приложение и на поверхность формы поместить компонент TListView. Переведите его в табличный режим просмотра (ViewStyle=vsReport) и в редакторе колонок создайте три колонки для отображения: идентификатора процесса, идентификатора родительского процесса, числа дочерних потоков и приоритета процесса. Подключите к проекту модуль tlhelp32.h и повторите приведенный в листинге 2.5 код.

**Листинг 2.5. Получение "снимка" выполняющихся задач**

```
#include "tlhelp32.h";
...
void __fastcall TFormMain::GetProcessInformation()
{
    PROCESSENTRY32 PrcEntry;
    HANDLE Snapshot;

    LVProcesses->Items->BeginUpdate(); //начало обновления
    LVProcesses->Items->Clear(); //удаляем старые данные
    PrcEntry.dwSize=sizeof (PROCESSENTRY32); //инициализируем структуру
    Snapshot=CreateToolhelp32Snapshot (TH32CS_SNAPPROCESS,0); //делаем снимок

    if (Process32First (Snapshot,&PrcEntry))
        do
            /***** передаем сведения в LVProcesses:TListView *****/
            { TListItem* LI=LVProcesses->Items->Add();
                HANDLE ProcessHandle=
                    OpenProcess (PROCESS_ALL_ACCESS,False,PrcEntry.th32ProcessID);

                LI->Caption=PrcEntry.szExeFile;
                String s=Format ("%d",ARRAYOFCONST ((PrcEntry.th32ProcessID)));
                LI->SubItems->Add(s); //id процесса
                s=Format ("%d",ARRAYOFCONST ((PrcEntry.th32ParentProcessID)));
                LI->SubItems->Add(s); //родительский процесс
                s=Format ("%d",ARRAYOFCONST ((PrcEntry.cntThreads)));
                LI->SubItems->Add(s); //потоков
                s=Format ("%d",ARRAYOFCONST ((PrcEntry.pcPriClassBase)));
                LI->SubItems->Add(s); //приоритет
            } /*****
        while (Process32Next (Snapshot,&PrcEntry));

    CloseHandle (Snapshot);
    LVProcesses->Items->EndUpdate();
}
```

На рис. 2.1 представлен экранный снимок утилиты, собирающей сведения о процессах с помощью библиотеки Tool Help Library.

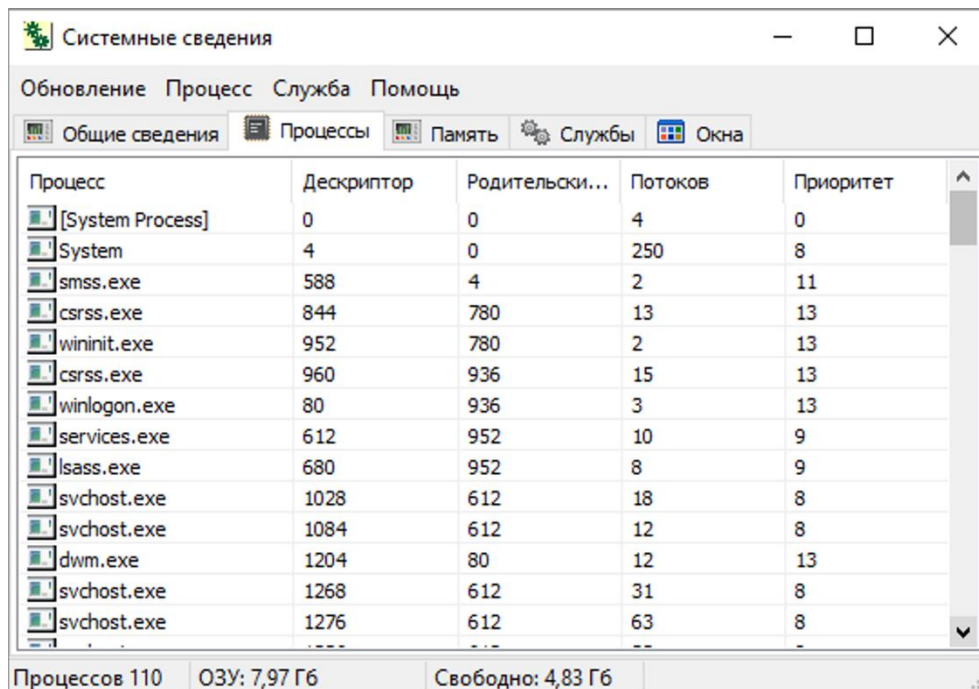


Рис. 2.1. Сбор сведений о процессах Windows

#### Замечание

Кроме рассмотренных функций, в состав Tool Help Library входят методы, позволяющие получить информацию о выполняющихся в данный момент потоках (Thread32First и Thread32Next), модулях (Module32First и Module32Next) и кучах памяти (Heap32First, Heap32Next, Heap32ListFirst и Heap32ListNext).

## Время выполнения процесса

При анализе производительности системы часто возникает необходимость осуществить хронометраж процесса, т. е. выяснить, какое время уделит процессор компьютера тому или иному из процессов.

```

BOOL WINAPI GetProcessTimes (
    __in        HANDLE hProcess,
    __out       LPFILETIME lpCreationTime,
    __out       LPFILETIME lpExitTime,
    __out       LPFILETIME lpKernelTime,
    __out       LPFILETIME lpUserTime
);

```

Здесь: hProcess — дескриптор процесса; lpCreationTime — время создания; lpExitTime — время приостановки процесса; lpKernelTime — время выполнения процесса в режиме ядра, здесь хранится суммарное время выполнения всех потоков в режиме ядра; lpUserTime — время выполнения процесса в пользовательском режиме. При работе в VCL обработка возвращаемых этой функцией значений сопряжена с некоторыми трудностями преобразования данных: четыре последних параметра представлены типом данных `_FILETIME` (в терминах библиотеки VCL — `TFileTime`) (листинг 2.6).

#### Листинг 2.6. Объявление структуры `_FILETIME`

```

typedef struct _FILETIME {
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
} FILETIME, *PFILETIME;

```

В этом типе данных время описано в несколько непривычном для работающего в VCL программиста формате: 32-разрядные элементы записи объединяются в учетверенное слово и образуют хранилище для 64-разрядного значения. За единицу измерения принят интервал в 100 наносекунд, а за точку отсчета — полночь 1 января 1601 года. За декодирование значений такого типа отвечают функции Windows API `FileTimeToSystemTime()` и `SystemTimeToFileTime()`.

Функции обеспечивают взаимное преобразование `_FILETIME` в `_SYSTEMTIME` (аналог структура `TSystemTime`) и наоборот. К сожалению, на этом трудности не заканчиваются, т. к. тип данных `SYSTEMTIME` по-прежнему несовместим с базовым для VCL типом `TDateTime`. В этом нетрудно убедиться, единожды взглянув на структуру `TSystemTime` (листинг 2.7).

#### Листинг 2.7. Объявление записи `TSystemTime`

```
typedef struct _SYSTEMTIME {WORD wYear;    //год, например, 2013
                             WORD wMonth;  //месяц, например 1, 2, ..., 12
                             WORD wDayOfWeek; //день недели 1, 2, ..., 7
                             WORD wDay;    //день месяца 1, 2, ..., 31
                             WORD wHour;   //час 1, 2, ..., 23
                             WORD wMinute; //минута 1, 2, ..., 59
                             WORD wSecond; //секунда 1, 2, ..., 59
                             WORD wMilliseconds; //миллисекунда 1, 2, ..., 999
                             } SYSTEMTIME, *PSYSTEMTIME;
```

Однако в этом формате записи время "разложено по полочкам", и уже дело техники — перевод значений, представленных структурой, в `TDateTime`: вспомните функции `EncodeDateTime()` и `DecodeDateTime()`, пример работы с которыми предложен в листинге 2.8.

#### Листинг 2.8. Дата/время запуска текущего процесса

```
HANDLE hProcess =GetCurrentProcess();    //получаем дескриптор процесса

_FILETIME CreationTime, ExitTime, KernelTime, UserTime;

GetProcessTimes(hProcess, &CreationTime, &ExitTime,
                &KernelTime, &UserTime);

_SYSTEMTIME SystemTime;
FileTimeToSystemTime(&CreationTime, &SystemTime);

/* необходим модуль <System.DateUtils.hpp> */
TDateTime DateTime=EncodeDateTime(SystemTime.wYear,
                                   SystemTime.wMonth, SystemTime.wDay, SystemTime.wHour,
                                   SystemTime.wMinute, SystemTime.wSecond, SystemTime.wMilliseconds);

String msg="Старт процесса : \n"+DateTimeToStr(DateTime)+" по Гринвичу";
ShowMessage(msg);
```

## Получение сведений о версии ОС

Зачастую для управления процессами следует владеть полной информацией о версии операционной системы, под управлением которой выполняется наше приложение. Для получения этих сведений в C++ Builder следует обратиться к объявленной в модуле `System.SysUtils.hpp` структуре (листинг 2.9).

#### Листинг 2.9. Структура `TOSVersion`

```
struct DECLSPEC_DRECORD TOSVersion
{
```

```

public:
    #pragma option push -b-
    enum TArchitecture : unsigned char { arIntelX86, arIntelX64 };
    #pragma option pop

    #pragma option push -b-
    enum TPlatform : unsigned char { pfWindows, pfMacOS };
    #pragma option pop

private:
    //частные объявления
public:
    static bool __fastcall Check(int AMajor)/* overload */;
    static bool __fastcall Check(int AMajor, int AMinor)/* overload */;
    static bool __fastcall Check(int AMajor, int AMinor, int AServicePackMajor)/* overload */;
    static System::UnicodeString __fastcall ToString();
    __property TArchitecture Architecture = {read=FArchitecture};
    /* static */ __property int Build = {read=FBuild};
    /* static */ __property int Major = {read=FMajor};
    /* static */ __property int Minor = {read=FMinor};
    /* static */ __property System::UnicodeString Name = {read=FName};
    /* static */ __property TPlatform Platform = {read=FPlatform};
    /* static */ __property int ServicePackMajor = {read=FServicePackMajor};
    /* static */ __property int ServicePackMinor = {read=FServicePackMinor};
};

```

Структура инициализируется самостоятельно и сразу возвращает все сведения об установленной на компьютере ОС Windows или OS X (если идет речь о проекте FireMonkey).

### 3. Многопоточные приложения

*Вид занятия – практическая работа*

*Цель – разработка многопоточных приложений*

*Продолжительность – 2 часа*

Нам уже известно, что в момент старта процесса под управлением Windows обязательно создается главный поток процесса (поток управления). Перед главным потоком приложения поставлена серьезная задача — он отвечает за взаимодействие с ОС и за выполнение кода программы. Кроме обязательного потока управления процесс имеет возможность обладать некоторым набором вспомогательных потоков (нитей). Например, текстовый процессор Microsoft Word (в котором набирались эти строки) одновременно с вводом текста разбивает документ на страницы, контролирует мои знания русского языка (безжалостно подчеркивая грамматические и синтаксические оплошности), отправляет задание на печать и решает ряд других менее очевидных для обычного пользователя задач. Все это происходит одновременно благодаря тому, что в MS Word запущено несколько потоков. Каждый из потоков выполняет поставленную перед ним подзадачу и при желании может быть приостановлен или запущен вновь.

#### Поток *TThread*

Для программной реализации потока в VCL объявлен абстрактный класс *TThread*, на основе которого программист создаст свой полнофункциональный поток. На палитре компонентов элемент управления *TThread* вы не найдете. Для того чтобы самое обычное приложение сделать многопоточным, требуется к уже существующему приложению добавить специальный программный модуль. Для этого после создания нового приложения выберите пункт меню **File | New | Other** и на вкладке **New** диалогового окна **New Items** найдите значок **ThreadObject**. После нажатия кнопки **OK** в новом диалоговом окне требуется ввести имя нового класса — потока, например, *TDemoThread*.

В результате выполненных действий C++ Builder создаст представленный в листинге 3.1 шаблон, предназначенный для описания потока. Обратите внимание, что в секции *Protected* среда проектирования уже подготовила метод *Execute()*. В теле этой функции программист располагает программный код, который станет исполняться в отдельном потоке. Если подзадача вызывается многократно, то строки кода помещаются внутрь тела цикла, выход из которого завершает работу потока.

**Листинг 3.1. Модуль потока *TDemoThread***

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

#include "Unit2.h"
#pragma package(smart_init)
//-----
__fastcall TDemoThread::TDemoThread(bool CreateSuspended)
                                : TThread(CreateSuspended)
{
    //конструктор потока
}
//-----
void __fastcall TDemoThread::Execute()
{
    //---- программный код потока ----
}
//-----
```

При старте приложения автоматическое создание потока не произойдет, для этого следует явно обратиться к конструктору потока.

```
__fastcall TThread(bool CreateSuspended);
```

Единственный параметр конструктора определяет, каким образом производится запуск потока. Если параметр `CreateSuspended` установлен в `false`, то поток стартует немедленно сразу после создания экземпляра класса. Иначе поток создается в спящем режиме. В последнем случае для запуска потока вызывайте метод

```
void Start(void);
```

Говоря образно, эта функция заставит биться сердце потока.

Для временной приостановки выполнения потока используйте метод

```
void Suspend _DEPRECATED_ATTRIBUTE0 (void);
```

Для возобновления работы приостановленного потока вновь вызовем метод `Start()`.

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

```
__property bool Suspended;
```

Если вы установите свойство в `true`, то поток приостановится, `false` — вновь активизируется.

#### *Замечание*

В более ранних версиях C++ Builder вместо метода `Start()` применялся метод `Resume()`. Начиная с C++ Builder 2010, применение этого метода не рекомендуется.

Для временной приостановки потока программисты часто пользуются методом класса

```
static void Sleep(int Timeout);
```

В качестве параметра передается число миллисекунд, на которые необходимо заморозить подзадачу.

#### *Замечание*

Основное достоинство метода `Sleep()` в том, что поток, переведенный в состояние сна, не снижает производительность системы, т. к. не требует выделения ему квантов времени.

Признаком того, что поток завершил выполнение своей задачи, выступает значение `true`, возвращаемое свойством

```
__property bool Finished;
```

Для полной остановки потока вызывайте метод

```
void Terminate(void);
```

Это способ остановки называется мягким. Метод просто-напросто присваивает значение `true` свойству

```
__property bool Terminated;
```

В этом случае C++ Builder не настаивает на немедленном прекращении работы потока и разрешает ему достичь логического конца.

Если есть мягкий способ остановки потока, то где-то в недрах Windows можно обнаружить и "жесткий". Если вы сторонник радикальных решений и категорически настаиваете на том, чтобы поток завершился немедленно, то пригодится функция из арсенала Windows API.

```
BOOL WINAPI TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

Здесь: `hThread` — дескриптор потока; `ExitCode` — программный код выхода из потока, в простейшем случае передавайте значение 0. Если функция завершится успешно, то она вернет ненулевое значение.

Очень часто проверка состояния свойства служит сигналом к выходу из цикла обработки задачи потока (листинг 3.2).

**Листинг 3.2. Шаблон кода основного метода потока**

```
void __fastcall TDemoThread::Execute()
{
    do
    {
        //код потока
    }
    while (Terminated);
}
```

Останов потока сопровождается вызовом обработчика события

```
__property TNotifyEvent OnTerminate;
```

Если установить в true свойство

```
__property bool FreeOnTerminate;
```

то сразу после останова уничтожится экземпляр потока. Иначе за вызов деструктора потока отвечает программист.

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

```
__property Object* FatalException; //только для чтения
```

будет записан объект ошибки.

Еще одним способом, косвенно проверяющим корректность завершения работы потока, является использование свойства

```
__property int ReturnValue;
```

Это служебная переменная, в которую можно записывать данные, например, о количестве запусков потока. Свойство ReturnValue часто применяется совместно с методом ожидания WaitFor(). Функция предназначена для организации ожидания, пока какой-нибудь другой поток не выполнит поставленную перед ним задачу.

Существует альтернативный доступ к значению ReturnValue, его обеспечивает метод класса

```
static void SetReturnValue(int Value);
```

Благодаря методу мы сможем изменить возвращаемое потоком значение, даже не имея прямого доступа к экземпляру потока.

Для взаимодействия с функциями Windows API может пригодиться дескриптор потока, его вы найдете в свойстве

```
__property unsigned Handle;
```

Современные версии Windows поддерживают работу с несколькими процессорами, распределяя задачи между процессорами (или их ядрами, если речь идет о многоядерном процессоре). Класс TThread способен проинформировать программиста о процессоре компьютера. Свойство

```
__property bool IsSingleProcessor;
```

в состоянии true уведомляет разработчика, что поток будет запущен на однопроцессорной станции, если же свойство возвратит false, то стоит обратиться к свойству

```
__property int ProcessorCount;
```

возвращающему число процессоров (ядер).

**Внимание!**

Разработчику рекомендуется не запускать в приложении более 16 потоков на один процессор, иначе при превышении этого значения будет существенно снижена производительность системы.

Если в приложении допускается исполнение нескольких потоков, то мы имеем право попросить текущий процессор перейти на обслуживание очередного потока. Для этого предназначен метод класса

```
static void Yield();
```

Заметим, что окончательное решение на постановку потока на обслуживание всегда останется за операционной системой.

**Метод ожидания**

При проектировании многопоточного приложения, в котором несколько потоков обслуживают один и тот же ресурс (файл, коммуникационный порт и т. п.) или выполняют общий участок кода, надо исключить конфликт совместного доступа к ресурсу. Большую помощь в этом оказывает метод ожидания, позволяющий одному потоку дожидаться завершения другого.

```
unsigned WaitFor(void);
```

Функция возвращает значение, содержащееся в свойстве ReturnValue ожидаемого потока (табл. 3.1).

*Таблица 3.1. Значения, возвращаемые методом WaitFor()*

| Значение       | Описание                                                                                                                                                                                                                                                             |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WAIT_OBJECT_0  | Контролируемый объект перешел в сигнальное состояние. Это признак того, что выполняющийся ранее поток корректно завершил свою задачу                                                                                                                                 |
| WAIT_TIMEOUT   | Хотя время таймута истекло, но состояние опрашиваемого объекта синхронизации несигнальное. Это означает, что ресурс пока захвачен другим потоком                                                                                                                     |
| WAIT_ABANDONED | Выполнение обладавшего объектом синхронизации потока завершилось, но объект по-прежнему захвачен этим потоком. В виду некорректности такого положения вещей объект синхронизации передается во владение вызвавшего его потока и переводится в несигнальное состояние |
| WAIT_FAILED    | Ошибка выполнения метода                                                                                                                                                                                                                                             |

**Замечание**

Метод WaitFor() применяется для совместной работы не только с потоками, но и с такими объектами синхронизации как события, мьютексы и семафоры. Ключевая особенность метода ожидания заключается в том, что он не возвращает результат немедленно, а ждет, пока не будет удовлетворен некоторый перечень заданных программистом критериев. Таким образом, поток, обратившийся к объекту синхронизации при помощи метода WaitFor(), превращается в пленника, ожидая от метода ответа. Соответственно выполнение потока приостанавливается, и таким образом решается задача синхронизации.

**Управление приоритетом потока**

Обычный поток приложения (точно так, как и поток управления приложением) обладает определенным приоритетом, от которого зависит, сколько квантов времени будет выделено процессором компьютера для обработки подзадачи. Приоритет потока определяется значениями двух составляющих: приоритетом процесса-владельца потока и собственно приоритетом этого потока. Поэтому приоритет обычного потока часто называют *относительным приоритетом*. Такое название объясняется тем, что два потока, скажем, с приоритетом `tpNormal`, но порожденные процессами с различными приоритетами, не смогут претендовать на одинаковое процессорное время. Реальный приоритет потока определяется приоритетом порождающего его процесса и своим приоритетом. В Windows обычно приоритет потока оценивается значением, лежащим в диапазоне от 1 до 31 (табл. 3.2).

Таблица 3.2. Расчет приоритета потока приложения

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

В C++ Builder приоритет потока определяется свойством

```
__property TThreadPriority Priority; //по умолчанию tpNormal
enum TThreadPriority : unsigned char { tpIdle, // фоновый приоритет
    tpLowest, //низкий приоритет
    tpLower, //пониженный приоритет
    tpNormal, //нормальный приоритет
    tpHigher, //повышенный приоритет
    tpHighest, //высочайший приоритет
    tpTimeCritical}; //приоритет реального времени
```

#### **Внимание!**

Без особой на то необходимости старайтесь не создавать потоков с приоритетом выше нормального. Такой поступок может привести к тому, что системные ресурсы будут нацелены на обслуживание только этой задачи.

#### **Замечание**

В состав RAD Studio входит специальная утилита, предназначенная для просмотра потоков приложения. Для обращения к утилите этого выберите пункт меню **View | Debug Windows | Threads**.

## **Время выполнения потока**

В составе Windows API предусмотрена возможность оценки затрат процессорного времени на любой из потоков процесса:

```
BOOL WINAPI GetThreadTimes( __in HANDLE hThread,
    __out LPTIME lpCreationTime, __out LPTIME lpExitTime,
    __out LPTIME lpKernelTime, __out LPTIME lpUserTime);
```

В первый аргумент функции следует направить дескриптор исследуемого потока, остальные параметры и порядок их применения аналогичны параметрам функции `GetProcessTimes()` рассмотренной ранее в этой главе.

## **Синхронный и асинхронный вызовы внешнего метода**

Если из тела основного метода потока `Execute()` необходимо осуществить вызов внешнего метода (метода не принадлежащего потоку) или обратиться к свойству компонента библиотеки VCL, то не следует это делать напрямую, вместо этого надо использовать специализированные методы `Synchronize()` или `Queue()`. Задача методов — исключение взаимных блокировок потоков.

#### **Замечание**

Блокировка потоков возникает в том случае, когда поток или несколько потоков пытаются одновременно получить доступ к одному и тому же ресурсу. Если этот ресурс захвачен третьей стороной, например главным

потоком приложения, и корректный захват ресурса невозможен, то потоки попадают в тупик. Такая ситуация называется *взаимной блокировкой*.

Существует несколько перегружаемых версий методов, мы остановимся на наиболее простых:

```
void Synchronize(TThreadMethod AMethod);
void Queue(TThreadMethod AMethod);
```

Методы гарантируют, что к объекту (методу объекта) одновременно получит доступ только один поток. В качестве аргумента функции передается имя вызываемого метода. Разница между `Synchronize()` или `Queue()` заключается в порядке ожидания ответа от вызванного внешнего метода. Метод `Synchronize()` работает в *синхронном режиме*. Это означает, что поток, обратившись к внешнему методу, приостановит свою работу до тех пор, пока не получит ответа о том, что внешний метод выполнен. Метод `Queue()` отдает предпочтение *асинхронному режиму*, поток просто ставит задачу в очередь и, не дожидаясь ответа о ее выполнении, продолжает свою работу.

## Пример многопоточного приложения

В качестве примера многопоточного приложения рассмотрим задачу генерирования и отображения миниатюр большого количества графических файлов формата JPEG находящихся в определенном каталоге. Для хранения и вывода миниатюр стоит задействовать экземпляр класса `TImage`, а с ролью контейнера (на поверхности которого мы разместим компоненты-изображения `TImage` с миниатюрами) превосходно справится область с полосами прокрутки `TScrollBar` (рис. 3.1).

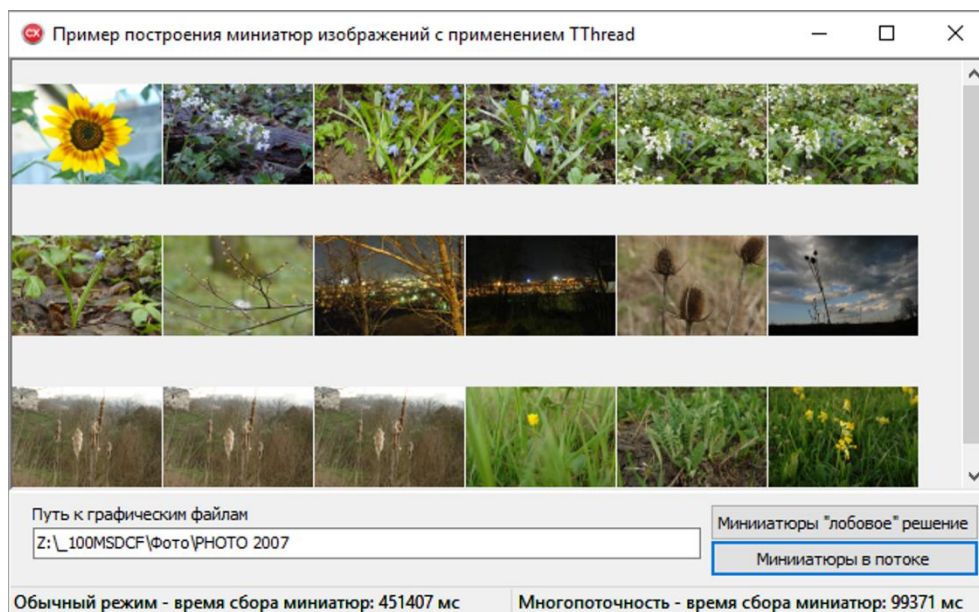


Рис. 3.1. Многопоточное приложение создания миниатюр

Для создания класса потока выберите пункт меню **File | New | Other**. На вкладке **New** найдите значок **ThreadObject** и нажмите кнопку **OK**. Назовите класс создаваемого потока `TThumbThread`. Вновь созданный модуль с шаблоном кода потока сохраните под именем `ThumbUnit.cpp`.

Обратимся к заголовочному файлу `ThumbUnit.h` только что созданного потока и добавим в определение класса `TThumbThread` два новых поля (листинг 3.3).

### Листинг 3.3. Определение класса потока в заголовочном файле `ThumbUnit.h`

```
#ifndef ThumbUnitH
#define ThumbUnitH
```

```
//-----
#include <System.Classes.hpp>
#include <Vcl.Imaging.jpeg.hpp>
//-----
class TThumbThread : public TThread
{
private:
protected:
    void __fastcall Execute();
public:
    __fastcall TThumbThread(bool CreateSuspended);

    TJPEGImage *JImage; //загруженный JPEG
    const TImage *Image; //ссылка на компонент с миниатюрой
};
//-----
#endif
```

Поле `JImage` предназначено для обслуживания целевого файла с картинкой в формате JPEG, именно из этой (возможно очень большой картинки) мы создадим миниатюру. Поле `Image` станет содержать ссылку на компонент `TImage` в котором будет отображена сделанная в рамках потока миниатюра.

Код создания и выполнения потока предложен в листинге 3.4.

#### Листинг 3.4. Создание и выполнение потока

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

#include "ThumbUnit.h"
#pragma package(smart_init)
//-----
__fastcall TThumbThread::TThumbThread(bool CreateSuspended)
    : TThread(CreateSuspended)
{
    JImage=new TJPEGImage(); //временный контейнер для JPEG рисунка
}
//-----
void __fastcall TThumbThread::Execute()
{
    JImage->DIBNeeded(); //преобразуем JPEG в Bitmap

    TBitmap* SourceBitmap=new TBitmap();
    SourceBitmap->Assign(JImage);

    double ScaleAmount=(double) Image->Width/SourceBitmap->Width;
    ScaleImage(SourceBitmap, Image->Picture->Bitmap, ScaleAmount);

    JImage->Free();
    SourceBitmap->Free();
    Terminate();
}
```

Весь процесс построения миниатюры сосредоточен в методе `Execute()` потока. Работа метода начинается с преобразования хранящегося в `JImage` изображения JPEG в несжатый формат BMP. Затем, рассчитав коэффициент масштабирования `ScaleAmount`, мы уменьшаем картинку и отправляем ее в `Image`, за эту операцию отвечает функция `ScaleImage()`.

Нам осталось разработать метод сбора сведений о файлах JPEG, находящихся в заданном каталоге и создания их миниатюр. Для этого напишем функцию `CreateThumbs()` с единственным параметром, содержащим имя исследуемого каталога (листинг 3.5).

**Листинг 3.5. Метод сбора сведений о файлах и создания миниатюр**

```
void __fastcall TForm1::CreateThumbs(UnicodeString DirName)
{
    TStringDynArray SDA=TDirectory::GetFiles(DirName, "*.jpg");

    for (int i = 0; i < SDA.High-1; i++) {

        TImage *Image=new TImage(ScrollBox1);
        Image->Parent=ScrollBox1;
        Image->Width=100; Image->Height=100;

        TThumbThread *ThumbThread=new TThumbThread(true);
        ThumbThread->JImage->LoadFromFile(SDA[i]);
        ThumbThread->Image=Image;
        ThumbThread->FreeOnTerminate=True;
        ThumbThread->Start();
    }
}
```

В рамках разработанного метода мы строим список файлов с расширением имени \*.jpg и последовательно, вызывая поток, создаем для каждого из них миниатюру.

## Синхронизация потоков

Одной из серьезных проблем, с которой сталкиваются разработчики многопоточных приложений, заключается в организации совместного доступа двух (и более) потоков к общему ресурсу. Конкурируя за обладание общим ресурсом, потоки из помощников превращаются в хулиганов, инициируя трудно выявляемые ошибки. Поэтому при реализации взаимодействующих потоков программист обязан решить задачу корректного доступа потоков к разделяемому ресурсу. Для исключения ошибок при подключении к общему ресурсу потокам необходимо синхронизировать свои действия так, чтобы ресурс принадлежал не более чем одному потоку одновременно. При попытке обращения к захваченному ресурсу остальных потоков они должны быть приостановлены до момента его освобождения.

Операционная система Windows — большая мастерица по синхронизации процессов и потоков. Для этого реализовано несколько механизмов управления процессами и потоками, при которых Диспетчер задач не выделяет ожидающим освобождения ресурса ни кванта лишнего времени. Синхронизация производится с помощью объектов синхронизации: событий, критических секций, мьютексов, семафоров и т. д.

Объект синхронизации обладает двумя устойчивыми состояниями: *сигнальным* (signaled) и *несигнальным* (nonsignaled). Сигнальное состояние разрешает приостановленному потоку приступить к работе, несигнальное состояние объекта синхронизации, напротив, предлагает ожидающему своей очереди потоку спать дальше.

### *Замечание*

Все рассматриваемые в этой главе объекты синхронизации описаны в модуле `SyncObjs`.

## Синхронизация событием *TEvent*

К числу наиболее простых средств синхронизации потоков относится синхронизация с помощью описанного в модуле `SyncObjs` события `TEvent`. Как и все объекты синхронизации

Windows, событие может находиться в одном из двух состояний: сигнальном и несигнальном. Синхронизируемые потоки способны выяснить, в каком расположении духа находится экземпляр класса `TEvent`, и приостанавливать или, наоборот, возобновлять свою работу.

Как всегда, в VCL, создание объекта начинается с вызова его конструктора:

```
TEvent(PSecurityAttributes EventAttributes, bool ManualReset,
      bool InitialState, const UnicodeString Name, bool UseCOMWait);
```

Первый параметр конструктора `EventAttributes` — это дескриптор безопасности, применяемый в Windows. Параметр `ManualReset` разрешает (значение `true`) переводить событие в несигнальное состояние вручную (это умет делать метод `ResetEvent`), иначе сброс произойдет только после старта потока, ожидающего этот объект синхронизации. Параметр `InitialState` определяет, в какое из состояний переходит событие сразу после создания: в сигнальное (`true`) или несигнальное (`false`). Последний параметр назначает имя создаваемому объекту синхронизации. Будьте внимательны, имя должно быть уникальным. Впрочем, имя понадобится, только если этим событием надо воспользоваться извне процесса, создавшего экземпляр `TEvent`.

При обсуждении конструктора события `TEvent` упоминался метод

```
void ResetEvent(void);
```

предназначенный для перевода объекта-события в несигнальный режим "вручную".

Обратная операция, включающая событие, обеспечивается методом

```
void SetEvent(void);
```

В составе класса `TEvent` предусмотрен метод ожидания:

```
TWaitResult WaitFor(const TTimeSpan &Timeout);
enum TWaitResult : unsigned char { wrSignaled, wrTimeout, wrAbandoned,
                                   wrError, wrIOCompletion };
```

Время ожидания перехода объекта синхронизации в сигнальное состояние определяется единственным аргументом метода `Timeout` и измеряется в миллисекундах. Если по истечению заданного таймаута ничего не произошло, то метод возвратит `wrTimeout`; если же за это время объект-событие перейдет в сигнальное состояние, то метод, не дожидаясь истечения таймаута, вернет `wrSignaled`.

Дескриптор объекта синхронизации `Event` доступен из свойства:

```
__property unsigned Handle;
```

Код последней ошибки, связанной с объектом, доступен из свойства:

```
__property int LastError;
```

Признаком того, что все в порядке, будет нулевое значение.

Как уже отмечалось в начале подраздела, синхронизация событием очень проста. Создайте новый проект и разместите на его главной форме три компонента: шкалу `TProgressBar *ProgressBar1`, метку `TLabel *Label1` и кнопку `TButton *Button1`.

Откройте заголовочный файл главной формы проекта (по умолчанию он называется `Unit1.h`) и в разделе `public` определите событие синхронизации

```
TEvent *Event;
```

Добавьте к проекту новый поток, присвоив ему имя `TProgressThread`. Полный код программного модуля потока вы найдете в листингах 3.6 и 3.7.

#### Листинг 3.6. Заголовочный файл `ThreadUnit1.h`

```
#ifndef ThreadUnitH
#define ThreadUnitH
//-----
```

```
#include <System.Classes.hpp>
//-----
class TProgressThread : public TThread
{
private:
protected:
    void __fastcall Execute();
    void __fastcall ChangeProgress(); //метод изменения состояния шкалы
public:
    __fastcall TProgressThread(bool CreateSuspended);
};
//-----
#endif
```

### Листинг 3.7. Модуль ThreadUnit1.cpp потока TProgressThread

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

#include "ThreadUnit.h"
#include "Unit1.h"
#pragma package(smart_init)
//-----
__fastcall TProgressThread::TProgressThread(bool CreateSuspended)
    : TThread(CreateSuspended)
{
}
//-----
void __fastcall TProgressThread::ChangeProgress() {
    if (Form1->ProgressBar1->Position+1<Form1->ProgressBar1->Max)
        {Form1->ProgressBar1->Position=Form1->ProgressBar1->Position+1;
        Form1->Label1->Caption=
            Format("%d",ARRAYOFCONST((Form1->ProgressBar1->Position))+"%");
        }
    else Form1->ProgressBar1->Position=0;
}
//-----
void __fastcall TProgressThread::Execute()
{
    Form1->Event=new TEvent(NULL,true,false,"Event1",false);
    do {
        if (Form1->Event->WaitFor(500)!=wrSignaled) {
            /*...
            какой-то блок операций потока
            ...*/
            Synchronize(ChangeProgress);
        }
    }
    while (!Terminated);
    free(Form1->Event);
}
```

Просмотрев листинги 3.6 и 3.7, вы увидите, что в рамках программного модуля создается событие синхронизации `TEvent *Event`. Создание события осуществляется в методе `Execute()` потока. Визуальной стороной работы потока становится изменение положения шкалы `ProgressBar1`, эта операция производится в методе `ChangeProgress()`.

Теперь переключим наше внимание на главную форму проекта. Нам понадобится описать два события. В событии `OnShow()`, вызываемом в момент вывода формы на экран, создается низкоприоритетный поток. Щелчок по кнопке формы вызывает диалог редактирования позиции шкалы `ProgressBar1`. Для того чтобы в момент ввода нового значения поток приостановил свою работу, мы переводим событие `Event` в сигнальное состояние. Завершив редактирование, сигнальный режим отключается (листинг 3.8).

**Листинг 3.8. Раздел реализации программного модуля главной формы**

```
void __fastcall TForm1::FormShow(TObject *Sender)
{
    ProgressBar1->Max=100;
    TProgressThread *PT=new TProgressThread(true);

    PT->Priority=tpLowest;
    PT->FreeOnTerminate=true;
    PT->Start();
}
//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Event->SetEvent(); //активировали событие синхронизации
    String s;

    if (InputQuery("Шкала", "Новое значение (0..99)",s)== true)
    {
        int x=StrToIntDef(s,ProgressBar1->Position);
        if (x>=0 && x<100) ProgressBar1->Position=x;
    }
    Event->ResetEvent(); //сброс события синхронизации
}
```

## Критическая секция *TCriticalSection*

*Критическая секция* обычно применяется для синхронизации параллельно выполняющихся потоков-близнецов (т. е. потоков, порожденных от одного класса и имеющих один и тот же код) или для защиты разных потоков, обращающихся к одному и тому же коду. Критическая секция — это специальным образом помеченный участок кода, из которого и производится обращение к разделяемому ресурсу. Если код в этой секции выполняется хотя бы одним потоком, то вход в критическую секцию для остальных потоков запрещается.

Создание критической секции начинается с обращения к методу `Create()`. Появившаяся на свет в результате вызова конструктора секция приступит к выполнению своих прямых функциональных обязанностей после обращения к методу

```
void Enter(void);
```

С этого момента объект синхронизации встает на защиту отмеченного участка кода и не допустит к нему ни один из потоков до тех пор, пока не будет вызван метод

```
void Leave(void);
```

извещающий все остальные потоки, что критический участок кода больше не используется.

Критическая секция может ждать своего выполнения бесконечно долго, поэтому при разработке многопоточного приложения с критической секцией программист отвечает за то, чтобы:

- ☐ при аварийном завершении потока критическая секция немедленно освобождалась;
- ☐ время, затрачиваемое конкурирующими процессами на вход в критическую секцию, было минимальным.

Обычно код входа в критическую секцию оформляется в защищенной от ошибок секции `__try...__finally`, что гарантирует освобождение заблокированного кода (листинг 3.9).

**Листинг 3.9. Использование критической секции**

```
TCriticalSection *CriticalSection;

void __fastcall TDemoThread::Execute()
{
    do
    {
        __try
        {
            CriticalSection->Enter();
            //действия потока
            Synchronize(/*вызов внешнего метода*/);
        }
        __finally
        {
            CriticalSection->Leave();
        }
    }
    while (!Terminated);
}
```

## Мьютекс *TMutex*

Как и критическая секция, объект синхронизации *мьютекс* (mutex) может принадлежать только одному потоку. Если мьютекс свободен, то он устанавливается в сигнальное состояние, если же он кем-то захвачен — переходит в несигнальный режим. При попытке получить доступ к "приватизированному" другим владельцем мьютексу поток получает отказ и переводится в состояние ожидания его освобождения. Таким образом, мьютекс обеспечивает взаимное исключение (mutual exclusion) доступа потоков к разделяемому ресурсу.

### *Замечание*

Разница между мьютексом и критической секцией заключается в том, что мьютекс может быть доступен не только внутренним потокам приложения, но и другим процессам, протекающим в операционной системе. Единственным условием для этого является наличие у мьютекса уникального имени. Таким образом, мьютекс представляет собой средство синхронизации не только потоков внутри приложения, но и всех процессов операционной системы.

В VCL мьютекс описывается классом *TMutex*. Существуют три варианта создания мьютекса. Простейший из них обеспечивает конструктор

```
TMutex(bool UseCOMWait);
```

В итоге на свет появляется неименованный объект синхронизации, доступный только внутренним потокам одного приложения. Единственный параметр конструктора определяет способность мьютекса взаимодействовать с компонентами STA (single-threaded apartment) COM.

Наиболее востребованной версией конструктора считается

```
TMutex(PSecurityAttributes MutexAttributes, bool InitialOwner,
const UnicodeString Name, bool UseCOMWait);
```

Первый параметр метода представляет собой указатель на структуру атрибутов защиты `pSecurityAttributes = ^TSecurityAttributes`. Параметр определяет атрибуты безопасности, в простейшем случае в качестве этого аргумента допускается передача неопределенного указателя `NULL`. Параметр `InitialOwner` указывает, станет ли поток, создающий мьютекс, его владельцем. Для этого параметру передавайте значение `true`. Последний аргумент

содержит имя мьютекса. Если мьютекс не планируется разделять с внешними процессами, то метод удовлетворится пустым аргументом.

Третий конструктор открывает уже существующий мьютекс, обращаясь к нему по имени Name.

```
TMutex(unsigned DesiredAccess, bool InheritHandle,
        const UnicodeString Name, bool UseCOMWait);
```

Порядок доступа зависит от первого параметра, в который направляется комбинация флагов доступа: MUTEX\_ALL\_ACCESS, MUTEX\_MODIFY\_STATE, SYNCHRONIZE, \_DELETE, READ\_CONTROL, WRITE\_DAC, WRITE\_OWNER.

Для управления мьютексом достаточно пары методов:

```
Acquire();
Release();
```

Первая из функций захватывает мьютекс, вторая — освобождает.

### **Внимание!**

Не забывайте освобождать мьютекс, иначе первый поток (захватив мьютекс) станет единственным владельцем критического участка кода и остановит работу оставшихся потоков по крайней мере до момента своего физического уничтожения методом `Terminate()`. В этом случае мьютекс перейдет в разряд покинутых, и операционная система передаст права на него другому потоку.

## **Семафор *TSemaphore***

*Семафор* (semaphore) — логическое развитие мьютекса. Как вы помните, объект взаимного исключения мьютекс позволял владеть собой только одному потоку. В отличие от него семафор может находиться в совместном использовании сразу нескольких потоков, но при этом он умеет управлять количеством этих потоков — выступает в роли ограничителя их "популяции". Для этого он снабжен счетчиком, определяющим максимальное число потоков, которым разрешено выполняться совместно. С запуском каждого нового потока значение счетчика уменьшается на единицу, и когда оно достигнет нуля, семафор перейдет в несигнальное состояние. Это означает, что последующие потоки приостанавливаются и будут дожидаться момента, пока какой-нибудь из потоков не освободит семафор. Характерным примером области применения семафоров может послужить программное обеспечение какого-нибудь аппаратного устройства, с которым может работать только ограниченное количество потоков, или реализация веб-сервера (сервера баз данных), допускающего одновременный доступ ограниченного числа посетителей.

Семафор обладает сразу тремя перегружаемыми версиями конструктора:

```
TSemaphore(bool UseCOMWait);
TSemaphore(PSecurityAttributes SemaphoreAttributes,
            int AInitialCount, int AMaximumCount,
            const UnicodeString Name, bool UseCOMWait);
TSemaphore(unsigned DesiredAccess, bool InheritHandle,
            const UnicodeString Name, bool UseCOMWait);
```

Первый конструктор наиболее прост, он создает незатейливый семафор, по своим характеристикам почти не отличающийся от мьютекса. Наиболее полезна вторая версия конструктора, именно она формирует полноценный объект синхронизации. Третий конструктор предназначен не для создания, а для получения доступа к семафору (обычно создаваемому в другом приложении). Параметры конструкторов представлены в табл. 3.3.

*Таблица 3.3. Параметры конструктора семафора*

| Параметр            | Описание                                                                            |
|---------------------|-------------------------------------------------------------------------------------|
| SemaphoreAttributes | Ссылка на структуру TSecurityAttributes с атрибутами безопасности, допускается NULL |

|                            |                                                                                                                                                                                                                                                    |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>InitialCount</code>  | Начальное значение числа доступных потоков и максимальное значения счетчика семафора. Когда число свободных потоков подходит к нулю семафор приостанавливает доступ к ресурсу                                                                      |
| <code>AMaximumCount</code> |                                                                                                                                                                                                                                                    |
| <code>Name</code>          | Чувствительное к регистру символов уникальное имя семафора                                                                                                                                                                                         |
| <code>UseCOMWait</code>    | Параметр устанавливается в <code>true</code> если блокируемый ресурс должен поддерживать технологию STA COM                                                                                                                                        |
| <code>DesiredAccess</code> | Уточняет особенности доступа к семафору: <code>SEMAPHORE_ALL_ACCESS</code> , <code>SEMAPHORE_MODIFY_STATE</code> , <code>SYNCHRONIZE</code> , <code>_DELETE</code> , <code>READ_CONTROL</code> , <code>WRITE_DAC</code> , <code>WRITE_OWNER</code> |
| <code>InheritHandle</code> | Разрешает дочерним процессам наследовать дескриптор                                                                                                                                                                                                |

Захват семафора осуществляется после обращения потока к методу

```
virtual void Acquire(void);
```

Внутренний счетчик семафора получает приращение, и если он превысил ограничение на максимальное число потоков, доступ к ресурсу временно блокируется. Блокировка снимается только после того, как другой поток освободит семафор с помощью метода

```
virtual void Release(void);
```

```
HIDEBASE int Release(int AReleaseCount);
```

Несмотря на то, что семафор — достаточно сложный объект синхронизации, работа с ним не вызовет особых затруднений. Листинг 3.10 отражает процесс создания экземпляра семафора.

#### Листинг 3.10. Создание семафора

```
TSemaphore *Semaphore;
//-----
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    int AMaximumCount=TThread::ProcessorCount*8;
    Semaphore=new TSemaphore(NULL,
        AMaximumCount-1,AMaximumCount,"DemoSemaphor",false);
    ...
}
//-----
```

Обратите внимание на то, как в коде определяется максимальное число потоков, контролируемое семафором. Мы узнаем число процессоров в компьютере и, исходя из этого, ограничиваем количество потоков. Напомню, что не рекомендуется создавать более 16 потоков на одно ядро процессора.

Листинг 3.11 демонстрирует работу с семафором потока.

#### Листинг 3.11. Применение семафора

```
void __fastcall TDemoThread::Execute()
{
    Semaphore->Acquire();
    //работа потока с ресурсом (данными)
    Semaphore->Release();
}
```

#### **Внимание!**

Будьте аккуратны при использовании семафоров в зацикленных потоках (потоках, в методе `Execute()` которых используются операторы цикла). В таких случаях высока вероятность блокирования потока.

## Библиотека параллельных вычислений

В вашем устройстве установлен многоядерный процессор? И вы предполагаете, что ваши приложения используют его на полную катушку? Если вы не уверены в последнем, то добро пожаловать в библиотеку параллельных вычислений Parallel Programming Library (PPL). Включив PPL в свои проекты, вы сможете существенно повысить производительность приложений. Еще одна отрадная черта современной PPL заключается в том, что она с одинаковым успехом поддерживает как Windows, так и OS X, iOS и Android.

Когда можно применять параллельные алгоритмы? Если дать ответ, что параллельные алгоритмы необходимы в любом случае, когда требуется повысить скорость работы программы ответ окажется неполным.

Различают два способа повышения производительности системы за счет распараллеливания:

1. Распараллеливание по задачам.
2. Распараллеливание по данным.

В первом случае один поток выполняет одну часть задачи, а второй — другую. Во втором случае каждый поток выполняет одну и ту же задачу, но с разными данными.

Параллельные алгоритмы могут принести пользу только при условии, что программист сможет:

- ☐ выявить в своей программе нити вычисления, которые возможно выполнять параллельно;
- ☐ согласовать доступ к разделяемым данным с параллелизмом вычислений.

Основной код библиотеки PPL сосредоточен в модуле System.Threading. Здесь вы обнаружите все ключевые классы библиотеки, в том числе рассматриваемые в этой главе TParallel и TTask.

## Распараллеливание, класс TParallel

Класс TParallel специализируется на распараллеливании задач между ядрами процессора. Для этого в его арсенале имеется две категории методов:

1. Методы For позволяют выполнять две и более итерации for-цикла (цикла со счетчиком) одновременно, а не последовательно как это происходит в обычном режиме.
2. Методы Join группирует процедуры так, чтобы они могли выполняться в параллельных потоках.

## Оптимизация итераций циклов

В тех случаях, когда программная логика for-цикла вашего приложения построена так, что итерации цикла решают независимые друг от друга задачи, либо задачи последовательность выполнения, которых не имеет значения вполне логичным решением может стать распределение итераций между ядрами процессора. В результате одновременно могут быть выполнены две или более (все зависит от числа ядер процессора и их загрузки) итерации.

Для того, чтобы научиться распараллеливать итерации стоит познакомиться с ключевым методом класса TParallel:

```
static TLoopResult fastcall For(System::TObject* Sender,
    int AStride, int ALowInclusive, int AHighInclusive,
    TIteratorStateEvent AIteratorEvent,
    TThreadPool* APool)/* overload */;
```

Сразу оговоримся, что на самом деле в модуле `System.Threading` описано больше трех десятков перегружаемых версий метода. Они отличаются друг от друга перечнем параметров и разрядностью. Предложенный выше метод наиболее показательный, так как в нем представлен максимум аргументов.

#### **Замечание**

На самом деле все многочисленные версии метода `TParallel::For` в итоге обращаются к инкапсулированному в классе методу `ForWorker`.

Обилие аргументов на первых порах может несколько смутить читателя, но в утешение могу сказать, что все вместе они вряд ли понадобятся. На самом деле метод является некоторым подобием швейцарского ножа, подходящего на все случаи жизни, и вполне вероятно, что некоторые “лезвия” функции вам не пригодятся.

Рассмотрим обязательные параметры функции. В первую очередь это параметр `ALeratorEvent` описывающий основное событие, выполняемое в рамках итерации. Другими словами, это строки кода, подлежащие многократному запуску в теле цикла и именно их выполнение мы хотим распределить между ядрами процессора. В простейшем случае указанный код оформляется в виде метода

```
void __fastcall IteratorEvent(TObject *Sender,int AIndex);
```

Как вы уже наверняка догадались параметр `AIndex` процедуры соответствует значению счетчика.

Еще одна пара обязательных параметров `ALowInclusive` и `AHighExclusive` определяют начальное и конечное значение счетчика цикла, например, для реализации цикла `for (int i = 0; i < 9; i++)` значения аргументов должны соответствовать 0 и 9.

Остальные параметры факультативны, их краткий обзор вы найдете в таблице 3.4.

*Таблица 3.4. Необязательные параметры функции `ForWorker`*

| Параметр             | Описание                                                                                                                                                     |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Sender</code>  | Ссылка на экземпляр объекта, выступающего инициатором операции                                                                                               |
| <code>Astride</code> | Определяет шаг приращения счетчика цикла, если аргумент явно не назначен, то полагается, что шаг равен 1                                                     |
| <code>APool</code>   | Определяет минимальное и максимальное число параллельных потоков, выполняемых на ядрах процессора, максимальное значение не должно превышать количество ядер |

Вне зависимости от задействованных входных параметров возвращаемый результат функции всегда один и тот же — структура `TLoopResult`. Благодаря ней мы можем судить об успешности выполнения данной итерации, в этом случае поле `Completed` окажется установлено в состояние `true`. Кроме того, благодаря универсальному полю `LowestBreakIteration` (тип данных `Variant`) и методу `TLoopState::Break` мы узнаем значение индекса выполняемый в данный момент итерации цикла. Если же метод `TLoopState::Break` не вызывался, то в `LowestBreakIteration` окажется неопределенность `NULL`.

#### **Замечание**

При параллельном выполнении несколько итераций благодаря `TLoopResult`. `LowestBreakIteration` возвратит индекс самой ранней из работающих итераций.

### **Пример параллельного выполнения итераций цикла**

Реализуем на практике пример, демонстрирующий преимущество параллельных вычислений над последовательными. Для этого решим задачу подсчета суммы элементов двумерного массива размерностью  $M \times M$  двумя способами — традиционным последовательным и с применением библиотеки параллельных вычислений.

Разместите на форме проекта многострочный редактор Memo1 и пару кнопок. К строке uses подключите ряд модулей:

```
#include <System.Diagnostics.hpp>
#include <System.Threading.hpp>
#include <System.SyncObjs.hpp>.
```

Объявите глобальную константу, определяющую размерность массива, переменную массив и переменную хранящую сумму элементов массива:

```
const M=5000;
int a[M][M];
__int64 ArraySum;
```

Реализуем отдельную функцию (листинг 3.12) предназначенную для подсчета суммы элементов массива в отдельной строке массива.

#### Листинг 3.12. Функция вычисления суммы элементов массива в строке row

```
int rowsum(int row)
{int sum=0;
for (int j = 0; j < M; j++ )
    {sum+=a[row][j];
    };
return sum;}
```

В дальнейшем мы воспользуемся помощью RowSum как в последовательном, так и в параллельном способах подсчета. Что касается исходного кода (листинг 3.13) с обычным решением задачи суммирования элементов массива, то полагаю особых сложностей в его понимании у читателя не возникнет.

#### Листинг 3.13. Обычный последовательный график

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TStopwatch sw= TStopwatch::Create();
sw.Start();
//----- Расчет суммы -----
ArraySum=0;
for (int i = 0; i < M; i++)
    ArraySum+=rowsum(i);
//-----
sw.Stop();
Memo1->Lines->Add(Format("Последовательный график.
                        Результат %d получен за %d мс",
                        ARRAYOFCOST (ArraySum,sw.ElapsedMilliseconds)));
}
```

Единственная особенность рассматриваемого примера заключается в задействовании в нем диагностического класса TStopWatch (модуль System.Diagnostics.hpp), позволяющего с точностью до миллисекунды измерить время, затраченное на подсчеты. Получив желаемый результат выводим его в многострочном редакторе.

А теперь решим ту же задачу, но на этот раз за счет библиотеки параллельных вычислений, обратившись за помощью к методу For. Для этой цели сначала создадим событие итерации и объявим его как метод формы (листинг 3.14).

#### Листинг 3.14. Событие итерации

```
void __fastcall TForm1::IteratorEvent(TObject *Sender,int AIndex)
{
```

```
TInterlocked::Add(ArraySum, rowsum(AIndex));
}
```

Обратите внимание что здесь мы воспользовались услугами метода Add() класса TInterlocked. Подробно о возможностях мы TInterlocked поговорим буквально через пару страниц, а пока воспринимайте класс, как “волшебный”, а его метод рассматривайте как специалиста по суммированию значения переменной ArraySum и результата возвращаемый функцией rowsum().

Описав событие итерации опишем код щелчка по кнопке “распараллеливания” (листинг 3.15).

#### Листинг 3.15. Параллельный подсчет с помощью TParallel::For()

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    TStopwatch sw= TStopwatch::Create();
    sw.Start();
    //----- Расчет суммы -----
    ArraySum=0;
    TParallel::For(NULL, 0,M-1, IteratorEvent);
    //-----
    sw.Stop();
    Memo1->Lines->Add(
        Format("Параллельный график. Результат %d получен за %d мс",
            ARRAYOFCNST((ArraySum,sw.ElapsedMilliseconds))));
}
```

Результат применения метода For весьма впечатляет, свидетельством тому может служить предложенный на рисунке 3.2 экранный снимок мобильного приложения, выполняющегося под управлением Android. Обратите внимание на то, что благодаря распараллеливанию итераций циклов мы получили более чем двойное преимущество в скорости выполнения задачи, в приложениях для Windows и OS X преимущество окажется еще более значительным.

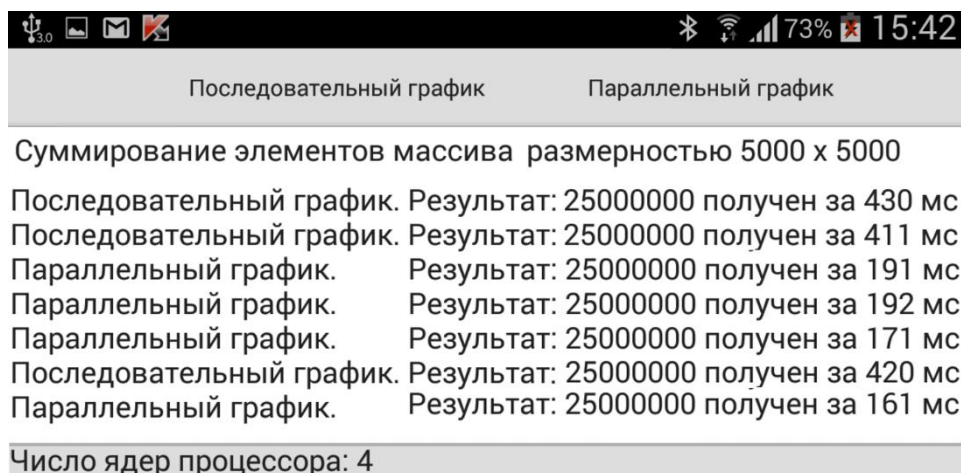


Рис. 3.2. Сравнительные результаты временных затрат на последовательный и параллельный графики

## Группировка процедур

Второй специализацией класса TParallel является группировка 2-х и более процедур с их последующим параллельным выполнением. Для этой цели в составе класса объявлено несколько перегружаемых версий функции класса Join(). Из них в качестве наиболее показательной выделим функцию

```
static __di_ITask __fastcall Join(System::TObject* Sender,
    System::Classes::TNotifyEvent *AEvents,
    const int AEvents_High)/* overload */;
```

позволяющую выполнить в параллельном режиме несколько событий и функцию

```
static _di_ITask __fastcall Join(
    System::Sysutils::_di_TProc const *AProcs,
    const int AProcs_High)/* overload */;
```

способную одновременно запустить несколько процедур. Все версии функции возвращают интерфейс задачи `_di_ITask`, а если говорить точнее интерфейса класса `Delphi TTask` (табл. 3.2).

## Класс–задача, *TTask*

Для выполнения атомарной задачи в одном из параллельных потоков проще всего задействовать класс `TTask` — интеллектуальную оболочку для функции (метода, события) которая подлежит выполнению в контексте отдельного потока.

Разработчиками библиотеки `PPL` предусмотрено три подхода к формированию задачи.

1. Традиционный способ создания экземпляра задачи с помощью конструктора. Появившаяся таким образом задача изначально приостановлена и ожидает от программиста явной команды на старт.
2. Создание задачи с немедленным запуском потока с помощью метода `Run()`.
3. Создание задачи способной в последующем возвращать в программу результат выполнения функции при посредничестве метода `Future()`. Как и в первом случае поток создается в приостановленном виде и ожидает команды на запуск.

Первый, из перечисленных способов создания задачи реализуется с помощью перегружаемого конструктора, самый простой из конструкторов выглядит следующим образом

```
static _di_ITask __fastcall Create(
    const System::Sysutils::_di_TProc Proc)/* overload */;
```

Метод позволит создать задачу на основе процедуры `Proc` и возвратит интерфейс задачи в формате `Delphi`.

### *Замечание*

Отметим, что основные методы по управлению задачей принадлежат не только экземпляру класса `TTask`, но и возвращаемому конструктором интерфейсу.

Задача, созданная любой версией конструктора класса `TTask` по умолчанию приостановлена. Для ее запуска следует задействовать упомянутый в таблице 2.2 метод `Start`. Такой подход удобен в ситуации, когда для запуска задачи должны быть соблюдены какие-то дополнительные условия. Если же вы планируете инициировать немедленное выполнение задачи сразу же после ее появления на свет, то вместо конструктора следует воспользоваться наиболее подходящей версией метода класса `Run`. Например, в случае, когда задача реализуется на основе функции, остановите выбор на

```
static _di_ITask __fastcall Run(
    const System::Sysutils::_di_TProc Func)/* overload */;
```

Если читатель заглянет в исходный код метода, то увидит что это просто надстройка над конструктором в которой сразу после вызова метода `Create` вызывается метод `Start`.

Если для формирования задачи вы воспользуетесь первым или вторым подходом, то вызвав конструктор либо метод `Run` в ответ получите интерфейс задачи — `_di_ITask`. Названный интерфейс позволит программисту применить к задаче наиболее востребованные операции (табл. 3.5).

*Таблица 3.5. Управление задачей с помощью интерфейса `_di_ITask`*

| Метод                                     | Описание                                             |
|-------------------------------------------|------------------------------------------------------|
| <code>bool Wait(unsigned Timeout =</code> | Определяет таймаут, отводимый на ожидание пока поток |

|                                                                                                                              |                                                                                                                                                                           |
|------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| (unsigned) (0xffffffff)) = 0 /* overload */;<br>bool Wait(const System::Timespan::TTimeSpan<br>&Timeout) = 0 /* overload */; | выполнит задачу. Если задача уже выполнена, то не ожидая завершения таймаута функция возвратит true. Если тайм-аут истечет до завершения задачи, функция возвратит false. |
| void Cancel(void) = 0 ;                                                                                                      | Отмена задачи                                                                                                                                                             |
| _di_ITask Start(void) = 0 ;                                                                                                  | Старт задачи с предварительной проверкой, что она еще не выполнена. В последнем случае генерируется Ис EInvalidOperation                                                  |
| TTaskStatus GetStatus(void) = 0 ;<br>__property TTaskStatus Status =<br>{read=GetStatus};                                    | Возвращает сведения о статусе задачи<br>(Created, WaitingToRun, Running, Completed, WaitingForChildren, Canceled, Exception);                                             |

В условиях, когда задача должна вернуть некоторое значение, которое позднее будет востребовано при выполнении основного кода программы следует задействовать возможности метода Future<T>. Предусмотрено несколько мудреных версий этого метода, например

```
inline System::DelphiInterface<IFuture__1<T> >
    __fastcall TTask::Future(System::TObject* Sender,
        T __fastcall (__closure *Event)(System::TObject* Sender));
```

Ключевой элемент представленной синтаксической конструкции — ссылка на метод Event. В коде данного метода и должна решаться задача.

Общая черта всех перегружаемых методов Future<T> заключается в том, что благодаря тому, что в качестве носителя кода встраиваемого в контекст потока выбрана обобщенная функция разработчик может заставить задачу вернуть в программу значение любого типа <T>.

#### **Замечание**

Главная особенность задач, запущенных при посредничестве Future<T> заключается в возможности запросить результат выполнения встраиваемой в задачу функции из любой точки вашей программы спустя любое время после выполнения задачи.

В результате выполнения функции класса мы получим усовершенствованный интерфейс IFuture\_\_1<T> (табл. 3.6).

**Таблица 3.6.** Управление задачей с помощью интерфейса IFuture\_\_1<T>

| Метод/свойство | Описание                                   |
|----------------|--------------------------------------------|
| Start          | Запуск функции в потоке                    |
| GetValue       | Возвратит результат выполнения функции <T> |
| Value          |                                            |

Рассмотри порядок работы с методом Future() на простом примере. Допустим в программе должно параллельно выполняться несколько задач в результате выполнения которых возвращается целочисленное значение (листинг 3.16).

#### **Листинг 3.16.** Применение метода TTask::Future()

```
//-----
int __fastcall TForm1::FutureHandler(TObject* Sender)
{
    /* код выполняемой задачи,
       для упрощения примера мы просто возвратим случайное значение*/
    return random(100);
};
```

```
//-----

//-----

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Memo1->Clear();
    const M=10;
    //массив задач F где M – число элементов в массиве
    DelphiInterface<IFuture__1<int>> >
        *F=new DelphiInterface<IFuture__1<int>> >[M];

    //---- 1-й цикл - Создание и запуск задач ----
    for (int i = 0; i < M; i++)
    {
        F[i]=TTask::Future<int>(0, this->FutureHandler);
        F[i]->Start();
    }

    //-----
    //...
    //---- 2-й цикл - Сбор результатов -----
    for (int i = 0; i < M; i++)
    {
        if (F[i]->Wait(30))
        {
            int r=F[i]->Value;
            Memo1->Lines->Add(r); //вывод результатов в Memo1
        }
    }
}
```

В нашем примере предполагается одновременная работа  $M$  задач. Для хранения их интерфейсов по щелчку по кнопке `Button1` создается динамический массив `F`, каждому элементу которого будет соответствовать интерфейс соответствующей задачи. Затем, в первом цикле события мы создаем и запускаем задачи. Во втором цикле листинга мы последовательно обращаемся ко всем интерфейсам и считываем хранящийся в свойстве `Value` результат. Обратите внимание на то, что обращение к интерфейсу осуществляется при посредничестве метода ожидания `Wait` с небольшим таймаутом (в нашем примере 30 мс). Благодаря таймауту мы даем задаче шанс завершить свое выполнение перед опросом.

## Ожидание и синхронизация

В той ситуации, когда запуск какой либо операции может быть осуществлен только после выполнения другой задачи (или нескольких задач) помощь окажет метод `WaitForAll()`, переводящий задачу в режим ожидания. У метода несколько перегружаемых версий, наиболее показательная

```
bool WaitForAll(_di_ITask const *Tasks, const int Tasks_High,
               const System::Timespan::TTimeSpan &Timeout)/* overload */;
```

Первый параметр `Tasks` содержит массив задач, от корректного выполнения которых зависит запуск ожидающей задачи. Второй параметр описывает размерность массива. Заключительный параметр `Timeout` назначает время таймаута (в миллисекундах) отводимого на выполнение всех задач из массива `Tasks`. После успешного запуска всех задач метод возвратит `true`, в противном случае (например, если таймаут истечет раньше, чем выполнятся все задачи), то функция возвратит `false`.

У метода `WaitForAll()` имеется коллега со схожим функционалом

```
int WaitForAny(_di_ITask const *Tasks, const int Tasks_High,
               const System::Timespan::TTimeSpan &Timeout)/* overload */;
```

Особенность данной функции в том, что она осуществит запуск ожидающего метода при условии, что выполнится хотя бы одна из занесенных в массив `Tasks` задач (а не все, как в предыдущем случае).

## Блокирование операций, класс `TInterlocked`

Описанный в модуле `System.SyncObjs` класс `TInterlocked` специализируется на осуществлении ряда наиболее распространенных атомарных операций с переменными, которые в результате выполнения параллельных вычислений могут оказаться доступными для нескольких потоков одновременно. Благодаря `TInterlocked` производится блокирование разделяемой переменной в момент обращения к ней потока, и разблокирование только после того, как захвативший ее поток прекратит с ней работу. Очередной поток не получит доступа к переменной до тех пор, пока она не будет освобождена предыдущим “захватчиком”. Таким образом исключается вероятный конфликт потоков.

Все методы класса `TInterlocked` статические, так что вам не придется создавать экземпляра класса для того, чтобы ими воспользоваться. Описание наиболее показательных функций класса вы обнаружите в таблице 3.7.

Таблица 3.7. Основные функции класса `TInterlocked`

| Метод класса                                                                                      | Описание                                                                                                                                                                                                                                                                                                                                   |
|---------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int Increment(int &amp;Target);</code>                                                      | Приращение переменной <code>Target</code> на 1                                                                                                                                                                                                                                                                                             |
| <code>int Decrement(int &amp;Target);</code>                                                      | Уменьшение переменной <code>Target</code> на 1                                                                                                                                                                                                                                                                                             |
| <code>int Add(int &amp;Target, int Increment);</code>                                             | Прибавляет к переменной <code>Target</code> значение <code>Increment</code>                                                                                                                                                                                                                                                                |
| <code>void * Exchange(void * &amp;Target, void * Value);</code>                                   | Меняет местами значения <code>Target</code> и <code>Value</code>                                                                                                                                                                                                                                                                           |
| <code>void * CompareExchange(<br/>void * &amp;Target, void * Value,<br/>void * Comparand);</code> | Сравнивает значения и если значение <code>Value</code> равно значению <code>Comparand</code> , то это значение передается в <code>Target</code>                                                                                                                                                                                            |
| <code>bool BitTestAndSet(int &amp;Target, TBitOffset BitOffset);</code>                           | Проверяет в переменной <code>Target</code> бит с порядковым номером <code>BitOffset</code> на наличие в нем 1. Если это так, то метод возвратит <code>true</code> , если же в бите находился 0 — возвращается <code>false</code> . Вне зависимости от результата проверки после выполнения функции контролируемый бит будет установлен в 1 |
| <code>bool BitTestAndClear(int &amp;Target, TBitOffset BitOffset);</code>                         | Поведение функции аналогично <code>BitTestAndSet()</code> за исключением того, что после проверки состояния бита в нем окажется 0                                                                                                                                                                                                          |

## 4. Динамически подключаемые библиотеки

*Вид занятия – практическая работа*

*Цель – разработка DLL*

*Продолжительность – 2 часа*

При освоении Windows вы, наверное, уже сталкивались с *динамически подключаемыми библиотеками* (Dynamic Link Library, DLL). Каждая из библиотек представляет собой отдельный программный модуль, содержащий код, данные или ресурсы (или все вместе) и допускающий совместное их использование несколькими приложениями. Например, библиотека kernel32.dll отвечает за управление памятью и инициализацию процессов, библиотека comdlg32.dll содержит все стандартные диалоговые окна, user32.dll поддерживает пользовательский интерфейс, gdi32.dll обслуживает графику.

Для того чтобы понять принцип работы динамических библиотек, надо знать ответ на вопрос: что происходит внутри операционной системы при запуске приложения (а если говорить еще точнее — процесса)? В 32-разрядных версиях Windows стартовый процесс в свое распоряжение получит отдельное адресное пространство размером 4 Гбайт виртуальной памяти, в 64-разрядных речь ведется об астрономическом размере в 8 Тбайт. Каждый из процессов наивно полагает, что все адресное пространство принадлежит только ему. Однако более просвещенному программисту надо понимать, что каждому выполняющемуся в Windows процессу невозможно предоставить по  $2^{32}$ , а тем более и  $2^{64}$  байт быстрой оперативной памяти. Поэтому для процесса распределяется участок памяти не только из оперативной памяти, но и из системного страничного файла (файла подкачки).

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

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

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

### Создание проекта DLL

Создание проекта динамически подключаемой библиотеки начинается с обращения к традиционному пункту меню **File | New | Other**, затем на странице **New** диалогового окна **New Items** следует выбрать пункт **Dynamic-link Library**. В появившемся на экране окне

**New Dynamic-link Library** все оставляем (рис. 4.1) без изменений и нажимаем на кнопку OK.

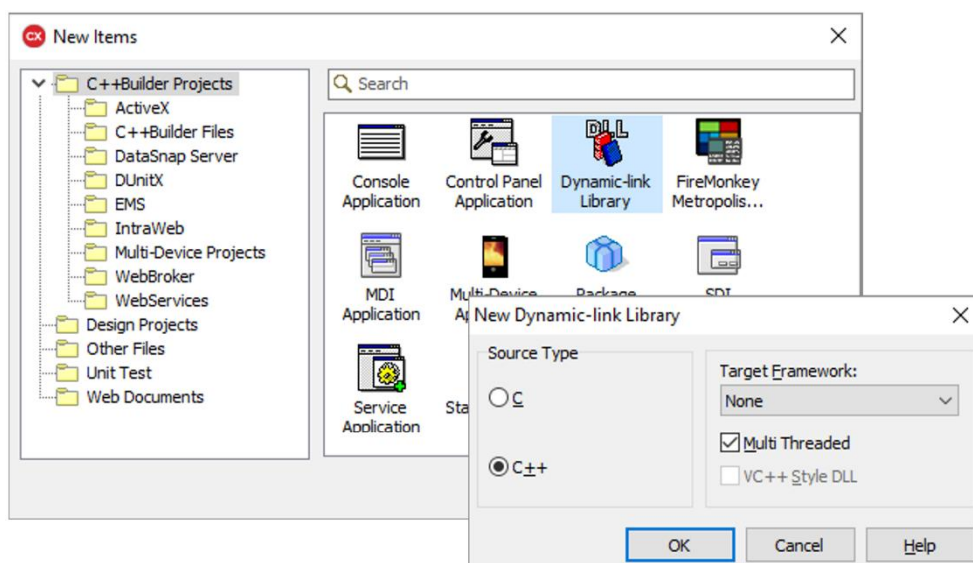


Рис. 4.1. Создание проекта DLL

В ответ на наши действия C++ Builder создаст шаблон кода с единственной функцией `_libmain()` — это и есть визитная карточка проекта библиотеки DLL. Сохраните код в отдельном каталоге под именем `mathlib.cbproj`. Как вы наверняка догадались по названию, наша библиотека будет связана с математическими расчетами. Прообраз нашей первой "тренировочной" библиотеки представлен в листинге 4.1.

#### Листинг 4.1. Шаблон кода библиотеки

```
#pragma hdrstop
#pragma argsused

extern "C" int _libmain(unsigned long reason)
{
    return 1;
}
```

Создавая DLL нам следует помнить о том, что если хотя бы одна из экспортируемых функций DLL требует данные строкового типа `String` (в качестве параметра или в виде возвращаемого значения), то к проекту библиотеки следует добавить модуль `MEMMGR.LIB`. Такая необходимость обусловлена особенностью хранения в памяти фирменного для библиотеки VCL строкового типа данных `String`. В свою очередь, для обслуживания программного модуля `MEMMGR.LIB` требуется еще один помощник — динамическая библиотека `borlndmm.dll` (ее вы обнаружите в каталогах `\bin` и `\bin64`), которую следует распространять вместе с проектом.

#### Замечание

При желании можно отказаться от услуг библиотеки `BORLNDMM.DLL`, но в этом случае текстовые данные следует передавать при помощи `char*` или `ShortString`.

## Объявление и экспорт функций в DLL

Для того чтобы стороннее приложение смогло воспользоваться описанными в библиотеке функциями, эти функции должны быть помечены, как экспортируемые. Синтаксис объявления экспортируемых библиотекой методов имеет ряд небольших отличий от уже привычного описания. Четко выделяются следующие особенности:

- ❑ между экспортирующей функцией библиотекой и выступающим в роли импортера приложением должно быть заключено *соглашение о вызовах*, оно необходимо для обеспечения корректной передачи параметров и возврата результата выполнения функции;
- ❑ объявление экспортируемой функции с ключевым словом `export` и использование секции `exports` для указания имен экспортируемых функций.

## Пример экспорта функций

Продолжим работу с проектом библиотеки `mathlib`. Раз наша динамическая библиотека предназначена для решения математических задач, то первой экспортируемой функцией станет функция, суммирующая пару чисел, назовем ее `add()`. Объявите заголовок и тело функции так, как это представлено в листинге 4.2.

**Листинг 4.2. Исходный код библиотеки `mathlib` с функцией `add()`**

```
#pragma hdrstop
#pragma argsused

extern "C" int _libmain(unsigned long reason)
{
    return 1;
}

extern "C" float __declspec(dllexport) add(float x, float y);
//----- Функция суммирования двух значений -----
float add(float x, float y)
{
    return x+y;
}
//-----
```

Собственно код функции `add()` суммирующий два действительных числа никакой сложности не представляет. Гораздо интереснее строка кода

```
extern "C" float __declspec(dllexport) add(float x, float y);
```

отвечающая за описание прототипа функции экспортируемой из динамической библиотеки.

Здесь модификатор `extern` с ключевым словом `"C"` указывает на то, что для обращения к функции должно использоваться соглашение для языка C++. Модификаторы `__declspec(dllexport)` говорят о том, что следующая далее функция подлежит экспорту и может быть доступна для других приложений и библиотек.

### **Внимание!**

Для компиляции библиотеки воспользуйтесь комбинацией клавиш `<Ctrl>+F9`. Не пытайтесь запустить библиотеку на выполнение, ведь это просто хранилище функций. Для проверки работоспособности библиотеки необходимо разработать полноценное приложение.

### **Внимание!**

Обратите внимание, что после компиляции библиотеки в результирующем каталоге кроме файла `dll` появился файл с расширением имени `lib`. Этот файл нам пригодится при построении проекта со статической загрузкой библиотеки.

## Пример хранения форм в библиотеке

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

Доступ приложения к хранящимся во внешнем модуле формам существенно отличается от обращения к форме из проекта. Здесь команда `Form1->Show()` к сожалению не сработает. Для взаимодействия с хранящейся в DLL формой придется создавать специальные функции-оболочки.

Достойным развитием математической библиотеки будет создание формы-калькулятора, позволяющей нам осуществлять любые расчеты. Создайте новый проект DLL и сохраните его под именем `calcklib`. Добавьте к библиотеке модуль формы, для этого обратитесь к меню **File | New | VCL Form – C++ Builder**. Переименуйте форму в `frmCalculator`, а соответствующий ей программный модуль сохраните под именем `calculator.cpp`.

По объективным причинам мы не станем обсуждать непосредственно код калькулятора (вы его найдете среди примеров к книге), а упомянем лишь самое важное с точки зрения реализации нашей библиотеки. На форме калькулятора (рис. 4.2) помимо других элементов управления расположены:

- ☐ метка `TLabel *lbRes`, в которой отображаются результаты расчетов;
- ☐ кнопка `TBitBtn *btnOK` со свойством `Kind`, установленным в состояние `bkOK`, при нажатии кнопки форма закрывается, возвращая модальный результат `idOK`;
- ☐ кнопка `TBitBtn *btnCancel` со свойством `Kind`, установленным в состояние `bkCancel`, при нажатии кнопки форма закрывается, возвращая модальный результат `idCancel`.

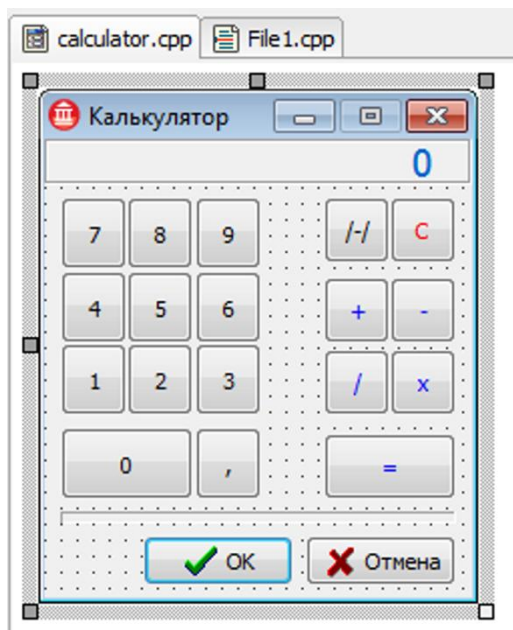


Рис. 4.2. Форма калькулятора

Опишем экспортируемую библиотекой функцию, благодаря которой к форме калькулятора будет возможно обратиться из других приложений (листинг 4.3).

**Листинг 4.3. Функция, экспортируемая модулем калькулятора**

```
extern "C" float __declspec(dllexport)
    showcalculator(HWND Handle, float InitValue);

float showcalculator(HWND Handle, float InitValue)
{
    Application->Handle=Handle;
    TfrmCalculator *frmCalculator=new TfrmCalculator(Application);
    frmCalculator->lbRes->Caption=FloatToStr(InitValue);
```

```

if (frmCalculator->ShowModal()==idOK)
{
    return StrToFloat(frmCalculator->lbRes->Caption);
}
else return 0;

frmCalculator->Release();
Application->Handle=0;
}

```

Объявленная в листинге функция `showcalculator()` обладает двумя параметрами. Параметр `Handle` необходим для передачи в создаваемую форму ссылки на дескриптор приложения, вызвавшего форму. Необязательный параметр `InitValue` содержит число, которым мы сможем проинициализировать начальное значение калькулятора. Форма калькулятора создается и выводится на экран в модальном режиме. Завершив расчеты, пользователь нажимает кнопку **ОК** (`TBitBtn *btnOK`), и результаты расчетов из метки `lbRes` передаются в нашу функцию.

Откомпилируйте библиотеку, нажав комбинацию клавиш `<Ctrl>+<F9>`. В результате должен создаться файл `mathlib.dll`.

## Вызов библиотеки из приложения

Чтобы динамической библиотекой смогло воспользоваться приложение, нам следует выбрать для нее подходящий каталог. Для этого подходят:

- ☐ каталог, из которого запускается основной исполняемый `exe`-файл;
- ☐ текущий каталог;
- ☐ системная папка ОС Windows (`c:\windows\system32` или `c:\windows\syswow64`);
- ☐ папка Windows;
- ☐ каталоги, включенные в переменную окружения `PATH`.

Именно в этой очередности осуществляется просмотр каталогов при поиске требуемой библиотеки. Обычно наиболее востребованные библиотеки размещаются в системной папке, там их могут найти все установленные в Windows приложения. А в нашем случае во время разработки проекта наиболее целесообразно размещать свои библиотеки в том же каталоге, где находится исполняемый файл.

## Неявное подключение DLL

Для того чтобы воспользоваться услугами динамической библиотеки, приложение должно отобразить DLL в свое адресное пространство. Различают два способа решения этой задачи: *неявное* (статическое) и *явное* (динамическое) подключение библиотеки.

Неявная загрузка предназначена для тех случаев, когда услуги библиотеки необходимы на протяжении всего периода выполнения приложения. В этом случае библиотека подключается к процессу в момент его старта и отключается вместе с завершением работы. Организация неявной загрузки включает следующие этапы:

- ☐ подключение к проекту файла `.lib` (файл создается автоматически в момент компиляции библиотеки).
- ☐ определение в проекте прототипа импортируемой функции.

Создайте новое приложение VCL Form и сохраните проект под любым именем.

Расположите на форме две строки ввода `Edit1` и `Edit2` — они позволят пользователю ввести два числа. Кроме того, нам понадобится кнопка `Button1`, которая инициирует обращение к функции суммирования. Результаты “сложнейших” вычислений мы направим в метку `Label1`.

Присоедините к приложению библиотечный файл с расширением имени `.lib`. Для этого в главном меню среды проектирования выберите пункт **Project | Add to Project** и

импортируем в проект файл `mathlib.lib` (по умолчанию он располагается в том же каталоге, что и библиотека `dll`).

Для определения в проекте прототипа импортируемой функции нам понадобится одна строка кода

```
extern "C" float __declspec(dllimport) add(float x, float y);
```

все остальное вы найдете в листинге 4.4.

#### Листинг 4.4. Статическое подключение

```
#include <vcl.h>
#pragma hdrstop
#include "Unit2.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

//описание прототипа импортируемой функции
extern "C" float __declspec(dllimport) add(float x, float y);
//-----
__fastcall TForm1::TForm1(TComponent* Owner)      : TForm(Owner)
{
}
//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    float x=StrToFloat(Edit1->Text);
    float y=StrToFloat(Edit2->Text);
    float z=add(x,y);
    Label2->Caption=FloatToStr(z);
}
```

Перед первым запуском приложения следует убедиться, что исполняемый `exe`-модуль и файл библиотеки `dll` находятся в одном каталоге. Чтобы не утруждать себя лишними операциями копирования файлов, проще всего внести незначительное изменение в поведение компилятора. Воспользуйтесь пунктом меню **Project | Options**, в появившемся окне опций проекта выделите узел верхнего уровня **Directories and Conditionals** и в строке **Final output directory** укажите папку, в которой размещена наша библиотека (рис 4.3). Нажав кнопку **OK**, сохраните настройки компилятора и только после этого запускайте проект на выполнение.

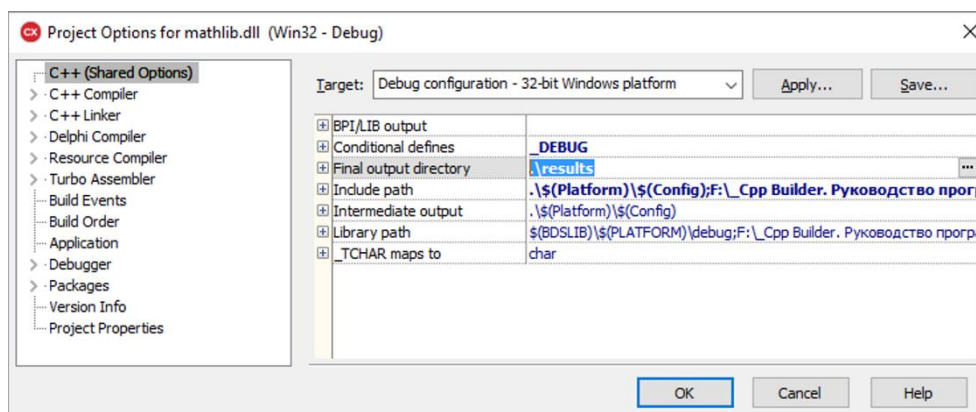


Рис. 4.3. Установка каталога для размещения откомпилированного файла

## Явное подключение DLL

Явное (динамическое) подключение библиотеки к проекту целесообразно применять в тех случаях, когда обращение к библиотеке происходит нерегулярно. В такой ситуации лучше всего отображать образ библиотеки в адресное пространство процесса только на время обращения к той или иной функции и сразу выгружать библиотеку.

При организации явной загрузки программист должен сделать четыре шага:

- ☐ загрузить DLL;
- ☐ получить адрес импортируемой функции;
- ☐ вызвать импортируемую функцию;
- ☐ выгрузить библиотеку из памяти.

Три из четырех действий реализуются при помощи функций Windows API. Отображение динамической библиотеки в адресное пространство процесса осуществляет функция

```
HMODULE WINAPI LoadLibrary(__in LPCSTR lpFileName);
```

Параметр `lpFileName` содержит имя (обычно без пути) подключаемой библиотеки.

В случае успеха функция возвратит дескриптор загрузившейся библиотеки.

Вторым этапом при использовании механизма явной загрузки библиотеки является этап получения адреса функции импортируемой из DLL. Для этого используется функция

```
FARPROC WINAPI GetProcAddress(__in HMODULE hModule,
                               __in LPCSTR lpProcName);
```

Параметр `hModule` соответствует дескриптору библиотеки, полученному при помощи функций `LoadLibrary()` или `LoadLibraryEx()`. Имя функции описывается в параметре `ProcName`. Возвращаемое значение не что иное, как указатель на вызываемую функцию.

На завершающем этапе необходимо выгрузить DLL из памяти системы. Для этого используют функцию

```
BOOL WINAPI FreeLibrary(__in HMODULE hModule);
```

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

Практическая сторона вопроса явной загрузки DLL и обращения к функции суммирования отражена в листинге 4.5, а процесс вызова формы калькулятора в 4.6.

**Листинг 4.5. Явное подключение библиотеки с функцией суммирования**

```
const wchar_t *libname=L"mathlib.dll"; //имя библиотеки
const char    *fname={"_add"};         //имя импортируемой функции
//прототип импортируемой функции
typedef float (__import fadd(float,float));

HMODULE hLib=LoadLibrary(libname); //загрузка библиотеки
if (hLib>0) { //если библиотека успешно загружена
    fadd *add;
    add=(fadd*)GetProcAddress(hLib,fname); //получаем адрес функции
    if (add) { //если получен адрес импортируемой функции
        float x=StrToFloat(Edit1->Text);
        float y=StrToFloat(Edit2->Text);
        float z=add(x,y);
        Label2->Caption=FloatToStr(z);
    }
    FreeLibrary(hLib); //выгружаем библиотеку
```

```
}

```

#### Листинг 4.6. Явное подключение библиотеки к приложению

```
const wchar_t *libname={L"calcklib.dll"}; //имя библиотеки
const char *fname={"_showcalculator"}; //имя импортируемой функции
//прототип импортируемой функции
typedef float (__import fshowcalculator(HWND,float));

HMODULE hLib=LoadLibrary(libname); //загрузка библиотеки

if (hLib>0) { //если библиотека успешно загружена
    fshowcalculator *showcalculator;
    //получаем адрес функции
    showcalculator=(fshowcalculator*)GetProcAddress(hLib,fname);
    if (showcalculator) { //если получен адрес импортируемой функции
        float x=StrToFloat(Edit1->Text);
        float y=showcalculator(Application->Handle,x);
        Label1->Caption=FloatToStr(y);
    }
}
FreeLibrary(hLib); //выгружаем библиотеку
} else ShowMessage("Библиотека не найдена!");
```

#### *Замечание*

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

## 5. Взаимодействие процессов

*Вид занятия – практическая работа*

*Цель – организация взаимодействия процессов в Windows*

*Продолжительность – 2 часа*

Уже давно прошло то время, когда в силу своей самодостаточности программе хватало опоры только на операционную систему. Современное приложение должно быть максимально открытым и способным взаимодействовать с другими выполняющимися как в системе, так и вне ее процессами. Любое взаимодействие нуждается в организации обмена данными. Но каким образом это осуществить? В этой главе мы получим представление о наиболее распространенных способах обмена данными между процессами, выполняющимися на одном компьютере (мы рассмотрим такие вопросы как обмен данными при посредничестве буфера обмена, обмен сообщениями и механизм файлового отображения) и на разных сетевых компьютерах (мы поговорим о почтовых слотах, именованных каналах и сокетах).

### Обмен данными через буфер обмена

В Microsoft Windows каждому процессу выделяется собственное виртуальное адресное пространство. Для 32-разрядных процессов его размер составляет 4 Гбайт, для 64-разрядных — 8 Тбайт. Адресное пространство одного процесса наглухо закрыто для другого, поэтому даже самые назойливые процессы не могут напрямую вмешиваться в деятельность своих коллег. Безусловно, один из процессов вправе в своем виртуальном адресном пространстве по какому-то адресу 0x12345678 разместить данные, которыми он не прочь поделиться с другими приложениями. Но беда в том, что для других процессов по адресу 0x12345678 может находиться все, что угодно, но только не требуемые данные, ведь речь идет об адресе в виртуальном адресном пространстве отдельного процесса, а он не имеет ничего общего с реальным физическим адресом.

Выход из положения заключается в создании области памяти, разделяемой всеми процессами и доступной для осуществления операций чтения/записи. Наиболее показательным примером решения такой проблемы выступает буфер обмена Windows.

Буфер обмена предназначен для передачи различного типа данных между приложениями или внутри одного приложения. Функционально буфер обмена представляет собой некоторый перечень методов Windows API, осуществляющих операции с областью памяти. Содержание операций заключается в действиях по загрузке в память и выгрузке из нее данных различного типа. Для обеспечения максимально удобной работы с буфером обмена в VCL введен специальный класс TClipboard, он описан в модуле Vcl.Clipbrd.hpp (который необходимо подключать к проекту). С присоединением данного модуля программист получит доступ к буферу обмена при посредничестве функции

```
TClipboard* Clipboard(void);
```

#### **Внимание!**

Для создания экземпляра класса TClipboard не требуется обращение к конструкторам, процесс построения буфера обмена инкапсулирован в функции Clipboard().

Буфер обмена служит универсальным хранилищем данных и позволяет размещать в себе обычную текстовую информацию, растровые и векторные рисунки, объекты OLE и т. п. Однако для различных типов данных, направляемых в буфер, требуется уникальный способ их хранения. Этот способ определяется *форматом буфера обмена*. Для ключевых, наиболее часто используемых типов объектов в Windows заранее подготовлены специальные зарегистрированные форматы буфера обмена. Однако если приложение предполагает размещать в буфере какую-то экзотическую информацию, то программист

имеет полное право описать и зарегистрировать в системе собственный — *пользовательский формат*.

Для выяснения, какие форматы доступны в буфере обмена компьютера, требуется обратиться к свойству

```
__property Word Formats[int Index]; //только для чтения
```

Свойство позволяет выяснить числовые значения — коды, зарегистрированных на данный момент форматов буфера обмена. Причем общее количество зарегистрированных форматов мы найдем в свойстве

```
__property int FormatCount; //только для чтения
```

Расшифровка наиболее распространенных форматов представлена в табл. 5.1.

*Таблица 5.1. Распространенные форматы буфера обмена*

| Константа       | Значение | В буфере обмена находится                                                                                                                   |
|-----------------|----------|---------------------------------------------------------------------------------------------------------------------------------------------|
| CF_TEXT         | 1        | Оканчивающиеся нулем символы в ASCII кодировке                                                                                              |
| CF_BITMAP       | 2        | Зависящий от устройства битовый образ                                                                                                       |
| CF_METAFILEPICT | 3        | Рисунок — метафайл с дополнительной информацией, описываемый структурой METAFILEPICT                                                        |
| CF_SYLK         | 4        | Специализированный формат данных Microsoft Symbolic Link (SYLK), применяемый, например, для передачи данных в буфер обмена программой Excel |
| CF_DIF          | 5        | Формат данных Data Interchange Format (DIF)                                                                                                 |
| CF_TIFF         | 6        | Изображения формата Tag Image File Format (TIFF)                                                                                            |
| CF_OEMTEXT      | 7        | Текст в кодировке OEM                                                                                                                       |
| CF_DIB          | 8        | Независящий от устройства битовый образ                                                                                                     |
| CF_PALETTE      | 9        | Цветовая палитра                                                                                                                            |
| CF_WAVE         | 12       | Стандартные аудиоданные в кодировке PCM                                                                                                     |
| CF_UNICODETEXT  | 13       | Текст в кодировке UNICODE                                                                                                                   |
| CF_ENHMETAFILE  | 14       | Расширенный метафайл                                                                                                                        |
| CF_HDROP        | 15       | Список файлов                                                                                                                               |
| CF_LOCALE       | 16       | Текстовые данные в соответствии с кодовой страницей                                                                                         |
| CF_PRIVATEFIRST | 200      | Диапазон значений, зарезервированный для пользовательских (нестандартных) форматов буфера обмена                                            |
| CF_PRIVATELAST  | 767      |                                                                                                                                             |

Для проверки нахождения в буфере обмена данных определенного типа применяют метод

```
bool HasFormat(System::Word Format);
```

На роль параметра претендуют константы из табл. 5.1 или соответствующие им числовые значения. Если формат действительно зарегистрирован системе, то метод возвращает true.

Посмотрите код примера в листинге 5.1. В нашем распоряжении есть командный объект `acPaste:TAction`, отвечающий за вставку текста из буфера обмена. В рамках события `OnUpdate()` мы проверяем наличие в буфере обмена текстовые данные и активируем (или переводим в пассивный режим) команду.

#### Листинг 5.1. Проверка наличия в буфере обмена текстовых данных

```
#include "clipbrd.hpp"
```

```
void __fastcall TForm1::acPasteUpdate(TObject *Sender)
{
    acPaste->Enabled=Clipboard()->HasFormat(CF_TEXT); //проверка буфера
}
```

Изначально буфер обмена создавался для переноса символьных строк, именно поэтому наиболее просто работать с текстовой информацией (формат CF\_TEXT). Для помещения текстовых данных в буфер или наоборот, чтения текста из него, используйте свойство

**\_\_property** UnicodeString AsText;

Для размещения в буфере текстовых данных воспользуйтесь функцией

```
void SetTextBuf(WideChar * Buffer);
```

Обратная задача — извлечение текста из буфера обмена — реализуется методом:

```
int GetTextBuf(WideChar * Buffer, int BufSize);
```

На содержимое буфера обмена укажет Buffer, параметр BufSize определяет, сколько символов надо копировать. Функция возвращает реальное количество скопированных символов.

Для отправки в буфер обмена любых других данных используйте метод

```
void SetAsHandle(Word Format, unsigned Value);
```

Указатель на копируемый объект передается в параметре Value. Параметр Format определяет формат буфера обмена.

Для получения указателя на данные, находящиеся в буфере обмена Windows, применяют функцию

```
unsigned GetAsHandle(System::Word Format);
```

Еще один способ копирования данных в буфер обеспечивает метод

```
virtual void Assign(TPersistent* Source);
```

Обратите внимание на то, что источник должен быть наследником класса TPersistent. В листинге 5.2 продемонстрированы способ копирования изображения в буфер обмена и обратная операция получение битового образа из буфера.

#### Листинг 5.2. Работа с графикой через буфер обмена

```
TBitmap *B=new TBitmap; //создаем объект растровой графики
B->LoadFromFile("c:\\picture.bmp"); //загружаем в него
//изображение из файла
Clipboard()->Assign(B); //копируем в буфер обмена
free(B); //уничтожаем объект BMP
Image1->Picture->Assign(Clipboard()); //достаем изображение из буфера
```

#### Замечание

Практически все компоненты имеющие отношение к редактированию текста (TEdit, TMemo и т. д.) и классы, предназначенные для отображения графики (TBitmap, TImage и т. д.), обладают собственными методами работы с буфером обмена.

Для очистки содержимого буфера предназначен метод

```
virtual void Clear(void);
```

Надо отметить, что буфер очищается автоматически перед помещением в него новых данных.

Иногда у приложения возникает необходимость забрать буфер обмена в монопольное использование, т. е. предотвратить возможность изменения содержимого буфера другими программами в отрезок времени, необходимый для выполнения каких-то операций нашим приложением. Для этой цели потребуется пара функций:

```
virtual void Open(void);
virtual void Close(void);
```

Метод `Open()` захватывает буфер обмена, метод `Close()` — освобождает. Эти методы всегда трудятся в паре, и мы должны быть уверены, что в любом случае, даже при возникновении ошибки во время выполнения программы метод `Close()` будет вызван гарантировано.

## Регистрация пользовательского формата буфера обмена

Для профессионального разработчика очень важно уметь создавать собственные форматы буфера обмена. Предположим, что в разрабатываемой программе часто используются структурированные данные, например, записи, содержащие поля: наименование товара, производитель, единица измерения, дата поступления и цена. Зачастую пользователю приходится вводить десяток практически одинаковых записей (отличающихся ценой или датой, или еще чем-нибудь), но для этого ему приходится монотонно набивать одни и те же данные. Не плохо бы в подобной ситуации оказать пользователю помощь и научить нашу программу реализовать методы копирования в буфер и вставки из буфера структурированных данных — записи. Но в Windows не определен формат буфера обмена, способный хранить данные такого рода. В подобных ситуациях приходится немного поработать головой.

Создайте новое приложение, состоящее из одной главной формы. При сохранении проекта традиционно переименуйте форму в `frmMain` и соответствующий ей программный модуль в `main.cpp`. Подключите к проекту модуль `Clipbrd`, объявите структуру `TDBRecord` (позднее она определит новый формат буфера обмена), объявите переменную `CF_MYFORMAT` (в ней окажется код нового зарегистрированного формата) и опишите секцию инициализации так, как предложено в листинге 5.3.

**Листинг 5.3. Подготовка модуля к работе с новым форматом буфера обмена**

```
struct TDBRecord {
    //товар, производитель, единица измерения
    wchar_t Name[40], Creator[40], Measure[5];
    Double Money; //цена за единицу
    TDate DateIn; //дата поступления
};

//-----
Word CF_MYFORMAT; //код формата
//-----

__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
    //регистрация формата
    CF_MYFORMAT=RegisterClipboardFormat(L"CF_MYFORMAT");
}
```

Первое, что происходит после страта модуля — регистрация нашего пользовательского формата данных. Для этого в секции инициализации вызывается функция Windows API:

```
UINT WINAPI RegisterClipboardFormat(__in LPCTSTR lpszFormat);
```

Функцию регистрации формата интересует единственный параметр, содержащий название нового формата. В случае успешной регистрации метод возвращает число, идентифицирующее новый формат буфера обмена. Полученное значение будет находиться в диапазоне от `0x0000` до `0xFFFF`. С этого времени формат станет доступным в списке форматов буфера обмена (вспоминайте методы `HasFormat()` и `Formats()`). Если несколько позднее любое другое приложение попытается зарегистрировать одноименный формат, то вместо повторной регистрации метод `RegisterClipboardFormat()` вернет идентификатор, полученный еще в момент первой регистрации. Таким образом, все

приложения, знающие имя и структуру формата, получают право обмениваться данными через буфер обмена.

Возвращаемся к нашему примеру. В секции публичных объявлений главной формы опишите заголовки методов копирования данных в буфер обмена и загрузки данных из буфера (листинг 5.4).

**Листинг 5.4. Объявление методов работы с буфером обмена**

```
#include <Vcl.Clipbrd.hpp>
//----- КОПИРОВАНИЕ В БУФЕР -----
void __fastcall CopyToClipboard(TDBRecord Value)
{
    HANDLE h=GlobalAlloc(GMEM_MOVEABLE,sizeof(TDBRecord));
    Clipboard()->Open();
    __try
    {LPVOID FirstByte=GlobalLock(h);
        Move(&Value,FirstByte,sizeof(TDBRecord));
        Clipboard()->SetAsHandle(CF_MYFORMAT, (unsigned int) h);
    }
    catch (...)
    { GlobalUnlock(h);
        Clipboard()->Close();
        MessageBeep(0);
    }
    GlobalUnlock(h);
    Clipboard()->Close();
}

//----- ЗАГРУЗКА ИЗ БУФЕРА -----
TDBRecord __fastcall LoadFromClipboard()
{
    TDBRecord DBRecord;
    ZeroMemory(&DBRecord,sizeof(TDBRecord));

    if (Clipboard()->HasFormat(CF_MYFORMAT)) {
        Clipboard()->Open();
        HANDLE h=(void *) Clipboard()->GetAsHandle(CF_MYFORMAT);
        if (h!=0) { LPVOID FirstByte=GlobalLock(h);
            __try
            {
                Move(FirstByte,&DBRecord,sizeof(TDBRecord));
            }
            __finally
            {
                GlobalUnlock(h);
                Clipboard()->Close();
            }
        }
    }
    return DBRecord;
}
```

Метод CopyToClipboard() несет ответственность за загрузку в память данных записи. Для этих целей функция GlobalAlloc() в общей куче распределяет некую область памяти, необходимую для размещения в ней содержимого записи TDBRecord. В первом параметре метода (при помощи константы GMEM\_MOVEABLE) мы указываем, что данная область памяти может быть перемещаемой, а во втором параметре определяем размер этой области в байтах. Функция возвращает дескриптор созданного объекта, а мы, в свою очередь, сохраняем его в переменную h. Следующая функция GlobalLock(), получив дескриптор

только что сформированного объекта `h`, транслирует его в физический адрес этого объекта — адрес его первого байта в области памяти. Помимо этого `GlobalLock()` информирует операционную систему о том, что мы приступили к работе с объектом `h`. Физически это выглядит, как приращение счетчика ссылок на объект на единицу. Функция `Move()` перемещает содержимое `DBRecord` в зарезервированную область памяти в байт с адресом `FirstByte^`. Нам остается воспользоваться уже знакомой функцией `SetAsHandle()` из коллекции методов класса `TClipboard`. В последних строках, в секции `finally` с помощью функции `GlobalUnlock()` мы уменьшаем на единицу счетчик ссылок на эту область памяти, тем самым информируем Windows о том, что приложение прекратило работать с объектом `h`.

Перейдем к методу `LoadFromClipboard()`, извлекающему данные из буфера обмена. С помощью функции `GetAsHandle()` в дескриптор `h` записывается указатель буфера обмена. Если дескриптор не равен нулю (признак того, что буфер не пуст), то узнаем адрес первого байта области памяти буфера и при помощи метода `Move()` копируем данные из буфера в структуру `TDBRecord`.

Для того чтобы проверить работоспособность кода, повторите пользовательский интерфейс приложения для работы с буфером обмена (рис. 5.1). Для этого разместите на главной форме строки ввода (`TEdit`) и кнопки (`TButton`).

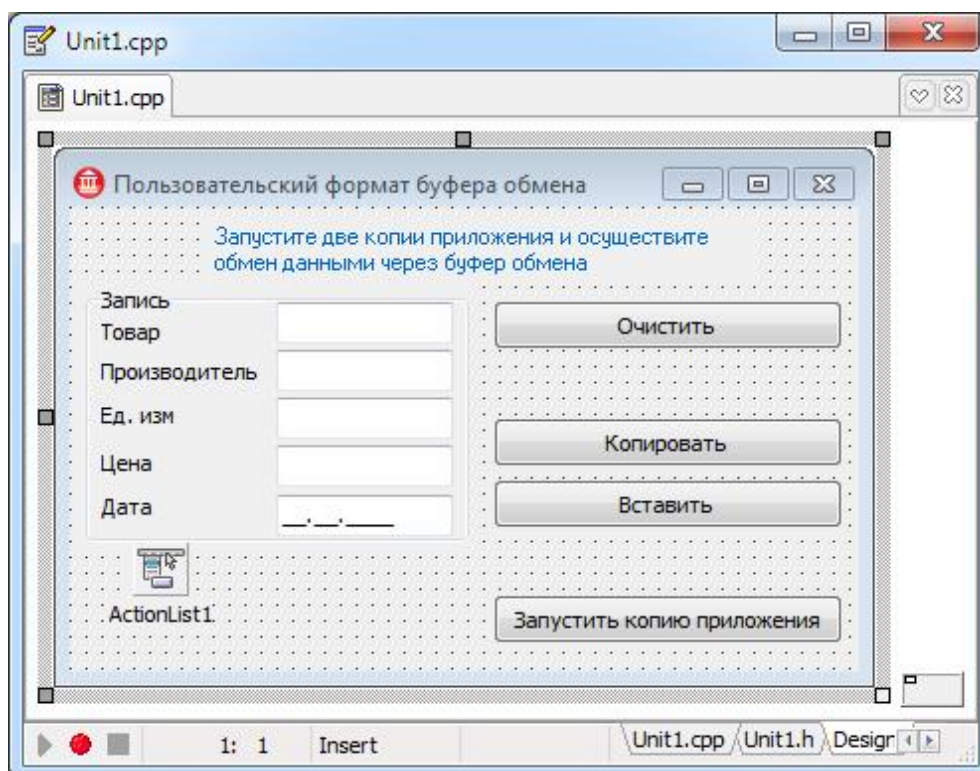


Рис. 5.1. Интерфейс приложения для работы с буфером обмена

Нам понадобятся 5 строк ввода `TEdit` для редактирования полей записи и кнопки для копирования данных в буфер и получения из буфера обмена (листинг 5.5).

#### Листинг 5.5. Объявление методов работы с буфером обмена

```
//----- Нажатие на кнопку копирования в буфер -----
void __fastcall TForm1::btnCopyClick(TObject *Sender)
{
    TDBRecord DBRecord;
    ZeroMemory(&DBRecord, sizeof(TDBRecord));
}
```

```

StrPCopy (DBRecord.Name, edName->Text);
StrPCopy (DBRecord.Creator, edCreator->Text);
StrPCopy (DBRecord.Measure, edMeasure->Text);
DBRecord.Money=StrToFloatDef (edMoney->Text, 0);
DBRecord.DateIn=StrToDateDef (edDate->Text, Date ());
CopyToClipboard (DBRecord);
}
//----- Нажатие на кнопку получения данных из буфера обмена -----
void __fastcall TForm1:: btnPasteClick (TObject *Sender)
{
TDBRecord DBRecord=LoadFromClipboard();
edName->Text=StrPas (DBRecord.Name);
edCreator->Text=StrPas (DBRecord.Creator);
edMeasure->Text=StrPas (DBRecord.Measure);
edMoney->Text=DBRecord.Money;
edDate->Text=FormatDateTime ("dd/mm/yyyy", DBRecord.DateIn);
}

```

Для тестирования кода откомпилируйте программу и запустите две копии приложения. Заполните все поля данных в одном из приложений и нажмите кнопку копирования данных в буфер обмена. Переключитесь на другой экземпляр программы. Нажатие кнопки вставки данных приведет к появлению в полях редактора данных из буфера обмена. Таким образом, мы реализовали один из самых распространенных способов обмена данными между приложениями.

## Обмен сообщениями

Наиболее распространенным способом взаимодействия процессов в Windows является обмен сообщениями. *Сообщение* представляет собой средство, с помощью которого Windows уведомляет процесс о том, что в системе что-то произошло, например, пользователь щелкнул кнопкой мыши, была нажата клавиша или системный таймер произвел очередной отсчет интервала времени.

Любое предназначенное для работы под управлением Windows приложение обладает хотя бы одним окном, называемым главным окном приложения. Напомню, что это ни в коем случае не главная форма приложения. Если приложение разработано в среде C++ Builder, то с его стартом создается невидимое главное окно, и для того чтобы добраться до него, нам потребуется свойство `Handle` объекта `Application`. На протяжении всего периода выполнения приложения оно находится под неусыпным контролем со стороны операционной системы. Для этого с момента создания до момента разрушения главного окна приложения операционная система поддерживает с ним активный диалог. Беседа ведется посредством отправки сообщений оконной функции главного окна приложения. Что понимается под словосочетанием "отправить сообщение"? Это значит, что операционная система осуществит запись о каком-то событии в область памяти, доступную оконной функции. Указанная область памяти называется *очередью сообщений*. Очередь может хранить несколько сообщений, для этого она устроена по принципу стека "первым пришел — первым вышел" (FIFO, First Input First Output). Программа также вправе отправить сообщение Windows. В этом случае сообщение от программы помещается в другую, общесистемную очередь Windows. Заметьте, что общесистемная очередь существует в одном-единственном экземпляре, и в нее помещаются сообщения от всех окон.

Окно также в состоянии отправлять сообщение другому окну, причем окна могут принадлежать и различным процессам. Именно на умении оконного элемента управления получать и обрабатывать сообщения основывается программирование приложений, управляемых событиями. Если вдруг главное окно приложения разучится "выуживать"

отправленную ему информацию из очереди сообщений, можно смело говорить о том, что программа "зависла". Помимо главного окна, приложение включает десятки и даже сотни обычных окон — это оконные элементы управления (потомки `TWinControl`), размещаемые программистом на формах проекта.

Для того чтобы приложение не прозевало предназначенное ей послание, оно периодически просматривает свой "почтовый ящик" на предмет новых поступлений. Для этого в любой программе (кстати, и в самой Windows) организуется *цикл обработки сообщений* (message loop). И если по какой-то причине у приложения остановится этот цикл, то оно безнадежно "зависнет", а если остановится цикл операционной системы, то нас спасет только перезагрузка.

#### **Замечание**

В библиотеке VCL получать сообщения умеют только оконные элементы управления (элементы управления, основанные на `TWinControl`).

## **Поиск окна**

Для того чтобы сообщение дошло до окна-получателя, мы должны знать его дескриптор. Если же дескриптор нам неизвестен, то можно воспользоваться функцией

```
HWND WINAPI FindWindow( __in_opt LPCTSTR lpClassName,
                        __in_opt LPCTSTR lpWindowName);
```

Задача функции — среди всех окон процессов попытаться найти именно то, которому предназначено наше послание.

Для поиска главного окна приложения первый параметр устанавливают в `NULL`, во второй параметр передается указатель на строку, заканчивающуюся нулем и содержащую название требуемого окна. В случае успеха функция вернет значение, отличное от нуля (листинг 5.6).

#### **Листинг 5.6. Объявление методов работы с буфером обмена**

```
HWND h=FindWindow(NULL,L"Project1") ;
if (h>0) {
    //...
}
```

Существует еще один способ поиска нужного окна. Так, функция

```
BOOL WINAPI EnumWindows(__in WNDENUMPROC lpEnumFunc,
                        __in LPARAM lParam);
```

позволяет построить список всех имеющихся в системе главных окон. В первый параметр функции следует передать ссылку на функцию обратного вызова, второй параметр предназначен для отправки в функцию обратного вызова дополнительных данных.

Для исследования функции нужно создать новое приложение и разместить на нем список `TListView *ListView1`. Переключите список в табличный режим представления данных (`ViewStyle=vsReport`) и в редакторе колонок создайте 4 колонки для хранения заголовка, класса окна, идентификатора потока управления и дескриптора окна. Все остальное вы найдете в листинге 5.7.

#### **Листинг 5.7. Построение списка окон**

```
//----- Функция обратного вызова перебора окон -----
bool CALLBACK EnumWndProc(HWND hwnd, LPARAM lParam)
{
    wchar_t WindowName[99];
    wchar_t ClassName[99];
    GetWindowText(hwnd, WindowName, sizeof(WindowName)); //заголовок окна
```

```

GetClassName(hwnd, ClassName, sizeof(ClassName)); //класс окна

Form1->ListView1->Items->BeginUpdate();

TListItem* LI=frmMain->LVWindows->Items->Add();
String s=Format("0x%.8x",ARRAYOFCONST(((int) hwnd)));
LI->Caption=s;
LI->SubItems->Add(ClassName);
LI->SubItems->Add(WindowName);

DWORD ID;
GetWindowThreadProcessId(hwnd,&ID); //узнаем ID потока
s=Format("0x%.8x",ARRAYOFCONST(((int) ID)));
LI->SubItems->Add(s);
LI->Data=&hwnd; //дескриптор окна сохраним в поле данных

Form1->ListView1->Items->EndUpdate();
return (true);
}
//----- Команда на построение перечня окон -----
void __fastcall TForm1::GetWindowsInformation(){
    ListView1->Clear();
    EnumWindows( (WNDENUMPROC) EnumWndProc, 0);
}

```

В момент вывода окна на экран осуществляется обращение к функции EnumWindows(), которая, в свою очередь, обращается за помощью к функции обратного вызова EnumWindowsProc(). Обратите внимание на то, что при описании функции обратного вызова мы воспользовались стандартным соглашением о вызовах stdcall, в противном случае код утратит работоспособность. В результате выполнения кода вы получите перечень из нескольких сотен окон, о существовании которых вы до этого момента даже и не подозревали (рис. 5.2).

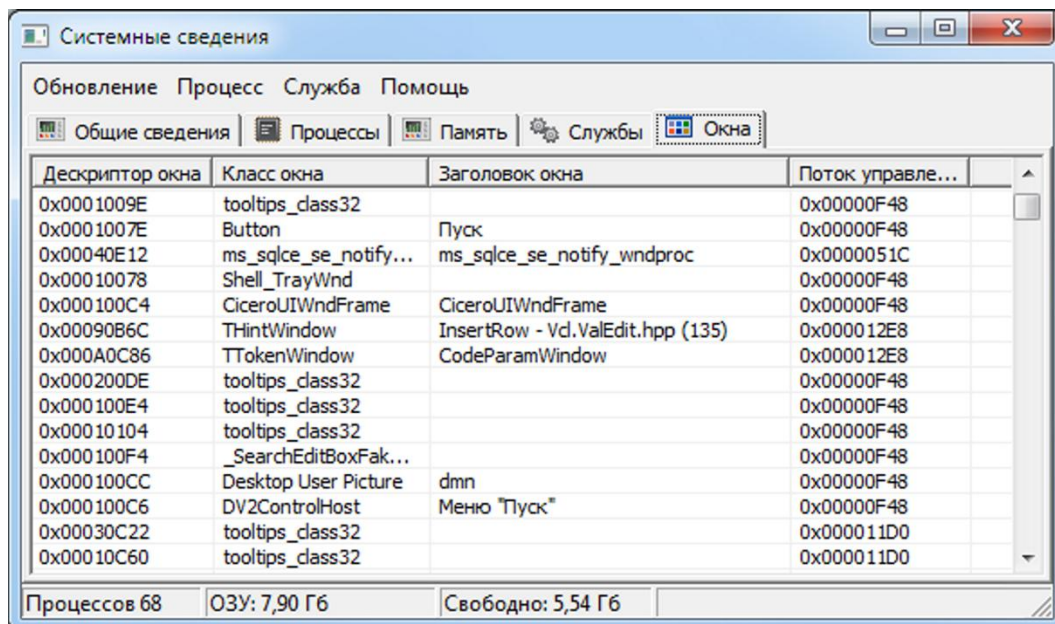


Рис. 5.2. Построение перечня окон с помощью функции EnumWindows()

## Регистрация пользовательских сообщений

В Windows зарегистрирован впечатляющий перечень сообщений, отправляемых для активизации окна, минимизации или распахивания его на весь экран. Сообщения

появляются на свет при нажатии и отпускании клавиш, при перемещении мыши или щелчке ее кнопки... Все сообщения описываются именованными константами, имена констант, как правило, начинаются с аббревиатуры "WM\_" (сокращение от Windows Message). Обязательно обратитесь к системе помощи для программистов MSDN (<http://msdn.microsoft.com>) и просмотрите обширный перечень зарегистрированных сообщений. Более того, ряд элементов управления способен воспринимать специфичные сообщения. Например, сообщения для комбинированного списка TComboBox начинаются с "CB\_", для дерева TTreeView — "TVM\_", для кнопки — "BN\_".

Операционная система допускает создание нестандартных сообщений — сообщений, определяемых программистами. Пользовательские сообщения могут применяться с целью организации взаимодействия между несколькими программами или в рамках одного приложения. Для регистрации своего сообщения применяется функция.

```
UINT RegisterWindowMessage(LPCTSTR lpString);
```

Единственный параметр метода рассчитывает получить имя сообщения. Функция регистрирует пользовательское сообщение и возвратит целое число — идентификатор сообщения. Если окажется, что сообщение с таким именем уже было заявлено ранее (например, другим приложением), то функция вернет идентификатор зарегистрированный другим приложением.

В адрес окна могут поступать сообщения от самых различных отправителей, как находящихся внутри этого же приложения, так и от внешних источников (других приложений и операционной системы). При поступлении сообщений к главному окну приложения у Application возникает событие

```
__property TMessageEvent OnMessage;
```

```
typedef void (__closure *TMessageEvent)(tagMSG &Msg, bool &Handled);
```

Содержимое сообщения передается в структуре tagMSG &Msg (листинг 5.8).

**Листинг 5.8. Объявление структуры tagMSG**

```
typedef struct tagMSG {
    HWND        hwnd;        //дескриптор получателя сообщения
    UINT        message;     //зарегистрированный идентификатор сообщения
    WPARAM      wParam;     //необязательный старший параметр
    LPARAM      lParam;     //необязательный младший параметр
    DWORD       time;        //время создания сообщения
    POINT        pt;         //экранные координаты указателя мыши
} MSG, *PMSG, NEAR *NPMSG, FAR *LPMSG;
```

Если в параметре Handled программист возвращает операционной системе значение true, то это означает, что сообщение обработано полностью. В свою очередь, значение false станет сигналом Windows о том, что при остром желании система может продолжить обрабатывать сообщение.

Для получения доступа к обработчику события OnMessage() можно пойти двумя путями. Первый из них — воспользоваться услугами компонента TApplicationEvents. Упомянутый компонент инкапсулирует все события приложения. Второй путь требует написания нескольких дополнительных строк кода, но освобождает программиста от необходимости "захламлять" свой проект излишними компонентами.

## Пример обмена сообщениями между процессами

Спроектируем два небольших приложения, демонстрирующих на практике процесс обмена сообщениями. Приложение-передатчик станет отправлять сообщение со своими экранными координатами в адрес приложения-приемника. Приложение-приемник (получив координаты) укажет пользователю направление на приложение-передатчик.

Начнем работу с приложения-передатчика. Создайте новый проект, на единственной форме проекта разместите компонент `TApplicationEvents` `*ApplicationEvents1`. Уменьшите размер формы до минимальных размеров. Сохраните проект под любым именем.

Чтобы научить приложение отправлять сообщения, понадобится минимум усилий, весь исходный код главной формы проекта предложен в листинге 5.9.

#### Листинг 5.9. Исходный код приложения-передатчика

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

#include "transmit.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TfrmTransmit *frmTransmit;

unsigned WM_MSG; //здесь будем хранить идентификаторы сообщений
//----- Регистрируем в Windows сообщение -----
__fastcall TfrmTransmit::TfrmTransmit(TComponent* Owner) : TForm(Owner)
{
    WM_MSG=RegisterWindowMessage(L"WM_SENDPOSITION");
}
//----- Отправка сообщения с координатами центра формы -----
void __fastcall TfrmTransmit::ApplicationEvents1Idle(TObject *Sender, bool &Done)

{
    //поиск главного окна приложения с названием "RECEIVER"
    HWND h=FindWindow(NULL,L"RECEIVER");
    if (h>0) // если окно обнаружено
    {
        TPoint FormCenter=Point(frmTransmit->ClientWidth / 2,
                                frmTransmit->ClientHeight / 2);
        FormCenter=frmTransmit->ClientToScreen(FormCenter);
        Label1->Caption=Format("Координаты (%d,%d)",
                               ARRAYOFCONST((FormCenter.X,FormCenter.Y)));
        PostMessage(h, WM_MSG, FormCenter.X, FormCenter.Y); //отправляем сообщение
    }
}
```

В момент инициализации приложение регистрирует пользовательское сообщение "WM\_SENDPOSITION" и результаты регистрации помещает в переменную `WM_MSG`. После этого в момент возникновения события `OnIdle` (событие будет генерироваться регулярно в момент перехода приложения-передатчика в состояние простоя, или, попросту говоря, "безделья") мы узнаем экранные координаты центра приложения-передатчика и отправляем их приемнику.

Обратите внимание, что поиск приемника осуществляет уже знакомая нам функция `FindWindow()`. Функция старается обнаружить окно с заголовком "RECEIVER". Это означает, что при проектировании приемника нам надо не забыть присвоить это имя заголовку главной формы.

Для отправки сообщения в адрес приложения-приемника используется функция

```
BOOL PostMessage(HWND hWnd, // дескриптор окна-получателя
                 UINT Msg, // отправляемое сообщение
                 WPARAM wParam, //старший параметр сообщения
                 LPARAM lParam); //младший параметр сообщения
```

У функции `PostMessage()` есть коллега с идентичным набором параметров

```
LRESULT SendMessage(HWND hWnd, // дескриптор окна-получателя
    UINT Msg, // отправляемое сообщение
    WPARAM wParam, // дополнительный старший параметр сообщения
    LPARAM lParam); // дополнительный младший параметр сообщения
```

Однако, не смотря на внешнюю схожесть, поведение функций существенно отличается друг от друга. Функция `SendMessage()` работает в синхронном режиме, она отправляет сообщение непосредственно функции окна и ожидает окончания его обработки. В отличие от нее асинхронная функция `PostMessage()` стандартным образом помещает сообщение в очередь Windows и сразу возвращает управление вызвавшей ее программе, не дожидаясь результатов обработки сообщения.

Синхронная функция `SendMessage()` относится к разряду потенциально опасных, т. к. если окно-получатель по каким-то причинам не сможет ответить на сообщение, то вызвавшее функцию приложение может даже потерять работоспособность. Поэтому вместо `SendMessage()` зачастую используют `SendMessageTimeout()`. Указанная функция, ожидает ответ в течение установленного таймаута, и если за этот промежуток времени ответа получено не было, она гарантированно возвратит управление своей программе.

Исходя из разницы в работе функций, различаются и возвращаемые ими значения. `SendMessage()` возвращает значение, полученное в результате обработки сообщения; `PostMessage()` всего лишь информирует, удалось ли ей поставить сообщение в очередь (значение `true`) или нет (`false`).

#### ***Замечание***

Для организации широковещательной рассылки в первый параметр функций `SendMessage()` и `PostMessage()` следует передать константу `HWND_BROADCAST`.

Переходим ко второй части нашей работы — разработке приложения-приемника. Создаем новое приложение и сразу же в свойство `Caption` главной формы проекта заносим заголовок окна "RECEIVER". Разместите на форме единственный компонент — метку `TLabel *Label1` — и сохраните проект под любым именем. Нам осталось разобраться с исходным кодом (листинг 5.10).

#### **Листинг 5.10. Исходный код приложения-приемника**

```
#include <vcl.h>
#include <Winuser.h>
#pragma hdrstop

#include "Unit1.h"
//-----

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

TPoint TransmitterPos; //координаты точки
unsigned WM_MSG; //идентификатор сообщения
//----- Регистрируем в Windows сообщение -----
__fastcall TForm1::TForm1(TComponent* Owner): TForm(Owner)
{
    WM_MSG=RegisterWindowMessage(L"WM_SENDPOSITION");
}
//-----

void __fastcall TForm1::OnMessage_Event(tagMSG &Msg, bool &Handled)
{
    if (Msg.message==WM_MSG)
    {
        Label1->Caption=Format("Координаты (%d,%d)",
                                ARRAYOFCONST((Msg.wParam,Msg.lParam)));
    }
}
```

```

        TransmitterPos.X=Msg.wParam;
        TransmitterPos.Y=Msg.lParam;
        Repaint();
    }
}

//-----
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Application->OnMessage=OnMessage_Event;
    Label1->Caption="";
}
//-----
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    TPoint Center=Point(Form1->ClientWidth / 2, Form1->ClientHeight / 2);

    Form1->Canvas->Brush->Color=Form1->Color;
    Form1->Canvas->FillRect(Form1->ClientRect);
    Form1->Canvas->Brush->Color=Form1->Canvas->Pen->Color;
    Form1->Canvas->Ellipse(Center.X-3,Center.Y-3,Center.X+3,Center.Y+3);
    Form1->Canvas->MoveTo(Center.X, Center.Y);

    TransmitterPos=ScreenToClient(TransmitterPos);
    Form1->Canvas->LineTo(TransmitterPos.X, TransmitterPos.Y);
}

```

Для работы приемника понадобятся две глобальные переменные. В переменную TransmitterPos мы станем записывать поступающие от приложения-передатчика экранные координаты, в WM\_MSG после инициализации приложения окажется числовой идентификатор сообщения "WM\_SENDPOSITION". Обработка поступающих сообщений производится в рамках метода OnMessage\_Event(). На этот раз мы отказались от применения в проекте компонента TApplicationEvents и обеспечили реакцию на событие "вручную".

Откомпилируйте и запустите оба приложения. На рис. 5.3 представлен экранный снимок разработанных приложений, благодаря механизму обмена сообщениями приемник владеет информацией о местонахождении главного окна передатчика и проводит линию в его направлении.

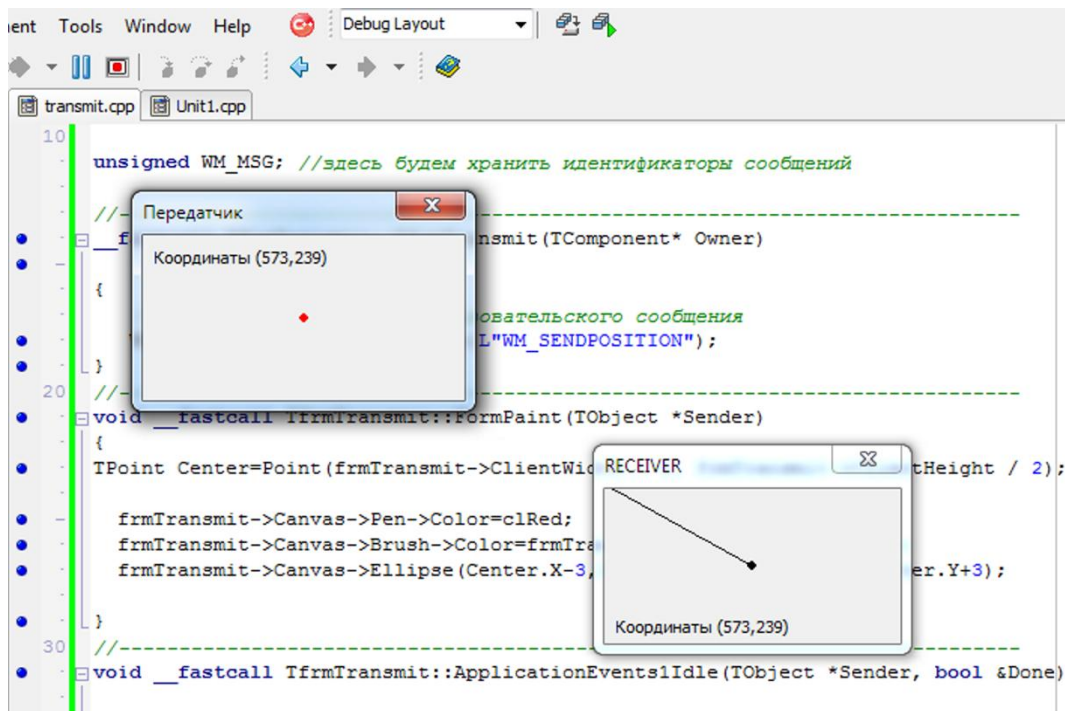


Рис. 5.3. Экранный снимок работы приложений передатчика и приемника

### Примечание

Для обмена данными между процессами часто применяют специальное сообщение `WM_COPYDATA`. Перед отправкой сообщения программист заполняет особую структуру `COPYDATASTRUCT`. Ссылка на нее передается через параметр `lParam` сообщения. Через параметр `wParam` необходимо передать идентификатор окна, посылающего сообщение.

## Файловое отображение

Каждому выполняющемуся под управлением Windows процессу выделяется индивидуальное адресное пространство. Таким образом, не имея прямой возможности изменять данные в чужом адресном пространстве, приложения защищаются от взаимных посягательств. Но это достоинство превращается в недостаток, когда нескольким процессам требуется манипулировать данными, принадлежащим адресному пространству одного из приложений. Выход из подобной ситуации очевиден — надо научиться отображать некоторый объем совместно используемой памяти в адресное пространство взаимодействующих приложений.

Одним из способов, позволяющих организовать работу приложения с разделяемыми данными, является применение отображаемых в память файлов. Для создания объекта файлового отображения используют функцию

```
HANDLE WINAPI CreateFileMapping(in HANDLE hFile,
                                in LPSECURITY_ATTRIBUTES lpAttributes,
                                in DWORD flProtect,
                                in DWORD dwMaximumSizeHigh,
                                in DWORD dwMaximumSizeLow,
                                in LPCTSTR lpName);
```

Перед попыткой создать именованный объект отображения функция обязательно проверит систему на наличие в ней объекта с именем `lpName`. Если объект отображения с таким именем уже существует (был сформирован ранее из этого или стороннего процесса), то вместо создания его дубликата функция возвратит дескриптор уже существующего объекта.

Первый параметр метода определяет дескриптор файла, на основе которого создается объект отображения, в нашем случае этому параметру присваивается константа

`INVALID_FILE_SIZE = DWORD(0xFFFFFFFF)`. Это указание создать отображение на основе файла подкачки. Второй параметр содержит указатель на структуру безопасности `LPSECURITY_ATTRIBUTES`. Если мы не намерены накладывать ограничения на доступ к объекту отображения, то вместо указателя на структуру атрибутов безопасности передаем неопределенный указатель `NULL`. Разрешения на операции чтения/записи определяются в параметре `flProtect`. Это может быть одна из трех констант: `PAGE_READONLY` (доступ только для чтения), `PAGE_READWRITE` (доступ для чтения и записи) и `PAGE_WRITECOPY` (в спроецированную область разрешено копирование и запись). Два очередных параметра определяют размер проецируемого файла, `dwMaximumSizeHigh` — старшие 32 бита, `dwMaximumSizeLow` — младшие 32 бита. Благодаря этой паре параметров мы сможем формировать 64-битный адрес. Если размер файла меньше 4 Гбайт, то в `dwMaximumSizeHigh` передается нулевое значение. Последний параметр `lpName` — указатель на строку с именем файла отображения. В случае успешного выполнения метод возвращает дескриптор объекта отображения файла, иначе функция возвратит неопределенный указатель `NULL`.

Распределенная посредством вызова `CreateFileMapping()` память может быть разделенной между несколькими процессами потому, что фактически она располагается вне адресного пространства процессов. Разделяемый участок в файле подкачки переносится в ОЗУ и может стать доступным для всех процессов, знающих имя отображаемого файла.

Функция `CreateFileMapping()` только создала файловый объект. Для отображения файла в адресное пространство нам понадобится помощь функции

```
LPCVOID WINAPI MapViewOfFile( __in HANDLE hFileMappingObject,
                               __in DWORD dwDesiredAccess,
                               __in DWORD dwFileOffsetHigh,
                               __in DWORD dwFileOffsetLow,
                               __in SIZE_T dwNumberOfBytesToMap );
```

Здесь параметр `hFileMappingObject` — возвращенный методом `CreateFileMapping()` дескриптор объекта отображения. Параметр `dwDesiredAccess` определяет тип доступа к представлению. Различают три основных режима:

- ☐ `FILE_MAP_WRITE` или `FILE_MAP_ALL_ACCESS` — доступ для чтения и записи;
- ☐ `FILE_MAP_READ` — доступ для чтения;
- ☐ `FILE_MAP_COPY` — доступ для копирования.

Параметры `dwFileOffsetHigh` и `dwFileOffsetLow` соответственно задают старшие и младшие 32 бита смещения внутри файла отображения. `dwNumberOfBytesToMap` — количество байтов файла, которые должны быть отображены; если параметр установлен в 0, то отображается весь файл. В случае успешного завершения метод возвратит указатель на самый первый байт спроецированного файла (базовый адрес объекта отображения).

Для отмены отображения представления файла из адресного пространства процесса предназначен метод

```
BOOL WINAPI UnmapViewOfFile( __in LPCVOID lpBaseAddress );
```

Единственный параметр функции — базовый адрес объекта отображения (результат работы метода `MapViewOfFile`).

Для практической реализации механизма файлового отображения нам потребуется создать новый проект и разместить на нем следующий перечень компонентов:

- ☐ в строку ввода `TEdit* Edit1` мы сможем вводить текст, который мы позднее сможем разделить между несколькими приложениями;
- ☐ кнопка `TButton* Button1` предназначена для передачи текста в объект файлового отображения;
- ☐ многострочный редактор `TMemo* Memo1` специализируется на визуализации содержимого объекта файлового отображения;
- ☐ компонент `TApplicationEvent* ApplicationEvent1` позволит нам в фоновом потоке читать данные из файлового отображения.

Все остальное вы обнаружите в листинге 5.11.

**Листинг 5.11. Демонстрация механизма файлового отображения**

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

#include "Unit1.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

static HANDLE hMapFile = NULL;
static LPTSTR lpMapAddress = NULL;
String recivmsg; //полученное текстовое сообщение
//-----Инициализация файлового отображения -----
__fastcall TForm1::TForm1(TComponent* Owner): TForm(Owner)
{
hMapFile = CreateFileMapping((HANDLE)0xFFFFFFFF, NULL,
PAGE_READWRITE, 0, 1024, L"DEMO_MAPPING" );
lpMapAddress =(LPTSTR) MapViewOfFile(hMapFile,FILE_MAP_ALL_ACCESS, 0, 0, 0);
}
//----- Формируем текст сообщения и передаем в файловый объект ----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
// формируем текст отправляемого сообщения
wcscpy(lpMapAddress, Edit1->Text.w_str());
FlushViewOfFile( lpMapAddress, lstrlen(lpMapAddress));
}
//---Фоновый поток, в котором осуществляется чтение из отображения --
void __fastcall TForm1::ApplicationEventsIdle(TObject *Sender, bool &Done)
{
HANDLE hFileMap;
LPTSTR lpMapAddress;
hFileMap = OpenFileMapping( FILE_MAP_ALL_ACCESS, FALSE, L"DEMO_MAPPING" );
if (hFileMap!=NULL) {
lpMapAddress =
(LPTSTR) MapViewOfFile( hFileMap,FILE_MAP_ALL_ACCESS, 0, 0, 0);
String s=StrPas(lpMapAddress); //полученный текст
if (s!=recivmsg) //сравниваем с предыдущим тестом
{ // это новое сообщение
recivmsg=s;//запоминаем
Memo1->Lines->Add(s);//выводим в Memo1
}
UnmapViewOfFile( lpMapAddress );
}
}
//----- Завершение работы -----
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
CloseHandle( hMapFile );
}
```

Объект файлового отображения создается в момент инициализации проекта. Сразу после этого мы регистрируем пользовательское сообщение, которое будет отправляться всем другим процессам в тот момент, как наше приложение осуществит запись в файловый объект.

Запись данных в объект файлового отображения осуществляется по щелчку по кнопке `Button1`. В этот момент текст из строки `Edit1` записывается в файловый объект. Чтение данных из файлового отображения производится в момент простоя приложения, для этого нам понадобится обработчик события `OnIdle()` компонента `ApplicationEvents1`.

Проект завершен. Запустите несколько экземпляров нашего приложения и проверьте работоспособность кода.

## 6. Взаимодействие в сети

*Вид занятия – практическая работа*

*Цель – организация межсетевого взаимодействия*

*Продолжительность – 2 часа*

В состав системы Microsoft Windows интегрирован обширный перечень сетевых технологий различного назначения и уровней сложности, позволяющих создавать сетевое программное обеспечение. В самом общем случае задача сетевого программного обеспечения заключается в обмене данными между двумя и более приложениями, выполняющимися на разных сетевых станциях. Обычно взаимодействующие в сети процессы подразделяются на серверные и клиентские. Клиент просит сервер оказать ему услугу, отправляя ему какой-то запрос. В свою очередь сервер, в силу своих возможностей, выполняет поручение. Никто не запрещает при разработке своего программного обеспечения одновременно наделять приложение полномочиями как сервера, так и клиента.

В этой главе мы рассмотрим несколько сетевых технологий. Свою работу начнем с наиболее простого *механизма почтовых слотов (mailslots)*. Используя названную технологию, приложение получает право отправить сообщение в какой-то почтовый слот, в произвольный момент времени владелец этого слота может письмо "прочитать". Упомянутый механизм допускает отправку сообщения как одному компьютеру, так и рассылку почты нескольким компьютерам одновременно, главное, чтобы они были объединены в единый домен. Еще один механизм, с которым мы познакомимся в этой главе, называется *каналом (pipe)*. Взаимодействие между процессами с использованием именованных каналов широко применяется в клиент-серверных технологиях. Кроме того, в настоящей главе большое внимание уделяется организации обмена данными с помощью интерфейса *сокетов* (технология WinSock).

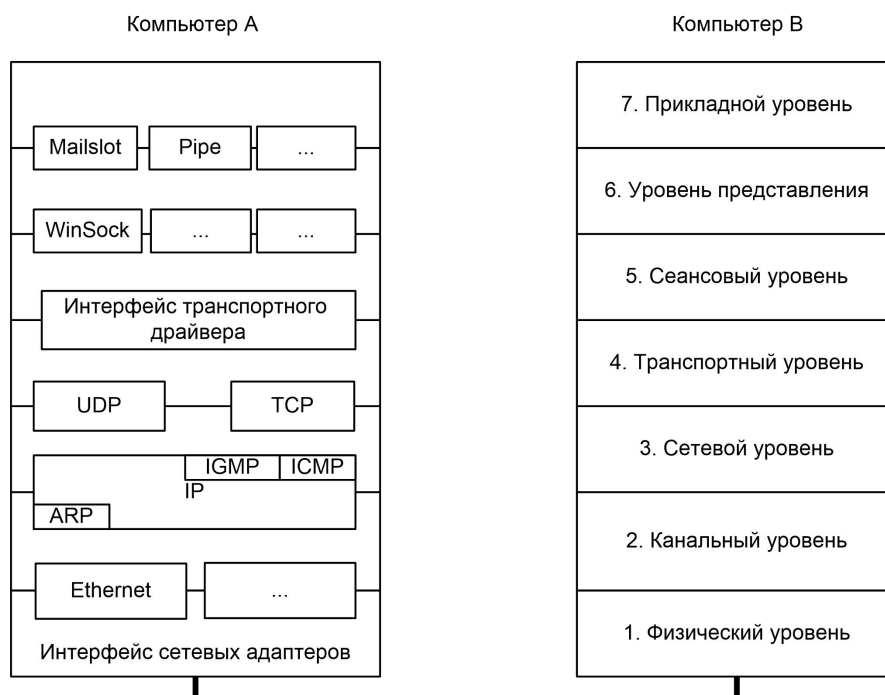


Рис. 6.1. Место сетевых технологий в модели Open System Interconnection

Взглянув на рис. 6.1, вы сразу поймете, что почтовые слоты и каналы являются высокоуровневыми технологиями, т. к. они расположились на стыке прикладного и

представительного уровня 7-уровневой модели OSI (Open System Interconnection). Сокеты Windows — относительно низкоуровневый сервис, он предоставляет программисту более гибкие возможности по разработке программного обеспечения, одновременно требуя больше знаний и затрат времени на отладку приложения.

## Почтовые слоты

Своим названием технология почтовых слотов обязана схожестью принципа работы с обычным почтовым ящиком. Серверное приложение имеет возможность создать собственный почтовый ящик, корреспонденцию в который имеет возможность отправить любой клиент (при условии обладания адресом слота).

Механизм mailslots односторонний. Передаваемое в виде дейтаграммы сообщение, попав (или не попав) в предназначенный для него почтовый слот, не уведомляет отправителя, что оно прибыло по назначению. Так что о гарантированной доставке почты речи идти не может.

Почтовые слоты часто используются для широковещательной рассылки небольших сообщений. Надо знать, что размер почтового отправления ограничен — он не должен превышать 424 байта.

Организация работы с почтовым слотом сравнительно проста. Сервер регистрирует файл-слот и размещает его в памяти компьютера. Правом читать данные из слота также обладает только сервер, причем для доступа к слоту применяются методы, весьма напоминающие процесс чтения из обычного файла. В свою очередь клиент имеет право лишь отправлять сообщения в почтовый слот.

## Определение имени почтового слота

Во время регистрации почтового слота ему присваивается уникальный сетевой адрес. Существует правило, согласно которому склеивается строка с текстом адреса:

- ☐ адрес начинается с двух наклонных черт, затем идет точка, вновь наклонная черта и ключевое слово "mailslot";
- ☐ необязательный путь, также начинающийся с наклонной черты;
- ☐ адрес завершается наклонной чертой и именем слота.

Например,

```
const wchar_t *MailSlotName=L"\\\\.\\mailslot\\sample_slot";
```

При обращении к почтовому ящику другого компьютера алгоритм создания адресной строки незначительно изменяется. В этом случае строка также начинается с двух наклонных черт, но теперь вместо точки указывается сетевое имя компьютера, к которому отправляется корреспонденция или имя домена, если мы одновременно обращаемся к целой группе компьютеров:

```
\\имя_компьютера\\mailslot\\[путь]имя_слота
\\имя_домена\\mailslot\\[путь]имя_слота
```

## Управление почтовым слотом

За создание почтового ящика отвечает приложение-сервер, оно обращается к функции прикладного программного интерфейса Windows

```
HANDLE WINAPI CreateMailslot(
    __in LPCTSTR lpName,
    __in DWORD nMaxMessageSize,
    __in DWORD lReadTimeout,
    __in_opt LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

Здесь `lpName` — указатель на строку с адресом слота. `nMaxMessageSize` — ограничитель максимальной длины послания в байтах, если параметру присвоить нулевое значение, то ограничения снимаются. Очередной параметр `lReadTimeout` оказывает влияние на процесс чтения корреспонденции из создаваемого ящика. Он задает время ожидания в миллисекундах. Если передается 0, то при обращении к пустому ящику функция, читающая корреспонденцию, возвратит результат немедленно. В противном случае она выждет установленный таймаут в надежде, что за этот промежуток времени нам кто-нибудь напишет. Для вечного ожидания передайте константу `MAILSLOT_WAIT_FOREVER`. Последний параметр `lpSecurityAttributes` — указатель на структуру атрибутов безопасности, в простейшем случае в него передают `NULL`. В случае успешного выполнения метод возвращает дескриптор созданного слота, иначе — нулевое значение.

Кроме функции создания почтового слота, Windows API способен похвастаться еще двумя функциями работы с `mailslot`. Во-первых, это функция, позволяющая настраивать время таймаута `lReadTimeout` в слоте, заданном дескриптором `hMailSlot`:

```
BOOL WINAPI SetMailslotInfo( __in HANDLE hMailslot,
                             __in DWORD lReadTimeout);
```

Во-вторых, это функция, информирующая о состоянии почтового слота:

```
BOOL WINAPI GetMailslotInfo( __in HANDLE hMailslot,
                             __out LPDWORD lpMaxMessageSize,
                             __out_opt LPDWORD lpNextSize,
                             __out_opt LPDWORD lpMessageCount,
                             __out_opt LPDWORD lpReadTimeout);
```

Эта функция вновь требует указать дескриптор интересующего нас слота в параметре `hMailslot`. А в результате метод выдаст данные о: `lpMaxMessageSize` — максимальном размере сообщения в байтах; `lpNextSize` — размере следующего сообщения, и если такового нет, то в параметре окажется константа `MAILSLOT_NO_MESSAGE`; `lpMessageCount` — общем числе сообщений в слоте; `lpReadTimeout` — выбранном времени таймаута почтового ящика.

## Получение и отправка корреспонденции

Чтение находящихся в почтовом слоте данных возможно только из программы-сервера, создавшей этот самый слот. Зная дескриптор слота, приложение обращается к нему, как к обыкновенному файлу. Для операции чтения подходят как функции Windows API `ReadFile()` и `ReadFileEx()`, так и их наследник — функция из состава VCL `FileRead()`.

Для отправки почты клиентское приложение просто осуществляет запись в виртуальный файл, олицетворяющий почтовый слот сервера-получателя. Обычно для этого применяют методы `FileCreate()`, `FileWrite()` и `FileClose()`.

## Пример почтового приложения

Приступим к разработке приложения, поддерживающего технологию почтовых слотов. Для этого потребуется создать новый проект, на главной форме которого надо разместить следующие элементы управления:

- ☐ строку ввода `cbComputers`, предназначенную для ввода сетевого имени компьютера, в адрес которого отправится почтовая корреспонденция;
- ☐ многострочный редактор `TMemo *MemoOut` позволит пользователю вводить текст сообщения, предназначенного для отправки;
- ☐ многострочный редактор `TMemo *MemoIn` раскроет входящие сообщения;
- ☐ менеджер команд `TActionManager *ActionManager` станет центром управления приложением.

Сохраните проект, предварительно переименовав главную форму в `frmMain`, название проекта выберите на свое усмотрение. Программирование начнем с регистрации почтового слота. Для этого объявите константу `MailSlotName`, в которую мы внесем имя

почтового слота, и обратитесь к событию `OnCreate()`, вызываемому в момент создания главной формы (листинг 6.1).

#### Листинг 6.1. Создание приложения и регистрация слота

```
const wchar_t *MailSlotName=L"\\\\.\\mailslot\\sample_slot";
HANDLE hMailSlot;
//-----СТАРТ ПРИЛОЖЕНИЯ И СОЗДАНИЕ СЛОТА -----
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    //создание слота
    hMailSlot=CreateMailslot(MailSlotName, 0, MAILSLOT_WAIT_FOREVER, NULL);

    if (hMailSlot==INVALID_HANDLE_VALUE) { /*если вызов метода завершился неудачей*/
        unsigned ErrorCode=GetLastError(); //узнаем код ошибки
        if (ErrorCode!=0) {
            String s=SysErrorMessage(ErrorCode); //расшифровка ошибки
            String err=Format("Ошибка %d %s", ARRAYOFCONST((ErrorCode, s)));
            MessageBox(Form1->Handle, err.w_str(), L"Ошибка",
                MB_ICONERROR | MB_OK);
        }
    }

    unsigned long w;
    wchar_t buf[MAX_COMPUTERNAME_LENGTH];
    GetComputerName(buf, &w);
    StatusBar1->Panels->Items[0]->Text="\\\\\\"+StrPas(buf);
}
```

После создания жизненный цикл почтового слота будет продолжаться до тех пор, пока мы не обратимся к функции `CloseHandle()`, освобождающей экземпляр слота. Эту операцию мы проведем в момент закрытия главной формы (листинг 6.2).

#### Листинг 6.2. Завершение работы приложения

```
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
    CloseHandle(hMailSlot);
}
```

Разработаем метод чтения почты из слота. Для этого в секции частных объявлений опишем заголовок метода `ReadMail()`, обладающий единственным параметром `aMailSlot`, в который следует передавать дескриптор почтового ящика, откуда будет прочитано почтовое отправление. Реализация метода представлена в листинге 6.3.

#### Листинг 6.3. Получение почты

```
void __fastcall TForm1::ReadMail(HANDLE aMailSlot)
{
    DWORD NextSize, MessageCount;
    //сначала узнаем о содержимом почтового слота
    if (GetMailslotInfo(aMailSlot, NULL, &NextSize, &MessageCount, NULL))
    {
        //если ящик не пуст, то просматриваем его
        while (MessageCount!=0) //собираем все сообщения из ящика
        {
            char *Buf=(char *)malloc(NextSize+1); //распределяем область памяти
            DWORD NumberOfBytesRead;
            //читаем данные из ящика
            ReadFile(aMailSlot, Buf, NextSize++, &NumberOfBytesRead, NULL);
            MemoIn->Lines->Add(Buf); //сообщения помещаем в мемо-поле
        }
    }
}
```

```

//в старых версиях надо писать MemoIn->Lines->Add(StrPas(Buf));
free(Buf);           //освобождаем память
MessageCount--;     //уменьшаем счетчик сообщений
}
}
}

```

Обратите внимание на то, что при чтении корреспонденции мы предполагаем, что текст представлен в однобайтовой ANSI-кодировке. Это соглашение следует учесть и при разработке метода, отправляющего сообщение.

#### **Замечание**

Кроме чтения содержимого сообщения, мы также имеем право узнать время его отправления. Помощь в этом окажет метод `FileGetDate()`.

Технология почтовых слотов не предусматривает возможности автоматического контроля за получением корреспонденции. Поэтому, для того чтобы организовать динамический просмотр почтового ящика, в приложении-сервере придется применить таймер или поток.

В секции частных объявлений опишите очередной метод `SendMail()`. Новый метод возьмет ответственность за отправку текстового сообщения `MailText` корреспонденту с адресом `MailAddress`. Реализация метода предложена в листинге 6.4.

#### **Листинг 6.4. Отправка почты**

```

void __fastcall TForm1::SendMail(String MailAddress,String MailText)
{
    char *Buf;
    unsigned hFile=FileCreate(MailAddress);/*обращаемся к слоту как
                                           к файлу используя имя слота*/
    if (hFile>0) //если получилось обратиться то:
    {
        char *Buf=(char *)malloc(MailText.Length()+1); //распределяем память
        ZeroMemory(Buf,sizeof(Buf));
        StrPCopy(Buf,MailText); //помещаем в буфер сообщение для передачи
        FileWrite(hFile,Buf,MailText.Length()+1);//отправляем сообщение в файл
        FileClose(hFile); //закрываем файл
        SysFreeMem(Buf); //освобождаем буфер
    }
}

```

Приложение уже почти готово. Нам осталось дважды щелкнуть по менеджеру команд `ActionManager1` и создать в нем команды `acSendMail` и `acReadMail`, отвечающие соответственно за отправку и получение почты. Листинг 6.5 раскрывает функциональную нагрузку этих команд.

#### **Листинг 6.5. Управление приложением**

```

//----- Чтение почты -----
void __fastcall TForm1::acReadMailExecute(TObject *Sender)
{
    ReadMail(hMailSlot);
}
//----- Отправка почты -----
void __fastcall TForm1::acSendMailExecute(TObject *Sender)
{
    // назначаем адрес компьютера-получателя
    // по правилу \\имя_компьютера\mailslot\[путь]имя_слота
    String MailAddress=MailSlotName;
    //в cbComputers->Text находится имя компьютера-получателя
}

```

```
MailAddress=cbComputers->Text+MailAddress.Delete(1,3);
//в StatusBar1->Panels->Items[0]->Text находится имя вашего компьютера
SendMail(MailAddress, StatusBar1->Panels->Items[0]->
        Text+": "+MemoOut->Text);
MemoOut->Clear();
}
```

Программирование завершено. Подключите команды к элементам управления (например, кнопкам). После этого разверните приложение на паре подключенных к сети компьютеров и протестируйте его работоспособность (рис. 5.5).

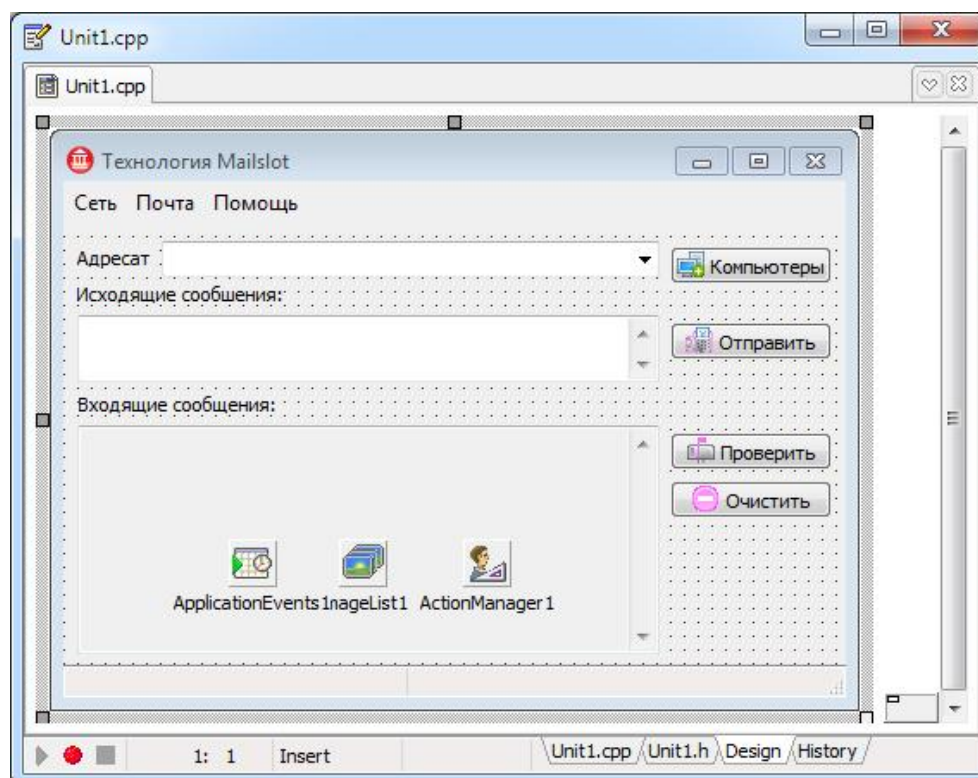


Рис. 6.2. Интерфейс приложения почтового слота

## Именованные каналы

Выбор термина "канал" (pipe) обусловлен тем, что связь между приложениями осуществляется по некоторому виртуальному каналу, проложенному в сети между двумя компьютерами. Механизм каналов построен таким образом, что приложения по большому счету и не задумываются, каким это чудесным образом их послания преодолевают расстояния между компьютерами. Приложения не интересуют, какие сетевые протоколы задействуются, а какие нет, как работают модемы или сетевые адаптеры. Это не их дело... Они считают, что канал — это разделяемая между процессами область памяти, используемая приложениями для связи между собой.

Каналы классифицируют как *неименованные* (anonymous pipe) и *именованные* (named pipe). Реализация анонимных каналов проще, они вообще не предназначены для работы в сети, но готовы выполнять функционал обмена данными между сервером и клиентом в рамках одного и того же компьютера. С помощью функции `CreatePipe()` сервер неименованного канала формирует разделяемую область памяти, а взамен получает два описывающих эту область дескриптора. Первый дескриптор предназначен для операций чтения из памяти, второй — для записи данных в эту же память. Операции чтения и записи осуществляются так, как будто клиент и сервер работают с файлом. Для этого превосходно подходят методы `FileRead()` и `FileWrite()` из модуля `SysUtils`. Для освобождения дескриптора, как всегда, вызывается метод `CloseHandle()`. Вот в двух словах перечень

подвигов, на которые способны неименованные каналы, нас же в большей степени интересуют их именованные коллеги. Ведь благодаря способностям именованного канала можно связать сервер и одно или несколько клиентских приложений, выполняющихся на разных компьютерах, узлами односторонней (симплексной) или двусторонней (дуплексной) связи.

## Определение имени именованного канала

Для определения имени канала во время разработки сервера программист должен придерживаться следующего синтаксиса: `\\pipe\<имя_канала>`.

Имя канала не чувствительно к регистру символов и не должно содержать более 256 символов. Например,

```
const wchar_t *PipeName = L"\\\\.\\pipe\\mypipe".
```

При обращении к именованному каналу клиентские приложения заменяют точку в его имени на сетевое имя сервера, предоставляющего услугу именованных каналов:

```
const wchar_t *PipeName = L"\\\\mainserver\\pipe\\mypipe";
```

## Создание именованного канала

Приложение-сервер, инициирующее создание именованного канала, вызывает весьма непростую функцию `CreateNamedPipe()`. Все особенности эксплуатации созданного при помощи нее именованного канала напрямую зависят от применяемых при вызове функции параметров. В свою очередь, характеристики созданного именованного канала в дальнейшем определяют порядок разработки клиентских приложений.

```
HANDLE WINAPI CreateNamedPipe (
    __in          LPCTSTR lpName,
    __in          DWORD dwOpenMode,
    __in          DWORD dwPipeMode,
    __in          DWORD nMaxInstances,
    __in          DWORD nOutBufferSize,
    __in          DWORD nInBufferSize,
    __in          DWORD nDefaultTimeOut,
    __in          LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

За присваивание каналу имени отвечает параметр `lpName`.

Второй параметр `dwOpenMode` решает триединую задачу: во-первых, определяет тип обмена данными и направление потока данных (табл. 6.1), во-вторых, устанавливает поведение метода при передаче больших объемов данных и управляет кэшированием (табл. 6.2) и, в-третьих, управляет атрибутами безопасности (табл. 6.3).

**Таблица 6.1.** Возможные значения аргументов для параметра `dwOpenMode`

| Режим доступа        | Права сервера   | Права клиента   | Тип обмена |
|----------------------|-----------------|-----------------|------------|
| PIPE_ACCESS_INBOUND  | Только чтение   | Только запись   | Симплекс   |
| PIPE_ACCESS_OUTBOUND | Только запись   | Только чтение   | Симплекс   |
| PIPE_ACCESS_DUPLEX   | Чтение и запись | Чтение и запись | Дуплекс    |

**Таблица 6.2.** Режимы обмена данными, определяемые параметром `OpenMode`

| Режим обмена         | Описание                                                                                                                                                                                                                                                                                    |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| FILE_FLAG_OVERLAPPED | Асинхронный режим ввода/вывода. Включение флага рекомендуется в случаях, когда объем передаваемых в канал данных достаточно велик и процесс их передачи может занимать ощутимый промежуток времени. Созданный с этим флагом канал допускает производить передачу данных в фоновом режиме. В |

|                                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                            | асинхронном режиме обмена функциями, осуществляющие передачу данных, не дожидаясь завершения отправки сообщения и немедленно возвращают управление программе. При включенном флаге <code>FILE_FLAG_OVERLAPPED</code> операции ввода и вывода должны быть реализованы с неизменным заполнением структуры <code>OVERLAPPED</code>                                                                                                                                                                                                                                                                                                    |
| <code>FILE_FLAG_WRITE_THROUGH</code>       | Этот флаг имеет смысл, только если данные передаются потоком байтов. Если флаг включен, то данные немедленно направляются в канал, если же флаг отключен, то перед началом передачи данные накапливаются в кэше и отправляются к корреспонденту в составе единого пакета только после его заполнения. Таким образом, оптимизируется сетевой трафик, т. к. вместо сотни одиночных отправок передается объединенный пакет сообщений. Если этот флаг при создании канала не применялся, то основные параметры пакета (размер и время ожидания) настраиваются операционной системой или методом <code>SetNamedPipeHandleState()</code> |
| <code>FILE_FLAG_FIRST_PIPE_INSTANCE</code> | Допускает создание только одного экземпляра канала, попытка создания второго экземпляра вызывает ошибку                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

Таблица 6.3. Атрибуты безопасности, определяемые параметром *OpenMode*

| Флаг                                | Описание                                                                                                                                           |
|-------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>WRITE_DAC</code>              | Вызывающий процесс должен иметь права доступа на запись к произвольному управляющему списку доступа именованного канала (access control list, ACL) |
| <code>WRITE_OWNER</code>            | Вызывающий процесс должен иметь права доступа на запись к процессу, владеющему именованным каналом                                                 |
| <code>ACCESS_SYSTEM_SECURITY</code> | Процесс должен иметь права доступа на запись к управляющему списку доступа именованного канала ACL                                                 |

Наступил черед третьего по счету параметра — `dwPipeMode`, он определяет:

- ☐ каким образом данные направляются в канал — в виде последовательности байтов `PIPE_TYPE_BYTE` или как поток сообщений `PIPE_TYPE_MESSAGE`;
- ☐ как производится чтение данных из канала — как поток байтов `PIPE_READMODE_BYTE` или поток `PIPE_READMODE_MESSAGE` сообщений;
- ☐ как будет происходить возврат управления методами, осуществляющими запись или чтение данных. Флаг `PIPE_WAIT` приказывает этим методам не отдавать управление программе, пока не будет завершена операция (такой режим называют синхронным блокирующим). Напротив, флаг `PIPE_NOWAIT` разрешает методам возвращать управление программе немедленно (синхронный неблокирующий режим).

#### Замечание

Режимы передачи данных в канал и чтение данных из него должны быть согласованы. Если отправляем сообщение в виде потока байтов `PIPE_TYPE_BYTE`, то и прием обязательно должен осуществляться потоком `PIPE_READMODE_BYTE`. Режим `PIPE_TYPE_MESSAGE` менее критичен и допускает чтение с любыми значениями флагов.

Четвертый параметр `nMaxInstances` накладывает ограничение на максимальное число экземпляров создаваемого канала. Допускается любое число в пределах от 1 до `PIPE_UNLIMITED_INSTANCES` (255 соединений). Наиболее интересным (но не единственным) решением, реализующим многоканальное приложение, станет создание для каждого канала отдельного программного потока. В этом случае клиентское приложение обращается к серверу в своем выделенном канале, а обслуживание клиента на серверной стороне осуществляется в отдельном потоке. В качестве другого варианта можно рассмотреть поочередное предоставление одного и того же канала для нескольких клиентов.

Пятый и шестой параметры метода (`nOutBufferSize` и `nInBufferSize`) определяют размеры выходного и входного буферов, в которых кэшируются отправляемые и принимаемые

данные. Определение размера — дело творческое, безусловно, размер должен быть согласован с объемом передаваемых по каналам сообщений. Однако при этом не стоит упускать из виду, что за резервирование памяти для буферов отвечает ядро операционной системы, с созданием каждого канала его буферам отводится своя область в памяти, причем память выделяется не в страничном пуле, а прямо в ОЗУ. Так что при определении размера кэша не стоит разбрасываться мегабайтами...

Время таймаута ожидания клиентом канала в миллисекундах определяет параметр `nDefaultTimeout`. Установленное в нем значение имеет смысл лишь тогда, когда на клиентской стороне процесс ожидания освобождения канала обеспечивается методом `WaitNamedPipe()` с включенным флагом `NMPWAIT_USE_DEFAULT_WAIT`.

Последний параметр `lpSecurityAttributes` представляет собой указатель на структуру, определяющую политику безопасности в вопросе получения доступа к дескриптору создаваемого именованного канала различными клиентскими процессами. По умолчанию этот параметр допускает передачу пустого указателя `NULL`.

Некоторые изменения в настройки уже созданного экземпляра канала можно внести с помощью функции

```
BOOL WINAPI SetNamedPipeHandleState(
    __in HANDLE hNamedPipe,
    __in LPDWORD lpMode,
    __in LPDWORD lpMaxCollectionCount,
    __in LPDWORD lpCollectDataTimeout);
```

Первый параметр — дескриптор канала. Параметр `lpMode` определяет, каким образом данные передаются в канал (флаг `PIPE_READMODE_BYTE` или `PIPE_READMODE_MESSAGE`), и как производится возврат из методов, обратившихся к каналу (флаг `PIPE_WAIT` или `PIPE_NOWAIT`). Еще два указателя `lpMaxCollectionCount` и `lpCollectDataTimeout` имеют смысл, если канал создавался без флага `FILE_FLAG_WRITE_THROUGH`. В этом случае указатели определяют основные параметры передаваемого пакета. Для оптимизации сетевого трафика перед отправкой данные накапливаются в пакете и передаются в канал, только после достижения пакетом определенного размера `lpMaxCollectionCount` или по истечении времени ожидания `lpCollectDataTimeout`.

## Управление соединением с клиентом

Создавший именованный канал сервер на правах старшинства управляет процессом подключения к каналу и отключения от него клиентских приложений. Такая привилегия подкреплена двумя серьезными функциями — `ConnectNamedPipe()` и `DisconnectNamedPipe()`. Обращение к `ConnectNamedPipe()` переводит сервер в режим ожидания, пока к экземпляру канала не подключится какое-нибудь клиентское приложение. Вызов функции целесообразен сразу после создания экземпляра канала (в особенности, если сервер создает в отдельных потоках сразу несколько каналов) или сразу после отключения от канала клиента.

```
BOOL WINAPI ConnectNamedPipe(
    __in HANDLE hNamedPipe,
    __in LPOVERLAPPED lpOverlapped);
```

Здесь: `hNamedPipe` — дескриптор именованного канала; `LPOVERLAPPED` — указатель на структуру, используемую при операциях асинхронного ввода/вывода. Указатель на эту структуру передается лишь в тех случаях, когда канал был создан с применением флага `FILE_FLAG_OVERLAPPED`. Если канал перевелся в режим готовности к подключению, метод возвратит значение `true`. Для канала, созданного в синхронном блокирующем режиме (с использованием `PIPE_WAIT`), рассматриваемая функция переходит в состояние ожидания соединения с клиентским процессом. Если канал создан в синхронном неблокирующем

режиме (PIPE\_NOWAIT), метод `ConnectNamedPipe()` немедленно возвращает управление с кодом `true`, если только клиент был отключен от данной реализации канала и возможно подключение этого клиента. В противном случае возвращается значение `false`.

#### **Замечание**

Возврат функцией `ConnectNamedPipe()` значения `false` не может считаться однозначным признаком того, что подключение завершилось неудачно. После выполнения этой функции программисту следует проанализировать значение последней ошибки. Так `GetLastError()` предоставит код ошибки `ERROR_PIPE_LISTENING` в случае, если вообще нет желающих для подключения к именованному каналу. Кроме этого, высока вероятность встретить ошибки: `ERROR_PIPE_CONNECTED` — канал уже занят, `ERROR_NO_DATA` — клиент отказался от канала, но не был отключен от него сервером с помощью `DisconnectNamedPipe()`.

Особенность совместной работы клиента и сервера именованных каналов в том, что отказ клиента от канала (вызов им функции `CloseHandle()`) физически не отключает его от сервера. Канал по-прежнему остается в занятом состоянии, хотя данные по нему уже не транслируются. В клиент-серверной технологии окончательный разрыв связи считается привилегией сервера, в частности при работе с именованным каналом для этого вызывается функция

```
BOOL WINAPI DisconnectNamedPipe(__in HANDLE hNamedPipe);
```

Программная логика сервера перед отключением канала должна предусматривать действия, заставляющие принудительно передать в ресурс все накопленные в кэше данные. Для этого обращаемся к `FlushFileBuffers()`.

## **Состояние канала**

Как сервер именованных каналов, так и клиентское приложение имеют право ознакомиться с основными техническими характеристиками работающего канала. Для этого предназначена функция

```
BOOL WINAPI GetNamedPipeInfo(
    __in        HANDLE hNamedPipe,
    __out       LPDWORD lpFlags,
    __out       LPDWORD lpOutBufferSize,
    __out       LPDWORD lpInBufferSize,
    __out       LPDWORD lpMaxInstances);
```

Первый параметр метода — дескриптор опрашиваемого канала, все остальные параметры информационные. Так `lpFlags` расскажет о типе канала (поток байтов `PIPE_TYPE_BYTE` или поток сообщений `PIPE_TYPE_MESSAGE`) и кем вызван метод — сервером `PIPE_SERVER_END` или клиентом `PIPE_CLIENT_END`. Параметры `lpOutBufferSize` и `lpInBufferSize` проинформируют о размерах выходного и входного буферов. Параметр `lpMaxInstances` вернет данные о максимально допустимом количестве каналов. В случае успешного завершения функция возвращает значение `true`, а при ошибке — `false`.

Передав дескриптор именованного канала `hPipe` в функцию

```
BOOL WINAPI GetNamedPipeHandleState(
    __in        HANDLE hNamedPipe,
    __out       LPDWORD lpState,
    __out       LPDWORD lpCurInstances,
    __out       LPDWORD lpMaxCollectionCount,
    __out       LPDWORD lpCollectDataTimeout,
    __out       LPTSTR lpUserName,
    __in        DWORD nMaxUserNameSize);
```

мы получим подробные сведения о канале.

О состоянии канала расскажет параметр `lpState`. Этот указатель на двойное слово `DWORD` может вернуть два флага. Флаг `PIPE_READMODE_MESSAGE` свидетельствует о том, что канал специализируется на передаче сообщений. Флаг `PIPE_NOWAIT` признак того, что при

обращении к каналу функции будут возвращать управление в программу немедленно, не дожидаясь окончания выполнения.

О числе существующих экземпляров канала поведает параметр `lpCurInstances`.

Клиент именованного канала осуществляет передачу данных не по мере их поступления, а накапливает их в пакет и потом отправляет серверу в составе пакета. Максимальный размер пакета (в байтах) будет возвращен при помощи указателя `lpMaxCollectionCount`.

Чтобы информация, находящаяся в пакете, была гарантированно передана серверу (даже если пакет не будет наполнен целиком) процесс передачи дополнительно контролируется таймером. О времени таймаута нас информирует параметр `lpCollectDataTimeout`.

#### **Замечание**

К сожалению, нельзя обращаться к параметрам: `lpMaxCollectionCount` и `lpCollectDataTimeout` в случае, если серверный и клиентский концы канала находятся на одной и той же станции. В этом случае вместо указателя на переменную надо передавать `NULL`.

Если вызов метода осуществляется со стороны сервера и клиент получил доступ к каналу с флагом `SECURITY_IMPERSONATION`, то также возможно узнать об имени пользователя, работающего на клиентском компьютере. Эти данные возвратит указатель `lpUserName`. При этом максимальная длина имени ограничивается последним параметром `nMaxUserNameSize`.

## **Подключение к каналу клиентского приложения**

Подключение клиентского приложения к каналу осуществляется либо методом `FileCreate()`, либо при помощи метода `CallNamedPipe()`. Первый вариант подходит в ситуациях, когда по каналу предполагается передавать поток байтов, второй, когда канал специализируется на работе с сообщениями. Если канал свободен, то в метод возвращает дескриптор канала, если же канал занят, то клиентское приложение имеет право подождать освобождения канала, но для этого потребуется вызвать метод `WaitNamedPipe()`.

### **Функция**

```
BOOL WINAPI CallNamedPipe(
    __in          LPCTSTR lpNamedPipeName,
    __in          LPVOID lpInBuffer,
    __in          DWORD nInBufferSize,
    __out         LPVOID lpOutBuffer,
    __in          DWORD nOutBufferSize,
    __out         LPDWORD lpBytesRead,
    __in          DWORD nTimeout);
```

подключается к именованному каналу, при необходимости ждет его освобождения, осуществляет одновременную операцию чтения и записи и отключается от канала. Здесь: `lpNamedPipeName` — имя канала; `lpInBuffer` — указатель на входной буфер записи (в котором будут накапливаться данные, направляемые в канал); `nInBufferSize` — размер буфера записи в байтах; `lpOutBuffer` и `nOutBufferSize` — соответственно указатель и размер буфера чтения; `lpBytesRead` — переменная, в которую будут передаваться данные о количестве байтов, считанных из канала. Время ожидания готовности канала определяется в миллисекундах и находится в последнем параметре — `nTimeout`. Кроме этого, вместо точного значения таймаута в этот параметр допустимо передавать одну из констант, представленных в табл. 6.4.

*Таблица 6.4. Управление таймаутом в функции `CallNamedPipe()`*

| Значение                          | Описание                                                                                          |
|-----------------------------------|---------------------------------------------------------------------------------------------------|
| <code>NMPWAIT_NOWAIT</code>       | Никогда не ждать. Если канал по какой-то причине недоступен — сразу возвращать ошибку подключения |
| <code>NMPWAIT_WAIT_FOREVER</code> | Неограниченное ожидание                                                                           |

|                          |                                                                                                                |
|--------------------------|----------------------------------------------------------------------------------------------------------------|
| NMPWAIT_USE_DEFAULT_WAIT | В качестве таймаута применять время, определенное сервером в параметре DefaultTimeout метода CreateNamedPipe() |
|--------------------------|----------------------------------------------------------------------------------------------------------------|

Для организации ожидания занятого канала клиентское приложение имеет право воспользоваться функцией

```
BOOL WINAPI WaitNamedPipe( __in LPCTSTR lpNamedPipeName,
                          __in DWORD nTimeout);
```

Название канала передается указателем lpNamedPipeName(), время ожидания назначается в миллисекундах во втором параметре nTimeout функции. Кроме этого, таймаут воспринимает две константы: NMPWAIT\_WAIT\_FOREVER и NMPWAIT\_USE\_DEFAULT\_WAIT. Константа NMPWAIT\_WAIT\_FOREVER переводит метод в состояние вечного ожидания, в этом состоянии WaitNamedPipe() не возвращает управления вызвавшей ее программе до тех пор, пока экземпляр вызываемого канала не станет доступным. Вторая константа NMPWAIT\_USE\_DEFAULT\_WAIT определяет, что время ожидания должно соответствовать времени таймаута, установленного при создании именованного канала. В результате, если запрашиваемый канал освободился за интервал времени, установленный в таймауте, то метод вернет значение true. Если по указанному адресу вообще отсутствуют экземпляры каналов, то метод завершит работу немедленно, не обращая внимания на значение своего таймаута.

## Разработка класса сервера именованного канала

При проектировании сервера именованного канала стоит объединить достоинства высокоскоростных функций прикладного программного интерфейса Windows API и удобства объектно-ориентированного языка C++ Builder и реализовать сервер в виде самостоятельного класса.

Листинг 6.6 содержит определение класса TPipeServer, способного создать отдельный именованный канал. Класс содержит минимальный набор полей и методов. Поля fHandle и Overlapped предназначены для хранения дескриптора канала и структуры, обеспечивающей асинхронный режим работы. Помимо конструктора и деструктора в арсенал возможностей класса войдет метод установления соединения и методы чтения/записи данных.

**Листинг 6.6. Объявление класса сервера именованного канала**

```
class TPipeServer
{
private: HANDLE fHandle;
        _OVERLAPPED Overlapped;
        unsigned fTimeout; // = 0
public:
    __fastcall TPipeServer(String PipeName, DWORD BufferSize);
    __fastcall ~TPipeServer();

    bool __fastcall Connect();
    int __fastcall Write(void *buf, int Count);
    int __fastcall Read(void *buf, int Count);
    String __fastcall GetClientComputerName();

    HANDLE __property Handle = {read=fHandle};
    unsigned __property Timeout = {read=fTimeout, write=fTimeout};
};
```

Канал создается в рамках конструктора класса (листинг 6.7).

**Листинг 6.7. Конструктор класса сервера именованного канала**

```

__fastcall TPipeServer::TPipeServer(String PipeName, DWORD BufferSize)
{
    //инициализация атрибутов безопасности
    _SECURITY_DESCRIPTOR SD;
    InitializeSecurityDescriptor(&SD, SECURITY_DESCRIPTOR_REVISION);

    _SECURITY_ATTRIBUTES SA;
    SA.nLength=sizeof(_SECURITY_ATTRIBUTES);
    SA.lpSecurityDescriptor=&SD;
    SA.bInheritHandle=true;

    //подготовка структуры для асинхронного режима
    ZeroMemory(&fOverlapped, sizeof(_OVERLAPPED));
    fOverlapped.hEvent= CreateEvent (NULL, true, false, NULL);

    BufferSize=4096;
    TimeOut=0;
    //создание канала
    fHandle = CreateNamedPipe(PipeName.w_str(),
        PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
        PIPE_WAIT | PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE,
        PIPE_UNLIMITED_INSTANCES,
        BufferSize,
        BufferSize,
        TimeOut,
        &SA);

    if (Handle == INVALID_HANDLE_VALUE | Handle==0)
    {
        throw (Exception("Ошибка создания канала!"));
    }
}

```

В первых строках метода иницилируем структуру `_SECURITY_ATTRIBUTES` для того, чтобы предоставить возможность обращаться к каналу клиентским процессам, не обладающим правами администратора. Затем распределяем память под структуру `Overlapped`, поддерживающую асинхронный режим работы, и создаем событие синхронизации. Для упрощения примера в конструктор передаются всего два аргумента: имя канала `PipeName` и размер входного и выходного буферов именованного канала — `BufferSize`. Далее формируем канал с наиболее востребованными характеристиками: дуплексный, асинхронный, работающий в режиме обмена сообщениями, с максимально допустимым числом экземпляров.

После создания экземпляра канала наступает очередь метода `Connect()`, отвечающего за перевод сервера в режим готовности к соединению (листинг 6.8).

#### Листинг 6.8. Метод соединения

```

bool __fastcall TPipeServer::Connect()
{
    bool Result=ConnectNamedPipe(Handle, &fOverlapped); /*готовность к соединению*/
    if (Result==false)
    {
        switch (GetLastError())
        {
            case ERROR_PIPE_CONNECTED: Result= true; break;
            case ERROR_IO_PENDING :
                if (WaitForSingleObject(fOverlapped.hEvent, INFINITE)
                    == WAIT_OBJECT_0) Result= true; break;
            default : throw Exception("Не готов к соединению!");
        }
    }
    return (Result);
}

```

```
}
```

Настал черед методов, отвечающих за операции чтения/записи в канал. Для этого достаточно задействовать классические для Windows API функции чтения и записи в файл (листинги 6.9 и 6.10).

#### Листинг 6.9. Метод чтения данных

```
int __fastcall TPipeServer::Read(void *buf, int Count)
{
    unsigned long NumberOfBytesToRead=0;

    if (!ReadFile(Handle, buf, Count, &NumberOfBytesToRead, &fOverlapped))
    {
        switch (GetLastError())
        { case ERROR_IO_PENDING:
            WaitForSingleObject(fOverlapped.hEvent, INFINITE);
            GetOverlappedResult(Handle, &fOverlapped, &NumberOfBytesToRead, true);
            break;
          case ERROR_BROKEN_PIPE: { throw Exception("Канал разорван!"); break; }
          default : throw Exception("Сбой чтения, ошибка"); break;
        }
    };

    return (NumberOfBytesToRead);
}
```

#### Листинг 6.10. Метод записи данных

```
int __fastcall TPipeServer::Write(void *buf, int Count)
{
    unsigned long NumberOfBytesToWrite=0;

    if (!WriteFile(Handle, buf, Count, &NumberOfBytesToWrite, &fOverlapped))
    { switch (GetLastError())
        { case ERROR_IO_PENDING :
            WaitForSingleObject (fOverlapped.hEvent, INFINITE); break;
          case ERROR_NO_DATA : throw Exception("Ошибка записи"); break;
        }
    }
    GetOverlappedResult(Handle, &fOverlapped, &NumberOfBytesToWrite, true);
    return (NumberOfBytesToWrite);
}
```

Работа над классом завершается описанием канала деструктором (листинг 6.11). В коде деструктора предусматривается не только уничтожение экземпляра канала, но и очистка буфера с данными и разрушение события синхронизации.

#### Листинг 6.11. Деструктор класса сервера именованного канала

```
__fastcall TPipeServer::~TPipeServer()
{
    FlushFileBuffers(Handle);
    DisconnectNamedPipe(Handle);
    CloseHandle(fOverlapped.hEvent);
    CloseHandle(Handle);
}
```

## Разработка класса клиента именованного канала

Разработаем класс `TPipeClient`, предназначенный для обслуживания именованного канала на клиентской стороне. Клиентский класс более прост, т. к. пользуется услугами уже существующего канала, поэтому кроме конструктора и деструктора нам потребуются всего два метода чтения и записи (листинг 6.12).

**Листинг 6.12. Объявление класса клиента именованного канала**

```
class TPipeClient
{
private:
    HANDLE fHandle;
    _OVERLAPPED OverlappedRead, OverlappedWrite;
public:
    __fastcall TPipeClient(String PipeName);
    __fastcall ~TPipeClient();
    int __fastcall Write(void *buf, int Count);
    int __fastcall Read(void *buf, int Count);
    HANDLE __property Handle= {read=fHandle};
};
```

Основная особенность клиента именованного канала в том, что он работает с каналом так, как будто это обычный файл. Этим объясняется вся логика функционирования класса `TPipeClient`. В момент создания экземпляра класса мы "обманываем" клиента, предоставляя в его распоряжение дескриптор канала под видом дескриптора файла (листинг 6.13).

**Листинг 6.13. Реализация конструктора класса клиента именованного канала**

```
__fastcall TPipeClient::TPipeClient(String PipeName)
{
    fHandle = CreateFile(PipeName.w_str(),
                        GENERIC_READ | GENERIC_WRITE,
                        0,
                        NULL,
                        OPEN_EXISTING, FILE_FLAG_OVERLAPPED, 0);

    String msg;
    if (Handle==NULL)
        switch (GetLastError())
        {
            case ERROR_SEEK_ON_DEVICE : {
                msg=Format("Канал %s не обнаружен", ARRAYOFCONST((PipeName)));
                throw (Exception(msg));
                break;
            }
            case ERROR_FILE_NOT_FOUND : {
                msg=Format("Канал %s не обнаружен", ARRAYOFCONST((PipeName)));
                throw (Exception(msg));
                break;
            }
            default : {
                msg=Format("Ошибка канала!", ARRAYOFCONST((PipeName)));
                throw (Exception(msg));
                break;
            }
        }

    ZeroMemory(&OverlappedRead, sizeof(OverlappedRead));
    ZeroMemory(&OverlappedWrite, sizeof(OverlappedWrite));

    OverlappedRead.hEvent = CreateEvent(NULL, true, false, NULL);
    OverlappedWrite.hEvent= CreateEvent(NULL, true, false, NULL);
}
```

Операции чтения и записи осуществляются стандартными функциями Windows `ReadFile()` и `WriteFile()`, при этом мы не забываем, что нам следует учитывать асинхронный режим доступа к каналу (листинги 6.14 и 6.15).

**Листинг 6.14. Метод чтения**

```
int __fastcall TPipeClient::Read(void *buf, int Count)
{
    unsigned long NumberOfBytesToRead=0;

    if (!ReadFile(Handle, buf, Count, &NumberOfBytesToRead, &OverlappedRead))
    {
        switch (GetLastError())
        {
            case ERROR_IO_PENDING:
                WaitForSingleObject(OverlappedRead.hEvent, INFINITE);
                GetOverlappedResult(Handle, &OverlappedRead,
                                    &NumberOfBytesToRead, true);

                break;
            case ERROR_BROKEN_PIPE: throw Exception("Канал разорван!"); break;
            default                  : throw Exception("Сбой чтения, ошибка");
        }
    };
    return (NumberOfBytesToRead);
}
```

**Листинг 6.15. Метод записи**

```
int __fastcall TPipeClient::Write(void *buf, int Count)
{
    unsigned long NumberOfBytesToWrite=0;

    if (WriteFile(Handle, buf, Count,
                  &NumberOfBytesToWrite, &OverlappedWrite) ==false)
    {
        switch (GetLastError())
        {
            case ERROR_IO_PENDING :
                WaitForSingleObject (OverlappedWrite.hEvent, INFINITE); break;
            case ERROR_NO_DATA    : throw Exception("Ошибка записи"); break;
        }
    }
    GetOverlappedResult(Handle, &OverlappedWrite,
                        &NumberOfBytesToWrite, true);
    return (NumberOfBytesToWrite);
}
```

#### **Примечание**

Объем книги не позволяет предоставить листинги приложений сервера и клиента именованных каналов в полном объеме. Вы их найдете в электронном архиве к книге.

## **Сокеты**

Одним из наиболее востребованных механизмов взаимодействия сетевых приложений считается технология *сокетов* (sockets). Важное преимущество сокетов в том, что они поддерживаются практически всеми современными операционными системами и поэтому позволяют организовывать сетевой обмен между ОС различных производителей.

#### **Замечание**

Появлению сокетов мы в первую очередь обязаны Калифорнийскому университету Беркли и операционной системе UNIX, в которой идея нашла свое первое применение.

Технология сокетов построена по принципу "клиент-сервер". Сразу после создания сервер переходит в режим ожидания связи (для этого предложен термин "*прослушивание*" —

listening). Услышав "стук в дверь", сервер открывает сеанс с клиентом, в рамках которого осуществляется односторонний или двусторонний обмен данными.

В отличие от почтовых слотов и каналов (для которых не существует официальных компонентов VCL) сокеты более удачливы. Обратившись к страницам **Indy Servers** и **Indy Clients** палитры компонентов, вы обнаружите десятки компонентов, инкапсулирующих интерфейс сокетов. По естественным причинам мы не сможем рассмотреть их всех, поэтому ограничимся лишь парой наиболее показательных классов: `TIdTCPServer` и `TIdTCPClient`.

#### **Внимание!**

В составе C++ Builder имеется большой перечень компонентов Indy, существенно упрощающих проектирование сетевых приложений. Эти компоненты разработаны вне стен Embarcadero, более подробную информацию вы сможете получить на сайте <http://www.indyproject.org>.

## Сервер `TIdTCPServer`

Компонент `TIdTCPServer` предназначен для создания серверных окончаний сетевых проектов, работающих на основе стека протоколов TCP/IP с использованием сокетов.

Компонент `TIdTCPServer` (как, впрочем, и подавляющее большинство других компонентов Indy) использует блокирующий режим вызова сокетов. Другими словами, осуществляется синхронный режим обработки в котором операции ввода/вывода не возвращают управление вызвавшему их приложению до тех пор, пока не получат подтверждения от другого приложения. Таким образом, работа приложения приостанавливается на время таймаута. Если по истечению таймаута ответ не был получен — генерируется исключительная ситуация.

В классе `TIdTCPServer` реализован многопоточный принцип обработки запросов клиентов. После активации сервера запускается поток прослушивания и сервер переходит в состояние ожидания клиентов. Получив запрос на обслуживание и осуществив так называемое “рукопожатие” с клиентом сервер создаёт отдельный сокет и дальнейший сеанс работы клиент-сервер осуществляется в отдельном сокете.

Каждое отдельное соединение с клиентом сервер рассматривает как отдельную задачу. Указанная задача отдается под управление специальному планировщику (экземпляру класса `TIdScheduler`). Планировщик обрабатывает соединение, исполняет задачу и завершает её выполнение.

#### **Замечание**

Существует два типа планировщиков доступных для сервера `TIdTCPServer`: на основе потоков (thread-based) и нитей (fiber-based). По умолчанию создается планировщик, ориентированный на работу в отдельном потоке.

Подготовка сервера к работе начинается с настройки свойства

`__property Word DefaultPort;`

определяющего номер порта который станет прослушивать сервер во время подготовки к установке нового соединения.

При необходимости допускается ограничить максимальное число клиентских соединений, для этого обращаемся к свойству

`__property int MaxConnections; //по умолчанию 0 — нет ограничений`

Включение сервера производится путем установки в `true` свойства

`__property bool Active;`

Ключевым событием сервера выступает

`__property TIdServerThreadEvent OnExecute;`

Именно здесь производится обслуживание клиента. Обработчик события возвращает всего один параметр — ссылку на контекст `TIdContext* AContext`. Это ничто иное как программное представление соединения клиента и сервера в рамках текущей сессии.

Простейшее приложение сервера способного получать от клиента текстовые сообщения предложено вниманию читателя в листинге 6.16.

**Листинг 6.16. Приложение-сервер обработки текста**

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    IdTCPServer1->DefaultPort=6000;
    IdTCPServer1->MaxConnections=16;
}
//-----
void __fastcall TForm1::IdTCPServer1Execute(TIdContext *AContext)
{
    String txt=AContext->Connection->IOHandler->ReadLn();
    Memo1->Lines->Add(txt);
    AContext->Connection->IOHandler->WriteLn("OK");
}
```

## Клиент TIdTCPClient

Основу клиентской стороны сетевого проекта, построенного на основе сокетов с использованием стека протоколов TCP/IP представляет компонент `TIdTCPClient`. Для того, чтобы клиент смог отправить серверу сообщение нам необходимо совершить несколько простых шагов.

Во-первых, надо определиться с адресом и портом сервера. Для этого обращаемся к свойствам:

```
__property UnicodeString Host; //адрес хоста
__property Word Port; //номер порта хоста
```

Адрес хоста может быть задан как в формате IP-адреса, так и в виде символического имени компьютера.

При необходимости осуществить аутентификацию/авторизацию пользователя следует задействовать пару свойств

```
__property UnicodeString Username;
__property UnicodeString Password;
```

Во-вторых, собственно для осуществления подключения к серверу обращаемся к методу

```
virtual void __fastcall Connect(void);
```

Допустим, что адрес сервера заносится пользователем в строку ввода `Edit1`, а собственно команда на соединение сервером передается при посредничестве переключателя `CheckBox1`. В таком случае для инициации процесса соединения можно воспользоваться примером из листинга 6.17.

**Листинг 6.17. Подключение клиента к серверу**

```
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
    if ((CheckBox1->Checked == True) && (Edit1->Text!=""))
    {
        IdTCPClient1->Host = Edit1->Text; //можно передать IP-адрес
        IdTCPClient1->Port = 6000;
        try
        { IdTCPClient1->Connect(); }
    }
```

```

catch (EIdException& E) //EidSocketError& E
{
    switch (MessageDlg("Ошибка подключения",
        mtError, mbAbortRetryIgnore, 0))
    {
        case mrRetry: {CheckBox1->OnClick(Sender); break;};
    };
    CheckBox1->Checked = False;
}
} else IdTCPClient1->Disconnect();
}

```

В примере упоминается ещё один важный метод

```
void __fastcall Disconnect(void);
```

Позволяющий отключить клиента от сервера.

В-третьих, о факте успешного соединения можно судить по генерации события

```
__property TNotifyEvent OnConnected;
```

Соответственно отключение от сервера сопровождается событием

```
__property TNotifyEvent OnDisconnected;
```

Наконец, в-четвертых. Для отправки в адрес сервера запроса (команды, текстового сообщения) нам понадобятся услуги инкапсулированного в клиентский компонент экземпляра класса `TIdIOHandler`, доступ к которому предоставит свойство клиента

```
__property TIdIOHandler* IOHandler;
```

Сразу отметим что `TIdIOHandler` предназначен для поддержки операций ввода/вывода и для этих целей в буквальном смысле наспигован многочисленными методами чтения/записи. Однако для реализации простейшего примера в котором клиент сможет отправить серверу текстовое сообщение (листинг 6.18) нам достаточно обратиться к простейшему методу записи строки `WriteLn()`.

#### Листинг 6.18. Отправка текстового сообщения

```

IdTCPClient1->IOHandler->WriteLn(Edit2->Text);
Edit2->Text="";
String s=IdTCPClient1->IOHandler->ReadLn();
if (s=="OK") ShowMessage("Сообщение получено сервером");

```

В предложенном примере задействован ещё один метод экземпляра класса `TIdIOHandler` — это `ReadLn()`. Названный метод позволяет прочитать текстовые данные возвращаемые сервером в адрес клиента.

У сокета реализовано несколько перегружаемых методов `Write()` и `Read()` способных работать с наиболее востребованными основными типами данных, а также с файлами и потоками.

## Пример приложения “Контроль рабочих столов”

Рассмотрим порядок работы с сокетами на примере интересной задачи — нам требуется разработать сетевое приложение, позволяющее администратору сети наблюдать за рабочими столами компьютеров рабочей группы. Подобное приложение может оказаться полезным в школах и вузах, где на занятиях по информационным технологиям педагогу необходимо контролировать действия своих подопечных.

Решить подобную задачу можно, разработав приложение-сервер, получающее от клиентских приложений экранные снимки по транспорту сокетов.

### Сокет-клиент

Каким образом научить клиентское приложение получать экранный снимок рабочего стола? На самом деле, в этом нет ничего сложного. Нам следует получить дескриптор дисплея, узнать размеры рабочего стола и с помощью функции `BitBlt()` скопировать пиксели экрана в битовый образ, созданный функцией `CreateCompatibleBitmap()`. Код функции, решающей эту задачу, едва превышает 10 строк (листинг 6.19), а в результате мы получаем дескриптор битового образа.

**Листинг 6.19. Получение битового образа со снимком рабочего стола**

```
HBITMAP GetScreenBitmap()
{
    HDC DC=CreateDC(L"DISPLAY",NULL, NULL, NULL); //дескриптор экрана
    HDC memDC=CreateCompatibleDC(DC); //совместимый дескриптор в памяти

    int W=GetDeviceCaps(DC,HORZRES); //горизонтальный размер
    int H=GetDeviceCaps(DC,VERTRES); //вертикальный размер

    HBITMAP BITMAP=CreateCompatibleBitmap(DC,W,H); //создаем битовый образ
    SelectObject(memDC, BITMAP); //передаем образ в память
    BitBlt(memDC,0,0,W,H,DC,0,0,SRCCOPY); //копируем экран в память

    DeleteDC(memDC);
    DeleteDC(DC);
    return (BITMAP);
}
```

После этого нам осталось написать функцию, позволяющую при посредничестве компонента `TIdTCPClient *IdTCPClient1` отправлять по заданному адресу снимок экрана. При разработке клиентской функции необходимо учесть тот факт, что растровая картинка современного экрана с разрешением  $1920 \times 1080$  пиксела с глубиной цвета 24 бит/пиксел займет примерно 8 Мбайт памяти, поэтому нам следует задуматься над решением проблемы снижения передаваемого в сеть потока данных. В листинге 6.20 для этих целей мы воспользовались услугами графического формата JPEG, который позволяет существенно сжимать растровые рисунки.

**Листинг 6.20. Передача снимка в адрес серверного сокета**

```
void __fastcall TfrmClient::SendScreenShot(String RemoteHost,
                                           Word RemotePort)
{
    IdTCPClient1->Host=RemoteHost; //адрес сервера
    IdTCPClient1->Port=RemotePort; //порт сервера
    IdTCPClient1->Connect(); //запрос соединения

    if (IdTCPClient1->Connected())
    {
        __try
        {
            TBitmap *B=new TBitmap;
            B->Handle=GetScreenBitmap(); //подключаем снимок рабочего стола

            //подключите #include <Vcl.Imaging.jpeg.hpp>
            TJPEGImage *J=new TJPEGImage(); //с помощью JPEG сжимаем снимок
            J->Assign(B); //забираем исходную картинку
            J->ProgressiveEncoding=true; //прогрессивная кодировка
            J->CompressionQuality=50; //степень сжатия JPEG
            J->JPEGNeeded(); //сжимаем образ
        }
        __except {}
    }
}
```

```

TMemoryStream *MS=new TMemoryStream();
J->SaveToStream(MS);          //загружаем снимок в поток MS
MS->Seek(0,0);                 //позиционируемся на первом байте потока

IdTCPClient1->IOHandler->Write(MS,MS->Size); //отправляем снимок
}
__finally
{
    if (B!=NULL) delete B;
    if (J!=NULL) delete J;
    if (MS!=NULL) delete MS;
}
IdTCPClient1->Disconnect();
}
}

```

## Сокет-сервер

Для реализации приложения-сервера нам потребуются три ингредиента:

- ❑ сокет-сервер TIdTCPServer;
- ❑ компонент TImage\* Image1, способный отобразить поступивший из сети рисунок со снимком рабочего стола.

Реализацию сервера начните с назначения номера порта и активации компонента TIdTCPServer, эти операции можно осуществить в момент создания главной формы приложения. А основной код сервера, предназначенный для получения снимка удаленного рабочего стола вы найдете в листинге 6.21.

### Листинг 6.21. Событие OnExecute() компонента TIdTCPServer

```

void __fastcall TfrmServer::IdTCPServer1Execute(TIdContext *AContext)
{
    TMemoryStream *MS=new TMemoryStream;
    AContext->Connection->IOHandler->ReadStream(MS,-1,true);

    MS->Position=0;
    if (MS->Size>0)
        __try//декодируем снимок рабочего стола
        {
            //подключите #include <Vcl.Imaging.jpeg.hpp>
            TJPEGImage *JPG=new TJPEGImage();
            JPG->LoadFromStream(MS);
            JPG->DIBNeeded();
            Image1->Picture->Bitmap->Assign(JPG);
        }
    __finally
    {
        delete JPG;
        delete MS;
    }
}

```

На рис. 5.3 предложен экранный снимок незначительно усложненной версии приложения-сервера, которое позволяет отображать изображения рабочих столов клиентских приложений в отдельных окнах.

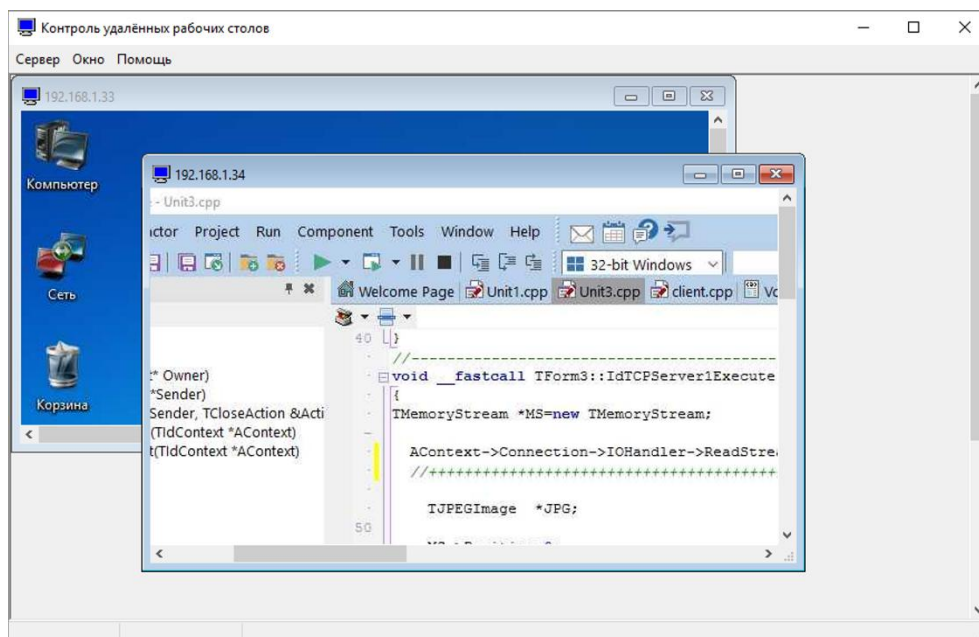


Рис. 6.3. Сбор сервером снимков рабочих столов от клиентских компьютеров

## 7. Сервисы и апплеты

*Вид занятия – практическая работа*

*Цель – разработка службы для Windows*

*Продолжительность – 2 часа*

В современных версиях Windows наряду с обычными процессами трудятся незаметные обычному пользователю помощники — *сервисы* (service application) или, как их еще часто называют, *службы*. Сервис — это низкоуровневый системный процесс, выполняющий функционал по поддержке других, протекающих в операционной системе процессов. Как правило, службы не предоставляют пользователю никакого пользовательского интерфейса. Они запускаются автоматически в момент старта компьютера или вручную из консоли управления службами. Очень важен тот факт, что старт сервиса не зависит от входа пользователя в систему, поэтому службы просто незаменимы при построении различного рода работающих автономно серверов.

Для того чтобы оператор компьютера смог добраться до служб и получить возможность ими управлять, он обязан зарегистрироваться в системе с правами администратора. После этого в Панели управления Windows следует найти значок **Администрирование** и во вновь открывшейся папке обратиться к апплету **Службы**.

### **Внимание!**

Не имея достаточных знаний о роли той или иной службы в жизнедеятельности локальной или удаленной станции, ни в коем случае останавливайте ее работу! Это может негативно отразиться на работе других служб или операционной системы в целом.

## **Менеджер управления сервисами**

В Windows работа сервисов складывается из взаимодействия трех элементов: собственно сервиса, *менеджера управления сервисами* (Service Control Manager, SCM) и *управляющего приложения* (Service Control Program, SCP).

При загрузке операционной системы Windows менеджер SCM стартует одним из первых. Сразу после собственной инициализации SCM берет в свои руки бразды контроля над службами — управляет процессом загрузки всех автоматически подключаемых служб. Во время работы операционной системы менеджер передает команды от управляющих программ (например, консоли управления сервисами) к имеющимся в его подчинении службам. К перечню основных команд относятся:

- ☐ команды инсталляции и деинсталляции служб;
- ☐ команды запуска, приостановки, возобновления выполнения или полной остановки службы;
- ☐ команды анализа состояния и изменения конфигурационных характеристик службы.

Менеджер SCM отвечает за конфигурационную базу данных служб системы, а всю информацию о зарегистрированных в операционной системе службах менеджер прячет в системном реестре Windows.

Для организации работы со службами компьютера следует получить идентификатор менеджера служб. С этой задачей превосходно справляется функция:

```
SC_HANDLE WINAPI OpenSCManager( __in LPCTSTR lpMachineName,
                                __in LPCTSTR lpDatabaseName,
                                __in DWORD dwDesiredAccess);
```

Здесь `lpMachineName` — указатель на сетевое имя интересующего нас компьютера. Сетевое имя станции должно начинаться с двух наклонных черт, например, `"\\NetComputer"`. Если идет речь не о сетевом компьютере, а о нашей собственной станции, то в этот параметр передаем неопределенный указатель `NULL`. Второй параметр `lpDatabaseName` представляет

собой указатель на строку с именем базы данных менеджера управления службами. На данный момент для инициализации этого параметра предусмотрена одна-единственная константа `SERVICES_ACTIVE_DATABASE`, и поэтому функция не обидится, если в ее второй параметр также будет направлен `NULL`. Последний параметр `dwDesiredAccess` определяет наши права доступа. Наиболее грозные возможности предоставит константа `SC_MANAGER_ALL_ACCESS` — это права администратора компьютера. В случае успешного выполнения функция возвращает дескриптор менеджера служб.

После того как необходимость в услугах менеджера служб отпадает, обязательно освободите SCM. Для этого понадобится функция:

```
BOOL WINAPI CloseServiceHandle( __in SC_HANDLE hSCObject);
```

Единственный параметр метода — дескриптор менеджера SCM.

Менеджер SCM выступает точкой опоры для всех операций со службами, например, для получения сведений об их состоянии. Предложенные в листинге 7.1 строки кода демонстрируют порядок решения подобных задач. В примере задействован компонент `TListView *ListView1`. Компонент переведен в табличный режим представления данных (`ViewStyle=vsReport`), и в его свойстве `Columns` заранее подготовлены колонки (рис. 7.1) для данных о системных сервисах.

#### Листинг 7.1. Сбор сведений о доступных службах Windows

```
#include "Winsvc.h"
...
//-----
void __fastcall TfrmMain::GetServicesInformation()
{
    SC_HANDLE hSCM=OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS); //дескриптор SCM

    __try {
        LVServices->Items->BeginUpdate();
        LVServices->Items->Clear();

        unsigned long BufSize=sizeof(ENUM_SERVICE_STATUS) + 256;
        _ENUM_SERVICE_STATUSW* Services
            =(_ENUM_SERVICE_STATUSW*) malloc(BufSize);

        do{
            unsigned long BytesNeeded, ResumeHandle, ServicesReturned;

            EnumServicesStatus(hSCM,
                            SERVICE_WIN32 | SERVICE_DRIVER,
                            SERVICE_STATE_ALL,
                            Services,
                            BufSize,
                            &BytesNeeded,
                            &ServicesReturned,
                            &ResumeHandle);

            DWORD Err=GetLastError();

            TListItem* LI=LVServices->Items->Add(); //создаем элемент в дереве
            LI->Caption=Services->lpDisplayName;
            LI->SubItems->Add(Services->lpServiceName);

            switch (Services->ServiceStatus.dwCurrentState) //состояние сервиса
            { case SERVICE_RUNNING      : {LI->SubItems->Add("Работает");break;
```

```

        case SERVICE_PAUSE_PENDING : {
            LI->SubItems->Add("Приостановлен");break;}
        case SERVICE_STOPPED
            : {LI->SubItems->Add("");break;}
    }

    String s;
    switch (Services->ServiceStatus.dwServiceType) //тип сервиса
    { case SERVICE_FILE_SYSTEM_DRIVER : {s="Драйвер файловой системы ";break;}
      case SERVICE_KERNEL_DRIVER      : {s="Драйвер ядра";break;}
      case SERVICE_WIN32_OWN_PROCESS  : {s="Пользовательская служба ";break;}
      case SERVICE_WIN32_SHARE_PROCESS: {s="Системная служба";break;}
    }

    LI->SubItems->Add(s);

    if (Err!=ERROR_MORE_DATA || ResumeHandle==0) {break;}
}
while (true);

free (Services);
}
__finally
{LVServices->SortType=Vcl::ComCtrls::stText;
 LVServices->Items->EndUpdate();
 CloseServiceHandle(hSCM);
}
}
}

```

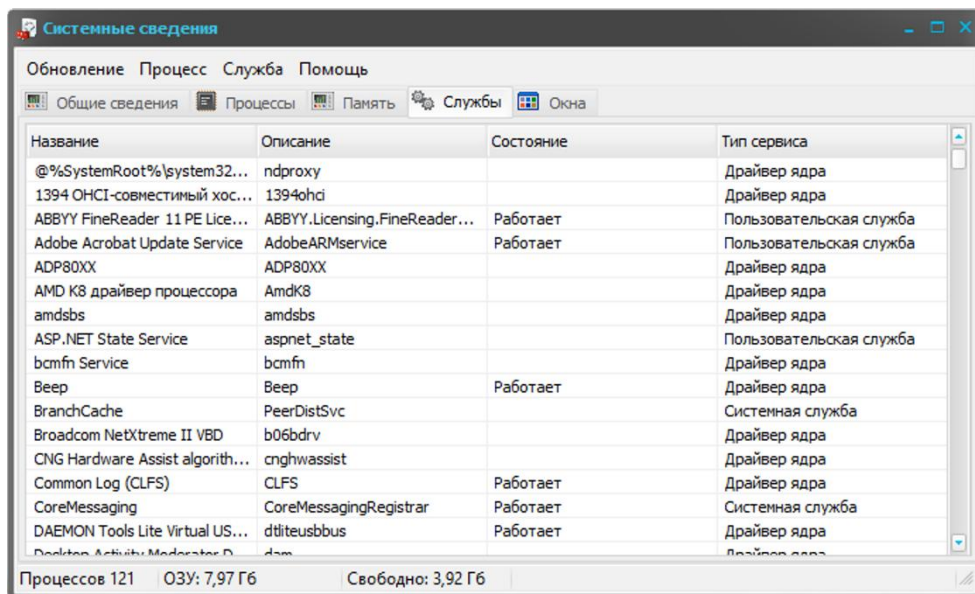


Рис. 7.1. Сбор сведений о системных службах

В листинге 7.1 для сбора информации о службах была задействована функция

```

BOOL WINAPI EnumServicesStatus (
    __in     SC_HANDLE hSCManager,
    __in     DWORD dwServiceType,
    __in     DWORD dwServiceState,
    __out    LPENUM_SERVICE_STATUS lpServices,
    __in     DWORD cbBufSize,
    __out    LPDWORD pcbBytesNeeded,
    __out    LPDWORD lpServicesReturned,
    __in_out LPDWORD lpResumeHandle);

```

Первый параметр функции `hSCManager` должен содержать ссылку на дескриптор менеджера SCM. Аргумент `dwServiceType` конкретизирует, что именно подлежит сбору, здесь совместно могут применяться два флага — `SERVICE_DRIVER` (драйверы) и `SERVICE_WIN32` (службы). Третий параметр `dwServiceState` еще более сужает область поиска: флаг `SERVICE_ACTIVE` указывает, что нас интересуют только активные службы; `SERVICE_INACTIVE` — остановленные службы; `SERVICE_STATE_ALL` — все службы. Оставшиеся параметры предназначены для возврата результатов выполнения функции, с ними вы сможете познакомиться в справочной системе.

## Управление сервисом

Для доступа к установленному в операционной системе сервису необходимо получить его дескриптор. Для этого предназначена функция

```
SC_HANDLE WINAPI OpenService(
    __in SC_HANDLE hSCManager,
    __in LPCTSTR lpServiceName,
    __in DWORD dwDesiredAccess);
```

Здесь: `hSCManager` — полученный усилиями функции `OpenSCManager()` дескриптор менеджера служб; `lpServiceName` — указатель на текстовую строку с именем сервиса; `dwDesiredAccess` — желательные права доступа к сервису, наибольшие возможности предоставляет права администратора `SC_MANAGER_ALL_ACCESS`. При удачном стечении обстоятельств функция вернет дескриптор затребованной нами службы.

Для запуска службы из приложения SCP используют функцию

```
BOOL WINAPI StartService(
    __in SC_HANDLE hService,
    __in DWORD dwNumServiceArgs,
    __in LPCTSTR* lpServiceArgVectors);
```

Первый параметр функции определяет указатель интересующего нас сервиса, второй параметр `dwNumServiceArgs` — количество строк в массиве аргументов, на который указывает третий по счету параметр `lpServiceArgVectors`. В простейшем случае (листинг 7.2) для запуска сервиса нам понадобится только его дескриптор.

### Листинг 7.2. Запуск службы

```
String SrvName=<имя службы>;

SC_HANDLE hSCM=OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS); //дескриптор SCM
SC_HANDLE hSrv=OpenService(hSCM,SrvName.w_str(),SC_MANAGER_ALL_ACCESS);

if (hSrv>0) StartService(hSrv, 0, NULL);

CloseServiceHandle(hSrv);
CloseServiceHandle(hSCM);
```

Обладая дескриптором сервиса, приложение вправе выдать команду на его приостановку, возобновление и завершение работы. Для этого применяется функция

```
BOOL WINAPI ControlService(
    __in SC_HANDLE hService,
    __in DWORD dwControl,
    __out LPSERVICE_STATUS lpServiceStatus);
```

Первый параметр метода — дескриптор сервиса. Параметр `dwControl` определяет наши требования к поведению службы: значение `SERVICE_CONTROL_STOP` остановит выполнение службы; `SERVICE_CONTROL_PAUSE` — временная остановка; `SERVICE_CONTROL_CONTINUE` запустит сервис на выполнение. Последний выходной параметр `lpServiceStatus` ссылается на объявленную в

модуле WinSvc структуру `_SERVICE_STATUS`, хранящую основные характеристики службы. В случае корректного завершения функция возвратит значение `true`.

Для определения текущего состояния службы стоит воспользоваться услугами функции

```
BOOL WINAPI QueryServiceStatus(
    __in SC_HANDLE hService,
    __out LPSERVICE_STATUS lpServiceStatus);
```

На вход функции подается дескриптор исследуемой службы, а вся информация о ней возвращается в структуру `_SERVICE_STATUS` (листинг 7.3).

#### Листинг 7.3. Состав структуры `_SERVICE_STATUS`

```
typedef struct _SERVICE_STATUS {
    DWORD dwServiceType;           //тип исполняемого модуля сервиса
    DWORD dwCurrentState;          //текущее состояние сервиса
    DWORD dwControlsAccepted;      //управляющие коды сервиса
    DWORD dwWin32ExitCode;         //код ошибки старта/останова сервиса
    DWORD dwServiceSpecificExitCode; //спецификатор ошибки
    DWORD dwCheckPoint;           //значение контрольной точки, необходимое для
                                //того, чтобы служба могла уведомлять о своей работе
    DWORD dwWaitHint;             //время, необходимое для изменения состояния
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

## Состояние службы

Для получения сведений о конфигурации службы программист может воспользоваться функцией

```
BOOL WINAPI QueryServiceConfig(
    __in SC_HANDLE hService,
    __out LPQUERY_SERVICE_CONFIG lpServiceConfig,
    __in DWORD cbBufSize,
    __out LPDWORD pcbBytesNeeded);
```

В функцию передается дескриптор сервиса `Service`, указатель на приемный буфер (структуру `ServiceConfig`), в который будут помещены результаты работы функции. Все поля информационной структуры `_QUERY_SERVICE_CONFIG` прокомментированы в листинге 7.4. Размер буфера определяется параметром `BufSize`. Последний выходной параметр возвращает пожелания функции по поводу размера буфера.

#### Листинг 7.4. Состав структуры `_QUERY_SERVICE_CONFIG`

```
typedef struct _QUERY_SERVICE_CONFIG {
    DWORD dwServiceType;           //тип сервиса
    DWORD dwStartType;             //тип запуска
    DWORD dwErrorControl;          //реакция на ошибки при загрузке системы
    LPTSTR lpBinaryPathName;       //путь к исполняемому файлу
    LPTSTR lpLoadOrderGroup;       //группа загрузки
    DWORD dwTagId;                 //номер запуска в группе
    LPTSTR lpDependencies;         //зависимые службы
    LPTSTR lpServiceStartName;     //учетная запись
    LPTSTR lpDisplayName;         //отображаемое на экране название
} QUERY_SERVICE_CONFIG, *LPQUERY_SERVICE_CONFIG;
```

## Конфигурирование службы

Для изменения основных параметров службы предназначена функция:

```
BOOL WINAPI ChangeServiceConfig(
    __in SC_HANDLE hService,       //дескриптор сервиса
    __in DWORD dwServiceType,     //тип сервиса
```

```

__in        DWORD dwStartType,           //особенности запуска
__in        DWORD dwErrorControl,       //реакция на ошибки
__in        LPCTSTR lpBinaryPathName,   //путь к исполняемому файлу
__in        LPCTSTR lpLoadOrderGroup,   //группа загрузки
__out       LPDWORD lpdwTagId,          //номер в группе
__in        LPCTSTR lpDependencies,     //зависимые службы
__in        LPCTSTR lpServiceStartName, //учетная запись
__in        LPCTSTR lpPassword,         //пароль к учетной записи
__in        LPCTSTR lpDisplayName      //выводимое на экран название
);

```

Отметим, что если планируется изменить не все, а лишь некоторые из конфигурационных параметров функции в неизменяемые параметры надо передавать константу `SERVICE_NO_CHANGE`.

## Удаление службы

Вся ирония программирования в том, что создаваемый долгими бессонными ночами сервис-шедевр удаляется из системы одной строкой кода:

```

BOOL WINAPI DeleteService( __in SC_HANDLE hService);

```

Единственный параметр функции — дескриптор неактуальной службы.

## Сервис в VCL, класс *TService*

В библиотеке VCL сервис представлен в виде класса `TService` и описывается в программном модуле `SvcMgr`. При выборе предка `TService` создатели VCL воспользовались услугами модуля данных `TDataModule`, специализирующегося на хранении невизуальных компонентов. Благодаря такому решению значительно упростился процесс программирования, в буквальном смысле службу можно "потрогать" руками и даже отдать в ее распоряжение ряд компонентов.

## Идентификация

Ключевое свойство, по которому будет идентифицироваться служба в системе, — ее имя:

```

__property TComponentName Name;

```

Имя сервиса нам пригодится при создании сервиса средствами Windows API (функция `CreateService()`) или при выяснении дескриптора (функция `OpenService()`).

Кроме имени, у службы может быть заголовок:

```

__property UnicodeString DisplayName;

```

Заголовок будет отображаться в колонке "Имя" консоли управления службами.

## Тип сервиса

Тип сервиса определяется свойством

```

__property TServiceType ServiceType; //по умолчанию stWin32

```

Различают три типа службы: `stWin32` — пользовательский или системный сервис; `stDevice` — драйвер устройства; `stFileSystem` — драйвер файловой системы.

Для совместимости со старыми разработками сохранено, на сегодня уже не актуальное свойство

```

__property bool Interactive; //по умолчанию false

```

Это свойство позволяло пользователю взаимодействовать со службой с помощью GUI. Поддержка интерактивности имела смысл в ситуации, когда мы наделяли сервис пользовательским интерфейсом, а это в современных версиях Windows проблематично.

**Внимание!**

Интерактивность служб поддерживалась только до Windows XP включительно, в более поздних ОС из соображений безопасности Microsoft запретила интерактивные сервисы. Если вашему сервису обязательно нужен доступ к GUI, то наиболее простой выход из положения будет связан с разработкой отдельного процесса-помощника, который будет запускаться вместе с сервисом и взаимодействовать с ним при посредничестве любой известной вам технологии (анонимных каналов, объектов файлового отображения и т. п.).

**Определение прав на управление сервисом**

Запуск сервиса всегда осуществляется от имени какой-либо зарегистрированной в системе учетной записи. Если все оставить на самотек, то служба стартует от имени системной учетной записи LocalSystem. Если же есть специальные требования к политике безопасности, то следует поэкспериментировать со свойством

```
__property UnicodeString ServiceStartName; //по умолчанию пусто
```

определяющим имя учетной записи. При запуске сервиса от имени локального пользователя формат имени должен выглядеть так: ".¥<ИМЯ>"; для доменного — "<ДОМЕН>¥<ИМЯ>".

Задавая учетную запись для сервиса типа stWin32, требуется указать пароль

```
__property UnicodeString Password;
```

Но будьте осторожны, потому что пароль сохраняется в реестре, а его корректность проверяется только в момент запуска сервиса. Специальные учетные записи (LocalSystem, LocalService и NetworkService) пароля не имеют.

**Загрузка и запуск службы**

Корректность функционирования службы состоит в прямой зависимости от того, запущены или нет обеспечивающие ее сервисы и загружены ли необходимые драйверы. Если основная служба не загружена или остановлена, то это, безусловно, отразится и на выполнении зависимого от нее сервиса. Менеджер служб всегда отслеживает такие зависимости.

При определении сервиса необходимо указать список служб, от которых зависит сервис. Для этого предназначено свойство-контейнер

```
__property TDependencies* Dependencies;
```

Доступ к элементу коллекции (экземпляру класса TDependency) осуществляется с помощью свойства

```
__property TDependency* Items[int Index];
```

При наличии явной зависимости между разрабатываемой нами службой и другими сервисами особую роль играет строгая очередность загрузки взаимосвязанных сервисов. Для определения последовательности загрузки служб и системных драйверов в Windows введено понятие групп загрузки. При разработке службы программист указывает, в составе какой из групп следует загружать его сервис. Название группы определяется в свойстве:

```
__property UnicodeString LoadGroup;
```

Список групп загрузки мы найдем в системном реестре по адресу

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\%ServiceGroupName
```

Перечень включает несколько десятков наименований. При необходимости программист имеет право определить в реестре собственную группу загрузки.

Службы из каждой группы стартуют только после того, как запустятся все автоматически стартующие службы из предыдущей группы. Сервисы, которые не принадлежат ни к одной из групп загрузки, запускаются в последнюю очередь.

Для того чтобы упорядочить очередность загрузки сервиса внутри группы загрузки, его порядковый номер передается в свойство:

`__property unsigned TagID; //по умолчанию 0`

Это свойство используется только при загрузке служб ядра. Если загрузочный номер неизвестен, то оставляйте в свойстве нулевое значение.

Особенности запуска службы определяются состоянием свойства:

```
__property TStartType StartType; //по умолчанию stAuto
enum TStartType : unsigned char { stBoot, //запуск во время загрузки ОС
    stSystem, //запуск после загрузки ОС
    stAuto, //автоматический запуск
    stManual, //служба стартует вручную
    stDisabled }; //служба отключена
```

Особенности страта сервиса могут быть определены в дополнительных параметрах запуска. Для обращения к отдельному параметру запуска направьте его индекс в массив:

`__property System::UnicodeString Param[int Index]; //только для чтения`

### Статус службы

О текущем состоянии службы нас проинформирует свойство

```
__property TCurrentStatus Status;
enum TCurrentStatus : unsigned char { csStopped, //сервис остановлен
    csStartPending, //сервис стартует
    csStopPending, //выполнение прекращается
    csRunning, //сервис выполняется
    csContinuePending, //ожидание продолжения
    csPausePending, //ожидание приостановки
    csPaused }; //сервис приостановлен
```

Свойство Status не только информационное, с его помощью программист имеет право управлять статусом службы из кода проекта. Любое изменение состояния службы согласуется с менеджером служб SCM. Для этого автоматически вызывается метод

`void ReportStatus(void);`

В теле этой функции заполняется структура TServiceStatus, содержащая несколько информационных полей, таких как тип, текущее состояние службы, код ошибки и т. п. Затем вся эта информация и предложение по смене состояния службы направляются менеджеру при помощи функции SetServiceStatus(). Если с нашими предложениями менеджер согласится, то служба переводится в запрашиваемое состояние.

### Сбои при старте сервиса

Реакцию операционной системы на сбой в момент старта службы определяет свойство

`__property TErrorSeverity ErrorSeverity; //по умолчанию esNormal`

Четыре возможных варианта поведения Windows представлены в табл. 7.1.

Таблица 7.1. Значения TErrorSeverity

| Значение   | Поведение                                                                                                                                            |
|------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| esIgnore   | Игнорировать ошибку. Данные об ошибке заносятся в журнал, служба продолжает выполняться                                                              |
| esNormal   | Ошибка заносится в журнал. На экран выводится сообщение об ошибке, после чего служба продолжит свою работу                                           |
| esSevere   | Ошибка заносится в журнал. Выполнение сервиса продолжится только в случае, если система сможет загрузиться в режиме "последняя удачная конфигурация" |
| esCritical | Критическая ошибка. Если происходит отказ этой службы, ошибка заносится в журнал и                                                                   |

|                                                                   |
|-------------------------------------------------------------------|
| система перезагружается в режиме "последняя удачная конфигурация" |
|-------------------------------------------------------------------|

## Остановка и возобновление службы

Не каждая служба может быть остановлена или приостановлена, например, такой возможностью не обладают системные драйверы. В роли индикатора того, что служба способна "взять паузу", выступает свойство

```
__property bool AllowPause;
```

Перед приостановкой службы вызывается событие `OnPause()`, а после возобновления — `OnContinue()`.

### Замечание

Напомню, что проще всего приостановить или возобновить выполнение сервиса с помощью функции `ControlService()`.

Еще одно свойство, с помощью которого проверяется, сможем ли мы остановить сервис:

```
__property bool AllowStop;
```

Если это свойство установлено в `true`, то перед остановкой службы вызывается событие `OnStop()`. Весь перечень событий, связанных с запуском и остановкой службы, представлен в табл. 7.2.

Таблица 7.2. События запуска и остановки сервиса

| Событие                                                                                                                              | Описание                                                                                                                                                                                         |
|--------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__property TStartEvent OnStart; typedef void (__closure *TStartEvent) (TService* Sender, bool &amp;Started);</code>            | Событием <code>OnStart()</code> сопровождается первый старт службы. В этот момент осуществляется инициализация сервиса                                                                           |
| <code>__property TPauseEvent OnPause; typedef void (__closure *TPauseEvent) (TService* Sender, bool &amp;Paused);</code>             | Генерируется в момент приостановки службы. Управляя параметром <code>Paused</code> , программист сможет запретить ( <code>false</code> ) или разрешить ( <code>true</code> ) приостановку службы |
| <code>__property TContinueEvent OnContinue; typedef void (__closure *TContinueEvent) (TService* Sender, bool &amp;Continued);</code> | Возобновление работы сервиса. Если мы не имеем ничего против возобновления выполнения службы, то передаем параметру <code>Continued</code> значение <code>true</code>                            |
| <code>__property TStopEvent OnStop; typedef void (__closure *TStopEvent) (TService* Sender, bool &amp;Stopped);</code>               | Событие вызывается в тот момент, когда менеджер служб завершает работу сервиса                                                                                                                   |
| <code>__property TServiceEvent OnShutdown;</code>                                                                                    | Событие генерируется в момент завершения работы операционной системы. Здесь мы можем описать действия, предшествующие принудительному разрушению сервиса                                         |

## Инсталляция и деинсталляция сервиса

В роли инициатора регистрации службы может выступить внешнее приложение или владеющий этим сервисом процесс, отметим, что это особый вид приложения, не использующий привычный для нас класс `TApplication`. Фундамент приложения-службы составляет специализированный класс `TServiceApplication`. С этим классом и процессом разработки приложений такого рода мы познакомимся несколько позднее, а пока (табл. 7.3) рассмотрим, каким образом класса `TService` реагирует на установку (и удаление) службы.

Таблица 7.3. События установки и удаления сервиса

| Событие                                                                                                          | Описание                                                                                               |
|------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <code>__property TServiceEvent BeforeInstall; typedef void (__closure *TServiceEvent) (TService* Sender);</code> | Начало инсталляции службы. Единственный параметр события <code>Sender</code> содержит ссылку на сервис |

|                                                        |                                          |
|--------------------------------------------------------|------------------------------------------|
| <code>__property TServiceEvent AfterInstall;</code>    | Завершение инсталляции службы            |
| <code>__property TServiceEvent BeforeUninstall;</code> | Начало деинсталляции службы              |
| <code>__property TServiceEvent AfterUninstall;</code>  | Деинсталляция службы успешно завершилась |

#### **Замечание**

Весь перечень установленных на компьютере служб регистрируется в системном реестре Windows. Информацию об имеющихся службах мы обнаружим в ветви  
 HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services

## **Выполнение службы, поток *TServiceThread***

В Windows каждая служба выполняется в отдельном потоке. Кроме того, для организации взаимодействия службы *TService* с менеджером служб для каждого экземпляра службы VCL создает новый поток (экземпляр класса *TServiceThread*). Ссылка на экземпляр потока хранится в свойстве службы

```
__property TServiceThread* ServiceThread;
```

Поток *TServiceThread* является прямым наследником классического потока *TThread*, его основное отличие от предка в наличии метода

```
void ProcessRequests (bool WaitForMessage);
```

Благодаря *ProcessRequests()* сервис взаимодействует с менеджером служб Windows. В рамках метода, взаимодействующего со службой, поток отслеживает пять управляющих сообщений, поступающих от менеджера SCM :

- ☐ *SERVICE\_CONTROL\_STOP* — остановить сервис;
- ☐ *SERVICE\_CONTROL\_PAUSE* — приостановить сервис;
- ☐ *SERVICE\_CONTROL\_CONTINUE* — продолжить сервис;
- ☐ *SERVICE\_CONTROL\_SHUTDOWN* — уничтожение сервиса;
- ☐ *SERVICE\_CONTROL\_INTERROGATE* — запрос о состоянии сервиса.

Таким образом, с помощью *ProcessRequests()* служба поддерживает постоянный контакт с менеджером служб Windows. Единственный параметр *WaitForMessage* функции переводит ее в синхронный (*true*) или асинхронный (*false*) режим работы. При выполнении функции в синхронном режиме поток станет ждать управляющей команды от SCM и соответственно приостановит выполнение службы до поступления этой команды. В асинхронном режиме обращение к *ProcessRequests()* не останавливает службу. А теперь посмотрим, каким образом служба *TService* сможет воспользоваться услугами этого метода, а для этого мы познакомимся с центральным событием, связанным с выполнением сервисом своих прямых обязанностей

```
__property TServiceEvent OnExecute;
```

Это именно то событие, в рамках которого описывается основной исполняемый код службы. Если в функциональные обязанности службы входит постоянное обслуживание какого-то устройства, непрерывный сбор информации или что-то подобное, то внутри *OnExecute()* реализуют цикл, выход из которого производится по команде менеджера SCM.

Цикл выполняется до тех пор, пока свойство

```
__property bool Terminated;
```

не перейдет в состояние *true* или его не остановит менеджер служб Windows. Свойство *Terminated* уведомляет программиста о том, что выполнение службы завершается, и она должна быть уничтожена.

## **Ведение протокола службы**

Любая профессионально разработанная служба обязана уметь общаться с журналом событий операционной системы. Благодаря записям журнала администратор компьютера

получает возможность контролировать корректность работы системы и при необходимости вмешиваться в деятельность того или иного процесса или службы. Каждая строка журнала включает информацию об имени службы, типе события, времени его возникновения и т. п.

Код ошибки, возвращаемый в момент некорректной остановки или при неудачном запуске сервиса, содержится в свойстве:

```
__property unsigned ErrCode;
```

Если свойство ErrCode не определено, то код ошибки будет взят из свойства

```
__property unsigned Win32ErrCode;
```

И, наконец, самый последний вариант развития событий, когда код ошибки не задан ни в одном из перечисленных свойств, в свойство Win32ErrCode передается константа ERROR\_SERVICE\_SPECIFIC\_ERROR.

О факте старта, остановки, приостановки, возобновления сервиса менеджер служб информируется автоматически без вмешательства программиста. Продолжительность отрезка времени между моментами запуска (приостановки, остановки) службы и отправки сообщения в адрес менеджера служб компьютера определяется свойством:

```
__property int WaitHint; //по умолчанию 5000 миллисекунд
```

Как правило, для взаимодействия с функциями Windows API установленного по умолчанию значения в 5 секунд вполне достаточно, но если служба выполняет длительную операцию, то, вызвав ReportStatus(), целесообразно принудительно проинформировать менеджера о состоянии службы.

Для добавления в журнал событий сведений о жизнедеятельности сервиса применяют функцию

```
void LogMessage(UnicodeString Message, unsigned EventType = (unsigned) (0x1), int Category = 0x0, int ID = 0x0);
```

Здесь: Message — текстовое содержание сообщения; EventType — тип события (табл. 7.4); категория события и его идентификационный номер определяются значениями, переданными в параметры Category и ID.

Таблица 7.4. Тип события EventType

| Константа                 | Описание типа события                    |
|---------------------------|------------------------------------------|
| EVENTLOG_ERROR_TYPE       | Произошла ошибка                         |
| EVENTLOG_WARNING_TYPE     | Предупреждение о потенциальной опасности |
| EVENTLOG_INFORMATION_TYPE | Обычное информационное уведомление       |
| EVENTLOG_AUDIT_SUCCESS    | Аудит завершился успешно                 |
| EVENTLOG_AUDIT_FAILURE    | Ошибка аудита                            |

## Приложение-сервис TServiceApplication

В библиотеке компонентов существует класс TServiceApplication существенно упрощающий проектирование службы Windows. Для создания нового сервисного приложения необходимо воспользоваться пунктом меню **File | New | Other** и на странице **New** диалогового окна **New Items** выбрать пункт **Service Application**.

В результате создается шаблон кода, в котором объявляется глобальный объект Application типа TServiceApplication. Кроме того, в проекте декларируется инкапсулирующий службу объект TService1 \*Service1. При необходимости мы можем добавить в приложение дополнительные сервисы. Для этого надо вновь воспользоваться пунктом главного меню **File | New | Other** и в окне **New Items** выбрать значок **Service**.

Приложение `TServiceApplication` лишь играет роль оболочки для входящих в его состав сервисов и поэтому практически не обладает индивидуальными свойствами и методами. Из имеющихся в нашем распоряжении свойств интерес представляет

```
__property int ServiceCount; //только для чтения
```

информирующее программиста о количестве сервисов (экземпляров класса `TService`).

Метод

```
void RegisterServices(bool Install, bool Silent);;
```

позволяет зарегистрировать (`Install=true`) в системе или снять с регистрации (`Install=false`) все принадлежащие приложению сервисы. Установив параметр `Silent` в состояние `true`, мы добьемся "тихой" работы метода — на экран компьютера не будет выведено сообщение о завершении операции регистрации.

## Пример

Если вам когда-нибудь приходилось администрировать файловые серверы, предназначенные для сбора документов со всего предприятия, то наверняка вы согласитесь, что одной из самых рутинных задач является сортировка поступающих из сети файлов. Теперь, когда мы получили представление о сервисах, мы сможем существенно упростить решение подобных задач.

Допустим, что у нас имеется предоставленная в общий доступ сетевая папка, в которую поступают различные документы со всех подразделений предприятия. Наша задача заключается в разработке сервиса `Windows`, отслеживающего все попадающие в эту папку файлы и распределяющего их по другим папкам по какому-то классификационному признаку, например, по типу файла.

Создадим новое приложение-сервис (`Service Application`) и приступим к его конфигурированию. Для этого выберем модуль службы с именем по умолчанию `Service1` и внесем ряд изменений в его свойства:

- ☐ тип сервиса `ServiceType=stWin32`;
- ☐ имя сервиса `Name=FolderControl`;
- ☐ отображаемое имя `DisplayName= "Контроль сетевой папки"`;
- ☐ тип запуска `StartType=stManual`;
- ☐ реакция на ошибки при запуске `ErrorSeverity=esIgnory`.

Сохраните проект в отдельном каталоге — модуль службы назовите `FolderControl_unit.cpp`, а весь проект — `fldsrv.cbproj`.

Объявите ряд переменных, необходимых для работы сервиса (листинг 7.5).

### Листинг 7.5. Глобальные переменные службы

```
String fldrINPUT, fldrDOC, fldrPICTURE, fldrOTHER, PICTURE_EXT, DOC_EXT;
```

Переменные `fldr...` хранят пути к папкам, контролируемым сервисом, в первую очередь это общедоступная системная папка (переменная `fldrINPUT`), в которую будут поступать файлы из сети. В переменные `PICTURE_EXT` и `DOC_EXT` мы позднее занесем перечень наиболее востребованных расширений имен файлов.

В момент старта сервиса обращаемся к системному реестру и читаем из него параметры сервиса (имена каталогов и перечень расширений имен файлов). Затем обязательно проконтролируем факт существования папок и, если они по какой-то причине отсутствуют, создадим их (листинг 7.6).

### Листинг 7.6. Событие старта сервиса

```

void __fastcall TFolderControl::ServiceStart(TService *Sender,
                                             bool &Started)
{
    const String REG_KEY="\\SYSTEM\\CurrentControlSet\\services\\FolderControl",
              REG_SECTION="PARAMETERS";

    TRegIniFile *R=new TRegIniFile("");
    R->RootKey=HKEY_LOCAL_MACHINE;
    R->OpenKey(REG_KEY,true);
    fldrINPUT  =R->ReadString(REG_SECTION,"INPUT",  "c:\\inputfolder\\");
    fldrDOC    =R->ReadString(REG_SECTION,"DOC",    "c:\\doc\\");
    fldrPICTURE=R->ReadString(REG_SECTION,"PICTURE","c:\\picture\\");
    fldrOTHER  =R->ReadString(REG_SECTION,"OTHER",  "c:\\other\\");

    PICTURE_EXT=R->ReadString(REG_SECTION,"PICTURE_EXT",
                              ".bmp .gif .tif .tiff .jpg .jpeg .png .wmf .emf");
    DOC_EXT     =R->ReadString(REG_SECTION,"DOC_EXT",
                              ".doc .rtf .docx .docm .txt");

    delete R;

    try //если папки отсутствуют - создаем их
    {
        if (!DirectoryExists(fldrINPUT))    CreateDir(fldrINPUT);
        if (!DirectoryExists(fldrDOC))      CreateDir(fldrDOC);
        if (!DirectoryExists(fldrPICTURE))  CreateDir(fldrPICTURE);
        if (!DirectoryExists(fldrOTHER))    CreateDir(fldrOTHER);
        Started=true;
    }
    catch (...)
    {
        Started=false;
    }
}

```

Отметим, что запрашиваемая в листинге 7.6 секция системного реестра пока отсутствует, поэтому в глобальные переменные поступают значения по умолчанию. Конечно, администратор сервера сможет заполнить реестр вручную, это гораздо лучше, чем каждый раз (при необходимости изменить имена папок или расширения имен файлов) требовать от программиста переписать сервис.

Нам осталось сделать самое важное — описать событие `OnExecute()` сервиса (листинг 7.7). В рамках этого события мы контролируем состояние папки, описываемой константой `fldrINPUT`. Структурно событие представляет собой два вложенных цикла `do..while`. Задача внешнего цикла — не пропустить момент поступления в папку любого файла, в этом ему поможет входящая в состав Windows API функция `FindFirstChangeNotification()`. По факту поступления файла активируется вложенный цикл, здесь с помощью функций `FindFirst()` и `FindNext()` мы собираем сведения о появившихся в каталоге файлах и распределяем их по другим папкам.

#### Листинг 7.7. Выполнение сервиса

```

void __fastcall TFolderControl::ServiceExecute(TService *Sender)
{
    TSearchRec SR;

    HANDLE h=FindFirstChangeNotificationW(fldrINPUT.w_str(),true, FILE_NOTIFY_CHANGE_FILE_NAME |
    FILE_NOTIFY_CHANGE_DIR_NAME);
    do
    {
        ServiceThread->ProcessRequests(false);
        unsigned c=WaitForSingleObject(h,5000);
    }
    while (h!=INVALID_HANDLE_VALUE);
}

```

```

    if (c=WAIT_OBJECT_0) //в контролируемой папке произошли изменения
//=====
    if (FindFirst(fldrINPUT+"*.*",faAnyFile,SR)==0)
    do
    {
    if (SR.Attr && faDirectory== 0)
        {
        String FileExt=(ExtractFileExt (SR.Name)).LowerCase();
        String oldfile=fldrINPUT+SR.Name;
        String newfile;
        //это графический файл
        if (PICTURE_EXT.Pos (FileExt)!=0)
            newfile=fldrPICTURE+SR.Name;

        else //это документ
        if (DOC_EXT.Pos (FileExt)!=0) newfile=fldrDOC+SR.Name;
            else newfile=fldrOTHER+SR.Name; //другой файл
            //перемещаем поступивший файл
            MoveFileW(oldfile.w_str(),newfile.w_str());
        }
        }
        while (FindNext (SR)==0);
        FindClose (SR);
    }
//=====
    ServiceThread->ProcessRequests (false);
}
while (!Terminated);
}

```

Отметим, что листинг 7.9 представляет собой лишь самое общее решение задачи распределения файлов. Например, в нем не предусмотрены действия при совпадении имен файлов. Тем не менее, для наших первых шагов на поприще разработки служб Windows мы сделали уже много. Будем полагать, что проект завершен. Откомпилируйте его. Теперь нам необходимо зарегистрировать сервис в SCM.

## Регистрация сервиса в ручном режиме

Для регистрации службы в операционной системе владеющее службой приложение должно быть запущено из командной строки с ключом /INSTALL. Например,  
 C:\DemoService\fldsrv.exe / INSTALL

Для снятия с регистрации применяют ключ /UNINSTALL. Процесс установки сервиса сопровождается выводом уведомляющего сообщения. Для отказа от показа окна уведомления используйте ключ /SILENT.

### Замечание

Сведения о зарегистрированных сервисах хранятся в разделе реестра  
 HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\<ИМЯ\_СЛУЖБЫ>.

Откройте консоль управления службами компьютера, найдите в ней наш сервис "Контроль общей папки" и запустите его на выполнение. Поместите в контролируемую папку любые файлы. Если вы безошибочно повторили код, то сразу получите результат — служба распределит эти файлы по соответствующим папкам.

Отладка службы — весьма непростое занятие по двум причинам: во-первых, службой управляет менеджер SCM, который весьма капризен по отношению внешних вмешательств; во-вторых, приложение-служба как минимум обладает двумя потоками. Если же ваше приложение-служба инкапсулирует несколько сервисов, то сложность

отладки возрастает в геометрической прогрессии в соответствии с количеством интегрируемых служб.

#### **Замечание**

Лучшая рекомендация по отладке служб звучит так: реализуйте весь функционал службы в обычном приложении. Выявите и устраните ошибки стандартным образом в отладчике C++ Builder и только затем перенесите апробированный код в приложение-службу.

## **Апплеты Панели управления**

Даже начинающий пользователь Windows знает о существовании Панели управления (Control Panel). Панель управления предоставляет удобный доступ к глобальным настройкам операционной системы, ее системным компонентам и обслуживаемых этой ОС устройствам. Для этого оператору компьютера необходимо найти интересующий его значок и дважды по нему щелкнуть левой кнопкой мыши. В результате инициализируется специализированная динамическая библиотека, называемая *апплетом Панели управления* (Control Panel Applet). Апплет управляет единственной подпрограммой, обеспечивая пользовательский контроль параметров настройки эксплуатационных режимов отдельного системного компонента Windows.

Помимо классического способа старта апплета двойным щелчком по значку в Windows предусмотрены альтернативные методы запуска апплета — из командной строки или посредством вызова функций `WinExec()` и `ShellExec()`. Первая функция предназначена для выполнения простейших апплетов (листинг 7.8).

#### **Листинг 7.8. Старт апплета с помощью функции `WinExec()`**

```
AnsiString s="control.exe /name Microsoft.AddHardware";
WinExec(s.c_str(),SW_NORMAL);
```

Функция `ShellExec()` востребована в случае, когда при старте апплета требуется передать дополнительные параметры (листинг 7.9).

#### **Листинг 7.9. Старт апплета с помощью функции `ShellExec()`**

```
AnsiString s="control.exe /name Microsoft.AddHardware";
ShellExecute(Application->Handle, L"Open", L"control.exe",
    L"desk.cpl,screensaver,@screensaver", 0, SW_NORMAL);;
```

В листинге 7.9 `control.exe` — название приложения контейнера апплетов, а `AddHardware` — название интересующего нас апплета. Перечень основных апплетов Панели управления Windows представлен в табл. 7.5.

*Таблица 7.5. Стандартные апплеты Windows*

| Апплет                    | Описание                                                               |
|---------------------------|------------------------------------------------------------------------|
| control.exe desktop       | Вывод окна <b>Свойства: экран</b>                                      |
| control.exe color         | Вывод окна <b>Свойства: экран</b> с активной страницей <b>Заставка</b> |
| control.exe date/time     | Вывод окна <b>Свойства: дата и время</b>                               |
| control.exe international | Вывод <b>Язык и региональные стандарты</b>                             |
| control.exe mouse         | Вывод окна <b>Свойства: мышь</b>                                       |
| control.exe keyboard      | Вывод окна <b>Свойства: клавиатура</b>                                 |
| control.exe printers      | Показ папки <b>Принтеры и факсы</b>                                    |
| control.exe fonts         | Показ папки <b>Шрифты</b>                                              |
| control.exe folders       | Вывод окна <b>Свойства папки</b>                                       |

|                            |                                           |
|----------------------------|-------------------------------------------|
| control.exe telephony      | Вывод окна <b>Телефон и модем</b>         |
| control.exe admintools     | Вывод окна <b>Администрирование</b>       |
| control.exe schedtasks     | Вывод окна <b>Назначенные задания</b>     |
| control.exe netconnections | Показ папки <b>Сетевые подключения</b>    |
| control.exe userpasswords  | Вывод <b>Учетные записи пользователей</b> |

## Апплет Панели управления, класс *TAppletModule*

В библиотеке VCL функционал апплета Панели управления инкапсулирован в модуле `CtlPanel.hpp` в виде класса `TAppletModule`. В иерархии наследования классов апплет следует сразу за модулем данных `TDataModule`. Таким образом, значительно упрощается интеграция в апплет невизуальных компонентов. Свойств, методов и событий у апплета совсем немного.

Заголовок модуля определяется свойством

```
__property UnicodeString Caption;
```

Связанный с апплетом значок назначается в свойстве

```
__ TIcon* AppletIcon;
```

Строка с текстом комментария заносится в свойство

```
__property UnicodeString Help;
```

Если же апплет инициализируется данными из ресурса, то вместо свойств `Caption`, `AppletIcon` и `Help` надо воспользоваться свойствами:

```
__property int ResidName; //идентификатор (номер) заголовка апплета
```

```
__property int ResidIcon; // идентификатор значка в ресурсе
```

```
__property int ResidInfo; //идентификатор информационной строки
```

Связанные с апплетом дополнительные данные ассоциируются со свойством

```
__property int Data;
```

Эти данные отправляются к управляемому приложению в момент вызова событий

`OnInquire()` или `OnNewInquire()`.

Своих методов (кроме видоизмененных конструктора и деструктора) у класса нет, зато `TAppletModule` может похвастаться пятью событиями (табл. 7.6).

**Таблица 7.6.** Обработчики событий апплета *TAppletModule*

| Событие                                                                                                                                                                     | Описание                                                                                                                                                                                                                                                                                                                            |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>__property TActivateEvent OnActivate; typedef void (__closure *TActivateEvent)(TObject* Sender, int Data);</pre>                                                       | Событие генерируется в момент двойного щелчка по значку апплета. Здесь: <code>Sender</code> — ссылка на активируемый апплет; <code>Data</code> — значение, устанавливаемое в событиях <code>OnInquire()</code> или <code>OnNewInquire()</code>                                                                                      |
| <pre>__property TStartWParamsEvent OnStartWParams; typedef void (__closure *TStartWParamsEvent)(TObject* Sender, System::UnicodeString Params);</pre>                       | Событие возникает совместно с <code>OnActivate()</code> , оно генерируется в момент вызова апплета из <code>RunDLL</code> . Параметр <code>Params</code> содержит данные, направляемые в адрес <code>RunDLL</code>                                                                                                                  |
| <pre>__property TINquireEvent OnInquire; typedef void (__closure *TINquireEvent)(TObject* Sender, int &amp;idIcon, int &amp;idName, int &amp;idInfo, int &amp;lData);</pre> | Событие вызывается в момент запроса связанной с апплетом информации о: заголовке <code>Caption</code> ; ассоциированном значке <code>Icon</code> ; информационной строке <code>Help</code> и данными <code>Data</code> . Перечисленные данные считываются из ресурса. Необходимые идентификаторы передаются в одноименные параметры |

|                                                                                                                                                                                                                    |                                                                                                                                                       |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>__property TNewInquireEvent OnNewInquire; typedef void (__closure *TNewInquireEvent) (TObject* Sender, int &amp;lData, HICON &amp;hIcon, UnicodeString &amp;AppletName, UnicodeString &amp;AppletInfo);</pre> | Назначение и вызов события аналогичны OnInquire(). Отличие в том, что в качестве параметров передаются не идентификаторы ресурса, а реальные значения |
| <pre>__property TStopEvent OnStop; typedef void (__closure *TStopEvent) (TObject* Sender, int Data);</pre>                                                                                                         | Событие генерируется в момент получения апплетом команды на завершение работы                                                                         |

## Приложение Панели управления *TAppletApplication*

Класс *TAppletApplication* инкапсулирует приложение Панели управления Windows. Класс описан в модуле *CtlPanel.hpp* и реализован на основе *TComponent*. Приложение Панели управления может содержать один или большее число апплетов — экземпляров класса *TAppletModule*.

### *Замечание*

Строго говоря, термин "приложение Панели управления" (Control Panel Application) не совсем корректен. На самом деле это библиотека. В этом можно убедиться, открыв головной модуль проекта — в нем вы столкнетесь с функцией *DllEntryPoint()* предоставляющей точку входа в библиотеку апплета.

После компиляции файл приложения Панели управления получает характерное расширение имени *срl* — этого требует директива `{$E срl}`.

Экспортируемая функция *CPlApplet()* предоставляет апплету точку входа в Панель управления компьютером.

```
int __stdcall CPlApplet(unsigned hwndCPl, unsigned uMsg,
int lParam1, int lParam2);
```

Благодаря этой функции апплет получает возможность посылать в адрес Панели управления неполный десяток сообщений. Например, в момент двойного щелчка по значку апплета отправляется сообщение *CPL\_DBLCLK*.

Связь с Панелью управления поддерживается с помощью свойства

```
__property unsigned ControlPanelHandle; //только для чтения
```

Здесь хранится дескриптор Панели управления или дескриптор приложения, вызвавшего функцию *CplApplet()*.

Перечень всех имеющихся в распоряжении приложения апплетов доступен благодаря свойству

```
__property TAppletModule* Modules[int Index];
```

хранящему список модулей. Сколько всего модулей скрывается в апплете, мы обнаружим в свойстве:

```
__property int ModuleCount;
```

Экземпляр класса *TAppletApplication* создается автоматически и будет доступен программисту посредством глобальной переменной *Application*.

Практически все методы приложения предназначены для внутреннего пользования и в обычных приложениях не применяются.

## Пример апплета управления сервисом Windows

В первой части главы мы разработали сервис, позволяющий распределить файлы, поступающие в сетевую папку файлового сервера. Однако у созданного нами сервиса имеются недостатки, самый главный среди них — отсутствие пользовательского

интерфейса управления параметрами сервиса. Предлагаю устранить этот недостаток, а для этого нам потребуется сделать два шага:

- ☐ разработать приложение управления параметрами сервиса;
- ☐ создать апплет панели управления.

## Приложение управления сервисом

Функционал, закладываемый в приложение управления сервисом весьма прост. Приложение должно предоставлять пользователю удобный интерфейс редактирования основных параметров сервиса управления каталогами, т. е. позволять назначать контролируемые сервисом папки и редактировать расширения имена файлов. Все внесенные пользователем изменения должны передаваться в соответствующую ветвь системного реестра, из которого их сможет считать сервис во время запуска.

Больше мы не станем углубляться в подробности работы приложения (его листинги вы найдете среди примеров).

## Апплет Панели управления

Воспользовавшись услугами пункта меню **File | New | Other**, создам новый проект **Control Panel Application**. По умолчанию он сразу содержит модуль апплета

`TAppletModule1AppletModule *AppletModule1AppletModule`. Выберите этот модуль и в Инспекторе объектов внесите изменения в его свойства:

- ☐ имя модуля апплета: `Name=fldrapplet`;
- ☐ заголовок: `Caption="Управление службой сетевой папки"`;
- ☐ на свое усмотрение подберите подходящий по смыслу значок (свойство `AppletIcon`) и подключите его к модулю апплета.

Сохраните проект в отдельной папке под любым именем (например `fldrnet.cbproj`).

Как правило, при обращении к апплету, на экран компьютера выводится окно, в котором пользователь осуществляет какие-то действия по настройке своего компьютера. В нашем случае это окно целесообразно создать на основе уже готовой главной формы приложения управления сервисом. Поэтому воспользуемся пунктом меню **Project | Add to project** и подключим к апплету программный модуль главной формы приложения управления сервисом.

Выбрав пункт меню **Project | View Source**, откройте головной модуль проекта апплета (листинг 7.10) и откажитесь от автоматического создания экземпляра формы `frmAppletControl` (именно так в примере к книге называется главная форма управления сервисом) при старте приложения).

**Листинг 7.10. Настройка головного модуля проекта апплета**

```
#include <Ctlpanel.hpp>

#pragma hdrstop

USEFORM("AppletModule1.cpp", fldrapplet); /* TAppletModule: File Type */
USEFORM("..\\Приложение управления\\Unit1.cpp", frmAppletControl);
//-----
#pragma link "Ctlpanel.obj"
//-----
extern "C" __declspec(dllexport) __stdcall int CPlApplet(
    unsigned hwndCPl, unsigned uMsg, int lParam1, int lParam2)
{
    return (Ctlpanel::CPlApplet(hwndCPl, uMsg, lParam1, lParam2));
}
```

```
#pragma argsused
//-----
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
    using Ctlpanel::Application;
    try {
        if (reason == DLL_PROCESS_ATTACH) {
            Application->Initialize();
            Application->CreateForm(__classid(Tfldrapplet), &fldrapplet);
            /*!!! Возьмите в комментарии следующую строку кода
            /*Application->CreateForm(__classid
                (TfrmAppletControl), &frmAppletControl);*/
            Application->Run();
        }
        catch (Exception &exception)
        {Application->HandleException(&exception);
        }
        return 1;
    }
    //-----
}
```

Создание формы перепоручим модулю апплета. Для этих целей лучше всего подходит событие `OnActivate()` (листинг 7.11).

#### Листинг 7.11. Активация головного модуля апплета

```
void __fastcall Tfldrapplet::AppletModuleActivate(TObject *Sender, int Data)
{
    TfrmAppletControl *frmAppletControl=new TfrmAppletControl(Ctlpanel::Application);
    frmAppletControl->ShowModal();
}
```

Уничтожение формы производим в момент остановки работы апплета (листинг 7.12).

#### Листинг 7.12. Остановка головного модуля апплета

```
void __fastcall Tfldrapplet::AppletModuleStop(TObject *Sender, int Data)
{
    frmAppletControl->Release();
}
```

В результате компиляции проекта создается файл апплета `fldrnet.cpl`, нам лишь осталось зарегистрировать его в Панели управления компьютером. Для этого достаточно скопировать `cpl`-файл в системный каталог `System32`. После этого откройте Панель управления компьютером, и вы получите доступ к разработанному апплету (рис. 7.3).

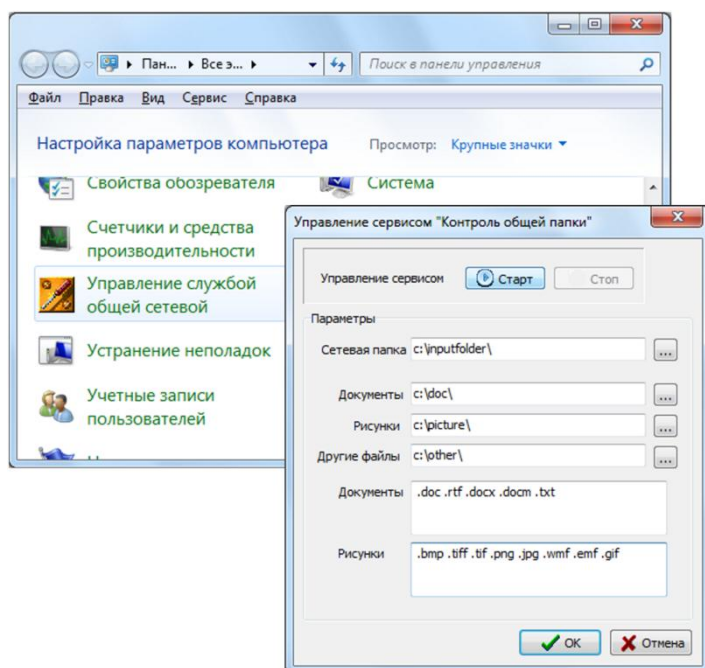


Рис. 7.2. Апплет управления сервисом

## 8. Многокомпонентная модель COM

*Вид занятия – практическая работа*

*Цель – разработка клиента и сервера COM*

*Продолжительность – 2 часа*

Разработанная в корпорации Microsoft *многокомпонентная модель объектов* (Component Object Model, COM) представляет собой спецификацию, описывающую способ взаимодействия программ. Программа-сервер (сервер COM) предоставляет в наше распоряжение определенные услуги, программа-клиент (клиент COM) пользуется этими услугами. Взаимодействующие COM-объекты могут функционировать как в рамках одного и того же процесса, так и в разных процессах, в том числе протекающих на разных машинах.

Компоненты COM состоят из программного кода, реализованного в виде исполняемого ехе-файла или динамической библиотеки DLL. С точки зрения стороннего наблюдателя создаваемый средствами COM программный компонент представляет собой черный ящик, обладающий следующими характеристиками:

- ☐ легкостью подключения к операционной системе Windows;
- ☐ автономностью;
- ☐ простотой интеграции в состав программного обеспечения;
- ☐ универсальным способом обращения к методам и свойствам из любого современного языка программирования. Благодаря строгой спецификации интерфейсов достигается единообразие работы клиента с любым COM-компонентом;
- ☐ прозрачностью, сервер COM расположенный на другой сетевой станции может рассматриваться клиентом так же, как будто он находится на вашем компьютере.

В интересах клиентских и серверных приложений COM в Windows активно трудится *библиотека COM* (COM Library), она способна:

- ☐ предоставлять в распоряжение программиста некоторый набор функций, облегчающий разработку приложений COM;
- ☐ генерировать уникальный идентификатор объекта, осуществлять поиск требуемого объекта по его идентификатору;
- ☐ производить вызов удаленных процедур тогда, когда сервер COM размещен на другом компьютере;
- ☐ управлять распределением памяти в интересах взаимодействующих процессов и контролировать освобождение памяти.

### COM-объект

В основу многокомпонентной модели положено понятие COM-объекта. Физически COM-объект, как и любой другой объект, соответствующий парадигме ООП, представляет собой совокупность данных и управляющих этими данными методов. Вместе с тем структура COM-объекта существенно отличается от построения привычных для нас объектов VCL. Ключевое отличие объекта COM от объекта Delphi в способе предоставления к полям и методам объекта.

Прямого доступа к данным COM-объекта извне в принципе не существует. Каждое поле объекта скрыто в его недрах и может обслуживаться только методами COM-объекта. Методы объекта также особой коммуникабельностью не отличаются, они упакованы так глубоко, что доступны лишь через свои указатели. В свою очередь указатели на методы хранятся в специальных таблицах адресов функций — в *виртуальных таблицах* объекта (virtual tables). Описание каждой из таких таблиц называют *интерфейсом* (interface). Сколько таблиц — столько у объекта интерфейсов. Но это еще не все, непосредственное обращение к интерфейсу (указателю на виртуальную таблицу адресов методов)

невозможно — вместо этого нам лишь разрешается работать с указателем на интерфейс, это единственное что доступно извне СОМ-объекта.

Число интерфейсов СОМ-объекта определяется замыслом разработчика. В границах отдельного интерфейса программисты обычно объединяют однородные по функциональному назначению методы, таким образом, чем больше задач решает объект, тем значительнее перечень его интерфейсов. Вне зависимости от того, сколько интерфейсов объявит программист, любой СОМ-объект поддерживает один обязательный базовый интерфейс `IUnknown`. Базовый интерфейс, предназначен для информирования клиента СОМ об имеющихся в распоряжении объекта интерфейсах. Получив доступ к обязательно присутствующему интерфейсу `IUnknown`, программа или любой другой объект сразу может обратиться к методу `QueryInterface()` и узнать обо всех остальных имеющихся у этого объекта интерфейсах.

В рассматриваемой в этой главе технологии СОМ доступ клиентского приложения к методам интерфейсов СОМ-объекта осуществляется с помощью механизма *раннего связывания*. Суть раннего связывания заключается в том, что будущему клиентскому приложению еще на этапе компиляции предоставляется полная информация о методах и свойствах интересующих его интерфейсов. В противном случае клиент просто не поймет, каким образом ему следует обращаться к серверу. Информация обо всех тайнах интерфейса описывается в виде специального файла — библиотеки типов.

## Понятие интерфейса

По своей сути интерфейс — это договор о намерениях между СОМ-объектом и клиентским приложением, в котором СОМ-объект гарантирует клиенту предоставление некоторых услуг. Услуги — это определенные в интерфейсе методы.

Весьма значимым аспектом интерфейса является то, что если СОМ-объект заявляет о существовании у него определенного интерфейса, то объявленный интерфейс в обязательном порядке должен быть поддержан внутренними методами объекта и возвращать какие-либо значения.

Интерфейсы могут наследоваться, но при условии, что дочерний интерфейс должен включать все методы своего предка и добавлять к ним свои. Все существующие интерфейсы наследуются от базового интерфейса `IUnknown`.

### Внимание!

Когда речь ведется о наследовании в СОМ надо понимать, что СОМ поддерживает наследование интерфейса, т. е. наследование классом интерфейса базового класса. Но СОМ сознательно не поддерживает наследование реализации, т. е. наследование кода от базового класса.

Выделим еще один ряд правил проектирования СОМ-объекта:

- ☐ клиенты не могут получить услуги от СОМ-объекта никоим образом, кроме как с помощью интерфейса объекта;
- ☐ единожды опубликованный интерфейс никогда не может быть изменен. Необходимость дополнить существующий интерфейс новыми функциональными возможностями приводит к созданию нового интерфейса. Выполнение этого правила гарантирует, что запрашивающее конкретный интерфейс клиентское приложение получит именно то, на что оно рассчитывает;
- ☐ в рамках отдельной системы каждый отдельный интерфейс должен иметь уникальный идентификатор и название начинающегося с символа "I".

На первый взгляд форма определения интерфейса весьма схожа с объявлением обычного класса VCL, но это совсем не так (листинг 8.1). Первое бросающееся в глаза отличие в том, что в момент декларации интерфейс должен идентифицироваться 128-битным *глобальным уникальным идентификатором* (Globally Unique Identifier, GUID). Когда GUID применяют для именования СОМ-интерфейсов, то идентификаторы интерфейсов обычно

именуют IID (Interface ID), если GUID именует класс — говорят об идентификаторе класса CLSID (Class ID).

#### Листинг 8.1. Обобщенная синтаксическая конструкция объявления интерфейса

```
type имя_интерфейса = interface (интерфейс_предок)
    ['{GUID}'] //глобальный уникальный идентификатор класса
    {объявление заголовков методов и их параметров}
    function MyFunction(const IID:GUID):HRESULT; safecall;
end;
```

Необходимость GUID объясняется элементарным здравым смыслом — весьма высока вероятность, что текстовые названия интерфейсов различных разработчиков стали бы совпадать. В результате приложение-клиент, обращаясь к услугам какого-то объекта COM, могло бы получить абсолютно неожиданный ответ от совсем другого объекта- "однофамильца". Генерация GUID осуществляется в соответствии с алгоритмом, предложенным консорциумом Open System Foundation. Алгоритм практически исключает вероятность того, что единожды сгенерированный GUID будет когда-нибудь повторен.

#### Замечание

Для автоматической генерации нового GUID, находясь в редакторе кода Delphi, нажмите комбинацию клавиш <Ctrl>+<Shift>+<G>.

Но на этом особенности определения интерфейса не ограничиваются. Хотя и интерфейс имеет право объявлять заголовки методов, но ни один из его методов не имеет описания — он лишь декларирует названия методов и свойств. Для того чтобы объявленные в интерфейсе методы стали работоспособными, приходится создавать специальные классы, реализующие интерфейс.

Почему интерфейсам не доверили, казалось, самое главное — реализацию? Ответ очевиден. Потому, что реализация интерфейса может изменяться вместе с развитием компонента, а интерфейс оставаться неизменным. Еще одна причина в том, что интерфейсы всегда описываются на стандартном языке определения интерфейсов IDL (благодаря этому интерфейсы COM могут быть доступны практически из любого языка программирования), а реализация интерфейса может создаваться любыми доступными для разработчика способами.

## Базовый интерфейс *IUnknown*

Отправной точкой для разработки всех интерфейсов выступает интерфейс *IUnknown*. Он описан в модуле *System* и поэтому доступен практически во всех проектах VCL. Базовый интерфейс отвечает за выполнение двух задач:

- ❑ поддержку механизма подсчета ссылок на интерфейс;
- ❑ проверку наличия у объекта определенного интерфейса.

Роль интерфейса *IUnknown* в иерархии интерфейсов в чем-то сродни роли абстрактного класса *TObject* в иерархии классов VCL — и тот, и другой являются родоначальниками для всех имеющихся в природе интерфейсов (в случае *IUnknown*) или классов (если речь ведется о *TObject*). Интерфейс обладает тремя базовыми методами (листинг 8.2).

#### Листинг 8.2. Определение базового интерфейса *IUnknown*

```
type
    IUnknown = IInterface;
    IInterface = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function AddRef: Integer; stdcall;
    function Release: Integer; stdcall;
```

```
function QueryInterface(const IID: TGUID; out Obj): HRESULT; stdcall;
end;
```

На порядок декларации входящих в коллекцию методов также наложен ряд ограничений, например, мы не вправе пользоваться классическими для Delphi директивами `virtual`, `dynamic`, `abstract` или `override`. После объявления метода программист должен указать используемое соглашение о вызовах. Так стандартное соглашение `stdcall` позволит обращаться к интерфейсам Windows, для работы с интерфейсами CORBA потребуется соглашение `safecall`. Если никакое соглашение не указано, то Delphi станет использовать соглашение `register`.

#### **Замечание**

Строго говоря, в VCL базовым интерфейсом выступает не `IUnknown`, а `IInterface`. Однако с точки зрения COM именно интерфейс `IUnknown` считается опорным по отношению ко всем интерфейсам.

Каждый интерфейс снабжен внутренним счетчиком ссылок. Благодаря *механизму подсчета ссылок* (reference counting) один и тот же COM-объект умеет обслуживать несколько клиентов и, кроме того, управлять своим собственным существованием. В момент создания интерфейсного объекта его внутреннему счетчику присваивается значение 1. При каждом последующем подключении нового клиента к интерфейсу, в момент получения им указателя на интерфейс, счетчик получает приращение, а при отключении клиента — счетчик уменьшается на единицу. В момент достижения счетчика нулевого значения объект самостоятельно выгружает себя из памяти. Управление счетчиком ведают методы `AddRef()` и `Release()`. Первый из них увеличивает значение счетчика, а второй — уменьшает.

Существенное значение отводится методу `QueryInterface()`. Благодаря ему мы запрашиваем объект на предмет наличия у него интерфейса, идентифицируемого параметром `IID`. Переменная `Obj` передается для получения ссылки на интерфейс. В случае положительного ответа функция самостоятельно обратится к методу `_AddRef()` вызываемого интерфейса и возвратит указатель на интерфейс, иначе выходной параметр `Obj` будет ссылаться "в никуда" — `nil`. У `QueryInterface()` есть еще одна примечательная черта. Когда мы с помощью интерфейса `IUnknown` обращаемся к составному объекту COM (объекту, построенному на базе многокомпонентного класса COM), то метод проверит все идентификаторы реализованных в составном объекте интерфейсов и возвратит нужный.

## **Реализация интерфейса**

Интерфейс — это не более чем договор о намерениях, предоставляемых COM-объектом на всеобщее обозрение. Клиентские приложения, вызвав интерфейс, очень рассчитывают получить запрашиваемую услугу. Но ведь интерфейс — это просто обещание оказать услугу, а каким образом обещания воплощаются в жизнь?

Допустим, что мы планируем создать лучший на свете интерфейс `IMySuperInterface`, тогда нам не остается ничего, кроме того, как объявить его (листинг 8.3).

#### **Листинг 8.3. Пример объявления интерфейса**

```
type IMySuperInterface = interface(IUnknown)
    ['{06D87C21-96A5-405F-AFC2-BB3F4EAE19D6}']
    function VeryGoodFunction: HRESULT;
    //другие функции интерфейса
end;
```

Для того чтобы интерфейс начал приносить пользу, он должен быть реализован — другими словами, заполнен программным кодом. Только выполнив код, COM-объект

поможет клиенту решить его задачу. В языке Delphi интерфейс реализуется только в классе. Для этого при объявлении класса, ответственного за реализацию интерфейса, мы, кроме названия класса-предка, перечисляем поддерживаемые классом интерфейсы (листинг 8.4).

#### Листинг 8.4. Включение интерфейса в объявление класса

```
type TMyClass = class(TComponent, IMySuperInterface)
    function VeryGoodFunction: HResult;
    //другие функции
    destructor Destroy; override; //деструктор объекта
end;
...
function TMyClass.VeryGoodFunction: HResult;
begin
    //код реализации
end;

destructor TMyClass.Destroy;
begin
    inherited;
    {!!! объект следует защищать от разрушения методами Destroy, Free и т. п.}
end;
```

После объявления имени класса следует перечень методов, имена и параметры которых должны целиком и полностью соответствовать методам интерфейса.

При проектировании классов, реализующих интерфейс, необходимо защитить экземпляр класса от несанкционированного уничтожения. Дело в том, что механизм СОМ сам позаботится о разрушении СОМ-объекта и освобождении памяти в тот момент времени, когда объект перестанет быть востребованным. Это событие произойдет в ту же секунду, когда счетчик ссылок на предоставляемый объектом интерфейс получит нулевое значение. Именно поэтому нам пришлось внести некоторые изменения в деструктор объекта.

Объекты, реализующие интерфейс, не следует уничтожать с помощью вызова их деструкторов, по крайней мере до тех пор, пока на их интерфейс существует хотя бы одна ссылка. Если вы полагаете, что объект больше не нужен, то вместо вызова метода `Destroy()` следует просто присвоить неопределенный указатель `nil` переменной, ссылающейся на интерфейс (листинг 8.5). Такое действие автоматически приведет к вызову метода `_Release` и уменьшению счетчика ссылок на единицу.

#### Листинг 8.5. Отказ от использования интерфейса

```
var MyClass:TMyClass;
    I:IMySuperInterface;
begin
    MyClass:=TMyClass.Create(nil);
    I:=MyClass;
    //работа с интерфейсом
    I:=nil;
end;
```

## Порядок вызова сервера СОМ

Прежде чем модель СОМ начнет функционировать, клиентское приложение должно отправить запрос на услуги СОМ-сервера. Каким образом клиентское приложение получит доступ к интересующему его СОМ-серверу? В особенности, если клиент модели СОМ работает в другом адресном пространстве на удаленном компьютере?

Механизм, позволяющий клиенту получать доступ к объектам, расположенным в другом адресном пространстве или на другом компьютере, называется *маршалингом (marshaling)*. Маршалинг отвечает за выполнение двух задач:

- ❑ получение указателя на интерфейс из процесса сервера и организацию доступа к нему из процесса клиента;
- ❑ передачу аргументов методов интерфейса в процесс удаленного объекта.

Еще одна особенность взаимодействия приложений в COM связана с системным реестром Windows. Во время установки COM-приложения в реестр операционной системы вносятся сведения об имеющихся в наличии объектах COM. В первую очередь это однозначно определяющий класс объекта идентификатор CLSID. Идентификаторы классов хранятся в подразделах:

```
HKEY_CLASSES_ROOT\CLSID\{идентификатор COM}
HKEY_LOCAL_MACHINE\SOFTWARE\Имя_класса\CLSID\
```

Кроме того, в реестре хранится имя модуля, содержащего сервер, или, в случае, когда сервер выполняется на другой машине — сетевой адрес.

Выяснив местонахождение сервера, клиент обращается к нему с целью получить ссылки на нужные интерфейсы. Если сервер еще не был запущен, то он стартует, и в нем создается экземпляр объекта, отвечающего за обслуживание обязательного для любого сервера COM интерфейса `IUnknown` и соответствующего ему метода `QueryInterface()`. С этого момента `IUnknown` способен рассказать клиенту обо всех возможностях своего хозяина.

Как только клиент начнет запрашивать у `IUnknown` другие интерфейсы и вызывать предоставляемые ими методы, сервер должен немедленно создать реализующие их экземпляры объектов. Для этих целей в распоряжении сервера имеется отдельный специалист — интерфейс *фабрики классов* `IClassFactory`. Для создания нового экземпляра класса фабрика пользуется шаблоном класса, или, как говорят, *фабричным образцом* (factory pattern). Фабричный образец — это особый класс, применяемый для создания экземпляра другого класса.

## Интерфейс *IClassFactory* и библиотека COM

Фабрика класса — это разновидность COM-объекта, поддерживающего интерфейс `IClassFactory`. Описание ключевого интерфейса `IClassFactory` мы найдем в недрах модуля `ActiveX`, в нем определены всего два метода (листинг 8.6).

### Листинг 8.6. Объявление интерфейса `IClassFactory`

```
IClassFactory = interface(IUnknown)
    ['{00000001-0000-0000-C000-000000000046}']
    function CreateInstance(const unkOuter: IUnknown;
                           const iid: TIID;
                           out obj): HRESULT; stdcall;
    function LockServer(fLock: BOOL): HRESULT; stdcall; ]
end;
```

Первый из указанных методов `CreateInstance()` используется для создания экземпляра COM-объекта. Параметр `Outer` предназначен для создания агрегатов и в обычных условиях не задействуется. Второй параметр `iid` рассчитывает получить идентификатор GUID запрашиваемого интерфейса. После успешного вызова метода в выходной параметр `obj` будет помещен указатель на интересующий нас интерфейс COM-объекта.

### Замечание

Под агрегатом понимается совокупность объектов COM, предоставляющих клиенту некоторый набор собственных интерфейсов. Всеми объектами управляет контроллер, он владеет управляющим интерфейсом IUnknown.

Еще одна особенность метода `CreateInstance()` в том, что она поддерживает два алгоритма создания экземпляра объекта. Если запрашиваемый клиентом COM-объект способен одновременно обслуживать несколько клиентов, то очередная копия объекта не создается, а система лишь увеличивает число ссылок на уже созданный ранее объект. Второй вариант поведения функции `CreateInstance()` основан на идее, что каждого нового клиента должен создаваться новый экземпляр класса.

Второй метод фабрики `LockServer()` блокирует COM-объект, не позволяя системе выгрузить его из памяти, даже когда его не использует ни одно из клиентских приложений. Для включения блокировки в параметр `Lock` должно быть передано значение `true`. Таким образом, мы принудительно увеличиваем на единицу счетчик блокировок и, соответственно, не допускаем уничтожение объекта. Такое поведение полезно в ситуации, когда мы знаем, что хотя в данный момент объект и не нужен, но вскоре вновь появится необходимость в его услугах. Сохранив экземпляр объекта в памяти, мы избавим систему от необходимости разрушения и повторного создания объекта, тем самым сократим временные затраты на рестарт. Для уменьшения счетчика на единицу, в параметр `Lock` передается `false`.

Для запуска COM-сервера в зависимости от стоящих перед клиентом задач применяется одна из трех функций библиотеки COM: `CoGetClassObject()`, `CoCreateInstance()` или `CoCreateInstanceEx()`.

Если планируется создать большое количество однотипных объектов или требуется построить объект с помощью интерфейса, отличного от `IClassFactory` (например, `IClassFactory2`), то следует задействовать функцию

```
function CoGetClassObject(const CLSID : TCLSID; dwClsContext : longint;  
    ServerInfo:PCoServerInfo; const IID:TIID; var pv) : HRESULT; stdcall;
```

Функция не способна напрямую создать экземпляр COM-объекта. Вместо этого на свет появляется посредник — фабрика класса, имеющая право собрать нужный COM-объект. В качестве параметров функция рассчитывает получить `CLSID` класса и `IID` интерфейса фабрики классов. Параметр `dwClsContext` ограничивает контекст исполнения компонента, с которым будет работать клиент. Это может быть: `CLSCTX_INPROC_SERVER` — адресное пространство клиентского процесса (в случае DLL); `CLSCTX_LOCAL_SERVER` — адресные пространства разных процессов, исполняющиеся совместно на одном компьютере; или `CLSCTX_REMOTE_SERVER` — адресные пространства процессов, распределенных в компьютерной сети. Параметр `ServerInfo` используется в технологии DCOM, в нем задаются параметры удаленного сервера. Собрав необходимые данные, функция разыскивает в системном реестре путь к серверу COM и запускает его. Нам возвращается ссылка `pv` на фабрику. Полученный указатель на интерфейс передается функции `CreateInstance()` фабрики классов.

Если сервер создается на том же компьютере, на котором выполняется клиент, и мы не планируем создавать несколько экземпляров COM-объекта, то задействуется функция

```
function CoCreateInstance(const clsid : TCLSID; const Outer: IUnknown;  
    dwClsContext : longint; const IID:TIID; var pv) : HRESULT; stdcall;
```

Функция воспользуется помощью рассмотренной ранее функции `CoGetClassObject()` и метода `CreateInstance()` и возвратит указатель непосредственно компонента (в то время как `CoGetClassObject` возвращает указатель для фабрики класса).

Если сервер расположен на удаленном компьютере, то вызову подлежит функция

```
function CoCreateInstanceEx(const clsid : TCLSID; const Outer: IUnknown;
    dwClsContext : longint; ServerInfo: PCoServerInfo;
    dwCount: longint; rgmqResults: PMultiQIArray) : HRESULT; stdcall;
```

Здесь предусмотрен параметр, хранящий имя удаленного компьютера, на котором располагается интересующий нас сервер. Информация об удаленном сервере должна находиться в реестре клиентской машины по адресу

```
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AppID\{AppID_GUID}\RemoteServerName = <server_name>
```

Функция возвращает массив указателей `rgmqResults`, число ячеек в массиве определяется параметром `dwCount`.

## Реализация фабрики класса, класс *TComObjectFactory*

Фабрика класса — это особый СОМ-объект, единственной целью которого является создание других объектов СОМ. С этой целью разработчик фабрики закладывает в нее специфические знания, необходимые для создания объекта определенного класса.

Реализующий фабричный интерфейс класс `TComObjectFactory` построен на основе `TObject`. Помимо основного для всех СОМ-объектов интерфейса `IUnknown` классом поддерживаются еще два, характерных для фабрики классов интерфейса: `IClassFactory` и `IClassFactory2`.

```
type TComObjectFactory = class(TObject, IUnknown, IClassFactory, IClassFactory2);
```

Интерфейс `IClassFactory` может быть использован для создания всех СОМ-объектов.

Интерфейс `IClassFactory2` выступает логическим развитием интерфейса `IClassFactory`, он позволяет управлять созданием экземпляров СОМ-объектов, для работы которых необходим лицензионный ключ.

В Delphi фабрика класса `TComObjectFactory` создается автоматически СОМ-сервером в момент инициализации приложения-сервера. Для этого вызывается впечатляющий по числу параметров конструктор

```
constructor Create(ComServer: TComServerObject; ComClass: TComClass;
    const ClassID: TGUID; const ClassName, Description: string;
    Instancing: TClassInstancing;
    ThreadingModel: TThreadingModel = tmSingle);
```

Здесь `ComServer` — параметр, связывающий фабрику класса с СОМ-сервером. Все остальные параметры идентичны ключевым свойствам фабрики (см. табл. 8.1 и 8.2) и определяют особенности создаваемого экземпляра класса.

Обычно команда на создание экземпляра фабрики класса подается в разделе инициализации приложения-сервера (листинг 8.7).

### Листинг 8.7. Раздел инициализации

```
//...
initialization
    TTypedComObjectFactory.Create(ComServer, TMyComObject,
        Class_MyComObject, ciMultiInstance, tmApartment);
end.
```

На первый взгляд может показаться, что конструктор не возвращает никаких результатов своих трудов. На самом деле, сразу после создания фабрика класса регистрируется у менеджера фабрик — глобального объекта `ComClassManager`.

Основные свойства фабрики класса (табл. 8.1) доступны только для чтения. Они инициализируются в момент создания экземпляра фабрики.

Таблица 8.1. Свойства фабрики класса *TComObjectFactory*

| Свойство                                                                                                                         | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>property</b> ComServer: TComServerObject;                                                                                     | Определяет COM-сервер, владеющий этой фабрикой класса                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>property</b> ComClass: TClass;                                                                                                | Класс COM-объекта, который будет создавать данный экземпляр фабрики класса. Параметр анализируется методом GetFactoryFromClass() менеджера COM-класса TComClassManager                                                                                                                                                                                                                                                                                                                      |
| <b>property</b> ClassID: TGUID;                                                                                                  | Идентификатор класса CLSID экземпляра фабрики                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>property</b> ClassName: string;                                                                                               | Имя COM-класса                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>property</b> Description: string;                                                                                             | Текстовое описание объекта                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>property</b> Instancing: TClassInstancing;<br><b>type</b> TClassInstancing = (ciInternal, ciSingleInstance, ciMultiInstance); | Значение ciInternal уведомляет, что сервер выполняется внутри адресного пространства вызывающего его приложения. Сервер будет недоступен вне породившего его процесса. Значения ciSingleInstance и ciMultiInstance говорят, что сервер функционирует в отдельном процессе. Разница между этими режимами в том, что ciMultiInstance позволяет одному серверу обслуживать несколько клиентов одновременно. Режим ciSingleInstance требует соблюдения правила: каждому клиенту свой COM-сервер |
| <b>property</b> ProgID: string;                                                                                                  | Информационное свойство, хранящее имя разработчика, название класса и его версию                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>property</b> ThreadingModel: TThreadingModel;                                                                                 | Свойство определяет особенности одновременного вызова COM-объекта несколькими клиентами (табл. 8.2).<br><b>type</b> TThreadingModel = (tmSingle, tmApartment, tmFree, tmBoth, tmNeutral)                                                                                                                                                                                                                                                                                                    |

Таблица 8.2. Потокковая модель TThreadingModel

| Значение    | Описание                                                                                                                                                                       | Особенности реализации                                                                                                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| tmSingle    | Сервер не поддерживает многопоточное обслуживание. Одновременное обращение нескольких клиентов к COM-объекту невозможно                                                        | Запросы клиентов обрабатываются поочередно                                                                                                                                                              |
| tmApartment | Модель COM гарантирует, что в одно и то же время к объекту может обращаться только один клиент. Все клиенты работают с COM-объектом в том же потоке, в котором он и был создан | Объекты могут работать с собственными данными, но глобальные данные должны быть защищены (например, критическими секциями, мьютексами или семафорами)                                                   |
| tmFree      | Объекты могут вызывать любое число потоков в любое время                                                                                                                       |                                                                                                                                                                                                         |
| tmBoth      | Модель Both схожа с Free, за исключением того, что все обратные вызовы выполняются в том же самом потоке                                                                       | Максимальная производительность и гибкость                                                                                                                                                              |
| tmNeutral   | Клиенты могут обращаться к серверу в различных потоках в одно и то же время. Модель COM сама позаботится о том, чтобы одновременно выполняемые запросы не конфликтовали        | Модель доступна только при работе с COM+. Программист должен позаботиться о защите глобальных данных. Использование данной модели недопустимо при совместной работе с визуальными элементами управления |

Самый главный метод фабрики

**function** CreateComObject(const Controller: IUnknown): TComObject;

вызывает конструктор CreateFromFactory() обслуживающей данный класс фабрики.

Регистрация класса создаваемого объекта производится процедурой

```
procedure RegisterClassObject;
```

Метод вызывается в момент запуска exe-файла с COM-сервером.

Регистрация (или снятие с регистрации) объекта COM осуществляет метод

```
procedure UpdateRegistry(Register: Boolean);
```

Метод заносит в реестр сведения об объекте COM и путь к исполняемому файлу.

Для обеспечения лицензионной защиты объекта от вездесущих пиратов установите в `true` свойство

```
property SupportsLicensing: Boolean;
```

С этого момента для создания COM-объекта фабрика класса будет требовать ключ.

Лицензионная информация о COM-объекте находится в свойстве

```
property LicString: WideString;
```

Если вы хотите, чтобы все происходящие в COM исключительные ситуации сопровождалось выводом окна с сообщением об ошибке, то установите в `true` свойство

```
property ShowErrors: Boolean;
```

Идентификационный код интерфейса, с которым связана ошибка, доступен посредством свойства:

```
property ErrorIID: TGUID;
```

## Реализация COM-объекта в Delphi

В Delphi COM-объект создается на основе сотрудничества как минимум двух классов (рис. 8.1). Собственно сам COM-объект получается из вполне рядового класса `TComObject`. Это прямой наследник `TObject`, описанный в модуле `ComObj`. Как и большинство других классов VCL, наш новый знакомый обладает свойствами и методами и имеет полное право их реализовывать, т. е. заполнять программным кодом. Для того чтобы обычный класс `TComObject`, целиком и полностью исповедующий каноны ООП, смог претендовать на звание полноценного COM-объекта, ему нужен помощник. В терминах Delphi такой помощник (из-за приставки "Co", добавляемой к имени реального объекта в момент его создания) называется `CoClass`. Задача `CoClass` — создать вокруг стандартного `TComObject` такую оболочку, чтобы со стороны внешнего наблюдателя (клиента COM) наш объект выглядел точно так, как и любой другой объект COM. Благодаря `CoClass` наш объект приобретает всю атрибутику обычного COM-объекта, в том числе произвольное число интерфейсов.

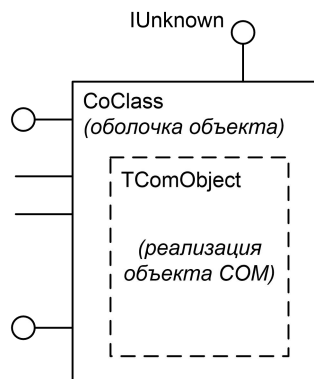


Рис. 8.1. Структура COM-объекта в Delphi

## Класс *TComObject*

Основная задача *TComObject* — наполнение программной логикой интерфейса *IUnknown* и поддержка основной функциональности СОМ-объекта. В частности здесь хранится идентификатор CLSID, имеется механизм построения агрегатов (составных СОМ-объектов) и поддерживается так необходимый для обработки исключительных ситуаций контроллеров автоматизации интерфейс *ISupportErrorInfo*.

```
type TComObject = class(TObject, IUnknown, ISupportErrorInfo);
```

Созданием экземпляра класса заведуют три конструктора, основной и наиболее универсальный из них:

```
constructor CreateFromFactory(Factory: TComObjectFactory;  
                             const Controller: IUnknown);
```

Параметр *Factory* определяет фабрику класса, а параметр *Controller* задействуется только при формировании агрегата.

Сразу после появления на свет объект проходит инициализацию:

```
procedure Initialize;
```

Благодаря свойству

```
property Factory: TComObjectFactory; //только для чтения
```

наш объект помнит об участвующей при его рождении фабрике класса.

Если СОМ-объект составной, то доступ к интерфейсу *IUnknown* вложенного в него объекта мы обнаружим в свойстве

```
property Controller: IUnknown;
```

Если объект не составной, то свойство возвратит *nil*.

Число ссылок на интерфейс СОМ-объекта находится в свойстве

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

Управлением ссылками занимается интерфейс *IUnknown*, поэтому внутри *TComObject* реализуются три метода: *AddRef()*, *Release()* и *QueryInterface()*.

## Класс *TTypedComObject*

В иерархии наследования сразу за *TComObject* следует класс *TTypedComObject*, позволяющий создавать объекты СОМ на основе библиотеки типов. Между предком и потомком всего одно отличие — интерфейс *IProvideClassInfo*.

```
type TTypedComObject = class(TComObject, IProvideClassInfo)
```

Названный выше интерфейс функционирует в интересах единственного введенного в классе метода

```
function GetClassInfo(out TypeInfo: ITypeInfo): HRESULT; stdcall;
```

Функция предоставляет нам информацию о типах, с этой целью она возвращает указатель на класс-надстройку *CoClass* в выходной параметр *TypeInfo*.

### *Замечание*

Библиотека типов (type library) представляет собой описание имеющихся в распоряжении объектов СОМ интерфейсов, методов, свойств и структур. Библиотека компилируется в двоичный файл и может использоваться клиентским приложением, как некий эквивалент справки. Именно благодаря библиотеке типов реализуется механизм раннего связывания, позволяющий будущему клиенту СОМ узнать все об интерфейсах сервера еще на этапе компиляции.

## Класс *TComServer*

Класс *TComServer* описан в модуле *ComServ*. Класс предназначен для создания экземпляров СОМ, для этого в рамках класса инкапсулировано приложение СОМ-сервер. Доступ к

серверу реализован посредством автоматически создаваемой глобальной переменной `ComServer`. Цепочка наследования класса невелика, `TComServer` построен на основе абстрактного класса `TComServerObject`, а он в свою очередь берет начало от `TObject`.

Модуль с сервером интегрируется в проект самостоятельно сразу после добавления в проект СОМ-объекта. Программисту не следует задумываться над созданием экземпляра `TComServer`, сервер создается автоматически, а доступ к нему можно получить через глобальную переменную `ComServer:TComServer`.

Имя сервера устанавливается процедурой

```
procedure SetServerName(const Name: string);
```

Если сервер работает совместно с библиотекой типов, то имя сервера назначается в соответствии с установками библиотеки.

Библиотека типов загружается методом

```
procedure LoadTypeLib;
```

Процессом регистрации (и снятия с регистрации) СОМ-сервера в системном реестре ведает процедура

```
procedure Initialize;
```

В рамках метода `Initialize()` вызывается процедура

```
procedure UpdateRegistry(Register: Boolean);
```

Для регистрации в системном реестре всех входящих в сервер объектов в процедуру передается значение `true`. Обратный результат (стирание данных о СОМ-объектах в реестре) достигается при передаче `false`.

Из девяти свойств СОМ-сервера всего одно доступно для редактирования:

```
property UIInteractive: Boolean;
```

Это свойство определяет, должен ли СОМ-сервер иметь интерактивный интерфейс с пользователем. Интерактивность (состояние `true`) проявляется в информировании сервером пользователя обо всех возникающих исключительных ситуациях, если же сервер выполняется как отдельная служба (сервис), то целесообразно передавать в это свойство значение `false`.

В табл. 8.3 представлен перечень информационных свойств сервера автоматизации.

*Таблица 8.3. Информационные свойства сервера автоматизации `TComServer`*

| Свойство                                 | Описание                                                                                                                                                                                                  |
|------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>property</b> ObjectCount: Integer;    | Указывает число работающих в сервере объектов                                                                                                                                                             |
| <b>property</b> IsInprocServer: Boolean; | Значение <code>true</code> указывает, что это внутренний сервер                                                                                                                                           |
| <b>property</b> ServerFileName: string;  | Полный путь и имя файла сервера                                                                                                                                                                           |
| <b>property</b> HelpFileName: string;    | Имя связанного с модулем файла помощи                                                                                                                                                                     |
| <b>property</b> ServerKey: string;       | Значение "LocalServer32" указывает на то, что это исполняемый ехе-файл. Значение "InprocServer32" — сервер реализован в dll-библиотеке                                                                    |
| <b>property</b> ServerName: string;      | Имя файла сервера                                                                                                                                                                                         |
| <b>property</b> StartMode: TStartMode;   | smAutomation — сервер запускается в качестве сервера автоматизации в ответ на обращение контроллера автоматизации, smRegServer — сервер запущен лишь с целью регистрации в реестре, smStandalone — сервер |

|                                    |                                                                        |
|------------------------------------|------------------------------------------------------------------------|
|                                    | запущен пользователем, smUnregServer — приложение удаляется из реестра |
| <b>property</b> TypeLib: ITypeLib; | Обеспечивает доступ к ассоциированной с сервером библиотеке типов      |

В классе TComServer предусмотрен один-единственный обработчик события

```
property OnLastRelease: TLastReleaseEvent;
```

```
type TLastReleaseEvent = procedure(var Shutdown: Boolean) of object;
```

Он призван реагировать на тот момент, когда последний клиент отказывается от услуг сервера. Единственный параметр метода Shutdown определяет: прекращать работу сервера (значение true) или оставить его в памяти (false).

## Пример СОМ-проекта

После такого большого теоретического вступления применим наши знания о СОМ в практической области. На начальном этапе не станем проявлять излишние амбиции и просто научим СОМ-сервер простейшему — осуществлять элементарные арифметические операции. Задачи клиентского приложения и того проще — суметь воспользоваться услугами СОМ-сервера.

Создайте новую папку, например, с названием COM\_DEMO. Внутри нее еще две паки — SERVER и CLIENT, соответственно для размещения исходных файлов сервера и клиента. Свою работу мы начинаем с проектирования СОМ-сервера.

## СОМ-сервер

Создаем новый обычный проект VCL Forms Application. Переименуем главную форму проекта в frmMain и сразу же сохраним проект будущего СОМ-сервера в каталоге <x:\COM\_DEMO\Server>. Во время сохранения модуль, соответствующий главной форме, назовем main.pas, а сам проект — mathcomsrv.dproj.

Дабы приложение СОМ получило почетное право называться сервером, оно должно обладать хотя бы одним СОМ-объектом. Для внедрения СОМ-объекта выберите пункт главного меню **File | New | Other**. В появившемся диалоговом окне **New Items** обратитесь к страничке **ActiveX**, найдите на ней значок **COM Object** и смело нажмите кнопку **OK**. Наши старания будут вознаграждены — на экране возникнет новое диалоговое окно определения основных характеристик будущего СОМ-объекта — **COM Object Wizard** (рис. 8.2).

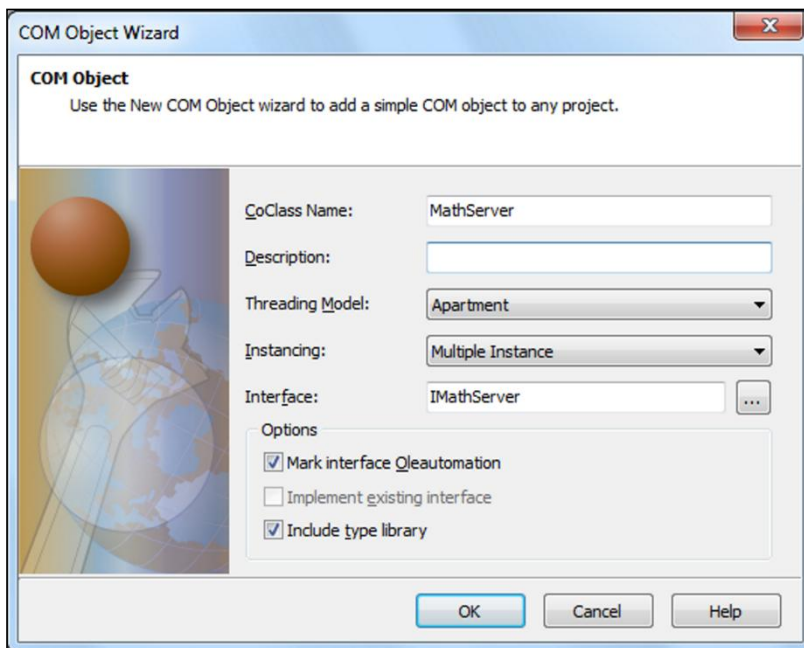


Рис. 8.2. Помощник настройки COM-объекта

## Помощник настройки COM-объекта

Первое, что потребует от нас помощник **COM Object Wizard**, — определить имя вспомогательного класса **CoClass Name**. Придумываем новое имя, раскрывающее смысл нашей задумки (я предлагаю `MathServer`), и вводим его в самой верхней строке диалогового окна. Заметьте, что, определяя имя класса, я не использую принятый для классов VCL префиксный символ "T".

Вторая строка предназначена для комментариев разработчика.

Третья строка диалога **COM Object Wizard** определяет модель потоков — особенность одновременного вызова COM-объекта несколькими клиентами. Это седьмой параметр `ThreadingModel` метода `Create()` класса `TComObjectFactory` (см. табл. 8.2). В нашем примере мы воспользуемся моделью `Free`.

В четвертой строке окна помощника указываются особенности запуска содержащего COM-объект приложения-сервера. Помощник предлагает нам выбрать один из трех вариантов поведения: **Internal**, **Single Instance** и **Multiple Instances**. Это не что иное, как шестой параметр конструктора `Create()` фабрики класса, соответствующий типу данных `TClassInstancing`. Возможные варианты были описаны ранее в табл. 8.1. В нашем примере лучше всего воспользоваться режимом **Multiple Instances**, допускающим создание неограниченного числа COM-объектов в приложении-сервере.

Заключительная строка диалогового окна **COM Object Wizard** содержит имя интерфейса, она заполняется автоматически — интерфейс получает имя `IMathServer`.

Оставьте установленным флажок **Mark Interface Oleautomation**. Хотя мы и не нацелены на создание сервера автоматизации, это действие упростит регистрацию интерфейсов в системном реестре.

При включенном флажке **Include type library** помощник сгенерирует библиотеку типов, содержащую всю информацию об интерфейсах COM-объекта. Эти данные упростят нашу работу при проектировании клиентского приложения, поэтому не снимайте "галочку" с элемента управления.

После настройки всех первоначальных характеристик COM-объекта щелкните по кнопке **ОК**, и помощник сформирует шаблон кода.

## Шаблон кода с описанием класса

Сразу после нажатия кнопки **ОК** диалогового окна **COM Object Wizard** среда Delphi примется за работу.

- ❑ На экране нашего компьютера появится окно редактора библиотеки типов с заголовком `mathcomsrv.ridl`. Для Delphi файл в формате RIDL (Restricted Interface Definition Language) относительная новинка, он появился в 2009-й версии. Предыдущие версии среды работали с бинарным файлом в формате TLB.
- ❑ К исходному коду нашего проекта добавится еще один модуль (по умолчанию он назван `Unit1.pas`) с описанием нового класса `TMathServer`.

Вновь даем команду на сохранение всего проекта. В процессе сохранения переименуем невзрачное `Unit1` в название, отражающее предназначение модуля — `Unit_TMathServer.pas`, тем самым мы оставим себе подсказку, что указанный модуль содержит реализацию класса `TMathServer`. Остальные модули не переименовывайте и сохраняйте с именами предлагаемыми системой.

На первое время редактор библиотеки типов оставим в покое, к нему мы вернемся немного позднее, а пока уделим внимание шаблону `Unit_TMathServer` с описанием класса объекта `TMathServer` (листинг 8.8).

### *Замечание*

Директива `{$WARN SYMBOL_PLATFORM OFF}` указывает компилятору на то, что следующий за ней код может работать на всех известных Delphi платформах.

### Листинг 8.8. Шаблон математического сервера

```
unit Unit_TMathServer;

{$WARN SYMBOL_PLATFORM OFF}

interface
uses Windows, ActiveX, Classes, ComObj, mathcomsrv_TLB, StdVcl;

type
  TMathServer = class(TTypedComObject, IMathServer)
  end;

implementation
uses ComServ;

initialization
  TTypedComObjectFactory.Create(ComServer, TMathServer, Class_MathServer,
    ciMultiInstance, tmFree);
end.
```

Сразу обратите внимание на строку `uses` со списком подключаемых модулей. Некоторые имена библиотек нам пока не знакомы:

- ❑ модуль `AciveX` содержит определения основных интерфейсов, например фабрики классов `IClassFactory`;
- ❑ модуль `ComObj` включает описание базовых для модели COM-классов: `TComObject`, `TComObjectFactory`, `TComServerObject` и т. п.;
- ❑ модуль `StdVcl` содержит объявления стандартных интерфейсов библиотеки VCL;
- ❑ модуль `mathcomsrv_TLB` автоматически создается средой программирования и содержит информацию о фабрике классов, интерфейсе COM-объекта и их GUID.

В начале раздела `implementation` к проекту подключается еще один модуль — `ComServ`. В этом модуле описан один-единственный класс `TComServer`, инкапсулирующий глобальный

объект COM-сервер. Доступ к объекту обеспечивается через глобальную переменную `ComServer`.

В модуле декларируется описанный нами в диалоге настройки COM-объекта класс `TMathServer`, он формируется на базе `TTypedComObject` и предназначен для обслуживания "одноименного" (пока не определенного) интерфейса `IMathServer`. Раздел инициализации содержит строку, ответственную за создание фабрики класса `TTypedComObjectFactory`. Синтаксис конструктора не вполне стандартный — мы привыкли, что конструктор Delphi возвращает созданный им объект, здесь на первый взгляд такого не происходит. Вместо этого, втайне от нас, фабрика класса регистрируется у менеджера фабрик. Для этого в теле конструктора фабрики скрыт метод `ComClassManager.AddObjectFactory(self)`.

## Библиотека типов

Одной из проблем, возникающих перед программистом во время изучения COM-объекта, становится поиск поясняющей документации. Переливающийся всеми цветами радуги класс может оказаться бесполезным, если мы не понимаем, как с ним обращаться. Так, при проектировании COM-приложения клиентское приложение обязано владеть полной информацией об интерфейсной части COM-сервера, иначе оно не сможет воспользоваться его услугами. В частности клиент COM должен знать о:

- ☐ инкапсулированных в сервере COM-объектах;
- ☐ списке интерфейсов, поддерживаемых каждым из объектов;
- ☐ составе функций в каждом из интерфейсов;
- ☐ перечне аргументов функций.

Наилучшим хранилищем знаний такого рода считается *библиотека типов*. Библиотека типов не выступает обязательной частью компонента COM, но она в состоянии значительно упростить работу пользователя COM-объекта, в особенности, когда идет речь о компоненте, созданном сторонним производителем на другом языке программирования. Кроме того, помимо справочной информации, на основании данных из библиотеки могут генерироваться необходимые заголовочные файлы.

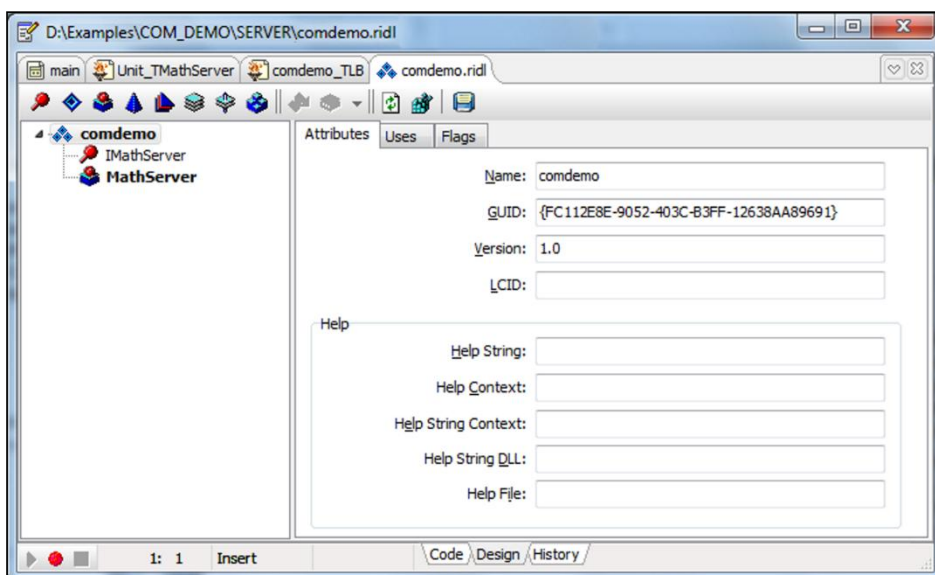


Рис. 8.3. Редактор библиотеки типов

Среда разработки Delphi вооружена редактором библиотеки типов (рис. 8.3). Для вызова редактора в главном меню Delphi найдите пункт **View | Type Library**. Если вернуться к нашему примеру, то после обращения к редактору в дереве объектов библиотеки типов

мы увидим два объекта: интерфейс `IMathServer` и оболочку будущего COM-объекта — `CoClass MathServer`.

Рассмотрим, какую помощь сможет оказать библиотека типов при создании нового метода. Как и все настоящие математики, свою работу начнем с реализации операции сложения двух чисел. Тогда прототип будущей функции будет выглядеть примерно так:

```
function Addition(X: Double; Y: Double;
                 out Outcome: OleVariant): HRESULT; stdcall;
```

У метода имеются два входных параметра — `x` и `y`, они предназначены для передачи суммируемых чисел. В случае успеха метод возвратит результат сложения в выходной параметр `Outcome`. Заметьте, что в технологии COM исходящие параметры проще типизировать как `OleVariant`.

Выделите узел интерфейса `IMathServer` и нажмите кнопку **New Method**. Под узлом `IMathServer` появится дочерний элемент — прототип будущей функции. Переименуйте этот элемент в **Addition** — это название новой функции. Для нового метода системой сразу назначается новый идентификатор `ID=101` (страница атрибутов метода **Attributes**). Для определения параметров перейдите к странице **Parameters**. Воспользовавшись кнопкой **Add**, добавьте три строки с параметрами функции (рис. 8.4). Результат сложения мы возвращаем в выходной параметр `Outcome`, а сам метод возвращает значение типа `HRESULT`.

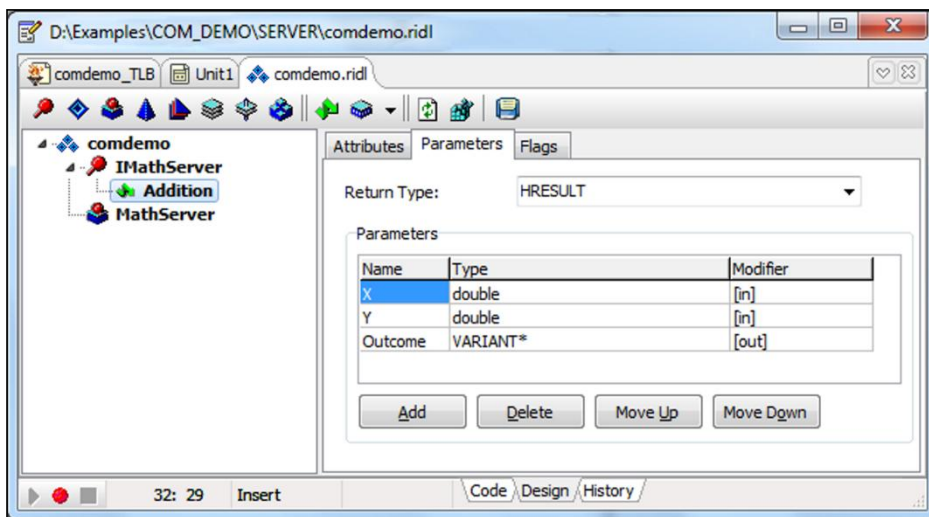


Рис. 8.4. Редактор библиотеки типов с новым методом `Addition()`

#### **Внимание!**

Все COM-методы должны возвращать значение типа `HRESULT`. В случае успешного завершения метод вернет значение `S_OK`, значение `S_FALSE` свидетельствует о том, что метод не может выполнить операцию, если интерфейс не поддерживается — `E_INTERFACE`, катастрофическая ошибка — `E_UNEXPECTED`, и т. п.

На панели управления редактора нажмите кнопку **Refresh Implementation**. Это действие заставит редактор библиотеки типов объявить метод в интерфейсе `IMathServer` и включить его в перечень методов класса `TMathServer`, отвечающего за реализацию нового метода.

#### **Замечание**

Выходные параметры интерфейсных методов должны объявляться в виде указателей.

Давайте проверим, насколько хорошо потрудились редактор библиотеки и типов. Код файла `mathcomsrv_TLB.pas` (в нем находится описание интерфейса `IMathServer`) представлен в листинге 8.9, для большей краткости из него исключены комментарии, автоматически создаваемые Delphi.

### Листинг 8.9. Результаты работы редактора библиотеки типов

```
unit mathcomsrv_TLB;
{$TYPEDADDRESS OFF} {$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON} {$VARPROPSETTER ON} {$ALIGN 4}

interface
uses Windows, ActiveX, Classes, Graphics, OleServer, StdVCL, Variants;

const
  mathcomsrvMajorVersion = 1;
  mathcomsrvMinorVersion = 0;
  LIBID_mathcomsrv: TGUID = '{FC112E8E-9052-403C-B3FF-12638AA89691}';
  IID_IMathServer: TGUID = '{457E9417-CBC2-47C3-8DA8-4AEECADF34FB}';
  CLASS_MathServer: TGUID = '{F24AA614-54E5-4525-9B13-AF46E9B87C55}';

type
  IMathServer = interface;
  MathServer = IMathServer;
  IMathServer = interface(IUnknown)
    ['{457E9417-CBC2-47C3-8DA8-4AEECADF34FB}']
    function Addition(X: Double; Y: Double;
      out Outcome: OleVariant): HRESULT; stdcall;
  end;

  CoMathServer = class
    class function Create: IMathServer;
    class function CreateRemote(const MachineName: string): IMathServer;
  end;

implementation
uses ComObj;

class function CoMathServer.Create: IMathServer;
begin
  Result := CreateComObject(CLASS_MathServer) as IMathServer;
end;
class function CoMathServer.CreateRemote(const MachineName: string): IMathServer;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_MathServer) as IMathServer;
end;
end.
```

В разделе `type` модуля у интерфейса `IMathServer` появилось объявление заголовка метода `Addition()`. Как и положено для интерфейсов, в модуле вы не найдете ни строчки кода, отвечающей за реализацию метода.

Сразу за описанием интерфейса следует объявление класса-оболочки `CoMathServer`. Напомню (см. рис. 8.1), что он предназначен для создания надстройки над реализующим интерфейс классом `TMathServer`. В разделе `implementation` модуля мы обнаружим два метода фабрики класса, предназначенные для запуска СОМ-сервера: `CreateRemote()` — на удаленном компьютере; `Create()` — на том же компьютере с приложением СОМ-клиента.

За реализацию интерфейса `IMathServer` в нашем проекте отвечает класс `TMathServer`. Откройте в редакторе кода листинг с описанием класса (модуль `Unit_TMathServer`). В секции защищенных объявлений `protected` появился заголовок нашей новой функции, а в секции `implementation` описана заготовка ее реализации. Дополните листинг единственной строкой `Outcome=X+Y` (листинг 8.10).

### Листинг 8.10. Доработка модуля `Unit_TMathServer`

```

unit Unit_TMathServer;
{$WARN SYMBOL_PLATFORM OFF}

interface
uses Windows, ActiveX, Classes, ComObj, mathcomsrv_TLB, StdVcl;

type TMathServer = class(TTypedComObject, IMathServer)
protected
    function Addition(X, Y: Double;
                      out Outcome: OleVariant): HRESULT; stdcall;
end;

implementation
uses ComServ;

function TMathServer.Addition(X, Y: Double;
                              out Outcome: OleVariant): HRESULT;
begin
    Outcome:=X+Y; {реализация метода}
end;

initialization
    TTypedComObjectFactory.Create(ComServer, TMathServer, Class_MathServer,
                                  ciMultiInstance, tmFree);
end.

```

Научившись описывать методы, попробуем свои силы на определении свойств. Допустим, что нам надо объявить свойство, способное обслуживать целочисленное значение.

Для определения нового свойства в редакторе типов выделите узел с интерфейсом `IMathServer` и на панели управления программой найдите кнопку **New Property**. Щелчок по кнопке приведет к созданию двух новых одноименных узлов **Property1**. Такое "раздвоение личности" свойства объясняется тем, что один из узлов отвечает за чтение данных из свойства, второй узел решает обратную задачу — записывает в свойство данные.

Переименуйте любой из узлов нового свойства, например в **IntValue**. Тип данных, описываемых свойством, определяется на странице **Attributes** в раскрывающемся списке **Type**. Для решения нашей задачи подойдет значение по умолчанию — целое число `long` (рис. 8.5).

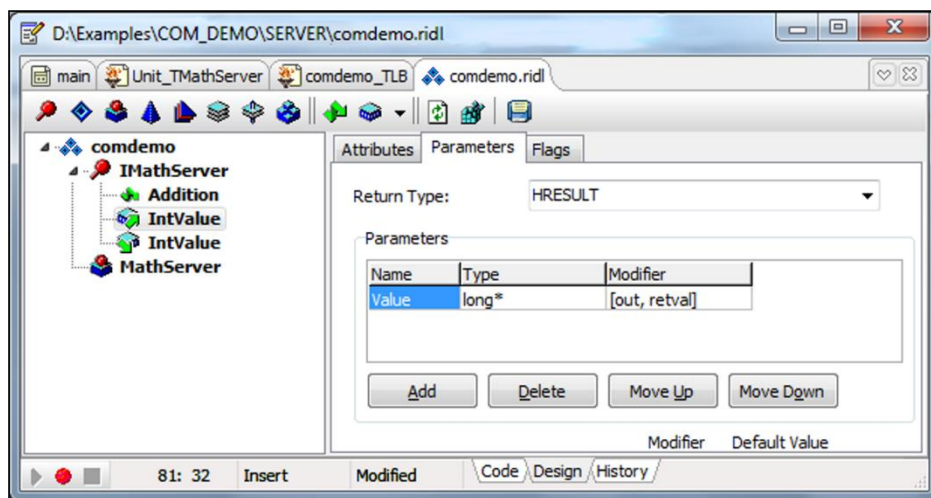


Рис. 8.5. Создание свойства

Чтение значения из свойства и запись нового значения в свойство производится посредством вызова соответствующих методов. Имена методов формируются по принципу:

- ❑ чтение из свойства: "Get\_" + <Имя\_свойства>;
- ❑ запись в свойство: "Set\_" + <Имя\_свойства>.

Страница **Parameters** определяет параметры для соответствующих методов Get... и Set....

Для того чтобы необходимые изменения попали в модули описания интерфейса и реализации, вновь на панели управления редактора библиотеки типов нажмите кнопку **Refresh Implementation**.

После этого в модуле Unit\_TMathServer мы найдем объявление и реализацию методов чтения из свойства и записи в свойство (листинг 8.11). Хранилищем значения выступит поле fIntValue.

**Листинг 8.11. Объявление свойства в модуле Unit\_TMathServer**

```
type
  TMathServer = class(TTypedComObject, IMathServer)
  protected
    fIntValue: integer;
    function Addition(X, Y: Double;
                     out Outcome: OleVariant): HRESULT; stdcall;
    function Get_IntValue(out Value: Integer): HRESULT; stdcall;
    function Set_IntValue(Value: Integer): HRESULT; stdcall;
  end;
```

Нам осталось заполнить методы чтения/записи кодом из листинга 8.12.

**Листинг 8.12. Чтение и запись значения в свойство**

```
function TMathServer.Get_IntValue(out Value: Integer): HRESULT;
begin
  Value := fIntValue; //читаем свойство
end;

function TMathServer.Set_IntValue(Value: Integer): HRESULT;
begin
  fIntValue := Value; //заносим данные в свойство
end;
```

#### **Внимание!**

Занесенное в свойство значение доступно только подключенному к COM-серверу клиенту и хранится только на время сеанса. Поэтому свойства интерфейса не могут использоваться для обмена данными между различными клиентами.

## **Главная форма сервера**

По большому счету, разрабатываемый нами сервер поддержки математических операций не нуждается в визуальных элементах управления. Все запрашиваемые клиентом услуги реализуются в памяти и не требуют отображения на экране. С другой стороны наличие в проекте сервера формы может оказать хорошую услугу для администратора системы, информируя его о текущем состоянии COM-сервера. Для реализации задуманного нам потребуется помощь трех компонентов: многострочного редактора Memo1, метки Label1 и таймера Timer1.

В листинге 8.13 представлен фрагмент кода main.pas, предлагающий вариант применения информационных свойств сервера.

### Листинг 8.13. Работа с информационными свойствами сервера

```
unit main;
...
var frmMain: TfrmMain;

implementation
  uses ComServ;
  {$R *.dfm}

procedure TfrmMain.FormShow(Sender: TObject);
begin
  with Memol.Lines do
    begin
      Clear;
      Add('Имя: '+#9+ComServer.ServerName);
      Add('Тип: '+#9+ComServer.ServerKey);
      Add('Имя файла: '+#9+ComServer.ServerFileName);
      case ComServer.StartMode of
        smAutomation : Add('Режим запуска - по запросу клиента COM');
        smStandalone  : Add('Режим запуска - запущен пользователем');
      end;
    end;
end;

procedure TfrmMain.Timer1Timer(Sender: TObject);
begin
  Label1.Caption:=Format('Активных клиентов %d',[ComServer.ObjectCount]);
end;
end.
```

## Регистрация сервера

Запустите исполняемый файл сервера. В идеальной ситуации в момент своего первого запуска COM-сервер автоматически регистрируется сам и регистрирует свои COM-объекты в системном реестре Windows. Это операция должна осуществляться в момент инициализации приложения — вызов метода `Application.Initialize()`.

Однако требования политики безопасности современных версий Windows (в первую очередь речь идет о Windows Vista, Windows Server 2008 и Windows 7) активно препятствует такому своеволию со стороны приложения, разработанного не в стенах Microsoft. Поэтому для гарантированной регистрации COM-сервера на компьютере лучше воспользоваться командной строкой, запустив с ее помощью сервер с ключом `/regserver` (рис. 8.6). В этом случае операционная система беспрекословно выполнит наши указания.

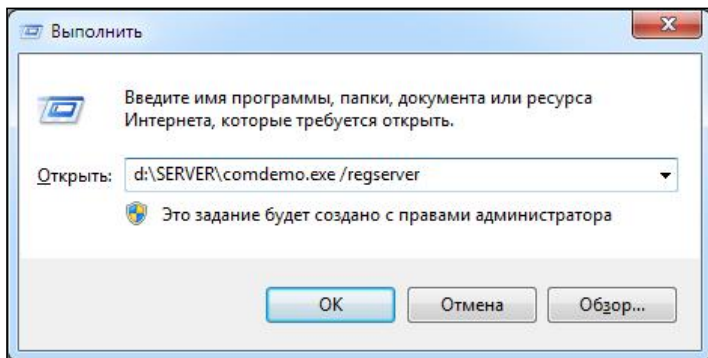


Рис. 8.6. Регистрация COM-сервера из командной строки

Для регистрации серверов, представленных в виде динамических библиотек, целесообразно воспользоваться стандартной утилитой Windows RegSvr32.exe. Регистрация COM производится в ветви реестра:

HKEY\_CLASSES\_ROOT\CLSID\{идентификатор COM}

В названном подразделе сохраняются две наиболее важные составляющие: имя файла сервера и его CLSID. Кроме того, в подраздел **LocalServer32** передается путь к исполняемому файлу сервера, **TypeLib** — идентификатор библиотеки типов, **Version** — версия сервера (рис. 8.7).

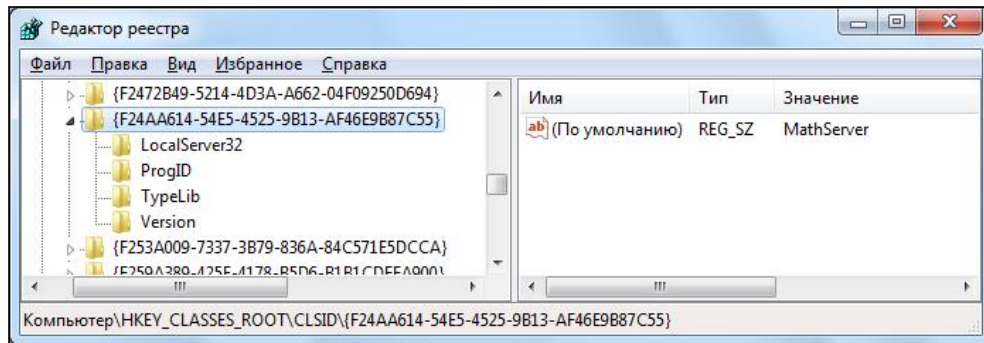


Рис. 8.7. Ветвь реестра с регистрационными данными COM-сервера

Интерфейсы регистрируются в ветви:

HKEY\_CLASSES\_ROOT\Interface\{идентификатор интерфейса}

#### **Внимание!**

Для удаления из системного реестра всех регистрационных данных COM-сервера, его COM-объектов и интерфейсов запустите приложение-сервер из командной строки с ключом /unregserver.

В современных версиях Delphi для регистрации (или снятия с регистрации) COM-сервера можно воспользоваться пунктами меню **Run | ActiveX Server | Register** и **Run | ActiveX Server | Unregister**.

## COM-клиент

В сравнении с COM-сервером процесс создания клиентского приложения удивительно прост. Он включает один обязательный этап — подключение к приложению описания интерфейсной части входящих в состав сервера COM-объектов. После этого нам остается лишь обращаться к нужному интерфейсу. Проверим это утверждение на практике. Для этого создадим новое приложение VCL и сохраним его в каталоге CLIENT под именем MathClient.dproj.

## Импорт библиотеки типов

Для того чтобы клиентское приложение могло воспользоваться услугами COM-сервера, оно нуждается в знаниях об интерфейсной части сервера. Эта информация сохранена нами в каталоге приложения-сервера в библиотеке типов — файле mathcomsrv.ridl. В Delphi предусмотрено несколько способов импорта. В Delphi XE2 простейший из них — с помощью помощника импорта компонентов. Выбрав пункт меню **Component | Import Component...**, мы должны ответить на ряд вопросов помощника (рис. 8.8).

1. В первом окне выбираем пункт **Import Type Library** и нажимаем кнопку **Next**.
2. Во втором окне помощника находим интересующий нас сервер. В нашем случае это mathcomsrv. Если в перечне доступных библиотек вы не смогли обнаружить требуемый сервер, то повторите операцию регистрации приложения-сервера.
3. В третьем окне снимаем флажок **Generate Component Wrapper**. Этим действием мы проинформируем Delphi, что не собираемся генерировать новый компонент VCL. Также можно изменить путь к модулю,

для этого стоит настроить элемент управления **Unit Dir Name**. Остальные элементы управления оставляем в состоянии по умолчанию.

4. В заключительном 4-м окне выделяем переключатель **Add unit to MathClient.dproj project**, тем самым уведомляем Delphi, что в результате импорта библиотека типов должна быть подключена к проекту клиента.

В результате к клиентскому приложению будет подключен уже знакомый нам по COM-серверу интерфейсный модуль `mathcomsrv_TLB`.

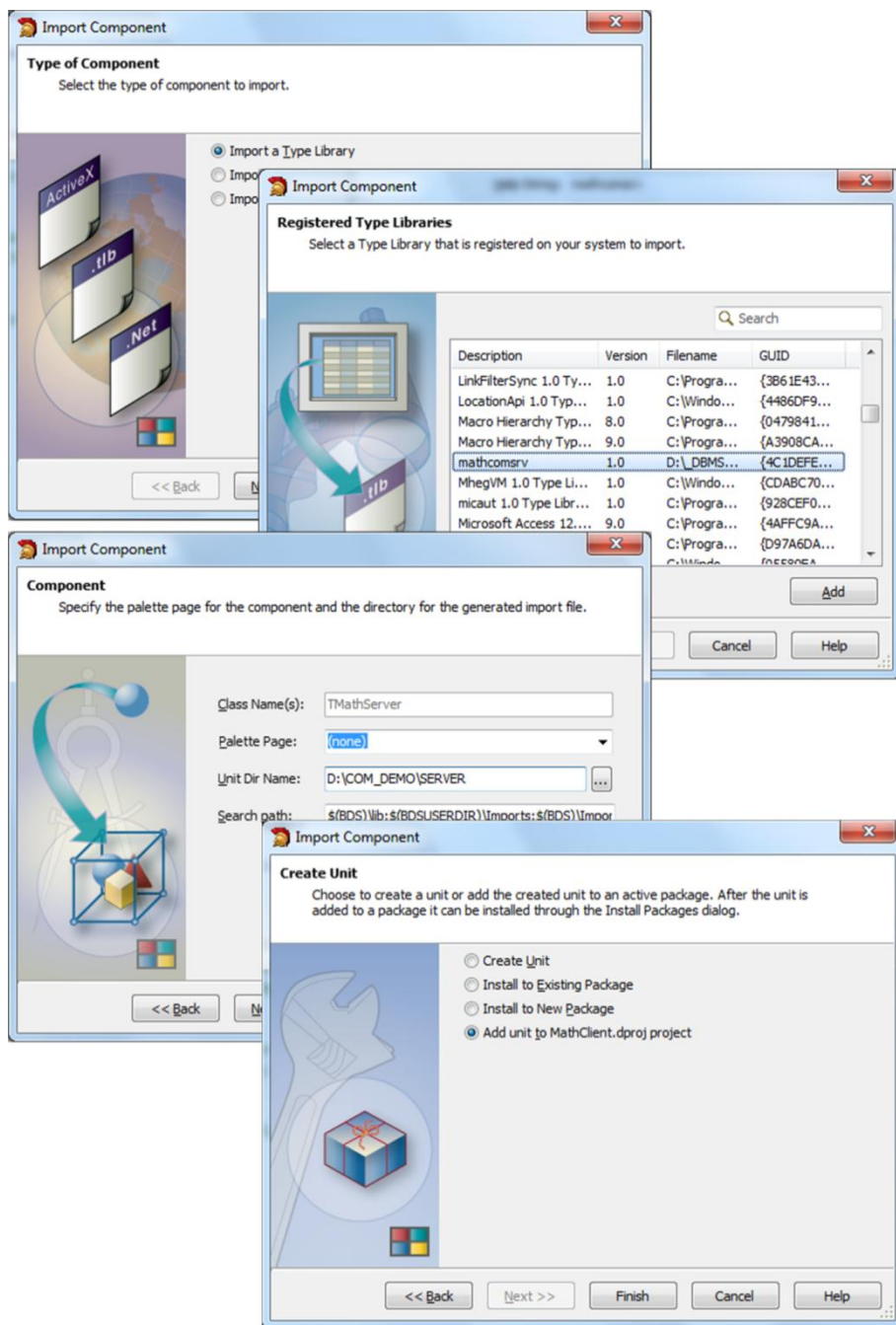


Рис. 8.8. Этапы импорта библиотеки типов

## Обращение к COM-объекту

Обращение к методу `Addition()` из клиентского приложения предложено в листинге 8.14.

### Листинг 8.14. Вызов метода сложения

```
function _Addition(X, Y: double): double;
```

```

var Outcome: OleVariant;
    I : IMathServer;
begin
    try
        I:=CoMathServer.Create;    //создаем объект и получаем его интерфейс
        I.Addition(X,Y,Outcome);    //обращаемся к методу сложения
        Result:=Outcome;            //получаем результат
        I:=nil;                    //освобождаем интерфейс
    except
        on EIntfCastError do
            raise Exception.Create('Интерфейс не поддерживается!');
        end;
    end;
end;

```

Для работы со свойством `IntValue` в интерфейсе предусмотрены два метода: `Set_IntValue()` и `Get_IntValue()`. Пример работы представлен в листинге 8.15.

**Листинг 8.15. Обращение к свойству**

```

var I : IMathServer;
    Value:integer;
begin
    i:=CoMathServer.Create;
    i.Set_IntValue(1);    //присвоим свойству значение
    i.Get_IntValue(value); //читаем данные из свойства
    i:=nil;
    ShowMessage(IntToStr(value));
end;

```

Напомню, что хранящиеся в свойствах интерфейса данные доступны только на время сессии клиента с сервером. После отключения клиента от сервера свойства обнуляются. Поэтому если услуги сервера необходимы на протяжении всего времени работы клиента, подключение к интерфейсу должно производиться в момент запуска клиента, а отключение — в момент останова. Если же клиент обращается к серверу от случая к случаю, подключение и отключение надо осуществлять как можно быстрее — в рамках одной процедуры.

## 9. Графика

*Вид занятия – практическая работа*

*Цель – изучение графического механизма Windows*

*Продолжительность – 2 часа*

В основу идеи графического механизма в ОС Microsoft Windows положена концепция аппаратно-независимого видеовывода. Это означает, что разрабатывающий прикладное программное обеспечение программист может не задумываться над тем, каким образом его приложение отправит данные о своем графическом интерфейсе видеоадаптеру, а тот, в свою очередь, "зажжет" необходимые пиксели на мониторе. Задача программиста заключается лишь в грамотном использовании в коде программы функций прорисовки графических примитивов (точек, линий, кривых, простейших геометрических фигур), заливки областей и вывода текста.

Для того чтобы приобрести возможность нарисовать линию или написать строку текста, программисту достаточно получить доступ к контексту нужного графического устройства и воспользоваться услугами стандартных функций двумерной графики Windows (Graphics Device Interface, GDI). Прямое обращение к функциям GDI для начинающего программиста достаточно затруднительно, поэтому разработчики C++ Builder создали уникальный класс TCanvas, который обеспечивает простой доступ к заданному контексту графического устройства и инкапсулирует наиболее востребованные функции GDI. Более подробно возможности класса TCanvas мы обсудим в следующей главе, но уже сейчас продемонстрируем его возможности. Листинг 9.1 показывает, как мы получили возможность графического вывода на поверхности рабочего стола Windows.

**Листинг 9.1. Доступ к контексту рабочего стола Windows**

```
TCanvas* Canvas=new TCanvas;  
    Canvas->Handle=CreateDC(L"DISPLAY",NULL, NULL, NULL);  
    Canvas->Font->Size=20;  
    Canvas->Font->Color=clRed;  
    Canvas->TextOut(10,10,"Hello, world!");  
  
delete Canvas;
```

В коде примера создается экземпляр класса TCanvas и с помощью функции CreateDC() связывается с графическим контекстом экрана вашего компьютера. Факт успешного завершения эксперимента подтвердит надпись "Hello, world!" в левом верхнем углу рабочего стола.

Контекст графического устройства представляет собой весьма сложную структуру в памяти компьютера (в открытом пользовании нет ни одного официального документа Microsoft, описывающего структуру контекста Windows) и набор функций прикладного программного интерфейса, прямо или косвенно влияющих на процесс графического вывода. Ко всему прочему, контекст хранит сведения о своих текущих настройках — атрибутах контекста. В этой главе мы познакомимся с ключевыми атрибутами: кистью, пером и шрифтом. В объектно-ориентированной библиотеке VCL перечисленные атрибуты представлены в виде самостоятельных объектов — экземпляров класса TBrush, TPen и TFont соответственно. Но сначала мы поговорим о самом важном участнике всех графических операций — цвете.

### **Представление цвета**

Любой из цветов радуги, который вы увидите на экране монитора, получается в результате сложения (в той или иной пропорции) всего-навсего трех базовых цветовых

составляющих: красного, синего и зеленого цветов. Такую цветовую модель называют RGB, по первым буквам английских названий цветов (Red, Green и Blue).

Почему выбраны именно эти цвета? Все объясняется физиологическими особенностями строения человеческого глаза, он воспринимает цвета в диапазоне длин волн примерно 400—700 нм. Излучения с длинами волн 380—470 нм имеют фиолетовый и синий цвета, 480—500 нм — сине-зеленый, 510—560 нм — зеленый, 570—590 нм — желто-оранжевый, 600—760 нм — красный. Восприятие цвета глазом человека осуществляется за счет трех типов цветочувствительных фоторецепторов (колбочек). Существуют всего три типа этих фоторецепторов — колбочки, реагирующие на красный, зеленый или синий цвета. В зависимости от интенсивности той или иной цветовой составляющей соответствующий фоторецептор отправляет сигнал в наш головной мозг, где и формируется результирующая картинка.

Современное программное обеспечение ориентировано на работу с 24- или 32-разрядным представлением цвета (режим True Color). Для описания каждой цветовой составляющей отводится по 8 бит памяти. Если цвет 32-разрядный, то оставшийся четвертый байт хранит значение прозрачности (альфа-канал). Умножьте 4 байта на разрешение вашего рабочего стола, и вы получите весьма большое число. Например, для отображения одной картинке рабочего стола Windows на современном дисплее потребуется  $1920 \times 1080 \times 4$  байта = 8 294 400 байт (8 Мбайт) памяти.

В VCL на обслуживании цвета специализируется тип данных

```
enum TColor {clMin=-0xffffffff-1, clMax=0xffffffff};
```

При назначении цвета можно воспользоваться системными константами, определяющими цвет в соответствии со стилем оформления Windows, назначить цвет вручную или обратиться к услугам макросов GDI. У каждого из перечисленных способов есть свои достоинства и недостатки.

Все системные константы объявлены в модуле Graphics, их имена начинаются с приставки cl, например, константа черного цвета называется clBlack, белого — clWhite. Есть и константы, хранящие настройки выбранной пользователем цветовой схемы. Например, константа clBtnFace помнит значение цвета поверхности кнопки, а константа clCaptionText — цвет, которым выводятся надписи в заголовках окна. По объективным причинам число системных констант весьма ограничено, поэтому они не в состоянии перекрыть весь цветовой спектр.

Для определения цвета из кода программы достаточно вспомнить, что каждая из цветовых составляющих должна находиться в пределах от 0 до 255 (от 0x00 до 0xFF в шестнадцатеричном представлении). Все остальное — дело техники. Например, для создания кисти яркого зеленого цвета достаточно воспользоваться значением 0x00FF00, красного — 0x0000FF, белого — 0xFFFFFFFF и т. п. Как видите, это очень быстрый способ, который позволит определить любой цвет, однако он не вполне удобен для человека. Попробуйте определить, какому цвету соответствует значение 0xF0CAA6? Получилось? Правильный ответ — небесно-голубой...

Третий способ определения значения цвета опирается на четверку макросов (табл. 9.1). Их основное достоинство заключается в простоте использования и наглядности. Кроме того, макросы незаменимы в том случае, когда мы намерены управлять цветами в динамике.

Таблица 9.1. Макросы для работы с цветом

| Макрос                                               | Назначение                                                                |
|------------------------------------------------------|---------------------------------------------------------------------------|
| COLORREF RGB(BYTE byRed, BYTE byGreen, BYTE byBlue); | Макрос формирует результирующий цвет на основе сложения трех составляющих |

|                            |                                                                                                             |
|----------------------------|-------------------------------------------------------------------------------------------------------------|
| BYTE GetRValue(DWORD rgb); | Группа макросов, решающих обратную задачу, они выделяют из смешанного цвета отдельную цветовую составляющую |
| BYTE GetGValue(DWORD rgb); |                                                                                                             |
| BYTE GetBValue(DWORD rgb); |                                                                                                             |

Разместите на форме три компонента TTrackBar, передайте в свойство Max каждого из элементов управления значение 255. Наиболее прозорливый читатель уже наверняка догадался, что, изменяя положение ползунка компонента TTrackBar, мы станем управлять одной из цветовых составляющих. Для того чтобы это реализовать в коде, выберите событие OnChange() любого из элементов управления и напишите в нем строки кода, предложенные в листинге 9.2.

#### Листинг 9.2. Пример работы с макросом RGB()

```
void __fastcall TForm1::tbRedChange(TObject *Sender)
{
    BYTE R=tbRed->Position;
    BYTE G=tbGreen->Position;
    BYTE B=tbBlue->Position;
    Form1->Color=RGB(R,G,B);
}
```

Не забудьте сделать событие OnChange() общим для всех компонентов TTrackBar и запускайте проект на выполнение. Перемещение любого из ползунков приведет к изменению цвета поверхности формы.

## Кисть TBrush

Класс TBrush предназначен для описания кисти, с помощью которой осуществляется закраска указанной программистом области. Перед проведением "малярных работ" следует настроить важнейшие характеристики кисти. Цвет кисти устанавливается в свойстве

**\_\_property** System::TColor Color;

Стиль кисти

**\_\_property** TBrushStyle Style;

отвечает за узор, которым покроется закрашиваемая поверхность. Предусмотрены 8 вариантов стандартных узоров кисти (табл. 9.2).

Таблица 9.2. Константы штриховки кисти TBrushStyle

| Константа    | Описание                                                                                                              |
|--------------|-----------------------------------------------------------------------------------------------------------------------|
| bsSolid      | Сплошная кисть                                                                                                        |
| bsClear      | Пустая кисть                                                                                                          |
| bsHorizontal | Горизонтальная штриховка                                                                                              |
| bsVertical   | Вертикальная штриховка                                                                                                |
| bsFDiagonal  | Штриховка с наклоном 45 градусов, линии направлены из левого верхнего угла в правый нижний угол закрашиваемой области |
| bsBDiagonal  | Штриховка с углом 45 градусов, линии направлены из правого верхнего угла в левый нижний угол                          |
| bsCross      | Штриховка из комбинации горизонтальных и вертикальных линий                                                           |
| bsDiagCross  | Диагональная штриховка из наклонных линий                                                                             |

На внешний вид кисти оказывает влияние цвет заполнения фона. В представленном в листинге 9.3 примере мы закрашиваем два ряда прямоугольных областей, изменяя не только цвет и стиль кисти, но и цвет заполнения с помощью функции Windows API `SetBkColor()`. Верхний ряд выводится белой кистью с черным фоном, нижний — наоборот, черной кистью с белым фоном.

#### Листинг 9.3. Управление стилем кисти

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    Form1->Color=clWhite; //клиентская часть формы окрашена в белый цвет
    TRect Rct=Rect(5,15,55,65);

    for (int i=0; i < 8; i++) {
        Form1->Canvas->Brush->Style=(TBrushStyle) i; //изменяем стиль кисти
        Form1->Canvas->Brush->Color=clWhite;           //цвет кисти белый
        SetBkColor(Form1->Canvas->Handle,clBlack); //устанавливаем черный фон
        Form1->Canvas->FillRect(Rct);               //закрашиваем область
        OffsetRect(Rct,55,0);                       //сдвигаем область вправо
    }

    Rct=Rect(5,70,55,120);
    for (int i=0; i < 8; i++) {

        Form1->Canvas->Brush->Style=(TBrushStyle) i; //изменяем стиль кисти
        Form1->Canvas->Brush->Color=clBlack;         //цвет кисти белый
        SetBkColor(Form1->Canvas->Handle,Form1->Color); //фон формы
        Form1->Canvas->FillRect(Rct);               //закрашиваем область
        OffsetRect(Rct,55,0);                     //сдвигаем область вправо
    }
};
```

Результат работы программы представлен на рис. 9.1.

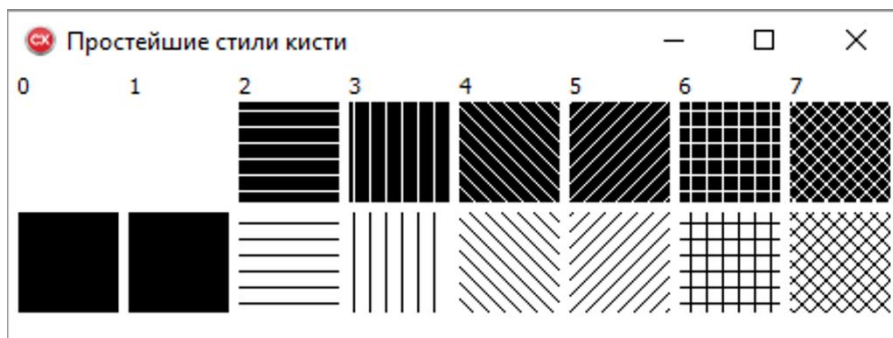


Рис. 9.1. Варианты закрашки областей кистью различного стиля

Программист может создавать кисти с узором, созданным на базе битового образа. Для этого необходимо загрузить файл с картинкой в свойство

```
__property TBitmap* Bitmap;
```

Имейте в виду, что по умолчанию свойство `Bitmap` кисти не инициализируется, поэтому перед загрузкой растрового рисунка необходимо создать экземпляр объекта `TBitmap` и присоединить его к кисти так, как представлено в листинге 9.4.

#### Листинг 9.4. Создание кисти из растровой картинки

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
```

```

Form1->Canvas->Brush->Bitmap=new TBitmap; //инициализация свойства
//загрузка шаблона кисти
Form1->Canvas->Brush->Bitmap->LoadFromFile("picture.bmp");
}
//-----
void __fastcall TForm1::FormPaint(TObject *Sender)
{
Form1->Canvas->FillRect(Form1->ClientRect); //закраска поверхности
}
//-----
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
delete Form1->Canvas->Brush->Bitmap; //очистка ресурса
}

```

Завершая рассказ, посвященный кисти, упомянем свойство, в котором находится дескриптор кисти:

```
__property HBRUSH Handle;
```

Дескриптор может пригодиться если в вашем проекте используются функции GDI.

## Перо TPen

Перо предназначено для определения характеристик линий, рисуемых на поверхности графического устройства. Язык C++ Builder позволяет создавать перья двух базовых разновидностей.

- ❑ *Косметическое перо*, описываемое тремя параметрами: цветом, стилем линии и толщиной. Это перо как нельзя лучше подходит для черчения простых тонких линий и кривых, и его услуг вполне достаточно в 9 из 10 случаев.
- ❑ *Геометрическое перо* развивает возможности своего "коллеги" — оно дополнено еще четырьмя атрибутами: узором, типом наконечника, типом соединения (стыком) и штриховкой. Одна из существенных особенностей геометрического пера в том, что проведенные им линии чувствительны к двумерным аффинным преобразованиям, поэтому при работе приложения в мировых координатах геометрические перья превосходят своих косметических "коллег".

### Замечание

По умолчанию на базе класса TPen формируется простое косметическое перо, для создания более совершенного геометрического пера необходимо обратиться за помощью к функции Windows API ExtCreatePen().

После вызова конструктора по умолчанию создается сплошное черное перо толщиной в 1 пиксел. Если атрибуты пера по умолчанию не соответствуют пожеланиям разработчика, то к его услугам предоставлены три свойства: цвет, толщина и стиль. Цвет пера определяется в свойстве

```
__property TColor Color;
```

Толщина линии в логических единицах:

```
__property int Width;
```

Стиль пера выбирается из перечня стилей

```
__property TPenStyle Style;
```

Предусмотрены 9 стилей пера (табл. 9.3).

Таблица 9.3. Стили пера TPenStyle

| Константа | Описание                     |
|-----------|------------------------------|
| psSolid   | Сплошная линия               |
| psDash    | Линия, состоящая из отрезков |

|               |                                                                                 |
|---------------|---------------------------------------------------------------------------------|
| psDot         | Линия точек                                                                     |
| psDashDot     | Линия из отрезков и точек                                                       |
| psDashDotDot  | Отрезок + две точки                                                             |
| psClear       | Невидимая линия                                                                 |
| psInsideFrame | Сплошная линия                                                                  |
| psUserStyle   | Пользовательский стиль                                                          |
| psAlternate   | Стиль используется только косметическим пером, он определяется набором пикселей |

Первые семь стилей представляют собой базовый набор для обычного косметического пера. Использование перечисленных стилей (за исключением `psSolid` и `psInsideFrame`) имеет смысл только при толщине пера не более 1 пикселя. В противном случае вы получите не штриховую, а обычную сплошную линию. Особых комментариев достойны стили `psClear` и `psInsideFrame`. Стиль `psClear` соответствует пустому (невидимому) перу — рисование отключается. Стиль `psInsideFrame` обычно применяется при выводе простейших геометрических фигур пером шириной более 1 пикселя. Дело в том, что при использовании утолщенного пера (`Style!=psInsideFrame`) границы фигуры расширяются относительно заданных координат. Если вы нарисуете прямоугольник с координатами левого верхнего угла (10, 10) и правого нижнего (100, 100) пером `psSolid` толщиной 10 пикселей, действительный верхний угол окажется в точке (1, 1), а нижний — в точке (109, 109). Границы фигуры расширятся ровно на ширину пера. Установка текущего пера в состояние `psInsideFrame` запретит расширяться границам фигуры — линия будет утолщена вовнутрь. Листинг 9.5 демонстрирует порядок управления стилями пера при рисовании обычного отрезка.

#### Листинг 9.5. Управление стилем пера при рисовании отрезка

```
int y=20;
for (int ps = 0; ps < 9; ps++)
{
    Form1->Canvas->TextOut(10,y,IntToStr(ps));
    Form1->Canvas->Pen->Style=(TPenStyle) ps;
    Form1->Canvas->MoveTo(20,y);
    Form1->Canvas->LineTo(200,y);
    y+=20;
}
```

Результаты выполнения кода отражены на рис. 9.2. Обратите внимание на то, что линии, нарисованные пером с двумя последними стилями (`psUserStyle=7` и `psAlternate=8`), представляют собой обычную сплошную линию. Дело в том, что перечисленные стили предназначены для работы совместно с функцией Windows API `ExtCreatePen()` и в обычном режиме работы никакого смысла не имеют.

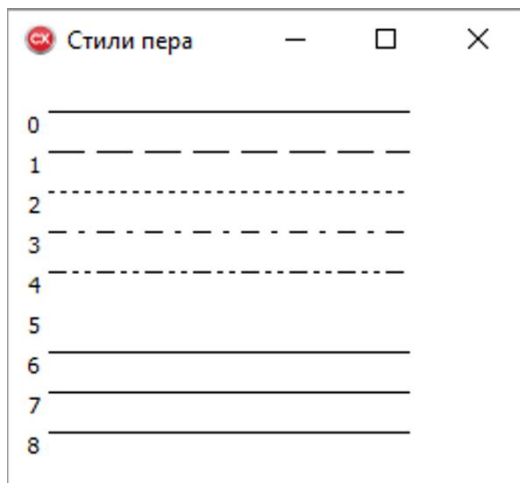


Рис. 9.2. Линии, которые рисует косметическое перо

Для создания геометрического пера следует обратиться к функции Win API

```
HPEN ExtCreatePen(  DWORD dwPenStyle,
                    DWORD dwWidth,
                    CONST LOGBRUSH *lpLb,
                    DWORD dwStyleCount,
                    CONST DWORD *lpStyle );
```

Вид пера определяется в первом параметре функции — геометрическому перу соответствует константа `PS_GEOMETRIC`. Ко всему прочему, в него передается константа, которая описывает наконечник пера, определяющего способ прорисовки окончания линий, и константа, назначающая способ соединения (стык) между линиями. Ширину пера определяет аргумент `dwWidth`. Более подробные сведения о функции вы сможете подчеркнуть в документации MS Windows SDK, а сейчас рассмотрим пример (листинг 9.6) создания косметического пера.

#### Листинг 9.6. Создание геометрического пера

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    LOGBRUSH LB;

    //1 вариант пера
    LB.lbColor=clRed;      //цвет кисти - красный
    LB.lbStyle=BS_SOLID;  //стиль кисти - сплошная кисть
    Form1->Canvas->Pen->Handle=
    ExtCreatePen(PS_GEOMETRIC | PS_ENDCAP_ROUND | PS_SOLID,19,&LB,0,NULL);
    Form1->Canvas->MoveTo(10,20); Form1->Canvas->LineTo(200,20);

    //2 вариант пера
    LB.lbColor= clGreen;   //цвет кисти - зеленый
    LB.lbStyle=BS_HATCHED; //стиль - узор
    LB.lbHatch= HS_CROSS;  //вид узора - крест
    Form1->Canvas->Pen->Handle=
    ExtCreatePen(PS_GEOMETRIC | PS_ENDCAP_SQUARE,19, &LB, 0, NULL);
    Form1->Canvas->MoveTo(10,50); Form1->Canvas->LineTo(200,50);

    //3 вариант пера
    LB.lbColor=clBlue;     //цвет кисти - синий
    LB.lbStyle=BS_HATCHED; //стиль - узор
    LB.lbHatch=HS_DIAGCROSS; //вид узора - диагональные линии
    Form1->Canvas->Pen->Handle=
    ExtCreatePen(PS_GEOMETRIC | PS_ENDCAP_FLAT,19, &LB, 0, NULL);
```

```
Form1->Canvas->MoveTo(20,80); Form1->Canvas->LineTo(350,80);
}
```

В листинге 9.6 последовательно создаются 3 пера. Но теперь для того чтобы передать их в распоряжение холста, нам потребуется не только помощь функции `ExtCreatePen()`, но и свойство, в которое мы направим дескриптор нового пера:

```
__property HPEN Handle;
```

Результаты выполнения листинга отображены на рис. 9.3. Обратите внимание на ряд уникальных особенностей отрезков, нарисованных геометрическими перьями. Во-первых, концы верхнего отрезка закруглены, этого результата мы добились благодаря константе `PS_ENDCAP_ROUND`, переданной в первый параметр функции `ExtCreatePen()`. Во-вторых, благодаря изменению стиля кисти `LogBrush` нижний отрезок заполнен диагональным узором.

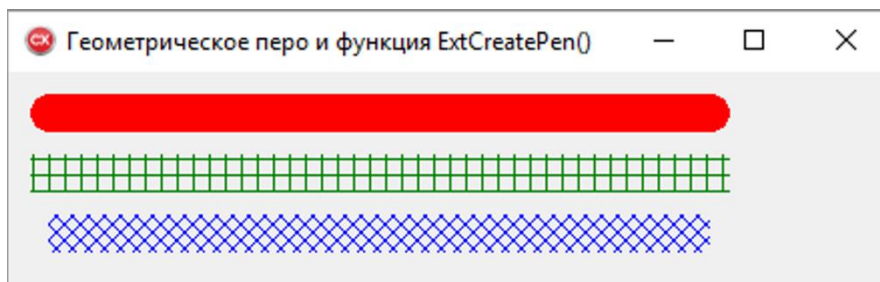


Рис. 9.3. Линии, проведенные геометрическими перьями

Создавая перо, мы имеем возможность задать режим прорисовки линии на холсте

```
__property TPenMode Mode;
enum TPenMode : unsigned char {pmBlack, pmWhite, pmNop, pmNot, pmCopy,
                                pmNotCopy, pmMergePenNot, pmMaskPenNot,
                                pmMergeNotPen, pmMaskNotPen, pmMerge,
                                pmNotMerge, pmMask, pmNotMask, pmXor, pmNotXor};
```

Представление цветной линии в таком случае определяется не только цветом пера, но и цветом области холста (по которой проходит линия) и характером сложения этих цветов. Эта операция называется *бинарной растровой операцией*.

## Шрифт TFont

Все элементы управления из палитры компонентов VCL, способные выводить текст и базовый графический класс `TCanvas`, обладают свойством `Font`. Это свойство предоставляет доступ к очередному важному атрибуту контекста графического устройства — объекту `TFont` (шрифту).

Самое главное свойство шрифта содержится в свойстве

```
__property TFontName Name;
```

Здесь определяется имя шрифта, а точнее имя семейства шрифтов. Семейство — это группа объединенных одним названием (гарнитурой) шрифтов со сходными характеристиками. Так в семейство одного из самых распространенных шрифтов гарнитуры Arial входят: **Arial**, **Arial Bold**, **Arial Bold Italic** и **Arial Italic**. Заметьте, что имена физических шрифтов формируются из названия гарнитуры и начертания шрифта.

Наличие разных семейств шрифтов предполагает существование классификации шрифтов по принадлежности к семейству. Такая классификация предложена в табл. 9.4.

Таблица 9.4. Классификация семейств шрифтов

| Семейство  | Описание                                                                   | Пример             |
|------------|----------------------------------------------------------------------------|--------------------|
| Decorative | Оформительский шрифт                                                       | <b>Old English</b> |
| Dontcare   | Шрифт с произвольными характеристиками                                     |                    |
| Modern     | Шрифт с одинаковой шириной символов                                        | Courier New        |
| Roman      | Пропорциональный шрифт, обладающий переменной шириной символов и засечками | Times New Roman    |
| Script     | Векторный шрифт, имитирующий рукописный текст                              | <i>Script</i>      |
| Swiss      | Пропорциональный шрифт без засечек, с переменной шириной символов          | Arial              |

Описание свойств класса шрифтов предложено в табл. 9.5

Таблица 9.5. Свойства шрифта TFont

| Свойство                                                                                                                                                                              | Описание                                                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>__property</b> TFontCharset Charset;                                                                                                                                               | Номер набора символов шрифта, по умолчанию соответствует DEFAULT_CHARSET, русская кириллица — RUSSIAN_CHARSET                                                                                           |
| <b>__property</b> TColor Color;                                                                                                                                                       | Цвет шрифта                                                                                                                                                                                             |
| <b>__property</b> _di_IChangeNotifier FontAdapter;                                                                                                                                    | Автоматически информирует объекты ActiveX о шрифте                                                                                                                                                      |
| <b>__property</b> HFONT Handle;                                                                                                                                                       | Дескриптор шрифта для работы с функциями Win API                                                                                                                                                        |
| <b>__property</b> int Height;                                                                                                                                                         | Высота шрифта в пикселах                                                                                                                                                                                |
| <b>__property</b> int Size;                                                                                                                                                           | Высота шрифта в пунктах Windows                                                                                                                                                                         |
| <b>__property</b> TFontPitch Pitch;                                                                                                                                                   | Способ установки ширины символов: <ul style="list-style-type: none"> <li>• fpDefault — по умолчанию;</li> <li>• fpFixed — у всех ширина одинакова;</li> <li>• fpVariable — переменная ширина</li> </ul> |
| <b>__property</b> int PixelsPerInch;                                                                                                                                                  | Число точек на дюйм. Используется Windows для установления соответствия между изображениями шрифта на экране и принтере                                                                                 |
| <b>__property</b> int Orientation;                                                                                                                                                    | Угол поворота текстовой строки в градусах относительно оси x                                                                                                                                            |
| <b>__property</b> TFontStyles;<br>typedef System::Set<TFontStyle, TFontStyle::fsBold, TFontStyle::fsStrikeOut> TFontStyles;                                                           | Способ начертания шрифта: жирный, курсив, подчеркнутый и перечеркнутый                                                                                                                                  |
| <b>__property</b> TFontQuality Quality;<br>enum class TFontQuality : unsigned char { fqDefault, fqDraft, fqProof, fqNonAntialiased, fqAntialiased, fqClearType, fqClearTypeNatural }; | Определяет требования к качеству вывода на экран текстовых надписей                                                                                                                                     |

#### Замечание

Размер шрифта можно изменять при помощи свойств Height или Size. Свойства связаны формулой:  

$$\text{Font.Size} = -\text{Font.Height} \times 72 / \text{Font.PixelsPerInch}.$$

Приведенный в листинге 9.7 код демонстрирует способы сбора информации об установленных в системе шрифтах в список ListBox1.

#### Листинг 9.7. Сбор сведений об установленных шрифтах

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    ListBox1->Items->Assign(Screen->Fonts);
    ListBox1->Style=lbOwnerDrawFixed;
}
```

Для того чтобы предоставить пользователю возможность просмотреть, как будет выглядеть текстовая строка, выведенная шрифтом той или иной гарнитуры, воспользуемся листингом 9.8.

#### Листинг 9.8. Просмотр особенностей начертания шрифта

```
void __fastcall TForm1::ListBox1DrawItem(TWinControl *Control, int Index, TRect &Rect,
TOwnerDrawState State)
{
    const String S="Abcdef...";

    TFontName OldFontName=Font->Name;
    ListBox1->Canvas->FillRect(Rect);

    ListBox1->Canvas->Font->Name=ListBox1->Items->Strings[Index];
    ListBox1->Canvas->Font->Color=clNavy;
    TRect R=Rect;
    R.Right=ListBox1->Canvas->TextWidth(S);
    ListBox1->Canvas->TextRect(R, R.Left, R.Top, S);

    ListBox1->Canvas->Font->Name=OldFontName;
    ListBox1->Canvas->Font->Color=clWindowText;
    Rect.Left=90;
    ListBox1->Canvas->TextRect(Rect, Rect.Left+10,
                             Rect.top, ListBox1->Items->Strings[Index]);
}
```

В листинге 9.8 описана реакция компонента на событие `OnDrawItem()`, в котором осуществляется перерисовка каждого из элементов списка. Обратите внимание на то, что кроме гарнитуры шрифта в элементе списка выводится демонстрационная строка, дающая представление об особенностях начертания глифов (рис. 9.4).

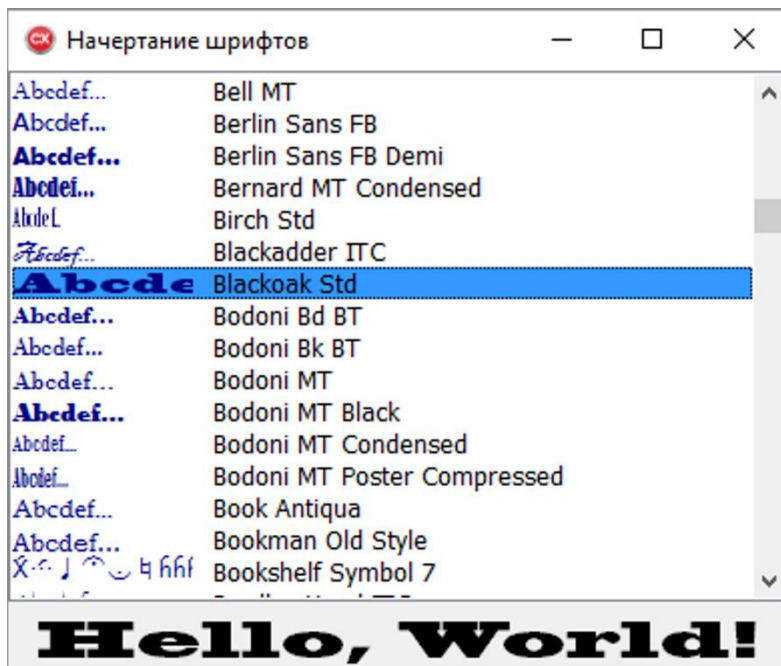


Рис. 9.4. Сбор сведений об установленных шрифтах

## Холст TCanvas в VCL

С целью максимального упрощения работы с графикой и освобождения начинающего программиста от необходимости работать со сложными графическими функциями Windows API в состав VCL включен класс TCanvas. Холст (именно так переводится слово Canvas с английского) описан в модуле Graphics и представляет собой объектно-ориентированное воплощение контекста графического устройства Windows. Судите сами: с одной стороны, объект TCanvas является собственно контекстом графического устройства, а с другой, холст осуществляет все графические операции при посредничестве классических свойств и методов.

С точки зрения цепочки наследования класс TCanvas находится в самой верхушке иерархии VCL, он является прямым потомком класса TPersistent. Несмотря на такую близость к праотцам, TCanvas — уже не абстрактный класс и в состоянии создавать полноценные объекты. Столь полезная способность позволяет разработчикам компонентов внедрять холст в экземпляры собственных объектов, таким образом, наделяя их почти всем спектром возможностей GDI.

### Замечание

Холст присутствует в большинстве визуальных компонентах, доступ к нему осуществляется посредством опубликованного у них свойства Canvas. Кроме того, в VCL имеется специальный объект, готовый исполнять обязанности холста. Это компонент TPaintBox.

Экземпляр класса TCanvas никогда не забывает, что он в проектах C++ Builder представляет интересы реального контекста графического устройства. Дополнительное напоминание об этом можно обнаружить в свойстве, хранящем дескриптор контекста:

```
__property HDC Handle;
```

Дескриптор пригодится нам для обращения к тем функциям Windows API, которым в описании класса TCanvas по тем или иным причинам не нашлось места.

Доступ к основным атрибутам холста обеспечивают свойства:

```
__property TBrush* Brush; //кисть
__property TPen* Pen;     //перо
__property TFont* Font;   //шрифт
```

## Закраска области

Среди всех методов холста, способных закрасить текущей кистью область, наибольшей скоростью обладает функция

```
void FillRect(const TRect &Rect);
```

У метода `FillRect()` всего один недостаток — он умеет работать только с прямоугольными областями, представленными в форме структуры `Rect`.

Более значительными возможностями обладает метод

```
void FloodFill(int X, int Y, TColor Color, TFillStyle FillStyle);
```

Это метод с зачатками интеллекта, он способен залить кистью область произвольной формы. Заливка начинается из точки  $(X, Y)$ . Режим заливки определяется значением параметра `FillStyle`. Если параметр принимает значение `fsSurface`, то заливка осуществляется до тех пор, пока вокруг точки  $(X, Y)$  не останется точек с цветом `Color`. В альтернативном режиме `fsBorder` заливка прекратится при выходе на границу с цветом `Color`.

Пример работы обеих функций закраски областей предложен в листинге 9.9. В рамках события перерисовки формы `OnPaint()` мы осуществляем заливку всей клиентской области формы сплошной кистью белого цвета. Затем в событии `OnMouseDown()`, генерируемом в момент нажатия кнопки мыши, над поверхностью формы иницилируем вызов интеллектуального метода `FloodFill()`, осуществляющего заливку области, которой принадлежит точка с координатами  $(X, Y)$ .

### Листинг 9.9. Закраска области методами `FillRect()` и `FloodFill()`

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    Form1->Canvas->Brush->Style=bsSolid;
    Form1->Canvas->Brush->Color=clWhite;
    Form1->Canvas->FillRect(Form1->ClientRect);

    Form1->Canvas->Pen->Color=clBlack;
    Form1->Canvas->Ellipse(Form1->ClientRect);
    Form1->Canvas->MoveTo(0,0);
    Form1->Canvas->LineTo(Form1->ClientWidth,Form1->ClientHeight);
    Form1->Canvas->MoveTo(Form1->ClientWidth,0);
    Form1->Canvas->LineTo(0,Form1->ClientHeight);

    const String s="Щелкните по закрашиваемой области";
    Form1->Canvas->TextOut(Form1->ClientWidth / 2 - Form1->Canvas->TextWidth(s) / 2,30,s);
}
//-----
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button, TShiftState
Shift, int X, int Y)
{
    Randomize();

    Form1->Canvas->Brush->Style=(TBrushStyle)Random(8);
    Form1->Canvas->Brush->Color=RGB(Random(255),Random(255),Random(255));
    Form1->Canvas->FloodFill(X, Y, clBlack, fsBorder);
}
```

Для большей наглядности перед вызовом метода `FloodFill()` мы присваиваем кисти случайное значение цвета и заливки.

## Градиентная заливка

Для осуществления градиентной заливки прямоугольной области хорошо подходит функция

```
void GradientFillCanvas(const TCanvas* ACanvas,
                       const TColor AStartColor, const TColor AEndColor,
                       const TRect &ARect, const TGradientDirection Direction);
```

Сразу заметим, что это не метод класса TCanvas, а самостоятельная функция, объявление которой вы обнаружите в модуле GraphUtil.hpp. Она требует указать холст ACanvas, на котором будет проводиться операция, стартовое AStartColor и конечное AEndColor значения цвета градиентной заливки, координаты обслуживаемой области ARect. Последний параметр функции Direction определяет направление заливки gdHorizontal (горизонтальное) либо gdVertical (вертикальное).

### Листинг 9.10. Пример градиентной заливки клиентской области формы

```
#include "GraphUtil.hpp";
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    GradientFillCanvas(Application->MainForm->Canvas,
                      clRed, clWhite,
                      Application->MainForm->ClientRect, gdVertical);
}
```

Предложенная в листинге 9.10 строка кода демонстрирует способ градиентной заливки клиентской области главной формы приложения.

## Графические примитивы

Класс TCanvas обладает существенным набором методов для рисования разнообразных графических примитивов. Но самый простейший способ рисования обеспечивает не метод, а свойство

```
__property TColor Pixels[int X][int Y];
```

Благодаря Pixels мы получим доступ к любой точке холста по ее координатам X и Y, как к элементу двумерного массива. Свойство позволяет выяснить, какой цвет установлен в заданной точке холста, и назначить требуемой точке необходимый цвет.

Прямо скажем, что закраска точек — достаточно трудоемкий и медленный процесс, и имеются другие, более расторопные способы графического вывода. Однако в некоторых случаях свойство Pixels может оказаться незаменимым в ситуации, когда требуется построить график сложной функции. Один из примеров подобных функций отражает листинг 9.11, он демонстрирует порядок поточечного вывода на поверхности формы геометрической фигуры гипоциклоиды. Не станем вдаваться в математические тонкости прорисовки кривой со столь мудреным названием, отметим лишь то, что внешний вид гипоциклоиды зависит от значения, возвращаемого свойством Position компонента TrackBar1.

### Листинг 9.11. Применение свойства Pixels для построения гипоциклоиды

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    Form1->Canvas->Brush->Color=clWhite;
    Form1->Canvas->FillRect(Form1->ClientRect);

    //находим центр клиентской области формы
    int offsetX=Form1->ClientWidth / 2;
    int offsetY=Form1->ClientHeight / 2;
```

```

float R1;
//подбираем радиус внешней окружности
if (Form1->ClientWidth>Form1->ClientHeight)
    R1=Form1->ClientHeight / 2-40;
    else
        R1=Form1->ClientWidth / 2 -40;

float R2=R1/TrackBar1->Position;

float K=R1/R2;

float t=0;
while (t<=2*3.1426)
{
    float X=R2*(K-1)*(cos(t)+cos((K-1)*t)/(K-1));
    float Y=R2*(K-1)*(sin(t)-sin((K-1)*t)/(K-1));
    Form1->Canvas->Pixels[X+offsetX][Y+offsetY]=clBlack;
    t=t+0.005;
}
}

```

Результаты наших трудов представлены на рис. 9.5. Изменив положение ползунка, вы придадите гипоциклоиде новую форму.

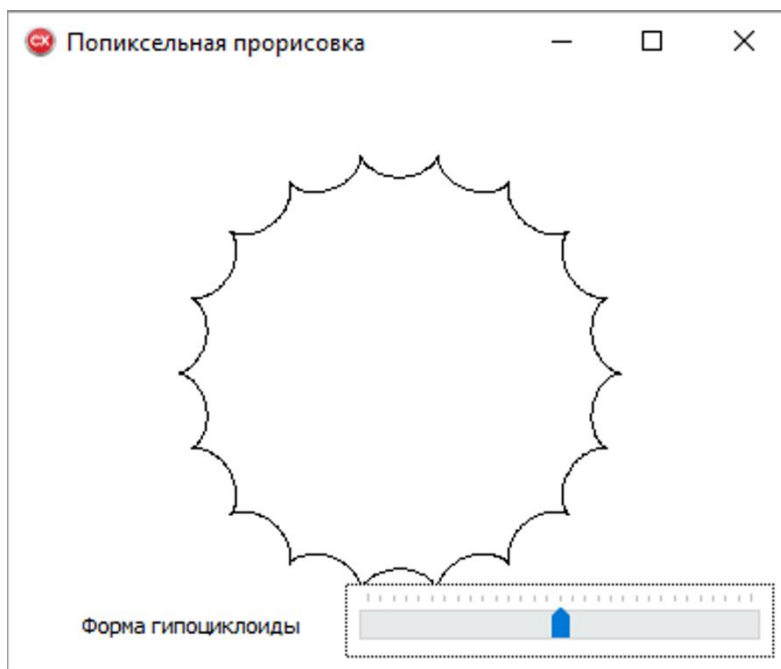


Рис. 9.5. Внешний вид гипоциклоиды

## Линии

Хотя свойство холста `Pixels[]` наиболее универсально и позволяет нарисовать изображение любой степени сложности, но с другой стороны это чрезмерно трудоемкий и медленный процесс, и им не стоит злоупотреблять. Тем более в классе `TCanvas` собран обширный набор методов черчения линий и простейших геометрических фигур.

Для того чтобы вывести простейшую прямую, понадобятся два метода холста:

```

virtual void MoveTo(int X, int Y);
virtual void LineTo(int X, int Y);

```

Функция `MoveTo()` позиционирует перо на начальную точку отрезка. Метод `LineTo()` проводит отрезок из текущей точки к конечной (причем конечная точка не закрашивается) и оставляет перо в этой точке. Если вы вновь вызовете метод `LineTo()`, то вывод линии начнется из последней точки.

Если требуется выяснить текущую позицию пера, то следует познакомиться со свойством `__property TPoint PenPos;`

Свойство доступно не только для чтения, но и для записи, так что это своего рода замена метода `MoveTo()` для перемещения пера по холсту.

Если логика программы предполагает построение нескольких соединенных под разным углом друг к другу отрезков, то стоит обратиться к методу

```
void Polyline(TPoint const *Points, const int Points_Size) = 0;
```

Для изображения ломаной линии необходимо заполнить массив координат точек, через которые пройдет линия. Начальной и конечной точкам ломаной линии соответствуют первый и последний элементы массива `Points`.

Ближайший родственник ломаной линии — многоугольник:

```
void Polygon(TPoint const *Points, const int Points_Size) = 0;
```

Отличие рассмотренного метода от функции `Polyline()` заключается в том, что последняя объявленная в массиве точка соединяется линией с первой и область внутри многоугольника закрасится текущей кистью.

#### *Замечание*

Если вы планируете вывести на экран не весь массив точек, а только его часть, то воспользуйтесь функцией `Slice()`. Второй параметр функции ограничивает верхнюю границу массива: `Form1.Canvas.Polygon(Slice(Arr, 10));`

## Простейшие геометрические фигуры

В перечень простейших геометрических фигур входят: прямоугольник, эллипс и прямоугольник со скругленными углами.

Для рисования прямоугольника проще всего воспользоваться методом

```
virtual void Rectangle(int X1, int Y1, int X2, int Y2) = 0;  
void Rectangle(const TRect &Rect);
```

Функция является перегружаемой и в качестве параметров принимает координаты левого верхнего (`Left`, `Top`) и правого нижнего (`Right`, `Bottom`) углов прямоугольника или координаты, заданные в формате прямоугольной структуры `TRect`.

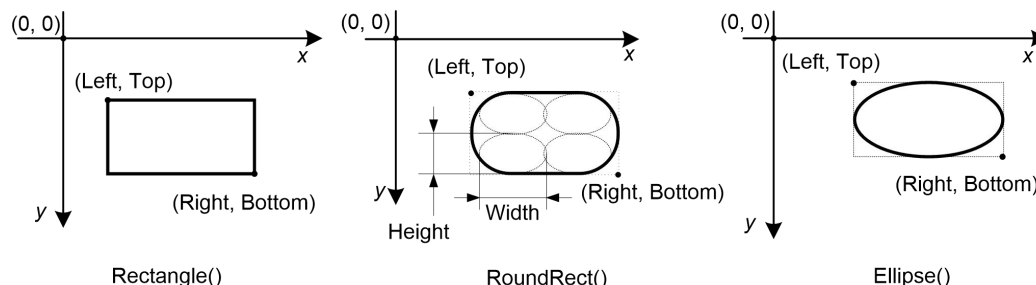


Рис. 9.6. Порядок вывода простейших геометрических фигур

Надо сказать, что большинство методов, рисующих простейшие фигуры, тесно связано с прямоугольником (рис. 9.6). Например: эллипс вписывается в рамки прямоугольника с координатами, аналогичными функции `Rectangle()`.

```
void Ellipse(int X1, int Y1, int X2, int Y2) = 0;  
void Ellipse(const TRect &Rect);
```

Прямоугольник с более эстетичным контуром получится при использовании метода для рисования прямоугольников со скругленными углами:

```
void RoundRect(int X1, int Y1, int X2, int Y2, int X3, int Y3) = 0;  
void RoundRect(const TRect &Rect, int CX, int CY);
```

Первые четыре параметра соответствуют координатам вершин функции `Rectangle()`. Значения `Width` и `Height` определяют степень сглаживания углов, задавая ширину и высоту скругляющего эллипса.

#### *Замечание*

Методы `Rectangle()`, `Ellipse()`, `RoundRect()` и рассмотренные немного ранее `Chord()` и `Pie()` предназначены не только для рисования простейших геометрических фигур, но и для закрашивания ограниченной этими фигурами области текущей кистью.

На этом методы для работы с прямоугольной областью не заканчиваются. Контур толщиной в 1 пиксел для области `Rect` нарисует функция

```
virtual void FrameRect(const TRect &Rect) = 0;
```

Оконтуривание осуществляется не пером, а текущей кистью.

Эффект наличия фокуса ввода на прямоугольнике `Rect` обеспечивает метод

```
virtual void DrawFocusRect(const TRect &Rect) = 0;
```

При повторном вызове функции фокус ввода снимается — изображение приводится к первоначальному виду.

## Дуги

Концепция построения всех дуг основана на идее вывода части эллипса.

```
virtual void Arc(int X1, int Y1, int X2, int Y2,  
                int X3, int Y3, int X4, int Y4) = 0;  
virtual void Chord(int X1, int Y1, int X2, int Y2,  
                  int X3, int Y3, int X4, int Y4) = 0 ;  
virtual void Pie(int X1, int Y1, int X2, int Y2,  
                 int X3, int Y3, int X4, int Y4) = 0;
```

Функция `Arc()` чертит сегмент, `Chord()` — хорду, `Pie()` — сектор эллипса. Суть происходящего отражает рис. 9.7. Все изображения вписываются в эллипс (а точнее, в прямоугольник, содержащий эллипс) с координатами вершин (`Left`, `Top`) и (`Right`, `Bottom`). Логические координаты (`X3`, `Y3`) и (`X4`, `Y4`) описывают воображаемые лучи, исходящие из центра эллипса. Места пересечения лучей с контуром эллипса соединяются дугой. Начальной точкой дуги считается пересечение эллипса с лучом (`X3`, `Y3`), конечной — (`X4`, `Y4`).

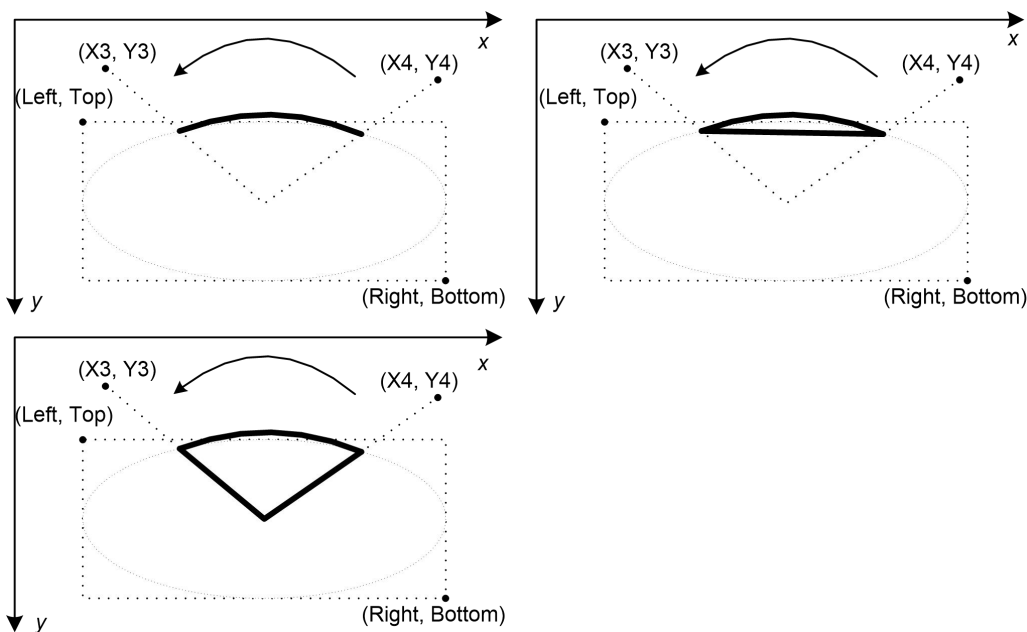


Рис. 9.7. Порядок вывода дуги, хорды и сектора

## Сплайн Безье

Программы инженерной графики, редакторы векторной графики, системы автоматизированного проектирования, издательские пакеты, приложения мультимедиа, игры, одним словом — современное программное обеспечение не в состоянии решить свои задачи за счет ограниченного перечня функций, рисующих прямые линии и дуги. Попробуйте начертить коленчатый вал или контуры истребителя средствами пусть даже очень хорошей функции построения линий! Поэтому в составе методов класса TCanvas имеется пара методов, способных нарисовать кривую линию практически любой степени сложности.

### Примечание

В основу идеи рисования кривой был положен математический аппарат, разработанный на стыке 1950—1960-х годов П. Безье (P. Bezier) и П. де Кастелло (P. De Casteljau). Эти математики (независимо друг от друга) создали эффективный и одновременно простой способ описания кривых линий.

Технически для описания кривой Безье достаточно определить всего четыре точки: две точки концов кривой и две точки управления, они отклоняют кривую от центральной оси. От расположения отклоняющих точек зависит, где и каким образом кривая сменит направление. Для построения кривой Безье (кубического сплайна) воспользуйтесь методом

```
virtual void PolyBezier(TPoint const *Points,
                      const int Points_Size) = 0;
```

Для задания первого сплайна необходимы 4 точки. Каждому последующему сплайну достаточно 3 новых точек, поскольку отправная точка новой кривой есть конечная точка предыдущей. Координатами конечных точек первой кривой служат первый и четвертый элементы массива Points. В качестве точек смещения выступают второй и третий элементы массива. Параметр Points\_Size определяет индекс последнего элемента массива (отсчет ведется от 0). Поведение сплайна, состоящего из 4-х точек, демонстрирует рис. 9.8.

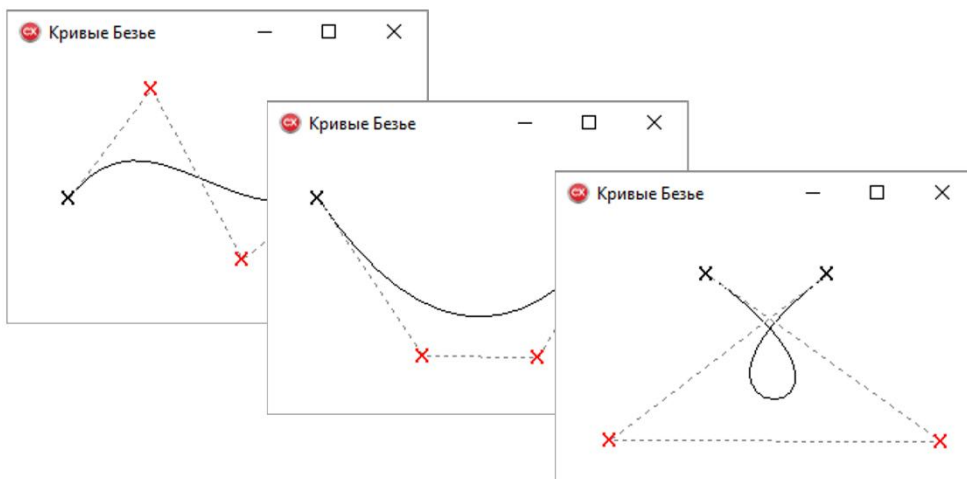


Рис. 9.8. Для определения сплайна Безье достаточно 4 точек

Рисунок 9.8 отражает две отличительные особенности сплайна Безье:

- ☐ кривая всегда находится внутри фигуры с вершинами в определяющих точках кривой;
- ☐ проведенная из конечной в ближайшую отклоняющую точку прямая всегда представляет собой касательную к сплайну в его конечной точке.

В арсенале холста предусмотрен еще один метод, специализирующийся на построении сплайнов:

```
void PolyBezierTo(TPoint const *Points, const int Points_Size) = 0;
```

Единственное отличие от функции `PolyBezier()` заключается в том, что в качестве первой точки кривой Безье используется текущая позиция пера.

## Копирование части холста

Иногда вместо создания собственного графического шедевра хочется воспользоваться чужими трудами. Обычное копирование изображения, находящегося в пределах прямоугольника `Source` из холста `Canvas`, произведет функция

```
void CopyRect(const TRect &Dest, TCanvas* Canvas, const TRect &Source);
```

Результат копирования помещается в прямоугольник `Dest` холста, вызвавшего функцию. Изменяя размер прямоугольника-получателя, можно получить эффект увеличения или уменьшения изображения. Например, для того чтобы "стащить" изображение рабочего стола и разместить его на своем холсте, нам потребуется всего 10 строк кода (листинг 9.12).

### Листинг 9.12. Копирование изображения рабочего стола

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TCanvas* DesktopCanvas=new TCanvas;

    DesktopCanvas->Handle=CreateDC(L"DISPLAY",NULL, NULL, NULL);
    Application->MainForm->Canvas->CopyRect(
        Application->MainForm->ClientRect, DesktopCanvas, Screen->DesktopRect);
    delete DesktopCanvas;
}
```

Щелчок по кнопке `Button1` приводит к тому, что на поверхности формы появляется текущая картина экрана.

Копирование допустимо осуществлять и из растрового изображения, например, скопировать прямоугольник Source из битовой карты Bitmap в прямоугольник Dest поможет функция

```
virtual void BrushCopy(const TRect &Dest, TBitmap* Bitmap,  
                      const TRect &Source, TColor Color);
```

Первая особенность метода заключается в том, что при копировании цвет Color замещается текущим цветом кисти. Знание этого нюанса позволяет нарисовать прозрачную картинку. Для этого достаточно заменить фоновый цвет копируемого рисунка фоновым цветом холста. Вторая особенность метода — возможность копировать только часть исходного изображения. Для этого следует определиться с прямоугольными координатами Source.

Способ копирования изображения на поверхность холста определяется свойством `__property int CopyMode;`

По умолчанию оно установлено в режим копирования "один в один" — `cmSrcCopy`. В этом режиме пиксеты изображения переносятся на холст-получатель без каких-либо изменений. Вместе с этим существует множество альтернативных вариантов вывода картинки на холсте (инвертирование пикселей, сложение пикселей и т. д.). Более подробно о растровых операциях мы поговорим в разделе посвященном сложным приемам программирования графики.

## Сложные графические задачи

В основе графических решений визуальной библиотеки компонентов VCL лежит объектно-ориентированное воплощение контекста графического устройства — холст `TCanvas`. Класс инкапсулирует ключевой набор функций Windows API, обеспечивающих работу с деловой графикой и весьма упрощает работу программиста. Однако класс `TCanvas` далеко не всесилен, в его составе, по тем или иным причинам не нашлось места для достаточно существенного перечня функций GDI, в их числе функции, определяющие режимы отображения Windows, страничные и мировые преобразования, сокращены возможности по работе с регионами отсечения и т. д.

В этой главе мы поговорим о ряде программных решений, позволяющих существенно расширить возможности графического вывода в приложениях C++ Builder. В основе всех этих решений лежит использование функций прикладного интерфейса программирования Windows API в союзе с классом `TCanvas`.

## Растровые операции

Представьте себе все доступные пиксеты экрана в виде двумерного массива  $D$  размером  $x \times y$ . Процесс прорисовки пиксела заключается в указании его координат и передаче по этому адресу нового значения цвета  $P$ , скажем:  $D[x, y] = P$ . Все очень просто и как нельзя лучше подойдет для вывода где-то на экране точки с заданным цветом. Но в реальной жизни все обстоит гораздо сложнее. Например, когда краска с кисти художника ложится на холст, она взаимодействует с поверхностью холста или с другими красками, изменяя результирующий цвет. Почему бы не реализовать такую интересную возможность и в программах C++ Builder?

Раз перед назначением пикселу нового цвета мы намерены учитывать его предыдущее состояние, то предложенная ранее упрощенная формула  $D[x, y] = P$  преобразуется в более серьезную  $D[x, y] = f(D[x, y], P)$ . Здесь  $f$  — функция, описывающая взаимодействие старого  $D[x, y]$  и нового  $P$  цветов пиксела. Подобная функция двух параметров называется *бинарной растровой операцией* (binary Raster Operation, ROP2).

### Замечание

Существуют и более сложные растровые операции, именуемые *тернарными* и *кватернарными*.

Управляя процессом вывода растровой картинки на экран, GDI производит поразрядные булевы операции между вовлеченными в процесс объектами. В случае ROP2 в роли участников растровой операции выступают два действующих лица:

- ❑ пиксел-получатель *D* (destination);
- ❑ пиксел пера *P* (pen).

С только что анонсированными действующими лицами осуществляются следующие логические операции:

- ❑ конъюнкция AND (логическое "И");
- ❑ дизъюнкция OR (логическое "ИЛИ");
- ❑ инверсия NOT (логическое "НЕТ");
- ❑ XOR (логическое исключающее "ИЛИ").

Обсуждая ключевой для графической подсистемы класс TCanvas, мы уже упоминали о существовании свойств холста, определяющих порядок проведения растровых операций. В первую очередь это свойство CopyMode, управляющее процессом переноса на холст исходного графического образа. По умолчанию свойство установлено в состояние cmSrcCopy, что означает прямое копирование пикселей из образа на экран. Кроме того, у пера (класс TPen) предусмотрено свойство Mode, определяющее порядок сложения пикселей пера и пикселей холста.

Существует и другой способ управления растровыми операциями, он основан на обращении к паре функций Win API:

```
int GetROP2(HDC hdc);  
int SetROP2(HDC hdc, int fnDrawMode);
```

Первая из функций информирует программиста о том, какой именно режим ROP2 в текущий момент установлен у графического устройства с дескриптором контекста DC. По умолчанию это режим R2\_COPYPEN, соответствующий обычному переносу цвета пера в пиксел-приемник. Для смены растровой операции предназначена функция SetROP2(). Для этого она вооружена параметром DrawMode, в который и направляется подходящий код ROP2 (табл. 9.6). В случае успешного выполнения функция возвращает код предыдущего состояния ROP2.

Таблица 9.6. Коды основных растровых операций

| Код ROP2         | Операция    | Описание                                                         |
|------------------|-------------|------------------------------------------------------------------|
| R2_BLACK         | D=0         | Пиксел-получатель D устанавливается в черный цвет                |
| R2_COPYPEN       | D=P         | Пиксел D принимает цвет пера P                                   |
| R2_MASKNOTPEN    | D=D && (!P) | Конъюнкция цветов пиксела D и инвертированного пера P            |
| R2_MASKPEN       | D=D && P    | Конъюнкция пера P и приемника D                                  |
| R2_MASKPENNNOT   | D=(!D) && P | Конъюнкция цветов пера P и инвертированного пиксела D            |
| R2_MERGEOTPEN    | D=D    (!P) | Дизъюнкция пиксела D и инвертированного пера P                   |
| R2_MERGEPEN      | D=D    P    | Пиксел D представляет собой дизъюнкцию цветов пера P и пиксела D |
| R2_MERGEOTPENNOT | D=(!D)    P | Дизъюнкция цветов инвертированного пиксела D и пера P            |
| R2_NOP           | D=D         | Цвет пиксела D не изменяется                                     |
| R2_NOT           | D=!D        | Инверсия цвета пиксела D                                         |
| R2_NOTCOPYPEN    | D=!P        | Цвет пиксела D инвертируется по отношению к цвету пера P         |

|                |                     |                                                            |
|----------------|---------------------|------------------------------------------------------------|
| R2_NOTMASKPEN  | $D = !(D \& P)$     | Инверсия операции R2_MASKPEN                               |
| R2_NOTMERGEPEN | $D = !(D    P)$     | Инверсия операции R2_MERGEPEP                              |
| R2_NOTXORPEN   | $D = !(D \wedge P)$ | Инверсия операции R2_XORPEN                                |
| R2_WHITE       | $D = 1$             | Пиксел D устанавливается в белый цвет                      |
| R2_XORPEN      | $D = D \wedge P$    | Пиксел D и перо P объединяются по правилу исключения "ИЛИ" |

Опираясь на возможности операций ROP2, можно получить весьма нетривиальные результаты. Надеюсь, вы не забыли пример из листинга 9.2, в котором мы управляли цветом формы, изменяя положение бегунков трех компонентов TTrackBar (каждый из компонентов отвечал за свою цветовую составляющую RGB). Этот пример можно существенно улучшить за счет бинарной растровой операции.

Итак, в нашем распоряжении три компонента TTrackBar с предельным значением свойства Max, равным 255. Компонент tbRed отвечает за значение красного цвета, tbGreen — зеленого и tbBlue — синего. Обращаемся к событию OnPaint() формы проекта и в его рамках рисуем три эллипса (соответственно красного, зеленого и синего цветов) так, чтобы они частично перекрывали друг друга (листинг 9.13).

#### Листинг 9.13. Пример управления сложением цветов с помощью ROP2

```
void __fastcall TForm1::PaintBox1Paint(TObject *Sender)
{
    //запоминаем текущую бинарную операцию
    int ROP2=GetROP2(PaintBox1->Canvas->Handle);

    PaintBox1->Canvas->Brush->Color=clBlack;
    PaintBox1->Canvas->FillRect(ClientRect);

    SetROP2(PaintBox1->Canvas->Handle,R2_MERGEPEP);
    //красный эллипс
    PaintBox1->Canvas->Brush->Color=RGB(tbRed->Position,0,0);
    PaintBox1->Canvas->Ellipse(1,1,PaintBox1->ClientWidth*2 / 3,PaintBox1->ClientHeight*2 / 3);

    //зеленый эллипс
    PaintBox1->Canvas->Brush->Color=RGB(0,tbGreen->Position,0);
    PaintBox1->Canvas->Ellipse(PaintBox1->ClientWidth / 3,1,PaintBox1->ClientWidth,PaintBox1->ClientHeight*2 / 3);

    //синий эллипс
    PaintBox1->Canvas->Brush->Color=RGB(0,0,tbBlue->Position);
    PaintBox1->Canvas->Ellipse(PaintBox1->ClientWidth / 6,PaintBox1->ClientHeight / 3,PaintBox1->ClientWidth-PaintBox1->ClientWidth / 6,PaintBox1->ClientHeight);

    //восстанавливаем исходную бинарную операцию
    SetROP2(PaintBox1->Canvas->Handle,ROP2);
};
```

Самое важное в примере то, что перед началом прорисовки эллипсов мы устанавливаем растровую операцию холста в состояние R2\_MERGEPEP, что заставит систему осуществлять логическое сложение пиксела холста и пиксела эллипса. В результате дизъюнкции пикселей красного и зеленого цветов мы получаем желтый цвет, сложение зеленого и синего цветов приводит к появлению голубого цвета, и т. д. К сожалению, черно-белая печать книги не позволяет передать все оттенки цветов, поэтому на рис. 9.9 пришлось оставить пояснительные подписи...

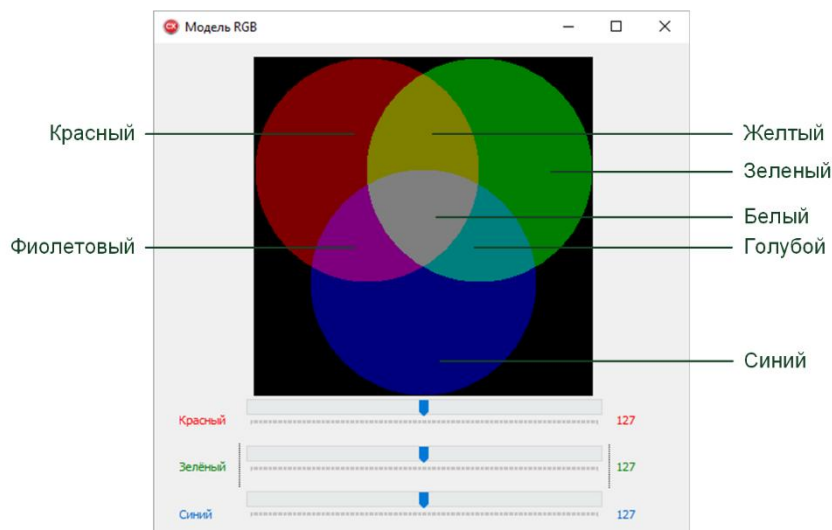


Рис. 9.9. Растровая операция R2\_MERGEPEEN с пикселями красного, зеленого и синего цветов

## Управление прозрачностью

Рассуждая о способе описания 32-битного цвета, мы отметили факт существования специального байта, предназначенного для хранения коэффициента прозрачности. В этом случае, кроме обработки красной, зеленой и синей составляющих, система обязана анализировать состояние альфа-канала и осуществлять растровую операцию с учетом его настоятельных пожеланий. Абсолютно непрозрачному пикселу соответствует значение \$FF (255 в десятичном представлении), а нулевое значение указывает на полную прозрачность.

### Замечание

По сути, альфа-наложение представляет собой очередную разновидность растровой операции, но теперь при формировании результирующего пиксела учету подлежит дополнительный альфа-коэффициент.

Очень гибкими возможностями по управлению прозрачностью обладает объявленная в заголовочном файле `Wingdi.h` функция

```
BOOL AlphaBlend( HDC hdcDest, int nXOriginDest, int nYOriginDest,
                 int nWidthDest, int nHeightDest, HDC hdcSrc, int nXOriginSrc,
                 int nYOriginSrc, int nWidthSrc, int nHeightSrc,
                 BLENDFUNCTION blendFunction);
```

Функция осуществляет перенос пикселей из контекста источника (описываемого дескриптором `dcSrc`) в контекст получателя (дескриптор `dcDest`) с заданной степенью прозрачности. Параметры `XOriginDest`, `YOriginDest`, `WidthDest`, `HeightDest` описывают геометрию приемной области, параметры `XOriginSrc`, `YOriginSrc`, `WidthSrc`, `HeightSrc` характеризуют область-источник. Изюминка функции в последнем параметре, в него направляется структура `TBlendFunction` (листинг 9.14), при посредничестве которой программист определяет порядок взаимодействия между пикселями контекста-источника и контекста-получателя.

### Листинг 9.14. Объявление структуры `BLENDFUNCTION`

```
typedef struct _BLENDFUNCTION {
    BYTE    BlendOp;
    BYTE    BlendFlags;
    BYTE    SourceConstantAlpha;
    BYTE    AlphaFormat;
} BLENDFUNCTION, *PBLENDFUNCTION, *LPBLENDFUNCTION;
```

Если формат альфа-наложения (параметр `AlphaFormat`) равен нулю, то `SourceConstantAlpha` определяет степень прозрачности для всех пикселей раstra. Диапазон ограничен значениями 0—255, где 0 соответствует абсолютной прозрачности, а 255 — полной

непрозрачности. Если поле AlphaFormat структуры TBlendFunction принимает значение AC\_SRC\_ALPHA, то функция приобретает возможность оценивать альфа-коэффициент для каждого пиксела в отдельности.

Допустим, что в нашем распоряжении есть объект BMP: TBitmap, в который загружен растровый рисунок. Этот рисунок нам предстоит отобразить на поверхности формы, постепенно изменяя коэффициент прозрачности. В этом случае стоит воспользоваться исходным кодом из листинга 9.15.

**Листинг 9.15. Управление прозрачностью вывода растрового изображения**

```
TBitmap *B;

void __fastcall TForm1::FormPaint(TObject *Sender)
{
    TBlendFunction BlendFunction;
    const AlphaStep=5; //шаг приращения коэффициента прозрачности

    HDC DC=Form1->Canvas->Handle; //дескриптор холста

    //настройка параметров прозрачности
    BlendFunction.BlendOp=AC_SRC_OVER;
    BlendFunction.BlendFlags=0;
    BlendFunction.SourceConstantAlpha=0;
    BlendFunction.AlphaFormat=0;

    HDC memDC=CreateCompatibleDC(DC); //совместимый контекст в памяти
    SelectObject(memDC,B->Handle); //вносим в память картинку BMP:TBitmap

    int Alpha,Y,X;
    X=Y=Alpha=0;

    while (Alpha<=255) //изменяем коэффициент прозрачности в цикле
    {
        ::AlphaBlend(DC,X,Y,B->Width,B->Height,
                    memDC,0,0,B->Width,B->Height,BlendFunction);

        String s=IntToStr(Alpha);
        TextOut(DC,X,Y+33,s.w_str(),s.Length());

        X+=B->Width;
        if (X>Form1->ClientWidth-B->Width) //вычисляем место вывода
        {
            Y+=B->Height;
            X=0;
        }
        Alpha+=AlphaStep;
        BlendFunction.SourceConstantAlpha=Alpha;
    }

    DeleteDC(memDC); //освобождаем совместимый контекст
}
```

Основная особенность представленного в листинге 9.15 кода в том, что во время перерисовки формы мы заносим растровый образ BMP в искусственно созданный с помощью функции Win API CreateCompatibleDC() контекст графического устройства в памяти. Этот контекст обладает характеристиками, абсолютно идентичными характеристикам

холста формы, поэтому мы его называем *совместимым*. Далее в цикле `while` мы выводим битовый образ на поверхность формы, постепенно уменьшая коэффициент прозрачности, поэтому наш рисунок мало-помалу "проявляется" на белом фоне окна (рис. 9.10).



Рис. 9.10. Вывод битового образа с разными коэффициентами прозрачности

## Системы координат и режимы отображения

До сих пор, осуществляя графический вывод (например, на поверхности формы), мы с вами исходили из предположения, что начало координат располагается в левом верхнем углу формы и координатная ось  $x$  направлена вправо, а координатная ось  $y$  — вниз. При этом мы были уверены, что единственно возможной единицей измерения должен выступать пиксел. Таким образом, обратившись к методу `TextOut(5, 5, 'Hello, world!')`, мы рассчитываем увидеть надпись "Hello, world!" в левом верхнем углу формы, причем левый верхний пиксел текста должен находиться в точке с координатами (5, 5). Но настал тот час, когда мы поймем, что это весьма упрощенное представление о системе координат Windows.

Современные операционные системы Microsoft Windows Vista/7 для определения областей на поверхности вывода используют четыре системы координат: мировые координаты, страничные координаты, координаты устройства и физические координаты.

*Мировые координаты* предназначены для осуществления *двумерных аффинных преобразований*. Для начинающего программиста этот термин наверняка нов, поэтому пока достаточно знать, что аффинные преобразования используются для осуществления разнообразных визуальных эффектов (операций вращения раstra, сдвига, скручивания и т. п.).

*Страничные координаты* соответствуют координатам любой области, обладающей контекстом устройства. Например, в проектах построенных на фундаменте VCL, контекстом описывается поверхность формы (класс `TForm`) и поверхность области для рисования (класс `TPaintBox`). Это система координат, устанавливаемая по умолчанию для всех контекстов во всех версиях Windows, так что, даже не подозревая этого, все свои графические задачи мы до сих пор решали именно в этой системе.

*Координаты устройства* определяются техническими характеристиками аппаратного устройства, с которым работает Windows. Здесь все устроено привычным для нас образом: в качестве единицы измерения всегда применяется пиксел, начало координат находится в левой верхней точке, ось  $x$  направлена слева направо, а ось  $y$  проведена сверху вниз.

*Физические координаты* напрямую используются драйвером графического устройства и явным образом недоступны из обычных программ.

### Замечание

Во время программирования с помощью методов холста TCanvas наши проекты по умолчанию ориентируются на страничные координаты. Для перехода к мировым координатам потребуется помощь функций Win API.

Все четыре координатные системы взаимоувязаны. И, если мы что-то нарисуем в мировых координатах, прежде чем изображение появится на графическом устройстве, Windows последовательно преобразует рисунок из мировых координат в страничные, затем из страничных в координаты устройства и в самом конце (опираясь на поддержку драйвера графического устройства) трансформирует координаты устройства в физические.

Методы класса TCanvas в качестве параметров требуют передачи логических координат. Направляя C++ Builder строку кода `TextOut(5, 5, 'Hello, word!')`, мы информируем систему вывода о том, что рассчитываем увидеть надпись приветствия в точке с координатами (5, 5). Но в этом случае совсем не обязательно речь ведется о пикселах, как мы привыкли до сих пор. Вполне вероятно, что контекст графического устройства в качестве единицы измерения применяет дюймы, миллиметры или другую единицу. Более того, может оказаться, что точка начала отсчета координат появится не в привычном для нас левом верхнем углу рабочей области окна, а где-то в другом месте.

Говоря о графическом интерфейсе пользователя, мы не раз применяли термин "окно". С точки зрения графического вывода окно — это некоторый прямоугольный регион в страничной системе координат. На практике это может быть клиентская область формы TForm стандартного проекта C++ Builder или поверхность любого другого элемента управления. При работе со страничными координатами надо усвоить еще один термин — *область вывода*. В отличие от окна, область вывода — это прямоугольник, но уже в системе координат устройства, например вашего видеоадаптера. Именно в эту систему координат будет спроецирован образ из страничной системы. Управляя начальными координатами и размерами окна, мы получаем возможность определять отображаемую часть изображения. В свою очередь, настраивая область просмотра, мы назначаем место вывода отображаемой части изображения на поверхности устройства.

## Перенос начала координат

Операционная система позволяет изменять местоположение точки начала отсчета координат как области вывода, так и окна. Для этого предусмотрены две функции Windows API: `SetViewportOrgEx()` и `SetWindowOrgEx()`. Для установки начала координат области вывода описываемого контекстом устройства DC предназначен метод:

```
BOOL SetViewportOrgEx(HDC dc, int X, int Y, LPPOINT Point);
```

Здесь: `dc` — дескриптор контекста графического устройства, `X`, `Y` — координаты нового начала координат области вывода в единицах измерения устройства (другими словами, это физические координаты, в качестве единицы измерения которых выступает пиксел). `Point` — указатель на необязательную структуру `TPoint`, в нее сохранится предыдущее значение начала координат.

Попробуем научиться переносить точку начала координат с помощью функции `SetViewportOrgEx()`. Для этого создаете новый проект и в секции `private` объявите переменную `P:TPoint`. После этого нам понадобится поддержка двух событий формы. Событие `OnMouseDown()`, возникающее по щелчку кнопкой мыши по поверхности формы позволит нам запомнить координаты указателя мыши в переменной `P` в момент щелчка. Последствия переноса отразятся во время перерисовки клиентской области формы, для этого задействуется событие `OnPaint()`. Код обработчиков событий представлен в листинге 9.16. После переноса начала координат в новое место мы рисуем две линии, точно соответствующие новым координатным осям `X` и `Y`. Затем, чуть ниже точки начала координат выводим подпись "(0, 0)".

#### Листинг 9.16. Перенос начала координат

```
TPoint P;
...
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button, TShiftState
Shift, int X, int Y)
{
    P=Point(X,Y);
    Form1->Repaint();
}

void __fastcall TForm1::FormPaint(TObject *Sender)
{
    HDC DC=Form1->Canvas->Handle;

    //перенос центра координат проекции окна с помощью SetViewportOrgEx()
    SetViewportOrgEx(DC,P.X,P.Y,NULL);
    Form1->Canvas->MoveTo(-200, 0);
    Form1->Canvas->LineTo(200,0); //рисуем ось x из точки -200 в 200
    Form1->Canvas->MoveTo(0,-200);
    Form1->Canvas->LineTo(0,200); //рисуем ось y из точки -200 в 200
    Form1->Canvas->Font->Color=clBlack;
    Form1->Canvas->TextOut(2,2,"(0,0)"); //подпись начала координат
}
```

При желании точно такой же результат может быть получен с помощью функции

```
BOOL SetViewportOrgEx( HDC dc, int X, int Y, LPPOINT Point);
```

Однако на этот раз функция определяет начало координат вывода окна (а не проекции окна, как ее коллега по цеху `SetViewportOrgEx`). Еще одно отличие этой функции `SetWindowOrgEx()` от `SetViewportOrgEx()` в том, что здесь аргументы *X* и *Y* задаются в логических координатах. Таким образом, метод позволяет манипулировать не только пикселями, но и любыми другими удобными для пользователя единицами измерения, например миллиметрами и дюймами.

## Управление страничными координатами

Логические координаты созданы с целью максимального упрощения нашей работы. А операционная система без вмешательства извне сумеет преобразовать логические координаты в физические координаты принтера или дисплея. Мы уже привыкли к тому, что по умолчанию система координат берет начало в левом верхнем углу рабочей поверхности — в точке с координатами (0, 0). Горизонтальная ось называется осью *x* и проведена из крайней верхней точки поверхности рисования в крайнюю правую точку. Ось *y* берет начало в той же точке (0, 0) и идет вертикально вниз. Теперь пора узнать, что этот самый распространенный режим отображения называется *текстовым* (в терминах Windows ему соответствует идентификатор `MM_TEXT`). Причина названия режима "текстовым" заключается в том, что оси *x* и *y* системы координат режима `MM_TEXT` проведены точно так, как мы читаем текст — слева направо, сверху вниз. Как видите, в данном случае физические единицы устройства — пиксели — полностью совпадают с нашими логическими координатами. Однако в Windows, помимо текстового режима, реализовано еще целых семь режимов отображения (табл. 9.7).

Таблица 9.7. Режимы отображения Windows

| Классификация   | Режим отображения | Логические единицы | Направления осей                      |
|-----------------|-------------------|--------------------|---------------------------------------|
| Текстовый режим | MM_TEXT           | Пиксел             | <i>x</i> — вправо,<br><i>y</i> — вниз |

|                         |                |                                      |                          |
|-------------------------|----------------|--------------------------------------|--------------------------|
| Метрические режимы      | MM_LOMETRIC    | 0,1 мм                               | x — вправо,<br>y — вверх |
|                         | MM_HIMETRIC    | 0,01 мм                              |                          |
|                         | MM_LOENGLISH   | 0,01 дюйма<br>(примерно 0,254 мм)    |                          |
|                         | MM_HIENGLISH   | 0,001 дюйма<br>(примерно 0,0254 мм)  |                          |
|                         | MM_TWIPS       | 1/1440 дюйма<br>(примерно 0,0176 мм) |                          |
| Пользовательские режимы | MM_ISOTROPIC   | Настраиваемые                        | Настраивается            |
|                         | MM_ANISOTROPIC | Настраиваемые                        | Настраивается            |

По сути, метрические режимы изображения — это заранее подготовленные шаблоны, устанавливающие фиксированные протяженности окна и области вывода. Для России наиболее подходят режимы MM\_LOMETRIC и MM\_HIMETRIC, которые нацелены на работу с миллиметрами. Логическая единица грубого режима MM\_LOMETRIC соответствует 0,1 мм. Прецизионный режим MM\_HIMETRIC различает величину 0,01 мм.

Ключевая особенность пользовательских режимов в том, что они способны изменять направления осей абсцисс и ординат. Еще одна положительная черта режимов MM\_ISOTROPIC и MM\_ANISOTROPIC в том, что в отличие от стандартных метрических режимов пользовательские режимы отображения предоставляют программисту право самостоятельно назначать логическую единицу измерения. Тем самым мы получаем дополнительную степень свободы, благодаря которой сможем создавать программное обеспечение, например нацеленное на промышленное черчение или на построение топографических систем с единицами измерения, на порядки отличающимися от фундаментальных дюймов и миллиметров. Различие между двумя пользовательскими режимами раскрывается в их названиях. Изотропный (одинаковый) режим отображения разрешает программисту самостоятельно назначать логическую единицу измерения, но только при условии, что логические единицы измерений осей  $x$  и  $y$  будут эквивалентны друг другу. Напротив, анизотропный (неодинаковый) режим допускает, чтобы логические единицы измерений горизонтальной и вертикальной осей были различными, например: в качестве единиц измерения для оси  $x$  программист может назначить миллиметры, а для  $y$  — километры.

За управление режимом отображения отвечает пара функций Win API:

```
int GetMapMode( HDC dc );
int SetMapMode( HDC dc, int fnMapMode );
```

Первая функция информирует программиста о текущем режиме контекста, а вторая изменяет режим отображения. В случае успешного выполнения функция SetMapMode() возвращает значение предыдущего режима отображения, в противном случае 0.

Переключение режимов отображения с параллельным изменением логической единицы очень часто применяется в разнообразных текстовых и графических редакторах, системах автоматизированного проектирования, картографических приложениях. Одним словом, во всех программных продуктах, ориентированных на стандартные системы измерения. Листинг 9.7 демонстрирует способ начертания сантиметровой линейки, которая может пригодиться, например, для горизонтальной разметки текста в текстовом редакторе.

#### Листинг 9.17. Вывод сантиметровой линейки

```
void __fastcall TForm1::FormPaint( TObject *Sender)
{
```

```

HDC DC=Form1->Canvas->Handle;
SetMapMode(DC,MM_TEXT); //обычный режим отображения

Form1->Canvas->Pen->Color=clGray;
Form1->Canvas->MoveTo(0,Form1->ClientHeight / 2);
Form1->Canvas->LineTo(Form1->ClientRect.Right,Form1->ClientHeight / 2);
Form1->Canvas->Pen->Color=clSilver;
Form1->Canvas->MoveTo(0,1+Form1->ClientHeight / 2);
Form1->Canvas->LineTo(Form1->ClientRect.Right,1+Form1->ClientHeight / 2);

//перенос точки начала координат
SetViewportOrgEx(DC, 0, Form1->ClientHeight / 2, NULL);

SetMapMode(DC,MM_LOMETRIC); //метрический режим отображения
int x=0;
do
{
    Form1->Canvas->Pen->Color=clGray;
    Form1->Canvas->MoveTo(x,-50);
    Form1->Canvas->LineTo(x,0); //риски линейки
    SetBkMode(DC,TRANSPARENT);
    Form1->Canvas->Font->Size=20;
    Form1->Canvas->TextOut(x+5,0,x / 100); //подпись значений
    x+=100;
}
while (x<10000);
}
//--

```

## Мировые координаты и аффинные преобразования

В первых версиях Windows 95/98/ME понятия мировых координат не существовало, поэтому для поддержки программ старого парка даже в современной Windows контекст графического устройства по умолчанию устанавливается в страничную систему координат.

Появление мировых координат в современных версиях Windows предоставило программисту удобный механизм осуществления двумерных преобразований на плоскости. Двумерные преобразования основаны на идее отображения одного множества точек в другое, в перечень возможностей преобразований входит как элементарный перенос изображения с одного места на другое без внесения в него каких-либо искажений, так и сложные процедуры растяжения (сжатия), деформации и вращения. Все операции преобразований опираются на математический аппарат, именуемый *двумерными аффинными преобразованиями*.

### *Замечание*

С практической точки зрения разработчика программного обеспечения аффинное преобразование определяет порядок перехода от прямоугольной декартовой системы координат к назначаемой программистом системе (при необходимости даже косоугольной).

Для выяснения, в какой из систем координат установлен контекст графического устройства, обращаемся к функции Windows API

```
int GetGraphicsMode( HDC dc);
```

информирующей нас о текущей системе координат контекста устройства с дескриптором dc. В случае успешного выполнения функция вернет одну из двух констант: GM\_COMPATIBLE или GM\_ADVANCED. Значение GM\_COMPATIBLE свидетельствует о том, что контекст установлен в страничные координаты, или, как часто говорят, в совместимый графический режим.

Константа `GM_ADVANCED` сигнализирует о том, что система координат контекста настроена на работу в мировых координатах и готова к осуществлению аффинных преобразований.

Для смены системы координат предназначена функция

```
int SetGraphicsMode( HDC dc, int iMode);
```

В первый параметр передается дескриптор контекста графического устройства, во второй — одна из двух упомянутых ранее констант.

Собственно процессом преобразования мировых координат в страничные для контекста устройства `DC` управляет одна-единственная функция GDI:

```
BOOL SetWorldTransform( HDC dc, CONST XFORM *lpXform );
```

Результат работы функции определяется правилами двумерных аффинных преобразований, передаваемых программистом в матрицу аффинных преобразований `XForm`, описываемую структурой:

```
typedef struct _XFORM {
    FLOAT eM11;
    FLOAT eM12;
    FLOAT eM21;
    FLOAT eM22;
    FLOAT eDx;
    FLOAT eDy;
} XFORM, *PXFORM;
```

Каждое поле структуры входит в математические выражения, определяющие алгоритм преобразования точек. В результате этих преобразований исходная точка  $(x, y)$  транслируется в точку с координатами  $(x', y')$ . Наиболее простое назначение у двух последних полей структуры (`eDx` и `eDy`) — сдвигать координаты на указанное число логических единиц вдоль осей абсцисс и ординат. Роль полей `eM11`, `eM12`, `eM21`, `eM22` гораздо значительнее, для ее разъяснения предлагаю обратиться к табл. 9.8.

*Таблица 9.8. Назначение полей `eM11`, `eM12`, `eM21` и `eM22` структуры `TXForm`*

| Вид операции                 | Поля структуры                             |                               |                                 |                                          |
|------------------------------|--------------------------------------------|-------------------------------|---------------------------------|------------------------------------------|
|                              | <code>eM11</code>                          | <code>eM12</code>             | <code>eM21</code>               | <code>eM22</code>                        |
| Тождественное отображение    | 1                                          | 0                             | 0                               | 1                                        |
| Отражение                    | Отражение по горизонтальной оси $-1$       | 0                             | 0                               | Отражение по вертикальной оси $-1$       |
| Поворот на $\Theta$ градусов | Косинус угла $\Theta$                      | Синус угла $\Theta$           | Отрицательный синус $\Theta$    | Косинус угла $\Theta$                    |
| Масштабирование              | Коэффициент масштабирования по горизонтали | 0                             | 0                               | Коэффициент масштабирования по вертикали |
| Поперечная деформация        | 1                                          | Константа сдвига по вертикали | Константа сдвига по горизонтали | 1                                        |

Таким образом, если, например, нам требуется изменить масштаб изображения, то следует передать соответствующие коэффициенты масштабирования в поля `eM11` и `eM22` записи `TXForm`, при этом не забыв записать в остальные поля нулевое значение. Для поворота изображения нам следует лишь пересчитать значения синуса и косинуса для заданного угла и направить значения в поля `eM11`, `eM12`, `eM21`, `eM22`. Если по какой-либо причине мы не

можем получить требуемого результата за счет единственного аффинного преобразования, допускается создать их комбинацию. Специально для этого реализована функция:

```
BOOL CombineTransform( LPXFORM lpxformResult,  CONST XFORM *lpxform1,
                      CONST XFORM *lpxform2  );
```

Здесь: константы `lpxform1` и `lpxform2` — первая и вторая структуры `TXForm`; результат их комбинирования окажется в первом параметре — параметре `lpxformResult`.

Наличие мировых координат существенно упрощает решение ряда графических задач. Например, до того как в Windows появилась поддержка двумерных аффинных преобразований, программисту приходилось изрядно потрудиться над решением даже такой элементарной задачи как поворот отрезка вокруг своей оси на заданное число градусов. А теперь, благодаря мировым координатам, сложность подобных задач уменьшилась на порядок. Для подтверждения этих слов напишем небольшую программу, рисующую обычные стрелочные часы.

Создайте новый проект и разместите на главной форме таймер `TTimer`. С каждым возникновением события `OnTimer` (листинг 9.18) декодируем системное время (функция `DecodeTime`), заполняя заранее подготовленные переменные `Hour`, `Min`, `Sec` и `MSec` (часы, минуты, секунды и миллисекунды соответственно).

**Листинг 9.18. Пересчет текущего времени в углы поворота стрелок часов**

```
double eHour, eMin, eSec; //углы поворотов стрелок
const float pi=3.1426;

//...
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    Word Hour, Min, Sec, MSec;
    //пересчет текущего времени в значения углов поворотов стрелок
    DecodeTime(Time(), Hour, Min, Sec, MSec);
    eSec=Sec*pi/30;
    eMin=Min*pi/30;
    eHour=Hour*pi/6+Min/60*pi/6;

    TRect R=Form1->ClientRect;
    InvalidateRect(Form1->Handle, &R, true);
}
```

На следующем этапе программа конвертирует полученные значения в углы поворота стрелок часов. Например, угол поворота минутной и секундной стрелок (переменные `eMin` и `eSec`) вычисляется следующим образом: текущее значение времени умножается на величину  $\pi/4$ . С углом поворота часовой стрелки немножко сложнее. При определении ее местоположения учитываются не только часы (значение `Hour`), но и минуты (значение `Min`). Переменная `eHour` определяет грубое местоположение стрелки. Например, если системное время — 12 часов ровно, то стрелка укажет на 12-часовую отметку циферблата. Переменная `Min` уточняет степень отклонения стрелки от часовой риски. Если системное время — 12 часов 30 минут, то за счет переменной `Min` к углу поворота стрелки добавится значение  $\pi/12$ . Таким образом, часовая стрелка получит дополнительное смещение от вертикального положения, указывающее, что скоро наступит 13 часов дня.

На рис. 9.11 представлен подробный чертеж будущего циферблата нашего проекта. Обратите внимание на следующее:

- ☐ начала координат перенесены в центр окна;
- ☐ изменено поведение оси `y`, теперь она направлена вверх;

- установлена протяженность окна  $100 \times 100$  логических единиц, это решение позволит безболезненно масштабировать часы при любом изменении размеров формы.

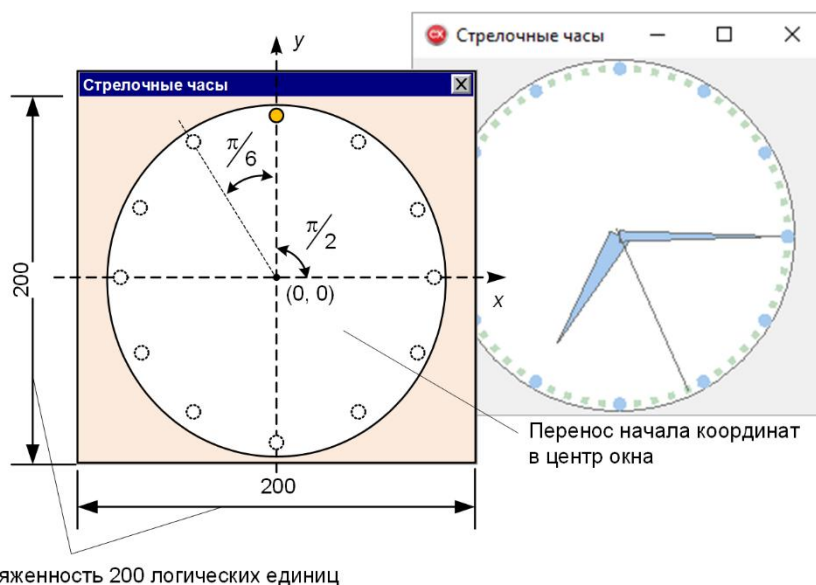


Рис. 9.11. Циферблат стрелочных часов

Листинг 9.19 раскрывает порядок графического вывода часов. Для этих целей мы воспользовались событием `OnPaint()` формы. В первых строках события переводим страничную систему координат в изотропный режим отображения и устанавливаем начало координатных осей в центр окна. В результате ось  $y$  направляется из центра окна на 12-часовую отметку, а ось  $x$  — из центра окна на 3-часовую отметку циферблата. Для вывода на экран минутной и часовой стрелок вызывается функция `Polygon()`, в которую передается массив `Ar`, состоящий из трех точек. С секундной стрелкой еще проще — это обычный отрезок, проведенный парой функций `MoveToEx()` и `LineTo()`. По умолчанию все стрелки "смотрят" на 12-часовую отметку часов. Для того чтобы они повернулись на требуемый угол, воспользуемся аффинными преобразованиями, благо требуемые углы стрелок уже посчитаны в листинге 9.18.

#### Листинг 9.19. Отображение часов в событии `OnPaint()` формы

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    HDC DC=Form1->Canvas->Handle;
    TRect R=Form1->ClientRect;

    SetGraphicsMode(DC, GM_COMPATIBLE);
    SetMapMode(DC, MM_ISOTROPIC);
    //начала координат в центре окна
    SetViewportOrgEx(DC, R.Right / 2, R.Bottom / 2, NULL);
    SetWindowExtEx(DC, 200, 200, NULL); //протяженность 200 x 200 лог. единиц
    SetViewportExtEx(DC, R.Right, R.Bottom, NULL);

    Form1->Canvas->Pen->Color=clGrayText;
    Form1->Canvas->Brush->Color=clWindow;
    Form1->Canvas->Ellipse(-99,-99,99,99); //циферблат

    SetGraphicsMode(DC, GM_ADVANCED);
    TXForm XForm;

    double CS, SN;
```

```

eLabel=0; //рисует 12 часовых меток
while (eLabel<2*pi)
{
    CS=cos(eLabel); SN=sin(eLabel);
    XForm.eM11=CS; XForm.eM12=SN; XForm.eM21=-SN; XForm.eM22=CS;
    SetWorldTransform(DC, &XForm);
    Form1->Canvas->Brush->Color=clSkyBlue;
    Form1->Canvas->Pen->Style=psClear;
    Form1->Canvas->Ellipse(98,4,90,-4);
    eLabel=eLabel+pi/6;
}

TPoint Ar[2];
//часовая стрелка
CS=cos(eHour); SN=sin(eHour);
XForm.eM11=CS; XForm.eM12=SN; XForm.eM21=-SN; XForm.eM22=CS;
SetWorldTransform(DC, &XForm);
Ar[0]=Point(-6,0); Ar[1]=Point(6,0); Ar[2]=Point(0,-70);
Form1->Canvas->Pen->Style=psSolid;
Form1->Canvas->Polygon(Ar,3);
//минутная стрелка
CS=cos(eMin); SN=sin(eMin);
XForm.eM11=CS; XForm.eM12=SN; XForm.eM21=-SN; XForm.eM22=CS;
SetWorldTransform(DC, &XForm);
Ar[0]=Point(-3,0); Ar[1]=Point(3,0); Ar[2]=Point(0,-90);
Form1->Canvas->Polygon(Ar,3);
//Секундная стрелка
CS=cos(eSec); SN=sin(eSec);
XForm.eM11=CS; XForm.eM12=SN; XForm.eM21=-SN; XForm.eM22=CS;
SetWorldTransform(DC, &XForm);
Form1->Canvas->MoveTo(0, 5); Form1->Canvas->LineTo(0,-95);
}

```

Если код повторен безошибочно, то в вашем распоряжении окажутся обещанные стрелочные часы.

## Список использованной литературы

1. Шилдт, Геберт. Полный справочник по C++, 4-е издание.: Пер. с англ. — М.: Издательский дом “Вильямс”, 2010. — 800 с.: ил.
2. Бьерн Страуструп. Язык программирования C++. Специальное издание. Пер. с англ. — М.: Издательство Бином, 2011. — 11365 с.: ил.
3. Лафоре Р. Объектно-ориентированное программирование в C++. Классика Computer Science. 4-е изд. — СПб.: Питер, 2003. — 928 с.:ил.
4. C++ Стандартная библиотека для профессионалов/ Н. Джосьютис. — СПб.: Питер, 2004. — 730 с.:ил.
5. Мюссер, Дэвид Р., Дердж, Жилмер Дж., Сейни, Атул. C++ и STL: справочное руководство, 2-е изд. (серия C++ in Depth): Пер. с англ. — М.: ООО “И.Д. Вильямс”, 2010. — 432 с.: ил. — Парал. тит. англ.
6. Ленн Аммерааль. STL для программистов на C++. Пер. с англ./ Леен Аммерааль — М.: ДМК, 1999 — 240 с., ил.
7. Осипов Д. Л. Delphi XE2. — СПб.: БХВ-Петербург, 2012. — 912 с.:ил.
8. Осипов Д. Л. Delphi. Программирование для Windows, OS X, iOS и Android. — СПб.: БХВ-Петербург, 2014. — 464 с.: ил. — (Профессиональное программирование)
9. Осипов Д. Л. C++ Builder. Основы программирования. —[б.м.]: Издательские решения, 2016. — 790 с.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
Федеральное государственное автономное образовательное учреждение  
высшего образования  
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

## **РАЗРАБОТКА ЗАЩИЩЕННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

### **Методические рекомендации к самостоятельной работе студентов**

Направление подготовки 10.05.01 «Компьютерная безопасность»

Направленность (профиль): «Информационно-аналитическая и техническая  
экспертиза компьютерных систем»

Квалификация выпускника – специалист

Год начала обучения – 2026 г.

Форма обучения – очная

Ставрополь  
2026

## СОДЕРЖАНИЕ

|                                                        |            |
|--------------------------------------------------------|------------|
| <b>Пояснительная записка .....</b>                     | <b>175</b> |
| <b>План-график выполнения СРС по дисциплине .....</b>  | <b>175</b> |
| <b>Методические рекомендации к СРС .....</b>           | <b>175</b> |
| Форма контроля СРС .....                               | 178        |
| Задания для СРС .....                                  | 178        |
| Примерные темы рефератов .....                         | 178        |
| Примерные темы докладов .....                          | 178        |
| Примерные темы индивидуальных творческих заданий ..... | 178        |
| Рекомендуемая литература и интернет-ресурсы .....      | 179        |

## 1. Пояснительная записка

В ходе изучения дисциплины студенты должны выполнить домашнее задание, способствующее приобретению навыков разработки и анализа алгоритмов. Студенту предлагается смоделировать предметную область и спроектировать приложение. По итогам выполнения работы преподавателю предоставляется набор листингов, разработанная студентом программа и оформленный отчет с полученными результатами.

Текущий контроль усвоения знаний осуществляется путем прохождения 6 контрольных точек, подготовки и сдачи отчетов по итогам выполнения лабораторных работ, проверки выполнения домашнего задания, опросов на практических занятиях.

## 2. План-график выполнения СРС по дисциплине

| № контрольной точки | Вид самостоятельной работы         | Форма контроля   | Зачетные единицы (часы) |
|---------------------|------------------------------------|------------------|-------------------------|
| <b>9 семестр</b>    |                                    |                  |                         |
| 5,6                 | Подготовка к лабораторным занятиям | Защита лаб.работ | 40                      |
| 5                   | Самостоятельное изучение темы      | Тестирование     | 16                      |
| 6                   | Индивидуальное творческое задание  | Проверка         | 16                      |
|                     | <b>Итого за 7 семестр</b>          |                  | <b>72</b>               |
|                     |                                    |                  |                         |
|                     | <b>Итого</b>                       |                  | <b>72</b>               |

## 3. Методические рекомендации к СРС

### Тема 1. Уязвимости программного обеспечения

Цели и задачи курса. Место дисциплины в учебном процессе. Методические рекомендации по изучению курса. Обзор литературы.

Концепция типа данных. Представление структур в памяти. Элементарные и простые структуры данных. Сложные структуры данных. Статические, полустатические и динамические структуры. Элементарные структуры данных (Строительные блоки, массивы, связанные списки, обработка простых списков, распределение памяти под списки, строки, составные структуры данных).

Абстрактные типы данных (Абстрактные объекты и коллекции объектов, АД для стека магазинного типа, примеры программ-клиентов, использующих АД стека, реализации АД стека, создание нового АД, очереди FIFO и обобщенные очереди, повторяющиеся и индексные элементы, АД первого класса).

Понятие последовательности, последовательное представление данных. Операции над файлами. Базовые алгоритмы работы с файлами

### Тема 2. Обработка исключений

Понятие алгоритма с точки зрения методов программирования. Требования к алгоритму. Понятия модели вычислений и оценки сложности алгоритмов. Критерии оценки сложности алгоритмов. Методы оценки алгоритмов. Рост функций. О-нотация.

Отладка и тестирование программного обеспечения. Модульное тестирование, тестирование чёрного ящика. Создание проектной документации. Особенности сопровождения разработанного программного обеспечения. Принципы тестирования. Тестирование спецификаций. Тестирование на основании анализа текста программы. Тестовые данные. Этапы тестирования. Примеры тестирования циклов. Тестирование программ численных расчетов. Тестирование модулей. Средства тестирования Отладка.

### **Тема 3. Переполнение стека**

Постановка задачи сортировки. Основные понятия и стратегии сортировки. Элементарные методы сортировки. Сортировка выбором. Сортировка вставками. Пузырьковая сортировка. Характеристики производительности элементарных методов сортировки. Сортировка методом Шелла. Сортировка других типов данных. Сортировка по индексам и указателям. Сортировка связанных списков. Метод распределяющего подсчета.

Быстрая сортировка. Базовый алгоритм. Характеристики производительности быстрой сортировки. Размер стека. Подфайлы небольших размеров. Метод разделения с вычислением медианы из трех элементов. Дублированные ключи. Строки и векторы. Выборка.

Очереди по приоритетам и пирамидальная сортировка. Элементарные реализации. Пирамидальная структура данных. Алгоритмы для сортирующих деревьев. Пирамидальная сортировка. Абстрактный тип данных очереди по приоритетам. Очередь по приоритетам для индексных элементов. Биномиальные очереди

Сортировка последовательных файлов. Основные понятия внешней сортировки. Двухпутевое слияние. Абстрактное обменное слияние. Нисходящая сортировка слиянием. Усовершенствования базового алгоритма. Восходящая сортировка слиянием. Производительность сортировки слиянием. Реализация сортировки слиянием, ориентированной на связанные списки.

Основные понятия поиска. Алгоритмы исчерпывающего поиска. Поиск в последовательно организованном файле. Поиск в деревьях. Оптимальные деревья двоичного поиска. Сбалансированные деревья.

### **Тема 4. Внедряемый код**

Нелинейные структуры данных, графы и деревья. Представление графа в виде матрицы смежности и виде списка смежных вершин.

Поиск на графе. Поиск в глубину. Поиск в ширину. Отделимость и бисвязность. Алгоритмы поиска связанных и двусвязных компонент в неориентированных графах. Алгоритм поиска сильносвязных компонент в ориентированных графах. Обобщённый поиск в графах.

Алгоритмы нахождения остовного дерева минимального веса. Алгоритмы Прима и поиск по приоритету. Алгоритмы Крускала. Алгоритм Борувки. Алгоритмы построения дерева MST.

Алгоритмы определения кратчайших расстояний между вершинами графа, нахождения транзитивного замыкания. Алгоритм Дейкстры. Кратчайшие пути между всеми парами. Кратчайшие пути в ациклических сетях. Эвклидовы сети. Сведение.

### **Тема 5. Дефекты форматных строк**

Моделирование равномерно распределенных случайных величин. Методы моделирования дискретных и непрерывных случайных величин.

Детерминированные ГПСЧ. ГПСЧ с источником энтропии или ГСЧ. ГПСЧ в криптографии. Аппаратные ГПСЧ.

Тесты DIEHARD. Метод Монте-Карло. Случайное простое число.

Алгоритмы порождения перестановок в лексикографическом порядке, циклическим сдвигом и в порядке минимального изменения. Коды Грея.

### **Тема 6. Переполнение кучи**

Избыточность данных. Метод Running. Метод Huffman. Метод LZW.

Вопросы построения хеш-функций и хеш-таблиц. Недостатки таблиц с прямой адресацией. Раздельное связывание. Линейное зондирование. Двойное хеширование. Динамические хеш-таблицы.

Способы представления данных о цвете и цветовые модели. Алгоритмы вывода простейших графических примитивов. Пошаговая выборка. Способы пространственного разделения. Алгоритмы сканирования на плоскости. Удаление невидимых поверхностей. Двумерные аффинные преобразования.

### **Тема 7. Внедряемый код для Windows**

Этапы развития технологии программирования. Жизненный цикл и этапы разработки программного обеспечения (ПО). Оценка качества процессов создания ПО. Понятие защищённого ПО. Концепция безопасности SD3. Модули и их свойства.

Особенности создания программного продукта (принципы работы с требованиями к программному обеспечению, проблематика проектирования, оценка стоимости ошибок, управление требованиями, последовательность работы с требованиями, анализ проблемы, преграды на пути выявления требований)

Оценка качества процессов создания программного обеспечения (Серия стандартов ISO 9000, CMM, процесс сертификации программ на базе информации об их использовании)

Жизненный цикл программы (понятие технологии разработки программы, основа разработки программного обеспечения, модели жизненного цикла, Rational Objectory Process — модель жизненного цикла, жизненный цикл UML (Rational Objectory Process), специфицирование и планирование, процесс разработки, выпуск продукта и механизмы обратной связи)

### **Тема 8. Проектирование ПО при структурном подходе**

Средства описания структурных алгоритмов. Классификация программных продуктов по функциональному признаку. Основные эксплуатационные требования к программным продуктам. Подготовка к проектированию ПО.

Проектирование программного обеспечения при структурном подходе (структурная схема разрабатываемого программного обеспечения, функциональная схема, метод пошаговой детализации при составлении алгоритмов, структурные карты Константайна, структурные карты Джексона, CASE-технологии, ускорение разработки программного обеспечения, методология RAD).

### **Тема 9. Обход фильтров**

Концепция объектно-ориентированного программирования. Классы и объекты. Проектирование программного обеспечения при объектном подходе (разработка структуры программного обеспечения при объектном подходе, диаграммы кооперации)

Технология СОМ. Интерфейсы и СОМ-объекты. Особенности проектирования клиент-серверного и распределённого ПО. Параллельная обработка данных. Перспективные возможности современных языков программирования по созданию высокотехнологичного ПО. Совершенствование действующего ПО, рефакторинг кода. Управление командой разработчиков. Роль пользователя в жизненном цикле разработки ПО. Стоимость программного продукта.

## Форма контроля СРС

Качество усвоения знаний студентами осуществляется в форме тестирования, индивидуального собеседования, проверки доклада, реферата и проверки индивидуального творческого задания.

## Задания для СРС

### Примерные темы рефератов

1. Понятие структуры данных. Стеки, деки, очереди, линейные списки и деревья.
2. Алгоритмы сортировки сравнениями. Вывод нижней оценки для трудоемкости работы алгоритмов данного типа.
3. Алгоритм сортировки вставками и вывод оценки его трудоемкости.
4. Обменная сортировка. Алгоритм быстрой сортировки.
5. Сортировка выбором. Турнирная и пирамидальная сортировки.
6. Сортировки слиянием и распределяющие сортировки.
7. Алгоритмы исчерпывающего поиска.
8. Алгоритмы поиска в последовательно организованных файлах.
9. Алгоритмы поиска в деревьях.
10. Хеширование и способы разрешения коллизий.
11. Способы задания графа. Остовное дерево. Алгоритм поиска в глубину.
12. Алгоритм поиска сильно связанных компонент.
13. Нахождение двусвязных компонент.
14. Алгоритм построения минимального остовного дерева.
15. Моделирование равномерно распределенных случайных величин.
16. Алгоритмы порождения перестановок в лексикографическом порядке и циклическим сдвигом.

### Примерные темы докладов

1. Обменная сортировка со слиянием алгоритм Бэтчера.
2. Лексикографическая сортировка.
3. Внешняя сортировка.
4. Алгоритмы поиска с возвращениями.
5. Индексно-последовательный поиск.
6. Построение оптимальных бинарных деревьев поиска.
7. Алгоритм сортировки в дерево.
8. Балансировка деревьев по высоте.
9. Балансировка деревьев по весу.
10. Цифровой поиск.
11. Алгоритм поиска в ширину.
12. Алгоритм нахождения кратчайшего пути.
13. Алгоритм транзитивного замыкания.
14. Алгоритм нахождения кратчайших расстояний от источника до всех остальных вершин.
15. Моделирование равномерно распределенных дискретных случайных величин.
16. Моделирование равномерно распределенных дискретных случайных величин.
17. Алгоритмы порождения перестановок в порядке минимального изменения.
18. Графические алгоритмы заливки замкнутых областей.

### Примерные темы индивидуальных творческих заданий

1. Напишите программу, реализующую один из алгоритмов сортировки.

2. Напишите программу, реализующую один из алгоритмов поиска.
3. Получить эмпирические оценки трудоемкости алгоритма сортировки или поиска.
4. Произвести сравнительный анализ трудоемкости работы нескольких алгоритмов.
5. Напишите программу, реализующую алгоритм поиска сильно связанных компонент в графе.
6. Напишите программу, реализующую алгоритм поиска двусвязных компонент в графе.
7. Напишите программу, реализующую алгоритм порождения перестановок в лексикографическом порядке.
8. Напишите программу, реализующую алгоритм порождения перестановок циклическим сдвигом.

## **Рекомендуемая литература и интернет-ресурсы**

### Основная литература:

1. Галатенко В.А. Стандарты информационной безопасности: Курс лекций: Для студентов вузов, обучающихся по спец. 351400 "Приклад. информатика" / В.А. Галатенко; Под ред. В.Б. Бетелина; Интернет-Ун-т Информ. Технологий. - М.: Интернет-Университет Информационных Технологий, 2004. – 328 с. – (Основы информационных технологий). – Библиогр.: с. 315. – ISBN 5-9556-0007-8
2. Ярочкин В.И. Информационная безопасность: учебник / В.И. Ярочкин. – 5-е изд. – М.: Академический Проект, 2008. – 544 с.: ил. – (Учебник для вузов) (Gaudeamus). – На учебнике гриф: Рек. МО. – Прил.: с. 334-533. – Библиогр.: с. 534-541. – ISBN 978-5-8291-0987-5

### Дополнительная литература:

1. Козиол Дж., и др. Искусство взлома и защиты систем. – СПб. Питер, 2006. – 416 с.
2. Казарин О.В. Теория и практика защиты программ. – М. 2004. 450 с.
3. Касперски, К. Техника отладки программ без исходных текстов: учеб. пособие / К. Касперски. – СПб.: БХВ-Петербург, 2005. – 832 с.: ил. – (PRO. Профессиональное программирование). – Прил.: с. 777-814. – ISBN 5-94157-229-8
4. Искусство взлома и защиты систем: учеб. пособие / [Дж. Козиол, Д. Личфилд, Д. Эйтэл и др.]; пер. с англ. Е. Матвеева. – СПб.: Питер, 2006. – 416 с.: ил. – ISBN 5-469-01233-6
- 8.2. Перечень учебно-методического обеспечения самостоятельной работы обучающихся по дисциплине (модулю)
- 8.3. Перечень ресурсов информационно-телекоммуникационной сети «Интернет», необходимых для освоения дисциплины (модуля)
1. <http://catalog.ncfu.ru/catalog/ncfu> – Электронный каталог научной библиотеки СКФУ.

2.

### Интернет-ресурсы:

1. [www.microsoft.com](http://www.microsoft.com) – сайт компании Microsoft
2. [www.embarcadero.com](http://www.embarcadero.com) – сайт компании Embarcadero
3. [www.qt.io](http://www.qt.io) – сайт компании Qt