HIGH TECH

ДМИТРИЙ ОСИПОВ

ГРАФИКА В ПРОЕКТАХ DELPHI

СИМВОЛ®

HIGH TECH

Графика в проектах Delphi

Дмитрий Осипов

Санкт-Петербург — Москва
2008

Серия «High Tech»

Дмитрий Осипов

Графика в проектах Delphi

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Ю. Бочина*
Художник	*В. Гренда*
Корректор	*Л. Минина*
Верстка	*Д. Орлова*

Осипов Д.

Графика в проектах Delphi. – СПб: Символ-Плюс, 2008. – 648 с.: цв. ил.

ISBN-13: 978-5-93286-134-9
ISBN-10: 5-93286-134-7

В книге Дмитрия Осипова «Графика в проектах Delphi» представлен уникальный материал, посвященный программированию деловой графики для современных версий Windows. Рассмотрены графический механизм системы, функции прикладного интерфейса программирования GDI (Graphics Device Interface), методы работы с графикой средствами визуальной библиотеки Delphi и тонкости современной графической библиотеки Windows GDI+. Обсуждаются особенности управления цветом и вывода текста, рисование примитивов, страничные и мировые преобразования, форматы растровых и векторных рисунков, организация работы с печатающим устройством, обработка метаданных в современной цифровой фотографии и приемы улучшения качества изображений, цветовая коррекция и многое другое, без чего нельзя создать интерфейс современного программного продукта.

Книгу отличает глубина и ясность изложения материала, поэтому она будет полезна как начинающему программисту, так и профессионалу, который сможет использовать ее как справочник по функциям и методам среды разработки Delphi.

ISBN–13: 978–5–93286–134–9
ISBN–10: 5–93286–134–7

© Дмитрий Осипов, 2008
© Издательство Символ-Плюс, 2008

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 29.08.2008. Формат 70×100^{1}/$_{16}$. Печать офсетная.
Объем 40,5 печ. л. Тираж 2000 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Введение

Говорят, что когда в начале 70-х годов XX века Николауса Вирта попросили дать определение программы, он предложил краткую и изящную формулировку:

программа = структуры + алгоритмы

Интересно, что бы ответил разработчик языка программирования Pascal на этот же вопрос, если бы он был задан сегодня? Не берусь говорить от имени этого блестящего ученого, профессора Цюрихского университета, обладателя высшей награды в области информатики – премии им. Тьюринга. Со времен создания языка Pascal минуло четыре десятилетия. За это время электронные технологии достигли невиданных для тех времен высот. Сегодня в любой квартире или офисе мы обнаружим компьютер с процессором, производительность которого на порядки превосходит возможности самых передовых ЭВМ 1970-х годов. Вместе с развитием электроники совершенствовались и операционные системы (ОС) – от простейших DOS до современных многозадачных систем. Одним из значимых этапов в развитии операционных систем стало появление у них доброжелательных графических оболочек. Первопроходцем в этой области считается Apple Computer с разработанной в январе 1983 года ОС «Liza». Ровно год спустя Apple анонсировала знаменитый Macintosh. Первая же версия Windows (в те времена еще не столь широко известной компании Microsoft) увидела свет только в 1985 году.

Полагаю, что сегодня в определение современной программы кроме структур и алгоритмов необходимо добавить третий ингредиент – графический интерфейс. В подтверждение этих слов можно привести небывалый успех Windows. Во многом благодаря продвинутым графическим возможностям вышедшая в 1990 году операционная среда Windows 3.0, а несколько позднее (в 1995 г.) – полноценная операционная система Windows 95 практически полностью захватили нишу домашних и офисных систем для компьютеров, совместимых с IBM PC. Причем со сцены были сметены не только невзрачные DOS различных версий, но и весьма удачные ОС, например OS/2 Warp фирмы IBM. На сегодняшний день можно говорить если не о тотальном господстве Windows (существует общепризнанная система Unix, ряд энтузиастов работает в Linux, QNX, FreeBSD), то, по крайней мере, о существенном распространении современных версий операционных систем Microsoft (Windows 2000, Windows XP, Windows 2003, Windows XP x64 и сверхновой Windows Vista).

Настоящая книга посвящена программированию двумерной графики для 32- и 64-разрядных версий Microsoft® Windows® в среде Delphi. Подавляющее большинство приведенных примеров сохранят работоспособность практически во всех версиях среды, начиная с Borland Delphi 2.0 и заканчивая современной Delphi от CodeGear. Ареал нашего исследования отображен на рис. 1.

Что такое Windows GDI (Graphics Device Interface)? Это набор стандартных функций двухмерной графики, которые обеспечивают приложения возможностью общаться с любым установленным в Windows графическим устройством. Прикладной программе совсем не обязательно владеть всеми техническими тонкостями эксплуатации принтера или монитора! Библиотека GDI безукоризненно выполнит обязанности посредника между запросами приложения и драйверами графических устройств.

Группа функций Windows GDI – это существенный подкласс функций, входящих в состав прикладного интерфейса программирования Windows API. Основная часть этих функций сосредоточена в системной библиотеке gdi32.dll. Из этой библиотеки они экспортируются модулем windows – доминирующим элементом из состава библиотеки времени выполнения (Run Time Library, RTL) среды программирования Delphi. За ним следует царство визуальной библиотеки компонентов (Visual Components Library, VCL). Обращения к функциям GDI инкапсулируются в графических классах Delphi, ключевые классы объединены в модуле graphics.

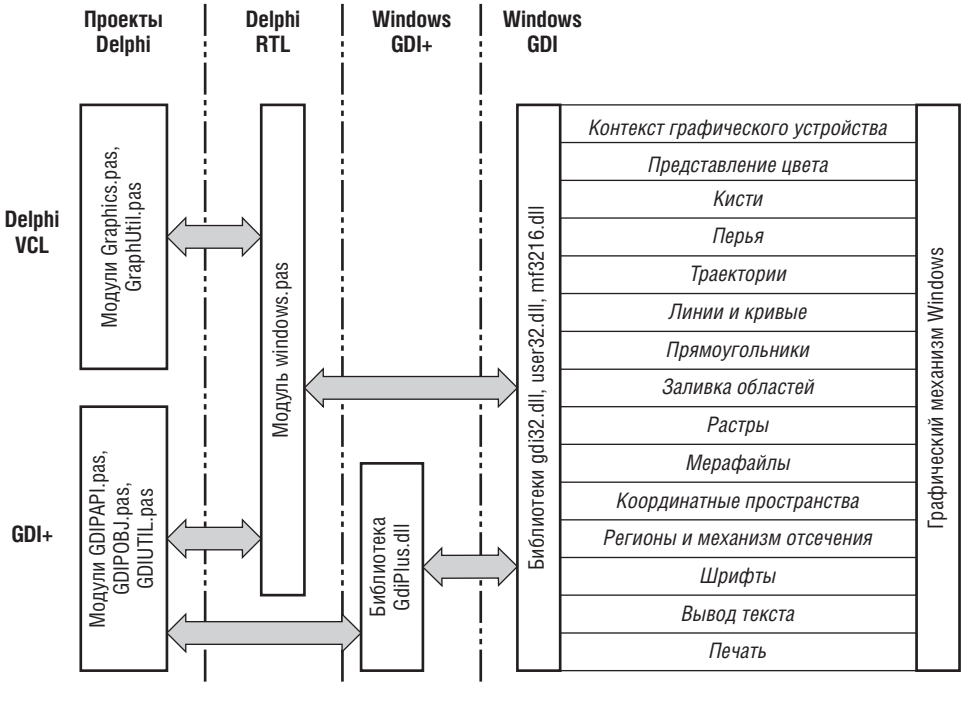


Рис. 1. Взаимодействие Delphi и графических библиотек Windows

Технология GDI+ представляет собой дальнейшее развитие графического механизма операционной системы Windows. Это объектно-ориентированное воплощение библиотеки GDI, значительно расширяющее ее возможности, реализованное в виде новой библиотеки GdiPlus.dll, появившейся на свет с выходом Windows XP.

Излагаемый в книге материал разделен на три части. В первой части подробно рассматриваются функции прикладного интерфейса программирования Windows API, посвященные разработке графического интерфейса приложений. По ходу изучения глав первой части мы научимся:

- создавать приложения без помощи VCL – нашей опорой станут функции операционной системы Windows;
- работать с графическим контекстом устройства;
- создавать перья, кисти, регионы, траектории;
- рисовать линии, кривые и простейшие геометрические фигуры;
- управлять режимами отображения;
- осуществлять страничные и мировые преобразования;
- проводить сложные растровые операции;
- представлять изображения в растровых (BMP, DIB) и векторных форматах (WMF и EMF);
- выводить простейшие и форматированные текстовые надписи.

Во второй части книги рассматриваются особенности программирования графики средствами визуальной библиотеки компонентов Delphi. Здесь мы обсудим:

- особенности инкапсуляции функций GDI в компонентах Delphi;
- графические классы и элементы управления среды;
- диалоговые окна, обеспечивающие настройку принтера;
- организацию работы с печатающими устройствами.

В третьей части книги изучается современная библиотека GDI+. Материал посвящен:

- изучению новейших возможностей объектов GDI+;
- работе с графическими форматами JPEG, TIFF, GIF, EMF+;
- обработке метаданных современной цифровой фотографии EXIF;
- методам улучшения качества изображений и цветовой коррекции.

Соглашения

Для акцентирования внимания читателя на ключевых частях излагаемого материала такой текст отмечен особым форматированием и пиктограммами. Речь идет о трех следующих замечаниях:

«На заметку»
Таким способом в тексте книги выделен материал, который вы должны принять к сведению. Обычно это советы, комментарии или замечания.

«Внимание!»
Текст, отмеченный восклицательным знаком, определяет однозначные правила действий программиста в той или иной ситуации.

«Стоп!»
Однозначный запрет, заостряет внимание на характерных ошибках. Короче говоря – никогда так не делайте.

Описываемые в книге функции могут быть отмечены пиктограммами «Windows API» или «Windows GDI».

Первая из пиктограмм говорит о том, что обсуждаемая функция относится к функциям прикладного интерфейса программирования Microsoft Windows и отражена в справочной системе Windows SDK (Platform Software Development Kit). Вторая пиктограмма конкретизирует, что указанная функция относится к подклассу функций графического интерфейса Windows API (соответствует ветви «Graphics and Multimedia» в SDK). Синтаксические конструкции всех функций API «переведены» с языка Си на язык Delphi. Если функция или процедура не снабжена пиктограммой, то это означает, что она входит в состав функций Borland Delphi.

Кроме того:

- Впервые встречающиеся термины и определения выделены курсивом.
- Впервые встречающиеся аббревиатуры комментируются.
- Все таблицы и рисунки в книге пронумерованы по принципу: номер главы, точка-разделитель, порядковый номер в главе.
- Код примеров, синтаксические конструкции даны моноширинным шрифтом, зарезервированные слова выделены полужирным шрифтом.
- Помимо содержания книга включает подробный предметный указатель, позволяющий найти в тексте интересующую читателя функцию, класс или структуру по ее названию.

В тексте можно обнаружить ссылки на примеры, исходные тексты программ, и графические файлы, доступные для скачивания на сайте издательства по адресу: *www.symbol.ru/library/Delphi_graphic*.

I

Graphics Device Interface (GDI)

1

Программирование на Windows API

Раз наша книга посвящена исследованию неотъемлемой части прикладного интерфейса программирования – функций GDI и GDI+, то наиболее подходящим практическим способом их изучения станет разработка программ на базе функций Windows API. Такой подход позволяет заглянуть в само сердце графической подсистемы и исключает необходимость отвлекаться на частности. Поэтому подавляющее число примеров написано без использования визуальной библиотеки компонентов (VCL) – на «голом» API. Исключение составили лишь листинги для глав, посвященных графике VCL, и примеры повышенной сложности. Именно поэтому первая глава книги посвящена не сколько графике, сколько основам программирования на Windows API.

Библиотека визуальных компонентов Delphi значительно упрощает процесс создания программы (скрывая от нас механизм взаимодействия приложения с операционной системой), но за это мы расплачиваемся относительно неэффективным результирующим кодом. Для устранения этого недостатка следует работать на более низком уровне – уровне Windows API. Программирование без опоры на библиотеку визуальных компонентов обязывает программиста в совершенстве знать операционную систему Microsoft Windows, особенности взаимодействия программы и ОС, возможности, предоставляемые функциями Windows API.

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 ведется 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, 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.

Для того чтобы приложение не пропустило предназначенное ему 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.

После того как мы выяснили самую страшную тайну Microsoft о том, что идея жизнедеятельности Windows позаимствована у обычного почтового отделения, а программы – большие специалисты в эпистолярном жанре (просто-напросто посылают друг-другу письма), обозначим этапы, через которые надо пройти программисту для того, чтобы создать простейшее жизнеспособное приложение для Windows. Для этого необходимо:

- Описать макет будущего главного окна приложения. Для этого заполняются поля специализированной структуры, называемой классом окна.
- Зарегистрировать этот класс в Windows.
- Создать экземпляр главного окна приложения.
- Отобразить окно на экране компьютера.
- Организовать цикл обработки сообщений Windows.
- Описать оконную процедуру, в рамках которой наша программа станет обслуживать все поступающие в ее адрес послания.

Дескрипторы, или особенность доступа к объектам Windows

На протяжении всех глав книги, посвященных GDI и GDI+, для обращения к тому или иному объекту мы будем пользоваться *дескрипторами* объектов. Для начинающего программиста, работающего исключительно с компонентами среды программирования Delphi, это не вполне понятный термин. В Delphi каждый объект гордо несет уникальное имя, что значительно упрощает обращение к его свойствам и методам. Допустим, нам необходимо изменить заголовок кнопки. В стандартном проекте Delphi для этого достаточно указать имя компонента, через точку – соответствующее свойство и присвоить ему новое значение:

```
Button1.Caption:='Я - кнопка!';
```

При работе на Windows API все несколько усложняется. Операционная система Windows тщательно охраняет свои ключевые объекты, она не допускает получение 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*). Кроме того, по существующим соглашениям для ключевых дескрипторов закреплены постоянные имена, так, например, при чтении документации SDK можно быть твердо 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 соотнесете 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 создается 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: TFNWndProc;  //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 счету 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 табл. 1.1.

Таблица 1.1. Стили окна, поле style структуры TWndClass

Стиль	Описание
CS_BYTEALIGNCLIENT	Автоматическое выравнивание клиентской области окна по горизонтали по границе байта. Этот флаг позволяет несколько повысить производительность вывода изображения.
CS_BYTEALIGNWINDOW	То же самое, что и CS_BYTEALIGNCLIENT, только теперь речь идет обо всем окне.
CS_CLASSDC	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.
CS_DBLCLKS	Позволяет окну реагировать на двойной щелчок.
CS_DROPSHADOW	При перетаскивании окна включается эффект тени. Стиль появился в Windows XP.
CS_GLOBALCLASS	Создаваемый класс окна станет доступен всем приложениям. Этот флаг часто применяется при хранении окон в динамических библиотеках.
CS_HREDRAW	При изменении горизонтального размера окна автоматически поступает команда на перерисовку этого окна.
CS_NOCLOSE	Отключается системная кнопка закрытия окна в его заголовке.
CS_OWNDC	Для каждого из окон одного класса создается уникальный контекст графического устройства.
CS_PARENTDC	Каждое дочернее окно, созданное на базе этого класса, будет использовать контекст устройства родительского окна.
CS_SAVEBITS	Позволяет быстро восстанавливать часть окна, скрытую под другими окнами. Для этого затененная область сохраняется в виде растрового образа и копируется в окно в момент его перерисовки.
CS_VREDRAW	При изменении вертикального размера окна автоматически поступает команда на перерисовку этого окна.

Особо подчеркну то, что если программист рассчитывает использовать комбинацию стилей, то между стилями 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. Ее *второе* поле (lpfnWndProc) хранит адрес оконной процедуры, в которой описывается реакция приложения на поступающие извне сообщения. *Третье* и *четвертое* 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*, *седь-*

мое и *восьмое* поля соответственно хранят дескрипторы иконки, курсора и кисти (заливающей клиентскую часть) окна. *Девятое* поле lpszMenuName задает 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, ATOM). 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 черед вводить в бой тяжелую 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;
```

Первый параметр `ClassName` представляет собой имя только что зарегистрированного класса окна. В Windows предусмотрен целых 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`). Второй параметр 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 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 создаете дочерние окна (не забывайте, что 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 создается окно. Это глобальная переменная, за ее объявление отвечает среда программирования. Последний необязательный параметр `lpParam` – указатель на дополнительные данные окна. При удачном выполнении функция возвратит уникальный дескриптор 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 по горизонтали
      cw_UseDefault,         // позиция верхнего левого угла окна по вертикали
      cw_UseDefault,         // ширина окна
      cw_UseDefault,         // высота окна
      0,                     // родительское окно
      0,                     // меню
      HInstance,             // указывает на экземпляр программы
      nil);
```

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

Отображение окна

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

Win API

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

С первым параметром все просто, это дескриптор только что созданного окна (результат работы функции `CreateWindow`). Второй параметр устанавливает режим отображения окна. Например, для того чтобы вывести окно, нужно использовать константу `SW_SHOW`. Наиболее часто применяемые константы перечислены в табл. 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:

Win API

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

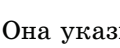
Единственный аргумент функции – дескриптор обновляемого окна. Функция информирует Windows о необходимости отправки окну сообщения WM_PAINT, заставляющего перерисовать клиентскую область окна.

Зачастую при формировании графического интерфейса приложения возникает необходимость в обновлении не всего окна, а только его части (например, области, над которой находилось окно другого приложения). Для этих целей предназначена функция:

```
function InvalidateRect(Wnd: HWND; Rct: PRect; Erase: Boolean): Boolean;
```

Здесь Wnd – дескриптор обслуживаемого окна; Rct – координаты прямоугольника, нуждающегося в обновлении; Erase – в состоянии true указывает GDI на то, что перед повторной прорисовкой области Rct она должна быть полностью стерта. Обращение к функции из кода программы вынуждает систему отправить окну Wnd сообщение WM_PAINT. Если вместо дескриптора окна функция получит неопределенный указатель, то обновятся все имеющиеся в данный момент окна (будет осуществлена широковещательная рассылка сообщений WM_ERASEBKGND и WM_NCPAINT, инициирующих перерисовку рамки, заголовка и фона окна).

Обратная задача поставлена перед функцией:



```
function ValidateRect(Wnd: HWND; Rct: PRect): Boolean;
```

Она указывает на то, что при очередной перерисовке окна Wnd прямоугольник Rct не должен быть затронут, вследствие чего этот прямоугольник удаляется из обновляемой области окна.

Функции InvalidateRect() *и* ValidateRect() *пользуются услугами функций обновления регионов –* InvalidateRgn() *и* ValidateRgn().

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 в очереди соответствующих процессов. Задача программиста – научить свое приложение обрабатывать сообщения, отправляемые ему операционной системой. Задача приложения – извлекать сообщения из своей очереди, осуществлять их обработку и при этом (желательно) не «зависнуть». Общепринятое название блока команд, извлекающего сообщения из очереди, – *цикл 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 сообщения для всех созданных программой окон, но в этом случае вместо дескриптора окна необходимо передавать нулевое значение. Третий и четвертый 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 мыши 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 табл. 1.3 приведены некоторые типы оконных сообщений Windows, они пригодятся в процессе описания оконной процедуры.

Таблица 1.3. Ключевые оконные сообщения Windows

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

Сообщение WM_PAINT *– ключевой, но далеко не единственный представитель набора системных сообщений, связанных с перерисовкой. Список могут дополнить сообщения:* WM_NCPAINT, WM_NCCLASSSIZE *и* WM_SIZE.

В визуальной библиотеке компонентов Delphi обработка сообщения WM_PAINT *закамуфлирована в рамках события* OnPaint(). *В частности, указанное событие имеется у ключевого элемента проектов Delphi – формы (экземпляра класса* TForm*).*

Сообщение WM_PAINT

По сути, 90 процентов материала этой книги посвящено организации обработки сообщения WM_PAINT, связанного 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 регионе имеется участок с изменившимся с момента последней перерисовки содержимым, то тогда система поместит в очередь даже не одно, а целых три сообщения: WM_NCPAINT, WM_ERASEBKGROUND и WM_PAINT. Первое из сообщений укажет окну, что требуется перерисовать рамку и заголовок окна. Сообщение WM_ERASEBKGROUND даст команду о необходимости восстановления фона окна, для этих целей будет задействована кисть из поля hbrBackground структуры TWnd, используемой при регистрации класса окна. Наконец, WM_PAINT выполнит основную работу – обновит ту часть клиентской области окна, которая нуждается в восстановлении.

Как правило, без особой на то необходимости программисты явным образом не описывают обработку сообщений WM_NCPAINT *и* WM_ERASEBKGROUND. *В этом случае сообщения обслуживаются процедурой по умолчанию.*

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 еще 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 путем вызова функции BeginPaint().
2. Собственно процесс перерисовки.
3. Обращение к функции EndPaint() для освобождения дескриптора контекста.

Программисту Delphi не мешает помнить, что в библиотеке VCL сообщение WM_PAINT *обслуживается на уровне класса* TWinControl. *Немного покопавшись в секции частных объявлений класса, вы обнаружите заголовок соответствующей процедуры:*

```
procedure WMPaint(var Message: TWMPaint); message WM_PAINT; _
```

По ходу книги для нас наибольший интерес будет представлять сообщение `WM_PAINT`. *Цель сообщения – уведомление окна о том, что оно или его часть нуждается в перерисовке. Приложение также в состоянии стать инициатором команды перерисовки, но для этого ему надо обратиться к функциям* `UpdateWindow()` *или* `RedrawWindow()`. *В следующей главе, посвященной контексту графического устройства, мы вновь вернемся к вопросу обслуживания сообщения* `WM_PAINT`.

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 вынуждает программиста запереться на пару лет в келье, чтобы 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 определенного сообщения поведение окна должно чем-то отличаться от стандартного, то будьте добры – садитесь за клавиатуру. Мы об этом еще поговорим, в частности, в подразделе, посвященном 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 `SendMessge()` 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;
```

Win API

```
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, несмотря 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 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, 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 возвращает 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 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` устанавливаем ссылки на файлы `Windows` и `Messages`. Сохраняем проект под именем ex01_01.dpr. Если вы внимательно следовали инструкциям, то в редакторе кода останутся лишь эти строки:

```
program ex01_01;
uses Windows, Messages;

begin

end.
```

Еще 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 ни 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);
    lpszMenuName := '';
    lpszClassName := 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 (рис. 1.1). Предвижу возмущение читателя: «Столько работы для того, чтобы создать элементарный проект!» Предлагаю взглянуть на это с другой стороны. Например, размер полученного исполняемого файла минимален – не более 16 Кбайт! 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 Кбайт, вот такая

Рис. 1.1. Окно простейшей программы

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 рис. 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.

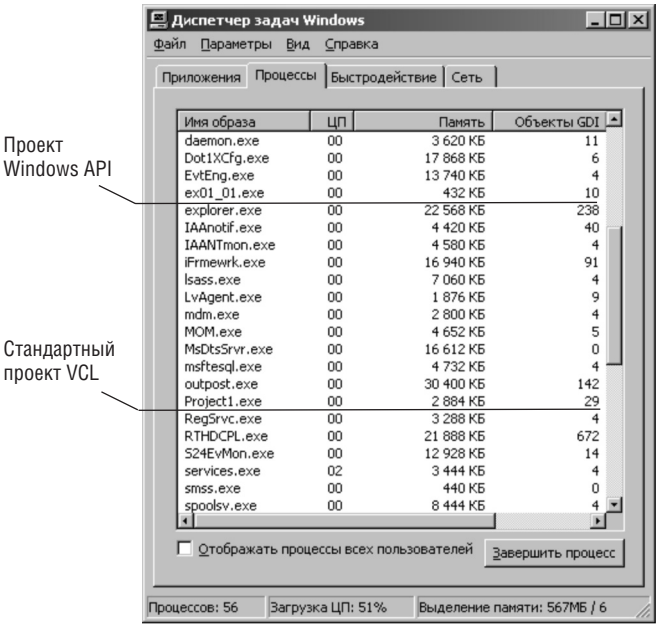


Рис. 1.2. Экранный снимок диспетчера задач

Дочерние окна

Для уяснения еще 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',
                  'Закрыть',
```

```
                    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 еще 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('Закрыть окно?'),
            PChar('Подтвердите'),
            MB_OKCANCEL+MB_ICONINFORMATION)=idOK
        then SendMessage(Wnd,WM_DESTROY,0,0); {сами себе отправляем WM_DESTROY}
       END;
END;
```

Субклассирование

Читатель, внимательно изучивший предыдущий материал, уяснил порядок обслуживания сообщений в Windows. Когда операционная система получает сообщение для какого-то из окон, она выясняет адрес оконной процедуры окна-получателя и вызывает ее, передавая окну-получателю все содержание сообщения. В теории все просто. Но это поверхностное впечатление, как всегда на практике возникают проблемы. Допустим, что наше приложение содержит цепочку окон, в которой 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 (рис. 1.3).

Для воплощения нашей идеи на практике целесообразно воспользоваться сообщением WM_CREATE, вызываемым сразу после создания главного окна приложения. В рамках обработки сообщения создаем панели группировки (описываемые дескрипторами grBox1 и grBox2) и шесть переключателей (rBtn1... rBtn6), причем первые три кнопки принадлежат первой, а вторые три – второй

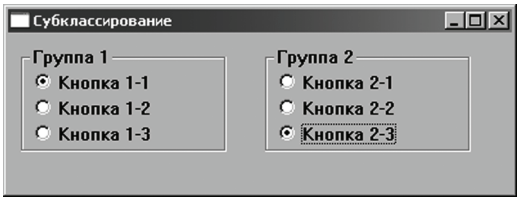


Рис. 1.3. Внешний вид окна с двумя группами переключателей

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 столкнемся с определенной 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 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. Проблема в том, что в проекте нет оконных процедур для панелей группировок кнопок-переключателей...

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

Хакерский прием 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, в 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 еще рядом силовых приемов, например она позволяет изменить стиль окна (табл. 1.4). Третий параметр NewLong определяет подменяемое значение. В случае успешного выполнения функция возвращает предыдущее 32-разрядное значение (если речь идет 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 которой 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 ее 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 жестких требований. Перечень ее параметров обязан полностью соответствовать списку параметров обычной оконной процедуры. Также нельзя путать последовательность аргументов. На первом месте всегда идет 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: TFNWndProc; 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 передается в первый параметр функции, в нашем примере это не что иное, как возвращаемое функцией `SetWindowLong()` значение.

РЕЗЮМЕ

Прикладной программный интерфейс Windows API позволяет создавать эффективные программные продукты, по ряду параметров существенно превосходящие приложения, построенные в классическом для Delphi стиле.

Именно поэтому большая часть примеров в этой книге написана на API. Такой подход позволит нам не только глубже понять особенности графического механизма Windows, но и получить необходимые для настоящего профессионала навыки системного программирования.

Вместе с тем опора на Windows API ни в коем случае не отвергает применение в наших проектах визуальной библиотеки компонентов Delphi. Ведь корни VCL также опираются на функции API, а компоненты инкапсулирует основной функционал Windows. Но это не обычное подражание. Инженеры Borland значительно упростили и, одновременно, усовершенствовали процесс программирования. Хорошим тому подтверждением стал графический класс `TCanvas`, не только впитавший в себя ключевые функции Windows GDI, но и приобретший преимущества классического класса VCL. Именно поэтому программист Delphi должен уметь разумно сочетать в своих программах преимущества Windows API и VCL.

2

Контекст графического устройства

Операционная система Windows своим успехом во многом обязана великолепной инженерной идее, реализованной в виде *контекста графического устройства* (*Device Context*). Контекст представляет собой центр визуального интерфейса системы – в его руках хранятся все нити управления выводом информации на экран или на печатающее устройство.

В чем его заслуги перед Windows? Вряд ли получится ответить на этот вопрос в двух словах. Начнем с того, что контекст позволяет отделить аппаратную часть компьютера (устройство графического вывода) от программной. Именно поэтому все функционирующие под управлением операционной системы компоненты чудесным образом превращаются в аппаратно-независимые. Забота о преобразовании команд GDI на вывод графических примитивов в команды, понятные аппаратной части компьютера, возложена на плечи операционной системы и драйвера устройства, но ни в коем случае не на программиста. В итоге существенно облегчается труд разработчика как прикладного, так и системного программного обеспечения, ведь теперь при написании кода практически нет нужды задумываться о технических особенностях построения того или иного устройства графического вывода. Различия между устройствами сглаживаются до такой степени, что с точки зрения программирования процесс вывода изображения в окне приложения практически не отличается от печати рисунка на принтере – графическое устройство превращается в некую весьма удобную для нас абстракцию. Вторая примечательная особенность контекста устройства в том, что он содержит данные о текущих настройках графического устройства, на котором мы планируем что-то отобразить. Например, в перечень атрибутов контекста входят описание логической кисти и пера, шрифта, режим отображения, палитра и многое другое. Поэтому в простейшем случае при вызове функций GDI достаточно указать координаты, определяющие место вывода графических примитивов, а их цвет, толщину пера, стиль, используемый шрифт, высоту символов и многое другое функции GDI выяснят у контекста.

С точки зрения Windows контекст графического устройства представляет собой иерархию взаимосвязанных структур и объектов, находящихся в адресном пространстве пользовательского режима приложения и в адресном пространстве ядра системы. С одной стороны контекст взаимодействует с драйвером графического устройства, с другой – с нашими приложениями (рис. 2.1).

Разработчики Windows не раскрывают нам тайну реализации контекста и оставляют за собой право изменять его в новых операционных системах, но вне зависимости от версии ОС в роли основной нити, связывающей наши программы и устройства графического вывода, выступает *дескриптор контекста устройства*:

```
HDC = type LongWord;
```

Технически это 32-разрядный идентификатор, позволяющий программисту отправлять команды GDI в адрес конкретного устройства вывода, будь это окно программы, графический элемент управления, принтер, плоттер, факс или виртуальное устройство.

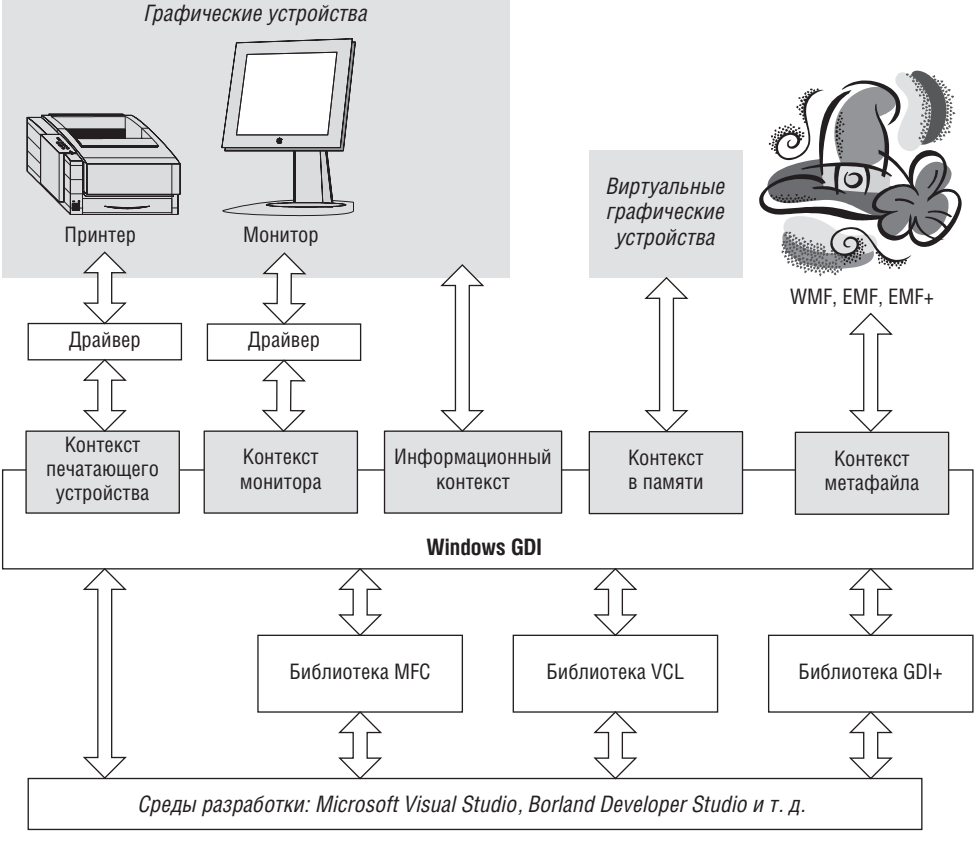


Рис. 2.1. Место контекста устройства в GDI

Дескриптор контекста графического устройства является обязательным параметром для подавляющего большинства функций GDI.

Документация SDK выделяет четыре типа контекстов: контекст дисплея, контекст печатающего устройства, контекст в памяти и информационный контекст. Если проявить принципиальность, то стоит отметить факт существования пятого типа контекста – метафайлового (см. рис. 2.1), но о нем мы поговорим далее в главе 18, посвященной расширенным метафайлам Windows.

Контекст графического устройства, как конгломерат сложных структур и объектов, содержит некоторый перечень полей, определяющих наиболее общие характеристики контекста. Список весьма значителен, но в нем можно выделить атрибуты, хранящие следующие данные:

- регионы контекста, определяющие порядок отсечения;
- установленная система координат и текущий режим отображения;
- параметры двумерных аффинных преобразований в мировых координатах;
- параметры заливки областей;
- параметры черчения линий;
- цветовые характеристики графического устройства;
- выбранный шрифт и особенности вывода текста.
- сложение цветов или, говоря научным языком, установленная бинарная растровая операция.
- параметры траектории.

Список можно продолжить, но пока переведем дыхание, тем более, что со всеми ключевыми атрибутами контекста мы повстречаемся в следующих главах книги. А теперь настал черед знакомства со способами получения дескриптора контекста устройства.

Основная задача контекста устройства заключается в предоставлении доступа к графической поверхности, состоящей из двумерного массива пикселов. Как правило, каждый из элементов массива доступен как для операций чтения, так и для записи.

Дескриптор контекста для экрана и печатающих устройств

Итак, с точки зрения программиста контекст графического устройства представляет собой мост между любым аппаратным графическим устройством и нацеленным на рисование исходным кодом. В роли графического устройства способен выступить не только экран монитора, но принтеры, плоттеры и другие аппаратные устройства, связанные с выводом или вводом графической информации. В системе Windows они такие же полноправные обладатели контекста. Для доступа к дескриптору контекста такого рода требуется познакомиться с функцией `CreateDC()`:

Win GDI

```
Function CreateDC(Driver, Device, Output : PAnsiChar;
                  Init : pDeviceModeA ):HDC;
```

Здесь Driver – имя драйвера устройства, в качестве аргумента могут выступать только три значения: строка «DISPLAY» (для работы с экраном), строка «WINSPOOL» (для работы с печатающим устройством) и неопределенный указатель nil. Второй параметр Device – это системное имя устройства. В современных 32- и 64-разрядных версиях Windows параметр Output не применяется. Параметр Init представляет собой указатель на структуру PDeviceModeA, благодаря которой мы сможем инициализировать контекст. Все четыре аргумента одновременно практически никогда не используются. Например, для обращения к контексту дисплея достаточно только первого параметра.

```
Procedure SCREEN_PAINT;
begin
   DC:=CreateDC(PChar('DISPLAY'),nil, nil, nil);
     SetBkMode(DC,TRANSPARENT);
     TextOut(DC, 10, 10, PChar('Контекст экрана'), 15);
   DeleteDC(DC);
end;
```

Благодаря CreateDC() мы получаем право рисовать на поверхности рабочего стола Windows, что может пригодиться приложениям, выполняющим свои задачи в фоновом режиме или в свернутом виде, например для оповещения пользователя о получении почты или окончании записи компакт-диска.

Для доступа к принтеру необходим второй параметр функции:

```
DC:= CreateDC(nil, PChar('Canon CP660PS'),nil, nil);
```

Функция CreateDC() превосходно подходит для обращения к контекстам печатающих устройств или экрана компьютера, но она не позволит программисту работать с контекстом окна приложения. Для решения этой задачи предназначены другие функции.

По умолчанию при создании любой контекст устанавливается в текстовый режим отображения (MM_TEXT). Позднее, в главе 11 «Системы координат и режимы отображения», мы обсудим все применяемые в Windows режимы отображения. А пока достаточно знать, что в текстовом режиме:

– начало координат располагается в левом верхнем углу;

– горизонтальная ось X направлена слева направо, а вертикальная ось Y – сверху вниз;

– в качестве единицы измерения выступает пиксел.

Контекст окна приложения

Если программа научена откликаться на сообщение WM_PAINT (знакомой нам по предыдущей главе команды на перерисовку окна), то наиболее рациональным способом получения дескриптора контекста устройства станет обращение к функции BeginPaint(). Эту функцию мы уже активно использовали в примерах первой главы, но незаслуженно мало времени уделили ее описанию.